Verified Decision Procedures on Context-Free Grammars

Yasuhiko Minamide

Department of Computer Science
University of Tsukuba
minamide@cs.tsukuba.ac.jp

Abstract. We verify three decision procedures on context-free grammars utilized in a program analyzer for a server-side programming language. One of the procedures decides inclusion between a context-free language and a regular language. The other two decide decision problems related to the well-formedness and validity of XML documents. From its formalization, we generate executable code for a balancedness checking procedure and incorporate it into an existing program analyzer.

1 Introduction

We have been developing a program analyzer for the server-side scripting language PHP, which approximates the string output of a program with a context-free grammar [8]. We adopted and developed several advanced decision procedures on context-free grammars to check properties of a program with the analyzer [9, 17]. Although the correctness of those decision procedures is often intuitively clear, their detailed proofs can be rather complicated. That motivated us to verify them by using a proof assistant.

In this paper, we verify three procedures on context-free grammars used in our analyzer in Isabelle/HOL [13]. They decide the following three problems:

- inclusion between a context-free language and a regular language;
- whether a context-free language is balanced or not;
- inclusion between a context-free language and a regular hedge language.

The second and third procedures are used to check the well-formedness and validity, respectively, of dynamically generated XML documents in the analyzer.

Of the three decision procedures, we generated executable code of the second, the balancedness checking procedure, with Isabelle's code generator. Although the formalization of the procedure is almost executable, the formalization must be revised to obtain executable code and achieve reasonable efficiency. We incorporated the generated code into the PHP string analyzer by replacing the corresponding handwritten code. The analyzer with the generated code has been applied successfully to real PHP programs.

Our formalization and verification were conducted using the development version of Isabelle. We made positive use of new features introduced in the development version, such as a new function definition package by Krauss [7] and the revised locale mechanism, which make it possible to write proof scripts in a more natural manner. The proof scripts of the formalization in this paper are available from `http://www.score.cs.tsukuba.ac.jp/~minamide/cfgv/`.

2 Context-Free Grammars

In this section, we formalize context-free grammars (CFGs) and several basic procedures on them. The decision procedures in this paper are based on interpretation of a CFG over a monoid. The interpretation can be naturally formalized by considering a CFG over the monoid. Thus, we extend the notion of CFGs and formalize CFGs over a monoid.

2.1 Formalization of CFGs

To simplify our formalization, we only consider a grammar in a normal form such that each production has one of the forms: $x \to a$ and $x \to yz$ where x, y, and z are nonterminals (variables) and a is a terminal. A CFG is represented with the following record type:

record $('v,\ 'a)\ \mathit{cfg} =$
  $\mathit{prod1} :: ('v \times 'a)\ \mathit{set}$
  $\mathit{prod2} :: ('v \times 'v \times 'v)\ \mathit{set}$
  $\mathit{start} :: 'v$

where $'v$ and $'a$ are the types of nonterminals and terminals, respectively. This declaration introduces a record type with three fields *prod1*, *prod2*, and *start*. The fields *prod1* and *prod2* contain productions of the form $x \to a$ and $x \to yz$, respectively. A component of a record can be accessed with the field name, *e.g.*; *prod1 r* accesses the *prod1* field of a record *r*. A CFG over strings is modeled with type $('v,\ 'a\ \mathit{list})\ \mathit{cfg}$. Note that this type permits a production of the form $x \to w$ for any sting w. Thus, the normal form above is not as strict as Chomsky normal form.

Although a grammar itself can be given without considering a monoid operation, the operation is necessary to define the language of the grammar. The set of monoid elements generated from a nonterminal is defined as an inductively defined relation: $(x,v) \in \mathit{derive\ cfg\ opr}$ formalizes "v is derived from nonterminal x".

$\mathit{derive} :: [('v,\ 'a)\ \mathit{cfg},\ ['a,\ 'a] \Rightarrow 'a] \Rightarrow ('v \times 'a)\ \mathit{set}$
inductive *derive cfg opr*
 $(x,v) \in \mathit{prod1\ cfg} \Longrightarrow (x,v) \in \mathit{derive\ cfg\ opr}$
 $[\![(y,v) \in \mathit{derive\ cfg\ opr};\ (z,w) \in \mathit{derive\ cfg\ opr};\ (x,y,z) \in \mathit{prod2\ cfg}\ ]\!] \Longrightarrow$
                                        $(x, \mathit{opr}\ v\ w) \in \mathit{derive\ cfg\ opr}$

For a CFG over strings, concatenation of lists, the infix operator @ in Isabelle, is used for *opr*. Although this relation is defined even if *opr* does not satisfy the laws of monoids, we only use *opr* satisfying the laws in this paper. The language of a CFG is the set of monoid elements derived from the start symbol.

$lang\text{-}of :: [('v,\ 'a)\ cfg,\ ['a,\ 'a] \Rightarrow\ 'a]\ \Rightarrow\ 'a\ set$
$lang\text{-}of\ cfg\ opr \equiv\ \{\ v.\ (start\ cfg,\ v) \in derive\ cfg\ opr\}$

2.2 Computing the Language of a CFG over a Finite Monoid

The language of a CFG can also be characterized as a fixed point of a monotone function over $'v \Rightarrow\ 'a\ set$ below.

$onestep :: [('v,\ 'a)\ cfg,\ ['a,'a] \Rightarrow\ 'a,\ 'v \Rightarrow\ 'a\ set] \Rightarrow\ 'v \Rightarrow\ 'a\ set$
$onestep\ cfg\ opr\ m\ x \equiv m\ x \cup \{v.\ (x,v) \in\ prod1\ cfg\} \cup$
$\qquad \{opr\ v\ w \mid v\ w.\ \exists\, y\ z.\ (x,y,z) \in prod2\ cfg \wedge v \in m\ y \wedge w \in m\ z\}$

The fixed point of this function can be computed if we consider a CFG over a *finite* monoid. The procedure can be formalized as follows using the predefined *while*-combinator in Isabelle.

$compute\text{-}langs :: [('v,\ 'a)\ cfg,\ ['a,\ 'a] \Rightarrow\ 'a,\ 'v] \Rightarrow\ 'a\ set$
$compute\text{-}langs\ cfg\ opr \equiv$
$\qquad while\ (\lambda x.\ onestep\ cfg\ opr\ x \neq x)\ (onestep\ cfg\ opr)\ (\lambda x.\ \{\})$

The following equality is proved by the rule of Hoare logic for *while* formalized in Isabelle.

theorem fixes $cfg{::}('v{::}finite,\ 'a{::}finite)\ cfg$
 shows $compute\text{-}langs\ cfg\ opr\ x = \{v.\ (x,v) \in derive\ cfg\ opr\}$

To guarantee that the while loop terminates, both the sets of nonterminals and terminals must be finite. The constraints are given as the type class *finite* of $'v$ and $'a$.

2.3 Image of a Context-Free Language

An interpretation of a CFG with a monoid is considered as an image of its language under a homomorphism to the monoid.

Let us consider a homomorphism h between monoids $'a$ and $'b$. A standard result of the theory of CFGs is that the image of a context-free language (CFL) over $'a$ under the homomorphism h is a CFL over $'b$. This is shown by constructing a grammar over $'b$ as follows:

$image\text{-}of :: [('v,\ 'a)\ cfg,\ 'a \Rightarrow\ 'b] \Rightarrow ('v,\ 'b)\ cfg$
$image\text{-}of\ cfg\ f \equiv$
$\qquad (\!|\ prod1 = prod1\ cfg \odot rel\text{-}of\ f,\ prod2 = prod2\ cfg,\ start = start\ cfg\ |\!)$

where $s \odot t$ is the composition[1] of two relations and *rel-of* converts a function into a relation.

$$s \odot t \equiv \{(x,z).\ \exists\, y.\ (x,y) \in s \wedge (y,z) \in t\} \qquad rel\text{-}of\ f \equiv \{(x,y).\ f\ x = y\}$$

[1] The order of the arguments is different from the standard composition operator in Isabelle/HOL.

Then, the following formalizes the result above.

theorem assumes $\forall\, v\ w.\ h\ (opr\ v\ w) = opr'\ (h\ v)\ (h\ w)$
shows $\mathit{lang\text{-}of}\ \ (\mathit{image\text{-}of}\ cfg\ h)\ opr' =\ \ h\ `\ \mathit{lang\text{-}of}\ cfg\ opr$

where $h\ `\ xs$ is the image of the set xs under the function h.

2.4 Reachable and Generating Nonterminals

We formalize two basic procedures on CFGs: computing the set of reachable nonterminals and computing the set of generating nonterminals. The latter procedure is then extended to generate a witness for each generating nonterminal.

We say a nonterminal X is reachable if there exists a derivation $S \overset{*}{\Rightarrow} \alpha X \beta$ where α and β are strings over terminals and nonterminals, and S is the start symbol. The standard procedure to compute the set of reachable nonterminals applies depth-first search by considering production rules as a graph.

A CFG is converted into a graph with *next-rel* and, we formalize reachability based on the graph as follows.

$\mathit{next\text{-}rel}\ cfg \equiv \{(x,y).\ \exists\, z.\ (x,y,z) \in \mathit{prod2}\ cfg\} \cup \{(x,z).\ \exists\, y.\ (x,y,z) \in \mathit{prod2}\ cfg\}$
$\mathit{reachable}\ cfg\ xs \equiv (\mathit{next\text{-}rel}\ cfg)^*\ ``\ xs$

where $r\ ``\ xs$ is the image of the set xs under the relation r.

We formalize a depth-first search to compute the set of reachable nonterminals as follows.

function
$\mathit{dfs} :: [('v{::}\mathit{finite},'a)\ cfg,\ 'v\ list,\ 'v\ set] \Rightarrow 'v\ set$ **where**
$\mathit{dfs}\ cfg\ []\ ys = ys$
$\mathit{dfs}\ cfg\ (x\#xs)\ ys =$
$\quad (\mathit{if}\ x \in ys\ \mathit{then}\ \mathit{dfs}\ cfg\ xs\ ys\ \mathit{else}\ \mathit{dfs}\ cfg\ (\mathit{nexts}\ cfg\ x@xs)\ (\mathit{insert}\ x\ ys))$

The function *nexts cfg x* computes a list of nodes adjacent to x. This formalization is almost identical to a depth-first search we write in a functional programming language. It is shown that this function always terminates by a lexicographic order similar to that used by Moore in his formalization of a graph search algorithm [10]. The correctness of this procedure is verified as the following theorem.

theorem $\mathit{dfs}\ cfg\ (\mathit{nexts}\ cfg\ x)\ \{x\} = \mathit{reachable}\ cfg\ \{x\}$

We say a nonterminal X is generating if $X \overset{*}{\Rightarrow} w$ where w is a terminal string. The following is a formalization of a standard algorithm to compute generating nonterminals, which is considered as a fixed-point computation.

$\mathit{genv\text{-}onestep} :: [('v,\ 'a)\ cfg,\ 'v\ set] \Rightarrow 'v\ set$
$\mathit{genv\text{-}onestep}\ cfg\ m \equiv m \cup \{x.\ \exists\, y\ z.\ (x,y,z) \in \mathit{prod2}\ cfg \wedge y \in m\ \wedge z \in m\}$

$\mathit{genv} :: ('v,\ 'a)\ cfg \Rightarrow 'v\ set$
$\mathit{genv}\ cfg \equiv$
$\quad \mathit{while}\ (\lambda x.\ \mathit{genv\text{-}onestep}\ cfg\ x \neq x)\ (\mathit{genv\text{-}onestep}\ cfg)\ \{x.\ \exists\, v.\ (x,v) \in \mathit{prod1}\ cfg\}$

Each iteration adds the nonterminals that are shown to be generating by *genv-onestep* from the nonterminals obtained in the previous iteration. The correctness of the function is proved as follows.

lemma fixes $cfg::('v::finite,\ 'a)\ cfg$
    shows $genv\ cfg = \{x.\ \exists\, w.\ (x,w) \in derive\ cfg\ opr\}$

The function *genv* is extended to obtain a map of type $s \rightharpoonup t$, which gives a witness for each generating nonterminal, *i.e.*, a string generated from it. Type $s \rightharpoonup t$ is a synonym of $s \Rightarrow t\ option$. The extended procedure is used in the balancedness checking procedure in Section 4.

To extend *genv* naturally, we use the function *choose* defined below

$choose\ xs \equiv if\ xs \neq \{\}\ then\ Some\ (SOME\ x.\ x{\in}xs)\ else\ None$

The expression $SOME\ x.\ P\ x$ represents an arbitrary element x satisfying $P\ x$. If there is no such element, it is an arbitrary element.

We thus obtain the following function that gives a witness for each generating nonterminal.

$genv\text{-}onestep' :: [('v,\ 'a)\ cfg,\ ['a,'a] \Rightarrow 'a,\ 'v \rightharpoonup 'a] \Rightarrow 'v \rightharpoonup 'a$
$genv\text{-}onestep'\ cfg\ opr\ m \equiv$
$(\lambda x.\ if\ x \in dom\ m\ then\ m\ x$
$else\ choose\ \{opr\ w1\ w2 \mid w1\ w2.\ \exists\, y\ z.\ (x,y,z) \in prod2\ cfg \wedge m\ y = Some\ w1 \wedge m\ z = Some\ w2\})$

$genv' :: [('v,\ 'a)\ cfg,\ ['a,'a] \Rightarrow 'a] \Rightarrow 'v \rightharpoonup 'a$
$genv'\ cfg\ opr \equiv while\ (\lambda x.\ genv\text{-}onestep'\ cfg\ opr\ x \neq x)\ (genv\text{-}onestep'\ cfg\ opr)$
$(\lambda x.\ choose\ \{v.\ (x,v) \in prod1\ cfg\})$

The structures of *genv* and *genv'* are almost identical, and thus we can easily show the following lemma.

lemma fixes $cfg::('v::finite,\ 'a)\ cfg$ **shows** $dom\ (genv'\ cfg\ opr) = genv\ cfg$

3 Decision Procedure for Inclusion between a CFL and a Regular Language

We verify a decision procedure checking inclusion $L(G) \subseteq L(M)$ for a CFG G and a nondeterministic finite automaton M. Textbooks on formal languages usually describe a decision procedure for this problem based on product construction of a pushdown automaton and a finite automaton. On the other hand, the procedure in this section directly operates on a grammar. It is a variant of the context-free graph reachability algorithm of Reps [15]. We use Nipkow's formalization of automata in Isabelle [12].

3.1 Nipkow's Formalization of Automata

We review Nipkow's formalization of nondeterministic automata. Nipkow formalized an automaton as a triple of a start state, a transition function, and a predicate defining final states. The following is its reformulation with a record type:

record $('a,'s)$ *na* =
  start :: $'s$
  next :: $['a, 's] \Rightarrow 's$ *set*
  fin :: $'s \Rightarrow$ *bool*

where $'a$ and $'s$ are the types of an alphabet and states, respectively. The formalization itself allows the set of states to be infinite.

Then, the extended transition function *delta* over lists and the function *accepts* describing accepted strings are defined in the standard manner:

delta A [] $p = \{p\}$
delta A $(a\#w)$ p = *Union*(*delta A w* ‘ *next A a p*)

accepts A w $\equiv \exists\, q \in$ *delta A w* (*start A*). *fin A q*

where the function *Union* gives the union of a set of sets and has type $'a$ *set set* $\Rightarrow$ $'a$ *set*.

3.2 Transition Monoid

The following theorem states that a regular language is characterized by a finite monoid [4].

Theorem 1. *Let Σ be a finite alphabet. The following are equivalent for $L \subseteq \Sigma^*$.*

1. *L is regular.*
2. *There exist a finite monoid $\mathcal{M}$, a homomorphism $h : \Sigma^* \to \mathcal{M}$, a subset $B \subseteq \mathcal{M}$ such that $L = h^{-1}(B)$.*

One of the easiest ways to construct the monoid, the homomorphism, and the subset in the theorem is to consider a monoid of relations over states, where the composition of relations plays the role of the monoid operation. This monoid of relations over states and the homomorphism associated with it implicitly appear in Nipkow's formalization. The function *steps* in the formalization translates a string into a relation and has the following property.

steps A w = $\{(p,q).\ q \in$ *delta A w p*$\}$

This function plays the role of the homomorphism in the theorem.

In addition to Nipkow's formalization, we introduce the following function, where *final-of na* plays the role of B in the theorem for the nondeterministic automaton *na*.

$\mathit{final\text{-}of} :: ('a,'s)\ \mathit{na} \Rightarrow ('s \times 's)\ \mathit{set\ set}$
$\mathit{final\text{-}of}\ \mathit{na} \equiv \{r.\ \exists\, q.\ (\mathit{start}\ \mathit{na},\ q) \in r \wedge\ \mathit{fin}\ \mathit{na}\ q\}$

Then, the following is a formalization of the theorem in Isabelle, which can be easily proved with the lemmas provided in Nipkow's formalization.

theorem $\mathit{steps}\ \mathit{na}\ w \in \mathit{final\text{-}of}\ \mathit{na} \longleftrightarrow \mathit{accepts}\ \mathit{na}\ w$

3.3 Decision Procedure

With the formalization of CFGs and automata, it is quite easy to formalize a decision procedure to check inclusion between the languages of a CFG and a nondeterministic finite automaton. The inclusion can be checked by interpreting a grammar with the monoid characterizing the regular language. We obtain the expression to compute the interpretation over $('s \times 's)\ \mathit{set}$ as follows.

$$\begin{aligned}
&(\mathit{steps}\ \mathit{na})\ \text{‘}\ \mathit{lang\text{-}of}\ \mathit{cfg}\ (\mathit{op}\ @)\\
&= \mathit{lang\text{-}of}\ (\mathit{image\text{-}of}\ \mathit{cfg}\ (\mathit{steps}\ \mathit{na}))\ (\mathit{op}\ \odot)\\
&= \mathit{compute\text{-}langs}\ (\mathit{image\text{-}of}\ \mathit{cfg}\ (\mathit{steps}\ \mathit{na}))\ (\mathit{op}\ \odot)\ (\mathit{start}\ \mathit{cfg})
\end{aligned}$$

This can then be used as the decision procedure as follows.

theorem **fixes** $\mathit{cfg} :: ('v::\mathit{finite},\ 'a\ \mathit{list})\ \mathit{cfg}$ **and** $\mathit{na} :: ('a,'s::\mathit{finite})\ \mathit{na}$
  shows $\mathit{lang\text{-}of}\ \mathit{cfg}\ (\mathit{op}\ @) \subseteq \{w.\ \mathit{accepts}\ \mathit{na}\ w\} \longleftrightarrow$
  $\mathit{compute\text{-}langs}\ (\mathit{image\text{-}of}\ \mathit{cfg}\ (\mathit{steps}\ \mathit{na}))\ (\mathit{op}\ \odot)\ (\mathit{start}\ \mathit{cfg}) \subseteq \mathit{final\text{-}of}\ \mathit{na}$

The type system of Isabelle recognizes that the type $('s \times 's)\ \mathit{set}$ is an instance of *finite* from $s::\mathit{finite}$. Hence, we can compute the language and decide inclusion between finite sets of type $('s \times 's)\ \mathit{set}$.

4 Balancedness Checking Procedure

We introduce a CFG over a paired alphabet and verify a balancedness checking procedure that decides whether the language of a grammar is balanced or not. The procedure was developed by Berstel and Boasson [2] and the formalization in this paper is based on [9].

4.1 Balanced Strings

For a base alphabet A, we consider a paired alphabet consisting of two sets $\acute{A}$ and $\grave{A}$:

$$\acute{A} = \{\, \acute{a} \mid a \in A \,\} \qquad \grave{A} = \{\, \grave{a} \mid a \in A \,\}$$

The elements of $\acute{A}$ and $\grave{A}$ are considered as left and right parentheses: $\acute{a}$ and $\grave{a}$ match. The fundamental notion on a string over a paired alphabet is whether it is balanced. For example, $\acute{a}\acute{b}\grave{b}\acute{c}\grave{c}\grave{a}$ and $\acute{a}\grave{a}\acute{b}\grave{b}$ are balanced, but $\acute{a}\grave{b}$ and $\acute{a}\acute{b}\grave{b}$ are not. We call the set of all balanced strings the Dyck set [1]. A language L is balanced if all $\phi \in L$ are balanced.

We formalize strings over a paired alphabet as lists over the following data type.

datatype $'a\ balphabet\ = L\ 'a \mid R\ 'a$

Then, the set of balanced strings, the Dyck set, is formalized as the following inductively defined set.

```
dyckset :: 'a balphabet list set
inductive dyckset
  [] ∈ dyckset
  xs ∈ dyckset ⟹ [L x]@xs@[R x] ∈ dyckset
  ⟦xs ∈ dyckset; ys ∈ dyckset ⟧ ⟹  xs@ys ∈ dyckset
```

4.2 Monoid for Balancedness Checking

The Dyck set is not regular and thus cannot be characterized with a finite monoid. However, there is an infinite monoid, that makes it possible to decide whether the language of a CFG is balanced or not.

We say a string ϕ is *partially balanced* if it is a substring of some balanced string. Each partially balanced ϕ can be uniquely factorized into the following form:

$$\phi = \phi_1 \grave{a}_1 \phi_2 \grave{a}_2 \ldots \phi_n \grave{a}_n \varphi \acute{b}_m \psi_m \cdots \acute{b}_2 \psi_2 \acute{b}_1 \psi_1$$

where ϕ_i, ψ_i, and φ are all balanced. We say $\grave{a}_1 \grave{a}_2 \ldots \grave{a}_n \acute{b}_m \cdots \acute{b}_2 \acute{b}_1$ the reduced form of the string and write $\rho(\phi)$ for it. The set of reduced forms $\grave{A}^* \acute{A}^*$ with $\bot$ constitute a monoid, where $\bot$ represents the reduced form of unbalanced strings. The balancedness checking procedure is developed based on this monoid.

We formalize the monoid over $\grave{A}^* \acute{A}^* \cup \{\bot\}$ with the following data type:

datatype $'a\ bmonoid =\ \ B\ 'a\ list\ 'a\ list \mid Bot$

where $B\ [a_1, a_2, \ldots, a_n]\ [b_1, b_2, \ldots, b_m]$ represents $\grave{a}_1 \grave{a}_2 \ldots \grave{a}_n \acute{b}_m \cdots \acute{b}_2 \acute{b}_1$, and B [] [] is the unit of the monoid.

Let ϕ and ψ be partially balanced strings. The monoid operation between their reduced forms is defined by considering that of their concatenation $\phi\psi$. It is formalized as follows.

```
function
  concat :: ['a bmonoid, 'a bmonoid] ⇒ 'a bmonoid (infixr ◇ 65)  where
  B cs1 [] ◇ B cs2 ss2 = B (cs1@cs2) ss2
  B cs1 ss1 ◇ B [] ss2 = B cs1 (ss2@ss1)
  B cs1 (s#ss1) ◇ B (c#cs2) ss2 =  (if s = c then B cs1 ss1 ◇ B cs2 ss2 else Bot)
  Bot ◇ y = Bot
  x ◇ Bot = Bot
```

Intuitively, it is clear that this operation is associative. On the other hand, the proof of this is not straightforward. We derive an equivalent definition of the function using the prefix relation of lists and prove associativity based on the definition.

With the monoid laws of *'a bmonoid*, it is straightforward to show that h defined below is a homomorphism from lists over a paired alphabet to the monoid.

$\mathit{hom\text{-}of} :: [{}'a \Rightarrow {}'b\ \mathit{bmonoid},\ {}'a\ \mathit{list}] \Rightarrow {}'b\ \mathit{bmonoid}$
$\mathit{hom\text{-}of}\ h\ [] = B\ []\ []$
$\mathit{hom\text{-}of}\ h\ (x\#\mathit{xs}) = h\ x \diamondsuit \mathit{hom\text{-}of}\ h\ \mathit{xs}$

$h \equiv \mathit{hom\text{-}of}\ (\lambda x.\ \mathit{case}\ x\ \mathit{of}\ L\ x \Rightarrow B\ []\ [x] \mid R\ x \Rightarrow B\ [x]\ [])$

Finally, we show that the Dyck set is characterized by this monoid.

theorem $\mathit{xs} \in \mathit{dyckset} \longleftrightarrow h\ \mathit{xs} = B\ []\ []$

The implication from left to right is easily proved by induction on the derivation of $\mathit{xs} \in \mathit{dyckset}$. The other direction is rather difficult to prove. We introduce a notion similar to the partially balanced string as an inductively defined set and prove: $\mathit{xs} \notin \mathit{dyckset} \longrightarrow h\ \mathit{xs} \neq B\ []\ []$.

For balancedness checking, we also introduce an ordering over reduced forms, which was used to improve the time complexity of a balancedness checking procedure in [9]. The ordering is defined as follows.

$$\grave{a}_1 \cdots \grave{a}_n \acute{c}_m \cdots \acute{c}_1 \leq \grave{a}_1 \cdots \grave{a}_n \grave{b}_1 \cdots \grave{b}_j \acute{b}_j \cdots \acute{b}_1 \acute{c}_m \cdots \acute{c}_1$$

This is formalized as the following order over *bmonoid*.

$b1 \leq b2 \equiv (\exists\, cs\ ss\ zs.\ b1 = B\ cs\ ss \wedge b2 = B\ (cs@zs)\ (ss@zs))$
$\qquad\qquad \vee (b1 = \mathit{Bot} \wedge b2 = \mathit{Bot})$

4.3 Decision Procedure

Because the monoid introduced in the previous subsection is infinite, it does not directly give rise to a balancedness checking procedure. However, it is shown that balancedness can be checked by generating the monoid elements derived from each nonterminal to some bound.

In this section, we assume that a CFG is reduced. This means that every nonterminal is accessible from the start symbol and every nonterminal produces at least one terminal string. This condition is expressed with the following conditions on *cfg* in Isabelle.

$\forall x.\ x \in \mathit{reachable}\ \mathit{cfg}\ \{\mathit{start}\ \mathit{cfg}\}$ $\qquad$ $\forall x.\ \exists w.\ (x,\ w) \in \mathit{derive}\ \mathit{cfg}\ \mathit{opr}$

Because we assume that a grammar is reduced, for each nonterminal X, we can find terminal strings ϕ_1 and ϕ_2 such that $S \overset{*}{\Rightarrow} \phi_1 X \phi_2$. We then have $S \overset{*}{\Rightarrow} \phi_1 \psi \phi_2$ for any ψ such that $X \overset{*}{\Rightarrow} \psi$. For $\phi_1 \psi \phi_2$ to be balanced, the reduced forms of ϕ_1 and ϕ_2 must be the following forms: $\rho(\phi_1) = \acute{a}_n \ldots \acute{a}_1$ and $\rho(\phi_2) = \grave{b}_1 \cdots \grave{b}_m$. Furthermore, we have $\rho(\psi) \leq \grave{a}_n \ldots \grave{a}_1 \acute{b}_1 \cdots \acute{b}_m$. This observation is formalized as the following lemmas for *bmonoid*.

lemma assumes $b1 \diamondsuit b2 \diamondsuit b3 = B\ []\ []$
shows $(\exists\, ss.\ b1 = B\ []\ ss) \wedge (\exists\, cs.\ b3 = B\ cs\ [])$

lemma assumes $B\ []\ ss \diamondsuit b \diamondsuit B\ cs\ [] = B\ []\ []$ **shows** $b \leq B\ ss\ cs$

These properties enable us to use $\grave{a}_n \dots \grave{a}_1 \acute{b}_1 \cdots \acute{b}_m$ above as a bound when we check the balancedness of the language of a grammar.

We have the following decision procedure based on the observation so far by considering the interpretation of a CFG over the monoid.

1. Compute ϕ_1 and ϕ_2 such that $S \overset{*}{\Rightarrow} \phi_1 X \phi_2$ for each nonterminal X.
2. Generate the monoid elements derived from each nonterminal to the bound determined by ϕ_1 and ϕ_2.
3. If the bound is exceeded for some nonterminal, then the language is not balanced. Otherwise, the language is balanced iff the start symbol only generates the unit element, *i.e.*, B [] [] in Isabelle.

The first step of the procedure is based on the procedures to compute reachable and generating nonterminals. First, we must compute a witness for each generating nonterminal by the procedure in Section 2.4. Then, we apply a depth-first search procedure to compute ϕ_1 and ϕ_2 above. Then, from ϕ_1 and ϕ_2, the bound is constructed by the following function.

mkbound (*B* [] *ss*, *B cs* []) = *B ss cs*

The depth-first search procedure computes paths from the start symbol to all the reachable nonterminals, where a path corresponds to a pair ϕ_1 and ϕ_2 above. Please refer to the proof script for details.

The second step of the procedure is easily formalized as follows, where the bound is given as a function *bounds*.

```
compute-langs' :: [('v, 'a) cfg, ['a, 'a] ⇒ 'a, 'v ⇒ 'a set, 'v] ⇒ 'a set
compute-langs' cfg opr bounds ≡
   while (λm. onestep cfg opr m ≠ m ∧ (∀x. m x ⊆ bounds x))
         (onestep cfg opr) (λx. {})
```

If *bounds x* is finite for all *x*, the function terminates.

The following is the formalization of the whole decision procedure, where *mkcon* is the depth-first search procedure and finds a bound, a pair of ϕ_1 and ϕ_2 above, for each nonterminal by using a function *gf* giving a generated monoid element for each nonterminal.

```
bcheck cfg ≡
(let gf = λx. the (genv' cfg (op ◇) x);
     con = λx. the (mkcon (op ◇) (B [] []) cfg gf (start cfg) x) in
 (∀x. ∃ss cs. con x = (B [] ss, B cs [])) ∧
 (let bounds = λx. {b. b ≤ mkbound (con x)};
      result = compute-langs' cfg (op ◇) bounds in
  ∀x. result x ≤ bounds x ∧ result (start cfg) = {B [] []}))
```

The correctness of this procedure is verified in the following sense.

```
theorem fixes cfg::('v::finite, 'a balphabet list) cfg
  assumes ∀x. ∃w. (x, w) ∈ derive cfg (op @)
          ∀x. x ∈ reachable cfg {start cfg}
  shows bcheck (image-of cfg h) ⟷ lang-of cfg (op @) ⊆ dyckset
```

5 Decision Procedure for Inclusion between a Context-Free Language and a Regular Hedge Language

We formalize a decision procedure deciding inclusion between a context-free language and a regular hedge language [11]. The procedure can be considered as a combination of the previous two decision procedures, and was developed by Minamide and Tozawa [9].

5.1 Hedges and Balanced Strings

We call a sequence of trees over the unranked alphabet Σ a hedge. The sets of trees and hedges denoted by t and h are defined as follows:

$$\begin{aligned} t &::= a\langle h\rangle \\ h &::= \epsilon \mid t\, h \end{aligned}$$

where $a \in \Sigma$. We write $\mathcal{H}(\Sigma)$ for the set of hedges over Σ. By expanding t in the definition of hedges, hedges can also be defined as follows.

$$h ::= \epsilon \mid a\langle h\rangle h$$

We formalize this definition as the following datatype in Isabelle.

datatype *'a hedge = Empty | Br 'a 'a hedge 'a hedge*

Hedges can be considered as balanced strings by the following function.

```
hedge2word :: 'a hedge ⇒ 'a balphabet list
hedge2word Empty = []
hedge2word (Br a xs ys) = [L a]@hedge2word xs@[R a]@hedge2word ys
```

It is shown that this function is injective and its range is the set of balanced strings. The key to proving these properties is the following property of balanced strings. It is proved by using the monoid in Section 4.

```
lemma assumes [L a]@xs1@[R a]@xs2 = [L a]@ys1@[R a]@ys2
              xs1 ∈ dyckset ys1 ∈ dyckset
  shows xs1=ys1 ∧ xs2 = ys2
```

5.2 Regular Hedge Grammars and Binoids

A regular hedge grammar (RHG) is a grammar over hedges with production rules of the following forms.

$$X \to \epsilon \qquad\qquad X \to a\langle Y\rangle Z$$

The set of hedges generated by a RHG is called a regular hedge language (RHL). It is basically a regular tree language over an unranked alphabet.

RHGs are formalized with the following record type as CFGs, and the derivation and the language of a RHG are formalized in the same manner as those of CFGs.

```
record ('v, 'a) rhg =
  prod1 :: 'v set
  prod2 :: ('v × 'a × 'v × 'v) set
  start :: 'v
```

Pair and Quere showed that a RHL is characterized with an algebra called binoid [14]. A binoid $\mathcal{B}$ over Σ is a monoid with the following additional operation.

$$\hat{\_}(\_) : \Sigma \times \mathcal{B} \to \mathcal{B}$$

For example, the set of hedges itself constitutes a binoid, where the monoid operation is the concatenation of two hedges and the additional operation above is one that builds $a\langle h\rangle$ from a and h.

The following shows that a RHL can be characterized with a *finite* binoid.

Theorem 2. *The following are equivalent for a set of hedges $L \subseteq \mathcal{H}(\Sigma)$.*

1. *L is regular.*
2. *There exist a finite binoid $\mathcal{B}$, a homomorphism $h : \mathcal{H}(\Sigma) \to \mathcal{B}$, a subset $B \subseteq \mathcal{B}$ such that $L = h^{-1}(B)$.*

The binoid satisfying the theorem above can be obtained by considering a monoid of relations over nonterminals. The additional operation $\hat{\_}(\_)$ is defined as *up rhg a b* for a and b below.

```
up :: [('v, 'a) rhg, 'a, ('v × 'v) set] ⇒ ('v × 'v) set
up rhg a b ≡ {(x,y) |x y. ∃(z, f) ∈  b. f ∈ prod1 rhg ∧ (x,a,z,y) ∈ prod2 rhg}
```

Although this binoid is used in the decision procedure for CFL-RHL inclusion, any binoid that satisfies the theorem is also suitable for the procedure. Thus, we introduce the locale *binoid* below and describe the main part of the procedure in the locale. Finally, we obtain our decision procedure by instantiating it to the binoid above.

```
locale binoid =
  fixes up :: ['a, 'b] ⇒ 'b
  fixes prod :: ['b, 'b] ⇒ 'b (infixr ◊ 70)
  fixes unit :: 'b
  assumes assoc: (x ◊ y) ◊ z = x ◊ y ◊ z
  assumes unitl: unit ◊ x = x
  assumes unitr: x ◊ unit = x
```

5.3 Monoid for CFL-RHL Inclusion

Let us assume that a RHL is characterized with a binoid $\mathcal{B}$, a homomorphism $\_^{\circ}$, and a set B. The idea of the decision procedure is to interpret a set of strings generated from each nonterminal by using elements of the $\mathcal{B}$. However, each string ϕ such that $X \overset{*}{\Rightarrow} \phi$ is not necessarily balanced, but rather partially balanced. Therefore, we again use the factorization of ϕ:

$$\phi = \phi_1 \grave{a}_1 \phi_2 \grave{a}_2 \ldots \phi_n \grave{a}_n \varphi \acute{b}_m \psi_m \cdots \acute{b}_2 \psi_2 \acute{b}_1 \psi_1$$

where ϕ_i, ψ_i, and φ are all balanced. Then, we interpret ϕ with an element of $(\mathcal{B}\Sigma)^*\mathcal{B}(\Sigma\mathcal{B})^*$ as follows:

$$\phi_1^\circ \grave{a}_1 \phi_2^\circ \grave{a}_2 \ldots \phi_n^\circ \grave{a}_n \varphi^\circ \acute{b}_m \psi_m^\circ \cdots \acute{b}_2 \psi_2^\circ \acute{b}_1 \psi_1^\circ$$

where $\_^\circ$ is applied to balanced strings by considering them hedges. This is the ideas of the decision procedure.

The elements of the set $(\mathcal{B}\Sigma)^*\mathcal{B}(\Sigma\mathcal{B})^* \cup \{\bot\}$ constitute a monoid, and it is represented by the following datatype:

```
datatype ('b,'a) bmonoid = B ('b × 'a) list 'b ('b × 'a) list | Bot
```

where $'b$ and $'a$ are the types for $\mathcal{B}$ and Σ, respectively.

Then, by using the locale *binoid* we formalize the monoid operation with the following function, where *concat x y z* is written as $x \triangleleft y \triangleright z$ with Isabelle's mixfix annotation.

```
function (in binoid)
  concat :: [('b, 'a) bmonoid, 'b, ('b, 'a)  bmonoid] ⇒ ('b, 'a) bmonoid  where
  B cs1 b1 [] ◁x▷ B [] b2 ss2 = B cs1 (b1♢x♢b2) ss2
  B cs1 b1 [] ◁x▷ B ((cb2,c2)#cs2) b2 ss2 = B (cs1@(b1♢x♢cb2,c2)#cs2) b2 ss2
  B cs1 b1 ((sb1,s1)#ss1) ◁x▷ B [] b2 ss2 = B cs1 b1 (ss2@(sb1♢x♢b2,s1)#ss1)
  B cs1 b1 ((sb1,s)#ss1) ◁x ▷ B ((cb2,c)#cs2) b2 ss2 =
   (if s = c then B cs1 b1 ss1 ◁ up s (sb1♢x♢cb2) ▷ B cs2 b2 ss2  else Bot)
  Bot ◁x▷ b = Bot
  b ◁x▷ Bot = Bot
```

The monoid operation is then $\lambda xy.x \triangleleft unit \triangleright y$, where *unit* is the unit of the binoid.

The rest of the decision procedure is quite similar to the balancedness checking procedure and is formalized in the same manner.

6 Executing a Verified Decision Procedure

We generated executable code of the balancedness checking procedure from our formalization with Isabelle's code-generating facility [6]. The generated procedure was incorporated into the PHP string analyzer [8] by replacing the corresponding handwritten procedure. The analyzer checks whether a PHP program always generates a well-formed XHTML document with the procedure. The revised analyzer was tested on real PHP programs. We describe issues that arose during this experiment.

The formalization of the balancedness checking procedure we have described is almost executable, but still requires some revisions to obtain executable code. We have formalized decision procedures as abstractly as possible by using *set* and $s \rightharpoonup t$ instead of concrete data structures. They are barriers to generating code and making it efficient.

Isabelle has the library *ExecutableSet*, which allows us to generate code for finite sets using lists. We generated executable code for sets with this library, but some revisions of the formalization and an extension of the library were required. We explain the issues with the procedure for computing the set of generating nonterminals with a witness.

$genv\text{-}onestep'\ cfg\ opr\ m \equiv$
$(\lambda x.\ if\ x \in dom\ m\ then\ m\ x$
$else\ choose\ \{opr\ w1\ w2 \mid w1\ w2.\ \exists\, y\ z.\ (x,y,z) \in prod2\ cfg \wedge m\ y = Some\ w1 \wedge m\ z = Some\ w2\})$

$genv'\ cfg\ opr \equiv while\ (\lambda x.\ genv\text{-}onestep'\ cfg\ opr\ x \neq x)\ (genv\text{-}onestep'\ cfg\ opr)$
$(\lambda x.\ choose\ \{v.\ (x,v) \in prod1\ cfg\})$

The first issue is the set comprehension in the formalization. The definition includes the following set comprehension: $\{v.\ (x,v) \in prod1\ cfg\}$. Although this set is finite if *prod1* is finite, this is not explicit in the expression and executable code cannot be directly generated from it. To obtain executable code, we must revise the formalization by using the following property.

$$\{v.\ (x,\ v) \in prod1\ cfg\} = \bigcup (x',\ v){\in}prod1\ cfg.\ if\ x = x'\ then\ \{v\}\ else\ \{\}$$

The second problem concerns the *choose* function in the definition. The definition of *choose* uses *SOME*, *i.e.*, Hilbert's ϵ, for which code generation is not supported by the library. Furthermore, it appears that the code of *choose* cannot be implemented faithfully with lists. If *choose* is implemented with *choosel* over a list, then *choosel xs* = *choosel ys* should hold for lists *xs* and *ys* representing the same set. This property cannot be satisfied without any additional structure on its elements. To overcome this issue, we revised our formalization so that *choose* is used only for sets over types that are instances of *linorder*. The definition of *choose* was also revised so that it chooses the minimum element in a set.

We could generate code with these revisions. However, to obtain code with reasonable efficiency, more revisions were required. In our experiments, we represented nonterminals with type `int` in the ML side. Because the set of elements of `int` is finite and linearly ordered, the type satisfies the required conditions to use it as the type of nonterminals. However, the following two issues arise.

We verified the correctness of our decision procedures under the assumption that all nonterminals are generating and reachable. This simplified our verification, but it is not reasonable to assume it when we represent nonterminals with type `int` in ML. Thus, we revised our formalization so that the procedures assume that only the nonterminals used in a CFG are generating and reachable.

Finally, the evaluation strategy of functional programming languages becomes an issue for efficient execution: they do not evaluate the expression inside a lambda abstraction. In the decision procedure, type $s \rightharpoonup t$ is used to represent finite maps. The type is actually a function type to *option* type. Then, all the computation related to a map is delayed until the map is applied to an argument. That caused an exponential blowup of execution time. To avoid this blowup, we insert the function *reduce-fun* with the following type into the places where evaluation inside a lambda abstraction is desirable.

$reduce\text{-}fun :: [{'a}{::}\{finite,linorder\}\ set,\ 'a \Rightarrow 'b] \Rightarrow 'a \Rightarrow 'b$

An expression *reduce-fun f s* forces the evaluation of f for the values in the set s and reconstructs the function. For the definition of the function, please refer to the proof script.

By applying these revisions, we could run the analyzer with the code generated by Isabelle for real PHP programs. We tested it on two PHP programs, for which the analyzer generated CFGs with 170 and 70 production rules. With the handwritten code, the balancedness can be checked very quickly, taking 0.016 and 0.003 seconds for the two programs. On the other hand, the generated code was quite slow, taking 32.6 and 16.7 seconds. We checked execution times for each part of the procedure and found that the depth-first search is efficient, but the fixed-point computations used in *genv'* and *compute-langs'* are very slow. We think that this is because the formalization of depth-first search is very close to a standard implementation, but the formalization of the fixed-point computation is rather abstract. It will be necessary to revise its formalization and adopt a more efficient data structure to represent finite maps to obtain more efficient code.

7 Related Work

Our formalization is strongly influenced by Nipkow's formalization of automata, which formalizes automata and regular expressions, and verifies a lexical analyzer obtained from a regular expression [12]. There have been several other attempts to formalize the theory of formal languages in a proof assistant. Courant and Filliâtre formalized regular and context-free languages in Coq [3]. The formalization includes some standard theory of context-free languages such as transformation between a context-free grammar and a pushdown automaton. Rival and Goubault-Larrecq formalized tree automata in Coq [16] and executed some procedures on tree automata in Coq.

We have described several issues in obtaining an (efficient) executable balancedness checking procedure in Section 6. The importance of obtaining efficient executable code from an elegant formalization is recognized in the context of ACL2. Greve et al. reviewed issues there and described the features in ACL2 to support it [5]. This will be good guide to obtain really efficient executable code for our decision procedures.

8 Conclusion

We have verified three precision procedures on context-free grammars. The formalization and verification of the procedures went smoothly and took me about non-intensive two months. On the other hand, the revisions to generate executable code for the balancedness checking procedure required more time than we expected.

We plan to generate code for the other two procedures. Although code will be generated in the same manner as the balancedness checking procedure, we expect a more severe problem with efficiency. It is because the relations over states or nonterminals used there will become too large if `int` are used in the ML side.

References

1. Jean Berstel. *Transductions and Context-Free Languages*. Teubner Studienbucher, 1979.
2. Jean Berstel and Luc Boasson. Formal properties of XML grammars and languages. *Acta Informatica*, 38(9):649–671, 2002.
3. Judicael Courant and Jean-Christophe Filliâtre. Beginning of formal language theory, 1993. http://coq.inria.fr/contribs-eng.html.
4. Samuel Eilenberg. *Automata, Languages, and Machines*, chapter 3. Academic Press, 1974.
5. David A. Greve, Matt Kaufmann, et al. Efficient execution in an automated reasoning environment. *Journal of Functional Programming*, 2007. to appear.
6. Florian Haftmann. Code generation from Isabelle/HOL theories, 2007. available in the Isabelle distribution.
7. Alexander Krauss. Partial recursive functions in higher-order logic. In *Automated Reasoning, Third International Joint Conference*, volume 4130 of *LNCS*, pages 589–603, 2006.
8. Yasuhiko Minamide. Static approximation of dynamically generated Web pages. In *Proceedings of the 14th International World Wide Web Conference*, pages 432–441. ACM Press, 2005.
9. Yasuhiko Minamide and Akihiko Tozawa. XML validation for context-free grammars. In *Proc. of The Fourth ASIAN Symposium on Programming Languages and Systems*, volume 4279 of *LNCS*, pages 357–373, 2006.
10. J Strother Moore. An exercise in graph theory. In Matt Kaufmann, Panagiotis Manolios, and J Strother Moore, editors, *Computer-Aided Reasoning: ACL2 Case Studies*, chapter 5, pages 41–74. Kluwer Academic Publishers, 2000.
11. Makoto Murata. Hedge automata: a formal model for XML schemata, 1999. `http://www.xml.gr.jp/relax/hedge_nice.html`.
12. Tobias Nipkow. Verified lexical analysis. In *Theorem Proving in Higher Order Logics*, volume 1479 of *LNCS*, pages 1–15, 1998.
13. Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.
14. C. Pair and A. Quere. Définition et étude des bilangages réguliers. *Information and Control*, 13(6):565–593, 1968.
15. Thomas Reps. Program analysis via graph reachability. *Information and Software Technology*, 40(11–12):701–726, 2000.
16. Xavier Rival and Jean Goubault-Larrecq. Experiments with finite tree automata in Coq. In *Proc. 14th Int. Conf. Theorem Proving in Higher Order Logics (TPHOL'01)*, volume 2152 of *LNCS*, pages 362–377, 2001.
17. Akihiko Tozawa and Yasuhiko Minamide. Complexity results on balanced context-free languages. In *Proc. of Tenth International Conference on Foundations of Software Science and Computation Structures*, volume 4423 of *LNCS*, pages 346–360, 2007.