Verification of Code Generators via Higher-Order Model Checking

Takashi Suwa Takeshi Tsukada
Naoki Kobayashi
The University of Tokyo

Atsushi Igarashi
Kyoto University

Abstract

Dynamic code generation is useful for optimizing code with respect to information available only at run-time. Writing a code generator is, however, difficult and error prone. We consider a simple language for writing code generators and propose an automated method for verifying code generators. Our method is based on higher-order model checking, and can check that a given code generator can generate only closed, well-typed programs. Compared with typed multi-stage programming languages, our approach is less conservative on the typability of generated programs (i.e. can accept valid code generators that would be rejected by typical multi-stage languages) and can check a wider range of properties of code generators. We have implemented the proposed method and confirmed its effectiveness through experiments.

1. Introduction

Dynamic code generation, where code is generated by running a program (called a code generator), is useful for specializing a program with respect to information available at runtime. Writing a code generator is often difficult and error prone, however. A number of multi-stage languages have been introduced [3, 5, 8, 10, 13, 20, 30–33, 36] to make it easier to write program generators. Those languages are usually equipped with type systems to ensure that only "valid" programs can be generated. A drawback of such typed multi-staged languages is that the type systems are sometimes too conservative on the typability of generated programs. For example, $\lambda^{\bigcirc}$ [9] expresses the type of a program for generating a value of type τ as $\bigcirc\tau$. But then the following program:

```
if mode = "int" then next(0) else next(0.0)
```

will be rejected as ill-typed, because the then-part has type $\bigcirc$`int` whereas the else-part has type $\bigcirc$`float`. (Here, suppose that `next` M generates the code for evaluating M at the next stage.) The type systems of multi-stage languages are also not flexible enough on the guaranteed properties; for example, while the type system for $\lambda^{\bigcirc}$ can guarantee that an expression of type $\bigcirc$`int` only generates a code fragment evaluated to an integer, it does not guarantee that the expression generates only a code fragment evaluated to a positive integer; to guarantee such a property, one has to redesign the type system of the language.

We propose an alternative approach to safe code generation, based on higher-order model checking [15, 21]. We provide programmers with a simple functional language equipped with primitives for generating symbols and constructing code fragments. The programs in our language must be simply-typed, but unlike $\lambda^{\bigcirc}$ and other typed multi-stage languages, the generated code is dynamically typed; we have only a single type `code` for code and symbols. Thus, programs may generate ill-typed programs, or even those that do not respect variable bindings (like $\lambda x.y$, which contains an unbound variable). To reject those invalid code generators, we transform a code generator to a tree-generating program, which produces a certain tree representation of the generated code. By appropriately instrumenting the trees, various notions of "validity" (such as closedness and well-typedness) of the generated code are reduced to regular properties on the corresponding trees, which can be checked by using higher-order model checking. Higher-order model checking [21] is a higher-order extension of finite-state model checking, which can decide whether the tree generated by a given simply-typed tree grammar satisfies a given regular tree property.

We explain our approach through an example. Let us consider the following OCaml-like program, where the function `main` takes an integer n as an argument and generates code for the function that takes x and returns x^n.

```
let rec genpower n x =
  if n=0 then One else Times(x, genpower (n-1) x)
let main n =
  let x = gensym() in Abs(x, genpower n x)
```

Here, `One`, `Times`, and `Abs` are code constructors of arities 0, 2, and 2 respectively; they represent the code for constant 1, multiplication, and a λ-abstraction respectively. The function `main` takes an integer n, generates a fresh symbol and then passes n and the symbol to the function `genpower`, which returns a code fragment e of the form $\texttt{Times}(x, \cdots \texttt{Times}(x, \texttt{One}) \cdots)$. The `main` function then returns the whole code $\texttt{Abs}(x, e)$ (which represents the function $\lambda x.x^n$).

We convert the above program to the following, non-deterministic program for generating trees.

```
let gensym() = ...
let rec genpower x =
  if * then One else Times(x, genpower x)
let main() =
  let x = gensym() in Abs(x, genpower x)
```

Here, we have removed integer arguments and replaced the condition $n = 0$ with a non-deterministic Boolean $*$. The code construc-

tors have been replaced by tree constructors of the same name. The definition of `gensym` depends on the property to be verified. If we wish to verify that the code produced by the original program is $\texttt{Abs}(x, M)$, where M is of the form

$$\texttt{Times}(x_1, \texttt{Times}(x_2, \cdots \texttt{Times}(x_n, \texttt{One}) \cdots)),$$

then we can simply define `gensym` as a constant function that always returns a singleton tree `Var`:

```
let gensym() = Var
```

Then, the transformed program generates

$$\begin{array}{l}\texttt{Abs}(\texttt{Var}, \texttt{One}),\\ \texttt{Abs}(\texttt{Var}, \texttt{Times}(\texttt{Var}, \texttt{One})),\\ \texttt{Abs}(\texttt{Var}, \texttt{Times}(\texttt{Var}, \texttt{Times}(\texttt{Var}, \texttt{One}))), \ldots\end{array}$$

in a non-deterministic manner. Therefore, it suffices to check that all the generated trees belong to the regular tree language described by the grammar:

$$\begin{array}{ll} S \to \texttt{Abs}(\texttt{Var}, A) & \\ A \to \texttt{One} & A \to \texttt{Times}(\texttt{Var}, A), \end{array}$$

by using higher-order model checking.

To verify that the above program generates only closed programs (where `Abs` is considered a variable binder), one can instead define `gensym` by:

```
let gensym() = if * then Var else Ig
```

It returns either `Var` or `Ig`. Intuitively, `Var` represents variables whose binding information should be tracked, whereas `Ig` represents those that should be ignored. Then, to check the closedness, it suffices to check that whenever `Var` occurs in the generated code, it is in the scope of $\texttt{Abs}(\texttt{Var}, \ldots)$. Indeed, the trees generated by the transformed program are:

$$\begin{array}{l}\texttt{Abs}(\texttt{Var}, \texttt{One}), \texttt{Abs}(\texttt{Ig}, \texttt{One}),\\ \texttt{Abs}(\texttt{Var}, \texttt{Times}(\texttt{Var}, \texttt{One})), \texttt{Abs}(\texttt{Ig}, \texttt{Times}(\texttt{Ig}, \texttt{One})),\\ \texttt{Abs}(\texttt{Var}, \texttt{Times}(\texttt{Var}, \texttt{Times}(\texttt{Var}, \texttt{One}))),\\ \texttt{Abs}(\texttt{Ig}, \texttt{Times}(\texttt{Ig}, \texttt{Times}(\texttt{Ig}, \texttt{One}))), \ldots,\end{array}$$

all of which satisfy the condition above. If the main function were wrongly defined by:

```
let main n =
  let x = gensym() in let y = gensym() in
    Abs(x, genpower n y),
```

then, the set of the trees generated by the transformed program would contain:

$$\texttt{Abs}(\texttt{Ig}, \texttt{Times}(\texttt{Var}, \texttt{One})),$$

which indicates that open code may be generated. In this manner, with an appropriate instantiation of the `gensym` function, we can verify various properties of the programs generated by code generators.

We formalize the above idea and show how the closedness and well-typedness of the generated code can be verified by using higher-order model checking. Since it may be too low-level to directly write a program generator in the gensym language, we also design a simple two-stage programming language, which is similar to $\lambda^{\bigcirc}$, but does not require expressions of the next stage to be simply-typed, and formalize the translation from $\lambda^{\bigcirc}$ to the gensym language, so that our verification method can be applied. We have implemented a prototype verification tool based on our method, and confirmed its effectiveness.

The rest of this paper is structured as follows: Section 2 introduces the *gensym language*, the source language for describing code generators. Section 3 formalizes our verification methods for code generators. Section 4 reports an implementation and experiments on the verification methods propopsed in Section 3. Section 5 introduces a relaxed version of $\lambda^{\bigcirc}$ and provides a translation from it into the gensym language, so that we can apply our verification method to programs written in the relaxed $\lambda^{\bigcirc}$. Section 6 discusses limitations of our method and related work. Section 7 concludes the paper.

Notations

We write $\mathbf{N}$ for the set of natural numbers. For a natural number $n \in \mathbf{N}$, we write $[n]$ for $\{k \in \mathbf{N} \mid 1 \le k \le n\}$. For a binary relation $R \subseteq A \times B$, we write $\mathrm{dom}(R)$ and $\mathrm{im}(R)$ for $\{a \in A \mid {}^{\exists} b \in B.\ (a,b) \in R\}$ and $\{b \in B \mid {}^{\exists} a \in A.\ (a,b) \in R\}$ respectively. For a (partial) map f from A to B, $\mathrm{dom}(f)$ and $\mathrm{im}(f)$ are defined as special cases. For a map f, we write $f[a \mapsto b]$ for the map f' such that $\mathrm{dom}(f') = \mathrm{dom}(f) \cup \{a\}$, $f'(a) = b$, and $f'(x) = f(x)$ for $x \in \mathrm{dom}(f') \setminus \{a\}$.

2. The Language for Code Generation

This section introduces a language for writing code generators that we call the *gensym language*. To clairfy the essence of our verification method, the language is kept minimal; it is a call-by-name functional programming language with code constructors and primitives for generating fresh symbols, having only code as base type values. All the functions are named and defined at the top level. The language should be considered a core language, to which actual programs written by programmers are transformed before applying our verification method. For dealing with call-by-value programs (like the examples in Section 1), it suffices to apply the CPS transformation. For dealing with programs using other values such as integers, we can use predicate abstraction [17], and then encode Booleans as functions, code and functions as primitive values.

We assume a finite set of code constructors, and write $\mathcal{P}$ for the map from the set of code constructors to the set of natural numbers, representing the arities of code constructors. We assume that $\mathcal{P}$ contains at least the following bindings:

$$\texttt{APP} \mapsto 2, \qquad \texttt{ABS} \mapsto 2, \qquad \texttt{FIX} \mapsto 2, \qquad \texttt{IFTE} \mapsto 3.$$

They are constructors for function applications, abstractions, recursions, and conditional expressions (if-then-else). In examples, we use other constructors such as `TIMES` (of arity 2) and `'n` (of arity 0) for each integer n. We use the meta-variable P for code costructors.

A *function definition* is a pair:

$$(F, \boldsymbol{\lambda} x_1.\ \cdots \boldsymbol{\lambda} x_k.\ e),$$

where F is a function symbol and e ranges over the set of applicative terms, defined by:

$$e ::= F \mid x \mid e\, e \mid P \mid \mathbf{gensym}/\mathbf{cont}.$$

Here, F and x are meta-variables for function symbols and variables respectively. The primitive $\mathbf{gensym}/\mathbf{cont}$ e is a fresh symbol generator in continuation passing style; it takes an expression e as an argument, creates a fresh symbol, and passes it to e. We often write $F\ x_1\ \cdots\ x_\ell = e$ for the function definition $(F, \boldsymbol{\lambda} x_1.\ \cdots \boldsymbol{\lambda} x_\ell.\ e)$. We also often abbreviate a finite sequence of variables $x_1, \ldots, x_\ell$ to $\tilde{x}$, and write $(F, \boldsymbol{\lambda}\tilde{x}.\ e)$ or $F\ \tilde{x} = e$ for the function definition.

A *(source) program* is a pair $(\mathcal{D}, S)$ where $\mathcal{D}$ is a finite set of function definitions, and S, called the *main function*, is one of the function symbols defined in $\mathcal{D}$. We allow $\mathcal{D}$ to contain more than one function definition for each function symbol. That is for the purpose of modelling non-determinism that would be introduced when predicate abstraction is used.

The operational semantics of the language is given as follows. We assume a countably infinite set $\mathbf{Symbol}$ of symbols that may be generated by $\mathbf{gensym/cont}$, ranged over by α. The set of *runtime terms*, the set of *code values*, and the set of *evaluation contexts*, ranged over respectively by $\hat{e}$, v, and $\mathcal{C}$, are defined by:

$$\begin{aligned}\hat{e} &::= \alpha \mid F \mid \hat{e}\,\hat{e} \mid P \mid \mathbf{gensym/cont}\,,\\ v &::= \alpha \mid P \mid v\,v\,,\\ \mathcal{C} &::= \texttt{[]} \mid P\,v\,\cdots\,v\,\mathcal{C}\,\hat{e}\,\cdots\,\hat{e}\,.\end{aligned}$$

We write $\mathcal{C}\big[\hat{e}\big]$ for the term obtained by replacing the sole occurrence `[]` in the evaluation context $\mathcal{C}$ with the term $\hat{e}$. The reduction relation $\hookrightarrow_{\mathcal{D}}$ of the gensym language is defined as the least relation that satisfies the following rules:

$$\mathcal{C}\big[F\,\hat{e}_1\,\cdots\,\hat{e}_\ell\,\big] \hookrightarrow_{\mathcal{D}} \mathcal{C}\big[[\hat{e}_\ell/x_\ell]\cdots[\hat{e}_1/x_1]e\big] \quad \text{if } (F\,x_1\,\cdots\,x_\ell = e) \in \mathcal{D}$$

$$\mathcal{C}\big[\mathbf{gensym/cont}\,\hat{e}\big] \hookrightarrow_{\mathcal{D}} \mathcal{C}\big[\hat{e}\,\alpha\big] \quad \text{if } \alpha \in \mathbf{Symbol} \text{ does not occur in } \mathcal{C}\big[\mathbf{gensym/cont}\,\hat{e}\big]$$

If $S \hookrightarrow_{\mathcal{D}}{}^{*}\ v$, we say v is a code value *generated by* $(\mathcal{D}, S)$. We write $\mathcal{C}(\mathcal{D}, S)$ for the set of code values that the program generates.

Example 1. The program $(\mathcal{D}, S)$, where

$$\begin{aligned}\mathcal{D} \coloneqq \{&\mathit{GenPower}\ x = \texttt{'1},\\ &\mathit{GenPower}\ x = \texttt{TIMES}\ x\ (\mathit{GenPower}\ x),\\ &S = \mathbf{gensym/cont}\ K,\\ &K\ x = \texttt{ABS}\ x\ (\mathit{GenPower}\ x)\}\end{aligned}$$

corresponds to the `genpower` example in Section 1. The following is a reduction process of $(\mathcal{D}, S)$:

$$\begin{aligned}S &\hookrightarrow_{\mathcal{D}} \mathbf{gensym/cont}\ K \hookrightarrow_{\mathcal{D}} K\ \alpha \hookrightarrow_{\mathcal{D}} \texttt{ABS}\ \alpha\ (\mathit{GenPower}\ \alpha)\\ &\hookrightarrow_{\mathcal{D}} \texttt{ABS}\ \alpha\ (\texttt{TIMES}\ \alpha\ (\mathit{GenPower}\ \alpha))\\ &\hookrightarrow_{\mathcal{D}} \texttt{ABS}\ \alpha\ (\texttt{TIMES}\ \alpha\ (\texttt{TIMES}\ \alpha\ (\mathit{GenPower}\ \alpha)))\\ &\hookrightarrow_{\mathcal{D}} \texttt{ABS}\ \alpha\ (\texttt{TIMES}\ \alpha\ (\texttt{TIMES}\ \alpha\ (\texttt{TIMES}\ \alpha\ (\mathit{GenPower}\ \alpha))))\\ &\hookrightarrow_{\mathcal{D}} \texttt{ABS}\ \alpha\ (\texttt{TIMES}\ \alpha\ (\texttt{TIMES}\ \alpha\ (\texttt{TIMES}\ \alpha\ \texttt{'1})))\end{aligned}$$

We require that programs are simply-typed. The set of types, ranged over by τ, is defined by:

$$\tau ::= \texttt{code} \mid \tau \to \tau$$

where the type `code` describes the type of generated code. Note that every code fragment has the same type `code` and we do not aim to infer (more detailed) types for generated code fragments by this simple-type system, unlike typed multi-stage calculi such as $\lambda^{\bigcirc}$ [9]. A program $(\mathcal{D}, S)$ is *well-typed* if there exists a type environment Γ of the form $\left\{F_i \mapsto \tau_i \,\middle|\, F_i \in \mathrm{dom}(\mathcal{D})\right\}$ that satisfies $\Gamma(S) = \texttt{code}$ and $\Gamma \vdash (\boldsymbol{\lambda}\tilde{x}.\ e) : \Gamma(F)$ for each $(F\ \tilde{x} = e) \in \mathcal{D}$, where the typing rules for (runtime) terms are given as follows:

$$\frac{\Gamma[x_1 \mapsto \tau_1]\cdots[x_\ell \mapsto \tau_\ell] \vdash e : \texttt{code}}{\Gamma \vdash \boldsymbol{\lambda}x_1.\ \cdots \boldsymbol{\lambda}x_\ell.\ e : \tau_1 \to \cdots \to \tau_\ell \to \texttt{code}}$$

$$\frac{(\Gamma(x) = \tau)}{\Gamma \vdash x : \tau} \qquad \frac{(\Gamma(F) = \tau)}{\Gamma \vdash F : \tau} \qquad \frac{}{\Gamma \vdash \alpha : \texttt{code}}$$

$$\frac{\Gamma \vdash \hat{e}_1 : \tau_2 \to \tau \qquad \Gamma \vdash \hat{e}_2 : \tau_2}{\Gamma \vdash \hat{e}_1\,\hat{e}_2 : \tau}$$

$$\frac{}{\Gamma \vdash \mathbf{gensym/cont} : (\texttt{code} \to \texttt{code}) \to \texttt{code}}$$

$$\frac{}{\Gamma \vdash P : \underbrace{\texttt{code} \to \cdots \to \texttt{code}}_{\mathcal{P}(P)} \to \texttt{code}}$$

The following is a standard property of the simple type system.

Theorem 2. *Let $(\mathcal{D}, S)$ be a well-typed program of the gensym language. If $S \hookrightarrow_{\mathcal{D}}{}^{*}\ \hat{e}$, then (i) $\hat{e}$ is a value and $\varnothing \vdash \hat{e} : \texttt{code}$; or (ii) there exists a run-time term $\hat{e}'$ such that $\hat{e} \hookrightarrow_{\mathcal{D}} \hat{e}'$.*

3. Verification Method

This section describes methods to verify the code generated by a program of the gensym language. As mentioned in the introduction, we reduce verification problems to the decision problem called *higher-order model checking* [15, 21]. We shall briefly review (a variant of) higher-order model checking in Section 3.1 and then describe methods to verify closedness (Section 3.2) and well-typedness (Section 3.3) of the generated code.

3.1 Preliminaries: higher-order model checking

Higher-order model checking [15, 21] is the problem to decide whether, given a higher-order tree grammar (called a *higher-order recursion scheme*) $\mathcal{G}$ and a regular tree property $\mathcal{A}$, the trees generated by $\mathcal{G}$ satisfy $\mathcal{A}$.

Remark 3. The definition of higher-order model checking given below is actually a variant of the original problem in [21], which properly subsumes ours. The differences are: (1) in the original setting, a higher-order recursion scheme is deterministic and generates a possibly infinite tree, whereas in our setting, it is nondeterministic and generates a set of finite trees, and (2) a property in the original setting is ω-regular, whereas in our setting it is regular (since we do not handle infinite trees in our setting).

Higher-order recursion schemes Intuitively, a higher-order recursion scheme is a call-by-name, simply-typed, and non-deterministic functional program for generating a tree.

A *ranked alphabet* Σ is a finite collection of symbols equipped with their arities. Formally it is a map $\Sigma : \mathrm{dom}(\Sigma) \to \mathbf{N}$ from a finite set of symbols (called *terminal symbols*) to natural numbers.

A (non-deterministic) *higher-order recursion scheme* (a *HORS* for short) on a ranked alphabet Σ is a triple $\mathcal{G} = (\Sigma, \mathcal{R}, S)$ consisting of

- the ranked alphabet Σ;
- a finite set $\mathcal{R}$ of *rewriting rules* of the form:

  $$F\ \tilde{x} \to t,$$

  where F is a symbol called a *nonterminal*, $\tilde{x}$ is a (possibly empty) finite sequence of variables and t is an *applicative term* given by the grammar $t ::= a \mid F \mid x \mid t\ t$ (where a, F, and x range over $\mathrm{dom}(\Sigma)$, the set of non-terminals, and the set $\{\tilde{x}\}$ of variables);
- a special non-terminal S called the *start symbol*.

We write $\mathrm{dom}(\mathcal{R})$ for the set of nonterminals occurring on the left-hand side of a rule. Each non-terminal may have more than one rewriting rule, as in the language in Section 2. We require that the HORS be simply-typed in the following sense. Let $\mathbf{Sort}$ be the set of *sorts* (i.e. simple types for HORSs) defined by $\kappa ::= \mathtt{o} \mid \kappa \to \kappa$, where $\mathtt{o}$ is the sort of trees. The HORS $\mathcal{G} = (\Sigma, \mathcal{R}, S)$ is *well-typed* if there exists a map $\mathcal{K} : \mathrm{dom}(\mathcal{R}) \to \mathbf{Sort}$ from nonterminals to sorts such that

- for every rule $(F\,x_1\,\ldots\,x_\ell\ \to\ t)\ \in\ \mathcal{R}$, there exist sorts $\kappa_1, \ldots, \kappa_\ell$ that satisfy $\mathcal{K}[x_1 \mapsto \kappa_1] \cdots [x_\ell \mapsto \kappa_\ell] \vdash_\Sigma t : \mathtt{o}$ and $\mathcal{K}(F) = \kappa_1 \to \cdots \to \kappa_\ell \to \mathtt{o}$, and
- $\mathcal{K}(S) = \mathtt{o}$ holds for the start symbol S,

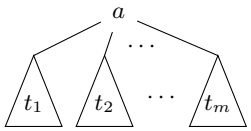
where each terminal $a \in \mathrm{dom}(\Sigma)$ has the type $\underbrace{\mathtt{o} \to \cdots \to \mathtt{o} \to}_{\Sigma(a)}\ \mathtt{o}$.

Given a HORS $\mathcal{G} = (\Sigma, \mathcal{R}, S)$, we define the *reduction relation* $\longrightarrow_\mathcal{G}$ as the least relation that satisfies:

- $(F\ t_1\ \cdots\ t_\ell) \longrightarrow_\mathcal{G} [t_\ell/x_\ell] \cdots [t_1/x_1]t$ if $(F\ x_1 \cdots x_\ell \to t) \in \mathcal{R}$,
- $(a\ t_1\ \cdots\ t_i\ \cdots\ t_m) \longrightarrow_\mathcal{G} (a\ t_1\ \cdots\ t'_i\ \cdots\ t_m)$ if $\Sigma(a) = m$ and $t_i \longrightarrow_\mathcal{G} t'_i$.

We write $\longrightarrow^*_\mathcal{G}$ for the reflexive and transitive closure of $\longrightarrow_\mathcal{G}$.

A *(finite) Σ-labeled ranked tree* (or simply a *tree*) is an applicative term consisting only of terminal symbols. As usual, we often depict the tree $a\,t_1\ \cdots\ t_n$ as follows.

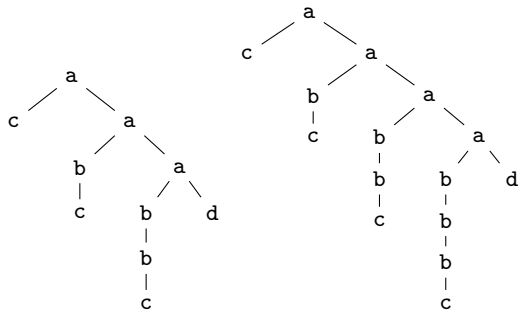


A *value tree* of a HORS $\mathcal{G}$ is a tree t such that $S \longrightarrow^*_\mathcal{G} t$. We write $\mathcal{L}(\mathcal{G})$ for the set of value trees generated by $\mathcal{G}$.

Example 4. Consider the HORS $\mathcal{G} = (\Sigma, \mathcal{R}, S)$ where:

$$\Sigma \coloneqq \{\mathtt{a} \mapsto 2, \mathtt{b} \mapsto 1, \mathtt{c} \mapsto 0, \mathtt{d} \mapsto 0\},$$
$$\mathcal{R} \coloneqq \{S \to F\ \mathtt{c}, F\ x \to \mathtt{a}\ x\ (F\ (\mathtt{b}\ x)), F\ x \to \mathtt{d}\}.$$

The following are value trees of $\mathcal{G}$:



Tree automata We use a tree automaton to specify a property of the trees generated by a HORS. A (non-deterministic, top-down) *tree automaton* on a ranked alphabet Σ is a quadruple $\mathcal{A} = (\Sigma, Q, \Delta, q^{\mathrm{INI}})$ consisting of

- the ranked alphabet Σ;
- a finite set Q of *states*;
- a finite set $\Delta \subseteq Q \times \Sigma \times Q^*$ of *transition rules* such that $(q, a, q_1 \cdots q_m) \in \Delta$ implies $m = \Sigma(a)$;
- a special state $q^{\mathrm{INI}} \in Q$ called the *initial state*.

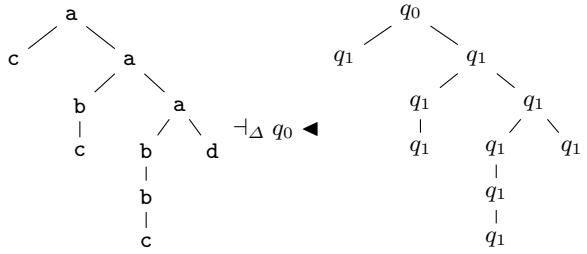
We write $q\ a \to q_1\ \cdots\ q_m$ for a transition rule $(q, a, q_1 \cdots q_m)$.

The set of *Q-labeled trees* is given by: $r\ ::=\ q\,r_1 \cdots r_m$ where q ranges over Q and $m \geq 0$. A Σ-labeled ranked tree t is said to be *accepted* by $\mathcal{A} = (\Sigma, Q, \Delta, q^{\mathrm{INI}})$ if there exists a (finite) Q-labeled tree r such that $t \dashv_\Delta q^{\mathrm{INI}} \blacktriangleleft r$, where the relation $t \dashv_\Delta q \blacktriangleleft r$ is inductively defined by: $(a\ t_1\ \cdots\ t_m) \dashv_\Delta q \blacktriangleleft r$ if r is of the form $(q\ r_1\ \cdots\ r_m)$ and there exists $(q\ a \to q_1\ \cdots\ q_m) \in \Delta$ such that $t_i \dashv_\Delta q_i \blacktriangleleft r_i$ for each $i \in [m]$. A Q-labeled tree r is called a *run-tree* of $\mathcal{A}$ over t if $t \dashv_\Delta q^{\mathrm{INI}} \blacktriangleleft r$ holds.

Example 5. Let Σ be a ranked alphabet in Example 4. Consider the automaton $\mathcal{A}_1 = (\Sigma, \{q_0, q_1\}, \Delta, q_0)$ where:

$$\Delta \coloneqq \{q_0\ \mathtt{a} \to q_1\ q_1, \quad q_1\ \mathtt{a} \to q_1\ q_1, \quad q_1\ \mathtt{b} \to q_1,$$
$$q_0\ \mathtt{c} \to \varepsilon, \quad q_1\ \mathtt{c} \to \varepsilon,$$
$$q_0\ \mathtt{d} \to \varepsilon, \quad q_1\ \mathtt{d} \to \varepsilon\}.$$

Intuitively, $\mathcal{A}_1$ accepts a tree t if every $\mathtt{b}$-labeled node of t has an ancestor node labeled with $\mathtt{a}$. For example, the tree on the left-hand side of Example 4 (as well as the one on the right-hand side) is accepted by $\mathcal{A}_1$, as witnessed by the following relation.



For another example of automata, let $\mathcal{A}_2 = (\Sigma, \{q\}, \Delta, q)$ where $\Delta \coloneqq \{q\ \mathtt{a} \to q\ q,\ q\ \mathtt{b} \to q,\ q\ \mathtt{c} \to \varepsilon\}$. This automaton checks whether every leaf of a given tree is labeled with $\mathtt{c}$. The tree above is not accepted by $\mathcal{A}_2$. ■

Higher-order model checking We say that a HORS $\mathcal{G}$ is *accepted* by a tree automaton $\mathcal{A}$ if all the value trees of $\mathcal{G}$ are accepted by $\mathcal{A}$. Higher-order model checking is the following decision problem:

> Given a HORS $\mathcal{G}$ and a tree automaton $\mathcal{A}$ on a common ranked alphabet Σ, is $\mathcal{G}$ accepted by $\mathcal{A}$?

Example 6. Let $\mathcal{G}_0 = (\Sigma, \mathcal{R}, S)$ be a HORS in Example 4 and $\mathcal{A}_1$ and $\mathcal{A}_2$ be automata in Example 5. The HORS $\mathcal{G}_0$ is accepted by $\mathcal{A}_1$, whereas $\mathcal{G}_1$ is not. ■

Higher-order model checking is decidable [21]. Despite its extremely high computational complexity, practical higher-order model checkers [2, 15, 24] are available that run fast for typical inputs.

3.2 Verifying closedness of generated code

Since the gensym language defined in Section 2 has independent mechanisms for generating (i.e. $\mathbf{gensym}/\mathbf{cont}$) and binding (i.e. $\mathtt{ABS}\ e_1\ e_2$ and $\mathtt{FIX}\ e_1\ e_2$) a symbol, it is possible for users to write mistakenly a program that can generate code with an unbound variable. In this section, we propose a method to check that every code generated by a given source program is closed. As sketched in the introduction, given a source program $(\mathcal{D}, S)$, we convert $(\mathcal{D}, S)$ to a HORS $\mathcal{G}_{\mathrm{cls}}(\mathcal{D}, S)$, and check whether $\mathcal{G}_{\mathrm{cls}}(\mathcal{D}, S)$ is accepted by an automaton $\mathcal{A}_{\mathrm{cls}}$ using higher-order model checking.

A code value v is *closed* if so is the obvious corresponding λ-term. This notion is formally defined as follows. A code value v is *well-formed* if, for every subterm of the form $\mathtt{ABS}\ v_1\ v_2$ or $\mathtt{FIX}\ v_1\ v_2$, $v_1 \in \mathbf{Symbol}$. We say that the occurrences of α in $\mathtt{ABS}\ \alpha\ v_2$ and $\mathtt{FIX}\ \alpha\ v_2$ are *binding occurrences* and every occurrence of the symbol α in v_2 is *bound*. An occurrence of a symbol is *unbound* if it is neither binding nor bound. A well-formed code value is *closed* if it has no unbound occurrence of a symbol.

Example 7. Let $v :\equiv \texttt{ABS}\ \alpha\ (\texttt{ABS}\ \beta\ (\texttt{ABS}\ \alpha\ (\texttt{APP}\ \alpha\ \beta)))$ be a code value. It has two binding occurrences of α, one binding occurrence of β, one bound occurrence of α and one bound occurrence of β. This is well-formed and closed. The code value $v' :\equiv \texttt{ABS}\ \alpha\ (\texttt{APP}\ \alpha\ \beta)$ is well-formed but not closed since the unique occurrence of β is unbound. The code value $v'' :\equiv \texttt{ABS}\ (\texttt{APP}\ \alpha\ \beta)\ \gamma$ and $v''' :\equiv \texttt{ABS}\ \texttt{'1}\ \texttt{'2}$ are not well-formed.

The main obstacles in the static verification of closedness of generated code are (i) the set of code values generated by the program may be infinite and (ii) the number of symbols occurring in a code value has no a priori bound. We can easily overcome the first problem by modeling a code generator as a HORS that generates the tree representation of code values. An additional trick is, however, required to address the second problem. Note that the set of terminal symbols occurring in a HORS and a tree automaton must be finite; thus, we need to properly model a code value containing an unbounded number of symbols with a finite set of terminals.

An important observation here is that whether an occurrence of α is bound is completely irrelevant to the usage of other symbols. Hence, for the purpose of checking if there is an unbound occurrence of α, the code value $v :\equiv \texttt{ABS}\ \alpha\ (\texttt{APP}\ \alpha\ (\texttt{ABS}\ \beta\ \gamma))$ has the same information as $\langle v\rangle_{\{\alpha\}} \equiv \texttt{ABS}\ \texttt{var}\ (\texttt{APP}\ \texttt{var}\ (\texttt{ABS}\ \texttt{ig}\ \texttt{ig}))$, where `var` represents the symbol that we currently focus on (that is α here) and `ig` represents the other symbols. Note that $\langle v\rangle_{\{\alpha\}}$ uses only a bounded number of symbols; in fact, it is a Σ_{cls}-labeled tree, where $\Sigma_{\mathrm{cls}} = \mathcal{P} \cup \{\texttt{var} \mapsto 0,\ \texttt{ig} \mapsto 0\}$. The notions of well-formedness and closedness are extended to $\langle v\rangle_{\{\alpha\}}$; we say that a Σ_{cls}-labeled tree is *closed* if it has no unbound `var`. Now closedness of v is reduced to that of $\langle v\rangle_{\{\alpha\}}$ for every $\alpha \in \mathbf{Symbol}$.

We formalize the above idea. Given a code value v and a subset $X \subseteq \mathbf{Symbol}$, the operation $\langle v\rangle_X$ replacing $\alpha \in X$ with `var` and $\beta \notin X$ with `ig` is defined by:

$$\langle \alpha\rangle_X :\equiv \begin{cases} \texttt{var} & (\text{if } \alpha \in X) \\ \texttt{ig} & (\text{otherwise}) \end{cases} \qquad \begin{aligned} \langle P\rangle_X &:\equiv P \\ \langle v_1\ v_2\rangle_X &:\equiv \langle v_1\rangle_X\ \langle v_2\rangle_X. \end{aligned}$$

As discussed above, given a code value v, it is well-formed and closed if and only if $\langle v\rangle_{\{\alpha\}}$ is well-formed and closed for every $\alpha \in \mathbf{Symbol}$. As we shall see in Lemma 9, this is equivalent to the condition that $\langle v\rangle_X$ is well-formed and closed for every $X \subseteq \mathbf{Symbol}$. (The latter is technically more convenient.)

Example 8. Let

$$v :\equiv \texttt{ABS}\ \alpha\ (\texttt{ABS}\ \beta\ (\texttt{ABS}\ \alpha\ (\texttt{APP}\ \alpha\ \beta)))$$

be a closed code value. There are essentially four choices of $X \subseteq \mathbf{Symbol}$, namely $X_0 := \varnothing$, $X_1 := \{\alpha\}$, $X_2 := \{\beta\}$ and $X_3 := \{\alpha, \beta\}$. We have

$$\begin{aligned} \langle v\rangle_{X_0} &\equiv \texttt{ABS ig}\ (\texttt{ABS ig}\ (\texttt{ABS ig}\ (\texttt{APP ig ig}))) \\ \langle v\rangle_{X_1} &\equiv \texttt{ABS var}\ (\texttt{ABS ig}\ (\texttt{ABS var}\ (\texttt{APP var ig}))) \\ \langle v\rangle_{X_2} &\equiv \texttt{ABS ig}\ (\texttt{ABS var}\ (\texttt{ABS ig}\ (\texttt{APP ig var}))) \\ \langle v\rangle_{X_3} &\equiv \texttt{ABS var}\ (\texttt{ABS var}\ (\texttt{ABS var}\ (\texttt{APP var var}))), \end{aligned}$$

all of which are closed. Consider a non-closed code value

$$v' :\equiv \texttt{ABS}\ \alpha\ (\texttt{APP}\ \alpha\ \beta),$$

in which β is unbound. Setting $X := \{\beta\}$, we have

$$\langle v'\rangle_X \equiv \texttt{ABS ig}\ (\texttt{APP ig var})$$

in which `var` is unbound.

Lemma 9. *For every code value v, the following conditions are equivalent:*

1. *v is well-formed and closed.*
2. *$\langle v\rangle_{\{\alpha\}}$ is well-formed and closed for all $\alpha \in \mathbf{Symbol}$.*
3. *$\langle v\rangle_X$ is well-formed and closed for all $X \subseteq \mathbf{Symbol}$.*

Proof. ($1 \Rightarrow 3$) If $\langle v\rangle_X$ is not well-formed, then v is not well-formed. Assume that $\langle v\rangle_X$ is not closed, i.e. it has an unbound occurrence of `var`. Let α be the symbol in v occurring at this position. Then this occurrence of α is unbound. ($3 \Rightarrow 2$) Obvious. ($2 \Rightarrow 1$) If v is not well-formed, then $\langle v\rangle_{\{\alpha\}}$ is not well-formed (for any α). Assume that v is not closed. Let α be a symbol which has an unbound occurrence. Then the corresponding occurrence of `var` in $\langle v\rangle_{\{\alpha\}}$ is unbound. ■

It is easy to see that the set of Σ_{cls}-labeled trees representing well-formed and closed code values is a regular tree language. Let $\mathcal{A}_{\mathrm{cls}}$ be the automaton that recognizes this language; the definition of $\mathcal{A}_{\mathrm{cls}}$ is given in Appendix.

Now what remains is to construct a HORS $\mathcal{G}_{\mathrm{cls}}(\mathcal{D}, S)$ for a given program $(\mathcal{D}, S)$ such that

$$\mathcal{L}(\mathcal{G}_{\mathrm{cls}}(\mathcal{D}, S)) = \left\{\langle v\rangle_X \;\middle|\; S \hookrightarrow_{\mathcal{D}}^{*} v, X \subseteq \mathbf{Symbol}\right\}.$$

Given a program $(\mathcal{D}, S)$, we define the HORS $\mathcal{G}_{\mathrm{cls}}(\mathcal{D}, S) = (\Sigma_{\mathrm{cls}}, \mathcal{R}, S)$ by:

$$\begin{aligned} \mathcal{R} :=\ & \left\{F\ \tilde{x} \to \langle e\rangle \;\middle|\; (F\ \tilde{x} = e) \in \mathcal{D}\right\} \\ & \cup \left\{\mathit{Gensym}\ k \to k\ \texttt{var},\ \mathit{Gensym}\ k \to k\ \texttt{ig}\right\} \end{aligned}$$

where the transformation $\langle -\rangle$ of terms into applicative terms of the HORS is given by:

$$\langle x\rangle :\equiv x, \qquad \langle F\rangle :\equiv F, \qquad \langle e_1\ e_2\rangle :\equiv \langle e_1\rangle\ \langle e_2\rangle,$$
$$\langle P\rangle :\equiv P, \qquad \langle \mathbf{gensym/cont}\rangle :\equiv \mathit{Gensym}.$$

The HORS $\mathcal{G}_{\mathrm{cls}}(\mathcal{D}, S)$ basically simulates the program $(\mathcal{D}, S)$ except for $\mathbf{gensym/cont}$. The nonterminal Gensym of the HORS (that corresponds to $\mathbf{gensym/cont}$) non-deterministically chooses whether or not the newly generated symbol should belong to $X \subseteq \mathbf{Symbol}$, and passes `var` or `ig` to the continuation accordingly. The enumeration of all the possible behaviours of Gensym corresponds to that of $X \subseteq \mathbf{Symbol}$ in $\langle -\rangle_X$.

Example 10. Consider the following variation of the `genpower` example in Section 1:

$$\begin{aligned} &\mathbf{letrec}\ \mathit{genpower\_fake}\ n\ x = \\ &\quad \mathbf{if}\ n \leq 0\ \mathbf{then}\ \texttt{'1}\ \mathbf{else} \\ &\qquad \texttt{TIMES}\ x\ (\mathit{genpower\_fake}\ (n-1)\ x)\ \mathbf{in} \\ &\boldsymbol{\lambda} n.\ \mathbf{let}\ x = \mathbf{gensym}\ ()\ \mathbf{in} \\ &\quad \mathbf{let}\ y = \mathbf{gensym}\ ()\ \mathbf{in}\ \texttt{ABS}\ y\ (\mathit{genpower\_fake}\ n\ x) \end{aligned}$$

and its corresponding program $(\mathcal{D}, S)$ in the gensym language:

$$\begin{aligned} \mathcal{D} = \{ & \mathit{GenPowerFake}\ x = \texttt{'1}, \\ & \mathit{GenPowerFake}\ x = \texttt{TIMES}\ x\ (\mathit{GenPowerFake}\ x), \\ & S = \mathbf{gensym/cont}\ K_1, \\ & K_1\ x = \mathbf{gensym/cont}\ (K_2\ x), \\ & K_2\ x\ y = \texttt{ABS}\ y\ (\mathit{GenPowerFake}\ x)\}. \end{aligned}$$

Clearly this program can produce a target code that is not closed.

It is transformed to the following HORS consisting of the following rules:

$$\begin{aligned} & \mathit{GenPowerFake}\ x \to \texttt{'1}, \\ & \mathit{GenPowerFake}\ x \to \texttt{TIMES}\ x\ (\mathit{GenPowerFake}\ x), \\ & S \to \mathit{Gensym}\ K_1, \end{aligned}$$

$K_1\ x \to \mathit{Gensym}\ (K_2\ x)$,
$K_2\ x\ y \to \texttt{ABS}\ y\ (\mathit{GenPowerFake}\ x)$,
$\mathit{Gensym}\ k \to k\ \texttt{var}, \quad \mathit{Gensym}\ k \to k\ \texttt{ig}$.

The HORS generates $\texttt{ABS ig}\ (\texttt{TIMES var}\ \cdots)$ if the first rule for Gensym is chosen in the body of K_1 and the second rule is chosen in the body of K_2. Thus, the HORS is rejected by $\mathcal{A}_{\mathrm{cls}}$ as expected. ■

We prove the correctness of our method below.

Lemma 11. *For every program $(\mathcal{D}, S)$ of the gensym language, we have*

$$\mathcal{L}(\mathcal{G}_{\mathrm{cls}}(\mathcal{D}, S)) = \left\{ \langle v \rangle_X \mid S \hookrightarrow_{\mathcal{D}}^* v, X \subseteq \mathbf{Symbol} \right\}.$$

Proof sketch. ($\supseteq$) Given a reduction sequence $S \hookrightarrow_{\mathcal{D}}^* v$ and a subset $X \subseteq \mathbf{Symbol}$, we construct a reduction sequence $S \longrightarrow_{\mathcal{G}_{\mathrm{cls}}(\mathcal{D},S)}^* t$ which simulates the reduction sequence of the program. A reduction step $\mathbf{gensym/cont}\ \hat{e} \hookrightarrow_{\mathcal{D}} \hat{e}\ \alpha$ corresponds to $\mathit{Gensym}\ t \longrightarrow_{\mathcal{G}_{\mathrm{cls}}(\mathcal{D},S)} t\ \texttt{var}$ if $\alpha \in X$ and otherwise $\mathit{Gensym}\ t \longrightarrow_{\mathcal{G}_{\mathrm{cls}}(\mathcal{D},S)} t\ \texttt{ig}$.

($\subseteq$) Assume a reduction sequence $S \longrightarrow_{\mathcal{G}_{\mathrm{cls}}(\mathcal{D},S)}^* t$. By using the bijective correspondence between rewriting rules in $\mathcal{G}_{\mathrm{cls}}(\mathcal{D}, S)$ and function definitions in $(\mathcal{D}, S)$, we can construct a reduction sequence $S \hookrightarrow_{\mathcal{D}}^* v$ in which $\mathbf{gensym/cont}$ does not generate the same symbol twice. The set $X \subseteq \mathbf{Symbol}$ is defined by $\alpha \in X$ if and only if $\mathbf{gensym/cont}\ \hat{e}' \hookrightarrow_{\mathcal{D}} \hat{e}'\ \alpha$ appears in the reduction sequence and the corresponding rewriting in the HORS is $\mathit{Gensym}\ t' \longrightarrow_{\mathcal{G}_{\mathrm{cls}}(\mathcal{D},S)} t'\ \texttt{var}$. Then $\langle v \rangle_X \equiv t$. ■

The next theorem is a consequence of Lemmas 9 and 11.

Theorem 12. *Let $(\mathcal{D}, S)$ be a source program of the gensym language. Then the following are equivalent:*

(1) *All code values generated by $(\mathcal{D}, S)$ are well-formed and closed.*

(2) *$\mathcal{L}(\mathcal{G}_{\mathrm{cls}}(\mathcal{D}, S))$ is accepted by $\mathcal{A}_{\mathrm{cls}}$.*

3.3 Verifying well-typedness of generated code

This section presents a method for checking that all the code values that can be generated by a given program are simply-typed. The set $\mathbf{TgType}$ of types of generated code fragments, ranged over by T, is defined by:

$$T ::= \texttt{Int} \mid \texttt{Float} \mid \texttt{Bool} \mid T \to T.$$

We assume every code constructor P in $\mathcal{P}$ (except ABS and FIX) is associated with a set of types:

$$\mathcal{P}^{\mathrm{type}} \colon \left(\mathrm{dom}(\mathcal{P}) \setminus \{\texttt{ABS}, \texttt{FIX}\}\right) \to 2^{\mathbf{TgType}}.$$

For example,

$$\mathcal{P}^{\mathrm{type}}(\texttt{APP}) := \left\{ (T_1 \to T_2) \to T_1 \to T_2 \mid (T_1 \to T_2) \in \mathbf{TgType} \right\}$$
$$\mathcal{P}^{\mathrm{type}}(\texttt{IFTE}) := \left\{ \texttt{Bool} \to T \to T \to T \mid T \in \mathbf{TgType} \right\}$$
$$\mathcal{P}^{\mathrm{type}}(\texttt{TIMES}) := \left\{ \texttt{Int} \to \texttt{Int} \to \texttt{Int}, \texttt{Float} \to \texttt{Float} \to \texttt{Float} \right\}.$$

Note that each constant may have more than one type. A code value v is well-typed (and has type T) if $\varnothing \vdash v : T$ is derivable using the rule for constants:

$$\frac{\left(T \in \mathcal{P}^{\mathrm{type}}(P)\right)}{\varnothing \vdash P : T}$$

and the standard typing rules for the simply-typed λ-calculus and recursion.

The problem that we address in this section is as follows:

Given a program $(\mathcal{D}, S)$ of the gensym language, are all the code values generated by $(\mathcal{D}, S)$ well-typed?

Example 13. Consider the following function generating a specialized power function for integers or floating numbers depending on the value of *option*:

$$\begin{array}{l}
\mathbf{letrec}\ \mathit{genpower_opt}\ \mathit{init}\ n\ x = \\
\quad \mathbf{if}\ n \le 0\ \mathbf{then}\ \mathit{init} \\
\qquad\qquad \mathbf{else}\ \texttt{TIMES}\ x\ (\mathit{genpower_opt}\ \mathit{init}\ (n-1)\ x)\ \mathbf{in} \\
\boldsymbol{\lambda} \mathit{option}.\ \boldsymbol{\lambda} n.\ \mathbf{let}\ x = \mathbf{gensym}\ ()\ \mathbf{in} \\
\quad \mathbf{if}\ \mathit{option}\ \mathbf{then}\ \mathit{genpower_opt}\ \texttt{'1}\ n\ x \\
\qquad\qquad \mathbf{else}\ \mathit{genpower_opt}\ \texttt{'1.0}\ n\ x
\end{array}$$

This function returns a well-typed code value for every pair of arguments but this is ill-typed under type systems for multi-stage languages such as $\lambda^{\bigcirc}$ [9]. In order to ensure well-typedness of generated code, it suffices to check well-typedness of all code values generated by the following program of the gensym language, which is an abstraction of the above function applied to random arguments:

$$\begin{array}{l}
\mathcal{D} := \{ S = \mathbf{gensym/cont}\ (\mathit{PowerOpt}\ \texttt{'1}), \\
\qquad S = \mathbf{gensym/cont}\ (\mathit{PowerOpt}\ \texttt{'1.0}), \\
\qquad \mathit{PowerOpt}\ \mathit{init}\ x = \mathit{init}, \\
\qquad \mathit{PowerOpt}\ \mathit{init}\ x = \texttt{TIMES}\ x\ (\mathit{PowerOpt}\ \mathit{init}\ x) \}.
\end{array}$$

As we shall see, our method automatically proves this.

Let $\mathcal{T} \subseteq_{\mathrm{fin}} \mathbf{TgType}$ be a finite subset of types that satisfies

$$(T_1 \to T_2) \in \mathcal{T} \quad \Rightarrow \quad \{T_1, T_2\} \subseteq \mathcal{T}.$$

We call an element of $\mathcal{T}$ a *candidate type*. Our method is paramaterized by $\mathcal{T}$ and checks whether a code value v has a type derivation that uses only types in $\mathcal{T}$. Let us fix $\mathcal{T}$ in the rest of this section.

Let v be a code value of which we would like to check well-typedness. Similarly to the idea in the previous subsection, we replace symbols in v with some elements of a finite set in such a way that the resulting tree has enough information to check the well-typedness of v. The key observation is that, once closedness of v has been established, it is sufficient to know the type of each symbol to check well-typedness. Let $\texttt{var}_T$ be a tree constructor of arity 0 for each $T \in \mathcal{T}$. Given a map $\theta \colon \mathbf{Symbol} \to \mathcal{T}$, we define $[v]_\theta^{\mathcal{T}}$ as the $\Sigma_{\mathrm{wt}}^{\mathcal{T}}$-labelled tree obtained by replacing α in v to $\texttt{var}_{\theta(\alpha)}$. The resulting trees are $(\mathcal{P} \cup \{\texttt{var}_T \mapsto 0 \mid T \in \mathcal{T}\})$-labelled trees.

We then give a type system for $(\mathcal{P} \cup \{\texttt{var}_T \mapsto 0 \mid T \in \mathcal{T}\})$-labelled trees as follows.

$$\frac{}{\texttt{var}_T : T} \qquad \frac{(T_1 \to \cdots \to T_n \to T) \in \mathcal{P}^{\mathrm{type}}(P) \quad t_1 : T_1\ \cdots\ t_n : T_n \quad T \in \mathcal{T}}{P\ t_1\ \cdots\ t_n : T}$$

$$\frac{v : T_2 \qquad (T_1 \to T_2) \in \mathcal{T}}{\texttt{ABS var}_{T_1}\ v : T_1 \to T_2} \qquad \frac{v : T_1 \to T_2}{\texttt{FIX var}_{T_1 \to T_2}\ v : T_1 \to T_2}$$

Thanks to the conditions $T \in \mathcal{T}$ and $(T_1 \to T_2) \in \mathcal{T}$ in the second and third rules, $v : T$ implies $T \in \mathcal{T}$. Since $\mathcal{T}$ is a finite set, one can construct an automaton that recognizes the set of all well-typed $(\mathcal{P} \cup \{\texttt{var}_T \mapsto 0 \mid T \in \mathcal{T}\})$-labelled trees. We write $\mathcal{W}_{\mathcal{T}}$ for this regular tree language.

Lemma 14. *Let v be a well-formed and closed code value. If there exists $\theta \colon \mathbf{Symbol} \to \mathcal{T}$ such that $[v]_\theta^{\mathcal{T}} \in \mathcal{W}_{\mathcal{T}}$, then v is well-typed.*

Example 15. Let $v :\equiv \mathtt{ABS}\ \alpha\ (\mathtt{ABS}\ \beta\ (\mathtt{IFTE}\ \alpha\ \beta\ \mathtt{'1}))$ and assume that $\mathcal{P}^{\mathrm{type}}(\mathtt{'1}) = \{\mathtt{Int}\}$. For a map θ such that $\theta(\alpha) \equiv \mathtt{Float}$ and $\theta(\beta) \equiv \mathtt{Bool}$, we have

$$[v]^{\mathcal{T}}_{\theta} \equiv \mathtt{ABS}\ \mathtt{var}_{\mathtt{Float}}\ (\mathtt{ABS}\ \mathtt{var}_{\mathtt{Bool}}\ (\mathtt{IFTE}\ \mathtt{var}_{\mathtt{Float}}\ \mathtt{var}_{\mathtt{Bool}}\ \mathtt{'1})),$$

which is not well-typed. For a map θ' such that $\theta'(\alpha) \equiv \mathtt{Bool}$ and $\theta'(\beta) \equiv \mathtt{Int}$, we have

$$[v]^{\mathcal{T}}_{\theta'} = \mathtt{ABS}\ \mathtt{var}_{\mathtt{Bool}}\ (\mathtt{ABS}\ \mathtt{var}_{\mathtt{Int}}\ (\mathtt{IFTE}\ \mathtt{var}_{\mathtt{Bool}}\ \mathtt{var}_{\mathtt{Int}}\ \mathtt{'1})),$$

which has type $\mathtt{Bool} \to \mathtt{Int} \to \mathtt{Int}$. Hence v has the same type and in particular well-typed. ■

Now the following proposition is a sufficient condition for the well-typedness of all the code values generated by $(\mathcal{D}, S)$:

$$\forall v \in \mathcal{C}(\mathcal{D}, S).\ \ \exists \theta : \mathbf{Symbol} \to \mathcal{T}.\ \ [v]^{\mathcal{T}}_{\theta} \in \mathcal{W}_{\mathcal{T}}.$$

We reduce this proposition to higher-order model checking. An important difference from the problem in Section 3.2 is that we have to handle the existential quantification. (Recall that higher-order model checking is a problem to decide if *every* value tree satisfies a certain property.)

We embed all possible choices of θ into a tree by the following technique. Let

$$\Sigma^{\mathcal{T}}_{\mathrm{wt}} \coloneqq \mathcal{P} \cup \left\{\mathtt{var}_T \mapsto 0 \;\middle|\; T \in \mathcal{T}\right\} \cup \left\{\mathtt{oneof} \mapsto \#\mathcal{T}\right\}$$

where $\#\mathcal{T}$ is the number of elements in $\mathcal{T}$. The tree constructor `oneof` means that at least one child should be well-typed. Hence it has the typing rule

$$\frac{t_i : T \text{ for some } i \in [n]}{\mathtt{oneof}\ t_1\ \cdots\ t_n : T}$$

Again well-typedness of $\Sigma^{\mathcal{T}}_{\mathrm{wt}}$-labelled trees is a regular property. We write $\mathcal{A}^{\mathcal{T}}_{\mathrm{wt}}$ for an automaton that recognizes the language; a construction of $\mathcal{A}^{\mathcal{T}}_{\mathrm{wt}}$ is given in Appendix.

A $\Sigma^{\mathcal{T}}_{\mathrm{wt}}$-labelled tree can be seen as a set of $(\Sigma^{\mathcal{T}}_{\mathrm{wt}} \setminus \{\mathtt{oneof}\})$-labelled tree. We write $\|t\|$ for the set of trees defined by:

$$\|\mathtt{oneof}\ t_1\ \cdots\ t_n\| \coloneqq \bigcup_{i \in [n]} \|t_i\|,$$

$$\|P\ t_1\ \cdots\ t_\ell\| \coloneqq \left\{P\ u_1\ \cdots\ u_\ell \;\middle|\; {}^{\forall} i \in [\ell].\ \ u_i \in \|t_i\|\right\}.$$

The next lemma justifies the intuition that `oneof` describes an existential quantification.

Lemma 16. *A $\Sigma^{\mathcal{T}}_{\mathrm{wt}}$-labelled tree t is accepted by $\mathcal{A}^{\mathcal{T}}_{\mathrm{wt}}$ if and only if ${}^{\exists} u \in \|t\|.\ \ u \in \mathcal{W}_{\mathcal{T}}$.*

For a code value v and a $\Sigma^{\mathcal{T}}_{\mathrm{wt}}$-labelled tree t, we write $v \sim t$ if $\|t\| = \left\{[v]^{\mathcal{T}}_{\theta} \;\middle|\; \theta\colon\ \mathbf{Symbol} \to \mathcal{T}\right\}$. Provided that $v \sim t$, then by Lemmas 14 and 16, a code value v is well-typed if t is accepted by $\mathcal{A}^{\mathcal{T}}_{\mathrm{wt}}$.

Given a program $(\mathcal{D}, S)$, we define a HORS $\mathcal{G}^{\mathcal{T}}_{\mathrm{wt}}(\mathcal{D}, S) = (\Sigma^{\mathcal{T}}_{\mathrm{wt}}, \mathcal{R}, S)$ such that, for every $v \in \mathcal{C}(\mathcal{D}, S)$, there exists $t \in \mathcal{L}(\mathcal{G}^{\mathcal{T}}_{\mathrm{wt}}(\mathcal{D}, S))$ such that $v \sim t$. The rewriting rules are defined by:

$$\mathcal{R} \coloneqq \left\{F\ \tilde{x} \to \langle e\rangle \;\middle|\; (F\ \tilde{x} = e) \in \mathcal{D}\right\}$$
$$\cup \left\{\mathit{Gensym}\ k \to \mathtt{oneof}\ (k\ \mathtt{var}_{T_1})\ \ \cdots (k\ \mathtt{var}_{T_n})\right\},$$

where $T_1, \ldots, T_n$ is an enumeration of $\mathcal{T}$. The operation $\langle e\rangle$ is defined in Section 3.2, which replaces $\mathbf{gensym}/\mathbf{cont}$ in e to Gensym. The HORS $\mathcal{G}^{\mathcal{T}}_{\mathrm{wt}}(\mathcal{D}, S)$ basically simulates the program $(\mathcal{D}, S)$ except for $\mathbf{gensym}/\mathbf{cont}$. The nonterminal Gensym guesses a correct type for the newly generated symbol. The next lemma shows that the HORS generates an expected tree language.

Lemma 17. *For every $v \in \mathcal{C}(\mathcal{D}, S)$, there exists $t \in \mathcal{L}(\mathcal{G}^{\mathcal{T}}_{\mathrm{wt}}(\mathcal{D}, S))$ such that $v \sim t$.*

Proof sketch. First we extend $[-]^{\mathcal{T}}_{\theta}$ to expressions and $\|-\|$ to applicative terms. For an expression $\hat{e}$ and an applicative term t (that is not necessarily a tree), the relation $\hat{e} \sim t$ is defined by the obvious way. The claim extended to arbitrary expressions and applicative terms can be proved by induction on the length of reduction sequences. ■

As a consequence of Lemmas 14, 16 and 17, we have the following theorem.

Theorem 18 (Soundness)**.** *Let $(\mathcal{D}, S)$ be a program of the gensym language generating well-formed and closed code values. If $\mathcal{L}(\mathcal{G}^{\mathcal{T}}_{\mathrm{wt}}(\mathcal{D}, S))$ is accepted by $\mathcal{A}^{\mathcal{T}}_{\mathrm{wt}}$, then all code values generated by $(\mathcal{D}, S)$ are well-typed.*

Note that whether $\mathcal{L}(\mathcal{G}^{\mathcal{T}}_{\mathrm{wt}}(\mathcal{D}, S))$ is accepted by $\mathcal{A}^{\mathcal{T}}_{\mathrm{wt}}$ is an instance of higher-order model checking and thus decidable.

Unfortunately, our method is incomplete, as discussed below. The first source of incompleteness comes from the choice of $\mathcal{T}$, which must be finite to keep $\mathcal{W}_{\mathcal{T}}$ regular. There is a program of the gensym language such that every generated code value has a type but the set of all types of the generated code values are infinite. An example is the program $(\mathcal{D}, S)$ defined by:

$$\mathcal{D} \coloneqq \{S = \mathtt{'1},\ S = \mathbf{gensym}/\mathbf{cont}\ T,\ T\ x = \mathtt{ABS}\ x\ S\}$$

which generates a code value $\lambda\alpha_1.\ \ldots \lambda\alpha_\ell.\ 1$ (written in the standard syntax) for every $\ell \in \mathbf{N}$.

The second source of incompleteness comes from the fact that we assign a type to each symbol. Consider a code value

$$v :\equiv \mathtt{ABS}\ \alpha\ (\mathtt{APP}\ \alpha\ (\mathtt{ABS}\ \alpha\ (\mathtt{TIMES}\ \alpha\ \mathtt{'1}))),$$

which can be written in the standard syntax as $\lambda\alpha.\ \alpha\,(\lambda\alpha.\ \alpha * 1)$. This code value is well-typed: an example of a correct type annotation is:

$$\lambda\alpha^{(\mathtt{Int}\to\mathtt{Int})\to\mathtt{Int}}.\ \alpha\,(\lambda\alpha^{\mathtt{Int}}.\ \alpha * 1).$$

Observe that different occurrences of α have different types. This is inevitable to type v. Hence all elements of $\left\{[v]^{\mathcal{T}}_{\theta} \;\middle|\; \theta\colon\ \mathbf{Symbol} \to \mathcal{T}\right\}$ are ill-typed, whatever $\mathcal{T}$ is.

Remark 19. The method above can be extended to intersection types and/or refinement types, as long as the set $\mathcal{T}$ of type candidates is finite. For example, with intersection types, the first two rules for typing labeled trees would become:

$$\frac{}{\mathtt{var}_{T_1 \wedge \cdots \wedge T_k} : T_i}$$

$$\frac{\begin{array}{c}(T_{1,1} \wedge \cdots \wedge T_{1,k_1}) \to \cdots \to (T_{n,1} \wedge \cdots \wedge T_{n,k_n}) \to T \\ \in \mathcal{P}^{\mathrm{type}}(P) \cap \mathcal{T} \\ t_i : T_{i,j} \text{ for each } i \in [n], j \in [k_i]\end{array}}{P\ t_1\ \cdots\ t_n : T}$$

As long as $\mathcal{T}$, the above rules can be expressed as transition rules of an alternating tree automaton. (The observation that the typability in intersection types can be captured by a tree automaton is not novel; see, e.g., [25].) Using intersection types and/or refinement types, we can check more complex properties of generated code.

Program	$\#\mathcal{T}$	Answer C	Answer WT	Time [s] C	Time [s] WT
	2				0.004
`genpower`	10	Yes	Yes	0.004	0.012
	16				0.016
`genpower_fake`	2	No	–	0.004	–
	4				0.008
`genpower_option`	10	Yes	Yes	0.004	0.014
	16				0.098
	2				0.006
`effgenpower`	10	Yes	Yes	0.004	0.025
	16				0.032
	4				0.008
`geniprod`	10	Yes	Yes	0.008	0.018
	16				0.069
	5				0.012
`gentranspose`	10	Yes	Yes	0.008	0.022
	16				0.091

Table 1. The Result of Preliminary Experiments

4. Experiments

Based on the verification methods described in Section 3, we have implemented a prototype verification tool for code generators and carried out preliminary experiments. The tool supports a call-by-value language with OCaml-like syntax and the gensym primitive. A source program is converted to a program in Section 2 by replacing all the conditionals with non-deterministic branches; we have not integrated predicate abstraction yet and applying the CPS transformation. The program is then transformed to HORS, based on the transformations described in Sections 3.2 and 3.3. HORSAT2 [2] is used as the backend higher-order model checker.

We have tested the tool on a machine with an Intel Core-i5 CPU with 2.60GHz and 4.00GB memory. The result is shown in Table 1, where "C" and "WT" mean the closedness check and the well-typedness check, respectively, and the column "$\#\mathcal{T}$" shows the number of types considered as candidates of the types of generated symbols in the well-typedness check. The benchmark programs used in the experiments are the following ones:

`genpower`: a function that takes a nonnegative integer n and returns code for $\lambda x.\ x^n$, described in Section 1 and Example 1.

`genpower_fake`: a buggy version of `genpower`, which is described in Example 10.

`genpower_option`: an extended version of `genpower`, which is shown in Example 13.

`effgenpower`: a variant of `genpower`, which generates code for a function that takes an integer x as an input and computes x^n in time $\mathrm{O}(\log n)$.

`geniprod`: a function that takes a nonnegative integer n and returns code that takes a couple of n-dimentional vectors u, v (represented by lists of length n) as inputs and computes the inner product $u^\top v$.

`gentranspose`: a function that takes nonnegative integers m, n and returns code that transposes matrices of size $m \times n$.

Here, the types of code values have been extended with list types: $T ::= \cdots \mid T\ \texttt{List}$, and we have added code costructors for lists: $(\texttt{NIL} \mapsto 0), (\texttt{CONS} \mapsto 2), (\texttt{CAR} \mapsto 1), (\texttt{CDR} \mapsto 1) \in \mathcal{P}$. The benchmark programs are available at `http://www-kb.is.s.u-tokyo.ac.jp/~suwa/benchmark1.zip`.

For all the programs, the tool terminated quickly and output expected answers.

5. Verification of Two-Stage Programs

Since it may be cumbersome to write code generators in the gensym language in Section 2, we introduce in this section a two-stage programming language called Relaxed $\lambda^{\bigcirc}$ and formalize a translation to (a variant of) the gensym language. Our two-stage programming language lies somewhere between the untyped multi-stage language λU [29] and the typed multi-stage language $\lambda^{\bigcirc}$ [9]. It has only quasiquotation and unquote as constructs for staging (like $\lambda^{\bigcirc}$). Typing is more relaxed, however, than $\lambda^{\bigcirc}$ proper in that it is applied only for stage 0; the types of terms at the next stage are collapsed to a single type `code`. The translation is essentially the one presented in Calcagno et al. [4][1], who study compilation from λU [29] to a single-stage language with a special datatype for ASTs, gensym, and reflection. We show that the translation preserves typing, which is not studied in [4] (because the source language is untyped). This property is important for our verification methods to be applicable to Relaxed $\lambda^{\bigcirc}$. We do not discuss in detail another important property that the translation preserves semantics but it should straightforwardly be adapted from the results in [4].

The sets of terms and types, ranged over by M and τ respectively, are defined by:

$$\begin{aligned} M &::= c(M,\ \ldots,\ M) \mid x \mid M\ M \mid \boldsymbol{\lambda} x.\ M \mid \mathbf{fix}\ x.\ \boldsymbol{\lambda} x.\ M \\ &\quad \mid \mathbf{if}\ M\ \mathbf{then}\ M\ \mathbf{else}\ M \mid \langle\!| M |\!\rangle \mid \tilde{}M\ , \\ \tau &::= \texttt{int} \mid \texttt{float} \mid \texttt{bool} \mid \texttt{code} \mid \tau \to \tau. \end{aligned}$$

Here, c and x range over the sets of constants (such as integers, booleans, and primitive operations on them) and variables, respectively. The expression $c(M_1,\ \ldots,\ M_m)$ applies the m-ary constant c to arguments $M_1, \ldots, M_m$. We often write c for $c()$. When $c = \oplus$ is binary, we sometimes use the infix notation and write $M_1 \oplus M_2$ for $\oplus(M_1,\ M_2)$. The term $\langle\!| M |\!\rangle$ corresponds to quasiquote in Lisp and represents (quoted) code of M; the term $\tilde{}M$—which should appear inside quasiquotation—escapes from the quote, evaluates M, and embeds the value of M, which is expected to be a code value, into the surrounding quotation. We omit the operational semantics, which we assume to be call-by-value, as it is basically the same as that of λU [4, 29].

Example 20. The program in Example 1 can be written in Relaxed $\lambda^{\bigcirc}$ as follows:

$$\begin{aligned} &\mathbf{letrec}\ \mathit{power}\ n\ x = \\ &\qquad \mathbf{if}\ n = 0\ \mathbf{then}\ \langle\!| 1 |\!\rangle\ \mathbf{else}\ \langle\!| \tilde{}x * \tilde{}(\mathit{power}\ (n-1)\ x) |\!\rangle\ \mathbf{in} \\ &\boldsymbol{\lambda} n.\ \langle\!| \boldsymbol{\lambda} x.\ \tilde{}(\mathit{power}\ n\ \langle\!| x |\!\rangle) |\!\rangle \end{aligned}$$

where $\mathbf{letrec}\ x_0\ x_1\ \cdots\ x_\ell\ =\ M_1\ \mathbf{in}\ M_2$ is a shorthand for $(\boldsymbol{\lambda} x_0.\ M_2)\ (\mathbf{fix}\ x_0.\ \boldsymbol{\lambda} x_1.\ \cdots \boldsymbol{\lambda} x_\ell.\ M_1)$. ■

We present a type system for Relaxed $\lambda^{\bigcirc}$. As we mentioned, we will restrict the set of stages to $\{0, 1\}$. The type judgments $\Gamma \vdash^0 M : \tau$ for stage 0 and $\Gamma \vdash^1 M : \bigcirc$ for stage 1 are defined as follows:

$$\frac{\big(\mathcal{C}(c) = \tau_1 \to \cdots \to \tau_m \to \tau\big) \qquad \Gamma \vdash^0 M_i : \tau_i \text{ for each } i \in [m]}{\Gamma \vdash^0 c(M_1,\ \ldots,\ M_m) : \tau}$$

$$\frac{\big(\Gamma(x) = \tau\big)}{\Gamma \vdash^0 x : \tau} \qquad \frac{\Gamma \vdash^0 M_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash^0 M_2 : \tau_1}{\Gamma \vdash^0 M_1\ M_2 : \tau_2}$$

[1] in fact, a degenerated case of theirs

$$\frac{\Gamma[x \mapsto \tau_1] \vdash^0 M : \tau_2}{\Gamma \vdash^0 \boldsymbol{\lambda} x.\, M : \tau_1 \to \tau_2} \qquad \frac{\Gamma[x \mapsto \tau] \vdash^0 M : \tau}{\Gamma \vdash^0 \mathbf{fix}\, x.\, M : \tau}$$

$$\frac{\Gamma \vdash^0 M_0 : \mathtt{bool} \qquad \Gamma \vdash^0 M_1 : \tau \qquad \Gamma \vdash^0 M_2 : \tau}{\Gamma \vdash^0 \mathbf{if}\, M_0\, \mathbf{then}\, M_1\, \mathbf{else}\, M_2 : \tau}$$

$$\frac{\Gamma \vdash^1 M : \bigcirc}{\Gamma \vdash^0 \langle\!| M |\!\rangle : \mathtt{code}} \qquad \frac{\Gamma \vdash^0 M : \mathtt{code}}{\Gamma \vdash^1 \tilde{}M : \bigcirc}$$

$$\frac{\big(\mathcal{C}(c) = \tau_1 \to \cdots \to \tau_m \to \tau\big) \quad \Gamma \vdash^1 M_i : \bigcirc \text{ for each } i \in [m]}{\Gamma \vdash^1 c(M_1,\, \ldots,\, M_m) : \bigcirc}$$

$$\frac{\big(\Gamma(x) = \bigcirc\big)}{\Gamma \vdash^1 x : \bigcirc} \qquad \frac{\Gamma \vdash^1 M_1 : \bigcirc \qquad \Gamma \vdash^1 M_2 : \bigcirc}{\Gamma \vdash^1 M_1\, M_2 : \bigcirc}$$

$$\frac{\Gamma[x \mapsto \bigcirc] \vdash^1 M : \bigcirc}{\Gamma \vdash^1 \boldsymbol{\lambda} x.\, M : \bigcirc} \qquad \frac{\Gamma[x \mapsto \bigcirc] \vdash^1 M : \bigcirc}{\Gamma \vdash^1 \mathbf{fix}\, x.\, M : \bigcirc}$$

$$\frac{\Gamma \vdash^1 M_0 : \bigcirc \qquad \Gamma \vdash^1 M_1 : \bigcirc \qquad \Gamma \vdash^1 M_2 : \bigcirc}{\Gamma \vdash^1 \mathbf{if}\, M_0\, \mathbf{then}\, M_1\, \mathbf{else}\, M_2 : \bigcirc}$$

Here, $\mathcal{C}$ maps each constant to its type. In Relaxed $\lambda^{\bigcirc}$, terms are simply-typed at stage 0, while only the occurrence of variables is controlled by types at stage 1.

The target language of the translation is actually an intermediate language, which was already used in Examples 1 and 10, between Relaxed $\lambda^{\bigcirc}$ and the gensym language. We first set $\mathcal{P}_{\mathcal{C}}$ to be the map defined by:

$$\begin{aligned}\mathcal{P}_{\mathcal{C}} \coloneqq \big\{ \mathtt{'c} \mapsto m \,\big|\, \mathcal{C}(c) \equiv \tau_1 \to \cdots \to \tau_m \to \tau \big\} \\ \cup \big\{ \mathtt{APP} \mapsto 2, \mathtt{ABS} \mapsto 2, \mathtt{FIX} \mapsto 2, \mathtt{IFTE} \mapsto 3 \big\}.\end{aligned}$$

For example, if $\mathcal{C}(+) \equiv \mathtt{int} \to \mathtt{int} \to \mathtt{int}$, then $\mathcal{P}_{\mathcal{C}}(\mathtt{'}+) = 2$. The set of terms, ranged over by N, is defined by:

$$\begin{aligned}N \;::=\; & c \mid x \mid N\, N \mid \boldsymbol{\lambda} x.\, N \mid \mathbf{fix}\, x.\, N \\ & \mid \mathbf{if}\, N\, \mathbf{then}\, N\, \mathbf{else}\, N \mid P \mid \mathbf{gensym}\,()\end{aligned}$$

where c and P range over the set of constants and $\mathrm{dom}(\mathcal{P}_{\mathcal{C}})$, respectively. The type judgment $\Gamma \vdash_{\mathrm{m}} N : \tau$ for the intermediate language is defined by the following typing rules:

$$\frac{\big(\mathcal{C}(c) = \tau\big)}{\Gamma \vdash_{\mathrm{m}} c : \tau} \qquad \frac{\big(\Gamma(x) = \tau\big)}{\Gamma \vdash_{\mathrm{m}} x : \tau}$$

$$\frac{\Gamma \vdash_{\mathrm{m}} N_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash_{\mathrm{m}} N_2 : \tau_1}{\Gamma \vdash_{\mathrm{m}} N_1\, N_2 : \tau_2}$$

$$\frac{\Gamma[x \mapsto \tau_1] \vdash_{\mathrm{m}} N : \tau_2}{\Gamma \vdash_{\mathrm{m}} \boldsymbol{\lambda} x.\, N : \tau_1 \to \tau_2} \qquad \frac{\Gamma[x \mapsto \tau] \vdash_{\mathrm{m}} N : \tau}{\Gamma \vdash_{\mathrm{m}} \mathbf{fix}\, x.\, N : \tau}$$

$$\frac{\Gamma \vdash_{\mathrm{m}} N_0 : \mathtt{bool} \qquad \Gamma \vdash_{\mathrm{m}} N_1 : \tau \qquad \Gamma \vdash_{\mathrm{m}} N_2 : \tau}{\Gamma \vdash_{\mathrm{m}} \mathbf{if}\, N_0\, \mathbf{then}\, N_1\, \mathbf{else}\, N_2 : \tau}$$

$$\frac{\Gamma \vdash_{\mathrm{m}} P : \underbrace{\mathtt{code} \to \cdots \to \mathtt{code}}_{\mathcal{P}_{\mathcal{C}}(P)} \to \mathtt{code}}{\Gamma \vdash_{\mathrm{m}} \mathbf{gensym}\,() : \mathtt{code}}$$

Note that type environments for Relaxed $\lambda^{\bigcirc}$ possibly map a variable to the symbol $\bigcirc$, while those for the intermediate language do not. The translation $\lfloor - \rfloor^i_{\Psi}$ of Relaxed $\lambda^{\bigcirc}$ to the intermediate language with a map Ψ from variables to the set $\{0, 1\}$ of stages is defined by:

$$\begin{aligned}
\lfloor c(M_1,\, \ldots,\, M_m) \rfloor^0_{\Psi} &\equiv c\, \lfloor M_1 \rfloor^0_{\Psi}\, \cdots\, \lfloor M_m \rfloor^0_{\Psi}, \\
\lfloor x \rfloor^0_{\Psi} &\equiv x \qquad \text{if } \Psi(x) = 0, \\
\lfloor M_1\, M_2 \rfloor^0_{\Psi} &\equiv \lfloor M_1 \rfloor^0_{\Psi}\, \lfloor M_2 \rfloor^0_{\Psi}, \\
\lfloor \boldsymbol{\lambda} x.\, M \rfloor^0_{\Psi} &\equiv \boldsymbol{\lambda} x.\, \lfloor M \rfloor^0_{\Psi[x \mapsto 0]}, \\
\lfloor \mathbf{fix}\, x.\, M \rfloor^0_{\Psi} &\equiv \mathbf{fix}\, x.\, \lfloor M \rfloor^0_{\Psi[x \mapsto 0]}, \\
\lfloor \mathbf{if}\, M_0\, \mathbf{then}\, M_1\, \mathbf{else}\, M_2 \rfloor^0_{\Psi} &\equiv \\
&\quad \mathbf{if}\, \lfloor M_0 \rfloor^0_{\Psi}\, \mathbf{then}\, \lfloor M_1 \rfloor^0_{\Psi}\, \mathbf{else}\, \lfloor M_2 \rfloor^0_{\Psi}, \\
\lfloor \langle\!| M |\!\rangle \rfloor^0_{\Psi} &\equiv \lfloor M \rfloor^1_{\Psi}, \\
\lfloor c(M_1,\, \ldots,\, M_m) \rfloor^1_{\Psi} &\equiv \mathtt{'c}\, \lfloor M_1 \rfloor^1_{\Psi}\, \cdots\, \lfloor M_m \rfloor^1_{\Psi}, \\
\lfloor x \rfloor^1_{\Psi} &\equiv x \qquad \text{if } \Psi(x) = 1, \\
\lfloor M_1\, M_2 \rfloor^1_{\Psi} &\equiv \mathtt{APP}\, \lfloor M_1 \rfloor^1_{\Psi}\, \lfloor M_2 \rfloor^1_{\Psi}, \\
\lfloor \boldsymbol{\lambda} x.\, M \rfloor^1_{\Psi} &\equiv \\
&\quad \mathbf{let}\, x = \mathbf{gensym}\,()\, \mathbf{in}\, \mathtt{ABS}\, x\, \lfloor M \rfloor^1_{\Psi[x \mapsto 1]}, \\
\lfloor \mathbf{fix}\, x.\, M \rfloor^1_{\Psi} &\equiv \\
&\quad \mathbf{let}\, x = \mathbf{gensym}\,()\, \mathbf{in}\, \mathtt{FIX}\, x\, \lfloor M \rfloor^1_{\Psi[x \mapsto 1]}, \\
\lfloor \mathbf{if}\, M_0\, \mathbf{then}\, M_1\, \mathbf{else}\, M_2 \rfloor^1_{\Psi} &\equiv \mathtt{IFTE}\, \lfloor M_0 \rfloor^1_{\Psi}\, \lfloor M_1 \rfloor^1_{\Psi}\, \lfloor M_2 \rfloor^1_{\Psi}, \\
\lfloor \tilde{} M \rfloor^1_{\Psi} &\equiv \lfloor M \rfloor^0_{\Psi}.
\end{aligned}$$

Here, we write $\mathbf{let}\, x = M_1\, \mathbf{in}\, M_2$ as a shorthand for $(\boldsymbol{\lambda} x.\, M_2)\, M_1$.

Example 21. The program in Example 20 is translated as follows:

$$\begin{aligned}
&\lfloor \mathbf{fix}\ \mathit{power}.\ \boldsymbol{\lambda} n.\ \boldsymbol{\lambda} x.\ \mathbf{if}\ n = 0\ \mathbf{then}\ \langle\!| 1 |\!\rangle\ \mathbf{else}\ \ldots \rfloor^0_{\varnothing} \\
&\equiv \mathbf{fix}\ \mathit{power}.\ \boldsymbol{\lambda} n.\ \boldsymbol{\lambda} x.\ \mathbf{if}\ n = 0\ \mathbf{then}\ \lfloor \langle\!| 1 |\!\rangle \rfloor^0_{\Psi_1}\ \mathbf{else} \\
&\qquad\qquad \lfloor \langle\!| \tilde{}x * \tilde{}(\mathit{power}\ (n-1)\ x) |\!\rangle \rfloor^0_{\Psi_1} \\
&\equiv \mathbf{fix}\ \mathit{power}.\ \boldsymbol{\lambda} n.\ \boldsymbol{\lambda} x.\ \mathbf{if}\ n = 0\ \mathbf{then}\ \mathtt{'1}\ \mathbf{else} \\
&\qquad\qquad \mathtt{TIMES}\ \lfloor \tilde{}x \rfloor^1_{\Psi_1}\ \lfloor \tilde{}(\mathit{power}\ (n-1)\ x) \rfloor^1_{\Psi_1} \\
&\equiv \mathbf{fix}\ \mathit{power}.\ \boldsymbol{\lambda} n.\ \boldsymbol{\lambda} x.\ \mathbf{if}\ n = 0\ \mathbf{then}\ \mathtt{'1}\ \mathbf{else} \\
&\qquad\qquad \mathtt{TIMES}\ x\ (\mathit{power}\ (n-1)\ x)
\end{aligned}$$

where $\Psi_1 \coloneqq \{\mathit{power} \mapsto 0, n \mapsto 0, x \mapsto 0\}$, and:

$$\begin{aligned}
&\lfloor \boldsymbol{\lambda} n.\, \langle\!| \boldsymbol{\lambda} x.\, \tilde{}(\mathit{power}\ n\, \langle\!| x |\!\rangle) |\!\rangle \rfloor^0_{\{\mathit{power} \mapsto 0\}} \\
&\equiv \boldsymbol{\lambda} n.\ \mathbf{let}\ x = \mathbf{gensym}\,()\ \mathbf{in} \\
&\qquad\qquad \lfloor \tilde{}(\mathit{power}\ n\, \langle\!| x |\!\rangle) \rfloor^1_{\{\mathit{power} \mapsto 0, n \mapsto 0, x \mapsto 1\}} \\
&\equiv \boldsymbol{\lambda} n.\ \mathbf{let}\ x = \mathbf{gensym}\,()\ \mathbf{in}\ \mathit{power}\ n\ x\ .
\end{aligned}$$

The following theorem asserts the validity of the translation:

Theorem 22. *Let Γ be a type environment, M be a term and τ be a type of Relaxed $\lambda^{\bigcirc}$. Then the following hold:*

(1) *If $\Gamma \vdash^0 M : \tau$, then $\lfloor \Gamma \rfloor \vdash_{\mathrm{m}} \lfloor M \rfloor^0_{\Psi_\Gamma} : \tau$.*

(2) *If $\Gamma \vdash^1 M : \bigcirc$, then $\lfloor \Gamma \rfloor \vdash_{\mathrm{m}} \lfloor M \rfloor^1_{\Psi_\Gamma} : \mathtt{code}$.*

where, for a type environment Γ, the map $\Psi_\Gamma \colon \mathrm{dom}(\Gamma) \to \{0, 1\}$ and the type environment $\lfloor \Gamma \rfloor$ are defined by:

$$\Psi_\Gamma(x) \coloneqq \begin{cases} 1 & (\textit{if } \Gamma(x) \equiv \bigcirc) \\ 0 & (\textit{otherwise}), \end{cases} \qquad \lfloor \Gamma \rfloor(x) \coloneqq \begin{cases} \mathtt{code} & (\textit{if } \Gamma(x) \equiv \bigcirc) \\ \Gamma(x) & (\textit{otherwise}). \end{cases}$$

Thanks to the theorem above, we can apply the verification methods proposed in Section 3 to programs written in Relaxed $\lambda^{\bigcirc}$; note that well-typed programs of the intermediate language can be trans-

lated to those of the gensym language of Section 2. The proof of Theorem 22 is given in Appendix.

6. Discussions

We discuss limitations of our approach in Section 6.1 and related work in Section 6.2.

6.1 Limitations

There are some fundamental limitations of our approach that come from the use of higher-order model checking. The first limitation is that the gensym language must be simply-typed; this is due to the limitation of higher-order model checking that models (called higher-order recursion schemes) must be simply typed in order for the model checking problem to be decidable. There are a few ways to relax the restriction, however. First, the let-polymorphism (without polymorphic recursion) can be allowed in the gensym language, because programs can be converted to simply-typed programs by inlining all the let-definitions. Second, a certain class of recursive data structures (like lists) can be encoded as functions [26]. Third, if we give up the decidability, we can replace higher-order model checking with μHORS model checking [16], an extension of higher-order model checking with recursive types.

Another limitation of our approach is that we cannot support the so-called *Run* primitive, the primitive for running target code at the current stage. With the run primitive, for example, we could express the following computation:

$$\begin{aligned}&(\mathbf{run}\ (\mathbf{let}\ x = \mathbf{gensym}\ ()\ \mathbf{in}\ \texttt{ABS}\ x\ (\mathit{genpower}\ 2\ x)))\ 3\\ &\to^* (\mathbf{run}\ (\texttt{ABS X (TIMES X (TIMES X '1))))}\ 3\\ &\to (\boldsymbol{\lambda} x.\ x * (x * 1))\ 3 \to^* 9\end{aligned}$$

where $\mathit{genpower}$ is the function defined in Section 1. The lack of the run-primitive comes from the limitation of higher-order model checking that trees cannot be deconstructed in higher-order recursion schemes. One way to remedy the problem would be to use EHMTT verification methods [34], instead of higher-order model checking.

6.2 Related Work

6.2.1 Multi-stage programming

Since the seminal work by Davies and Pfenning [10], a lot of type systems for safe dynamic code generation have been studied [3, 5, 8, 13, 20, 30–33, 36]. Although they differ in the choices of language primitives for code generation (and, in some cases, execution), most of them aim at checking validity of generated code when the code generator is typechecked[2]. As the line of those studies shows, it is not very easy to design a type system that is both flexible and safe. Also, the property we would like to check is hard-coded into the type system. With our approach, code values are given a single type, which gives much flexibility, and properties to check are not hard-coded. We can also make good use of standard techniques for model checking such as predicate abstraction. One drawback of our approach would be the lack of modularity: the whole code generator has to be available to check its property, whereas type systems usually allow for separate checking (if the types of free variables are given).

Static analysis for multi-stage languages has been studied by Choi et al. [6]. They give a translation from a multi-stage language to an ordinary, single-stage language, where their semantics correspondence enables static analysis on a multi-stage program by analyzing the translated program by standard abstract interpretation. In their translation, a code value is represented by a function from the values of its free variables to the execution result of the code. So, syntactic information is lost. In fact, they assume a certain type system similar to [13], which can check well-typedness of generated code and translation is type-based. In some sense, their focus is more on semantic properties of generated code. Rather, our technique is closer to string analysis such as [19] in the sense that data structures to be generated by a program are checked against a grammar. Our technique is more powerful than grammar-based string analysis because of the use of more powerful grammars, namely higher-order recursion schemes; indeed, we can extend our technique to checking more semantic properties as well, by encoding a more refined type system such as an intersection type system into a tree automaton (recall Remark 19 at the end of Section 3).

Other frameworks for verification of multi-stage languages have been also studied. Berger and Tratt [1] have developed a Hoare logic for a variant of MiniML$^{\Box}_{e}$ [10]; Concoqtion [11] and Ωmega [27] have introduced very expressive type systems with indexed types into multi-stage languages. Although they can also verify complex semantic properties of code generators, they require a lot of human assistance to verify.

Our translation from the staged language to the gensym language is very similar to the compilation scheme for multi-stage languages [4], where code values are represented by a special datatype for ASTs. Our setting is much simpler because we do not deal with eval and cross-stage persistence.

6.2.2 Higher-order model checking

Model checking of HORS and its application to program verification has been studied extensively [14, 15, 17, 18, 21, 22]. Verification of higher-order tree transducers studied by Kobayashi, Unno, and Tabuchi [18] is closely related to the present work in that it is a technique to verify the shape of trees output by a certain class of functional programs. One notable difference is that our work deals with binding structure in output trees. Kobayashi [14] showed reduction of the resource usage problem [12] to higher-order model checking. The way how multiple variable are treated in this work has been inspired by his method (which itself was inspired by [7]) to track the usages of mulitiple resources. As already mentioned above, we can use techniques for higher-order model checking to refine our analysis.

7. Conclusion

We have proposed an automated method for verifying code generators written in a gensym language, based on higher-order model checking. Our method can check that all the generated code values are closed and well-typed, while maintaining the flexibility of programming in the gensym language. We have implemented a prototype verification tool based on the proposed method. We have also designed a two-stage programming language and a translation to the gensym language, so that our verification method can also be applied to two-stage programs.

References

[1] M. Berger and L. Tratt. Program logics for homogeneous generative run-time meta-programming. *Logical Methods in Computer Science*, 11(1), 2015.

[2] C. H. Broadbent and N. Kobayashi. Saturation-based model checking of higher-order recursion schemes. In *Proceedings of CSL 2013*, volume 23 of *LIPIcs*, pages 129–148, 2013.

[3] C. Calcagno, E. Moggi, and T. Sheard. Closed types for a safe imperative MetaML. *J. Funct. Program.*, 13(3):545–571, May 2003.

[4] C. Calcagno, W. Taha, L. Huang, and X. Leroy. Implementing multi-stage languages using ASTs, gensym, and reflection. In *International*

[2] One notable exception is Shields et al.[28], in which typechecking of generated code is deferred until the execution of the generated code starts.

Conference on Generative Programming and Component Engineering, pages 57–76, 2003.

[5] C. Chen and H. Xi. Meta-programming through typeful code representation. In *Proc. of ICFP*, pages 275–286, 2003.

[6] W. Choi, B. Aktemur, K. Yi, and M. Tatsuta. Static analysis of multi-staged programs via unstaging translation. In *ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*, pages 81–92, Jan. 2011.

[7] B. Cook, A. Gotsman, A. Podelski, A. Rybalchenko, and M. Y. Vardi. Proving that programs eventually do something good. In *Proc. of POPL*, pages 265–276. ACM Press, 2007.

[8] R. Davies. A temporal-logic approach to binding-time analysis. In *IEEE Symposium on Logic in Computer Science (LICS'96)*, pages 184–195, July 1996.

[9] R. Davies. A temporal-logic approach to binding-time analysis. In *Proceedings, 11th Annual IEEE Symposium on Logic in Computer Science, New Brunswick, New Jersey, USA, July 27-30, 1996*, pages 184–195. IEEE Computer Society, 1996.

[10] R. Davies and F. Pfenning. A modal analysis of staged computation. *JACM*, 48(3):555–604, 2001.

[11] S. Fogarty, E. Pasalic, J. Siek, and W. Taha. Concoqtion: Indexed types now! In *Proc.Proceedings of Workshop on Partial Evaluation and Program Manipulation (PEPM2007)*, pages 112–121, 2007.

[12] A. Igarashi and N. Kobayashi. Resource usage analysis. *ACM Trans. Prog. Lang. Syst.*, 27(2):264–313, 2005.

[13] I.-S. Kim, K. Yi, and C. Calcagno. A polymorphic modal type system for lisp-like multi-staged languages. In *ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages (POPL'06)*, pages 257–269, Jan. 2006.

[14] N. Kobayashi. Types and higher-order recursion schemes for verification of higher-order programs. In *Proc. of POPL*, pages 416–428. ACM Press, 2009.

[15] N. Kobayashi. Model checking higher-order programs. *Journal of the ACM*, 60(3), 2013.

[16] N. Kobayashi and A. Igarashi. Model checking higher-order programs with recursive types. In *Proceedings of ESOP 2013*, volume 7792 of *LNCS*. Springer, 2013.

[17] N. Kobayashi, R. Sato, and H. Unno. Predicate abstraction and CEGAR for higher-order model checking. In *Proc. of PLDI*, pages 222–233. ACM Press, 2011.

[18] N. Kobayashi, N. Tabuchi, and H. Unno. Higher-order multi-parameter tree transducers and recursion schemes for program verification. In *Proc. of POPL*, pages 495–508. ACM Press, 2010.

[19] Y. Minamide. Static approximation of dynamically generated WWW pages. In *Proc.International Conference on World Wide Web*, pages 432–441, 2005.

[20] A. Nanevski and F. Pfenning. Staged computation with names and necessity. *J. Funct. Program.*, 15(5):893–939, 2005.

[21] C.-H. L. Ong. On model-checking trees generated by higher-order recursion schemes. In *LICS 2006*, pages 81–90. IEEE Computer Society Press, 2006.

[22] C.-H. L. Ong and S. Ramsay. Verifying higher-order programs with pattern-matching algebraic data types. In *Proc. of POPL*, pages 587–598. ACM Press, 2011.

[23] B. C. Pierce. *Types and Programming Languages*. The MIT Press, 2002.

[24] S. Ramsay, R. Neatherway, and C.-H. L. Ong. An abstraction refinement approach to higher-order model checking. In *Proceedings of POPL 2014*, 2014.

[25] J. Rehof and P. Urzyczyn. Finite combinatory logic with intersection types. In *Typed Lambda Calculi and Applications - 10th International Conference, TLCA 2011, Novi Sad, Serbia, June 1-3, 2011. Proceedings*, volume 6690, pages 169–183. Springer, 2011.

[26] R. Sato, H. Unno, and N. Kobayashi. Towards a scalable software model checker for higher-order programs. In *Proceedings of PEPM 2013*, pages 53–62. ACM Press, 2013.

[27] T. Sheard and N. Linger. Programming in Ωmega. In *Central European Functional Programming School, Second Summer School, CEFP 2007*, pages 158–227, 2007.

[28] M. Shields, T. Sheard, and S. L. Peyton Jones. Dynamic typing as staged type inference. In *Proc.ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages (POPL'98)*, pages 289–302, 1998.

[29] W. Taha. A sound reduction semantics for untyped CBN multi-stage computation. Or, the theory of MetaML is non-trivial. In *Proc.Proceedings of Workshop on Partial Evaluation and Program Manipulation (PEPM2000)*, 2000.

[30] W. Taha, Z. E.-A. Benaissa, and T. Sheard. Multi-stage programming: Axiomatization and type-safety. In *Proc.25th International Colloquium on Automata, Languages and Programming (ICALP'98)*, volume 1443 of *LNCS*, pages 918–929, Aalborg, Denmark, July 1998. Springer.

[31] W. Taha and M. F. Nielsen. Environment classifiers. In *ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages (POPL'03)*, pages 26–37, 2003.

[32] W. Taha and T. Sheard. Multi-stage programming with explicit annotations. In *Proceedings of Workshop on Partial Evaluation and Program Manipulation (PEPM'97)*, pages 203–217, Amsterdam, Netherlands, 1997.

[33] T. Tsukada and A. Igarashi. A logical foundation for environment classifiers. *Logical Methods in Computer Science*, 6(4:8):1–43, Dec. 2010.

[34] H. Unno, N. Tabuchi, and N. Kobayashi. Verification of tree-processing programs via higher-order model checking. In *Proceedings of APLAS 2010*, volume 6461 of *LNCS*, pages 312–327. Springer, 2010.

[35] A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Info. Comput.*, 115(1):38–94, 1994.

[36] Y. Yuse and A. Igarashi. A modal type system for multi-level generating extensions with persistent code. In *Proc.8th ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP'06)*, pages 201–212, Venice, Italy, July 2006.

Appendix

Proof of Theorem 2

Lemma 23. *Let Γ be a type environment, $\hat{e}$ be a term, and $\mathcal{C}$ be an evaluation context. Then we have $\Gamma \vdash \mathcal{C}\big[\hat{e}\big] : \texttt{code}$ if and only if $\Gamma \vdash \hat{e} : \texttt{code}$.*

Proof. By straightforward induction on the structure of $\mathcal{C}$. ■

Lemma 24. *Let Γ be a type environment, $\hat{e}_\mathrm{A}$ be an applicative term, e_B be a run-time term, and τ_A, τ_B be types. If $\Gamma \vdash \hat{e}_\mathrm{A} : \tau_\mathrm{A}$ and $\Gamma[x \mapsto \tau_\mathrm{A}] \vdash e_\mathrm{B} : \tau_\mathrm{B}$, then $\Gamma \vdash [\hat{e}_\mathrm{A}/x]e_\mathrm{B} : \tau_\mathrm{B}$*

Proof. By straightforward induction on the derivation of $\Gamma[x \mapsto \tau_\mathrm{A}] \vdash e_\mathrm{B} : \tau_\mathrm{B}$. Intuitively we can deduce $\Gamma \vdash [\hat{e}_\mathrm{A}/x]e_\mathrm{B} : \tau_\mathrm{B}$ by replacing every leaf of the form $\Gamma[x \mapsto \tau_\mathrm{A}] \vdash x : \tau_\mathrm{A}$ in the derivation tree of $\Gamma[x \mapsto \tau_\mathrm{A}] \vdash e_\mathrm{B} : \tau_\mathrm{B}$ with the derivation tree of $\Gamma \vdash \hat{e}_\mathrm{A} : \tau_\mathrm{A}$. ■

Lemma 25. *If $\Gamma \vdash v : \texttt{code}$, then $\varnothing \vdash v : \texttt{code}$.*

Proof. By induction on $\Gamma \vdash v : \texttt{code}$. ■

Proof of Theorem 2. It is proved in two steps (namely, by showing Preservation and Progress [23, 35]). We first show that, if $\Gamma \vdash \hat{e} : \tau$ and $\hat{e} \hookrightarrow_\mathcal{D} \hat{e}'$, then $\Gamma \vdash \hat{e} : \tau$. Second, we show that if $\Gamma \vdash \hat{e} : \tau$, then either (i) $\hat{e}$ is a value and $\varnothing \vdash \hat{e} : \texttt{code}$; or (ii) there exists a run-time term $\hat{e}'$ such that $\hat{e} \hookrightarrow_\mathcal{D} \hat{e}'$.

The first property is proved by induction on the definition of $\hookrightarrow_\mathcal{D}$, it suffices for us to consider the following two cases:

- The case $(F\ x_1\ \cdots\ x_\ell = e) \in \mathcal{D}$, $\hat{e}_\mathrm{A} \equiv \mathcal{C}\big[F\ \hat{e}_1\ \cdots\ \hat{e}_\ell\big]$ and $\hat{e}_\mathrm{B} \equiv \mathcal{C}\big[[\hat{e}_\ell/x_\ell]\cdots[\hat{e}_1/x_1]e\big]$: since $(\mathcal{D}, S)$ is simply-typed by the definition of the gensym language, there exist types $\tau_1, \ldots, \tau_\ell$ such that $\Gamma[x_1 \mapsto \tau_1]\cdots[x_\ell \mapsto \tau_\ell] \vdash e : \texttt{code}$ and $\Gamma(F) = \tau_1 \to \cdots \to \tau_\ell \to \texttt{code}$. By Lemma 23 we have $\Gamma \vdash F\,\hat{e}_1\ \cdots\ \hat{e}_\ell : \texttt{code}$, and considering the derivation of it we have $\Gamma \vdash \hat{e}_i : \tau_i$ for each $i \in [\ell]$. Thus, by repeated application of Lemma 24, we have $\Gamma \vdash [\hat{e}_\ell/x_\ell]\cdots[\hat{e}_1/x_1]e : \texttt{code}$. Again by Lemma 23 we have $\Gamma \vdash \mathcal{C}\big[[\hat{e}_\ell/x_\ell]\cdots[\hat{e}_1/x_1]e\big] : \texttt{code}$.
- The case $\hat{e}_\mathrm{A} \equiv \mathcal{C}\big[\mathbf{gensym/cont}\ \hat{e}\big]$ and $\hat{e}_\mathrm{B} \equiv \mathcal{C}\big[\hat{e}\ \alpha\big]$: by Lemma 23 we have $\Gamma \vdash \mathbf{gensym/cont}\ \hat{e} : \texttt{code}$, and considering the derivation for it $\Gamma \vdash \hat{e} : \texttt{code} \to \texttt{code}$ holds. Therefore, we can derive $\Gamma \vdash \hat{e}\ \alpha : \texttt{code}$, and by Lemma 23 we have $\Gamma \vdash \mathcal{C}\big[\hat{e}\ \alpha\big] : \texttt{code}$.

The second property is shown by induction on $\Gamma \vdash \hat{e} : \texttt{code}$, using Lemma 25. ■

Proof of Theorem 22

Proof. We show (1) and (2) at the same time by induction on the structure of M.

- The case where $M \equiv x$ is trivial.
- The case $M \equiv c(M_1, \ldots, M_m)$: assume $\Gamma \vdash^0 M : \tau$. By the derivation rules we have $\mathcal{C}(c) = \tau_1 \to \cdots \to \tau_m \to \tau$ and $\Gamma \vdash^0 M_i : \tau_i$ for each $i \in [m]$. Then by IH we have $\lfloor\Gamma\rfloor \vdash_\mathrm{m} \lfloor M_i\rfloor^0_{\Psi_\Gamma} : \tau$ for each $i \in [m]$, and by $\lfloor\Gamma\rfloor \vdash_\mathrm{m} c : \mathcal{C}(c)$ we can derive $\lfloor\Gamma\rfloor \vdash_\mathrm{m} c\ \lfloor M_1\rfloor^0_{\Psi_\Gamma}\ \cdots\ \lfloor M_m\rfloor^0_{\Psi_\Gamma} : \tau$. On the other hand, assume $\Gamma \vdash^1 M : \bigcirc$, then by the derivation rules we have $\Gamma \vdash^1 M_i : \bigcirc$ for each $i \in [m]$, and by IH we have $\lfloor\Gamma\rfloor \vdash_\mathrm{m} \lfloor M_i\rfloor^1_{\Psi_\Gamma} : \texttt{code}$ for each $i \in [m]$. By the assumption $m = \mathcal{P}_\mathcal{C}(\texttt{'c})$ and $\lfloor\Gamma\rfloor \vdash_\mathrm{m} \texttt{'c} : \underbrace{\texttt{code} \to \cdots \to \texttt{code}}_{m} \to \texttt{code}$, we can derive $\lfloor\Gamma\rfloor \vdash_\mathrm{m} \texttt{'c}\ \lfloor M_1\rfloor^1_{\Psi_\Gamma}\ \cdots\ \lfloor M_m\rfloor^1_{\Psi_\Gamma} : \texttt{code}$.
- The case $M \equiv M_1\ M_2$: assume $\Gamma \vdash^0 M_1\ M_2 : \tau$ holds. By the derivation rules we have $\Gamma \vdash^0 M_1 : \tau_1 \to \tau$ and $\Gamma \vdash^0 M_2 : \tau_1$ for some τ_1. Then by IH we can derive:

$$\frac{\lfloor\Gamma\rfloor \vdash_\mathrm{m} \lfloor M_1\rfloor^0_{\Psi_\Gamma} : \tau_1 \to \tau \qquad \lfloor\Gamma\rfloor \vdash_\mathrm{m} \lfloor M_2\rfloor^0_{\Psi_\Gamma} : \tau_1}{\lfloor\Gamma\rfloor \vdash_\mathrm{m} \lfloor M\rfloor^0_{\Psi_\Gamma} : \tau}\ .$$

  On the other hand, assume $\Gamma \vdash^1 M_1\ M_2 : \bigcirc$. By the derivation rules we have $\Gamma \vdash^1 M_1 : \bigcirc$ and $\Gamma \vdash^1 M_2 : \bigcirc$, and by IH we can derive:

$$\frac{\dfrac{\lfloor\Gamma\rfloor \vdash_\mathrm{m} \texttt{APP} : \texttt{code} \to \texttt{code} \to \texttt{code} \qquad \lfloor\Gamma\rfloor \vdash_\mathrm{m} \lfloor M_1\rfloor^1_{\Psi_\Gamma} : \texttt{code}}{\lfloor\Gamma\rfloor \vdash_\mathrm{m} \texttt{APP}\ \lfloor M_1\rfloor^1_{\Psi_\Gamma} : \texttt{code} \to \texttt{code}} \qquad \lfloor\Gamma\rfloor \vdash_\mathrm{m} \lfloor M_2\rfloor^1_{\Psi_\Gamma} : \texttt{code}}{\lfloor\Gamma\rfloor \vdash_\mathrm{m} \texttt{APP}\ \lfloor M_1\rfloor^1_{\Psi_\Gamma} \lfloor M_2\rfloor^1_{\Psi_\Gamma} : \texttt{code}}$$

  i.e. we have $\lfloor\Gamma\rfloor \vdash_\mathrm{m} \lfloor M\rfloor^1_{\Psi_\Gamma} : \texttt{code}$.
- The case $M \equiv \boldsymbol{\lambda} x.\ M'$: assume $\Gamma \vdash^0 \boldsymbol{\lambda} x.\ M' : \tau$ holds. By the derivation rules, τ is of the form $\tau_1 \to \tau_2$ and satisfies $\Gamma[x \mapsto \tau_1] \vdash^0 M' : \tau_2$. Since $\lfloor\Gamma[x \mapsto \tau_1]\rfloor = \lfloor\Gamma\rfloor[x \mapsto \tau_1]$ and $\Psi_{\Gamma[x\mapsto\tau_1]} = \Psi_\Gamma[x \mapsto 0]$, by IH we can derive:

$$\frac{\lfloor\Gamma\rfloor[x \mapsto \tau_1] \vdash_\mathrm{m} \lfloor M'\rfloor^0_{\Psi_\Gamma[x\mapsto 0]} : \tau_2}{\lfloor\Gamma\rfloor \vdash_\mathrm{m} \boldsymbol{\lambda} x.\ \lfloor M'\rfloor^0_{\Psi_\Gamma} : \tau_1 \to \tau_2}$$

  i.e. we have $\lfloor\Gamma\rfloor \vdash_\mathrm{m} \lfloor M\rfloor^0_{\Psi_\Gamma} : \tau$. On the other hand, if we have $\Gamma \vdash^1 \boldsymbol{\lambda} x.\ M' : \bigcirc$, by the derivation rules $\Gamma[x \mapsto \bigcirc] \vdash^1 M' : \bigcirc$ holds. Since we have $\lfloor\Gamma[x \mapsto \bigcirc]\rfloor = \lfloor\Gamma\rfloor[x \mapsto \texttt{code}]$ and $\Psi_{\Gamma[x\mapsto\bigcirc]} = \Psi_\Gamma[x \mapsto 1]$, by IH $\lfloor\Gamma\rfloor \vdash_\mathrm{m} \lfloor M\rfloor^1_{\Psi_\Gamma} : \texttt{code}$ can be derived; its derivation is shown in Figure 7.
- The case $M \equiv \mathbf{fix}\ x.\ M'$: if we have $\Gamma \vdash^0 \mathbf{fix}\ x.\ M' : \tau$, then by the derivation rules $\Gamma[x \mapsto \tau] \vdash^0 M' : \tau$ holds. Similarly to the previous case, by IH we can derive:

$$\frac{\lfloor\Gamma\rfloor[x \mapsto \tau] \vdash_\mathrm{m} \lfloor M'\rfloor^0_{\Psi_\Gamma[x\mapsto 0]} : \tau}{\lfloor\Gamma\rfloor \vdash_\mathrm{m} \mathbf{fix}\ x.\ \lfloor M'\rfloor^0_{\Psi_\Gamma} : \tau}\ .$$

  The subcase for $\Gamma \vdash^1 \mathbf{fix}\ x.\ M' : \tau$ is also similar to the previous case.
- The case where $M \equiv \mathbf{if}\ M_0\ \mathbf{then}\ M_1\ \mathbf{else}\ M_2$ is easy.
- The case $M \equiv \langle\!\langle M' \rangle\!\rangle$: assume $\Gamma \vdash^0 \langle\!\langle M' \rangle\!\rangle : \tau$. Then $\tau \equiv \texttt{code}$ and $\Gamma \vdash^1 M' : \bigcirc$ hold by the derivation rules, and by IH we have $\lfloor\Gamma\rfloor \vdash_\mathrm{m} \lfloor M'\rfloor^1_{\Psi_\Gamma} : \texttt{code}$. Since $\lfloor M'\rfloor^1_{\Psi_\Gamma} \equiv \lfloor\langle\!\langle M' \rangle\!\rangle\rfloor^0_{\Psi_\Gamma} \equiv \lfloor M\rfloor^0_{\Psi_\Gamma}$, we have $\lfloor\Gamma\rfloor \vdash_\mathrm{m} \lfloor M\rfloor^0_{\Psi_\Gamma} : \texttt{code}$.
- The case $M \equiv \tilde{}M'$: assume $\Gamma \vdash^1 \tilde{}M' : \bigcirc$. Then $\Gamma \vdash^0 M' : \texttt{code}$ holds by the derivation rules, and by IH we have $\lfloor\Gamma\rfloor \vdash_\mathrm{m} \lfloor M'\rfloor^0_{\Psi_\Gamma} : \texttt{code}$. Since $\lfloor M'\rfloor^0_{\Psi_\Gamma} \equiv \lfloor\tilde{}M'\rfloor^1_{\Psi_\Gamma} \equiv \lfloor M\rfloor^1_{\Psi_\Gamma}$, we have $\lfloor\Gamma\rfloor \vdash_\mathrm{m} \lfloor M\rfloor^1_{\Psi_\Gamma} : \texttt{code}$.

■

The definition of the automaton $\mathcal{A}_\mathrm{cls}$

The automaton $\mathcal{A}_\mathrm{cls} = \big(\Sigma_\mathrm{cls}, Q_\mathrm{cls}, \Delta_\mathrm{cls}, q^\mathrm{INI}_\mathrm{cls}\big)$ for the well-formedness and closedness check is defined as follows.

$$Q_\mathrm{cls} := \big\{q_0, q_1, q^\mathrm{var}, q^\mathrm{ig}\big\},$$

$$q^\mathrm{INI}_\mathrm{cls} := q_0,$$

$$\Delta_\mathrm{cls} := \{ \qquad\qquad q_1\ \texttt{var} \to \varepsilon,$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\lfloor \Gamma \rfloor[x \mapsto \texttt{code}] \vdash_{\mathrm{m}} \texttt{ABS} : \texttt{code} \to \texttt{code} \to \texttt{code} \quad \lfloor \Gamma \rfloor[x \mapsto \texttt{code}] \vdash_{\mathrm{m}} x : \texttt{code}}{\lfloor \Gamma \rfloor[x \mapsto \texttt{code}] \vdash_{\mathrm{m}} \texttt{ABS}\ x : \texttt{code} \to \texttt{code}} \quad \lfloor \Gamma \rfloor[x \mapsto \texttt{code}] \vdash_{\mathrm{m}} \lfloor M' \rfloor^1_{\Psi_\Gamma[x \mapsto 1]} : \texttt{code}}{\lfloor \Gamma \rfloor[x \mapsto \texttt{code}] \vdash_{\mathrm{m}} \texttt{ABS}\ x\ \lfloor M' \rfloor^1_{\Psi_\Gamma[x \mapsto 1]} : \texttt{code}}}{\lfloor \Gamma \rfloor \vdash_{\mathrm{m}} \boldsymbol{\lambda} x.\ \texttt{ABS}\ x\ \lfloor M' \rfloor^1_{\Psi_\Gamma} : \texttt{code} \to \texttt{code}} \quad \lfloor \Gamma \rfloor \vdash_{\mathrm{m}} \mathbf{gensym}\ () : \texttt{code}}{\lfloor \Gamma \rfloor \vdash_{\mathrm{m}} (\boldsymbol{\lambda} x.\ \texttt{ABS}\ x\ \lfloor M' \rfloor^1_{\Psi_\Gamma})\ \mathbf{gensym}\ () : \texttt{code}}$$

Figure 1. The derivation for $\lfloor \Gamma \rfloor \vdash_{\mathrm{m}} \lfloor \boldsymbol{\lambda} x.\ M' \rfloor^1_{\Psi_\Gamma} : \texttt{code}$ in the proof of Theorem 22

$$\begin{aligned}
&q_0\ \texttt{ig} \to \varepsilon, && q_1\ \texttt{ig} \to \varepsilon,\\
&q_0\ \texttt{ABS} \to q^{\mathrm{var}}\ q_1, && q_1\ \texttt{ABS} \to q^{\mathrm{var}}\ q_1,\\
&q_0\ \texttt{ABS} \to q^{\mathrm{ig}}\ q_0, && q_1\ \texttt{ABS} \to q^{\mathrm{ig}}\ q_1,\\
&q_0\ \texttt{FIX} \to q^{\mathrm{var}}\ q_1, && q_1\ \texttt{FIX} \to q^{\mathrm{var}}\ q_1,\\
&q_0\ \texttt{FIX} \to q^{\mathrm{ig}}\ q_0, && q_1\ \texttt{FIX} \to q^{\mathrm{ig}}\ q_1\}\\
\cup\ &\{q^{\mathrm{ig}}\ \texttt{ig} \to \varepsilon, && q^{\mathrm{var}}\ \texttt{var} \to \varepsilon\}\\
\cup\ &\Big\{q_0\ P \to \underbrace{q_0\ \cdots\ q_0}_{\mathcal{P}(P)} \Big|\ P \in (\operatorname{dom}\mathcal{P}) \setminus \{\texttt{ABS}, \texttt{FIX}\}\Big\}\\
\cup\ &\Big\{q_1\ P \to \underbrace{q_1\ \cdots\ q_1}_{\mathcal{P}(P)} \Big|\ P \in (\operatorname{dom}\mathcal{P}) \setminus \{\texttt{ABS}, \texttt{FIX}\}\Big\}.
\end{aligned}$$

The state q_1 means that `var` is currently not bound and q_0 means it is bound. The states q^{var} and q^{ig} only accept `var` and `ig`, respectively.

The definition of the automaton $\mathcal{A}^{\mathcal{T}}_{\mathrm{wt}}$

The automaton $\mathcal{A}^{\mathcal{T}}_{\mathrm{wt}} = \left(\Sigma^{\mathcal{T}}_{\mathrm{wt}}, Q^{\mathcal{T}}_{\mathrm{wt}}, \Delta^{\mathcal{T}}_{\mathrm{wt}}, q^{\mathrm{INI}}_{\mathrm{wt}}\right)$ for the well-typedness check is defined as follows.

$$\begin{aligned}
Q^{\mathcal{T}}_{\mathrm{wt}} &\coloneqq \{q_T \mid T \in \mathcal{T}\} \cup \{q^{\mathrm{any}}, q^{\mathrm{some}}\},\\
q^{\mathrm{INI}}_{\mathrm{wt}} &\coloneqq q^{\mathrm{some}},\\
\Delta^{\mathcal{T}}_{\mathrm{wt}} &\coloneqq \{q_T\ \texttt{var}_T \to \varepsilon \mid T \in \mathcal{T}\}\\
&\quad \cup \{q_T\ \texttt{oneof} \to \textsc{Array}(i, q_T) \mid i \in [\#\mathcal{T}]\ \wedge\ T \in \mathcal{T}\}\\
&\quad \cup \{q_{T_1 \to T_2}\ \texttt{ABS} \to q_{T_1}\ q_{T_2} \mid (T_1 \to T_2) \in \mathcal{T}\}\\
&\quad \cup \{q_T\ \texttt{FIX} \to q_T\ q_T \mid T \in \mathcal{T}\}\\
&\quad \cup \left\{ q_T\ P \to q_{T_1}\ \cdots\ q_{T_{\mathcal{P}(P)}} \;\middle|\; \begin{array}{l} (P, T_1 \to \cdots \to T_{\mathcal{P}(P)} \to T) \in \mathcal{P}^{\mathrm{type}} \\ \{T_1, \ldots, T_{\mathcal{P}(P)}, T\} \subseteq \mathcal{T} \end{array} \right\}\\
&\quad \cup \{q^{\mathrm{any}}\ \texttt{var}_T \to \varepsilon \mid T \in \mathcal{T}\}\\
&\quad \cup \{q^{\mathrm{any}}\ \texttt{oneof} \to \underbrace{q^{\mathrm{any}}\ \cdots\ q^{\mathrm{any}}}_{\#\mathcal{T}}\}\\
&\quad \cup \{q^{\mathrm{any}}\ P \to \underbrace{q^{\mathrm{any}}\ \cdots\ q^{\mathrm{any}}}_{\mathcal{P}(P)} \mid P \in \operatorname{dom}\mathcal{P}\}\\
&\quad \cup \{q^{\mathrm{some}}\ \texttt{var}_T \to \varepsilon \mid T \in \mathcal{T}\}\\
&\quad \cup \{q^{\mathrm{some}}\ \texttt{oneof} \to \textsc{Array}(i, q^{\mathrm{some}}) \mid i \in [\#\mathcal{T}]\}\\
&\quad \cup \{q^{\mathrm{some}}\ \texttt{ABS} \to q_{T_1}\ q_{T_2} \mid (T_1 \to T_2) \in \mathcal{T}\}\\
&\quad \cup \{q^{\mathrm{some}}\ \texttt{FIX} \to q_T\ q_T \mid T \in \mathcal{T}\}\\
&\quad \cup \left\{ q^{\mathrm{some}}\ P \to q_{T_1}\ \cdots\ q_{T_{\mathcal{P}(P)}} \;\middle|\; \begin{array}{l} T_1 \to \cdots \to T_{\mathcal{P}(P)} \to T \in \mathcal{P}^{\mathrm{type}}(P) \\ \{T_1, \ldots, T_{\mathcal{P}(P)}, T\} \subseteq \mathcal{T} \end{array} \right\}
\end{aligned}$$

Here $\textsc{Array}(i, q)$ is a sequence of length $\#\mathcal{T}$ consisting of q^{any} except for q at the i-th position. The sate q^{any} accepts all trees and q^{some} accepts a tree if it is accepted by q_T for some $T \in \mathcal{T}$.