World Age in Julia

Optimizing Method Dispatch in the Presence of Eval

JULIA BELYAKOVA, Northeastern University, USA
BENJAMIN CHUNG, Northeastern University, USA
JACK GELINAS, Northeastern University, USA
JAMESON NASH, Julia Computing, USA
ROSS TATE, Cornell University, USA
JAN VITEK, Northeastern University, USA and Czech Technical University in Prague, Czech Republic

Dynamic programming languages face semantic and performance challenges in the presence of features, such as `eval`, that can inject new code into a running program. The Julia programming language introduces the novel concept of world age to insulate optimized code from one of the most disruptive side-effects of `eval`: changes to the definition of an existing function. This paper provides the first formal semantics of world age in a core calculus named Juliette, and shows how world age enables compiler optimizations, such as inlining, in the presence of `eval`. While Julia also provides programmers with the means to bypass world age, we found that this mechanism is not used extensively: a static analysis of over 4,000 registered Julia packages shows that only 4–9% of packages bypass world age. This suggests that Julia's semantics aligns with programmer expectations.

CCS Concepts: • **Software and its engineering → Language features**; *General programming languages.*

Additional Key Words and Phrases: eval, method dispatch, compilation, dynamic languages

ACM Reference Format:
Julia Belyakova, Benjamin Chung, Jack Gelinas, Jameson Nash, Ross Tate, and Jan Vitek. 2020. World Age in Julia: Optimizing Method Dispatch in the Presence of Eval. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 207 (November 2020), 26 pages. https://doi.org/10.1145/3428275

1 INTRODUCTION

The Julia programming language [Bezanson et al. 2017] aims to decrease the gap between productivity and performance languages in scientific computing. While Julia provides productivity features such as dynamic types, optional type annotations, reflection, garbage collection, symmetric multiple dispatch, and dynamic code loading, its designers carefully arranged those features to allow for heavy compiler optimization. The key to performance lies in the synergy between language design, language-implementation techniques, and programming style [Bezanson et al. 2018].

The goal of this paper is to shed light on one particular design challenge: how to support `eval` that enables dynamic code loading, *and* achieve good performance. The `eval` construct comes from Lisp [McCarthy 1978] and is found in most dynamic languages, but its expressive power varies from one language to another. Usually `eval` takes a string or a syntax tree as an argument and executes

Authors' addresses: Julia Belyakova, Northeastern University, USA; Benjamin Chung, Northeastern University, USA; Jack Gelinas, Northeastern University, USA; Jameson Nash, Julia Computing, USA; Ross Tate, Cornell University, USA; Jan Vitek, Northeastern University, USA, Czech Technical University in Prague, Czech Republic.

2475-1421/2020/11-ART207
https://doi.org/10.1145/3428275

```
> x = 3.14
> f(x) = (
    eval(:(x = 0));
    x * 2)

> f(42) # 84
> x     # 0
```

Fig. 1. Scope of eval in Julia

```
> (defn g [] 2)
> (defn f [x]
    (eval `(defn g [] ~x))
    (* x (g)))
> (f 42) ; 1764
> (g)    ; 42
> (f 42) ; 1764
```

Fig. 2. Eval in Clojure

```
> g() = 2
> f(x) = (
    eval(:(g()=$x));
    x * g())
> f(42) # 84
> g()   # 42
> f(42) # 1764
```

Fig. 3. Eval in Julia

it in some environment. In JavaScript and R, `eval` may execute in the current lexical environment; in Lisp and Clojure, it is limited to the "top level". On this spectrum, Julia takes the latter approach, which enables compiler optimizations that would otherwise be unsound. For example, in a program in Fig. 1, multiplication `x*2` in the body of function `f` can be safely optimized to an efficient integer multiplication for the call `f(42)`. This is because `eval` only accesses the top-level environment and thus cannot change the value of a local parameter `x`, which is known to be the integer 42. For a global `x`, such an optimization would be unsound.

What is unique about the design of `eval` in Julia is the treatment of function definitions. Many compilers rely on the information about functions for optimizations, but those optimizations can be jeopardized by the presence of `eval`. To explain how Julia handles the interaction of `eval` and functions, we contrast it with the Clojure language. Fig. 2 shows a Clojure program with a call to a function `f` which, within its body, updates function `g` by invoking `eval`. Then, the call to `f` returns 1764 because the new definition of `g` is used. Fig. 3 shows a Julia equivalent of the same program. Here, the second call to `f` returns 1764 just like in Clojure, but *the first call returns 84*. This is because, while the first invocation of `f` is running, it does not see the redefinition of `g` made by `eval`: the redefinition becomes visible only after the first call (to `f(42)`) returns to the top level. From the compiler's point of view, this means that calling `eval` does not force recompilation of any methods that are "in-flight." Thus, it is safe to devirtualize, specialize, and inline functions in the presence of `eval` without the need for deoptimization. For example, `x*g()` can be safely replaced with `x*2` in `f` for the first call `f(42)`.

Julia made the choice to restrict access to newly defined methods due to pressing performance concerns. Julia heavily relies on symmetric multiple dispatch [Bobrow et al. 1986], which allows a function to have multiple implementations, called methods, distinguished by their parameter type annotations. At run time, a call is dispatched to the most specific method applicable to the types of its arguments. While some functions might have only one method, plenty have dozens or even hundreds of them. For example, the multiplication function alone has 357 standard methods (see an excerpt in Fig. 4). If Julia were to always use generic method invocation to dispatch `*`, programs would become unbearably slow. By constraining `eval`, the compiler can avoid generic invocations. In Fig. 1, the compiler can pick and inline the right definition of `*` when compiling `f(42)`. It should be noted that this optimization-friendly `eval` semantics does not apply to data. Function definitions are treated differently from variables, and changes to global variables (such as in Fig. 1) can be observed immediately.

```
*(x::Int,     y::Int)     = mul_int(x, y)
*(x::Float64, y::Float64) = mul_float(x,y)
*(a::Number,  b::AbstractVector) = ...
*(x::Bool,    y::Bool)    = x & y
```

Fig. 4. Multiple definitions of *

Arguably, despite being unusual, Julia's semantics is easy to understand for programmers. There is always a clear point where new definitions become visible—at the top level—and thus, users can avoid surprises and dependence on the exact position of `eval` in the code. However, in case the default semantics is not desirable, Julia also provides an escape hatch: the built-in function `invokelatest(f)`, which forces the implementation to invoke the most recent definition of `f`. A slower alternative to `invokelatest` is to call `f` within `eval`, which always executes in the top level.

This language mechanism that delays the effect of `eval` on function definitions is called *world age*. In the Julia documentation, world age is described operationally [Bezanson et al. 2018]: every method defined in a program is associated with an age, and for each function call, Julia ensures that the current age is larger than the age of the method about to be invoked. One can think of the world age as a counter that allows the implementation to ignore all methods that were born after the last top-level call started. Much of its specification is tied to implementation details and efficiency considerations. Our contributions are as follows:

- *A core calculus for world age:* We introduce Juliette, a calculus that models the notion of world age abstractly. In the calculus, the implementation-oriented world-age counters are replaced with method tables that are explicitly copied at the top level, and `eval` is simplified down to an operation that evaluates its argument in a specific method table.
- *Formalization of optimizations:* We formalize and prove correct three compiler optimizations, namely inlining, devirtualization, and specialization.
- *Corpus analysis:* We analyze Julia packages to understand how `eval` is used, and estimate the potential impact of world age on library code. We also identify a number of programming patterns by manual inspection of selected packages.
- *Testing the semantics:* We develop a Redex model of our calculus and optimizations to allow rapid experimentation and testing.

The corpus analysis and the Redex model are publicly available.[1] The formalization with detailed proofs can be found in the extended version of the paper [Belyakova et al. 2020].

2 BACKGROUND

We start with an overview of the features of Julia relevant to our work, and review related work.

2.1 Julia Overview

Despite the extensive use of types and type annotations for dispatch and compiler optimizations, Julia is not statically typed. A formalization of types and subtyping is provided by Zappa Nardelli et al. [2018], and a general introduction to the language is given by Bezanson et al. [2017].

Values. Values are either instances of *primitive types*—sequences of bits—or *composite types*—collection of fields holding values. Every value has a concrete type (or tag). This tag is either inferred statically or stored in the boxed value. Tags are used to resolve multiple dispatch semantically and can be queried with typeof.

Types. Programmers can declare three kinds of user-defined types: *abstract types*, *primitive types*, and *composite types*. Abstract types cannot be instantiated, while concrete types can. For example, `Float64` is concrete, and is a subtype of abstract type `Number`. Concrete types have no subtypes. Additionally, user-defined type constructors can have bounded type parameters and can declare up to a single supertype.

[1] https://github.com/julbinb/juliette-wa

Annotations. Type annotations include a number of built-in type constructors, such as union and tuple types. Tuple types, written `Tuple{A,...}`, describe immutable values that have a special role in the language: every method takes a single tuple argument. The `::` operator ascribes a type to a definition. We will use τ to denote annotations.

Subtyping. The subtyping relation, `<:`, is used in run-time casts and multiple dispatch. Julia combines nominal subtyping, union types, iterated union types, covariant and invariant constructors, and singleton types. Tuple types are covariant in their parameters, so, for instance, `Tuple{Float64,Float64}` is a subtype of `Tuple{Number,Number}`.

Multiple dispatch. A function can have multiple methods where each method declares what argument types it can handle; an unspecified type defaults to `Any`. At run time, dispatching a call `f(v)` amounts to picking the best applicable method from all the methods of function `f`. For this, the dispatch mechanism first filters out methods whose type annotations τ are not a supertype of the type tag of `v`. Then it takes the method whose type annotation τ_i is the most specific of the remaining ones. If the set of applicable methods is empty, or there is no single best method, a run-time error is raised.

Reflection. Julia provides a number of built-in functions for run-time introspection and meta-programming. For instance, the methods of any function `f` may be listed using `methods(f)`. All the methods are stored in a special data structure, called the *method table*. It is possible to search the method table for methods accepting a given type: for instance, `methods(*,(Int,Float64))` will show methods of `*` that accept an integer-float pair. The `eval` function takes an expression object and evaluates it in the global environment of a specified module. For example, `eval(:(1+2))` will take the expression `:(1+2)` and return `3`.
`Eval` is frequently used for meta-programming as part of code generation. For example, Fig. 5 generalizes some of the basic binary operators to three arguments, generating four new methods. Instead of building expressions explicitly, one can also invoke the parser on a string. For instance, `eval(Meta.parse("id(x) = x"))` creates an identity method.

```
for op in (:+, :*, :&, :|)
  eval(:($op(a,b,c) = $op($op(a,b),c)))
end
```

Fig. 5. Code generation

2.2 Related Work

This paper is concerned with controlling the visibility of function definitions. Most programming languages control *where* definitions are visible, as part of their scoping mechanisms. Controlling *when* function definitions become visible is less common.

Languages with an interactive development environment had to deal with the addition of new definitions for functions from the start [McCarthy 1978]. Originally, these languages were interpreted. In that setting, allowing new functions to become visible immediately was both easy to implement and did not incur any performance overhead.

Just-in-time compilation changed the performance landscape, allowing dynamic languages to have competitive performance. However, this meant that to generate efficient code, compilers had to commit to particular versions of functions. If any function is redefined, all code that depends on that function must be recompiled; furthermore, any function currently executing has to be deoptimized using mechanisms such as on-stack-replacement [Hölzle et al. 1992]. The drawback of deoptimization is that it makes the compiler more complex and hinders some optimizations. For

example, a special `assume` instruction is introduced as a barrier to optimizations by Flückiger et al. [2018], who formalized the speculation and deoptimization happening in a model compiler.

Java allows for dynamic loading of new classes and provides sophisticated controls for where those classes are visible. This is done by the class-loading framework that is part of the virtual machine [Liang and Bracha 1998]. Much research happened in that context to allow the Java compiler to optimize code in the presence of dynamic loading. Detlefs and Agesen [1999] describe a technique, which they call preexistence, that can devirtualize a method call when the receiver object predates the introduction of a new class. Further research looked at performing dependency analysis to identify which methods are affected by the newly added definitions, to be then recompiled on demand [Nguyen and Xue 2005]. Glew [2005] describes a type-safe means of inlining and devirtualization: when newly loaded code is reachable from previously optimized code, these optimizations must be rechecked.

Controlling *when* definitions take effect is important in dynamic software updating, where running systems are updated with new code [Cook and Lee 1983]. Stoyle et al. [2007] introduce a calculus for reasoning about representation-consistent dynamic software updating in C-like languages. One of the key elements for their result is the presence of an `update` instruction that specifies when an update is allowed to happen. This has similarities to the world-age mechanism described here.

Substantial amounts of effort have been put into building calculi that support `eval` and similar constructs. For example, Politz et al. [2012] described the ECMAScript 5.1 semantics for `eval`, among other features. Glew [2005] formalized dynamic class loading in the framework of Featherweight Java, and Matthews and Findler [2008] developed a calculus for `eval` in Scheme. These works formalize the semantics of dynamically modifiable code in their respective languages, but, unlike Julia, the languages formalized do not have features explicitly designed to support efficient implementation.

3 WORLD AGE IN JULIA

The world-age mechanism in Julia limits the set of methods that can be invoked from a given call site. World age fixes the set of method definitions reachable from the currently executing method, isolating it from dynamically generated ones. In turn, this allows the compiler to optimize code without need for deoptimization, and limits the number of required synchronization points in a multi-threaded program. If full access to methods is required, however, Julia provides escape hatches to bypass world age by sacrificing performance.

3.1 Defining World Age

The primary goal of the world-age mechanism is to align the language's semantics with the assumptions made by the Julia just-in-time compiler's optimizations. Semantically, newly added methods (i.e. ones defined using `eval`) only become visible when execution returns to the top level, and the set of callable methods for an execution is fixed when it leaves the top level. Compilation of methods is triggered—only at the top level—when one of the following holds: (1) a function is called with previously unobserved types of arguments, or (2) a previously compiled function needs to be recompiled due to a change in its own definition or one of its dependencies. Since the set of visible methods gets fixed at a top-level call, and compilation only occurs from the top level, the compiler may assume that the currently known set of methods is complete and can optimize accordingly.

For performance reasons, the world-age mechanism is implemented by a simple monotonic counter. The counter is incremented every time a method is defined, and its value becomes the method's "birth age". Every method also can a store a "death age" (that is initially infinity), which is set when it is replaced or deleted. Methods with their ages are stored in a global data structure

```
f() = ... # t1
g() = ... # t2
g() = ... # t3
```

$$f \mapsto [t_1, \infty]$$
$$g_1 \mapsto [t_2, t_3]$$
$$g_2 \mapsto [t_3, \infty]$$

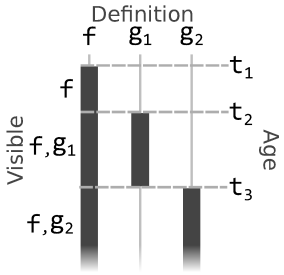

Fig. 6. Age ranges

```
> function ntl()
    eval(:(f() = 1))
    f()
  end
> ntl()
ERROR: MethodError: no method matching f()
The applicable method may be too new:
running in world age 26806, while current
world is 26807. Closest candidates are:
  f() at REPL[10]:2
```

Fig. 7. Age error

called a method table. The birth and death ages of a method determine the minimum and maximum world age from which the method can be invoked. This is illustrated in Fig. 6. Here, we define functions `f` and `g`, where `f` has only one method, whereas `g` has two methods, one replacing the other. Let us observe what definitions are visible at each step. Since `f` is defined once at time t_1 (i.e. when the world-age counter is equal to t_1) and never redefined, it has a birth age of t_1 and a death age of ∞; thus, it can be used anytime after t_1 and forevermore. In contrast, method g_1, created at time t_2, is redefined at time t_3, so its birth age is t_2 and death age is t_3; therefore, g_1 can be called from world age t where $t_2 \le t \le t_3$. Finally, g_2 is never redefined, so its age ranges from t_3 to ∞.

This language design can be restrictive. The easiest way to run afoul of world age is to attempt to define a method with `eval` and call it immediately thereafter. In Fig. 7, function `ntl` creates a new function `f` using `eval` and attempts to call it immediately without returning to the top level. Let us dissect the example and its error message. Function `ntl` was invoked from the top level with age of `26806`, thus limiting the set of visible methods to the ones born by this time. Then, `ntl` used `eval` to define `f`, giving it a birth age of `26807`. Finally, to call `f`, `ntl` needs a method of `f` that was born by `26806`, but none exists. Since the only method of `f` was created at `26807`, a `MethodError` is raised indicating that no method was found.

3.2 Breaking the Age Barrier

There are situations in which the world-age mechanism is too restrictive: for example, when a program wishes to programmatically generate code and then use it immediately. To accommodate these circumstances, Julia provides two ways for programmers to execute code ahead of its birth age. The first is `eval` itself, which executes its arguments as if they were at the top level, thus allowing any existing method to be called. However, `eval` needs to interpret arbitrary ASTs and is rather slow. Luckily, in many circumstances, the program only wishes to bypass the world-age restriction for a single function call. For this, one can use `invokelatest`, a built-in function that calls its argument from the latest world age. While substantially slower than a normal call, `invokelatest` is faster than `eval`. Moreover, `invokelatest` is passed arguments directly from the calling context so that values do not need to be inserted into `eval`'s AST. Both `eval` and `invokelatest` can be used to amend the example shown in Fig. 7 to call `f` in the latest world age. If we replace the bare call to `f` with a call to `eval(:(f()))`, then the call to `ntl` will produce 1. Similarly, `Base.invokelatest(f)` will get the same result.

Using either of these mechanisms, programmers can opt out from the limitation imposed by world age, but this comes with performance implications. Since neither `invokelatest` nor `eval` can be optimized, and both can kick off additional JIT compilation, they can have substantial performance impact. However, this impact is limited to only these explicitly impacted call sites.

As a result, programmers can carefully design their programs to minimize the number of broken barriers, thus minimizing the performance impact of the dynamism.

3.3 World Age in Practice

We have argued that world age is useful for performance of Julia programs, but does its semantics match programmers' expectations? We propose a slightly indirect answer to this, analyzing a corpus of programs and observing how often they use the two escape hatches mentioned above. Of those, `invokelatest` is the clearest indicator, as there is no reason to use it except to bypass the world age. Similarly, `eval`'ing a function call means evaluating that call in the latest world age, thereby allowing it to see the latest method definitions. The `eval` indicator is imprecise, however, as there are uses of `eval` that are not impacted by world age.

We take as our corpus all 4,011 registered Julia packages as of August 2020. The results of statically analyzing the code base are shown in Fig. 8. The analysis shows that 2,846 of the 4,011 packages used neither `eval` nor `invokelatest`, and thus are definitely age agnostic. Of the remaining packages, 1,094 used `eval` only, and so *could* be impacted. 15 packages used `invokelatest` only, and some 56 used both. We can reasonably presume that at least these latter 71 packages are impacted by world age because they bypass it using `invokelatest`. Drawing conclusions about `eval`-using packages requires further analysis.

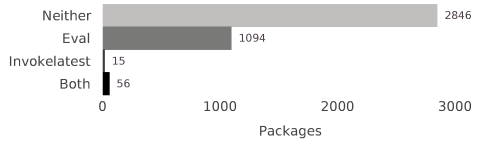

Fig. 8. `eval` and `invokelatest` use by package

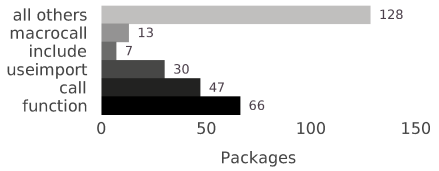

Fig. 9. Static use of AST forms in all packages

To understand if packages that only use `eval` are impacted by world age, we statically analyzed the location of calls to `eval` and their arguments by parsing files that contain `eval`. For each call, we classify the argument ASTs, recursively traversing them and counting occurrences of relevant nodes. The analysis is conservative: it assumes that an AST that is not statically obvious (such as a variable) could contain anything. Fig. 9 shows how many packages use world-age relevant AST forms. Only uses of `eval` from within functions—where world-age could be relevant—are shown; top-level uses of `eval`—which cannot be affected by world age—are filtered out. The "all-others" category encompasses all AST forms not relevant to world age. While this aggregate is, taken as a whole, more common than any other single AST form, none of the constituent AST forms is more prevalent than function calls. Therefore, most common arguments to `eval` are function definitions, followed by function calls and loading of modules and other files.

Using the results of the static analysis, we estimate that about 4–9% of the 4,011 packages might be affected by world age. The upper bound (360 packages) is a conservative estimate, which includes 289 packages with potentially world-age-related calls to `eval` but without calls to `invokelatest`, and the 71 packages that use `invokelatest`. The lower bound (186 packages) includes 115 `invokelatest`-free packages that call `eval` with both function definitions and function calls, and the 71 packages with `invokelatest`.

To validate our static results, we dynamically analyzed 32 packages out of the 186 identified as possibly affected by world age. These packages were selected by randomly sampling a subset of 49 packages, which was then further reduced by removing packages that did not run, whose tests failed, or that did not call `eval` or `invokelatest` at least once. Over this corpus, the dynamic

analysis was implemented by adding instrumentation to record calls to `eval` and `invokelatest`, recording the ASTs and functions, respectively, as well as the stack traces for each invocation.

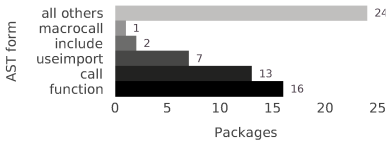

Fig. 10. Static use of AST forms used by package

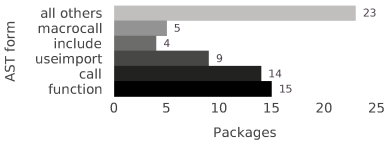

Fig. 11. Dynamic use of AST forms used by package

The results of the static and dynamic analysis of the 32 packages are given in Fig. 10 and Fig. 11, respectively. Both analysis methods agreed that the most common world-age-relevant use of `eval` was to define functions, followed by making function calls and importing other packages. In general, the dynamic analysis was able to identify more packages that used each AST form, as it can examine every AST ran through `eval`, not only statically declared ones. However, this accuracy is dependent on test coverage.

3.4 Programming with World Age

We now turn to common patterns found by manual inspection of select packages of the corpus.

Boilerplating. The most common use of `eval` is to automatically generate code for boilerplate functions. These generated functions are typically created at the top-level so that they can be used by the rest of the program. Consider the `DualNumbers.jl` package, which provides a common dual number representation for automatic differentiation. A dual number, which is a pair of the normal value and an "epsilon", which represents the derivative of the value, should support the same operations as any number does and mostly defers to the standard operations. For example, the `real` function, which gets the real component of a number when applied to a dual number should recurse into both the actual and epsilon value. `Eval` can generate all of the needed implementations at package load time (`@eval` is a macro that passes its argument to `eval` as an AST).

```
for op in (:real, :imag, :conj, :float, :complex)
   @eval Base.$op(z::Dual) = Dual($op(value(z)), $op(epsilon(z)))
end
```

A common sub-pattern is to generate proxies for interfaces defined by an external system. For this purpose, the `CxxWrap.jl` library uses `eval` at the top level to generate (with the aid of a helper method that generates the ASTs) proxies for arbitrary C++ libraries.

```
eval(build_function_expression(func, funcidx, julia_mod))
```

Defensive callbacks. The most widely used pattern for `invokelatest` deals with function values of unknown age. For example, when invoking a callback provided by a client, a library may protect itself against the case where the provided function was defined after the library was loaded. There are two forms of this pattern. The simplest uses `invokelatest` for all callbacks, such as the library `Symata.jl`:

```
for hook in preexecute_hooks
  invokelatest(hook)
end
```

Every hook in `preexecute_hooks` is protected against world-age errors (at the cost of slower function calls). To avoid this slowdown, the second common pattern catches world-age exceptions and falls back to `invokelatest` such as in from the `Genie.jl` web server:

```
fr::String = try
  f()::String
catch
  Base.invokelatest(f)::String
end
```

This may cause surprises, however. If a sufficiently old method exists, the call may succeed but invoke the wrong method.[2] This pattern may also catch unwanted exceptions and execute `f` twice, including its side-effects.

Domain-specific generation. As a language targeting scientific computing, Julia has a large number of packages that do various symbolic domain reasoning. Examples include symbolic math libraries, such as `Symata` and `GAP`, which have the functionality to generate executable code for symbolic expressions. `Symata` provides the following method to convert an internal expression (a `Mxpr`) into a callable function. Here, `Symata` uses a translation function `mxpr_to_expr` to convert the `Symata mxpr` into a Julia `Expr`, then wraps it in a function definition (written using explicit AST forms), before passing it to `eval`.

```
function Compile(a::Mxpr{:List}, body)
  aux = MtoECompile()
  jexpr = Expr(:function,
               Expr(:tuple, [mxpr_to_expr(x, aux) for x in margs(a)]...),
               mxpr_to_expr(body, aux))
  Core.eval(Main, jexpr)
end
```

Bottleneck. Generated code is commonly used in Julia as a way to mediate between a high-level DSL and a numerical library. Compilation from the DSL to executable code can dramatically improve efficiency while still retaining a high-level representation. However, functions generated thusly cannot be called from the code that generated them, since they are too new. Furthermore, this code is expected to be high-performance, so using `invokelatest` for every call is not acceptable. The bottleneck pattern overcomes these issues. The idea is to split the program into two parts: one that generates code, and another that runs it. The two parts are bridged with a single `invokelatest` call (the "bottleneck"), allowing the second part to call the generated code efficiently. The pattern is used in the `DiffEqBase` library, part of the DifferentialEquations family of libraries that provides numerical differential equation solvers.

```
if hasfield(typeof(_prob),:f) && hasfield(typeof(_prob.f),:f) &&
     typeof(_prob.f.f) <: EvalFunc
  Base.invokelatest(__solve,_prob,args...; kwargs...)
else
  __solve(_prob,args...;kwargs...)
end
```

[2]In Julia, higher-order functions are passed by name as generic functions, so a callback will be subject to multiple dispatch.

Here, if `_prob` has a field `f`, which has another field `f`, and the type of said inner-inner `f` is an `EvalFunc` (an internally-defined wrapper around any function that was generated with `eval`), then it will invoke the `__solve` function using `invokelatest`, thus allowing `__solve` to call said method. Otherwise, it will do the invocation normally.

Superfluous eval. This is a rare anti-pattern, probably indicating a misunderstanding of world age by some Julia programmers. For example, `Alpine.jl` package has the following call to `eval`:

```
if isa(m.disc_var_pick, Function)
  eval(m.disc_var_pick)(m)
```

Here, `eval(m.disc_var_pick)` does nothing useful but imposes a performance overhead. Because `m.disc_var_pick` is already a function value, calling `eval` on it is similar to using `eval(42)` instead of `42` directly; this neither bypasses the world age nor even interprets an AST.

Name-based dispatch. Another anti-pattern uses `eval` to convert function names to functions. For example, `ClassImbalance.jl` package chooses a function to call, using its uninterpreted name:

```
func = (labeltype == :majority) ? :argmax : :argmin
indx = eval(func)(counts)
```

It would be more efficient to operate with function values directly, i.e. `func =... : argmin` and then call it with `func(counts)`. Similarly, when a symbol being looked up is generated dynamically, as it is in the following example from `TextAnalysis.jl`, the use of `eval` could be avoided.

```
newscheme = uppercase(newscheme)
if !in(newscheme, available_schemes) ...
newscheme = eval(Symbol(newscheme))()
```

This pattern could be replaced with a call `getfield(TextAnalysis,Symbol(newscheme))`, where `getfield` is a special built-in function that finds a value in the environment by its name. Using `getfield` would be more efficient than `eval`.

4 JULIETTE, A WORLD AGE CALCULUS

To formally study world age, we propose a core calculus, named Juliette, that captures the essence of Julia's semantics and permits us to reason about the correctness of some of the optimizations performed by the compiler.

Designing such a calculus is always an exercise in parsimony, balancing the need to highlight principles while avoiding entanglements with particular implementation choices. The first decision to grapple with is how to represent world age. While efficient, counters are also pervasive and cause confusion.[3] Furthermore, they obscure reasoning about program-state equivalence; two programs with different initial counter values could, if care is not taken, appear different. Dispensing with the counters used by Julia's compiler is appealing.

An alternative that we chose is a more abstract representation of world age, one that captures its intent: control over method visibility. Juliette uses *method tables* to represent sets of methods available for dispatch. The *global table* is the method table that records all definitions and always reflects the "true age" of the world; the global table is part of Juliette program state. *Local tables*

[3] Although Julia's documentation attempts to explain world age [v1 2020], questions such as this one pop up periodically.

are method tables used to resolve method dispatch during execution and may lag behind the global table when new functions are introduced. Local tables are then baked into program syntax to make them explicit during execution. As in Julia, Juliette separates method tables (which represent code) from data: as mentioned in Sec. 1, the world-age semantics only applies to code. As global variables interact with `eval` in the standard way, we omit them from the calculus.

The treatment of methods is similar in both Juliette and Julia up to (lexically) local method definitions. In both systems, a generic function is defined by the set of methods with the same name. In Julia, local methods are syntactic sugar for global methods with fresh names. For simplicity, we do not model this aspect of Julia: Juliette methods are always added to the global method table. All function calls are resolved using the set of methods found in the current local table. A function value $\mathtt{m}$ denotes the name of a function and is not itself a method definition. Then, since Juliette omits global variables, its global environment is entirely captured by the global method table.

Although in Julia `eval` incorporates two features—top-level evaluation and quotation[4]—only top-level evaluation is relevant to world age, and this is what we model in Juliette. Instead of an `eval` construct, the calculus has operations for evaluating expressions in different method-table contexts. In particular, Juliette offers a *global evaluation construct* $(\!|\, \mathtt{e} \,|\!)$ (pronounced "banana brackets") that accesses the most recent set of methods. This is equivalent to `eval`'s behavior, which evaluates in the latest world age. Since Juliette does not have global variables, $(\!|\, \mathtt{e} \,|\!)$ reads from the local environment directly instead of using quotation.

Every function call $\mathtt{m}(\overline{\mathtt{v}})$ in Juliette gets resolved in the closest enclosing local method table M by using an *evaluation-in-a-table* construct $(\!|\, \mathtt{m}(\overline{\mathtt{v}}) \,|\!)_{\mathrm{M}}$. Any top-level function call first takes a snippet of the current global table and then evaluates the call in that *frozen* snippet. That is, $(\!|\, \mathtt{m}(\overline{\mathtt{v}}) \,|\!)$ steps to $(\!|\, \mathtt{m}(\overline{\mathtt{v}}) \,|\!)_{\mathrm{M}}$ where M is the current global table. Thus, once a snippet of the global table becomes local table, all inner function calls of $\mathtt{m}(\overline{\mathtt{v}})$ will be resolved using this table, reflecting the fact that a currently executing top-level function call does not see updates to the global table.

To focus on world age, Juliette omits irrelevant features such as loops or mutable variables. Furthermore, the calculus is parameterized over values, types, type annotations, a subtyping relation, and primitive operations. For the purposes of this paper, only minimal assumptions are needed about those.

4.1 Syntax

The surface syntax of Juliette is given in Fig. 12. It includes method definitions $\mathtt{md}$, function calls $\mathtt{e}(\overline{\mathtt{e}})$, sequencing $\mathtt{e}_1 \,;\, \mathtt{e}_2$, global evaluation $(\!|\, \mathtt{e} \,|\!)$, evaluation in a table $(\!|\, \mathtt{e} \,|\!)_{\mathrm{M}}$, variables x, values $\mathtt{v}$, primitive calls $\delta_l(\overline{\mathtt{e}})$, type tags σ, and type annotations τ. Values $\mathtt{v}$ include `unit` (unit value, called `nothing` in Julia) and $\mathtt{m}$ (generic function value). Primitive operators δ_l represent built-in functions such as `Base.mul_int`. Type tags σ include $\mathbb{1}$ (unit type, called `Nothing` in Julia) and $\mathbb{f}_{\mathtt{m}}$ (tag of function value $\mathtt{m}$). Type annotations τ include $\top \in \tau$ ($\top$ is the top type, called `Any` in Julia) and $\sigma \subseteq \tau$ (all type tags serve as valid type annotations).

4.2 Semantics

The internal syntax of Juliette is given in the top of Fig. 13. It includes evaluation result $\mathtt{r}$ (either value or error), method table M, and two evaluation contexts, $\mathtt{X}$ and $\mathtt{C}$, which are used to define small-step operational semantics of Juliette. Evaluation contexts $\mathtt{X}$ are responsible for simple sequencing, such as the order of argument evaluation; these contexts never contain global/table evaluation expressions $(\!|\, \cdot \,|\!)$ and $(\!|\, \cdot \,|\!)_{\mathrm{M}}$. World evaluation contexts $\mathtt{C}$, on the other hand, capture the full grammar of expressions.

[4]Represented with the `$` operator in Julia, as in `eval(:(g() = $x))` in Fig. 3.

e	::=		*Expression*
	|	v	value
	|	x	variable
	|	$\mathtt{e}_1$; $\mathtt{e}_2$	sequencing
	|	$\delta_I(\overline{\mathtt{e}})$	primop call
	|	$\mathtt{e}(\overline{\mathtt{e}})$	function call
	|	md	method definition
	|	$(\!\|\, \mathtt{e} \,\|\!)$	global evaluation
	|	$(\!\|\, \mathtt{e} \,\|\!)_{\mathrm{M}}$	evaluation in a table
p	::=	$(\!\|\, \mathtt{e} \,\|\!)$	*Program*
md	::=	$\triangleleft\ \mathtt{m}(\overline{x :: \tau}) = \mathtt{e} \triangleright$	*Method definition*

v	:=	...	*Value*
	|	unit	unit value
	|	m	generic function
σ	:=	...	*Type tag*
	|	$\mathbb{1}$	unit type
	|	$\mathbb{f}_{\mathtt{m}}$	type tag of function m
τ	:=	...	*Type annotation*
	|	$\top$	top type

Fig. 12. Surface syntax

Program state is a pair $\langle \mathrm{M}, \mathtt{C}\,[\mathtt{e}] \rangle$ of a global method table M and an expression $\mathtt{C}\,[\mathtt{e}]$. We define the semantics of the calculus using two judgments: a normal small-step evaluation denoted by $\langle \mathrm{M}, \mathtt{C}\,[\mathtt{e}] \rangle \ \longrightarrow\ \langle \mathrm{M}', \mathtt{C}\,[\mathtt{e}'] \rangle$, and a step to an error $\mathrm{M} \ \vdash\ \mathtt{C}\,[\mathtt{e}] \ \longrightarrow\ \mathtt{error}$. The $\mathrm{typeof}(\mathtt{v}) \in \sigma$ operator returns the tag of a value. We require that $\mathrm{typeof}(\mathtt{unit}) = \mathbb{1}$ and $\mathrm{typeof}(\mathtt{m}) = \mathbb{f}_{\mathtt{m}}$. We write $\mathrm{typeof}(\overline{\mathtt{v}})$ as a shorthand for $\overline{\mathrm{typeof}(\mathtt{v})}$. Function $\Delta(l, \overline{\mathtt{v}}) \in \mathtt{r}$ computes primop calls, and function $\Psi(l, \overline{\sigma}) \in \sigma$ indicates the tag of l's return value when called with arguments of types $\overline{\sigma}$. These functions have to agree, i.e. $\forall \overline{\mathtt{v}}, \overline{\sigma}.(\mathrm{typeof}(\overline{\mathtt{v}}) = \overline{\sigma} \wedge \Delta(l, \overline{\mathtt{v}}) = \mathtt{v}' \implies \mathrm{typeof}(\mathtt{v}') = \Psi(l, \overline{\sigma}))$. The subtyping relation $\tau_1 <: \tau_2$ is used for multiple dispatch. We require that $\tau <: \top$ ($\top$ is indeed the top type) and $\sigma_1 <: \sigma_2 \Leftrightarrow \sigma_1 \equiv \sigma_2$ (tags are final, i.e. do not have subtypes).

Normal Evaluation. These rules capture successful program executions. Rule E-Seq is completely standard: it throws away the evaluated part of a sequencing expression. Rules E-ValGlobal and E-ValLocal pass value v to the outer context. This is similar to Julia where `eval` returns the result of evaluating the argument to its caller. Rule E-MD is responsible for updating the global table: a method definition md will extend the current global table M into $\mathrm{M} \bullet \mathtt{md}$, and itself evaluate to m, which is a function value. Note that E-MD only extends the method table and leaves existing definitions in place. If the table contains multiple definitions of a method with the same signature, it is then the dispatcher's responsibility to select the right method; this mechanism is described below in more detail.

The two call forms E-CallGlobal and E-CallLocal form the core of the calculus. The rule E-CallGlobal describes the case where a method is called directly from a global evaluation expression. In Julia, this means either a top-level call, an `invokelatest` call, or a call within `eval` such as `eval(:(g(...)))`. The "direct" part is encoded with the use of a simple evaluation context X. In this global-call case, we need to save the current method table into the evaluation context for a subsequent use by E-CallLocal. To do this, we annotate the call $\mathtt{m}(\overline{\mathtt{v}})$ with a copy of the current global method table M, producing $(\!\|\, \mathtt{m}(\overline{\mathtt{v}}) \,\|\!)_{\mathrm{M}}$.

To perform a local call—or, equivalently, a call after the invocation has been wrapped in an annotation specifying the current global table—E-CallLocal is used. This rule resolves the call according to the tag-based multiple-dispatch semantics in the "deepest" method table M' (the use of X makes sure there are no method tables between M' and the call). Once an appropriate method has been found, it proceeds as a normal invocation rule would, replacing the method invocation with the substituted-for-arguments method body. Note that the body of the method is still wrapped

$$
\begin{array}{rcll}
r & ::= & & \textit{Result} \\
 & | & \mathtt{v} & \text{value} \\
 & | & \mathtt{error} & \text{error} \\
 & & & \\
\mathrm{M} & ::= & & \textit{Method table} \\
 & | & \varnothing & \text{empty table} \\
 & | & \mathrm{M} \bullet \mathtt{md} & \text{table extension}
\end{array}
\qquad
\begin{array}{rcll}
\mathtt{X} & ::= & & \textit{Simple evaluation context} \\
 & | & \square & \text{hole} \\
 & | & \mathtt{X}\,;\,\mathtt{e} & \text{sequence} \\
 & | & \delta_I(\overline{\mathtt{v}}\ \mathtt{X}\ \overline{\mathtt{e}}) & \text{primop call (argument)} \\
 & | & \mathtt{X}(\overline{\mathtt{e}}) & \text{function call (callee)} \\
 & | & \mathtt{v}(\overline{\mathtt{v}}\ \mathtt{X}\ \overline{\mathtt{e}}) & \text{function call (argument)} \\
 & & & \\
\mathtt{C} & ::= & & \textit{World evaluation context} \\
 & | & \mathtt{X} & \text{simple context} \\
 & | & \mathtt{X}\left[\!\left(\mathtt{C}\right)\!\right] & \text{global evaluation} \\
 & | & \mathtt{X}\left[\!\left(\mathtt{C}\right)\!\right]_{\mathrm{M}} & \text{evaluation in a table M}
\end{array}
$$

$$\textsc{E-Seq}\ \frac{}{\langle \mathrm{M}, \mathtt{C}\,[\mathtt{v}\,;\,\mathtt{e}]\rangle \ \rightarrow\ \langle \mathrm{M}, \mathtt{C}\,[\mathtt{e}]\rangle}$$

$$\textsc{E-Primop}\ \frac{\Delta(I, \overline{\mathtt{v}}) = \mathtt{v}'}{\langle \mathrm{M}, \mathtt{C}\,[\delta_I(\overline{\mathtt{v}})]\rangle \ \rightarrow\ \langle \mathrm{M}, \mathtt{C}\,[\mathtt{v}']\rangle}$$

$$\textsc{E-MD}\ \frac{\mathtt{md} \equiv \triangleleft\ \mathtt{m}(\overline{x :: \tau}) = \mathtt{e} \triangleright}{\langle \mathrm{M}, \mathtt{C}\,[\mathtt{md}]\rangle \ \rightarrow\ \langle \mathrm{M} \bullet \mathtt{md}, \mathtt{C}\,[\mathtt{m}]\rangle}$$

$$\textsc{E-CallGlobal}\ \frac{}{\langle \mathrm{M}, \mathtt{C}\left[\!\left(\mathtt{X}\,[\mathtt{m}(\overline{\mathtt{v}})]\right)\!\right]\rangle \ \rightarrow\ \langle \mathrm{M}, \mathtt{C}\left[\!\left(\mathtt{X}\left[\!\left(\mathtt{m}(\overline{\mathtt{v}})\right)\!\right]_{\mathrm{M}}\right)\!\right]\rangle}$$

$$\textsc{E-CallLocal}\ \frac{\mathrm{typeof}(\overline{\mathtt{v}}) = \overline{\sigma} \qquad \mathrm{getmd}(\mathrm{M}', \mathtt{m}, \overline{\sigma}) = \triangleleft\ \mathtt{m}(\overline{x :: \tau}) = \mathtt{e} \triangleright}{\langle \mathrm{M}, \mathtt{C}\left[\!\left(\mathtt{X}\,[\mathtt{m}(\overline{\mathtt{v}})]\right)\!\right]_{\mathrm{M}'}\rangle \ \rightarrow\ \langle \mathrm{M}, \mathtt{C}\left[\!\left(\mathtt{X}\,[\mathtt{e}[\overline{x} \mapsto \overline{\mathtt{v}}]]\right)\!\right]_{\mathrm{M}'}\right\rangle}$$

$$\textsc{E-ValGlobal}\ \frac{}{\langle \mathrm{M}, \mathtt{C}\left[\!\left(\mathtt{v}\right)\!\right]\rangle \ \rightarrow\ \langle \mathrm{M}, \mathtt{C}\,[\mathtt{v}]\rangle}$$

$$\textsc{E-ValLocal}\ \frac{}{\langle \mathrm{M}, \mathtt{C}\left[\!\left(\mathtt{v}\right)\!\right]_{\mathrm{M}'}\rangle \ \rightarrow\ \langle \mathrm{M}, \mathtt{C}\,[\mathtt{v}]\rangle}$$

$$\textsc{E-VarErr}\ \frac{}{\mathrm{M} \vdash \mathtt{C}\,[x] \rightarrow \mathtt{error}}$$

$$\textsc{E-PrimopErr}\ \frac{\Delta(I, \overline{\mathtt{v}}) = \mathtt{error}}{\mathrm{M} \vdash \mathtt{C}\,[\delta_I(\overline{\mathtt{v}})] \rightarrow \mathtt{error}}$$

$$\textsc{E-CalleeErr}\ \frac{\mathtt{v}_c \neq \mathtt{m}}{\mathrm{M} \vdash \mathtt{C}\,[\mathtt{v}_c(\overline{\mathtt{v}})] \rightarrow \mathtt{error}}$$

$$\textsc{E-CallErr}\ \frac{\mathrm{typeof}(\overline{\mathtt{v}}) = \overline{\sigma} \qquad \mathrm{getmd}(\mathrm{M}', \mathtt{m}, \overline{\sigma}) = \mathtt{error}}{\mathrm{M} \vdash \mathtt{C}\left[\!\left(\mathtt{X}\,[\mathtt{m}(\overline{\mathtt{v}})]\right)\!\right]_{\mathrm{M}'} \rightarrow \mathtt{error}}$$

$$
\begin{array}{rcl}
\mathrm{getmd}(\mathrm{M}, \mathtt{m}, \overline{\sigma}) & = & \min(\mathrm{applicable}(\mathrm{latest}(\mathrm{M}), \mathtt{m}, \overline{\sigma})) \\
 & & \\
\mathrm{latest}(\mathrm{M}) & = & \mathrm{latest}(\emptyset, \mathrm{M}) \\
\mathrm{latest}(\mathit{mds}, \varnothing) & = & \mathit{mds} \\
\mathrm{latest}(\mathit{mds}, \mathrm{M} \bullet \mathtt{md}) & = & \mathrm{latest}(\mathit{mds} \cup \mathtt{md}, \mathrm{M}) \text{ if } \neg\,\mathrm{contains}(\mathit{mds}, \mathtt{md}) \\
\mathrm{latest}(\mathit{mds}, \mathrm{M} \bullet \mathtt{md}) & = & \mathrm{latest}(\mathit{mds}, \mathrm{M}) \qquad \text{if } \mathrm{contains}(\mathit{mds}, \mathtt{md}) \\
 & & \\
\mathrm{applicable}(\mathit{mds}, \mathtt{m}, \overline{\sigma}) & = & \{\triangleleft\ \mathtt{m}(\overline{x :: \tau}) = \mathtt{e} \triangleright \in \mathit{mds} \mid \overline{\sigma} <: \overline{\tau}\} \\
 & & \\
\min(\mathit{mds}) & = & \triangleleft\ \mathtt{m}(\overline{x :: \tau}) = \mathtt{e} \triangleright \in \mathit{mds} \text{ such that } \forall \triangleleft\ \mathtt{m}(\overline{\_ :: \tau'}) = \_ \triangleright \in \mathit{mds}\ .\ \overline{\tau} <: \overline{\tau'} \\
\min(\mathit{mds}) & = & \mathtt{error} \text{ otherwise} \\
\mathrm{contains}(\mathit{mds}, \mathtt{md}) & = & \exists\, \mathtt{md}' \in \mathit{mds} \text{ such that} \\
 & & \quad (\mathtt{md} \equiv \triangleleft\ \mathtt{m}(\overline{\_ :: \tau}) = \_ \triangleright)\ \wedge\ (\mathtt{md}' \equiv \triangleleft\ \mathtt{m}(\overline{\_ :: \tau'}) = \_ \triangleright)\ \wedge\ \overline{\tau} <: \overline{\tau'} \wedge \overline{\tau'} <: \overline{\tau}
\end{array}
$$

Fig. 13. Internal Syntax and Semantics

in the $(\!|\ |\!)_{M'}$ context. This ensures that nested calls will be resolved in the same table (unless they are more deeply wrapped in a global evaluation $(\!|\ |\!)$).

An auxiliary meta-function $\mathrm{getmd}(M, \mathtt{m}, \overline{\sigma})$, which is used to resolve multiple dispatch, is defined in the bottom of Fig. 13. This function returns the most specific method applicable to arguments with type tags $\overline{\sigma}$, or errs if such a method does not exist. If the method table contains multiple equivalent methods, older ones are ignored. For example, for the program

$$(\!|\ \triangleleft\ \mathtt{g}()=2\ \triangleright\ ;\ \triangleleft\ \mathtt{g}()=42\ \triangleright\ ;\ \mathtt{g}()\ |\!),$$

function call $\mathtt{g}()$ is going to be resolved in the table $(\varnothing \bullet \triangleleft\ \mathtt{g}()=2\ \triangleright) \bullet \triangleleft\ \mathtt{g}()=42\ \triangleright$, which contains two equivalent methods (we call methods equivalent if they have the same name and their argument type annotations are equivalent with respect to subtyping). In this case, the function getmd will return method $\triangleleft\ \mathtt{g}()=42\ \triangleright$ because it is the newest method out of the two.

Note that functions can be mutually recursive because of the dynamic nature of function call resolution.

Error Evaluation. These rules capture all possible error states of Juliette. Rule E-VarErr covers the case of a free variable, an `UndefVarError` in Julia. E-PrimopErr accounts for errors in primitive operations such as `DivideError`. E-CalleeErr fires when a non-function value is called. Finally, E-CallErr accounts for multiple-dispatch resolution errors, e.g. when the set of applicable methods is empty (no method found), and when there is no best method (ambiguous method).

4.3 Example

Fig. 14 shows a translation of the program from Fig. 3 to Juliette. First note that, as part of the translation, we wrap the entire program in $(\!|\ |\!)$, indicating that the outermost scope is the top level. Translation of method calls and definitions then proceeds, using $\triangleleft\ \mathtt{m}(\overline{x})=\mathtt{e}\ \triangleright$ as a shorthand for $\triangleleft\ \mathtt{m}(\overline{x :: \top})=\mathtt{e}\ \triangleright$ where $\top$ is the top type. Method bodies are converted by replacing `eval` invocations with their expressions wrapped in $(\!|\ |\!)$. The $(\!|\ |\!)$ context of `e` in Juliette effectively acts the same way that `eval` of `e` does in Julia, but evaluates variables in `e` using local, rather than global, scope.

```
g()  = 2
f(x) = (eval(:(g() = $x)); x * g())
f(42)
```

$$(\!|\ \triangleleft\ \mathtt{g}()=2\ \triangleright\ ;\ \triangleleft\ \mathtt{f}(x)=(\!|\ \triangleleft\ \mathtt{g}()=x\ \triangleright\ |\!)\ ;\ x * \mathtt{g}())\ \triangleright\ ;\ \mathtt{f}(42)\ |\!)$$

Fig. 14. From Julia (left) to Juliette (right)

Now we will show the execution of this translated program according to our small-step semantics. The initial state is $\langle \varnothing, \mathtt{p} \rangle$ where `p` is the program on the right of Fig. 14 (and the $*$ operator is a primop). The first several steps of evaluation use rules E-MD and E-Seq to add the definitions of `g` and `f` to the global table. This produces the state

$$\langle M_0, (\!|\ \mathtt{f}(42)\ |\!) \rangle,$$

where

$$M_0 \quad = \quad (\varnothing \bullet \triangleleft\ \mathtt{g}()=2\ \triangleright) \\ \bullet \triangleleft\ \mathtt{f}(x)=(\!|\ \triangleleft\ \mathtt{g}()=x\ \triangleright\ |\!)\ ;\ x * \mathtt{g}())\ \triangleright$$

Next, using the E-CallGlobal rule, the top-level call $\mathtt{f}(42)$ steps to $(\!|\ \mathtt{f}(42)\ |\!)_{M_0}$. This then produces the state

$$\langle M_0, (\!|\ (\!|\ \mathtt{f}(42)\ |\!)_{M_0}\ |\!) \rangle,$$

copying the global table into the context $(\!|\cdot|\!)_{M_0}$. Now, rule E-CallLocal can be used to resolve the call $\mathtt{f}(42)$ in the table M_0. Method $\triangleleft\ \mathtt{f}(x)= \ldots \triangleright$ is the only method of $\mathtt{f}$ and it is applicable to the integer argument ($\mathrm{typeof}(42) = \mathtt{Int} <: \top$), so the program steps to:

$$\langle M_0, (\!|\ (\!|\triangleleft\ \mathtt{g}()=42\triangleright|\!)\ ;\ 42 * \mathtt{g}()\ |\!)_{M_0}\ |\!)\rangle.$$

The next expression to evaluate is the new g definition, $\triangleleft\ \mathtt{g}()=42\triangleright$. Rule E-MD fires and the program steps to

$$\langle M_1, (\!|\ (\!|\mathtt{g}|\!)\ ;\ 42 * \mathtt{g}()\ |\!)_{M_0}\ |\!)\rangle,$$

where

$$\begin{array}{rcll} M_1 & = & M_0 & = & ((\varnothing \bullet \triangleleft\ \mathtt{g}()=2\triangleright) \\ & & \bullet \triangleleft\ \mathtt{g}()=42\triangleright & & \bullet \triangleleft\ \mathtt{f}(x)= (\!|\triangleleft\ \mathtt{g}()=x\triangleright|\!)\ ;\ x * \mathtt{g}())\triangleright) \\ & & & & \bullet \triangleleft\ \mathtt{g}()=42\ \triangleright\,. \end{array}$$

The next two steps are:

$$\langle M_1, (\!|\ (\!|\mathtt{g}|\!)\ ;\ 42 * \mathtt{g}()\ |\!)_{M_0}\ |\!)\rangle \xrightarrow{\text{E-ValGlobal}} \langle M_1, (\!|\ (\!|\mathtt{g}\ ;\ 42 * \mathtt{g}()\ |\!)_{M_0}\ |\!)\rangle \quad (1)$$

$$\xrightarrow{\text{E-Seq}} \langle M_1, (\!|\ (\!|42 * \mathtt{g}()\ |\!)_{M_0}\ |\!)\rangle. \quad (2)$$

Note that the last program state is represented by $\langle M_1, \mathtt{C}\left[(\!|\mathtt{X}\,[\mathtt{g}()]\ |\!)_{M_0}\right]\rangle$, where $\mathtt{C} = (\!|\square|\!)$ and $\mathtt{X} = 42 * \square$. So we have to use E-CallLocal again to resolve $\mathtt{g}()$ in the M_0 that is fixed in the context. Table M_0 has only one definition of g, the one that returns 2, so the program steps to:

$$\langle M_1, (\!|\ (\!|42 * 2\ |\!)_{M_0}\ |\!)\rangle.$$

Finally, the application of E-Primop, E-ValLocal, and E-ValGlobal leads to the final state:

$$\langle M_1, 84\rangle. \quad (3)$$

Now, consider a modification of the original program where in the definition of f, the call $\mathtt{g}()$ is wrapped into a global evaluation $(\!|\mathtt{g}()\ |\!)$:

```
g()  = 2
f(x) = (eval(:(g() = $x)); x * eval(:(g())))
f(42)
```

$$(\!|\ \triangleleft\ \mathtt{g}()=2\triangleright\ ;\ \triangleleft\ \mathtt{f}(x)= (\!|\triangleleft\ \mathtt{g}()=x\triangleright|\!)\ ;\ x * (\!|\mathtt{g}()\ |\!))\triangleright\ ;\ \mathtt{f}(42)\ |\!)$$

At the beginning, the modified program will run similarly to the original one, and with step (2), it will reach the state:

$$\langle M_1, (\!|\ (\!|42 * (\!|\mathtt{g}()\ |\!)\ |\!)_{M_0}\ |\!)\rangle.$$

Here, $(\!|\ (\!|42 * (\!|\mathtt{g}()\ |\!)\ |\!)_{M_0}\ |\!)$ is represented by $\mathtt{C}\left[(\!|\mathtt{X}\,[\mathtt{g}()]\ |\!)\right]$, where $\mathtt{C} = (\!|\ (\!|42 * \square|\!)_{M_0}\ |\!)$ and $\mathtt{X} = \square$. Therefore, the call $\mathtt{g}()$ is back at the top level. With E-CallGlobal rule, the call steps to $(\!|\mathtt{g}()\ |\!)_{M_1}$ because M_1 is the *current global* table, thus producing the state:

$$\langle M_1, (\!|\ (\!|42 * (\!|\ (\!|\mathtt{g}()\ |\!)_{M_1}\ |\!)\ |\!)_{M_0}\ |\!)\rangle.$$

Resolved in M_1, call $\mathtt{g}()$ returns 42, and thus the whole program ends in the final state:

$$\langle M_1, 1764\rangle.$$

Note that the resulting global table is the same as in (3), but the return value is different.

$$
\begin{array}{lllllll}
\texttt{rdx} & ::= & & & & & \\
 & \mid \ x & & & & & \\
 & \mid \ \texttt{v ; e} & \mid \ \delta_l(\overline{\texttt{v}}) & \mid \ \texttt{md} & \mid \ (\!|\, \texttt{v} \,|\!) & \mid \ (\!|\, \texttt{X}\,[\texttt{m}(\overline{\texttt{v}})]\,|\!) & \\
 & & \mid \ \texttt{v}^{\neq \texttt{m}}(\overline{\texttt{v}}) & & \mid \ (\!|\, \texttt{v} \,|\!)_{\mathrm{M}} & \mid \ (\!|\, \texttt{X}\,[\texttt{m}(\overline{\texttt{v}})]\,|\!)_{\mathrm{M}} &
\end{array}
$$

Fig. 15. Redex bases

4.4 Properties

Juliette operational semantics is deterministic, and all failure states are captured by error evaluation (meaning that a Juliette program never gets stuck).

Lemma 4.1 (Unique Form of Expression). *Any expression* e *can be uniquely represented in one of the following ways:*

(a) $\texttt{e} = \texttt{v}$*; or*
(b) $\texttt{e} = \texttt{X}\,[\texttt{m}(\overline{\texttt{v}})]$*; or*
(c) $\texttt{e} = \texttt{C}\,[\texttt{rdx}]$*,*

where rdx *(shown in Fig. 15) is a subset of expressions driving the reduction.*

Proof. By induction on e. □

Theorem 4.2 (Progress). *For any program* p *and method table* M_g*, the program either reduces to a value, or it makes a step to another program, or it errs. That is, one of the following holds:*

(a) $\langle \mathrm{M}_g, \texttt{p} \rangle \longrightarrow \langle \mathrm{M}'_g, \texttt{v} \rangle$*; or*
(b) $\langle \mathrm{M}_g, \texttt{p} \rangle \longrightarrow \langle \mathrm{M}'_g, \texttt{p}' \rangle$*; or*
(c) $\mathrm{M}_g \vdash \texttt{p} \longrightarrow \texttt{error}$.

Proof. By case analysis on $\texttt{p} = (\!|\, \texttt{e} \,|\!)$, using Lemma 4.1. □

Theorem 4.3 (Determinism). *Juliette semantics is deterministic.*

Proof. The proof relies on the fact that (1) any expression that steps can be represented as $\texttt{C}\,[\texttt{rdx}]$, and (2) such a representation is unique by Lemma 4.1. By case analysis on rdx, we can see that for all redex bases except $\delta_l(\overline{\texttt{v}})$ and $(\!|\, \texttt{X}\,[\texttt{m}(\overline{\texttt{v}})]\,|\!)_{\mathrm{M}}$, there is exactly one (normal- or error-evaluation) rule applicable. For $\delta_l(\overline{\texttt{v}})$ and $(\!|\, \texttt{X}\,[\texttt{m}(\overline{\texttt{v}})]\,|\!)_{\mathrm{M}}$, there are two rules for each, but their premises are incompatible. Thus, for any expression $\texttt{C}\,[\texttt{rdx}]$, exactly one rule is applicable. □

4.5 Optimizations

The world-age semantics allows function-call optimization even in the presence of eval. Recall how an evaled or top-level function call $(\!|\, \texttt{m}(\overline{\texttt{v}}) \,|\!)$ steps. First, rule E-CallGlobal is applied: it fixes the current state of the global table M in the call's context, stepping the call to $(\!|\, \texttt{m}(\overline{\texttt{v}}) \,|\!)_{\mathrm{M}}$. Then, the call $\texttt{m}(\overline{\texttt{v}})$ itself, and all of its nested calls (unless they are additionally wrapped into $(\!|\ |\!)$), are resolved using the now-local table M. Therefore, M provides all necessary information for the resolution of such calls, and they can be optimized based on the method table M.

Next, we will focus on three generic-call optimizations: inlining, specialization, and transforming generic calls into direct calls (devirtualization). Namely, we provide formal definitions of these optimizations and show them correct.

```
g(x::Any)  = x + x    # g1
g(x::Bool) = x        # g2
f(x::Int)  = x * g(x)
f(5)
```

```
g(x::Any)  = x + x    # g1
g(x::Bool) = x        # g2
f(x::Int)  = x * g((println(x);x))
f(5)
```

Fig. 16. Candidate programs for inlining (on the left) and direct call optimization (on the right)

Inlining. If a function call is known to dispatch to a certain method using a *fixed* method table, it might be possible to *inline* the body of the method in place of the call. For example, consider a program on the left of Fig. 16. The call `f(5)` has no choice but to dispatch to the only definition of `f`. Because the call `g(x)` in `f(5)` is not wrapped in an `eval`, it is known that the call to `g` is going to be dispatched in the context with exactly two methods of `g`: `g1` and `g2`. Furthermore, since `x` is known to be of type (tag) `Int` inside `f`, we know that `g(x)` has to dispatch to the method `g1` (because `Int <: Any` but `Int </: Bool`). Thus, it is possible to optimize method `f` for the call `f(5)` by inlining `g(x)`, which yields the following optimized definition of `f`:

```
f(x::Int) = x * (x + x)
```

Direct-call optimization. When inlining is not possible or desirable, but it is clear which method is going to be invoked, a function call can be replaced by a direct invocation. Consider the example on the right of Fig. 16. The only difference from the previous example is that the argument of `g` inside `f` is not a variable but an expression `(println(x);x)`. This expression always returns an integer, so we know that at run time, that `g` will be dispatch to method `g1`. However, unlike previously, the call to `g` cannot be inlined using direct syntactic substitution. In that case, the value of `x` would be printed twice instead of just once, because inlining would transform `g((println(x);x))` into `(println(x);x)+(println(x);x)` and thus change the observable behavior of the program. It is still possible to optimize `f`, by replacing the generic call to `g` with a *direct call* to the method `g1`. In pseudo-code, this can be written as:

```
f(x::Int) = x * g@g1((println(x); x))
```

In the calculus, we model a direct call as a call to a new function with a single method such that the name of the function is not used anywhere in the original method table or expression. For example, for the program above, we can add function `h` with only one method `h(x::Int)=x+x`, allowing `f` to be optimized to:

```
f(x::Int) = x * h(println(x); x)
```

Specialization. The final optimization we consider is specialization of methods for argument types. In Fig. 16, method `g1` is defined for `x` of type `Any`, meaning that the call `x+x` can be dispatched to any of at least 166 standard methods. But because, within `f`, `g` is known to be called with an argument of type `Int` (due to `x` in `f` having that type), it is possible to generate a new implementation of `g` *specialized* for this argument type. The advantage is that the specialized implementation can directly use efficient integer addition. Thus, combined with the direct call, we have:

```
g(x::Int) = Base.add_int(x, x) # g3
f(x::Int) = x * g@g3((println(x); x))
```

In the calculus, specialization is modeled similarly to direct calls: as a function with a fresh name.

$$\begin{array}{rcll} \Gamma & ::= & & \textit{Typing environment} \\ & | & \varnothing & \\ & | & \Gamma, x : \tau & \end{array}$$

$$\textsc{T-Var}\ \frac{\Gamma(x) = \sigma}{\Gamma \vdash x : \sigma} \qquad \textsc{T-Val}\ \frac{\mathrm{typeof}(\mathsf{v}) = \sigma}{\Gamma \vdash \mathsf{v} : \sigma} \qquad \textsc{T-MD}\ \frac{}{\Gamma \vdash \triangleleft\, \mathtt{m}(\overline{x :: \tau}) = \mathsf{e} \triangleright : \mathbb{f}_{\mathtt{m}}} \qquad \textsc{T-Seq}\ \frac{\Gamma \vdash \mathsf{e}_2 : \sigma}{\Gamma \vdash (\mathsf{e}_1\,;\,\mathsf{e}_2) : \sigma}$$

$$\textsc{T-Primop}\ \frac{\Psi(l, \overline{\sigma}) = \sigma' \qquad \Gamma \vdash \mathsf{e}_i : \sigma_i}{\Gamma \vdash \delta_l(\overline{\mathsf{e}}) : \sigma'} \qquad \textsc{T-EvalGlobal}\ \frac{\Gamma \vdash \mathsf{e} : \sigma}{\Gamma \vdash (\!|\mathsf{e}|\!) : \sigma} \qquad \textsc{T-EvalLocal}\ \frac{\Gamma \vdash \mathsf{e} : \sigma}{\Gamma \vdash (\!|\mathsf{e}|\!)_{\mathrm{M}} : \sigma}$$

Fig. 17. Concrete-typing judgment

4.6 Optimization Correctness

In this section, we present a formal definition of optimizations and state the main theorem about their correctness. The general idea of optimizations is as follows: if an expression e is going to be executed in a fixed method table M, it is safe to instead execute e in a table M′ obtained by optimizing method definitions of M (like we did with the definition of f in the examples above).

As demonstrated by the examples, the first ingredient of optimizations is type information, which is necessary to "statically" resolve function calls; for this, we use a simple concrete-typing relation defined in Fig. 17. The relation $\Gamma \vdash \mathsf{e} : \sigma$ propagates information about variables and type tags of values, and succeeds only if the expression would always reduce to a value of concrete type σ *if* it reduces to any value. This is because to resolve a function call, we need to know the type tags of its arguments. A typing relation can be more complex to enable further optimization opportunities (and it is much more complex in Julia), but typing of Julia is a separate topic that is out of scope of this paper: here, we focus on compiler optimization and use concrete typing only as a tool.

Fig. 18 shows the judgments related to method-table optimization. The rule OT-MethodTable says that an optimized version M′ of table M (1) has to have all the methods of M, although they can be optimized, and (2) can have more methods given that their names do not appear in the original table M. The latter enables adding new methods that model direct calls and specializations, and the former allows for optimization of existing methods. According to the rule OD-MD, a method in M′ optimizes a method in M if it has the same signature (i.e. name and argument types), and its body is an optimization of the original body of the method being optimized.

The method-optimization environment Φ tracks direct calls and specializations: $\mathtt{m}(\overline{\sigma}) \rightsquigarrow \mathtt{m}'$ tells that when arguments of m have type tags $\overline{\sigma}$, a call to m in M can be replaced by a call to m′ in M′. Note that all entries of Φ need to be valid according to MethodOpt-Valid: assuming that the methods are in the optimization relation, their bodies indeed have to be in that relation (the assumption is needed to handle recursion). Both OD-MD and MethodOpt-Valid rely on expression optimization to relate method bodies.

Finally, the expression-optimization relation is shown in Fig. 19. Note that the rules do not allow for function-call optimizations inside the global-evaluation construct (| |): the only applicable rule in that case is OE-Global. Function calls can only be optimized if they are fixed-table calls. Rules

$$\begin{array}{lcll} \Phi & ::= & & \textit{Method-optimization environment} \\ & | & \varnothing & \\ & | & \Phi, \mathtt{m}(\overline{\sigma}) \rightsquigarrow \mathtt{m}' & \end{array}$$

MethodOpt-Valid

$$\frac{\begin{array}{c}\mathrm{getmd}(\mathrm{M}, \mathtt{m}, \overline{\sigma}) = \triangleleft\ \mathtt{m}(\overline{x :: \tau}) = \mathtt{e}_b\ \triangleright \\ \mathrm{getmd}(\mathrm{M}', \mathtt{m}', \overline{\sigma}) = \triangleleft\ \mathtt{m}'(\overline{x' :: \tau'}) = \mathtt{e}'_b\ \triangleright \\ \overline{x : \sigma} \vdash^{\Phi}_{\mathrm{M} \rightsquigarrow \mathrm{M}'} \mathtt{e}_b \rightsquigarrow \mathtt{e}'_b[\overline{x'} \mapsto \overline{x}]\end{array}}{\vdash^{\Phi}_{\mathrm{M} \rightsquigarrow \mathrm{M}'} \mathtt{m}(\overline{\sigma}) \rightsquigarrow \mathtt{m}'}$$

OD-MD

$$\frac{\overline{x : \tau} \vdash^{\Phi}_{\mathrm{M} \rightsquigarrow \mathrm{M}'} \mathtt{e} \rightsquigarrow \mathtt{e}'[x' \mapsto x]}{\vdash^{\Phi}_{\mathrm{M} \rightsquigarrow \mathrm{M}'} \triangleleft\ \mathtt{m}(\overline{x :: \tau}) = \mathtt{e}\ \triangleright \rightsquigarrow \triangleleft\ \mathtt{m}(\overline{x' :: \tau}) = \mathtt{e}'\ \triangleright}$$

OT-MethodTable

$$\frac{\begin{array}{c}\mathrm{M} = \mathtt{md}_1 \bullet \ldots \bullet \mathtt{md}_n \qquad \mathrm{M}' = \mathtt{md}'_1 \bullet \ldots \bullet \mathtt{md}'_n \bullet \mathtt{md}'_{n+1} \bullet \ldots \bullet \mathtt{md}'_k \qquad \forall 1 \le i \le n.\ \vdash^{\Phi}_{\mathrm{M} \rightsquigarrow \mathrm{M}'} \mathtt{md}_i \rightsquigarrow \mathtt{md}'_i \\ \forall (\mathtt{m}(\overline{\sigma}) \rightsquigarrow \mathtt{m}') \in \Phi.\ \vdash^{\Phi}_{\mathrm{M} \rightsquigarrow \mathrm{M}'} \mathtt{m}(\overline{\sigma}) \rightsquigarrow \mathtt{m}' \qquad \forall n+1 \le j \le k.\ \mathit{name}(\mathtt{md}'_j) \text{ does not occur in M}\end{array}}{\vdash^{\Phi} \mathrm{M} \rightsquigarrow \mathrm{M}'}$$

MNamesCompat

$$\frac{\forall \mathtt{m} \text{ referenced by e.}\ \mathtt{m} \in \mathit{dom}(\mathrm{M}) \iff \mathtt{m} \in \mathit{dom}(\mathrm{M}')}{\vdash_{\mathrm{M} \rightsquigarrow \mathrm{M}'} \mathtt{e}}$$

OT-MethodTable-Expr

$$\frac{\vdash^{\Phi} \mathrm{M} \rightsquigarrow \mathrm{M}' \qquad \vdash_{\mathrm{M} \rightsquigarrow \mathrm{M}'} \mathtt{e}}{\vdash^{\Phi}_{\mathtt{e}} \mathrm{M} \rightsquigarrow \mathrm{M}'}$$

Fig. 18. Method table & definition optimization

$$v \quad ::= \quad \mathtt{v} \mid x \qquad \textit{Near-value}$$

OE-Val

$$\frac{\mathtt{v} \neq \mathtt{m}}{\Gamma \vdash^{\Phi}_{\mathrm{M} \rightsquigarrow \mathrm{M}'} \mathtt{v} \rightsquigarrow \mathtt{v}}$$

OE-ValFun

$$\frac{\vdash_{\mathrm{M} \rightsquigarrow \mathrm{M}'} \mathtt{m}}{\Gamma \vdash^{\Phi}_{\mathrm{M} \rightsquigarrow \mathrm{M}'} \mathtt{m} \rightsquigarrow \mathtt{m}}$$

OE-Var

$$\frac{}{\Gamma \vdash^{\Phi}_{\mathrm{M} \rightsquigarrow \mathrm{M}'} x \rightsquigarrow x}$$

OE-Global

$$\frac{\vdash_{\mathrm{M} \rightsquigarrow \mathrm{M}'} \mathtt{e}}{\Gamma \vdash^{\Phi}_{\mathrm{M} \rightsquigarrow \mathrm{M}'} (\!|\mathtt{e}|\!) \rightsquigarrow (\!|\mathtt{e}|\!)}$$

OE-Local

$$\frac{\vdash_{\mathrm{M} \rightsquigarrow \mathrm{M}'} \mathtt{e}}{\Gamma \vdash^{\Phi}_{\mathrm{M} \rightsquigarrow \mathrm{M}'} (\!|\mathtt{e}|\!)_{\mathrm{M}_l} \rightsquigarrow (\!|\mathtt{e}|\!)_{\mathrm{M}_l}}$$

OE-MD

$$\frac{\vdash_{\mathrm{M} \rightsquigarrow \mathrm{M}'} \mathit{name}(\mathtt{md})}{\Gamma \vdash^{\Phi}_{\mathrm{M} \rightsquigarrow \mathrm{M}'} \mathtt{md} \rightsquigarrow \mathtt{md}}$$

OE-Seq

$$\frac{\Gamma \vdash^{\Phi}_{\mathrm{M} \rightsquigarrow \mathrm{M}'} \mathtt{e}_1 \rightsquigarrow \mathtt{e}'_1 \qquad \Gamma \vdash^{\Phi}_{\mathrm{M} \rightsquigarrow \mathrm{M}'} \mathtt{e}_2 \rightsquigarrow \mathtt{e}'_2}{\Gamma \vdash^{\Phi}_{\mathrm{M} \rightsquigarrow \mathrm{M}'} \mathtt{e}_1\,;\,\mathtt{e}_2 \rightsquigarrow \mathtt{e}'_1\,;\,\mathtt{e}'_2}$$

OE-Primop

$$\frac{\forall i.\ \Gamma \vdash^{\Phi}_{\mathrm{M} \rightsquigarrow \mathrm{M}'} \mathtt{e}_i \rightsquigarrow \mathtt{e}'_i}{\Gamma \vdash^{\Phi}_{\mathrm{M} \rightsquigarrow \mathrm{M}'} \delta_l(\overline{\mathtt{e}}) \rightsquigarrow \delta_l(\overline{\mathtt{e}'})}$$

OE-Call

$$\frac{\Gamma \vdash^{\Phi}_{\mathrm{M} \rightsquigarrow \mathrm{M}'} \mathtt{e}_c \rightsquigarrow \mathtt{e}'_c \qquad \forall i.\ \Gamma \vdash^{\Phi}_{\mathrm{M} \rightsquigarrow \mathrm{M}'} \mathtt{e}_i \rightsquigarrow \mathtt{e}'_i}{\Gamma \vdash^{\Phi}_{\mathrm{M} \rightsquigarrow \mathrm{M}'} \mathtt{e}_c(\overline{\mathtt{e}}) \rightsquigarrow \mathtt{e}'_c(\overline{\mathtt{e}'})}$$

OE-Inline

$$\frac{\forall i.\ \Gamma \vdash v_i : \sigma_i \qquad \mathrm{getmd}(\mathrm{M}, \mathtt{m}, \overline{\sigma}) = \triangleleft\ \mathtt{m}(\overline{x :: \tau}) = \mathtt{e}_b\ \triangleright \qquad \Gamma \vdash^{\Phi}_{\mathrm{M} \rightsquigarrow \mathrm{M}'} \mathtt{e}_b[\overline{x} \mapsto \overline{v}] \rightsquigarrow \mathtt{e}'}{\Gamma \vdash^{\Phi}_{\mathrm{M} \rightsquigarrow \mathrm{M}'} \mathtt{m}(\overline{v}) \rightsquigarrow \mathtt{unit}\,;\,\mathtt{e}'}$$

OE-Direct

$$\frac{\forall i.\ \Gamma \vdash^{\Phi}_{\mathrm{M} \rightsquigarrow \mathrm{M}'} \mathtt{e}_i \rightsquigarrow \mathtt{e}'_i \qquad \Gamma \vdash \mathtt{e}'_i : \sigma_i \qquad (\mathtt{m}(\overline{\sigma}) \rightsquigarrow \mathtt{m}') \in \Phi}{\Gamma \vdash^{\Phi}_{\mathrm{M} \rightsquigarrow \mathrm{M}'} \mathtt{m}(\overline{\mathtt{e}}) \rightsquigarrow \mathtt{m}'(\overline{\mathtt{e}'})}$$

Fig. 19. Expression optimization

OE-Inline and OE-Direct correspond to the inlining and the direct-call/specialization optimizations, respectively. As discussed earlier, inlining cannot be done if a function is called with expression arguments. Therefore, in OE-Inline we use an auxiliary definition v, "near-value", which is either a

$$\gamma \quad ::= \quad \overline{x \mapsto \mathsf{v}} \text{ where } \forall i, j . x_i \neq x_j \qquad \textit{Value substitution}$$

$$\frac{dom(\Gamma) = dom(\gamma) \qquad \forall x \in dom(\gamma).\ (\Gamma(x) = \sigma \iff \text{typeof}(\gamma(x)) = \sigma)}{\Gamma \vdash \gamma}\ \gamma\text{-Ok}$$

Fig. 20. Value substitution

value or a variable. Because we model direct-call and specialization optimizations as calls to freshly-named methods, the main job is done in the table-optimization rule OT-MethodTable; rule OE-Direct only records the fact of invoking a specific method. If the method definition of $\mathtt{m}'$ from $\mathtt{m}(\overline{\sigma}) \rightsquigarrow \mathtt{m}'$ has the same parameter-type annotations as $\mathtt{m}$, it represents a direct call to an original method of $\mathtt{m}$; otherwise, it represents a specialized method. Note that for all optimizations, function-call arguments have to be concretely typed. Otherwise, we do not know definitively how a function call is going to be dispatched at run time.

The optimizations defined in Fig. 18–19 are sound. That is, the evaluation of the original and optimized programs yield the same result. To show this, we establish a bisimulation relation between original and optimized expressions (after the following auxiliary lemmas):

Lemma 4.4 (Context Irrelevance). *For all* $\mathtt{C}$, $\mathtt{C}'$, $\mathtt{rdx}$, $\mathtt{e}'$, M_g, M'_g, *the following holds:*

$$\langle \mathrm{M}_g, \mathtt{C}\,[\mathtt{rdx}] \rangle \to \langle \mathrm{M}'_g, \mathtt{C}\,[\mathtt{e}'] \rangle \iff \langle \mathrm{M}_g, \mathtt{C}'\,[\mathtt{rdx}] \rangle \to \langle \mathrm{M}'_g, \mathtt{C}'\,[\mathtt{e}'] \rangle.$$

Proof. By analyzing normal-evaluation steps, we can see that only $\mathtt{rdx}$ matters for the reduction. Formally, the proof goes by inspecting a reduction step for $\mathtt{C}\,[\mathtt{rdx}]$ ($\mathtt{C}'\,[\mathtt{rdx}]$) and building a corresponding step for $\mathtt{C}'\,[\mathtt{rdx}]$ ($\mathtt{C}\,[\mathtt{rdx}]$). □

Lemma 4.5 (Simple-Context Irrelevance). *For all* M, $\mathtt{C}$, $\mathtt{e}$, M_g, $\mathtt{e}'$, M'_g, $\mathtt{X}$, *the following holds:*

$$\langle \mathrm{M}_g, \mathtt{C}\,[\!(\,\mathtt{e}\,)\!]_{\mathrm{M}} \rangle \to \langle \mathrm{M}'_g, \mathtt{C}\,[\!(\,\mathtt{e}'\,)\!]_{\mathrm{M}} \rangle \implies \langle \mathrm{M}_g, \mathtt{C}\,[\!(\,\mathtt{X}\,[\mathtt{e}]\,)\!]_{\mathrm{M}} \rangle \to \langle \mathrm{M}'_g, \mathtt{C}\,[\!(\,\mathtt{X}\,[\mathtt{e}']\,)\!]_{\mathrm{M}} \rangle.$$

Proof. By Lemma 4.1, $\mathtt{e}$ is either $\mathtt{v}$ or $\mathtt{X}_e\,[\mathtt{m}(\overline{\mathtt{v}})]$ or $\mathtt{C}_e\,[\mathtt{rdx}]$. If $\mathtt{e}$ is $\mathtt{v}$, the assumption of the lemma does not hold ($\mathtt{C}\,[\!(\,\mathtt{v}\,)\!]_{\mathrm{M}}$ would step to $\mathtt{C}\,[\mathtt{v}]$), so only $\mathtt{X}_e\,[\mathtt{m}(\overline{\mathtt{v}})]$ and $\mathtt{C}_e\,[\mathtt{rdx}]$ cases are possible.

- When $\mathtt{e}$ is $\mathtt{X}_e\,[\mathtt{m}(\overline{\mathtt{v}})]$, $[\!(\,\mathtt{e}\,)\!]_{\mathrm{M}} = [\!(\,\mathtt{X}_e\,[\mathtt{m}(\overline{\mathtt{v}})]\,)\!]_{\mathrm{M}}$ is a redex, and $\mathtt{C}\,[\!(\,\mathtt{e}\,)\!]_{\mathrm{M}}$ steps by rule E-CallLocal. But $[\!(\,\mathtt{X}\,[\mathtt{e}]\,)\!]_{\mathrm{M}} = [\!(\,\mathtt{X}\,[\mathtt{X}_e\,[\mathtt{m}(\overline{\mathtt{v}})]]\,)\!]_{\mathrm{M}}$ is also a redex, and $\mathtt{C}\,[\!(\,\mathtt{X}\,[\mathtt{e}]\,)\!]_{\mathrm{M}}$ steps by rule E-CallLocal similarly to $\mathtt{C}\,[\!(\,\mathtt{e}\,)\!]_{\mathrm{M}}$.
- When $\mathtt{e}$ is $\mathtt{C}_e\,[\mathtt{rdx}]$, $[\!(\,\mathtt{e}\,)\!]_{\mathrm{M}} = [\!(\,\mathtt{C}_e\,[\mathtt{rdx}]\,)\!]_{\mathrm{M}}$ and $\mathtt{C}\,[\!(\,\mathtt{e}\,)\!]_{\mathrm{M}} = \mathtt{C}'\,[\mathtt{rdx}]$ where $\mathtt{C}' = \mathtt{C}\,[\!(\,\mathtt{C}_e\,)\!]_{\mathrm{M}}$. Since $\mathtt{C}\,[\!(\,\mathtt{X}\,[\mathtt{e}]\,)\!]_{\mathrm{M}} = \mathtt{C}''\,[\mathtt{rdx}]$ for $\mathtt{C}'' = \mathtt{C}\,[\!(\,\mathtt{X}\,[\mathtt{C}_e]\,)\!]_{\mathrm{M}}$, $\mathtt{C}\,[\!(\,\mathtt{e}\,)\!]_{\mathrm{M}}$ and $\mathtt{C}\,[\!(\,\mathtt{X}\,[\mathtt{e}]\,)\!]_{\mathrm{M}}$ step similarly by Lemma 4.4.

□

Lemma 4.6 (Optimization Preserves Values). *For all* Φ, M, M', Γ, $\mathtt{e}$, $\mathtt{v}$, *the following hold:*

$$\Gamma \vdash^{\Phi}_{\mathrm{M} \rightsquigarrow \mathrm{M}'} \mathtt{v} \rightsquigarrow \mathtt{e} \implies \mathtt{e} = \mathtt{v} \qquad \textit{and} \qquad \Gamma \vdash^{\Phi}_{\mathrm{M} \rightsquigarrow \mathrm{M}'} \mathtt{e} \rightsquigarrow \mathtt{v} \implies \mathtt{e} = \mathtt{v}.$$

Proof. By case analysis on the optimization relation. □

Lemma 4.7 (Value Substitution Preserves Optimization). *For all* $\Phi, \Gamma, \mathtt{e}, \mathrm{M}, \mathtt{e}', \mathrm{M}', \gamma$, *such that* $\forall \mathtt{v} \in \gamma.\ \vdash_{\mathrm{M} \rightsquigarrow \mathrm{M}'} \mathtt{v}$, *the following holds:*

$$\left(\Gamma \vdash^{\Phi}_{\mathrm{M} \rightsquigarrow \mathrm{M}'} \mathtt{e} \rightsquigarrow \mathtt{e}' \ \wedge\ \Gamma \vdash \gamma\right) \implies \vdash^{\Phi}_{\mathrm{M} \rightsquigarrow \mathrm{M}'} \gamma(\mathtt{e}) \rightsquigarrow \gamma(\mathtt{e}').$$

Proof. By induction on the derivation of $\Gamma \vdash^{\Phi}_{\mathrm{M} \rightsquigarrow \mathrm{M}'} \mathtt{e} \rightsquigarrow \mathtt{e}'$. □

Lemma 4.8 (Bisimulation). *For all method tables* M, M'*, method-optimization environment* Φ*, and expressions* $\mathsf{e}_1, \mathsf{e}_1'$*, such that*

$$\vdash^{\Phi} \mathrm{M} \rightsquigarrow \mathrm{M}' \quad \textit{and} \quad \vdash^{\Phi}_{\mathrm{M}\rightsquigarrow\mathrm{M}'} \mathsf{e}_1 \rightsquigarrow \mathsf{e}_1',$$

for all global tables $\mathrm{M}_g, \mathrm{M}_g'$ *and world context* C*, the following hold:*

(1) Forward direction:

$$\begin{aligned}\forall \mathsf{e}_2.\ & \langle \mathrm{M}_g, \mathsf{C}\left[\!\left(\mathsf{e}_1\right)\!\right]_{\mathrm{M}}\rangle \ \to\ \langle \mathrm{M}_g', \mathsf{C}\left[\!\left(\mathsf{e}_2\right)\!\right]_{\mathrm{M}}\rangle \\ & \Longrightarrow \\ & \exists \mathsf{e}_2'.\ \langle \mathrm{M}_g, \mathsf{C}\left[\!\left(\mathsf{e}_1'\right)\!\right]_{\mathrm{M}'}\rangle \ \to\ \langle \mathrm{M}_g', \mathsf{C}\left[\!\left(\mathsf{e}_2'\right)\!\right]_{\mathrm{M}'}\rangle \ \wedge\ \vdash^{\Phi}_{\mathrm{M}\rightsquigarrow\mathrm{M}'} \mathsf{e}_2 \rightsquigarrow \mathsf{e}_2'.\end{aligned}$$

(2) Backward direction:

$$\begin{aligned}\forall \mathsf{e}_2'.\ & \langle \mathrm{M}_g, \mathsf{C}\left[\!\left(\mathsf{e}_1'\right)\!\right]_{\mathrm{M}'}\rangle \ \to\ \langle \mathrm{M}_g', \mathsf{C}\left[\!\left(\mathsf{e}_2'\right)\!\right]_{\mathrm{M}'}\rangle \\ & \Longrightarrow \\ & \exists \mathsf{e}_2.\ \langle \mathrm{M}_g, \mathsf{C}\left[\!\left(\mathsf{e}_1\right)\!\right]_{\mathrm{M}}\rangle \ \to\ \langle \mathrm{M}_g', \mathsf{C}\left[\!\left(\mathsf{e}_2\right)\!\right]_{\mathrm{M}}\rangle \ \wedge\ \vdash^{\Phi}_{\mathrm{M}\rightsquigarrow\mathrm{M}'} \mathsf{e}_2 \rightsquigarrow \mathsf{e}_2'.\end{aligned}$$

Proof. The proof goes by induction on the derivation of optimization $\vdash^{\Phi}_{\mathrm{M}\rightsquigarrow\mathrm{M}'} \mathsf{e}_1 \rightsquigarrow \mathsf{e}_1'$. For each case, both directions are proved by analyzing possible normal-evaluation steps. More specifically, the forward-direction proof strategy is as follows (the backward direction is similar):

(1) Observe that to make the required step, e_1 should have a certain representation. Consider all possible representations that satisfy this requirement.
(2) For each representation, analyze the suitable normal-evaluation rule (recall that the semantics is deterministic, so there will be just one such rule).
(3) If e_1 represents an immediate redex (e.g. $\mathsf{v}_{11}\,;\,\mathsf{e}_{12}$), the optimized expression will be an immediate redex too (possibly, of a different form). Otherwise, use induction hypothesis and auxiliary facts about contexts and evaluation to show that the optimized expression steps in a similar fashion, in particular, facts from Lemma 4.4 and Lemma 4.5.
(4) Finally, show that the resulting expressions are in the optimization relation. This will follow from the assumptions and induction.

As an example, consider the proof of the forward direction for the sequence case **OE-Seq**. By assumption, we have $\mathsf{e}_1 = \mathsf{e}_{11}\,;\,\mathsf{e}_{12}$ and $\mathsf{e}_1' = \mathsf{e}_{11}'\,;\,\mathsf{e}_{12}'$ where

$$\frac{\vdash^{\Phi}_{\mathrm{M}\rightsquigarrow\mathrm{M}'} \mathsf{e}_{11} \rightsquigarrow \mathsf{e}_{11}' \qquad \Gamma \ \vdash^{\Phi}_{\mathrm{M}\rightsquigarrow\mathrm{M}'} \mathsf{e}_{12} \rightsquigarrow \mathsf{e}_{12}'}{\vdash^{\Phi}_{\mathrm{M}\rightsquigarrow\mathrm{M}'} \mathsf{e}_{11}\,;\,\mathsf{e}_{12} \rightsquigarrow \mathsf{e}_{11}'\,;\,\mathsf{e}_{12}'}\ \textsc{OE-Seq}.$$

For $\mathsf{C}\left[\!\left(\mathsf{e}_{11}\,;\,\mathsf{e}_{12}\right)\!\right]_{\mathrm{M}}$ to reduce, by case analysis, we know there are three possibilities.

(1) $\mathsf{e}_{11} = \mathsf{v}_{11}$ and $\langle \mathrm{M}_g, \mathsf{C}\left[\!\left(\mathsf{v}_{11}\,;\,\mathsf{e}_{12}\right)\!\right]_{\mathrm{M}}\rangle \ \to\ \langle \mathrm{M}_g, \mathsf{C}\left[\!\left(\mathsf{e}_{12}\right)\!\right]_{\mathrm{M}}\rangle$ by rule E-Seq. Then by Lemma 4.6, $\mathsf{e}_{11}' = \mathsf{v}_{11}$ and the optimized expression steps by the same rule:

$$\langle \mathrm{M}_g, \mathsf{C}\left[\!\left(\mathsf{v}_{11}\,;\,\mathsf{e}_{12}'\right)\!\right]_{\mathrm{M}'}\rangle \ \to\ \langle \mathrm{M}_g, \mathsf{C}\left[\!\left(\mathsf{e}_{12}'\right)\!\right]_{\mathrm{M}'}\rangle.$$

The desired optimization relation holds by one of the assumptions: $\Gamma \ \vdash^{\Phi}_{\mathrm{M}\rightsquigarrow\mathrm{M}'} \mathsf{e}_{12} \rightsquigarrow \mathsf{e}_{12}'$.

(2) $\mathsf{e}_{11} = \mathsf{X}_1\,[\mathsf{m}(\overline{\mathsf{v}})]$ and the original expression steps by E-CallLocal:

$$\langle \mathrm{M}_g, \mathsf{C}\left[\!\left(\mathsf{X}_1\,[\mathsf{m}(\overline{\mathsf{v}})]\ ;\ \mathsf{e}_{12}\right)\!\right]_{\mathrm{M}}\rangle \ \to\ \langle \mathrm{M}_g, \mathsf{C}\left[\!\left(\mathsf{X}_1\,[\mathsf{e}_b[\overline{x}\mapsto\overline{\mathsf{v}}]]\ ;\ \mathsf{e}_{12}\right)\!\right]_{\mathrm{M}}\rangle.$$

Since $\mathsf{C}\left[\!\left(\mathsf{X}_1\,[\mathsf{m}(\overline{\mathsf{v}})]\right)\!\right]_{\mathrm{M}}$ reduces similarly, by the induction hypothesis, $\exists\, \mathsf{e}_{21}'$ such that

$$\langle \mathrm{M}_g, \mathsf{C}\left[\!\left(\mathsf{e}_{11}'\right)\!\right]_{\mathrm{M}'}\rangle \ \to\ \langle \mathrm{M}_g, \mathsf{C}\left[\!\left(\mathsf{e}_{21}'\right)\!\right]_{\mathrm{M}'}\rangle \quad \text{and} \quad \Gamma \ \vdash^{\Phi}_{\mathrm{M}\rightsquigarrow\mathrm{M}'} \mathsf{X}_1\,[\mathsf{e}_b[\overline{x}\mapsto\overline{\mathsf{v}}]] \rightsquigarrow \mathsf{e}_{21}'.$$

But then, by Lemma 4.5, the entire optimized expression $\mathtt{C}\left[\!\left(\mathtt{e}'_{11}\ ;\ \mathtt{e}'_{12}\right)\!\right]_{\mathrm{M}}$ steps too, and the desired optimization relation holds:

$$\frac{\vdash^{\Phi}_{\mathrm{M}\rightsquigarrow\mathrm{M}'}\ \mathtt{X}_1\ [\mathtt{e}_b[\overline{x}\mapsto\overline{\mathtt{v}}]] \rightsquigarrow \mathtt{e}'_{21} \qquad \vdash^{\Phi}_{\mathrm{M}\rightsquigarrow\mathrm{M}'}\ \mathtt{e}_{12} \rightsquigarrow \mathtt{e}'_{12}}{\vdash^{\Phi}_{\mathrm{M}\rightsquigarrow\mathrm{M}'}\ \mathtt{X}_1\ [\mathtt{e}_b[\overline{x}\mapsto\overline{\mathtt{v}}]]\ ;\ \mathtt{e}_{12} \rightsquigarrow \mathtt{e}'_{21}\ ;\ \mathtt{e}'_{12}}\ \textsc{OE-Seq}.$$

(3) $\mathtt{e}_{11} = \mathtt{C}_1\ [\mathtt{rdx}]$ and by Lemma 4.4:

$$\langle \mathrm{M}_g, \mathtt{C}\left[\!\left(\mathtt{C}_1\ [\mathtt{rdx}]\ ;\ \mathtt{e}_{12}\right)\!\right]_{\mathrm{M}}\rangle \ \to\ \langle \mathrm{M}'_g, \mathtt{C}\left[\!\left(\mathtt{C}_1\ [\mathtt{e}']\ ;\ \mathtt{e}_{12}\right)\!\right]_{\mathrm{M}}\rangle$$
$$\Longleftrightarrow$$
$$\langle \mathrm{M}_g, \mathtt{C}\left[\!\left(\mathtt{C}_1\ [\mathtt{rdx}]\right)\!\right]_{\mathrm{M}}\rangle \ \to\ \langle \mathrm{M}'_g, \mathtt{C}\left[\!\left(\mathtt{C}_1\ [\mathtt{e}']\right)\!\right]_{\mathrm{M}}\rangle.$$

Since $\mathtt{C}\left[\!\left(\mathtt{C}_1\ [\mathtt{rdx}]\right)\!\right]_{\mathrm{M}}$ reduces, by the induction hypothesis, $\exists\ \mathtt{e}'_{21}$ such that

$$\langle \mathrm{M}_g, \mathtt{C}\left[\!\left(\mathtt{e}'_{11}\right)\!\right]_{\mathrm{M}'}\rangle \ \to\ \langle \mathrm{M}_g, \mathtt{C}\left[\!\left(\mathtt{e}'_{21}\right)\!\right]_{\mathrm{M}'}\rangle \quad \text{and} \quad \Gamma\ \vdash^{\Phi}_{\mathrm{M}\rightsquigarrow\mathrm{M}'}\ \mathtt{C}_1\ [\mathtt{e}'] \rightsquigarrow \mathtt{e}'_{21}.$$

Similarly to the previous case, the entire $\mathtt{C}\left[\!\left(\mathtt{e}'_{11}\ ;\ \mathtt{e}'_{12}\right)\!\right]_{\mathrm{M}}$ steps, and the desired optimization relation holds.

□

Lemma 4.9 (Reflexivity of Optimization). *For all* $\mathrm{M}, \mathrm{M}', \Phi, \Gamma, \mathtt{e}$*, the following holds:*

$$\vdash^{\Phi}_{\mathtt{e}}\ \mathrm{M} \rightsquigarrow \mathrm{M}' \quad\Longrightarrow\quad \Gamma\ \vdash^{\Phi}_{\mathrm{M}\rightsquigarrow\mathrm{M}'}\ \mathtt{e} \rightsquigarrow \mathtt{e}.$$

Proof. By induction on $\mathtt{e}$. The only interesting cases are $\mathtt{m}$, $\mathtt{md}$, $\left(\!\left|\,\mathtt{e}'\,\right|\!\right)$, and $\left(\!\left|\,\mathtt{e}'\,\right|\!\right)_{\mathrm{M}_l}$. For example, consider the case of $\mathtt{m}$ (others are similar). Rule OE-ValFun requires a method named $\mathtt{m}$ to either exist in both tables or do not appear in either (this rules out the case where $\left(\!\left|\,\mathtt{m}()\,\right|\!\right)_{\mathrm{M}}$ would err but $\left(\!\left|\,\mathtt{m}()\,\right|\!\right)_{\mathrm{M}'}$ succeed). This requirement is guaranteed by the assumption that $\vdash^{\Phi}_{\mathtt{e}}\ \mathrm{M} \rightsquigarrow \mathrm{M}'$, which by inversion, gives the necessary $\vdash_{\mathrm{M}\rightsquigarrow\mathrm{M}'}\ \mathtt{e}$. □

The main result, Theorem 4.10, is a corollary of Lemma 4.8. It states that a fixed-table expression can be soundly evaluated in an optimized table.

Theorem 4.10 (Correctness of Table Optimization). *For all* $\mathrm{M}, \mathrm{M}', \Phi$*, e satisfying* $\vdash^{\Phi}_{\mathtt{e}}\ \mathrm{M} \rightsquigarrow \mathrm{M}'$*, for all* $\mathrm{M}_g, \mathrm{M}'_g, \mathtt{C}, \mathtt{v}$*, the following holds:*

$$\langle \mathrm{M}_g, \mathtt{C}\left[\!\left(\mathtt{e}\right)\!\right]_{\mathrm{M}}\rangle \ \to^*\ \langle \mathrm{M}'_g, \mathtt{v}\rangle \iff \langle \mathrm{M}_g, \mathtt{C}\left[\!\left(\mathtt{e}\right)\!\right]_{\mathrm{M}'}\rangle \ \to^*\ \langle \mathrm{M}'_g, \mathtt{v}\rangle.$$

Proof. First of all, note that $\vdash^{\Phi}_{\mathrm{M}\rightsquigarrow\mathrm{M}'}\ \mathtt{e} \rightsquigarrow \mathtt{e}$ by Lemma 4.9, and that $\vdash^{\Phi}\ \mathrm{M} \rightsquigarrow \mathrm{M}'$ follows from $\vdash^{\Phi}_{\mathtt{e}}\ \mathrm{M} \rightsquigarrow \mathrm{M}'$. Then, we proceed by induction on $\to^*$ (reflexive-transitive closure of normal evaluation). In the interesting case of the forward direction, we have:

$$\frac{\langle \mathrm{M}_g, \mathtt{C}\left[\!\left(\mathtt{e}\right)\!\right]_{\mathrm{M}}\rangle \ \to\ \langle \mathrm{M}''_g, \mathtt{C}\left[\!\left(\mathtt{e}'_1\right)\!\right]_{\mathrm{M}}\rangle \qquad \langle \mathrm{M}''_g, \mathtt{C}\left[\!\left(\mathtt{e}'_1\right)\!\right]_{\mathrm{M}}\rangle \ \to^*\ \langle \mathrm{M}'_g, \mathtt{v}\rangle}{\langle \mathrm{M}_g, \mathtt{C}\left[\!\left(\mathtt{e}\right)\!\right]_{\mathrm{M}}\rangle \ \to^*\ \langle \mathrm{M}'_g, \mathtt{v}\rangle}.$$

By applying Lemma 4.8 to the first premise, we get:

$$\langle \mathrm{M}_g, \mathtt{C}\left[\!\left(\mathtt{e}\right)\!\right]_{\mathrm{M}'}\rangle \ \to\ \langle \mathrm{M}''_g, \mathtt{C}\left[\!\left(\mathtt{e}'_2\right)\!\right]_{\mathrm{M}'}\rangle \quad \text{and} \quad \vdash^{\Phi}_{\mathrm{M}\rightsquigarrow\mathrm{M}'}\ \mathtt{e}'_1 \rightsquigarrow \mathtt{e}'_2.$$

By applying the induction hypothesis to the second premise, we get:

$$\langle \mathrm{M}''_g, \mathtt{C}\left[\!\left(\mathtt{e}'_2\right)\!\right]_{\mathrm{M}'}\rangle \ \to^*\ \langle \mathrm{M}'_g, \mathtt{v}\rangle.$$

By combining the results, we can get the desired derivation:

$$\frac{\langle \mathrm{M}_g, \mathtt{C}\left[\!\left(\mathtt{e}\right)\!\right]_{\mathrm{M}'}\rangle \ \to\ \langle \mathrm{M}''_g, \mathtt{C}\left[\!\left(\mathtt{e}'_2\right)\!\right]_{\mathrm{M}'}\rangle \qquad \langle \mathrm{M}''_g, \mathtt{C}\left[\!\left(\mathtt{e}'_2\right)\!\right]_{\mathrm{M}'}\rangle \ \to^*\ \langle \mathrm{M}'_g, \mathtt{v}\rangle}{\langle \mathrm{M}_g, \mathtt{C}\left[\!\left(\mathtt{e}\right)\!\right]_{\mathrm{M}'}\rangle \ \to^*\ \langle \mathrm{M}'_g, \mathtt{v}\rangle}.$$

The backward direction proceeds similarly. □

Theorem 4.10, in particular, justifies Julia's choice to execute top-level calls using optimized methods. Once a top-level call $\langle \mathrm{M}, (\!| \mathtt{m}(\overline{\mathtt{v}}) |\!) \rangle$ steps to a fixed-table call $\langle \mathrm{M}, (\!| \mathtt{m}(\overline{\mathtt{v}}) |\!)_{\mathrm{M}} \rangle$, it is sound to optimize table M into M′ (using inlining, direct calls, and specialization), and evaluate the call in the optimized table $\langle \mathrm{M}, (\!| \mathtt{m}(\overline{\mathtt{v}}) |\!)_{\mathrm{M}'} \rangle$.

4.7 Testing the Semantics

To check if Juliette behaves as we expect, we implemented it in Redex [Felleisen et al. 2009] and ran it along with Julia on a small set of 9 litmus tests (provided in App. A); Julia agrees with Juliette on all of them. The tests cover the intersection of the semantics of Juliette and Julia, and demonstrate the interaction of `eval`, method definitions, and method calls. In particular, the litmus tests ensure: that the executing semantics prohibits calls to too-new methods, that this restriction can be skipped with `eval` or `invokelatest`, and that the semantics of `eval` executes successive statements in the latest age.

Two of the litmus tests are shown in Fig. 21; each test is made up of a small program and its expected output. The tests examine the case where a method `r2` is placed "in between" the generated method `r1` and an older `m`. In the first test, `m` errs. While `r2` is callable from the age that `m` was called in, `r1` is not. In the second test, we use `invokelatest` to execute `r2` in the latest world age; this allows the invocation of the dynamically generated `r1`.

```
r2() = r1()
m()  = (
  eval(:(r1() = 2));
  r2())
m() # error
```

```
r2() = r1()
m()  = (
  eval(:(r1() = 2));
  Base.invokelatest(r2))
m() == 2 # passes
```

Fig. 21. Litmus tests

To use the litmus tests, we need to (1) translate them from Julia into our grammar and (2) implement the semantics of Juliette into an executable form. The former is done by translating ASTs. The latter is realized with a Redex mechanization, which is publicly available on GitHub[5] along with the litmus tests. The model implements the calculus almost literally. Values, tags, and type annotations are instantiated with several concrete examples, such as numbers and strings. Primitive operations include arithmetic and `print`. The only difference between the paper and Redex is handling of function names. Similar to Julia, in the Redex model, a definition of the method named `f` introduces a global constant f. When referenced, the constant evaluates to a function value `f`. Thus, instead of a single error evaluation rule E-Var from Fig. 13, the Redex model has the following two rules, one for normal evaluation and one for erroneous evaluation:

$$\textsc{E-VarMethod}\ \frac{x \in dom(\mathrm{M})}{\langle \mathrm{M}, \mathtt{C}\,[x] \rangle \longrightarrow \langle \mathrm{M}, \mathtt{C}\,[\mathtt{x}] \rangle} \qquad \textsc{E-VarErr}\ \frac{x \notin dom(\mathrm{M})}{\mathrm{M} \vdash \mathtt{C}\,[x] \longrightarrow \mathtt{error}.}$$

The new rules treat the global method table as a global environment: E-VarMethod evaluates a global variable to its underlying function value, and E-VarErr errs if a variable is not found in the global environment; all local variables should be eliminated by substitution. All paper-style programs can be written in the Redex model, and the extension makes it easier to compare and translate Julia programs to corresponding Redex programs. Thus, the litmus test on the left of Fig. 21 translates to the Redex model as follows (the grammar is written in S-expressions style):

[5] https://github.com/julbinb/juliette-wa

```
(evalg (seq (seq
  (mdef "r2" () (mcall r1)) # r2() = r1()
  (mdef "m"  () (seq
                  (evalg (mdef "r1" () 2)) # eval(:(r1() = 2))
                  (mcall r2))))            # r2()
  (mcall m))) # m()
```

The Redex model also implements the optimization judgments presented in Sec. 4.5, as well as a straightforward optimization algorithm that is checked against the judgments.

Discussion with Julia's developers confirmed that our understanding of world age is correct, and that the table-based semantics has a correspondence to the age-based implementation. Namely, it is possible to generate Juliette method tables from the global data structure used by Julia to store methods.

5 CONCLUSION

Julia's approach to dynamic code loading is distinct; instead of striving to achieve performance *in spite of* the language's semantics, the designers of Julia chose to restrict expressiveness so that they could keep their compiler simple *and* generate fast code. World age aligns Julia's dynamic semantics with its just-in-time compiler's static approximation. As a result, statically resolved function calls have the same behavior as dynamic invocations.

This equivalence—that statically and dynamically resolved methods behave the same—allows Julia to forsake some of the complexity of modern compilers. Instead of needing deoptimization to handle newly added definitions, Julia simply does not allow running code to see those definitions. Thus, optimizations can rely on the results of static reasoning about the method table, while remaining sound in the presence of `eval`. If necessary, the programmer can explicitly ask for newly defined methods, making the performance penalty explicit and user-controllable.

World age need not be limited to Julia. Any language that supports updating existing function definitions may benefit from such a mechanism, namely control over when those new definitions can be observed and when function calls can be optimized. From Java to languages like R, having a clear semantics for updating code, especially in the presence of concurrency, can be beneficial, as it would improve our ability to reason about programs written in those languages.

Although the world-age semantics presented in the paper follows Julia, *a* world-age semantics does not have to. For instance, an alternative world-age semantics could pick another point when the age counter is incremented. The notion of top level makes sense in the context of an interactive development environment, but is unclear in, for example, a web server that may receive new code to install from time to time. Such a continuously running system may need a definition of quiescence that is different from the top-level used in Julia. One alternative is to provide an explicit `freeze` construct that allows programmers to opt-in to the world-age system. This would allow existing languages to incorporate world age without affecting existing code.

The calculus we present here is a basic foundation intended to capture the operation of world age. Future work may build on this to formalize the semantics of Julia as a whole, but, notably, the additional semantics will not impact the world-age mechanism itself. Of particular note is mutable state: it is orthogonal to world age because Julia decouples code state from data state by design. This was a pragmatic decision, as the compiler depends on knowing the contents of the method table for its optimization. Optimizations based on global variables are much less frequent.

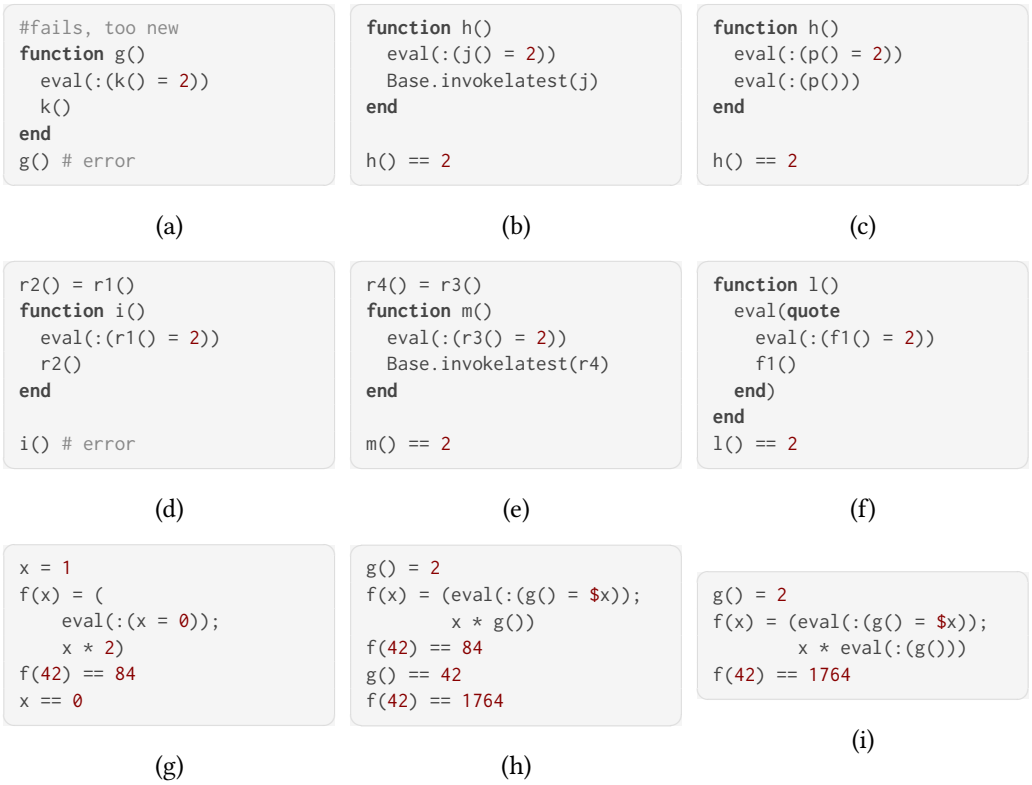

Fig. 22. Litmus Tests

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their insightful comments and suggestions to improve this paper. This work was supported by Office of Naval Research (ONR) award 503353, the National Science Foundation awards 1759736, 1925644 and 1618732, the Czech Ministry of Education from the Czech Operational Programme Research, Development, and Education, under grant agreement No. CZ.02.1.01/0.0/0.0/15_003/0000421, and the European Research Council under the European Union's Horizon 2020 research and innovation programme, under grant agreement No. 695412.

A LITMUS TESTS

As a basic test of functionality, we provide 9 litmus tests shown in Fig. 22, written in Julia, that exercise the basic world age semantics as well as key Julia semantics surrounding world age. The tests suffice to identify the following semantic characteristics:

(a) too-new methods cannot be called using a normal invocation;
(b) `invokelatest` uses the latest world age;
(c) `eval` uses the latest world age;
(d) successive `eval` statements run in the latest world age;
(e) only age at the top-level is relevant for invocation visibility;
(f) "latest" calls propagate the new world age;
(g) `eval` executes in the top-level scope;

(h) normal invocation uses overridden methods if added method too new;
(i) `eval` will use latest definition of an overridden method.

REFERENCES

Julia Belyakova, Benjamin Chung, Jack Gelinas, Jameson Nash, Ross Tate, and Jan Vitek. 2020. World Age in Julia: Optimizing Method Dispatch in the Presence of Eval (Extended Version). arXiv:2010.07516

Jeff Bezanson, Jiahao Chen, Ben Chung, Stefan Karpinski, Viral B. Shah, Jan Vitek, and Lionel Zoubritzky. 2018. Julia: Dynamism and Performance Reconciled by Design. *Proc. ACM Program. Lang.* 2, OOPSLA (2018). https://doi.org/10.1145/3276490

Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B. Shah. 2017. Julia: A Fresh Approach to Numerical Computing. *SIAM Rev.* 59, 1 (2017). https://doi.org/10.1137/141000671

Daniel G. Bobrow, Kenneth Kahn, Gregor Kiczales, Larry Masinter, Mark Stefik, and Frank Zdybel. 1986. CommonLoops: Merging Lisp and Object-oriented Programming. In *Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*. https://doi.org/10.1145/28697.28700

Robert P. Cook and Insup Lee. 1983. DYMOS: A Dynamic Modification System. In *Proceedings of the Symposium on High-Level Debugging*. https://doi.org/10.1145/1006147.1006188

David Detlefs and Ole Agesen. 1999. Inlining of Virtual Methods. In *European Conference on Object-Oriented Programming (ECOOP)*. https://doi.org/10.5555/646156.679839

Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. 2009. *Semantics Engineering with PLT Redex*. MIT Press. http://mitpress.mit.edu/catalog/item/default.asp?ttype=2&tid=11885

Olivier Flückiger, Gabriel Scherer, Ming-Ho Yee, Aviral Goel, Amal Ahmed, and Jan Vitek. 2018. Correctness of speculative optimizations with dynamic deoptimization. *Proc. ACM Program. Lang.* 2, POPL (2018). https://doi.org/10.1145/3158137

Neal Glew. 2005. Method Inlining, Dynamic Class Loading, and Type Soundness. *Journal of Object Technology* 4, 8 (2005). https://doi.org/10.5381/jot.2005.4.8.a2

Urs Hölzle, Craig Chambers, and David Ungar. 1992. Debugging Optimized Code with Dynamic Deoptimization, In Conference on Programming Language Design and Implementation (PLDI). https://doi.org/10.1145/143103.143114

Sheng Liang and Gilad Bracha. 1998. Dynamic class loading in the Java virtual machine. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. https://doi.org/10.1145/286936.286945

Jacob Matthews and Robert Bruce Findler. 2008. An operational semantics for Scheme. *Journal of Functional Programming* 18 (2008). Issue 1. https://doi.org/10.1017/S0956796807006478

John McCarthy. 1978. History of LISP. In *History of programming languages (HOPL)*. https://doi.org/10.1145/960118.808387

Phung Hua Nguyen and Jingling Xue. 2005. Interprocedural side-effect analysis and optimisation in the presence of dynamic class loading. In *Australasian Conference on Computer Science (ACSC)*. https://doi.org/10.5555/1082161.1082163

Joe Gibbs Politz, Matthew J. Carroll, Benjamin S. Lerner, Justin Pombrio, and Shriram Krishnamurthi. 2012. A Tested Semantics for Getters, Setters, and Eval in JavaScript. In *Symposium on Dynamic Languages (DLS)*. https://doi.org/10.1145/2384577.2384579

Gareth Stoyle, Michael Hicks, Gavin Bierman, Peter Sewell, and Iulian Neamtiu. 2007. Mutatis Mutandis: Safe and Predictable Dynamic Software Updating. *ACM Trans. Program. Lang. Syst.* 29, 4 (2007). https://doi.org/10.1145/1255450.1255455

Julia Language Manual v1. 2020. *Redefining Methods*. https://docs.julialang.org/en/v1/manual/methods/#Redefining-Methods-1

Francesco Zappa Nardelli, Julia Belyakova, Artem Pelenitsyn, Benjamin Chung, Jeff Bezanson, and Jan Vitek. 2018. Julia Subtyping: A Rational Reconstruction. *Proc. ACM Program. Lang.* 2, OOPSLA (2018). https://doi.org/10.1145/3276483