World Age in Julia

Optimizing Method Dispatch in the Presence of Eval

JULIA BELYAKOVA, Northeastern University
BENJAMIN CHUNG, Northeastern University
JACK GELINAS, Northeastern University
JAMESON NASH, Julia Computing
ROSS TATE, Cornell University
JAN VITEK, Northeastern University / Czech Technical University

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 4–7% of packages are impacted by 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.* 3, OOPSLA, Article 153 (October 2020), 37 pages. https://doi.org/10.1145/3360579

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*, the designers of Julia carefully arranged those features to allow for compiler optimizations. 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; Benjamin Chung, Northeastern University; Jack Gelinas, Northeastern University; Jameson Nash, Julia Computing; Ross Tate, Cornell University; Jan Vitek, Northeastern University / Czech Technical University.

2475-1421/2020/10-ART153
https://doi.org/10.1145/3360579

```
> 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)`.

```
*(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 `*`

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.

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 pick up 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 which delays the effect of `eval` on function definitions is called *world age*. In the Julia documentation, world age is described operationally (see e.g. [Bezanson et al. 2018]): every method defined in a program gets associated with an age (integer value), and for each function call, the language run-time ensures that the current age (also an integer) 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.

In this paper, we provide a number of contributions:

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

The paper is split into two major components: Sec. 3 talks about world age in the wild, and Sec. 4 presents its formalization.

2 BACKGROUND

This section gives an overview of the main features of Julia that are relevant for our work, and provides a brief review of 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**. We will use σ to denote tags.

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 cannot be subtyped. Additionally, any type can have bounded type parameters and can declare up to a single supertype.

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)` where `v` is a value with the tag σ, 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 σ, thus leaving only applicable methods. 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 (no effort was required) and did not come at any performance overhead (due to the lack of optimizations).

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 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 key elements for their result is the presence of `update` instruction that specifies when an update is allowed to happen. This has similarities to the 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.

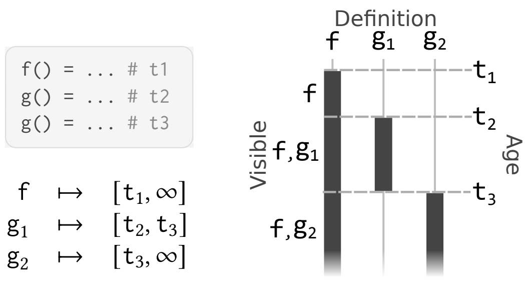
3 WORLD AGE IN JULIA

The world age mechanism in Julia limits the set of methods that can be invoked from a given call site [v1 2020], with a way to bypass this mechanism if necessary. World age fixes the set of method definitions reachable from a currently executing method, isolating it from dynamically generated ones. In turn, this allows Julia's compiler to optimize code without need for deoptimization, and limits the number of required synchronization points in a multi-threaded program. By giving up the performance, one can recover full access to methods.

3.1 Defining world age

The goal of the world age is to limit methods that can be invoked from any call site to those that are known "statically" (here, defined as "whenever the compiler runs"). In Julia, compilation happens whenever a function is called with a type signature that has not been observed before. Then, at every top-level statement, the compiler saves all of the current method definitions into the known set. This process is repeated each time the execution flow returns to the top level. As a result, if a recompilation is needed, it can be identified at the top-level by comparing the known set at original compilation time to that at the new call time. Thus, all recompilation occurs at top level calls, rather than potentially being buried deep inside a call stack.

For performance reasons, the world age mechanism is implemented by a simple monotonic counter. The counter is incremented every time a method is defined, thereby giving each method an age. Each method, saved in a global table of methods, then remembers its birth age, defining the minimum world age from which it can be accessed. In order to support method overwriting and deletion, each method also stores its death age, i.e. the maximum age for invocations. Thus, a call site can only jump to a method after its birth and before its death. This is illustrated in Fig. 6. Here, we define methods `f` and `g`, where `g`'s definition changes from g_1 to g_2 part-way through and observe what definitions are visible at each step. Since `f` is defined once at time t_1 and never



```
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

redefined, it has a minimum age of t_1 and a maximum of ∞; thus, it can be used anytime after t_1 and forevermore. In contrast, the g_1 created at time t_2 is replaced at t_3, so its minimum age is t_2 and maximum age is t_3. Finally, g_2 is never overwritten, 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 calling a dynamically added function. In Fig. 7, function `ntl` creates a new function `f` using `eval`, and then attempts to immediately call it without returning to the top-level. Dissecting the example and its error message, `ntl` was invoked from the top-level with age of 26806. It then used `eval` to define `f`, giving it a birth age of 26807. Then, `ntl` needs an implementation of `f` younger than 26806 to invoke, but none exists. Since the only definition of `f` was created at 26807, a `MethodError` is produced 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 minimum age (a.k.a. "break the age barrier"). The first is `eval` itself, which executes its arguments as if they were at the top level, thus allowing any function 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, and thus, 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 by calling `f` in the latest world age. Namely, 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 notated 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 programmer expectations? We propose a slightly indirect answer to this, by analyzing a corpus of programs and observing how often they use 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 *could* potentially be impacted. 15 packages used `invokelatest` only, and some 56 used both. We can say with confidence that at least these latter 71 packages were impacted by world age, because they tried to bypass it using `invokelatest`. Drawing conclusions about `eval`-using packages requires further analysis.

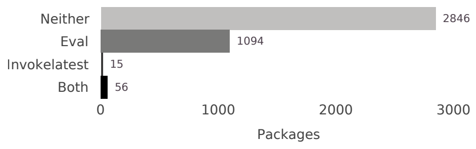


Fig. 8. `eval` and `invokelatest` use by package

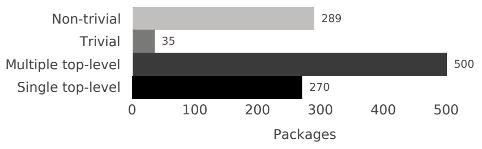


Fig. 9. Use of `eval` by package

To understand if packages that use `eval` but not `invokelatest` are impacted by world age, we statically analyzed arguments of `eval`. The analysis and its results are publicly available on GitHub [Gelinas et al. 2020]. In the analysis, we parse all files that contain `eval`, and then classify the AST elements passed as arguments to `eval` calls. We use a conservative analysis, assuming that any argument whose AST is not statically obvious (such as a parsed string, variable, etc) could contain any AST. We then recursively analyze any blocks or sequences that appear inside of the given ASTs, individually counting their elements.

The results of the `eval`-call analysis are shown in Fig. 9: many uses of `eval` neither impact nor bypass the world age. Top-level calls to `eval` are the most common. These calls have the same effect on world age as regular top-level definitions, as world age will be incremented immediately afterwards, making any definitions instantly usable. We found 270 packages with a single top-level `eval` call and a further 500 with multiple top-level `eval` calls. None of these packages impact, or are impacted by, world age. Similarly, a further 35 packages make `eval` calls whose ASTs preclude them from impacting world age, regardless of when they are called (e.g. such `eval`s update global variables or define new types). This leaves us with 289 packages that have some call to `eval` from within a function, and whose body *may* define a method or call a function. To discriminate inside of these packages further, one would need to manually examine the code or run a dynamic analysis.

To get a better sense of *how* programs use `eval`, we also looked at the distribution of AST forms in `eval` arguments, as shown in Fig. 10 and Fig. 11. This data excludes several known macro calls that do not impact world age (e.g. `@inline` or `@printf`). Fig. 10 shows the distribution of AST forms in top-level calls: function definitions dominate, with the other forms, including exports, being much less frequent. Fig. 11 shows the more interesting `eval`s, those occurring within the bodies of other functions. Here, we see that while many fewer packages use `eval` from within a function body,

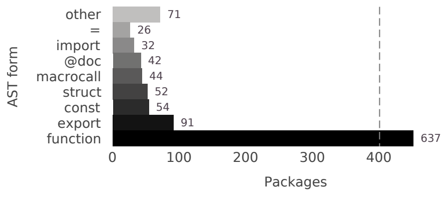


Fig. 10. Top-level evals

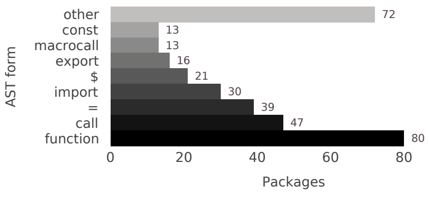


Fig. 11. Inside-function evals

most still use the mechanism to define functions. However, method calls and variable assignment are runners-up, followed intriguingly by **`import`**. This data indicates that for at least 115 (that `eval` a function definition or a call expression inside of another function) of the 289 packages we identified earlier as containing a non-trivial `eval` (which include those within a method body), world age is relevant. Thus, overall, at least 186 packages out of 4011 (4.6%) registered Julia packages might be affected by world age.

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 are typically created at the top-level, so that the generated functions 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 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 onto 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`; 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.[1] 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 thus 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 `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
```

[1] In Julia, higher-order functions are passed by name rather than value, so a callback will be dispatched.

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.[2] 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*

[2]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[3], 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 expression 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, instead of using quotation, $(\!|\, \mathtt{e} \,|\!)$ reads from the local environment directly.

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 uses the current global table to create a fresh local table. This new local table is then affixed to the banana brackets for the rest of this term's evaluation. For example, $(\!|\, \mathtt{m}(\overline{\mathtt{v}}) \,|\!)$ steps to $(\!|\, \mathtt{m}(\overline{\mathtt{v}}) \,|\!)_{\mathrm{M}}$ where M is the current global table. Thus, once the global table becomes local, all inner function calls of $\mathtt{m}(\overline{\mathtt{v}})$ will be resolved in 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 it is parameterized over values, types, type annotations, 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 $\mathtt{md}$ and function calls $\mathtt{e}(\overline{\mathtt{e}})$, sequencing $\mathtt{e}_1 \,;\, \mathtt{e}_2$, global evaluation $(\!|\, \mathtt{e} \,|\!)$, 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`) 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`) 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`) 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 a global evaluation expression $(\!|\, \cdot \,|\!)$. World evaluation contexts $\mathtt{C}$, on the other hand, capture the full grammar

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

of expressions. In addition, world evaluation contexts encapsulate the idea of a fixed world age with the construct $\llparenthesis \cdot \rrparenthesis_{\mathrm{M}}$ (evaluation under a fixed method table M).

Program state is a pair $\langle \mathrm{M}, \mathtt{C}\,[\mathtt{e}] \rangle$ of a global method table M and an expression $\mathtt{C}\,[\mathtt{e}]$. The global method table is empty at program start. We define the semantics of the calculus using two judgments, a normal small-step evaluation $\langle \mathrm{M}, \mathtt{C}\,[\mathtt{e}] \rangle \;\rightarrow\; \langle \mathrm{M}', \mathtt{C}\,[\mathtt{e}'] \rangle$ and stepping to an error $\mathrm{M} \vdash \mathtt{C}\,[\mathtt{e}] \;\rightarrow\; \mathtt{error}$. Complete program evaluation is defined in the middle of Fig. 13. The $\mathbf{typeof}(\mathtt{v}) \in \sigma$ operator returns the tag of a value. We require that $\mathbf{typeof}(\mathtt{unit}) = \mathbb{1}$ and $\mathbf{typeof}(\mathtt{m}) = \mathbb{f}_{\mathtt{m}}$. We write $\mathbf{typeof}(\overline{\mathtt{v}})$ as a shorthand for $\overline{\mathbf{typeof}(\mathtt{v})}$. 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. $\forall \overline{\mathtt{v}}, \overline{\sigma}.(\mathbf{typeof}(\overline{\mathtt{v}}) = \overline{\sigma} \land \Delta(l, \overline{\mathtt{v}}) = \mathtt{v}' \implies \mathbf{typeof}(\mathtt{v}') = \Psi(l, \overline{\sigma}))$. The subtyping relation $\tau_1 <: \tau_2$ is used for multiple dispatch. We require that $\forall \tau. \tau <: \top$ ($\top$ 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 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 use the special case of C and annotate the call $\mathtt{m}(\overline{\mathtt{v}})$ with a copy of the current global method table M, producing $\llparenthesis\, \mathtt{m}(\overline{\mathtt{v}})\, \rrparenthesis_{\mathrm{M}}$.

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

$$
\begin{array}{llllllll}
\mathtt{e} & ::= & & \textit{Expression} & & & & \\
 & | & \mathtt{v} & \text{value} & \mathtt{v} & := & \ldots & \textit{Value} \\
 & | & x & \text{variable} & & | & \mathtt{unit} & \text{unit value} \\
 & | & \mathtt{e}_1\,;\,\mathtt{e}_2 & \text{sequencing} & & | & \mathtt{m} & \text{generic function} \\
 & | & \delta_l(\overline{\mathtt{e}}) & \text{primop call} & & & & \\
 & | & \mathtt{e}(\overline{\mathtt{e}}) & \text{function call} & \sigma & := & \ldots & \textit{Type tag} \\
 & | & \mathtt{md} & \text{method definition} & & | & \mathbb{1} & \text{unit type} \\
 & | & \llparenthesis\, \mathtt{e}\, \rrparenthesis & \text{global evaluation} & & | & \mathbb{f}_{\mathtt{m}} & \text{type tag of function } \mathtt{m} \\
 & & & & & & & \\
\mathtt{p} & ::= & \llparenthesis\, \mathtt{e}\, \rrparenthesis & \textit{Program} & \tau & := & \ldots & \textit{Type annotation} \\
 & & & & & | & \top & \text{top type} \\
\mathtt{md} & ::= & \triangleleft\, \mathtt{m}(\overline{x :: \tau}) = \mathtt{e}\, \triangleright & \textit{Method definition} & & & &
\end{array}
$$

Fig. 12. Surface syntax

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

$$
\frac{\langle \mathrm{M}, \mathtt{C}\,[\mathtt{e}]\rangle \;\to\; \langle \mathrm{M}', \mathtt{C}\,[\mathtt{e}']\rangle}{\langle \mathrm{M}, \mathtt{C}\,[\mathtt{e}]\rangle \;\xrightarrow{c}\; \langle \mathrm{M}', \mathtt{C}\,[\mathtt{e}']\rangle}\;\textsc{E-Normal}
\qquad
\frac{\mathrm{M} \vdash \mathtt{C}\,[\mathtt{e}] \;\to\; \mathtt{error}}{\langle \mathrm{M}, \mathtt{C}\,[\mathtt{e}]\rangle \;\xrightarrow{c}\; \langle \mathrm{M}, \mathtt{error}\rangle}\;\textsc{E-Error}
$$

$$
\overset{\textsc{E-Seq}}{\frac{}{\langle \mathrm{M}, \mathtt{C}\,[\mathtt{v}\,;\,\mathtt{e}]\rangle \;\to\; \langle \mathrm{M}, \mathtt{C}\,[\mathtt{e}]\rangle}}
\qquad
\overset{\textsc{E-Primop}}{\frac{\Delta(l, \overline{\mathtt{v}}) = \mathtt{v}'}{\langle \mathrm{M}, \mathtt{C}\,[\delta_l(\overline{\mathtt{v}})]\rangle \;\to\; \langle \mathrm{M}, \mathtt{C}\,[\mathtt{v}']\rangle}}
\qquad
\overset{\textsc{E-MD}}{\frac{\mathtt{md} \equiv \triangleleft\, \mathtt{m}(\overline{x :: \tau}) = \mathtt{e}\, \triangleright}{\langle \mathrm{M}, \mathtt{C}\,[\mathtt{md}]\rangle \;\to\; \langle \mathrm{M} \bullet \mathtt{md}, \mathtt{C}\,[\mathtt{m}]\rangle}}
$$

$$
\overset{\textsc{E-CallGlobal}}{\frac{}{\langle \mathrm{M}, \mathtt{C}\,[\llparenthesis\,\mathtt{X}\,[\mathtt{m}(\overline{\mathtt{v}})]\,\rrparenthesis]\rangle \;\to\; \langle \mathrm{M}, \mathtt{C}\,[\llparenthesis\,\mathtt{X}\,[\llparenthesis\,\mathtt{m}(\overline{\mathtt{v}})\,\rrparenthesis_{\mathrm{M}}]\,\rrparenthesis]\rangle}}
\qquad
\overset{\textsc{E-CallLocal}}{\frac{\mathbf{typeof}(\overline{\mathtt{v}}) = \overline{\sigma} \qquad \mathbf{getmd}(\mathrm{M}', \mathtt{m}, \overline{\sigma}) = \triangleleft\, \mathtt{m}(\overline{x :: \tau}) = \mathtt{e}\, \triangleright}{\langle \mathrm{M}, \mathtt{C}\,[\llparenthesis\,\mathtt{X}\,[\mathtt{m}(\overline{\mathtt{v}})]\,\rrparenthesis_{\mathrm{M}'}]\rangle \;\to\; \langle \mathrm{M}, \mathtt{C}\,[\llparenthesis\,\mathtt{X}\,[\mathtt{e}[\overline{x \mapsto \mathtt{v}}]]\,\rrparenthesis_{\mathrm{M}'}]\rangle}}
$$

$$
\overset{\textsc{E-ValGlobal}}{\frac{}{\langle \mathrm{M}, \mathtt{C}\,[\llparenthesis\,\mathtt{v}\,\rrparenthesis]\rangle \;\to\; \langle \mathrm{M}, \mathtt{C}\,[\mathtt{v}]\rangle}}
\qquad
\overset{\textsc{E-ValLocal}}{\frac{}{\langle \mathrm{M}, \mathtt{C}\,[\llparenthesis\,\mathtt{v}\,\rrparenthesis_{\mathrm{M}'}]\rangle \;\to\; \langle \mathrm{M}, \mathtt{C}\,[\mathtt{v}]\rangle}}
$$

$$
\overset{\textsc{E-VarErr}}{\frac{}{\mathrm{M} \vdash \mathtt{C}\,[x] \;\to\; \mathtt{error}}}
\qquad
\overset{\textsc{E-PrimopErr}}{\frac{\Delta(l, \overline{\mathtt{v}}) = \mathtt{error}}{\mathrm{M} \vdash \mathtt{C}\,[\delta_l(\overline{\mathtt{v}})] \;\to\; \mathtt{error}}}
\qquad
\overset{\textsc{E-CalleeErr}}{\frac{\mathtt{v}_c \neq \mathtt{m}}{\mathrm{M} \vdash \mathtt{C}\,[\mathtt{v}_c(\overline{\mathtt{v}})] \;\to\; \mathtt{error}}}
$$

$$
\overset{\textsc{E-CallErr}}{\frac{\mathbf{typeof}(\overline{\mathtt{v}}) = \overline{\sigma} \qquad \mathbf{getmd}(\mathrm{M}', \mathtt{m}, \overline{\sigma}) = \mathtt{error}}{\mathrm{M} \vdash \mathtt{C}\,[\llparenthesis\,\mathtt{X}\,[\mathtt{m}(\overline{\mathtt{v}})]\,\rrparenthesis_{\mathrm{M}'}] \;\to\; \mathtt{error}}}
$$

$$
\begin{array}{rcl}
\mathbf{getmd}(\mathrm{M}, \mathtt{m}, \overline{\sigma}) & = & \min(\mathrm{applicable}(\mathbf{latest}(\mathrm{M}), \mathtt{m}, \overline{\sigma})) \\
\mathbf{latest}(\mathrm{M}) & = & \mathbf{latest}(\emptyset, \mathrm{M}) \\
\mathbf{latest}(\mathit{mds}, \varnothing) & = & \mathit{mds} \\
\mathbf{latest}(\mathit{mds}, \mathrm{M} \bullet \mathtt{md}) & = & \mathit{mds} \cup \mathtt{md} \\
 & & \text{if } \neg\, \mathrm{contains}(\mathit{mds}, \mathtt{md}) \\
\mathbf{latest}(\mathit{mds}, \mathrm{M} \bullet \mathtt{md}) & = & \mathit{mds} \\
 & & \text{if } \mathrm{contains}(\mathit{mds}, \mathtt{md}) \\
\mathrm{contains}(\mathit{mds}, \mathtt{md}) & = & \exists\, \mathtt{md}' \in \mathit{mds} \text{ s.t.} \\
 & & \quad (\mathtt{md} \equiv \triangleleft\, \mathtt{m}(\overline{\_ :: \tau}) = \_\, \triangleright) \;\wedge\; (\mathtt{md}' \equiv \triangleleft\, \mathtt{m}(\overline{\_ :: \tau'}) = \_\, \triangleright) \\
 & & \quad \wedge\, \overline{\tau} <: \overline{\tau}' \;\wedge\; \overline{\tau}' <: \overline{\tau} \\
\mathrm{applicable}(\mathit{mds}, \mathtt{m}, \overline{\sigma}) & = & \{\triangleleft\, \mathtt{m}(\overline{x :: \tau}) = \mathtt{e}\, \triangleright \in \mathit{mds} \mid \overline{\sigma} <: \overline{\tau}\} \\
\min(\mathit{mds}) & = & \triangleleft\, \mathtt{m}(\overline{x :: \tau}) = \mathtt{e}\, \triangleright \in \mathit{mds} \text{ s.t.} \\
 & & \quad \forall\, \triangleleft\, \mathtt{m}(\overline{\_ :: \tau'}) = \_\, \triangleright \in \mathit{mds}\,.\; \overline{\tau} <: \overline{\tau}' \\
\min(\mathit{mds}) & = & \mathtt{error} \text{ otherwise}
\end{array}
$$

Fig. 13. Internal Syntax and Semantics

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

$$\begin{aligned}\llparenthesis\ \ & \triangleleft\, \mathtt{g}() = 2 \,\triangleright\, ;\\ & \triangleleft\, \mathtt{f}(x) = (\llparenthesis\, \triangleleft\, \mathtt{g}() = x \,\triangleright\, \rrparenthesis\, ;\, x * \mathtt{g}()) \,\triangleright\, ;\\ & \mathtt{f}(42) \quad \rrparenthesis\end{aligned}$$

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

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 in the $\llparenthesis\ \rrparenthesis_{M'}$ context. This ensures that nested calls will be resolved in the same table, unless they are additionally wrapped into a global evaluation $\llparenthesis\ \rrparenthesis$.

An auxiliary meta-function **getmd**$(\mathrm{M}, \mathtt{m}, \overline{\sigma})$, 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 errors if such a method does not exist. If the method table contains multiple equivalent methods, older ones are ignored. For example, for the program

$$\llparenthesis\, \triangleleft\, \mathtt{g}() = 2 \,\triangleright\, ;\ \triangleleft\, \mathtt{g}() = 42 \,\triangleright\, ;\ \mathtt{g}()\, \rrparenthesis,$$

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 relation). 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 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 $\llparenthesis\ \rrparenthesis$ 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 $\llparenthesis\ \rrparenthesis$. The $\llparenthesis\ \rrparenthesis$ context of $\mathtt{e}$ in Juliette effectively acts the same way that `eval` of $\mathtt{e}$ does in Julia, but evaluates variables of $\mathtt{e}$ in local, rather than global, scope.

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 $\mathtt{p}$ is the program on the right of Fig. 14; $*$ operator is a primop. The first several steps of evaluation use rules E-MD and E-SEQ to add the definitions of $\mathtt{g}$ and $\mathtt{f}$ to the global table. This produces the state

$$\langle \mathrm{M}_0, \llparenthesis\, \mathtt{f}(42)\, \rrparenthesis \rangle,$$

where

$$\begin{aligned}\mathrm{M}_0 \quad = \quad & (\varnothing \bullet \triangleleft\, \mathtt{g}() = 2 \,\triangleright)\\ & \bullet \triangleleft\, \mathtt{f}(x) = (\llparenthesis\, \triangleleft\, \mathtt{g}() = x \,\triangleright\, \rrparenthesis\, ;\, x * \mathtt{g}()) \,\triangleright\end{aligned}$$

Next, using the E-CallGlobal rule, the top-level call $\texttt{f}(42)$ steps to $\llparenthesis\, \texttt{f}(42) \,\rrparenthesis_{M_0}$. This then produces the state

$$\langle M_0, \llparenthesis\, \llparenthesis\, \texttt{f}(42) \,\rrparenthesis_{M_0} \,\rrparenthesis \rangle,$$

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

$$\langle M_0, \llparenthesis\ \ \llparenthesis\, \llparenthesis \triangleleft\ \mathrm{g}()= 42 \triangleright \rrparenthesis\, ;\ 42 * \mathrm{g}() \,\rrparenthesis_{M_0}\ \ \rrparenthesis \rangle.$$

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

$$\langle M_1, \llparenthesis\ \ \llparenthesis\, \llparenthesis\, \mathrm{g} \,\rrparenthesis\, ;\ 42 * \mathrm{g}() \,\rrparenthesis_{M_0}\ \ \rrparenthesis \rangle,$$

where

$$\begin{array}{rclcl} M_1 & = & M_0 & = & ((\varnothing \bullet \triangleleft\ \mathrm{g}()= 2 \triangleright) \\ & & \bullet \triangleleft\ \mathrm{g}()= 42 \triangleright & & \bullet \triangleleft\ \texttt{f}(x)= (\llparenthesis \triangleleft\ \mathrm{g}()= x \triangleright \rrparenthesis\, ;\ x * \mathrm{g}()) \triangleright) \\ & & & & \bullet \triangleleft\ \mathrm{g}()= 42\ \triangleright . \end{array}$$

The next two steps are:

$$\begin{array}{rcll} \langle M_1, \llparenthesis\ \ \llparenthesis\, \llparenthesis\, \mathrm{g} \,\rrparenthesis\, ;\ 42 * \mathrm{g}() \,\rrparenthesis_{M_0}\ \ \rrparenthesis \rangle & \xrightarrow{\text{E-ValGlobal}} & \langle M_1, \llparenthesis\, \llparenthesis\, \mathrm{g}\, ;\ 42 * \mathrm{g}() \,\rrparenthesis_{M_0} \rrparenthesis \rangle & (1) \\ & \xrightarrow{\text{E-Seq}} & \langle M_1, \llparenthesis\, \llparenthesis\, 42 * \mathrm{g}() \,\rrparenthesis_{M_0} \rrparenthesis \rangle. & (2) \end{array}$$

Note that the last program state is represented by $\langle M_1, \texttt{C}\left[\llparenthesis\, \texttt{X}\, [\mathrm{g}()] \,\rrparenthesis_{M_0}\right] \rangle$, where $\texttt{C} = \llparenthesis \square \rrparenthesis$ and $\texttt{X} = 42 * \square$. So we have to use E-CallLocal again to resolve $\mathrm{g}()$ in 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:

$$\langle M_1, \llparenthesis\, \llparenthesis\, 42 * 2 \,\rrparenthesis_{M_0} \rrparenthesis \rangle.$$

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

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

Now, consider a modification of the original program where in the definition of $\texttt{f}$, the call $\mathrm{g}()$ is wrapped into a global evaluation $\llparenthesis\, \mathrm{g}() \,\rrparenthesis$:

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

$$\begin{array}{ll} \llparenthesis & \triangleleft\ \mathrm{g}()= 2 \triangleright ; \\ & \triangleleft\ \texttt{f}(x)= (\llparenthesis \triangleleft\ \mathrm{g}()= x \triangleright \rrparenthesis\, ;\ x * \llparenthesis\, \mathrm{g}() \,\rrparenthesis) \triangleright ; \\ & \texttt{f}(42) \quad \rrparenthesis \end{array}$$

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

$$\langle M_1, \llparenthesis\, \llparenthesis\, 42 * \llparenthesis\, \mathrm{g}() \,\rrparenthesis \,\rrparenthesis_{M_0} \rrparenthesis \rangle.$$

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

$$\langle M_1, \llparenthesis\, \llparenthesis\, 42 * \llparenthesis\ \ \llparenthesis\, \mathrm{g}() \,\rrparenthesis_{M_1}\ \ \rrparenthesis \,\rrparenthesis_{M_0} \rrparenthesis \rangle.$$

Resolved in M_1, call $\mathrm{g}()$ 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.

$$
\begin{array}{llll@{\qquad}llll}
\texttt{rdx} & ::= & & \textit{Redex base} & \texttt{rep} & ::= & & \textit{Form of expression} \\
 & | & x & & & | & \mathcal{V}(\texttt{v}) & \\
 & | & \texttt{v}\,;\,\texttt{e} & & & | & \mathcal{X}(\texttt{X},\ \texttt{m}(\overline{\texttt{v}})) & \\
 & | & \delta_I(\overline{\texttt{v}}) & & & | & \mathcal{C}(\texttt{C},\ \texttt{rdx}) & \\
 & & \ldots & & & & & \\
 & | & (\!|\, \texttt{X}\,[\texttt{m}(\overline{\texttt{v}})]\, |\!)_{\texttt{M}} & & & & \textbf{\textit{Can}}\,(\texttt{e as rep}) & \textit{Canonical form}
\end{array}
$$

$$
\text{CN-VAL}\ \frac{}{\textbf{\textit{Can}}\,(\texttt{v as } \mathcal{V}(\texttt{v}))}
\qquad
\text{CN-CALL}\ \frac{}{\textbf{\textit{Can}}\,(\texttt{m}(\overline{\texttt{v}}) \texttt{ as } \mathcal{X}(\square,\ \texttt{m}(\overline{\texttt{v}})))}
\qquad
\text{CN-REDEX}\ \frac{}{\textbf{\textit{Can}}\,(\texttt{rdx as } \mathcal{C}(\square,\ \texttt{rdx}))}
$$

$$
\text{CN-SEQX}\ \frac{\textbf{\textit{Can}}\,(\texttt{e}_1 \texttt{ as } \mathcal{X}(\texttt{X}_1,\ \texttt{m}(\overline{\texttt{v}})))}{\textbf{\textit{Can}}\,(\texttt{e}_1\,;\,\texttt{e}_2 \texttt{ as } \mathcal{X}(\texttt{X}_1\,;\,\texttt{e}_2,\ \texttt{m}(\overline{\texttt{v}})))}
\qquad
\text{CN-SEQC}\ \frac{\textbf{\textit{Can}}\,(\texttt{e}_1 \texttt{ as } \mathcal{C}(\texttt{C}_1,\ \texttt{rdx}))}{\textbf{\textit{Can}}\,(\texttt{e}_1\,;\,\texttt{e}_2 \texttt{ as } \mathcal{C}(\texttt{C}_1\,;\,\texttt{e}_2,\ \texttt{rdx}))}
\qquad
\text{CN-EVALLOCALC}\ \frac{\textbf{\textit{Can}}\,(\texttt{e as } \mathcal{C}(\texttt{C},\ \texttt{rdx}))}{\textbf{\textit{Can}}\,((\!|\,\texttt{e}\,|\!)_{\texttt{M}} \texttt{ as } \mathcal{C}((\!|\,\texttt{C}\,|\!)_{\texttt{M}},\ \texttt{rdx}))}
$$

Fig. 15. Redexes and canonical forms

4.4 Properties

Error evaluation captures all failure states of JULIETTE, meaning that a JULIETTE program never gets stuck: it either reduces to a value or an error, or can make a step to another program.

THEOREM 4.1 (PROGRESS). *For any program* p *and method table* M_g, *one of the following holds:*

(a) $\langle M_g, \texttt{p}\rangle \overset{c}{\longrightarrow} \langle M'_g, \texttt{r}\rangle$*; or*

(b) $\langle M_g, \texttt{p}\rangle \overset{c}{\longrightarrow} \langle M'_g, \texttt{p}'\rangle$.

PROOF. By case analysis on $\texttt{p} = (\!|\,\texttt{e}\,|\!)$. □

THEOREM 4.2 (DETERMINISM). *JULIETTE semantics is deterministic.*

PROOF. The full proof relies on a number of auxiliary lemmas about the form of expressions that can make a (normal- or error-evaluation) step. The key observation is that (1) any expression that steps can be represented as C [rdx], where rdx is a subset of expressions driving the reduction, and (2) there is exactly one rule for each rdx form. To help reason about contexts in proofs, we employ an additional judgment ***Can*** (e as rep), which describes the canonical representation of an expression. Fig. 15 shows excerpts from the definitions of rdx and ***Can***. The reason we need ***Can*** is that contexts X and C are intertwined; when looking at a program C [e], it is not obvious how to represent it as a C′ [e′] that can make a step. For any expression, ***Can*** does exactly that: it destructs the expression in a way that is suitable for applying evaluation rules. □

4.5 Optimizations

The world age semantics allows function call optimization even in the presence of `eval`. Recall how an `eval`ed or a 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 $(\!|\,\texttt{m}(\overline{\texttt{v}})\,|\!)_{\texttt{M}}$. 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.

```
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)

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.

Inlining. If a function call is statically known to dispatch to a certain method, 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 into `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, function call can be replaced by a direct invokation. 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, `g` will be dispatched to the 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 an efficient integer additon. 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.

Optimization Formalization. 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 $\Gamma \vdash \texttt{e} : \sigma$ propagates information about variables and type tags of values, and succeeds only if the expression can be concretely typed. This is because to resolve a function call, we need to know 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) M′ can have more methods given that their names appear neither in the original table M nor in the expression e for which the table is getting optimized. 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: $\texttt{m}(\overline{\sigma}) \rightsquigarrow \texttt{m}'$ 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 well-defined according to MethodOpt-WellDef: 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-WellDef 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 OE-Inline and OE-Direct correspond to inlining and direct call/specialization optimization, 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 ν, "near-value", which is either a value or a variable. Because we model direct call and specialization optimizations as calls to freshly-named methods, main job is done in the table optimization rule OT-MethodTable; rule OE-Direct only records the fact of invoking a specific method. If m′ from $\texttt{m}(\overline{\sigma}) \rightsquigarrow \texttt{m}'$ has the same parameter type annotations as m, it represents a direct call to the original method of m; otherwise, it represents a specialized method. Note that for all optimizations, function call arguments have to be concretely-typed at σ_i. Otherwise, we do not know definitively how a function call is going to be dispatched at run-time.

$$
\begin{array}{llll}
\Gamma & ::= & & \textit{Typing environment} \\
 & \mid & \varnothing & \\
 & \mid & \Gamma, x : \tau & \\
 & & & \\
\gamma & ::= & \overline{x \mapsto \mathsf{v}} & \textit{Value substitution} \\
 & & \forall i, j . x_i \neq x_j &
\end{array}
\qquad
\begin{array}{llll}
\nu & ::= & & \textit{Near-value} \\
 & \mid & \mathsf{v} & \text{value} \\
 & \mid & x & \text{variable} \\
 & & & \\
\Phi & ::= & & \textit{Method optimization environment} \\
 & \mid & \varnothing & \\
 & \mid & \Phi, \mathtt{m}(\overline{\sigma}) \rightsquigarrow \mathtt{m}' &
\end{array}
$$

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

$$
\textsc{T-Primop}\ \frac{\Psi(l, \overline{\sigma}) = \sigma' \qquad \Gamma \vdash \mathtt{e}_i : \sigma_i}{\Gamma \vdash \delta_l(\overline{\mathtt{e}}) : \sigma'}
\qquad
\gamma\textsc{-Ok}\ \frac{dom(\Gamma) = dom(\gamma) \qquad \forall x \in dom(\gamma).\ [\Gamma(x) = \sigma \iff \textbf{typeof}(\gamma(x)) = \sigma]}{\Gamma \vdash \gamma}
$$

Fig. 17. Concrete-typing judgment and value substitution

$$
\textsc{MethodOpt-WellDef}\ \frac{\begin{array}{c}\textbf{getmd}(\mathrm{M}, \mathtt{m}, \overline{\sigma}) = \triangleleft\, \mathtt{m}(\overline{x :: \tau}) = \mathtt{e}_b \,\triangleright \\ \textbf{getmd}(\mathrm{M}', \mathtt{m}', \overline{\sigma}) = \triangleleft\, \mathtt{m}'(\overline{x' :: \tau'}) = \mathtt{e}'_b \,\triangleright \\ \overline{x : \sigma} \vdash^{\Phi} (\!|\, \mathtt{e}_b \,|\!)_{\mathrm{M}} \rightsquigarrow (\!|\, \mathtt{e}'_b[\overline{x}' \mapsto \overline{x}] \,|\!)_{\mathrm{M}'}\end{array}}{\vdash^{\Phi}_{\mathrm{M} \rightsquigarrow \mathrm{M}'} \mathtt{m}(\overline{\sigma}) \rightsquigarrow \mathtt{m}'}
\qquad
\textsc{OD-MD}\ \frac{\overline{x : \tau} \vdash^{\Phi} (\!|\, \mathtt{e} \,|\!)_{\mathrm{M}} \rightsquigarrow (\!|\, \mathtt{e}'[x' \mapsto x] \,|\!)_{\mathrm{M}'}}{\vdash^{\Phi}_{\mathrm{M} \rightsquigarrow \mathrm{M}'} \triangleleft\, \mathtt{m}(\overline{x :: \tau}) = \mathtt{e} \,\triangleright \rightsquigarrow \triangleleft\, \mathtt{m}(\overline{x' :: \tau}) = \mathtt{e}' \,\triangleright}
$$

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

Fig. 18. Method table & definition optimization

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 first establish an auxiliary fact that there is a bisimulation relation between an original and optimized expressions:

Lemma 4.3. *For all method tables* M, M', *method optimization environment* Φ, *typing environment* Γ, *and expressions* $\mathtt{e}_1$, $\mathtt{e}_1'$, *such that*

$$\vdash^{\Phi}_{\mathtt{e}_1} \mathrm{M} \rightsquigarrow \mathrm{M}' \quad \textit{and} \quad \Gamma \vdash^{\Phi} (\!|\, \mathtt{e}_1 \,|\!)_{\mathrm{M}} \rightsquigarrow (\!|\, \mathtt{e}'_1 \,|\!)_{\mathrm{M}'},$$

for all value substitutions γ *such that*

$$\Gamma \vdash \gamma,$$

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

(1) Forward direction:

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

$$\textsc{OE-Val}\ \frac{\mathtt{v} \neq \mathtt{m}}{\Gamma \vdash^{\Phi} \llparenthesis\, \mathtt{v} \,\rrparenthesis_{\mathrm{M}} \rightsquigarrow \llparenthesis\, \mathtt{v} \,\rrparenthesis_{\mathrm{M}'}} \qquad \textsc{OE-ValFun}\ \frac{\mathtt{m} \in \mathit{dom}(\mathrm{M}) \iff \mathtt{m} \in \mathit{dom}(\mathrm{M}')}{\Gamma \vdash^{\Phi} \llparenthesis\, \mathtt{m} \,\rrparenthesis_{\mathrm{M}} \rightsquigarrow \llparenthesis\, \mathtt{m} \,\rrparenthesis_{\mathrm{M}'}} \qquad \textsc{OE-Var}\ \frac{}{\Gamma \vdash^{\Phi} \llparenthesis\, x \,\rrparenthesis_{\mathrm{M}} \rightsquigarrow \llparenthesis\, x \,\rrparenthesis_{\mathrm{M}'}}$$

$$\textsc{OE-Global}\ \frac{}{\Gamma \vdash^{\Phi} \llparenthesis\, \llparenthesis\, \mathtt{e} \,\rrparenthesis \,\rrparenthesis_{\mathrm{M}} \rightsquigarrow \llparenthesis\, \llparenthesis\, \mathtt{e} \,\rrparenthesis \,\rrparenthesis_{\mathrm{M}'}} \qquad \textsc{OE-Local}\ \frac{}{\Gamma \vdash^{\Phi} \llparenthesis\, \llparenthesis\, \mathtt{e} \,\rrparenthesis_{\mathrm{M}_l} \,\rrparenthesis_{\mathrm{M}} \rightsquigarrow \llparenthesis\, \llparenthesis\, \mathtt{e} \,\rrparenthesis_{\mathrm{M}_l} \,\rrparenthesis_{\mathrm{M}'}} \qquad \textsc{OE-MD}\ \frac{}{\Gamma \vdash^{\Phi} \llparenthesis\, \mathtt{md} \,\rrparenthesis_{\mathrm{M}} \rightsquigarrow \llparenthesis\, \mathtt{md} \,\rrparenthesis_{\mathrm{M}'}}$$

$$\textsc{OE-Seq}\ \frac{\Gamma \vdash^{\Phi} \llparenthesis\, \mathtt{e}_1 \,\rrparenthesis_{\mathrm{M}} \rightsquigarrow \llparenthesis\, \mathtt{e}_1' \,\rrparenthesis_{\mathrm{M}'} \qquad \Gamma \vdash^{\Phi} \llparenthesis\, \mathtt{e}_2 \,\rrparenthesis_{\mathrm{M}} \rightsquigarrow \llparenthesis\, \mathtt{e}_2' \,\rrparenthesis_{\mathrm{M}'}}{\Gamma \vdash^{\Phi} \llparenthesis\, \mathtt{e}_1\,;\,\mathtt{e}_2 \,\rrparenthesis_{\mathrm{M}} \rightsquigarrow \llparenthesis\, \mathtt{e}_1'\,;\,\mathtt{e}_2' \,\rrparenthesis_{\mathrm{M}'}} \qquad \textsc{OE-Primop}\ \frac{\Gamma \vdash^{\Phi} \llparenthesis\, \mathtt{e}_i \,\rrparenthesis_{\mathrm{M}} \rightsquigarrow \llparenthesis\, \mathtt{e}_i' \,\rrparenthesis_{\mathrm{M}'}}{\Gamma \vdash^{\Phi} \llparenthesis\, \delta_l(\overline{\mathtt{e}}) \,\rrparenthesis_{\mathrm{M}} \rightsquigarrow \llparenthesis\, \delta_l(\overline{\mathtt{e}'}) \,\rrparenthesis_{\mathrm{M}'}}$$

$$\textsc{OE-Call}\ \frac{\Gamma \vdash^{\Phi} \llparenthesis\, \mathtt{e}_c \,\rrparenthesis_{\mathrm{M}} \rightsquigarrow \llparenthesis\, \mathtt{e}_c' \,\rrparenthesis_{\mathrm{M}'} \qquad \Gamma \vdash^{\Phi} \llparenthesis\, \mathtt{e}_i \,\rrparenthesis_{\mathrm{M}} \rightsquigarrow \llparenthesis\, \mathtt{e}_i' \,\rrparenthesis_{\mathrm{M}'}}{\Gamma \vdash^{\Phi} \llparenthesis\, \mathtt{e}_c(\overline{\mathtt{e}}) \,\rrparenthesis_{\mathrm{M}} \rightsquigarrow \llparenthesis\, \mathtt{e}_c'(\overline{\mathtt{e}'}) \,\rrparenthesis_{\mathrm{M}'}}$$

$$\textsc{OE-Inline}\ \frac{\Gamma \vdash v_i : \sigma_i \qquad \mathbf{getmd}(\mathrm{M}, \mathtt{m}, \overline{\sigma}) = \blacktriangleleft \mathtt{m}(\overline{x :: \tau}) = \mathtt{e}_b \blacktriangleright \qquad \Gamma \vdash^{\Phi} \llparenthesis\, \mathtt{e}_b[\overline{x \mapsto v}] \,\rrparenthesis_{\mathrm{M}} \rightsquigarrow \llparenthesis\, \mathtt{e}' \,\rrparenthesis_{\mathrm{M}'}}{\Gamma \vdash^{\Phi} \llparenthesis\, \mathtt{m}(\overline{v}) \,\rrparenthesis_{\mathrm{M}} \rightsquigarrow \llparenthesis\, \mathtt{unit}\,;\,\mathtt{e}' \,\rrparenthesis_{\mathrm{M}'}}$$

$$\textsc{OE-Direct}\ \frac{\Gamma \vdash^{\Phi} \llparenthesis\, \mathtt{e}_i \,\rrparenthesis_{\mathrm{M}} \rightsquigarrow \llparenthesis\, \mathtt{e}_i' \,\rrparenthesis_{\mathrm{M}'} \qquad \Gamma \vdash \mathtt{e}_i' : \sigma_i \qquad (\mathtt{m}(\overline{\sigma}) \rightsquigarrow \mathtt{m}') \in \Phi}{\Gamma \vdash^{\Phi} \llparenthesis\, \mathtt{m}(\overline{\mathtt{e}}) \,\rrparenthesis_{\mathrm{M}} \rightsquigarrow \llparenthesis\, \mathtt{m}'(\overline{\mathtt{e}'}) \,\rrparenthesis_{\mathrm{M}'}}$$

Fig. 19. Expression optimization

(2) *Backward direction:*

$$\begin{aligned} \forall \mathtt{e}_2'.\ \ & \langle \mathrm{M}_g, \mathtt{C}\left[\llparenthesis\, \gamma(\mathtt{e}_1') \,\rrparenthesis_{\mathrm{M}}\right] \rangle \longrightarrow \langle \mathrm{M}_g', \mathtt{C}\left[\llparenthesis\, \mathtt{e}_2' \,\rrparenthesis_{\mathrm{M}}\right] \rangle \\ & \Longrightarrow \\ & \exists \mathtt{e}_2.\ \langle \mathrm{M}_g, \mathtt{C}\left[\llparenthesis\, \gamma(\mathtt{e}_1) \,\rrparenthesis_{\mathrm{M}'}\right] \rangle \longrightarrow \langle \mathrm{M}_g', \mathtt{C}\left[\llparenthesis\, \mathtt{e}_2 \,\rrparenthesis_{\mathrm{M}'}\right] \rangle \ \wedge\ \vdash^{\Phi} \llparenthesis\, \mathtt{e}_2 \,\rrparenthesis_{\mathrm{M}} \rightsquigarrow \llparenthesis\, \mathtt{e}_2' \,\rrparenthesis_{\mathrm{M}'}. \end{aligned}$$

Proof. The proof goes by induction on derivation of optimization $\Gamma \vdash^{\Phi} \llparenthesis\, \mathtt{e}_1 \,\rrparenthesis_{\mathrm{M}} \rightsquigarrow \llparenthesis\, \mathtt{e}_1' \,\rrparenthesis_{\mathrm{M}'}$. 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, $\gamma(\mathtt{e}_1)$ should have a certain representation. For example, if $\gamma(\mathtt{e}_1)$ is a sequence, to step within $\llparenthesis \cdot \rrparenthesis_{\mathrm{M}}$ context, it has to be some $\mathtt{C}\,[\mathtt{rdx}]$.
(2) Consider all canonical representations that satisfy this requirement. For instance, a sequence can be $\mathtt{C}\,[\mathtt{rdx}]$ in exactly two cases, CN-Redex and CN-SeqC.
(3) For each canonical representation, analyze the suitable normal-evaluation rule (recall that the semantics is deterministic, so there will be just one such rule).
(4) If $\gamma(\mathtt{e}_1)$ represents an immediate redex (e.g. $\mathtt{v}_{11}\,;\,\mathtt{e}_{12}$), 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.
(5) Finally, show that the resulting expressions are in the optimization relation. This will follow from the assumptions and induction.

For example, here is a proof sketch for the CN-SeqC case of sequence in the forward direction. By assumption, we have $\mathsf{e}_1 = (\mathsf{e}_{11}\,;\,\mathsf{e}_{12})$ and $\mathsf{e}'_1 = (\mathsf{e}'_{11}\,;\,\mathsf{e}'_{12})$ where:

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

By definition, $\gamma(\mathsf{e}_{11}\,;\,\mathsf{e}_{12}) = \gamma(\mathsf{e}_{11})\,;\,\gamma(\mathsf{e}_{12})$ and $\gamma(\mathsf{e}'_{11}\,;\,\mathsf{e}'_{12}) = \gamma(\mathsf{e}'_{11})\,;\,\gamma(\mathsf{e}'_{12})$. We will denote the results as $\mathsf{e}_{\gamma 1}\,;\,\mathsf{e}_{\gamma 2}$ and $\mathsf{e}'_{\gamma 1}\,;\,\mathsf{e}'_{\gamma 2}$. By the fact that value substitution preserves optimization:

$$\frac{\vdash^{\Phi} \llparenthesis\, \mathsf{e}_{\gamma 1} \,\rrparenthesis_{\mathrm{M}} \rightsquigarrow \llparenthesis\, \mathsf{e}'_{\gamma 1} \,\rrparenthesis_{\mathrm{M}'} \qquad \vdash^{\Phi} \llparenthesis\, \mathsf{e}_{\gamma 2} \,\rrparenthesis_{\mathrm{M}} \rightsquigarrow \llparenthesis\, \mathsf{e}'_{\gamma 2} \,\rrparenthesis_{\mathrm{M}'}}{\vdash^{\Phi} \llparenthesis\, \mathsf{e}_{\gamma 1}\,;\,\mathsf{e}_{\gamma 2} \,\rrparenthesis_{\mathrm{M}} \rightsquigarrow \llparenthesis\, \mathsf{e}'_{\gamma 1}\,;\,\mathsf{e}'_{\gamma 2} \,\rrparenthesis_{\mathrm{M}'}.}\ \textsc{OE-Seq}$$

Recall that CN-SeqC means:

$$\frac{\textbf{\textit{Can}}\ (\mathsf{e}_{\gamma 1}\ \mathbf{as}\ \mathcal{C}(\mathsf{C}_1,\ \mathsf{rdx}))}{\textbf{\textit{Can}}\ (\mathsf{e}_{\gamma 1}\,;\,\mathsf{e}_{\gamma 2}\ \mathbf{as}\ \mathcal{C}(\mathsf{C}_1\,;\,\mathsf{e}_{\gamma 2},\ \mathsf{rdx}))}\ \textsc{CN-SeqC}$$

Thus, we know that $\mathsf{C}\left[\llparenthesis\, \mathsf{e}_{\gamma 1}\,;\,\mathsf{e}_{\gamma 2} \,\rrparenthesis_{\mathrm{M}}\right] = \mathsf{C}\left[\llparenthesis\, \mathsf{C}_1\,[\mathsf{rdx}]\,;\,\mathsf{e}_{\gamma 2} \,\rrparenthesis_{\mathrm{M}}\right]$, which means

$$\begin{gathered}\langle \mathrm{M}_g, \mathsf{C}\left[\llparenthesis\, \mathsf{C}_1\,[\mathsf{rdx}]\,;\,\mathsf{e}_{\gamma 2} \,\rrparenthesis_{\mathrm{M}}\right]\rangle \ \to\ \langle \mathrm{M}'_g, \mathsf{C}\left[\llparenthesis\, \mathsf{C}_1\,[\mathsf{e}']\,;\,\mathsf{e}_{\gamma 2} \,\rrparenthesis_{\mathrm{M}}\right]\rangle \\ \Longleftrightarrow \\ \langle \mathrm{M}_g, \mathsf{C}\left[\llparenthesis\, \mathsf{C}_1\,[\mathsf{rdx}] \,\rrparenthesis_{\mathrm{M}}\right]\rangle \ \to\ \langle \mathrm{M}'_g, \mathsf{C}\left[\llparenthesis\, \mathsf{C}_1\,[\mathsf{e}'] \,\rrparenthesis_{\mathrm{M}}\right]\rangle,\end{gathered}$$

which is the same as

$$\langle \mathrm{M}_g, \mathsf{C}\left[\llparenthesis\, \mathsf{e}_{\gamma 1} \,\rrparenthesis_{\mathrm{M}}\right]\rangle \ \to\ \langle \mathrm{M}'_g, \mathsf{C}\left[\llparenthesis\, \mathsf{C}_1\,[\mathsf{e}'] \,\rrparenthesis_{\mathrm{M}}\right]\rangle.$$

(Note that $\mathsf{C}_1\,[\mathsf{e}']$ plays the role of e_2 from the theorem statement.) Next, recall the assumption $\Gamma \vdash^{\Phi} \llparenthesis\, \mathsf{e}_{11} \,\rrparenthesis_{\mathrm{M}} \rightsquigarrow \llparenthesis\, \mathsf{e}'_{11} \,\rrparenthesis_{\mathrm{M}'}$. By induction hypothesis, $\exists \mathsf{e}'_{21}$ such that

$$\langle \mathrm{M}_g, \mathsf{C}\left[\llparenthesis\, \mathsf{e}'_{\gamma 1} \,\rrparenthesis_{\mathrm{M}'}\right]\rangle \ \to\ \langle \mathrm{M}'_g, \mathsf{C}\left[\llparenthesis\, \mathsf{e}'_{21} \,\rrparenthesis_{\mathrm{M}'}\right]\rangle \quad \text{and} \quad \vdash^{\Phi} \llparenthesis\, \mathsf{C}_1\,[\mathsf{e}'] \,\rrparenthesis_{\mathrm{M}} \rightsquigarrow \llparenthesis\, \mathsf{e}'_{21} \,\rrparenthesis_{\mathrm{M}'}.$$

The way $\mathsf{C}\left[\llparenthesis\, \mathsf{e}'_{\gamma 1} \,\rrparenthesis_{\mathrm{M}'}\right]$ steps, means that $\mathsf{e}'_{\gamma 1} = \mathsf{C}'_1\,[\mathsf{rdx}']$. By context manipulations similar to the above, we get that

$$\langle \mathrm{M}_g, \mathsf{C}\left[\llparenthesis\, \mathsf{e}'_{\gamma 1}\,;\,\mathsf{e}'_{\gamma 2} \,\rrparenthesis_{\mathrm{M}'}\right]\rangle \ \to\ \langle \mathrm{M}'_g, \mathsf{C}\left[\llparenthesis\, \mathsf{e}'_{21}\,;\,\mathsf{e}'_{\gamma 2} \,\rrparenthesis_{\mathrm{M}'}\right]\rangle.$$

Finally, we can show that the optimization relation holds:

$$\frac{\vdash^{\Phi} \llparenthesis\, \mathsf{C}_1\,[\mathsf{e}'] \,\rrparenthesis_{\mathrm{M}} \rightsquigarrow \llparenthesis\, \mathsf{e}'_{21} \,\rrparenthesis_{\mathrm{M}'} \qquad \vdash^{\Phi} \llparenthesis\, \mathsf{e}_{\gamma 2} \,\rrparenthesis_{\mathrm{M}} \rightsquigarrow \llparenthesis\, \mathsf{e}'_{\gamma 2} \,\rrparenthesis_{\mathrm{M}'}}{\vdash^{\Phi} \llparenthesis\, \mathsf{C}_1\,[\mathsf{e}']\,;\,\mathsf{e}_{\gamma 2} \,\rrparenthesis_{\mathrm{M}} \rightsquigarrow \llparenthesis\, \mathsf{e}'_{21}\,;\,\mathsf{e}'_{\gamma 2} \,\rrparenthesis_{\mathrm{M}'}.}\ \textsc{OE-Seq}$$

Other cases proceed similarly. □

The main result, Theorem 4.4, is a corollary of Lemma 4.3. It states that a fixed-table expression can be soundly evaluated in an optimized table.

Theorem 4.4. *For all* $\mathrm{M}, \mathrm{M}', \Phi, e$ *satisfying* $\vdash^{\Phi}_{\mathsf{e}} \mathrm{M} \rightsquigarrow \mathrm{M}'$, *for all* $\mathrm{M}_g, \mathrm{M}'_g, \mathsf{C}, \mathsf{v}$,

$$\langle \mathrm{M}_g, \mathsf{C}\left[\llparenthesis\, \mathsf{e} \,\rrparenthesis_{\mathrm{M}}\right]\rangle \ \to^*\ \langle \mathrm{M}'_g, \mathsf{v}\rangle \iff \langle \mathrm{M}_g, \mathsf{C}\left[\llparenthesis\, \mathsf{e} \,\rrparenthesis_{\mathrm{M}'}\right]\rangle \ \to^*\ \langle \mathrm{M}'_g, \mathsf{v}\rangle.$$

Proof. First of all, note that $\vdash^{\Phi} \llparenthesis\, \mathsf{e} \,\rrparenthesis_{\mathrm{M}} \rightsquigarrow \llparenthesis\, \mathsf{e} \,\rrparenthesis_{\mathrm{M}'}$: this can be shown by induction on e, given $\vdash^{\Phi}_{\mathsf{e}} \mathrm{M} \rightsquigarrow \mathrm{M}'$. Then, we proceed by induction on $\to^*$ (reflexive transitive closure of normal evaluation). In the interesting case of the forward direction, when

$$\frac{\langle \mathrm{M}_g, \mathsf{C}\left[\llparenthesis\, \mathsf{e} \,\rrparenthesis_{\mathrm{M}}\right]\rangle \ \to\ \langle \mathrm{M}''_g, \mathsf{C}\left[\llparenthesis\, \mathsf{e}'_1 \,\rrparenthesis_{\mathrm{M}}\right]\rangle \qquad \langle \mathrm{M}''_g, \mathsf{C}\left[\llparenthesis\, \mathsf{e}'_1 \,\rrparenthesis_{\mathrm{M}}\right]\rangle \ \to^*\ \langle \mathrm{M}'_g, \mathsf{v}\rangle}{\langle \mathrm{M}_g, \mathsf{C}\left[\llparenthesis\, \mathsf{e} \,\rrparenthesis_{\mathrm{M}}\right]\rangle \ \to^*\ \langle \mathrm{M}'_g, \mathsf{v}\rangle,}$$

by applying Lemma 4.3 to the first premise, we have:

$$\langle \mathrm{M}_g, \mathrm{C}\left[\llparenthesis\, \mathrm{e} \,\rrparenthesis_{\mathrm{M}'}\right]\rangle \;\longrightarrow\; \langle \mathrm{M}''_g, \mathrm{C}\left[\llparenthesis\, \mathrm{e}'_2 \,\rrparenthesis_{\mathrm{M}'}\right]\rangle \quad \text{and} \quad \vdash^{\Phi} \llparenthesis\, \mathrm{e}'_1 \,\rrparenthesis_{\mathrm{M}} \rightsquigarrow \llparenthesis\, \mathrm{e}'_2 \,\rrparenthesis_{\mathrm{M}'}.$$

Therefore, by applying induction hypothesis to the second premise, we get:

$$\langle \mathrm{M}''_g, \mathrm{C}\left[\llparenthesis\, \mathrm{e}'_2 \,\rrparenthesis_{\mathrm{M}'}\right]\rangle \;\longrightarrow^*\; \langle \mathrm{M}'_g, \mathrm{v}\rangle,$$

which enables the desired derivation:

$$\frac{\langle \mathrm{M}_g, \mathrm{C}\left[\llparenthesis\, \mathrm{e} \,\rrparenthesis_{\mathrm{M}'}\right]\rangle \;\longrightarrow\; \langle \mathrm{M}''_g, \mathrm{C}\left[\llparenthesis\, \mathrm{e}'_2 \,\rrparenthesis_{\mathrm{M}'}\right]\rangle \qquad \langle \mathrm{M}''_g, \mathrm{C}\left[\llparenthesis\, \mathrm{e}'_2 \,\rrparenthesis_{\mathrm{M}'}\right]\rangle \;\longrightarrow^*\; \langle \mathrm{M}'_g, \mathrm{v}\rangle}{\langle \mathrm{M}_g, \mathrm{C}\left[\llparenthesis\, \mathrm{e} \,\rrparenthesis_{\mathrm{M}'}\right]\rangle \;\longrightarrow^*\; \langle \mathrm{M}'_g, \mathrm{v}\rangle.}$$

The backwards direction proceeds similarly. □

Thus, in particular, once a top-level call $\llparenthesis\, \mathrm{m}(\overline{\mathrm{v}}) \,\rrparenthesis$ steps to a fixed-table call $\llparenthesis\, \mathrm{m}(\overline{\mathrm{v}}) \,\rrparenthesis_{\mathrm{M}}$, it is sound to optimize table M into M′ (using inlining, direct calls, and specialization), and evaluate the call in the optimized table, as $\llparenthesis\, \mathrm{m}(\overline{\mathrm{v}}) \,\rrparenthesis_{\mathrm{M}'}$.

4.6 Testing the semantics

To check if Juliette behaves as we expect, we implemented it in Redex [Felleisen et al. 2009] and ran along with Julia on a small set of litmus tests; 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. 20; 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` isn't. 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. 20. 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 [Gelinas et al. 2020] 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 and one for error evaluation:

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

The new rules treat global method table as a global environment: E-VarMethod evaluates global variable to its underlying function value, and E-VarErr errors 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. 20 translates to the Redex model as follows (the grammar is written in S-expressions style):

```
(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, which are being fixed for top-level calls, 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 needs 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 from program 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.

REFERENCES

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

Jack Gelinas, Julia Belyakova, and Benjamin Chung. 2020. Juliette: World Age in Julia. https://github.com/julbinb/juliette-wa.

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

6 APPENDIX

6.1 Proofs

To simplify proofs, we slightly tweak Juliette syntax as shown in Fig. 21:

- First, we extend expressions $\mathtt{e}$ with evaluation in a method table $(\!|\, \mathtt{e} \,|\!)_{\mathrm{M}}$ (earlier, this syntax form was a part of the context $\mathtt{C}$ but not the surface language expressions $\mathtt{e}$).
- Second, we introduce a new value form for non-functional values $\mathtt{v}^{\neq \mathtt{m}} = \mathtt{v} \setminus \mathtt{m}$.

e	::=		*Internal Expression*
	|	v	value
	|	...	
	|	$(\!\| \mathtt{e} \|\!)$	global evaluation
	|	$(\!\| \mathtt{e} \|\!)_{\mathrm{M}}$	evaluation in table M
p	::=	$(\!\| \mathtt{e} \|\!)$	*Program*
i	:=	e | error	*Expression or Error*
$\mathtt{v}^{\neq \mathtt{m}}$	:=	...	*Simple Value*
	|	unit	unit value
v	:=		*Value*
	|	$\mathtt{v}^{\neq \mathtt{m}}$	simple value
	|	m	generic function

Fig. 21. Updated language syntax of Juliette

In what follows, we will use the following facts without a reference:

$$\forall \mathtt{X}_1.\forall \mathtt{X}_2.\exists \mathtt{X}. \quad \mathtt{X}_1[\mathtt{X}_2] = \mathtt{X} \tag{4}$$

$$\forall \mathtt{C}.\forall \mathtt{X}.\exists \mathtt{C}'. \quad \mathtt{X}[\mathtt{C}] = \mathtt{C}' \tag{5}$$

$$\forall \mathtt{C}_1.\forall \mathtt{C}_2.\exists \mathtt{C}. \quad \mathtt{C}_1[\mathtt{C}_2] = \mathtt{C} \tag{6}$$

They are easy to show by induction on X (for the first two) and C (for the last one).

6.1.1 Progress of Juliette programs. To prove Theorem 4.1, we need several auxiliary lemmas below. Roughly, the idea is to show that any Juliette program can be represented in a certain way (Lemma 6.1) that allows for concluding progress (Lemma 6.2). The representation closely follows normal- and error-evaluation rules; it relies on a subset of expressions rdx, which we call redex bases. A redex base gives rise to an "interesting" reduction, in the spirit of β-reduction from the traditional context-based semantics of lambda-calculus. Redex bases are defined in Fig. 22; the figure hints which part of the semantics every redex base corresponds to (normal, error, or both).

Lemma 6.1 (Form of Expression). *For any expression* e*, one of the following holds:*

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

Proof. By induction on the structure of e.

x This is (c): $\mathtt{e} = \square[x]$.

v This is (a): $\mathtt{e} = \mathtt{v}$.

$\mathbf{e}_1 ; \mathbf{e}_2$ By induction hypothesis, $\mathtt{e}_1$ is one of the following:

- $\mathbf{v}_1$ Then $\mathtt{e} = \square[\mathtt{v}_1 ; \mathtt{e}_2]$, which gives us (c).
- $\mathbf{X}_1[\mathbf{m}_1(\overline{\mathbf{v}}_1)]$ Then $\mathtt{e} = (\mathtt{X}_1[\mathtt{m}_1(\overline{\mathtt{v}}_1)]) ; \mathtt{e}_2 = (\mathtt{X}_1 ; \mathtt{e}_2)[\mathtt{m}_1(\overline{\mathtt{v}}_1)] = \mathtt{X}[\mathtt{m}_1(\overline{\mathtt{v}}_1)]$ for $\mathtt{X} = (\mathtt{X}_1 ; \mathtt{e}_2)$, which gives us (b).

$\mathtt{rdx}$	$::=$		*Redex Base*	
	$\mid$	x	variable	(error)
	$\mid$	$\mathtt{v}\,;\,\mathtt{e}$	sequencing	(normal)
	$\mid$	$\delta_l(\overline{\mathtt{v}})$	primop call	(normal/error)
	$\mid$	$\mathtt{v}^{\neq\mathtt{m}}(\overline{\mathtt{v}})$	non-function call	(error)
	$\mid$	$\mathtt{md}$	method definition	(normal)
	$\mid$	$\llparenthesis\, \mathtt{v} \,\rrparenthesis$	value in global context	(normal)
	$\mid$	$\llparenthesis\, \mathtt{v} \,\rrparenthesis_{\mathrm{M}}$	value in table context	(normal)
	$\mid$	$\llparenthesis\, \mathtt{X}\,[\mathtt{m}(\overline{\mathtt{v}})] \,\rrparenthesis$	function call in global context	(normal)
	$\mid$	$\llparenthesis\, \mathtt{X}\,[\mathtt{m}(\overline{\mathtt{v}})] \,\rrparenthesis_{\mathrm{M}}$	function call in table context	(normal/error)

Fig. 22. Redex Bases

$\mathtt{C}_1\,[\mathbf{rdx}_1]$ Then $\mathtt{e} = (\mathtt{C}_1\,[\mathtt{rdx}_1])\,;\,\mathtt{e}_2 = (\mathtt{C}_1\,;\,\mathtt{e}_2)\,[\mathtt{rdx}_1] = \mathtt{C}\,[\mathtt{rdx}_1]$ for $\mathtt{C} = (\mathtt{C}_1\,;\,\mathtt{e}_2)$, which gives us (c).

$\delta_l(\overline{\mathbf{e}})$ Reasoning similarly to $\mathtt{e}_1\,;\,\mathtt{e}_2$, by induction hypotheses, several cases are possible:

- $\mathtt{e} = \delta_l(\overline{\mathtt{v}}) = \square\,[\delta_l(\overline{\mathtt{v}})]$, which is case (c).
- $\mathtt{e} = \delta_l(\overline{\mathtt{v}},\ \mathtt{X}_i\,[\mathtt{m}_i(\overline{\mathtt{v}_i})]\,,\ \overline{\mathtt{e}}) = \delta_l(\overline{\mathtt{v}}\ \mathtt{X}_i\ \overline{\mathtt{e}})\,[\mathtt{m}_i(\overline{\mathtt{v}_i})] = \mathtt{X}\,[\mathtt{m}_i(\overline{\mathtt{v}_i})]$ for $\mathtt{X} = \delta_l(\overline{\mathtt{v}}\ \mathtt{X}_i\ \overline{\mathtt{e}})$, i.e. case (b).
- $\mathtt{e} = \delta_l(\overline{\mathtt{v}},\ \mathtt{C}_i\,[\mathtt{rdx}_i],\ \overline{\mathtt{e}}) = \delta_l(\overline{\mathtt{v}}\ \mathtt{C}_i\ \overline{\mathtt{e}})\,[\mathtt{rdx}_i] = \mathtt{C}\,[\mathtt{rdx}_i]$ for $\mathtt{C} = \delta_l(\overline{\mathtt{v}}\ \mathtt{C}_i\ \overline{\mathtt{e}})$, i.e. case (c).

$\mathbf{e}_c(\overline{\mathbf{e}})$ Similarly to the above, several cases are possible:

- $\mathtt{e} = \mathtt{v}_c(\overline{\mathtt{v}})$
  - $\mathtt{v}_c = \mathtt{v}_c^{\neq\mathtt{m}}$ gives case (c), for $\mathtt{e} = \square\,[\mathtt{v}_c^{\neq\mathtt{m}}(\overline{\mathtt{v}})]$.
  - $\mathtt{v}_c = \mathtt{m}$ gives case (b), for $\mathtt{e} = \square\,[\mathtt{m}(\overline{\mathtt{v}})]$.
- $\mathtt{e} = (\mathtt{X}_c\,[\mathtt{m}_c(\overline{\mathtt{v}_c})])(\overline{\mathtt{e}})$ or $\mathtt{e} = \mathtt{v}_c(\overline{\mathtt{v}},\ \mathtt{X}_i\,[\mathtt{m}_i(\overline{\mathtt{v}_i})]\,,\ \overline{\mathtt{e}})$ give case (b) analogously to the case of $\mathtt{e} = \delta_l(\overline{\mathtt{v}},\ \mathtt{X}_i\,[\mathtt{m}_i(\overline{\mathtt{v}_i})]\,,\ \overline{\mathtt{e}})$.
- $\mathtt{e} = (\mathtt{C}_c\,[\mathtt{rdx}_c])(\overline{\mathtt{e}})$ or $\mathtt{e} = \mathtt{v}_c(\overline{\mathtt{v}},\ \mathtt{C}_i\,[\mathtt{rdx}_i],\ \overline{\mathtt{e}})$ give case (c) analogously to the case of $\mathtt{e} = \delta_l(\overline{\mathtt{v}},\ \mathtt{C}_i\,[\mathtt{rdx}_i],\ \overline{\mathtt{e}})$.

$\mathbf{md}$ This is (c): $\mathtt{e} = \square\,[\mathtt{md}]$.

$\llparenthesis\,\mathbf{e}'\,\rrparenthesis$ By induction hypothesis, $\mathtt{e}'$ is one of the following:

$\mathbf{v}'$ Then $\mathtt{e} = \llparenthesis\,\mathtt{v}'\,\rrparenthesis = \square\,\big[\llparenthesis\,\mathtt{v}'\,\rrparenthesis\big]$, which gives us (c).

$\mathbf{X}'\,\big[\mathbf{m}'(\overline{\mathbf{v}'})\big]$ Then $\mathtt{e} = \llparenthesis\,\mathtt{X}'\,\big[\mathtt{m}'(\overline{\mathtt{v}'})\big]\,\rrparenthesis = \square\,\big[\llparenthesis\,\mathtt{X}'\,\big[\mathtt{m}'(\overline{\mathtt{v}'})\big]\,\rrparenthesis\big]$, which gives us (c).

$\mathbf{C}'\,[\mathbf{rdx}']$ Then $\mathtt{e} = \llparenthesis\,\mathtt{C}'\,[\mathtt{rdx}']\,\rrparenthesis = \mathtt{C}\,[\mathtt{rdx}']$ for $\mathtt{C} = \square\,\big[\llparenthesis\,\mathtt{C}'\,\rrparenthesis\big]$, which gives us (c).

$\llparenthesis\,\mathbf{e}'\,\rrparenthesis_{\mathrm{M}}$ Reasoning similarly to $\llparenthesis\,\mathtt{e}'\,\rrparenthesis$, we have case (c).

□

Along with world evaluation context $\mathtt{C}$, redex base $\mathtt{rdx}$ makes a redex, i.e. expression $\mathtt{C}\,[\mathtt{rdx}]$ that can make a step.

Lemma 6.2 (Redex Steps). *For any redex base* $\mathtt{rdx}$*, world context* $\mathtt{C}$*, and global table* M_g*,*

(a) either $\langle \mathrm{M}_g, \mathtt{C}\,[\mathtt{rdx}]\rangle \ \longrightarrow\ \langle \mathrm{M}_g', \mathtt{C}\,[\mathtt{e}']\rangle$,

(b) or $\quad \mathrm{M}_g \ \vdash\ \mathtt{C}\,[\mathtt{rdx}] \ \longrightarrow\ \mathtt{error}$.

Proof. By case analysis on $\mathtt{rdx}$, using rules from Fig. ?? and Fig. ??.

x By E-VarErr, $\mathrm{M}_g \ \vdash\ \mathtt{C}\,[x] \ \longrightarrow\ \mathtt{error}$, which is case (b).

$\mathbf{v}\,;\,\mathbf{e}$ By E-Seq, $\langle \mathrm{M}_g, \mathtt{C}\,[\mathtt{v}\,;\,\mathtt{e}]\rangle \ \longrightarrow\ \langle \mathrm{M}_g, \mathtt{C}\,[\mathtt{e}]\rangle$, which is case (a).

$\delta_l(\overline{\mathbf{v}})$ Depending on $\Delta(l, \overline{\mathtt{v}})$,

- either by E-Primop, $\langle \mathrm{M}_g, \mathtt{C}\,[\delta_l(\overline{\mathtt{v}})]\rangle \ \longrightarrow\ \langle \mathrm{M}_g, \mathtt{C}\,[\mathtt{v}']\rangle$, i.e. case (a),

- or by E-PrimopErr $\mathrm{M}_g \vdash \mathtt{C}\left[\delta_l(\overline{\mathtt{v}})\right] \longrightarrow \mathtt{error}$, i.e. case (b).

$\mathbf{v}^{\neq \mathbf{m}}(\overline{\mathbf{v}})$ By E-CalleeErr, $\mathrm{M}_g \vdash \mathtt{C}\left[\mathtt{v}^{\neq \mathtt{m}}(\overline{\mathtt{v}})\right] \longrightarrow \mathtt{error}$, i.e. case (b).

md By E-MD, $\langle \mathrm{M}_g, \mathtt{C}\left[\mathtt{md}\right]\rangle \longrightarrow \langle \mathrm{M}_g \bullet \mathtt{md}, \mathtt{C}\left[\mathtt{m}\right]\rangle$, which is case (a).

$(\!|\, \mathbf{v} \,|\!)$ By E-ValGlobal, $\langle \mathrm{M}_g, \mathtt{C}\left[(\!|\, \mathtt{v} \,|\!)\right]\rangle \longrightarrow \langle \mathrm{M}_g, \mathtt{C}\left[\mathtt{v}\right]\rangle$, which is case (a).

$(\!|\, \mathbf{v} \,|\!)_{\mathrm{M}}$ By E-ValLocal, $\langle \mathrm{M}_g, \mathtt{C}\left[(\!|\, \mathtt{v} \,|\!)_{\mathrm{M}}\right]\rangle \longrightarrow \langle \mathrm{M}_g, \mathtt{C}\left[\mathtt{v}\right]\rangle$, which is case (a).

$(\!|\, \mathbf{X}\left[\mathbf{m}(\overline{\mathbf{v}})\right] \,|\!)$ By E-CallGlobal, $\langle \mathrm{M}_g, \mathtt{C}\left[(\!|\, \mathtt{X}\left[\mathtt{m}(\overline{\mathtt{v}})\right] \,|\!)\right]\rangle \longrightarrow \langle \mathrm{M}_g, \mathtt{C}\left[(\!|\, \mathtt{X}\left[(\!|\, \mathtt{m}(\overline{\mathtt{v}}) \,|\!)_{\mathrm{M}_g}\right] \,|\!)\right]\rangle$, i.e. case (a).

$(\!|\, \mathbf{X}\left[\mathbf{m}(\overline{\mathbf{v}})\right] \,|\!)_{\mathrm{M}}$ Depending on the result of method call resolution $\mathtt{m}(\overline{\mathtt{v}})$ for M,
- either E-CallLocal gives us case (a),
- or E-CallErr gives us case (b).

□

Finally, we can prove the Progress Theorem 4.1. Let us recall its statement:
For any program p *and method table* M_g*, one of the following holds:*

(a) $\langle \mathrm{M}_g, \mathtt{p}\rangle \xrightarrow{c} \langle \mathrm{M}_g', \mathtt{r}\rangle$; or

(b) $\langle \mathrm{M}_g, \mathtt{p}\rangle \xrightarrow{c} \langle \mathrm{M}_g', \mathtt{p}'\rangle$.

Proof. Remember that r is either v or error. From $\mathtt{p} = (\!|\, \mathtt{e} \,|\!)$ and Lemma 4.3, we know that one of the following holds:

(1) $\mathtt{p} = (\!|\, \mathtt{v} \,|\!)$. This gives us case (a) by E-ValGlobal:

$$\frac{\langle \mathrm{M}_g, (\!|\, \mathtt{v} \,|\!)\rangle \longrightarrow \langle \mathrm{M}_g, \mathtt{v}\rangle}{\langle \mathrm{M}_g, (\!|\, \mathtt{v} \,|\!)\rangle \xrightarrow{c} \langle \mathrm{M}_g, \mathtt{v}\rangle.}\ \textsc{E-Normal}$$

(2) $\mathtt{p} = (\!|\, \mathtt{X}\left[\mathtt{m}(\overline{\mathtt{v}})\right] \,|\!)$. This gives us case (b) by E-CallGlobal:

$$\frac{\langle \mathrm{M}_g, (\!|\, \mathtt{X}\left[\mathtt{m}(\overline{\mathtt{v}})\right] \,|\!)\rangle \longrightarrow \langle \mathrm{M}_g, (\!|\, \mathtt{X}\left[(\!|\, \mathtt{m}(\overline{\mathtt{v}}) \,|\!)_{\mathrm{M}_g}\right] \,|\!)\rangle}{\langle \mathrm{M}_g, (\!|\, \mathtt{X}\left[\mathtt{m}(\overline{\mathtt{v}})\right] \,|\!)\rangle \xrightarrow{c} \langle \mathrm{M}_g, (\!|\, \mathtt{X}\left[(\!|\, \mathtt{m}(\overline{\mathtt{v}}) \,|\!)_{\mathrm{M}_g}\right] \,|\!)\rangle.}\ \textsc{E-Normal}$$

(3) $\mathtt{p} = (\!|\, \mathtt{C}\left[\mathtt{rdx}\right] \,|\!) = \mathtt{C}'\left[\mathtt{rdx}\right]$ for $\mathtt{C}' = (\!|\, \mathtt{C} \,|\!)$. By Lemma 6.2, one of the following holds:
- $\langle \mathrm{M}_g, \mathtt{C}'\left[\mathtt{rdx}\right]\rangle \longrightarrow \langle \mathrm{M}_g', \mathtt{C}'\left[\mathtt{e}'\right]\rangle$. This gives us (b) because $(\!|\, \mathtt{C}\left[\mathtt{e}'\right] \,|\!) = \mathtt{p}'$:

$$\frac{\langle \mathrm{M}_g, (\!|\, \mathtt{C}\left[\mathtt{rdx}\right] \,|\!)\rangle \longrightarrow \langle \mathrm{M}_g', (\!|\, \mathtt{C}\left[\mathtt{e}'\right] \,|\!)\rangle}{\langle \mathrm{M}_g, (\!|\, \mathtt{C}\left[\mathtt{rdx}\right] \,|\!)\rangle \xrightarrow{c} \langle \mathrm{M}_g', (\!|\, \mathtt{C}\left[\mathtt{e}'\right] \,|\!)\rangle.}\ \textsc{E-Normal}$$

- $\mathrm{M}_g \vdash \mathtt{C}'\left[\mathtt{rdx}\right] \longrightarrow \mathtt{error}$. This gives us (a):

$$\frac{\mathrm{M}_g \vdash (\!|\, \mathtt{C}\left[\mathtt{rdx}\right] \,|\!) \longrightarrow \mathtt{error}}{\langle \mathrm{M}_g, (\!|\, \mathtt{C}\left[\mathtt{rdx}\right] \,|\!)\rangle \xrightarrow{c} \langle \mathrm{M}_g, \mathtt{error}\rangle.}\ \textsc{E-Error}$$

□

6.1.2 Canonical representation of expressions. In this section, we show that:

(1) The form of expression e described in Lemma 6.1 ($\mathtt{v}$, $\mathtt{X}\left[\mathtt{m}(\overline{\mathtt{v}})\right]$, or $\mathtt{C}\left[\mathtt{rdx}\right]$) is in fact unique.

(2) Any normal- or error-evaluation step of e is determined by its $\mathtt{C}\left[\mathtt{rdx}\right]$ representation.

To simplify proofs, we will use an auxiliary syntax rep to distinguish between expression forms and an auxiliary judgment ***Can*** (e **as** rep) to remember how those forms are built. These new definitions are provided in Fig. 23.

Lemma 6.3 (Reconstruction from Canonical Forms). *For all expressions* e*, the following holds:*

(a) ***Can*** (e **as** $\mathcal{V}(\mathtt{v})$) $\quad\Longrightarrow\quad \mathtt{e} = \mathtt{v}$.

$$
\begin{array}{lcll}
\texttt{rep} & ::= & & \textit{Form of expression} \\
 & | & \mathcal{V}(\texttt{v}) & \\
 & | & \mathcal{X}(\texttt{X},\ \texttt{m}(\overline{\texttt{v}})) & \\
 & | & \mathcal{C}(\texttt{C},\ \texttt{rdx}) &
\end{array}
$$

$$\boldsymbol{\mathit{Can}}\,(\texttt{e}\ \textbf{as}\ \texttt{rep}) \qquad \textit{Canonical form}$$

$$
\textsc{CN-Val}\ \frac{}{\boldsymbol{\mathit{Can}}\,(\texttt{v}\ \textbf{as}\ \mathcal{V}(\texttt{v}))}
\qquad
\textsc{CN-Call}\ \frac{}{\boldsymbol{\mathit{Can}}\,(\texttt{m}(\overline{\texttt{v}})\ \textbf{as}\ \mathcal{X}(\square,\ \texttt{m}(\overline{\texttt{v}})))}
\qquad
\textsc{CN-Redex}\ \frac{}{\boldsymbol{\mathit{Can}}\,(\texttt{rdx}\ \textbf{as}\ \mathcal{C}(\square,\ \texttt{rdx}))}
$$

$$
\textsc{CN-SeqX}\ \frac{\boldsymbol{\mathit{Can}}\,(\texttt{e}_1\ \textbf{as}\ \mathcal{X}(\texttt{X}_1,\ \texttt{m}(\overline{\texttt{v}})))}{\boldsymbol{\mathit{Can}}\,(\texttt{e}_1\,;\,\texttt{e}_2\ \textbf{as}\ \mathcal{X}(\texttt{X}_1\,;\,\texttt{e}_2,\ \texttt{m}(\overline{\texttt{v}})))}
\qquad
\textsc{CN-SeqC}\ \frac{\boldsymbol{\mathit{Can}}\,(\texttt{e}_1\ \textbf{as}\ \mathcal{C}(\texttt{C}_1,\ \texttt{rdx}))}{\boldsymbol{\mathit{Can}}\,(\texttt{e}_1\,;\,\texttt{e}_2\ \textbf{as}\ \mathcal{C}(\texttt{C}_1\,;\,\texttt{e}_2,\ \texttt{rdx}))}
$$

$$
\textsc{CN-PrimopX}\ \frac{\boldsymbol{\mathit{Can}}\,(\texttt{e}_i\ \textbf{as}\ \mathcal{X}(\texttt{X}_i,\ \texttt{m}(\overline{\texttt{v}})))}{\boldsymbol{\mathit{Can}}\,(\delta_l(\overline{\texttt{v}},\ \texttt{e}_i,\ \overline{\texttt{e}})\ \textbf{as}\ \mathcal{X}(\delta_l(\overline{\texttt{v}},\ \texttt{X}_i,\ \overline{\texttt{e}}),\ \texttt{m}(\overline{\texttt{v}})))}
\qquad
\textsc{CN-PrimopC}\ \frac{\boldsymbol{\mathit{Can}}\,(\texttt{e}_i\ \textbf{as}\ \mathcal{C}(\texttt{C}_i,\ \texttt{rdx}))}{\boldsymbol{\mathit{Can}}\,(\delta_l(\overline{\texttt{v}},\ \texttt{e}_i,\ \overline{\texttt{e}})\ \textbf{as}\ \mathcal{C}(\delta_l(\overline{\texttt{v}},\ \texttt{C}_i,\ \overline{\texttt{e}}),\ \texttt{rdx}))}
$$

$$
\textsc{CN-CallX}\ \frac{\boldsymbol{\mathit{Can}}\,(\texttt{e}_i\ \textbf{as}\ \mathcal{X}(\texttt{X}_i,\ \texttt{m}(\overline{\texttt{v}})))}{\boldsymbol{\mathit{Can}}\,(\texttt{m}(\overline{\texttt{v}},\ \texttt{e}_i,\ \overline{\texttt{e}})\ \textbf{as}\ \mathcal{X}(\delta_l(\overline{\texttt{v}},\ \texttt{X}_i,\ \overline{\texttt{e}}),\ \texttt{m}(\overline{\texttt{v}})))}
\qquad
\textsc{CN-CallC}\ \frac{\boldsymbol{\mathit{Can}}\,(\texttt{e}_i\ \textbf{as}\ \mathcal{C}(\texttt{C}_i,\ \texttt{rdx}))}{\boldsymbol{\mathit{Can}}\,(\texttt{m}(\overline{\texttt{v}},\ \texttt{e}_i,\ \overline{\texttt{e}})\ \textbf{as}\ \mathcal{C}(\texttt{m}(\overline{\texttt{v}},\ \texttt{C}_i,\ \overline{\texttt{e}}),\ \texttt{rdx}))}
$$

$$
\textsc{CN-EvalGlobalC}\ \frac{\boldsymbol{\mathit{Can}}\,(\texttt{e}\ \textbf{as}\ \mathcal{C}(\texttt{C},\ \texttt{rdx}))}{\boldsymbol{\mathit{Can}}\,((\!|\,\texttt{e}\,|\!)\ \textbf{as}\ \mathcal{C}((\!|\,\texttt{C}\,|\!),\ \texttt{rdx}))}
\qquad
\textsc{CN-EvalLocalC}\ \frac{\boldsymbol{\mathit{Can}}\,(\texttt{e}\ \textbf{as}\ \mathcal{C}(\texttt{C},\ \texttt{rdx}))}{\boldsymbol{\mathit{Can}}\,((\!|\,\texttt{e}\,|\!)_{\mathrm{M}}\ \textbf{as}\ \mathcal{C}((\!|\,\texttt{C}\,|\!)_{\mathrm{M}},\ \texttt{rdx}))}
$$

Fig. 23. Canonical forms of internal Juliette expressions

(b) $\boldsymbol{\mathit{Can}}\,(\texttt{e}\ \textbf{as}\ \mathcal{X}(\texttt{X},\ \texttt{m}(\overline{\texttt{v}}))) \implies \texttt{e} = \texttt{X}\,[\texttt{m}(\overline{\texttt{v}})]$.
(c) $\boldsymbol{\mathit{Can}}\,(\texttt{e}\ \textbf{as}\ \mathcal{C}(\texttt{C},\ \texttt{rdx})) \implies \texttt{e} = \texttt{C}\,[\texttt{rdx}]$.

Proof. By induction on $\boldsymbol{\mathit{Can}}\,(\texttt{e}\ \textbf{as}\ \texttt{rep})$.

(a) This case is trivial: there is only one constructor of $\boldsymbol{\mathit{Can}}$ with $\mathcal{V}$ expression form, CN-Value, and thus $\texttt{e} = \texttt{v}$ by inversion.

(b) There are multiple $\boldsymbol{\mathit{Can}}$ constructors with $\mathcal{X}$ forms.

CN-Call Another trivial case. By inversion, $\texttt{e} = \texttt{m}(\overline{\texttt{v}})$, and also $\square\,[\texttt{m}(\overline{\texttt{v}})] = \texttt{m}(\overline{\texttt{v}})$.

CN-SeqX By inversion, $\texttt{e} = (\texttt{e}_1\,;\,\texttt{e}_2)$. By induction hypothesis, $\texttt{e}_1 = \texttt{X}_1\,[\texttt{m}(\overline{\texttt{v}})]$. Therefore,

$$(\texttt{X}_1\,;\,\texttt{e}_2)\,[\texttt{m}(\overline{\texttt{v}})] = \texttt{X}_1\,[\texttt{m}(\overline{\texttt{v}})]\ ;\,\texttt{e}_2 = \texttt{e}_1\,;\,\texttt{e}_2.$$

... Other cases are similar to CN-SeqX.

(c) There are multiple $\boldsymbol{\mathit{Can}}$ constructors with $\mathcal{C}$ forms.

CN-Redex This is a trivial case. By inversion, $\texttt{e} = \texttt{rdx}$, and also $\square\,[\texttt{rdx}] = \texttt{rdx}$.

CN-SeqC By inversion, $\texttt{e} = (\texttt{e}_1\,;\,\texttt{e}_2)$. By induction hypothesis, $\texttt{e}_1 = \texttt{C}_1\,[\texttt{rdx}]$. Therefore,

$$(\texttt{C}_1\,;\,\texttt{e}_2)\,[\texttt{rdx}] = \texttt{C}_1\,[\texttt{rdx}]\ ;\,\texttt{e}_2 = \texttt{e}_1\,;\,\texttt{e}_2.$$

... Other cases are similar to CN-SeqC.

□

Lemma 6.4 ($\mathtt{X}$ preserves Canonical $\mathcal{C}$). *For all contexts* $\mathtt{X}$ *and expressions* $\mathtt{e}$,

$$\textbf{\textit{Can}}\,(\mathtt{e}\ \textbf{as}\ \mathcal{C}(\mathtt{C},\ \mathtt{rdx})) \quad\Longrightarrow\quad \textbf{\textit{Can}}\,(\mathtt{X}\,[\mathtt{e}]\ \textbf{as}\ \mathcal{C}(\mathtt{X}\,[\mathtt{C}],\ \mathtt{rdx}))\,.$$

Proof. By induction on $\mathtt{X}$.

$\square$ Since $\square\,[\mathtt{e}] = \mathtt{e}$ and $\square\,[\mathtt{C}] = \mathtt{C}$, $\textbf{\textit{Can}}\,(\square\,[\mathtt{e}]\ \textbf{as}\ \mathcal{C}(\square\,[\mathtt{C}],\ \mathtt{rdx}))$ by assumption.

$\mathbf{X}_1\,;\,\mathbf{e}_2$ By definition, $(\mathtt{X}_1\,;\,\mathtt{e}_2)\,[\mathtt{e}] = \mathtt{X}_1\,[\mathtt{e}]\ ;\ \mathtt{e}_2$ and $(\mathtt{X}_1\,;\,\mathtt{e}_2)\,[\mathtt{C}] = \mathtt{X}_1\,[\mathtt{C}]\ ;\ \mathtt{e}_2$. By induction hypothesis for $\mathtt{X}_1$, $\textbf{\textit{Can}}\,(\mathtt{X}_1\,[\mathtt{e}]\ \textbf{as}\ \mathcal{C}(\mathtt{X}_1\,[\mathtt{C}],\ \mathtt{rdx}))$. Then:

$$\frac{\textbf{\textit{Can}}\,(\mathtt{X}_1\,[\mathtt{e}]\ \textbf{as}\ \mathcal{C}(\mathtt{X}_1\,[\mathtt{C}],\ \mathtt{rdx}))}{\textbf{\textit{Can}}\,(\mathtt{X}_1\,[\mathtt{e}]\ ;\ \mathtt{e}_2\ \textbf{as}\ \mathcal{C}(\mathtt{X}_1\,[\mathtt{C}]\ ;\ \mathtt{e}_2,\ \mathtt{rdx}))\,.}\ \textsc{CN-SeqC}$$

. . . Other cases are similar.

$\square$

Lemma 6.5 ($\mathtt{C}$ preserves Canonical $\mathcal{C}$). *For all contexts* $\mathtt{C}'$ *and expressions* $\mathtt{e}$,

$$\textbf{\textit{Can}}\,(\mathtt{e}\ \textbf{as}\ \mathcal{C}(\mathtt{C},\ \mathtt{rdx})) \quad\Longrightarrow\quad \textbf{\textit{Can}}\,(\mathtt{C}'\,[\mathtt{e}]\ \textbf{as}\ \mathcal{C}(\mathtt{C}'\,[\mathtt{C}],\ \mathtt{rdx}))\,.$$

Proof. By induction on $\mathtt{C}'$.

$\mathbf{X}$ This case is covered by Lemma 6.4.

$\mathbf{X}\left[(\!|\,\mathbf{C}''\,|\!)\right]$ By definition, $(\mathtt{X}\left[(\!|\,\mathtt{C}''\,|\!)\right])\,[\mathtt{e}] = \mathtt{X}\left[(\!|\,\mathtt{C}''\,[\mathtt{e}]\,|\!)\right]$ and $(\mathtt{X}\left[(\!|\,\mathtt{C}''\,|\!)\right])\,[\mathtt{C}] = \mathtt{X}\left[(\!|\,\mathtt{C}''\,[\mathtt{C}]\,|\!)\right]$. By induction hypothesis for $\mathtt{C}''$, $\textbf{\textit{Can}}\,(\mathtt{C}''\,[\mathtt{e}]\ \textbf{as}\ \mathcal{C}(\mathtt{C}''\,[\mathtt{C}],\ \mathtt{rdx}))$. Then:

$$\frac{\textbf{\textit{Can}}\,(\mathtt{C}''\,[\mathtt{e}]\ \textbf{as}\ \mathcal{C}(\mathtt{C}''\,[\mathtt{C}],\ \mathtt{rdx}))}{\textbf{\textit{Can}}\,((\!|\,\mathtt{C}''\,[\mathtt{e}]\,|\!)\ \textbf{as}\ \mathcal{C}((\!|\,\mathtt{C}''\,[\mathtt{C}]\,|\!),\ \mathtt{rdx}))\,,}\ \textsc{CN-EvalGlobalC}$$

and by Lemma 6.4,

$$\textbf{\textit{Can}}\,(\mathtt{X}\left[(\!|\,\mathtt{C}''\,[\mathtt{e}]\,|\!)\right]\ \textbf{as}\ \mathcal{C}(\mathtt{X}\left[(\!|\,\mathtt{C}''\,[\mathtt{C}]\,|\!)\right],\ \mathtt{rdx}))\,.$$

$\mathbf{X}\left[(\!|\,\mathbf{C}''\,|\!)_{\mathrm{M}}\right]$ Similar to the previous case.

$\square$

Lemma 6.6 ($\mathtt{X}\,[\mathtt{m}(\overline{\mathtt{v}})]$ is Canonical). *For all* $\mathtt{X}$ *and* $\mathtt{m}(\overline{\mathtt{v}})$,

$$\textbf{\textit{Can}}\,(\mathtt{X}\,[\mathtt{m}(\overline{\mathtt{v}})]\ \textbf{as}\ \mathcal{X}(\mathtt{X},\ \mathtt{m}(\overline{\mathtt{v}})))\,.$$

Proof. By induction on $\mathtt{X}$.

$\square$ In this case, $\square\,[\mathtt{m}(\overline{\mathtt{v}})] = \mathtt{m}(\overline{\mathtt{v}})$, and by $\textsc{cn-Call}$, $\textbf{\textit{Can}}\,(\mathtt{m}(\overline{\mathtt{v}})\ \textbf{as}\ \mathcal{X}(\square,\ \mathtt{m}(\overline{\mathtt{v}})))$.

$\mathbf{X}_1\,;\,\mathbf{e}_2$ By definition, $(\mathtt{X}_1\,;\,\mathtt{e}_2)\,[\mathtt{m}(\overline{\mathtt{v}})] = (\mathtt{X}_1\,[\mathtt{m}(\overline{\mathtt{v}})]\ ;\ \mathtt{e}_2)$. By induction hypothesis for $\mathtt{X}_1$ and $\textbf{\textit{Can}}$ constructor:

$$\frac{\textbf{\textit{Can}}\,(\mathtt{X}_1\,[\mathtt{m}(\overline{\mathtt{v}})]\ \textbf{as}\ \mathcal{X}(\mathtt{X}_1,\ \mathtt{m}(\overline{\mathtt{v}})))}{\textbf{\textit{Can}}\,(\mathtt{X}_1\,[\mathtt{m}(\overline{\mathtt{v}})]\ ;\ \mathtt{e}_2\ \textbf{as}\ \mathcal{X}(\mathtt{X}_1\,;\,\mathtt{e}_2,\ \mathtt{m}(\overline{\mathtt{v}})))\,.}\ \textsc{CN-SeqX}$$

. . . Other cases are similar.

$\square$

Lemma 6.7 ($\mathtt{X}\,[\mathtt{rdx}]$ is Canonical). *For all* $\mathtt{X}$ *and* $\mathtt{rdx}$,

$$\textbf{\textit{Can}}\,(\mathtt{X}\,[\mathtt{rdx}]\ \textbf{as}\ \mathcal{C}(\mathtt{X},\ \mathtt{rdx}))\,.$$

Proof. By induction on $\mathtt{X}$.

$\square$ In this case, $\square\,[\mathtt{rdx}] = \mathtt{rdx}$, and by $\textsc{cn-Redex}$, $\textbf{\textit{Can}}\,(\mathtt{rdx}\ \textbf{as}\ \mathcal{C}(\square,\ \mathtt{rdx}))$.

$\mathbf{X}_1$; $\mathbf{e}_2$ By definition, $(\mathtt{X}_1 ; \mathtt{e}_2)\,[\mathtt{rdx}] = (\mathtt{X}_1\,[\mathtt{rdx}] ; \mathtt{e}_2)$. By induction hypothesis for $\mathtt{X}_1$ and ***Can*** constructor:

$$\frac{\textbf{\textit{Can}}\,(\mathtt{X}_1\,[\mathtt{rdx}]\ \textbf{as}\ \mathcal{C}(\mathtt{X}_1,\ \mathtt{rdx}))}{\textbf{\textit{Can}}\,(\mathtt{X}_1\,[\mathtt{rdx}]\ ;\ \mathtt{e}_2\ \textbf{as}\ \mathcal{C}(\mathtt{X}_1 ; \mathtt{e}_2,\ \mathtt{rdx}))\,.}\ \textsc{CN-SeqC}$$

... Other cases are similar.

□

Lemma 6.8 ($\mathtt{C}\,[\mathtt{rdx}]$ is Canonical). *For all* $\mathtt{C}$ *and* $\mathtt{rdx}$,

$$\textbf{\textit{Can}}\,(\mathtt{C}\,[\mathtt{rdx}]\ \textbf{as}\ \mathcal{C}(\mathtt{C},\ \mathtt{rdx}))\,.$$

Proof. By induction on $\mathtt{C}$.

X This case is covered by Lemma 6.7.

$\mathbf{X}\left[(\!| \mathbf{C}' |\!)\right]$ By definition, $(\mathtt{X}\left[(\!|\mathtt{C}'|\!)\right])\,[\mathtt{rdx}] = \mathtt{X}\left[(\!|\mathtt{C}'\,[\mathtt{rdx}]\,|\!)\right]$. By induction hypothesis for $\mathtt{C}'$ and ***Can*** constructor:

$$\frac{\textbf{\textit{Can}}\,(\mathtt{C}'\,[\mathtt{rdx}]\ \textbf{as}\ \mathcal{C}(\mathtt{C}',\ \mathtt{rdx}))}{\textbf{\textit{Can}}\,((\!|\mathtt{C}'\,[\mathtt{rdx}]\,|\!)\ \textbf{as}\ \mathcal{C}((\!|\mathtt{C}'|\!),\ \mathtt{rdx}))\,.}\ \textsc{CN-EvalGlobalC}$$

Then, by Lemma 6.4,

$$\textbf{\textit{Can}}\,(\mathtt{X}\left[(\!|\mathtt{C}'\,[\mathtt{rdx}]\,|\!)\right]\ \textbf{as}\ \mathcal{C}(\mathtt{X}\left[(\!|\mathtt{C}'|\!)\right],\ \mathtt{rdx}))\,.$$

$\mathbf{X}\left[(\!|\mathbf{C}'|\!)_{\mathrm{M}}\right]$ Similar to the previous case.

□

Lemma 6.9 (Canonical Form is Unique). *For all expressions* $\mathtt{e}$ *and representations* $\mathtt{rep}_1, \mathtt{rep}_2$,

$$\textbf{\textit{Can}}\,(\mathtt{e}\ \textbf{as}\ \mathtt{rep}_1) \land \textbf{\textit{Can}}\,(\mathtt{e}\ \textbf{as}\ \mathtt{rep}_2) \quad\Longrightarrow\quad \mathtt{rep}_1 = \mathtt{rep}_2.$$

Proof. By induction on $\mathtt{e}$.

v There is only one ***Can*** constructor for $\mathtt{v}$, CN-Value. Therefore, $\mathtt{rep}_1 = \mathcal{V}(\mathtt{v}) = \mathtt{rep}_2$.

x There is only one ***Can*** constructor for x, CN-Redex. Therefore, $\mathtt{rep}_1 = \mathcal{C}(\square, x) = \mathtt{rep}_2$.

$\mathbf{e}_1$; $\mathbf{e}_2$ By inversion of $\textbf{\textit{Can}}\,(\mathtt{e}_1 ; \mathtt{e}_2\ \textbf{as}\ \mathtt{rep}_1)$, three cases are possible.

(1) $\mathtt{e}$ is a redex with $\mathtt{e}_1 = \mathtt{v}_1$ and $\textbf{\textit{Can}}\,(\mathtt{v}_1 ; \mathtt{e}_2\ \textbf{as}\ \mathcal{C}(\square, \mathtt{v}_1 ; \mathtt{e}_2))$, and thus $\textbf{\textit{Can}}\,(\mathtt{e}_1\ \textbf{as}\ \mathcal{V}(\mathtt{v}_1))$.
(2) $\textbf{\textit{Can}}\,(\mathtt{e}_1 ; \mathtt{e}_2\ \textbf{as}\ \mathcal{X}(\mathtt{X}_1 ; \mathtt{e}_2,\ \mathtt{m}(\overline{\mathtt{v}})))$ with $\textbf{\textit{Can}}\,(\mathtt{e}_1\ \textbf{as}\ \mathcal{X}(\mathtt{X}_1,\ \mathtt{m}(\overline{\mathtt{v}})))$.
(3) $\textbf{\textit{Can}}\,(\mathtt{e}_1 ; \mathtt{e}_2\ \textbf{as}\ \mathcal{C}(\mathtt{C}_1 ; \mathtt{e}_2,\ \mathtt{rdx}))$ with $\textbf{\textit{Can}}\,(\mathtt{e}_1\ \textbf{as}\ \mathcal{C}(\mathtt{C}_1,\ \mathtt{rdx}))$.

By inversion of $\textbf{\textit{Can}}\,(\mathtt{e}_1 ; \mathtt{e}_2\ \textbf{as}\ \mathtt{rep}_2)$, similar three cases are possible, with different representations of $\mathtt{e}_1$. However, by induction hypothesis, we know that the canonical representation of $\mathtt{e}_1$ is unique. And thus the representations of $\mathtt{e}_1 ; \mathtt{e}_2$ will coincide too.

... Other cases are similar.

□

Theorem 6.10 (Unique Canonical Representation of Expression). *Any expression* $\mathtt{e}$ *can be uniquely represented in one of the following ways:*

(a) $\mathtt{e} = \mathtt{v}$ *with* $\textbf{\textit{Can}}\,(\mathtt{e}\ \textbf{as}\ \mathcal{V}(\mathtt{v}))$*; or*
(b) $\mathtt{e} = \mathtt{X}\,[\mathtt{m}(\overline{\mathtt{v}})]$ *with* $\textbf{\textit{Can}}\,(\mathtt{e}\ \textbf{as}\ \mathcal{X}(\mathtt{X},\ \mathtt{m}(\overline{\mathtt{v}})))$*; or*
(c) $\mathtt{e} = \mathtt{C}\,[\mathtt{rdx}]$ *with* $\textbf{\textit{Can}}\,(\mathtt{e}\ \textbf{as}\ \mathcal{C}(\mathtt{C},\ \mathtt{rdx}))$.

Proof. By Lemma 6.1, we know that at least one of the $\mathtt{e} = \ldots$ conditions above holds. Next, by CN-Value, Lemma 6.6, and Lemma 6.8, $\mathtt{e}$ has canonical representations corresponding to those conditions. But by Lemma 6.9, the canonical representation is unique. Thus, for example, if $\mathtt{e} = \mathtt{C}_1\,[\mathtt{rdx}_1]$ and $\mathtt{e} = \mathtt{C}_2\,[\mathtt{rdx}_2]$, then $\mathtt{C}_1 = \mathtt{C}_2$ and $\mathtt{rdx}_1 = \mathtt{rdx}_2$. □

We will use this theorem implicitly in various proofs that follow.

6.1.3 *Determinism of semantics.*

Lemma 6.11 (Only Redex Steps). *For all expressions* $\mathtt{e}$ *and method tables* M,

$$\langle \mathrm{M}, \mathtt{e}\rangle \xrightarrow{c} \langle \mathrm{M}', \mathtt{i}\rangle \iff \mathtt{e} = \mathtt{C}\,[\mathtt{rdx}].$$

Proof. The left-to-right direction is easy to establish by analyzing normal- and error-evaluation rules. The right-to-left direction can be established by case analysis on $\mathtt{rdx}$. □

Theorem 6.12 (Juliette Semantics is Deterministic). *For all expressions* $\mathtt{e}$ *and method tables* M*, if* $\mathtt{e}$ *can make a normal- or error-evaluation step, the step is unique.*

Proof. By Lemma 6.11, we know that if $\mathtt{e}$ can make a step, then exist some $\mathtt{C}$ and $\mathtt{rdx}$ such that $\mathtt{e} = \mathtt{C}\,[\mathtt{rdx}]$. By Theorem 6.10, we know that such a representation is unique. Finally, by case analysis on $\mathtt{rdx}$, we can see that for all redexes except for $\delta_l(\overline{\mathtt{v}})$ and $⦇\,\mathtt{X}\,[\mathtt{m}(\overline{\mathtt{v}})]\,⦈_{\mathrm{M}}$, there is exactly one (normal- or error-evaluation) rule applicable. For $\delta_l(\overline{\mathtt{v}})$ and $⦇\,\mathtt{X}\,[\mathtt{m}(\overline{\mathtt{v}})]\,⦈_{\mathrm{M}}$, there are two rules for each, but their premises are incompatible. Thus, for any $\mathtt{C}\,[\mathtt{rdx}]$, exactly one rule is applicable. □

Lemma 6.13 (Context Irrelevance). *For all* $\mathtt{rdx}$, $\mathtt{C}$, $\mathtt{C}'$, M*, the following holds:*

(1) For all M′, $\mathtt{e}'$,

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

(2)

$$\mathrm{M} \vdash \mathtt{C}\,[\mathtt{rdx}] \to \mathtt{error} \iff \mathrm{M} \vdash \mathtt{C}'\,[\mathtt{rdx}] \to \mathtt{error}.$$

Proof. By analyzing reduction 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 $\mathtt{C}'\,[\mathtt{rdx}]$ ($\mathtt{C}\,[\mathtt{rdx}]$). □

6.1.4 *Preservation of concrete typing.*

Lemma 6.14. *For all* $\mathtt{e}$, σ, M, M′, $\mathtt{e}'$,

$$\vdash \mathtt{e} : \sigma \;\wedge\; \langle \mathrm{M}, \mathtt{e}\rangle \to \langle \mathrm{M}', \mathtt{e}'\rangle \implies \vdash \mathtt{e}' : \sigma.$$

Proof. By induction on derivation of $\vdash \mathtt{e} : \sigma$.

T-Val By inversion, $\mathtt{e} = \mathtt{v}$ and $\vdash \mathtt{v} : \sigma$, but $\mathtt{v}$ cannot make a step.

T-MD By inversion, $\mathtt{e}$ is a method definition $\triangleleft\, \mathtt{m}(\overline{x :: \tau}) = \mathtt{e} \,\triangleright$with $\vdash \triangleleft\, \mathtt{m}(\overline{x :: \tau}) = \mathtt{e} \,\triangleright : \mathbb{f}_{\mathtt{m}}$. By E-MD, $\langle \mathrm{M}, \triangleleft\, \mathtt{m}(\overline{x :: \tau}) = \mathtt{e}\,\triangleright\rangle \to \langle \mathrm{M} \bullet \triangleleft\, \mathtt{m}(\overline{x :: \tau}) = \mathtt{e}\,\triangleright, \mathtt{m}\rangle$, and $\vdash \mathtt{m} : \mathbb{f}_{\mathtt{m}}$ because $\mathbf{typeof}(\mathtt{m}) = \mathbb{f}_{\mathtt{m}}$.

T-Seq TODO:

T-Primop TODO:

□

6.1.5 *Value Substitution and Optimization.*

Lemma 6.15 (Value Substitution Preserves Typing).

$$\Gamma \vdash \mathtt{e} : \sigma \;\wedge\; \Gamma \vdash \gamma \implies \vdash \gamma(\mathtt{e}) : \sigma.$$

Proof. By induction on derivation of $\Gamma \vdash \mathtt{e} : \sigma$.

T-Var By inversion of $\Gamma \vdash x : \sigma$, we know that $\Gamma(x) = \sigma$. Therefore, from $\Gamma \vdash \gamma$, we have

$$\frac{\mathbf{typeof}(\gamma(x)) = \sigma}{\Gamma \vdash \gamma(x) : \sigma}\;\text{T-Val}\;.$$

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

$$\gamma\text{-OK}\quad \frac{dom(\Gamma) = dom(\gamma) \qquad \forall x \in dom(\gamma).\ [\Gamma(x) = \sigma \iff \textbf{typeof}(\gamma(x)) = \sigma]}{\Gamma \vdash \gamma}$$

Fig. 24. Value substitution

T-Val Since $\gamma(\mathtt{v}) = \mathtt{v}$, $\Gamma \vdash \gamma(\mathtt{v}) : \sigma$ by T-Val.

T-MD Since $\gamma(\triangleleft\, \mathtt{m}(\overline{x :: \tau}) {=}\, \mathtt{e} \triangleright) = \triangleleft\, \mathtt{m}(\overline{x :: \tau}) {=}\, \gamma'(\mathtt{e}) \triangleright$ where $\gamma' = \gamma \setminus \overline{x}$, T-MD still holds.

T-Seq By inversion of $\Gamma \vdash (\mathtt{e}_1 \,;\, \mathtt{e}_2) : \sigma$, we know that $\Gamma \vdash \mathtt{e}_2 : \sigma$. By induction hypothesis on typing of $\mathtt{e}_2$, $\vdash \gamma(\mathtt{e}_2) : \sigma$. By definition, $\gamma(\mathtt{e}_1 \,;\, \mathtt{e}_2) = \gamma(\mathtt{e}_1) \,;\, \gamma(\mathtt{e}_2)$. Therefore:

$$\frac{\vdash \gamma(\mathtt{e}_2) : \sigma}{\vdash \gamma(\mathtt{e}_1) \,;\, \gamma(\mathtt{e}_2) : \sigma.}\ \text{T-Seq}$$

T-Primop Similarly to the previous case.

□

Lemma 6.16 (Value Substitution Preserves Optimization).

$$(\Gamma \vdash^{\Phi} (\!|\, \mathtt{e} \,|\!)_{\mathrm{M}} \rightsquigarrow (\!|\, \mathtt{e}' \,|\!)_{\mathrm{M}'} \wedge \Gamma \vdash \gamma) \quad\Longrightarrow\quad \vdash^{\Phi} (\!|\, \gamma(\mathtt{e}) \,|\!)_{\mathrm{M}} \rightsquigarrow (\!|\, \gamma(\mathtt{e}') \,|\!)_{\mathrm{M}'}.$$

Proof. By induction on derivation of $\Gamma \vdash^{\Phi} (\!|\, \mathtt{e} \,|\!)_{\mathrm{M}} \rightsquigarrow (\!|\, \mathtt{e}' \,|\!)_{\mathrm{M}'}$.

OE-Val By assumption, $\Gamma \vdash^{\Phi} (\!|\, \mathtt{v} \,|\!)_{\mathrm{M}} \rightsquigarrow (\!|\, \mathtt{v} \,|\!)_{\mathrm{M}'}$ with $\mathtt{v} \neq \mathtt{m}$, $\gamma(\mathtt{v}) = \mathtt{v}$ by definition, and by constructor, the optimization holds for any typing environment, including the empty one:

$$\frac{\mathtt{v} \neq \mathtt{m}}{\vdash^{\Phi} (\!|\, \mathtt{v} \,|\!)_{\mathrm{M}} \rightsquigarrow (\!|\, \mathtt{v} \,|\!)_{\mathrm{M}'}.}\ \text{OE-Val}$$

OE-Seq By assumption, $\Gamma \vdash^{\Phi} (\!|\, \mathtt{e}_1 \,;\, \mathtt{e}_2 \,|\!)_{\mathrm{M}} \rightsquigarrow (\!|\, \mathtt{e}'_1 \,;\, \mathtt{e}'_2 \,|\!)_{\mathrm{M}'}$. Then, by inversion, we also have judgments $\Gamma \vdash^{\Phi} (\!|\, \mathtt{e}_1 \,|\!)_{\mathrm{M}} \rightsquigarrow (\!|\, \mathtt{e}'_1 \,|\!)_{\mathrm{M}'}$ and $\Gamma \vdash^{\Phi} (\!|\, \mathtt{e}_2 \,|\!)_{\mathrm{M}} \rightsquigarrow (\!|\, \mathtt{e}'_2 \,|\!)_{\mathrm{M}'}$. By definition,

$$\gamma(\mathtt{e}_1 \,;\, \mathtt{e}_2) = \gamma(\mathtt{e}_1) \,;\, \gamma(\mathtt{e}_2) \ \text{ and } \ \gamma(\mathtt{e}'_1 \,;\, \mathtt{e}'_2) = \gamma(\mathtt{e}'_1) \,;\, \gamma(\mathtt{e}'_2).$$

By induction hypothesis on optimization of $\mathtt{e}_1$, we have

$$\vdash^{\Phi} (\!|\, \gamma(\mathtt{e}_1) \,|\!)_{\mathrm{M}} \rightsquigarrow (\!|\, \gamma(\mathtt{e}'_1) \,|\!)_{\mathrm{M}'},$$

and similarly for $\mathtt{e}_2$. Therefore:

$$\frac{\vdash^{\Phi} (\!|\, \gamma(\mathtt{e}_1) \,|\!)_{\mathrm{M}} \rightsquigarrow (\!|\, \gamma(\mathtt{e}'_1) \,|\!)_{\mathrm{M}'} \qquad \vdash^{\Phi} (\!|\, \gamma(\mathtt{e}_2) \,|\!)_{\mathrm{M}} \rightsquigarrow (\!|\, \gamma(\mathtt{e}'_2) \,|\!)_{\mathrm{M}'}}{\vdash^{\Phi} (\!|\, \gamma(\mathtt{e}_1) \,;\, \gamma(\mathtt{e}_2) \,|\!)_{\mathrm{M}} \rightsquigarrow (\!|\, \gamma(\mathtt{e}'_1) \,;\, \gamma(\mathtt{e}'_2) \,|\!)_{\mathrm{M}'}.}\ \text{OE-Seq}$$

OE-Inline In this case, $\mathtt{e} = \mathtt{m}(\overline{v})$, $\mathtt{e}' = (\mathtt{unit} \,;\, \mathtt{e}'')$, and the assumption is:

$$\frac{\begin{array}{c}\Gamma \vdash v_i : \sigma_i \\ \textbf{getmd}(\mathrm{M}, \mathtt{m}, \overline{\sigma}) = \triangleleft\, \mathtt{m}(\overline{x :: \tau}) {=}\, \mathtt{e}_b \triangleright \qquad \Gamma \vdash^{\Phi} (\!|\, \mathtt{e}_b[\overline{x \mapsto v}] \,|\!)_{\mathrm{M}} \rightsquigarrow (\!|\, \mathtt{e}'' \,|\!)_{\mathrm{M}'}\end{array}}{\Gamma \vdash^{\Phi} (\!|\, \mathtt{m}(\overline{v}) \,|\!)_{\mathrm{M}} \rightsquigarrow (\!|\, \mathtt{unit} \,;\, \mathtt{e}'' \,|\!)_{\mathrm{M}'}.}\ \text{OE-Inline}$$

By induction hypothesis on optimization of $\mathtt{e}_b[\overline{x \mapsto v}]$, we have

$$\vdash^{\Phi} (\!|\, \gamma(\mathtt{e}_b[\overline{x \mapsto v}]) \,|\!)_{\mathrm{M}} \rightsquigarrow (\!|\, \gamma(\mathtt{e}'') \,|\!)_{\mathrm{M}'}.$$

Because $\mathtt{e}_b$ should not have free variables bound in Γ and $dom(\Gamma) = dom(\gamma)$ (thanks to $\Gamma \vdash \gamma$), $\gamma(\mathtt{e}_b[\overline{x} \mapsto \overline{v}]) = \mathtt{e}_b[\overline{x} \mapsto \gamma(\overline{v})]$. Next, by definition, $\gamma(\mathtt{m}(\overline{v})) = \mathtt{m}(\gamma(\overline{v}))$ and $\gamma(\mathtt{unit}\,;\mathtt{e}'') = \mathtt{unit}\,;\gamma(\mathtt{e}'')$. Because $\Gamma \vdash \gamma$, by Lemma 6.15, we have $\vdash\ \gamma(v_i)\ :\ \sigma_i$. Thus:

$$\frac{\mathbf{getmd}(\mathrm{M}, \mathtt{m}, \overline{\sigma}) = \triangleleft\, \mathtt{m}(\overline{x :: \tau}) = \mathtt{e}_b \triangleright \qquad \begin{array}{c}\vdash\ \gamma(v_i)\ :\ \sigma_i \\ \vdash^{\Phi}\ (\!|\, \mathtt{e}_b[\overline{x} \mapsto \gamma(\overline{v})]\, |\!)_{\mathrm{M}} \rightsquigarrow (\!|\, \gamma(\mathtt{e}'')\, |\!)_{\mathrm{M}'}\end{array}}{\vdash^{\Phi}\ (\!|\, \mathtt{m}(\gamma(\overline{v}))\, |\!)_{\mathrm{M}} \rightsquigarrow (\!|\, \mathtt{unit}\,; \gamma(\mathtt{e}'')\, |\!)_{\mathrm{M}'}.}\ \textsc{OE-Inline}$$

OE-Specialize In this case, $\mathtt{e} = \mathtt{m}(\overline{\mathtt{e}})$, $\mathtt{e}' = \mathtt{m}'(\overline{\mathtt{e}}')$, and the assumption is:

$$\frac{\Gamma\ \vdash^{\Phi}\ (\!|\, \mathtt{e}_i\, |\!)_{\mathrm{M}} \rightsquigarrow (\!|\, \mathtt{e}_i'\, |\!)_{\mathrm{M}'} \qquad \Gamma\ \vdash\ \mathtt{e}_i'\ :\ \sigma_i \qquad (\mathtt{m}(\overline{\sigma}) \rightsquigarrow \mathtt{m}') \in \Phi}{\Gamma\ \vdash^{\Phi}\ (\!|\, \mathtt{m}(\overline{\mathtt{e}})\, |\!)_{\mathrm{M}} \rightsquigarrow (\!|\, \mathtt{m}'(\overline{\mathtt{e}}')\, |\!)_{\mathrm{M}'}}\ \textsc{OE-Specialize}$$

By induction hypothesis on optimization of $\mathtt{e}_i$,

$$\vdash^{\Phi}\ (\!|\, \gamma(\mathtt{e}_i)\, |\!)_{\mathrm{M}} \rightsquigarrow (\!|\, \gamma(\mathtt{e}_i')\, |\!)_{\mathrm{M}'}.$$

By Lemma 6.15,

$$\vdash\ \gamma(\mathtt{e}_i')\ :\ \sigma_i.$$

By definition, $\gamma(\mathtt{m}(\overline{\mathtt{e}})) = \mathtt{m}(\gamma(\overline{\mathtt{e}}))$ and $\gamma(\mathtt{m}'(\overline{\mathtt{e}}')) = \mathtt{m}'(\gamma(\overline{\mathtt{e}}'))$. Therefore, the desired optimization holds:

$$\frac{\vdash^{\Phi}\ (\!|\, \gamma(\mathtt{e}_i)\, |\!)_{\mathrm{M}} \rightsquigarrow (\!|\, \gamma(\mathtt{e}_i')\, |\!)_{\mathrm{M}'} \qquad \vdash\ \gamma(\mathtt{e}_i')\ :\ \sigma_i \qquad (\mathtt{m}(\overline{\sigma}) \rightsquigarrow \mathtt{m}') \in \Phi}{\vdash^{\Phi}\ (\!|\, \mathtt{m}(\gamma(\overline{\mathtt{e}}))\, |\!)_{\mathrm{M}} \rightsquigarrow (\!|\, \mathtt{m}'(\gamma(\overline{\mathtt{e}}'))\, |\!)_{\mathrm{M}'}}\ \textsc{OE-Specialize}$$

... Other cases go similarly to the considered ones.

□

Lemma 6.17 (Optimization Preserves Values).

$$\Gamma\ \vdash^{\Phi}\ (\!|\, \mathtt{v}\, |\!)_{\mathrm{M}} \rightsquigarrow (\!|\, \mathtt{e}'\, |\!)_{\mathrm{M}'} \quad \Longrightarrow \quad \mathtt{e}' = \mathtt{v}.$$

Proof. By case analysis on optimization relation. □

6.1.6 *Correctness of Optimization.*

Proof. We sketch a few cases here. Note that we use some facts implicitly, e.g. Lemma 6.13:

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

- Case OE-Val, $\Gamma\ \vdash^{\Phi}\ (\!|\, \mathtt{v}\, |\!)_{\mathrm{M}} \rightsquigarrow (\!|\, \mathtt{v}\, |\!)_{\mathrm{M}'}$, is trivial: $\gamma(\mathtt{e}_1) = \mathtt{v} = \gamma(\mathtt{e}_1')$, and there are no steps that $\mathtt{C}\left[(\!|\, \mathtt{v}\, |\!)_{\mathrm{M}}\right]$ can make to arrive into $\mathtt{C}\left[(\!|\, \_\, |\!)_{\mathrm{M}}\right]$ (because $\mathtt{v}$ is not $\mathtt{rdx}$). The only step possible for $\mathtt{C}\left[(\!|\, \mathtt{v}\, |\!)_{\mathrm{M}}\right]$ is
$$\langle \mathrm{M}_g, \mathtt{C}\left[(\!|\, \mathtt{v}\, |\!)_{\mathrm{M}}\right] \rangle \rightarrow \langle \mathrm{M}_g, \mathtt{C}\,[\mathtt{v}] \rangle,$$
and thus (1) and (2) are vacuously true.
- Case OE-Seq, where $\mathtt{e}_1 = (\mathtt{e}_{11}\,;\,\mathtt{e}_{12})$ and $\mathtt{e}_1' = (\mathtt{e}_{11}'\,;\,\mathtt{e}_{12}')$:
$$\frac{\Gamma\ \vdash^{\Phi}\ (\!|\, \mathtt{e}_{11}\, |\!)_{\mathrm{M}} \rightsquigarrow (\!|\, \mathtt{e}_{11}'\, |\!)_{\mathrm{M}'} \qquad \Gamma\ \vdash^{\Phi}\ (\!|\, \mathtt{e}_{12}\, |\!)_{\mathrm{M}} \rightsquigarrow (\!|\, \mathtt{e}_{12}'\, |\!)_{\mathrm{M}'}}{\Gamma\ \vdash^{\Phi}\ (\!|\, \mathtt{e}_{11}\,;\,\mathtt{e}_{12}\, |\!)_{\mathrm{M}} \rightsquigarrow (\!|\, \mathtt{e}_{11}'\,;\,\mathtt{e}_{12}'\, |\!)_{\mathrm{M}'}.}\ \textsc{OE-Seq}$$
By definition, $\gamma(\mathtt{e}_{11}\,;\,\mathtt{e}_{12}) = \gamma(\mathtt{e}_{11})\,;\,\gamma(\mathtt{e}_{12})$ and $\gamma(\mathtt{e}_{11}'\,;\,\mathtt{e}_{12}') = \gamma(\mathtt{e}_{11}')\,;\,\gamma(\mathtt{e}_{12}')$. We will denote the results as $\mathtt{e}_{\gamma 1}\,;\,\mathtt{e}_{\gamma 2}$ and $\mathtt{e}_{\gamma 1}'\,;\,\mathtt{e}_{\gamma 2}'$. By Lemma 6.16 (value substitution preserves typing):
$$\frac{\vdash^{\Phi}\ (\!|\, \mathtt{e}_{\gamma 1}\, |\!)_{\mathrm{M}} \rightsquigarrow (\!|\, \mathtt{e}_{\gamma 1}'\, |\!)_{\mathrm{M}'} \qquad \vdash^{\Phi}\ (\!|\, \mathtt{e}_{\gamma 2}\, |\!)_{\mathrm{M}} \rightsquigarrow (\!|\, \mathtt{e}_{\gamma 2}'\, |\!)_{\mathrm{M}'}}{\vdash^{\Phi}\ (\!|\, \mathtt{e}_{\gamma 1}\,;\,\mathtt{e}_{\gamma 2}\, |\!)_{\mathrm{M}} \rightsquigarrow (\!|\, \mathtt{e}_{\gamma 1}'\,;\,\mathtt{e}_{\gamma 2}'\, |\!)_{\mathrm{M}'}.}\ \textsc{OE-Seq}$$

Now, let us consider the forward direction (1) first. The source expression can make a step

$$\langle M_g, \mathsf{C}\left[\llparenthesis\, \mathsf{e}_{\gamma 1}\, ;\, \mathsf{e}_{\gamma 2}\, \rrparenthesis_{M}\right]\rangle \longrightarrow \langle M_g', \mathsf{C}\left[\llparenthesis\, \_\, \rrparenthesis_{M}\right]\rangle$$

only if $\mathsf{e}_{\gamma 1}\, ;\, \mathsf{e}_{\gamma 2} = \mathsf{C}'\,[\mathsf{rdx}]$. By analyzing $\boldsymbol{Can}\,(\mathsf{e}_{\gamma 1}\, ;\, \mathsf{e}_{\gamma 2}\ \mathbf{as}\ \mathcal{C}(\mathsf{C}',\ \mathsf{rdx}))$, we know that there are only 2 such possibilities:

(1) CN-Redex with $\mathsf{e}_{\gamma 1} = \mathsf{v}_1$ and $\boldsymbol{Can}\,(\mathsf{v}_1\, ;\, \mathsf{e}_{\gamma 2}\ \mathbf{as}\ \mathcal{C}(\square,\ \mathsf{rdx}))$. Therefore:

$$\langle M_g, \mathsf{C}\left[\llparenthesis\, \mathsf{v}_1\, ;\, \mathsf{e}_{\gamma 2}\, \rrparenthesis_{M}\right]\rangle \longrightarrow \langle M_g, \mathsf{C}\left[\llparenthesis\, \mathsf{e}_{\gamma 2}\, \rrparenthesis_{M}\right]\rangle.$$

Because $\vdash^{\Phi}\ \llparenthesis\, \mathsf{v}_1\, \rrparenthesis_{M} \rightsquigarrow \llparenthesis\, \mathsf{e}_{\gamma 1}'\, \rrparenthesis_{M'}$, by Lemma 6.17, $\mathsf{e}_{\gamma 1}' = \mathsf{v}_1$. Therefore, the optimized expression steps:

$$\langle M_g, \mathsf{C}\left[\llparenthesis\, \mathsf{v}_1\, ;\, \mathsf{e}_{\gamma 2}'\, \rrparenthesis_{M'}\right]\rangle \longrightarrow \langle M_g, \mathsf{C}\left[\llparenthesis\, \mathsf{e}_{\gamma 2}'\, \rrparenthesis_{M'}\right]\rangle.$$

And we already know that $\vdash^{\Phi}\ \llparenthesis\, \mathsf{e}_{\gamma 2}\, \rrparenthesis_{M} \rightsquigarrow \llparenthesis\, \mathsf{e}_{\gamma 2}'\, \rrparenthesis_{M'}$.

(2) CN-SeqC where:

$$\frac{\boldsymbol{Can}\,(\mathsf{e}_{\gamma 1}\ \mathbf{as}\ \mathcal{C}(\mathsf{C}_1,\ \mathsf{rdx}))}{\boldsymbol{Can}\,(\mathsf{e}_{\gamma 1}\, ;\, \mathsf{e}_{\gamma 2}\ \mathbf{as}\ \mathcal{C}(\mathsf{C}_1\, ;\, \mathsf{e}_{\gamma 2},\ \mathsf{rdx}))}\ \text{CN-SeqC}$$

Thus, we know that $\mathsf{C}\left[\llparenthesis\, \mathsf{e}_{\gamma 1}\, ;\, \mathsf{e}_{\gamma 2}\, \rrparenthesis_{M}\right] = \mathsf{C}\left[\llparenthesis\, \mathsf{C}_1\,[\mathsf{rdx}]\, ;\, \mathsf{e}_{\gamma 2}\, \rrparenthesis_{M}\right]$, which means

$$\begin{gathered}\langle M_g, \mathsf{C}\left[\llparenthesis\, \mathsf{C}_1\,[\mathsf{rdx}]\, ;\, \mathsf{e}_{\gamma 2}\, \rrparenthesis_{M}\right]\rangle \longrightarrow \langle M_g', \mathsf{C}\left[\llparenthesis\, \mathsf{C}_1\,[\mathsf{e}']\, ;\, \mathsf{e}_{\gamma 2}\, \rrparenthesis_{M}\right]\rangle \\ \Longleftrightarrow \\ \langle M_g, \mathsf{C}\left[\llparenthesis\, \mathsf{C}_1\,[\mathsf{rdx}]\, \rrparenthesis_{M}\right]\rangle \longrightarrow \langle M_g', \mathsf{C}\left[\llparenthesis\, \mathsf{C}_1\,[\mathsf{e}']\, \rrparenthesis_{M}\right]\rangle,\end{gathered}$$

which is the same as

$$\langle M_g, \mathsf{C}\left[\llparenthesis\, \mathsf{e}_{\gamma 1}\, \rrparenthesis_{M}\right]\rangle \longrightarrow \langle M_g', \mathsf{C}\left[\llparenthesis\, \mathsf{C}_1\,[\mathsf{e}']\, \rrparenthesis_{M}\right]\rangle.$$

(Note that $\mathsf{C}_1\,[\mathsf{e}']$ plays the role of e_2 from the theorem statement.) Next, recall the assumption $\Gamma\ \vdash^{\Phi}\ \llparenthesis\, \mathsf{e}_{11}\, \rrparenthesis_{M} \rightsquigarrow \llparenthesis\, \mathsf{e}_{11}'\, \rrparenthesis_{M'}$. By induction hypothesis, $\exists \mathsf{e}_{21}'$ such that

$$\langle M_g, \mathsf{C}\left[\llparenthesis\, \mathsf{e}_{\gamma 1}'\, \rrparenthesis_{M'}\right]\rangle \longrightarrow \langle M_g', \mathsf{C}\left[\llparenthesis\, \mathsf{e}_{21}'\, \rrparenthesis_{M'}\right]\rangle$$

and

$$\vdash^{\Phi}\ \llparenthesis\, \mathsf{C}_1\,[\mathsf{e}']\, \rrparenthesis_{M} \rightsquigarrow \llparenthesis\, \mathsf{e}_{21}'\, \rrparenthesis_{M'}.$$

The way $\mathsf{C}\left[\llparenthesis\, \mathsf{e}_{\gamma 1}'\, \rrparenthesis_{M'}\right]$ steps, means that $\mathsf{e}_{\gamma 1}' = \mathsf{C}_1'\,[\mathsf{rdx}']$. By context manipulations similar to the above, we get that

$$\langle M_g, \mathsf{C}\left[\llparenthesis\, \mathsf{e}_{\gamma 1}'\, ;\, \mathsf{e}_{\gamma 2}'\, \rrparenthesis_{M'}\right]\rangle \longrightarrow \langle M_g', \mathsf{C}\left[\llparenthesis\, \mathsf{e}_{21}'\, ;\, \mathsf{e}_{\gamma 2}'\, \rrparenthesis_{M'}\right]\rangle.$$

Finally, we can show that the optimization relation holds:

$$\frac{\vdash^{\Phi}\ \llparenthesis\, \mathsf{C}_1\,[\mathsf{e}']\, \rrparenthesis_{M} \rightsquigarrow \llparenthesis\, \mathsf{e}_{21}'\, \rrparenthesis_{M'} \qquad \vdash^{\Phi}\ \llparenthesis\, \mathsf{e}_{\gamma 2}\, \rrparenthesis_{M} \rightsquigarrow \llparenthesis\, \mathsf{e}_{\gamma 2}'\, \rrparenthesis_{M'}}{\vdash^{\Phi}\ \llparenthesis\, \mathsf{C}_1\,[\mathsf{e}']\, ;\, \mathsf{e}_{\gamma 2}\, \rrparenthesis_{M} \rightsquigarrow \llparenthesis\, \mathsf{e}_{21}'\, ;\, \mathsf{e}_{\gamma 2}'\, \rrparenthesis_{M'}.}\ \text{OE-Seq}$$

□

6.2 Litmus Tests

As a basic test of functionality, we provide 9 litmus tests shown in Fig. 25, 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

```julia
#fails, too new
function g()
  eval(:(k() = 2))
  k()
end
g() # error
```

(a)

```julia
function h()
  eval(:(j() = 2))
  Base.invokelatest(j)
end

h() == 2
```

(b)

```julia
function h()
  eval(:(p() = 2))
  eval(:(p()))
end

h() == 2
```

(c)

```julia
r2() = r1()
function i()
  eval(:(r1() = 2))
  r2()
end

i() # error
```

(d)

```julia
r4() = r3()
function m()
  eval(:(r3() = 2))
  Base.invokelatest(r4)
end

m() == 2
```

(e)

```julia
function l()
  eval(quote
    eval(:(f1() = 2))
    f1()
  end)
end
l() == 2
```

(f)

```julia
x = 1
f(x) = (
    eval(:(x = 0));
    x * 2)
f(42) == 84
x == 0
```

(g)

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

(h)

```julia
g() = 2
f(x) = (eval(:(g() = $x));
        x * eval(:(g())))
f(42) == 1764
```

(i)

Fig. 25. Litmus Tests

6.3 Optimization Algorithm

$$
\begin{array}{llll}
\Omega & ::= & & \textit{Inline environment} \\
 & | & \varnothing & \text{empty environment} \\
 & | & \Omega, \mathtt{m}(\overline{\tau}) \rightsquigarrow \mathbb{N} & \text{inline mapping extension} \\
\mathtt{I} & & & \text{max inline count} \\
\mathtt{S} & & & \text{max specialization count} \\
 & & & \\
\Phi^{\tau} & ::= & & \textit{Direct call environment} \\
 & | & \varnothing & \text{empty environment} \\
 & | & \Phi^{\tau}, \mathtt{m}(\overline{\tau}) \rightsquigarrow \mathtt{m}' & \text{specialization mapping extension} \\
 & & & \\
\mathtt{E} & ::= & & \textit{Optimization context} \\
 & | & \square & \text{hole} \\
 & | & \mathtt{E}\,;\,\mathtt{e} \mid \mathtt{e}\,;\,\mathtt{E} & \text{sequence} \\
 & | & \delta_l(\overline{\mathtt{e}}\;\mathtt{E}\;\overline{\mathtt{e}}') & \text{primop call (argument)} \\
 & | & \mathtt{E}(\overline{\mathtt{e}}) & \text{function call (callee)} \\
 & | & \mathtt{e}(\overline{\mathtt{e}}\;\mathtt{E}\;\overline{\mathtt{e}}') & \text{function call (argument)}
\end{array}
$$

OE-Inline

$$
\frac{\begin{array}{c}\Gamma \vdash v_i : \sigma_i \\ \textbf{getmd}(M, \mathtt{m}, \overline{\sigma}) = \triangleleft\, \mathtt{m}(\overline{x :: \tau}) = \mathtt{e}_b \triangleright \qquad \textbf{iCount}(\Omega,\ \mathtt{m}(\overline{\tau}))\ <\ \mathtt{I} \qquad \textbf{iCount}^{++}(\Omega,\ \mathtt{m}(\overline{\tau})) = \Omega'\end{array}}{\Gamma \vdash \langle \Omega,\ \Phi^{\tau},\ \Phi,\ \llparenthesis\, \mathtt{E}\,[\mathtt{m}(\overline{v})]\, \rrparenthesis_M \rangle \rightsquigarrow \langle \Omega',\ \Phi^{\tau},\ \Phi,\ \llparenthesis\, \mathtt{E}\,[\mathtt{unit}\,;\,\mathtt{e}_b[\overline{x} \mapsto \overline{v}]]\, \rrparenthesis_M \rangle}
$$

OE-Specialize-Existing

$$
\frac{\exists\, \mathtt{e}_i \in \overline{\mathtt{e}} : \textbf{typeof}(\mathtt{e}_i) \notin \sigma \qquad \nexists\, \overline{\sigma''}, \mathtt{m}'' : \mathtt{m}''(\overline{\sigma''}) \rightsquigarrow \mathtt{m} \in \Phi \qquad \Gamma \vdash \mathtt{e}_i : \sigma_i \qquad \mathtt{m}(\overline{\sigma}) \rightsquigarrow \mathtt{m}' \in \Phi}{\Gamma \vdash \langle \Omega,\ \Phi^{\tau},\ \Phi,\ \llparenthesis\, \mathtt{E}\,[\mathtt{m}(\overline{\mathtt{e}})]\, \rrparenthesis_M \rangle \rightsquigarrow \langle \Omega,\ \Phi^{\tau},\ \Phi,\ \llparenthesis\, \mathtt{E}\,[\mathtt{m}'(\overline{\mathtt{e}})]\, \rrparenthesis_M \rangle}
$$

OE-Specialize-New

$$
\frac{\begin{array}{c}\exists\, \mathtt{e}_i \in \overline{\mathtt{e}} : \textbf{typeof}(\mathtt{e}_i) \notin \sigma \qquad \nexists\, \overline{\sigma''}, \mathtt{m}'' : \mathtt{m}''(\overline{\sigma''}) \rightsquigarrow \mathtt{m} \in \Phi \qquad \Gamma \vdash \mathtt{e}_i : \sigma_i \\ \textbf{getmd}(M, \mathtt{m}, \overline{\sigma}) = \triangleleft\, \mathtt{m}(\overline{x :: \tau}) = \mathtt{e}_b \triangleright \qquad \nexists\, \mathtt{m}_{opt} : \mathtt{m}(\overline{\sigma}) \rightsquigarrow \mathtt{m}_{opt} \in \Phi \qquad \textbf{sCount}(\Phi,\ \mathtt{m})\ <\ \mathtt{S} \\ \mathtt{m}' \text{ does not occur in } M \qquad M' = \triangleleft\, \mathtt{m}'(\overline{x :: \sigma}) = \mathtt{e}_b \triangleright \bullet M \qquad \Phi' = \Phi, \mathtt{m}(\overline{\sigma}) \rightsquigarrow \mathtt{m}'\end{array}}{\Gamma \vdash \langle \Omega,\ \Phi^{\tau},\ \Phi,\ \llparenthesis\, \mathtt{E}\,[\mathtt{m}(\overline{\mathtt{e}})]\, \rrparenthesis_M \rangle \rightsquigarrow \langle \Omega,\ \Phi^{\tau},\ \Phi',\ \llparenthesis\, \mathtt{E}\,[\mathtt{m}'(\overline{\mathtt{e}})]\, \rrparenthesis_{M'} \rangle}
$$

OE-Direct-Call-Existing

$$
\frac{\begin{array}{c}\exists\, \mathtt{e}_i \in \overline{\mathtt{e}},\ \textbf{typeof}(\mathtt{e}_i) \notin \sigma \\ \nexists\, \overline{\sigma''}, \mathtt{m}'' : \mathtt{m}''(\overline{\sigma''}) \rightsquigarrow \mathtt{m} \in \Phi \qquad \Gamma \vdash \mathtt{e}_i : \sigma_i \qquad \textbf{getmd}(M, \mathtt{m}, \overline{\sigma}) = \triangleleft\, \mathtt{m}(\overline{x :: \tau}) = \mathtt{e}_b \triangleright \\ \nexists\, \mathtt{m}_{opt},\ \mathtt{m}(\overline{\sigma}) \rightsquigarrow \mathtt{m}_{opt} \in \Phi \qquad \textbf{sCount}(\Phi,\ \mathtt{m})\ \geq\ \mathtt{S} \qquad \mathtt{m}(\overline{\tau}) \rightsquigarrow \mathtt{m}' \in \Phi^{\tau} \qquad \Phi' = \Phi, \mathtt{m}(\overline{\sigma}) \rightsquigarrow \mathtt{m}'\end{array}}{\Gamma \vdash \langle \Omega,\ \Phi^{\tau},\ \Phi,\ \llparenthesis\, \mathtt{E}\,[\mathtt{m}(\overline{\mathtt{e}})]\, \rrparenthesis_M \rangle \rightsquigarrow \langle \Omega,\ \Phi^{\tau},\ \Phi',\ \llparenthesis\, \mathtt{E}\,[\mathtt{m}'(\overline{\mathtt{e}})]\, \rrparenthesis_M \rangle}
$$

OE-Direct-Call-New

$$
\frac{\begin{array}{c}\exists\, \mathtt{e}_i \in \overline{\mathtt{e}} : \textbf{typeof}(\mathtt{e}_i) \notin \sigma \\ \nexists\, \overline{\sigma''}, \mathtt{m}'' : \mathtt{m}''(\overline{\sigma''}) \rightsquigarrow \mathtt{m} \in \Phi \qquad \Gamma \vdash \mathtt{e}_i : \sigma_i \qquad \textbf{getmd}(M, \mathtt{m}, \overline{\sigma}) = \triangleleft\, \mathtt{m}(\overline{x :: \tau}) = \mathtt{e}_b \triangleright \\ \nexists\, \mathtt{m}_{opt} : \mathtt{m}(\overline{\sigma}) \rightsquigarrow \mathtt{m}_{opt} \in \Phi \qquad \textbf{sCount}(\Phi,\ \mathtt{m})\ \geq\ \mathtt{S} \qquad \mathtt{m}' \text{ does not occur in } M \\ M' = \triangleleft\, \mathtt{m}'(\overline{x :: \tau}) = \mathtt{e}_b \triangleright \bullet M \qquad \Phi' = \Phi, \mathtt{m}(\overline{\sigma}) \rightsquigarrow \mathtt{m}' \qquad \Phi^{\tau'} = \Phi^{\tau}, \mathtt{m}(\overline{\tau}) \rightsquigarrow \mathtt{m}'\end{array}}{\Gamma \vdash \langle \Omega,\ \Phi^{\tau},\ \Phi,\ \llparenthesis\, \mathtt{E}\,[\mathtt{m}(\overline{\mathtt{e}})]\, \rrparenthesis_M \rangle \rightsquigarrow \langle \Omega,\ \Phi^{\tau'},\ \Phi',\ \llparenthesis\, \mathtt{E}\,[\mathtt{m}'(\overline{\mathtt{e}})]\, \rrparenthesis_{M'} \rangle}
$$

Fig. 26. Expression optimization with evaluation context: transitive closure of $\rightsquigarrow$

$$\begin{array}{rcl}
\textbf{iCount}(\Omega, \mathtt{m}(\overline{\tau})) & = & n \\
 & & \text{if } \mathtt{m}(\overline{\tau}) \rightsquigarrow n \in \Omega \\
\textbf{iCount}(\Omega, \mathtt{m}(\overline{\tau})) & = & 0 \\
 & & \text{if } \nexists n : \mathtt{m}(\overline{\tau}) \rightsquigarrow n \in \Omega \\
\textbf{iCount}^{++}(\Omega, \mathtt{m}(\overline{\tau})) & = & (\varnothing, \mathtt{m}(\overline{\tau}) \rightsquigarrow n+1) \cup \Omega_{rest} \\
 & & \text{if } (\varnothing, \mathtt{m}(\overline{\tau}) \rightsquigarrow n) \cup \Omega_{rest} == \Omega \;\&\& \\
 & & (\varnothing, \mathtt{m}(\overline{\tau}) \rightsquigarrow n) \cap \Omega_{rest} == \varnothing \\
\textbf{iCount}^{++}(\Omega, \mathtt{m}(\overline{\tau})) & = & \Omega, \mathtt{m}(\overline{\tau}) \rightsquigarrow 1 \\
 & & \text{if } \nexists n : \mathtt{m}(\overline{\tau}) \rightsquigarrow n \in \Omega \\
\textbf{sCount}(\varnothing,\ \mathtt{m}) & = & 0 \\
\textbf{sCount}((\Phi_{rest}, \mathtt{m}(\overline{\sigma}) \rightsquigarrow \mathtt{m}'),\ \mathtt{m}_{cmp}) & = & \textbf{sCount}(\Phi_{rest},\ \mathtt{m}_{cmp}) \\
 & & \text{if } \mathtt{m}_{cmp} \neq \mathtt{m} \\
\textbf{sCount}((\Phi_{rest}, \mathtt{m}(\overline{\sigma}) \rightsquigarrow \mathtt{m}'),\ \mathtt{m}_{cmp}) & = & 1 + \textbf{sCount}(\Phi_{rest},\ \mathtt{m}_{cmp}) \\
 & & \text{if } \mathtt{m}_{cmp} == \mathtt{m}
\end{array}$$

Fig. 27. Expression optimization auxiliary helpers