TYPE STABILITY IN JULIA

A Simple and Efficient Optimization Technique

ARTEM PELENITSYN

A dissertation submitted in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Khoury College of Computer Sciences
Northeastern University
Boston, USA
2023

ABSTRACT

The design space for just-in-time (JIT) compilers is big, and Julia represents one viewpoint. The outstanding features of this viewpoint is simplicity and efficiency, which are enabled by a clever co-design of the language and its implementation. The combination of simplicity and efficiency also allows users to employ language strengths and avoid common pitfalls that threaten the wide family of JIT compilers.

My work has been focused on type stability in Julia—a program property enabling key optimizations in the compiler. Informally, a function is type stable if the type of the output depends only on the types of the inputs, not their values. In this dissertation, I make the following contributions related to type stability. First, an analysis of how widespread the property is in publicly available Julia code, and what features may be related to the property. Second, a formal model of a JIT compiler recognizing the property at run time and performing optimizations accordingly. Third, an automated approach to approximate type stability without running the program.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

This work would not be possible without... many people, actually! It is astonishing how many people were instrumental in getting where I am now. I cannot name everyone. But I will try my best.

My advisor Jan Vitek introduced me to the Julia programming language and, later on, type stability. For someone working only with statically typed languages since high school, this was a blast, but a manageable one (Jan has interests in languages way crazier than Julia). Jan also took me as a student with no clear idea about what to work on and with a notably distant interests from his own, which, in my view, is a great advance. He broadened my understanding of what programming languages research can be — probably the greatest scientific influence I ever had.

I came to Northeastern after spending 13 years in I.I. Vorovich Institute of Mathematics, Mechanics and Computer Science (MMCS), Southern Federal University (Rostov-on-Don, Russia) — first, as a student, and then, as a teaching faculty. Several people there left a lasting impact on my academic interests, but many more helped to create a sense of belonging and a safe space where I could pursue my teaching and research aspirations. I cannot express how much I enjoyed each day during those 13 years.

My BSc/MSc advisor, late Vladimir Deundyak, spent many hours with me talking about error-correcting codes (topic of my MSc thesis), mathematics (his true passion despite going undercover as a security researcher), academia, and politics. If I had to choose one quote from him that taught me the most, that would be: "A mathematician has very few joys in life, so we should run them to the ground". And by "mathematician" he really meant "scientist".

Stanislav Mikhalkovich has been working on what I consider the largest (at least, in terms of the user base) compiler project in Russia — PascalABC.NET. It is a compiler and an IDE for teaching programming in a .NET-enabled and radically modernized dialect of Pascal — a language historically dominating the Russian educational landscape. Besides providing deep expertise in compilers, Stanislav shared and encouraged curiosity about programming language theory, which led several of us to organize a reading group on John C. Mitchell's

Foundations for Programming Languages. Stanislav also accepted me as a teacher in his "Sunschool", a place where middle-school kids can learn about computers, digital creativity, and programming. A completely different perspective on computer science (and a critical source of income) for me over the years.

While at MMCS, I supervised a number of BSc and one MSc student. I still talk to some of them occasionally — they are all very special people to me not only because they learned something from me (I hope!), but also because I learned something from them in a unique way. Gosha Lukyanov, Volodya Pankov, Masha Vturina, Olya Filippskaya, Anya Bolotina, and others helped me feel better about what I am doing in academia. Gosha raised to earn a PhD in computer science from Newcastle U with Andrey Mokhov — his achievement I keep bragging about like I have anything to do with it! I am glad that Anya is on a similar track.

After I left MMCS in 2017, a ton of bright students and postdocs working with Jan helped merealize the wide field of programming languages. For creating a sense of a second professional family, I want to thank Filip K%r ikava, Peta Maj, Konrad Siek, Ryan Culpepper, Jan Jecmen. Jan joined me in building the prototype of the type stability inference tool and brought several deep insights (backed by pull requests) in how to improve it. The Czech lab and the year I spent in Prague are always in my heart.

PRL students during my tenure at NEU were all amazing, bright, and each taught me something. Starting from the old timers: Max, William, Ben G, Justin. And continuing with "my cohort": Ellen, Alexi, Aviral, Hyeyoung, Celeste, Aaron, Ben C, Ming-Ho.

Ming-Ho Yee helped to improve my writing and presentations tremendously. He also took the lead in organizing social events in the lab when we needed them most — when the pandemic hit. Ming-Ho taught me that sharing your joys as well as grievances with colleagues may be appropriate and helpful to both parties in the conversation.

Mitch Wand, Emeritus Professor at the lab, taught an amazing course on technical writing that was taken over by the PRL crowd and turned into pure joy for the whole semester. If I learned one thing from Mitch, that would be the advice about putting an example in the introduction section of the paper to motivate the problem: when the paper comes to an end, the example should be solved one way or another. I keep seeing Mitch's name on numerous old papers referenced byongoing research and cannot imagine what a great contribution he has made to our field.

My first lead co-author on a "big" paper (OOPSLA '18) Francesco Zappa Nardelli showed me an example of endless perseverance while digging deep into the dark corners of Julia and its subtyping relation. Later on, he agreed to provide his time and highest expertise by serving on my thesis committee.

I owe my thesis committee a big thank you: Frank, Arjun, Francesco, and Jan were helpful and generous in sharing their insights about my work.

Besides my parents and close relatives, the two most influential people in my life are my wife Julia Belyakova and my best friend Vitaly Bragilevsky. Perhaps surprisingly, both of them not only shared a great amount of love, support, and care, but also influenced my research and work in academia.

I learned about lambda calculus in my second year of college, but it would remain a funny notation from the dusty folio of Barendregt with no real value if not for Vitaly, who helped me discover Haskell and programming languages theory. That was many years after he had helped me get geometry in 9th grade in my high school ("Liceum"), and some time before he invited me to teach at MMCS. We share so many memories that my only worry is that, one day, I start forgetting some things. "Memory Almost Full". But there is still space to add more.

I came to take for granted that Julia always sees the better in me. But she also actively helps me grow professionally and personally. I doubt I will ever be able to pay back for everything she gave me.

My mom and dad did a lot for me, but I want to specifically mention that I got the best education I can imagine given the circumstances because of them. Again, the debt is simply unimaginable.

I think my great grandma Nina who turned 98 three weeks before my defense beats everyone in the world in how much effort she has put into me. My single wish is to meet her in person again.

I am writing this while watching my soon-to-be 1-year-old daughter Sophia trying very hard to start walking. These past weeks she taught me about determination more than I ever knew. She is already a great teacher who I will keep learning from.

1 INTRODUCTION

Performance is serious business for a scientific programming language. Success in that niche hinges on the availability of a rich ecosystem of high-performance mathematical and algorithm libraries. Julia is a relative newcomer in this space. Its approach to scientific computing is predicated on the bet that users can write efficient numerical code in Julia without needing to resort to C or Fortran. The design of the language is a balancing act between productivity features, such as multiple dispatch and garbage collection, and performance features, such as limited inheritance and restricted-by-default dynamic code loading. Julia has been designed to ensure that a relatively straightforward path exists from source code to machine code, and its performance results are encouraging. They show, on some simple benchmarks, speeds in between those of C and Java. In other words, a new dynamic language written by a small team of language engineers can compete with mature, statically typed languages and their highly tuned compilers.

Writing efficient Julia code is best viewed as a dialogue between the programmer and the compiler. From its earliest days, Julia exposed the compiler's intermediate representation to users, encouraging them to (1) observe if and how the compiler is able to optimize their code, and (2) adapt their coding style to warrant optimizations. This came with a simple execution model: each time a function is called with a different tuple of concrete argument types, a new *specialization* is generated by Julia's just-in-time compiler. That specialization leverages the run-time type information about the arguments, to apply *unboxing* and *devirtualization* transformations to the code. The former lets the compiler manipulate values without indirection and stack allocate them; the latter sidesteps the powerful but costly multiple-dispatch mechanism and enables inlining of callees.

One key to performance in Julia stems from the compiler's success in determining the *concrete* return type of any function call it encounters. The intuition is that in such cases, the compiler is able to propagate precise type information as it traverses the code, which, in turn, is crucial for unboxing and devirtualization. More precisely, Julia makes an important distinction between concrete and abstract types:

a concrete type such as `Int64`, is final in the sense that the compiler knows the size and exact layout of values of that type; for values of abstract types such as `Number`, the compiler has no information. The property I alluded to is called *type stability*. Its informal definition states that a method is type stable if the concrete type of its output is entirely determined by the concrete types of its arguments.[1] Folklore suggests that one should strive to write type-stable methods outright, or, if performance is an issue, refactor methods so that they become type stable.

My goal in this dissertation is three-fold. First, I show that type stability is exhibited widely in practical Julia packages. Second, I formalize the relationship between type-stable code and the ability of a just-in-time (JIT) compiler to perform type-based optimizations during program execution. Finally, I devise a procedure to statically approximate type stability, wich allows Julia programmers to check stability without running their code or supplying sample inputs.

1.1 THESIS STATEMENT

Optimizing dynamic languages is difficult and remains an active research field. Several state-of-the-art JIT compilers perform the job well at the expense of being opaque and unpredictable for the user. Julia's unique design seemingly proposes a better approach based on the notion of type stability. My thesis is, therefore:

> *Type stability is a widely used program property that can be leveraged by a compiler to generate correct and efficient code and can be approximated by automated techniques.*

To validate this thesis, I make the following contributions:

1. Assess how widely type-stable code is deployed inside Julia's ecosystem, and whether any code patterns are associated with type-stable code.
2. Establish a formal correspondence between type stability and code optimizations and show that optimized code is semantically equivalent to the initial version.
3. Design an approach for approximating type stability through a static analysis and demonstrate the practicality of the approach by building a tool for source code analysis.

1 https://docs.julialang.org/en/v1/manual/faq/#man-type-stability

This work builds on a number of papers that I have (co-)written, in particular:

- *Julia Subtyping: A Rational Reconstruction* (OOPSLA 2018)
  By Francesco Zappa Nardelli, Julia Belyakova, **Artem Pelenitsyn**, Benjamin Chung, Jeff Bezanson, and Jan Vitek [Zappa Nardelli et al. 2018]

- *Type Stability in Julia: Avoiding Performance Pathologies in JIT Compilation* (OOPSLA 2021)
  By **Artem Pelenitsyn**, Julia Belyakova, Benjamin Chung, Ross Tate, and Jan Vitek [Pelenitsyn et al. 2021]

- *Approximating Type Stability in The Julia JIT (work-in-progress)* (VMIL 2023)
  By **Artem Pelenitsyn** [Pelenitsyn 2023]

2 BACKGROUND AND MOTIVATION

2.1 TYPEFUL JULIA

The Julia language is designed around multiple dispatch [Bezanson et al. 2017]. Programs consist of *functions* that are implemented by multiple *methods* of the same name; each method is distinguished by a distinct type signature, and all methods are stored in a so-called method table. At run time, the Julia implementation dispatches a function call to the *most specific* method by comparing the types of the arguments to the types of the parameters of all methods of that function. As an example of a function and its constituent methods, consider the + function. As of version 1.8.5 of the language, there are 206 implementations of +, each covering a specific case determined by its type signature. Fig. 2.1 displays custom implementations for 16-bit floating point numbers, missing values, big-floats/big-integers, and complex arithmetic. Although at the source-code level, multiple methods look similar to overloading known from languages like C++ and Java, the key difference is that those languages resolve overloading statically whereas Julia does that dynamically using multiple dispatch.

The expressive power of multiple dispatch stems from the way it constrains the applicability of a method to a particular set of values. With it, programmers can write code that is concise and clear, as special cases can be relegated to dedicated methods. To pick the

```
# 206 methods for generic function "+":
[1] +(a::Float16, b::Float16) in Base at float.jl:398
[2] +(::Missing, ::Missing) in Base at missing.jl:114
[3] +(::Missing) in Base at missing.jl:100
[4] +(::Missing, ::Number) in Base at missing.jl:115
[5] +(a::BigFloat, b::BigFloat, c::BigFloat, d::BigFloat) in Base
    .MPFR at mpfr.jl:541
[6] +(a::BigFloat, b::BigFloat, c::BigFloat) in Base.MPFR at mpfr
    .jl:535
[7] +(x::BigFloat, c::BigInt) in Base.MPFR at mpfr.jl:394
[8] +(x::BigFloat, y::BigFloat) in Base.MPFR at mpfr.jl:363
...
```

Figure 2.1: Methods from the standard library

most specific applicable method, Julia's runtime relies on a subtype relation, denoted with `<:`, between run-time argument types and method signatures. Method signatures are defined with a rich type language that supports user-defined nominal types, built-in tuples and unions, and bounded existential types. The type language is discussed below, and a detailed discussion of subtyping can be found in [Zappa Nardelli et al. 2018].

TOP AND BOTTOM. The abstract type `Any` is the type of all values and is the default when type annotations are omitted. The empty union `Union{}` is a subtype of all types; it is not inhabited by any value. Unlike many common languages, Julia does not have a null value or a null type that is a subtype of all types.

DATATYPES. Datatypes can be *abstract* or *concrete*. Abstract datatypes may have subtypes but cannot have fields. Concrete datatypes have fields but cannot have declared subtypes. Every value is an instance of a concrete `DataType` that has a size, storage layout, supertype (`Any` if not otherwise declared), and, optionally, field names. Consider the following definitions:

```
abstract type Integer <: Real
end

primitive type Bool <: Integer 8
end

mutable struct PointRB <: Any
  x::Real
  y::Bool
end
```

The first declaration introduces `Integer` as a subtype of `Real`. The type is abstract; as such it cannot be instantiated. The second declaration introduces a concrete, primitive, type for boolean values and specifies that its size is 8 bits; this type cannot be further subtyped. The last declaration introduces a concrete, mutable structure `PointRB` with two fields, `x` of abstract type `Real` and `y` of concrete type `Bool`. Abstract types are always stored as references, while concrete types are unboxed.

PARAMETRIC DATATYPES. The following defines an immtuable, parametrized, concrete type.

```
struct Rational{T<:Integer} <: Real
```

```
    num :: T
    den :: T
end
```

`Rational`, with no argument, is a valid type, containing all instances `Rational{Int}`, `Rational{UInt}`, `Rational{Int8}`, etc. Thus, the following holds:

`Rational{Int} <: Rational`.

Type parameters are *invariant*, thus the following does not hold even though `Int <: Integer`:

`Rational{Int} <: Rational{Integer}`.

This restriction stems from practical considerations: the memory layout of abstract types (`Integer`) and concrete types (`Int`) is different and can impact the representation of the parametric type. In a type declaration, parameters can be used to instantiate the supertype. This allows the declaration of a `Diagonal` as an `AbstractMatrix` of values of type `T`:

```
struct Diagonal{T,V<:AbstractVector{T}} <: AbstractMatrix{T}
  diag::V
  ...
end
```

Julia allows instantiating type variables of parametric types with primitive values. For example, `Array{Int,1}` denotes a type for arrays with one dimension and integer elements. There is no checking that parameters instantiated in a sensible way except when constructing values: defining a method with a parameter of type `Array{1,Int}` (note the reversed order of the arguments) is allowed, although such method will never be called because there are no values of this type, while trying to create a value of this type with the standard notation: `Array{1,Int}([1,2,3])` will fail with a message that no constructors for such type are defined.

TUPLE TYPES. Tuples are an abstraction of the arguments of a function; a tuple type is a parametrized immutable type where each parameter is the type of one field. Tuple types may have any number of parameters, and they are *covariant* in their parameters: `Tuple{Int}` is a subtype of `Tuple{Any}`. `Tuple{Any}` is considered an abstract type; tuple types are only concrete if their parameters are.

UNION TYPES. A union is an abstract type which includes, as values, all instances of any of its argument types. Thus, the type `Union{Integer,AbstractString}` denotes any values from the set of `Integer` and `AbstractString` values.

EXISTENTIAL TYPES. A parametric type without arguments like `Rational` acts as a supertype of all its instances (`Rational{Int}` etc.) because it is a different kind of type called a *UnionAll* type. Julia documentation describes UnionAll types as "the iterated union of types for all values of some parameter"; a more accurate way to write such type is `Rational{T} where Union{} <: T <: Any`, meaning all values whose type is `Rational{T}` for some value of `T`. UnionAll types correspond to bounded existential types in the literature, and a more usual notation for the type above would be $\exists$ `T.Rational{T}`. Julia does not have explicit *pack/unpack* operations; UnionAll types are abstract. Each `where` introduces a single type variable. The combination of parametric and existential types is expressive: the type of 1-dimensional arrays can be simply specified by `Array{T,1} where T`. Type variable bounds can refer to outer type variables. For example,

```
Tuple{T,Array{S}} where S<:AbstractArray{T} where T<:Real
```

refers to 2-tuples whose first element is some `Real`, and whose second element is an `Array` of any kind of array whose element type contains the type of the first tuple element. The `where` keyword itself can be nested. Consider the types `Array{Array{T,1} where T,1}` and `Array{Array{T,1},1} where T`. The former defines a 1-dimensional array of 1-dimensional arrays; each of the inner arrays consists of objects of the same type, but this type may vary from one inner array to the next. The latter type instead defines a 1-dimensional array of 1-dimensional arrays all of whose inner arrays must have the same type. Existential types can be explicitly instantiated with the type application syntax $(t$ `where T`$)\{t'\}$; partial instantiation is supported, and, for instance, `Array{Int}` denotes arrays of integers of arbitrary dimension.

MULTIPLE DISPATCH. Any function call in a program, such as `x+y`, requires choosing one of the methods of the target function. *Method dispatch* chooses the method using a multi-step process. First, the implementation obtains the concrete types of arguments. Second, it retrieves applicable methods by checking for subtyping between argument types and type annotations of the methods. Next, it sorts

these methods into subtype order. Finally, the call is dispatched to the most specific method—a method such that no other applicable method is its strict subtype. If no such method exists, an error is produced. As an example, consider the above definition of `+`: a call with two `BigFloat`'s dispatches to definition 8 from Fig. 2.1:

```
[8] +(x::BigFloat, y::BigFloat)
```

Function calls are pervasive in Julia, and their efficiency is crucial for performance. However, the many complex type-level operations involved in dispatch make the process slow. Moreover, the language implementation, as of this writing, does not perform inline caching [Deutsch and Schiffman 1984], meaning that dispatch results are not cached across calls. To attain acceptable performance, the compiler attempts to remove as many dispatch operations as it can. This optimization leverages run-time type information whenever a method is compiled, i.e., when it is called for the first time with a novel set of argument types. These types are used by the compiler to infer types in the method body. Then, this type information frequently allows the compiler to devirtualize and inline the function calls within a method [Aigner and Hölzle 1996], thus improving performance. However, this optimization is not always possible: if type inference cannot produce a sufficiently specific type, then the call cannot be devirtualized. Consider the prior example of `x+y`: the method to call cannot be determined if `y` is known to be one of `BigFloat` or `BigInt`. This problem arises for various reasons, for example, accessing a struct field of an abstract type, or the type inferencer losing precision due to a branching statement. A more detailed description of the compilation strategy and its performance is given in [Bezanson et al. 2018].

2.2 TYPE STABILITY: A KEY TO PERFORMANCE?

Removing dispatch is the key to performance, but to perform the optimization, the compiler needs precise type information. Thus, while developers are encouraged to write generic code, the code also needs to be conducive to type inference and type-based optimizations. In this section, I give an overview of the appropriate coding discipline, and explain how it enables optimizations.

2.2.1 Julia Performance

To illustrate performance implications of careless coding practices, consider Fig. 2.2, which displays a method for one of the Julia microbenchmarks, `pisum`. For the purposes of this example, we have added an identity function `id` which was initially implemented to return its argument in both branches, as well-behaved identities do. Then, the `id` method was changed to return a string in the impossible branch (`rand()` returns a value from 0 to 1). The impact of that change was about a 40% increase in the execution time of the benchmark (Julia 1.5.4).

```
function id(x)
  (rand() == 4.2) ? "x" : x
end


function pisum()
  sum = 0.0
  for j = 1:500
    sum = 0.0
    for k = 1:10000
      sum += id(1/(k*id(k)))
    end
  end
  sum
end
```

(a) Microbenchmark, redacted

```
julia> @code_warntype id(5)
Variables
  #self#b@::Core.Compiler.
       Const(id, false)b@
  x::Int64
Body::Union{Int64, String}
1 - %1 = Main.rand()::Float64
|   %2 = (%1 == 4.2)::Bool
+--       goto #3 if not %2
2 -       return "x"
3 -       return x

julia> @code_warntype pisum()
...
|   %20 = k::Int64
|   %21 = Main.id(k)::Union{
     Int64, String}
```

(b) Julia session

Figure 2.2: A Julia microbenchmark (a) illustrating performance implications of careless coding practices: changing `id` function to return values of different types leads to longer execution because of the `Union` type of `id(..)`, which propagates to `pisum` (b).

When a performance regression occurs, it is common for developers to study the intermediate representation produced by the compiler. To facilitate this, the language provides a macro, `code_warntype`, that shows the code along with the inferred types for a given function invocation. Fig. 2.2 demonstrates the result of calling `code_warntype` on `id(5)`. Types that are imprecise, i.e., not concrete, show up in red: they indicate that concrete type of a value may vary from run to run. Here, we see that when called with an integer, `id` may return either an `Int64` or a `String`. Moreover, the imprecise return type of `id` propagates to the caller, as can be seen by inspecting `pisum` with `code_warntype`. Such type imprecision can impact performance in two ways. First, the `sum` variable has to be boxed, adding a level of indirection to any operation

performed therein. Second, it is harder for the compiler to devirtualize and inline consecutive calls, thus requiring dynamic dispatch.

2.2.2 Type Stability

Julia's compilation model is designed to accommodate source programs with flexible types, yet to make such programs efficient. The compiler, by default, creates an *instance* of each source method for each distinct tuple of argument types. Thus, even if the programmer does not provide any type annotations, like in the `id` example, the compiler will create method instances for *concrete* input types seen during an execution. For example, since in `pisum`, function `id` is called both with a `Float64` and an `Int64`, the method table will hold two method instances in addition to the original, user-defined method. Because method instances have more precise argument types, the compiler can leverage them to produce more efficient code and infer more precise return types.

In Julia parlance, a method is called *type stable* if its inferred return type depends solely on the types of its arguments. In the example, `id` is not type stable, as its return type may change depending on the input *value* (in principle). On the contrary, the traditional implementation of the `id` function is type stable: its return type is always the same as the type of its sole input and does not depend on the input value, so, given the input type, the return type is deducible. The definition of type stability, though, is somewhat slippery. The community has multiple, subtly different, informal definitions that capture the same broad idea, but describe varying properties. The canonical definition from the Julia documentation describes type stability as

> *"[...] the type of the output is predictable from the types of the inputs. In particular, it means that the type of the output cannot vary depending on the values of the inputs."*

However, elsewhere, the documentation also states that *"An analogous type-stability problem exists for variables used repeatedly within a function:"*

```
function foo()
    x = 1
    for i = 1:10
        x /= rand()
    end
    x
end
```

This function will always return a `Float64`, which is the type of `x` at the end of the `foo` definition, regardless of the (nonexistent) inputs. However, the manual says that it is a type stability issue nonetheless. This is because the variable `x` is initialized to an `Int64` but then assigned a `Float64` in the loop. Some versions of the compiler boxed `x` as it could hold two different types; of course, in this example, one level of loop unrolling would alleviate the performance issue, but in general, imprecise types limit compiler optimizations. Conveniently, the `code_warntype` macro mentioned above will highlight imprecise types for *all* intermediate variables. Furthermore, the documentation states that

> *"[t]his serves as a warning of potential type instability."*

Effectively, there are two competing, type-related properties of function bodies. To address this confusion, I define and refer to them using two distinct terms:

- *type stability* is when a function's return type depends only on its argument types, and
- *type groundedness* is when every variable's type depends only on the argument types.

Although type stability is strictly weaker than type groundedness, we are interested in both properties. The latter, type groundedness, is useful for performance of the function itself, as it implies that unboxing, devirtualization, and inlining can occur. The former, type stability, allows the function to be used efficiently by other functions: namely, type-grounded functions may call a function that is only type stable but not grounded. For brevity, when the context is clear, I will refer to type stability and type groundedness as stability and groundedness in what follows.

2.2.3 Flavors of Type Stability

Type stability is an inter-procedural property, and in the worst case, it can depend on the whole program. Consider the functions of Fig. 2.3. Function `f0` is trivially type unstable, regardless of the type of its input: if `good(x)` returns true, `f0` returns an `Int64` value, otherwise `f0` returns a `String`. Function `f1` is trivially type stable as all control-flow paths return a constant of `Int64` type. Function `f2` is type stable as long as the negation operator is type stable and returns

```
function f0(x)
  if (good(x))
    0
  else
    "not 0"
  end
end

function f1(x)
  if (good(x))
    0
  else
    1
  end
end
```

```
function f2(x)
  if (good(x))
    x
  else
    -x
  end
end

function -(x::Float64)
  if (x==0)
    0
  else
    Base.neg_float(x)
  end
end
```

Figure 2.3: Flavors of stability: `f0` is unstable, `f1` is stable; `f2` is stable if `(-)` is stable; `(-)` is unstable.

a value of the same type as its argument. In the example, method `-(::Float64)` of the negation operator causes `f2(::Float64)` to lose type stability. This is a common bug where the function `(-)` either returns a `Float64` or `Int64` due to the constant `0` being of type `Int64`. The proper, Julia-style implementation for this method is to replace the constant `0` with `zero(x)`, which returns the zero value for the type of `x`, in this case `0.0`. The example of function `f2` illustrates the fact that stability is a whole-program property. Adding a method may cause some, seemingly unrelated, method to lose type stability.

2.2.4 Type Stability Versus Type Groundedness

Type stability of a method is important for the groundedness of its callers. Consider the function

```
h(x :: Int64) = g(x) + 1
```

If

```
g(x) = x
```

it follows that `h` is both stable and grounded, as `g` will always return an `Int64`, and so will `h`. However, if we define

```
g(x) = (x == 2) ? "Ho" : 4
```

then `h` suddenly loses both properties. To recover stability and groundedness of `h`, it is necessary to make `g` type stable, yet it does not have to be grounded. For example, despite the presence of the imprecise variable `y`, the following definition makes `h` grounded:

```
g(x) =
  let
    y = (x==2 ? "Ho" : 4)
  in
    x
  end
```

In practice, type stability is sought after when making architectural decisions. Idiomatic Julia functions are small and have no internal type-directed branching; instead, branches on types are replaced with multiple dispatch. Once type ambiguity is lifted into dispatch, small functions with specific type arguments are relatively easy to make type stable. In turn, this architecture allows for effective devirtualization in a caller, as in many cases, the inferred type at a call site will determine its callee at compilation time.

Thus, writing type-stable functions is a good practice, for it provides callers of those functions with an opportunity to be efficient. However, stability of callees is not a sufficient condition for the efficiency of their callers: the callers themselves need to strive for type groundedness, which requires eliminating type imprecision from control flow.

2.2.5 Patterns of Instability

There are several code patterns that are inherently type unstable. For one, accessing abstract fields of a structure is an unstable operation: the concrete type of the field value depends on the struct value, not just struct type. In Julia, it is recommended to avoid abstractly typed fields if performance is important, but they are a useful tool for interacting with external data sources and representing heterogenous data.

Another example is sum types (algebraic data types or open unions), which can be modeled with subtyping in Julia. Take a hierarchy of an abstract type `Expr` and its two concrete subtypes, `Lit` and `BinOp`. Such a hierarchy is convenient, because it allows for an `Expr` evaluator written with multiple dispatch:

```
run(e :: Lit) = ...
run(e :: BinOp) = ...
```

If we want the evaluator to be called with the result of a function `parse(s ::String)`, the latter cannot be type stable: `parse` will return values of different concrete types, `Lit` and `BinOp`, depending on the input string. If one does want `parse` to be stable, they need to always

return the same concrete type, e.g. an S-expression-style struct. Then, `run` has to be written without multiple dispatch, as a big if-expression, which may be undesirable.

3 TRACING TYPE STABILITY: AN EMPIRICAL STUDY

Anecdotal evidence suggests that type stability is discussed in the Julia community, but does Julia code exhibit the properties of stability and groundedness in practice? And if so, are there any indicators correlated with instability and ungroundedness? To find out, I ran a dynamic analysis on a corpus of Julia packages. All the packages come from the main language registry and are readily available via Julia's package manager; registered packages have to pass basic sanity checks and usually have a test suite.

The main questions of this empirical study are:

1. How uniformly are type stability and groundedness spread over Julia packages? How much of a difference can users expect from different packages?
2. Are package developers aware of type stability?
3. Are there any predictors of stability and groundedness in the code and do they influence how type-stable code looks?

3.1 METHODOLOGY

I take as a main corpus the 1000 most starred (using GitHub stars) packages from the Julia package registry; as of the beginning of 2021, the registry contained about 5.5K packages. The main corpus is used for an automated, high-level, aggregate analysis. I also take the 10 most starred packages from the corpus to perform finer-grained analysis and manual inspection. Out of the 1000 packages in the corpus, tests suits of only 760 succeeded on Julia 1.5.4, so these 760 comprise the final corpus. The reasons of failures are diverse, spanning from missing dependencies to the absence of tests, to timeout.

For every package of interest, the dynamic analysis runs the package test suite, analyzes compiled method instances, and records information relevant to type stability and groundedness. Namely, once a test suite runs to completion, we query Julia's virtual machine for the

	Stable	Grounded
Mean	74%	57%
Median	80%	57%
Std. Dev.	22%	24%

Table 3.1: Aggregate statistics for stability and groundedness

current set of available method instances, which represent instances compiled during the tests' execution. To avoid bias towards the standard library, instances of methods defined in standard modules, such as `Base`, `Core`, etc., are removed, which typically leaves us with several hundreds to several thousands of instances. For these remaining instances, type stability and groundedness are analyzed. As type information is not directly available for compiled, optimized code, I retrieve the original method of an instance and run Julia's type inferencer to obtain each register's type. In rare cases, type inference may fail; on the corpus, this almost never happened, with at most 5 failures per package. With the inference results at hand, we check the concreteness of the register typing and record a yes/no answer for both stability and groundedness. In addition to that, several metrics are recorded for each method: method size, the number of gotos and returns in the body, whether the method has varargs or `@nospecialize` arguments[1], and how polymorphic the method is, i.e. how many instances were compiled for it. This information is then used to identify possible correlations between the metrics and stability/groundedness.

To get a better understanding of type stability and performance, I employ several additional tools to analyze the 10 packages. For example, I look at their documentation, especially at the stated goals and domain of a package, and check the Git history to see if and how type stability is mentioned in the commits.

3.2 PACKAGE-LEVEL ANALYSIS

The aggregate results of the dynamic analysis for the 760 packages are shown in Table 3.1: 74% of method instances in a package are stable and 57% grounded, on average; median values are close to the means. The standard deviation is noticeable, so even on small samples

1 `@nospecialize` tells the compiler to *not* specialize for that argument and leave it abstract.

of packages, we expect to see packages with large deflections from the means.

A more detailed analysis of the 10 most starred packages, in alphabetical order, is shown in Table 3.2. A majority of these packages have stability numbers very close to the averages shown above, with the exception of Knet, which has only 16% of stable and 8% of grounded instances.

Package	Methods	Instances	Varargs	Stable	Grounded
DifferentialEquations	1355	7381	3%	70%	44%
Flux	381	4288	13%	76%	70%
Gadfly	1100	4717	10%	81%	58%
Gen	973	2605	2%	64%	43%
Genie	532	1401	12%	93%	78%
IJulia	39	136	8%	84%	60%
JuMP	2377	36406	7%	83%	63%
Knet	594	9013	7%	16%	8%
Plots	1167	5377	8%	74%	58%
Pluto	727	2337	4%	80%	66%

Table 3.2: Stability, groundedness, and polymorphism in 10 popular Julia packages

The Knet package is a type stability outlier. A quick search over project's documentation and history shows that the only kind of stability ever mentioned is numerical stability; furthermore, the word "performance" is mostly used to reference the performance of neural networks or CUDA-related utilities. Indeed, the package primarily serves as a communication layer for a GPU; most computations are done by calling the CUDA API for the purpose of building deep neural networks. Thus, in this specific domain, type stability of Julia code appears to be irrelevant.

On the other side of the stability spectrum is the 93% stable (78% grounded) Genie package, which provides a web application framework. Inspecting the package, we can confirm that its developers were aware of type stability and intentional about performance. For example, Genie's Git history contains several commits mentioning (improved) "type stability." The project README file states that the authors build upon

> *"Julia's strengths (high-level, high-performance, dynamic, JIT compiled)."*

Furthermore, the tutorial claims:

> *"Genie's goals: unparalleled developer productivity, excellent runtime performance."*

TYPE STABILITY (NON-)CORRELATES One parameter that I conjectured may correlate with stability is the average number of method instances per method (Inst/Meth column of Table 3.2), as it expresses the amount of polymorphism discovered in a package. Most of the packages compile just 2–4 instances per method on average, but Flux, JuMP, and Knet have this metric 5–6 times higher, with JuMP and Knet exploiting polymorphism the most, at 15.3 and 15.2 instances per method, respectively. The latter may be related to the very low type stability index of Knet. However, the other two packages are more stable than the overall average. Analyzing JuMP and Flux further, we order their methods by the number of instances. In JuMP, the top 10% of most instantiated methods are 5% less stable and grounded than the package average, whereas in Flux, the top 10% have about the same stability characteristics as on average. Overall, I cannot conclude that method polymorphism is related to type stability.

Another dimension of polymorphism is the variable number of arguments in a method (Varargs column of Table 3.2). I looked into three packages with a higher than average (9%) number of varargs methods in the 10 packages: Flux, Gadfly and Genie. Relative to the total number of methods, Flux has the most varargs methods—13%—and those methods are only 55% stable and 44% grounded, which is a significant drop of 21% and 26% below this package's averages. However, the other two packages have higher-than-package-average stability rates, 82% (Gadfly) and 99% (Genie), with groundedness being high in Genie, 93%, and low in Gadfly, 38%. Again, no general conclusion about the relationship between varargs methods and their stability can be made.

3.3 METHOD-LEVEL ANALYSIS

In this section, I inspect stability of individual methods in its possible relationship with other code properties like size, control flow (number of goto and return statements), and polymorphism (number of compiled instances and varargs). This analysis consists of two steps: first, I plot histograms showing the number of methods with particular values of properties, and second, I manually sample some of the methods with more extreme characteristics.

3.3.1 Graphical Analysis

I use two-dimensional histograms like those presented in Fig. 3.1 to discover possible relationships between stability of code and its other properties. The vertical axis measures stability (on the left diagram) or groundedness (on the right): 1 means that all recorded instances of a method are stable/grounded, and 0 means that none of them are. The horizontal axis measures the property of interest; in the case of Fig. 3.1, it is method size (actual numbers are not important here: they are computed from Julia's internal representation of source code). The color of an area reflects how many methods have characteristics corresponding to that area's position on the diagram; e.g. in Fig. 3.1, the lonely yellow areas indicate that there are about 500 (400) small methods that are stable (grounded).

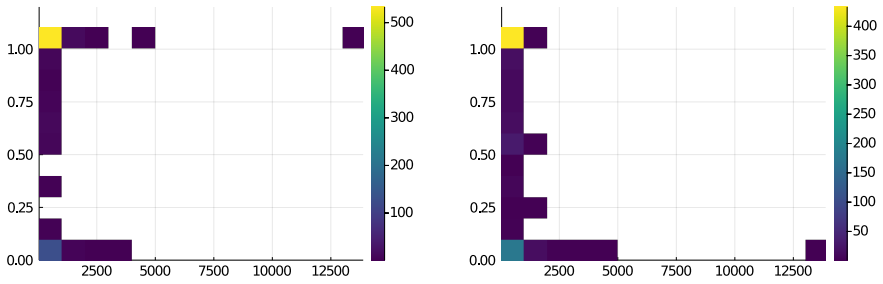

Figure 3.1: Stability (left, OY axis) and groundedness (right, OY) by method size (OX) in Pluto

I generate graphs for all of the 10 packages listed in Table 3.2, for all combinations of the properties of interest; the graphs are provided in Appendix A. Most of the graphs look very similar to the ones from Fig. 3.1, which depicts Pluto—a package for creating Jupyter-like notebooks in Julia. In the following paragraphs, I discuss features of these graphs and highlight the discrepancies.

The first distinctive feature of the graphs is the hot area in the top-left corner: most of the 10 packages employ many small, stable/grounded methods; the bottom-left corner is usually the second-most hot, so a significant number of small methods are unstable/ungrounded. For the Knet package, these two corners are reversed; for DifferentialEquations, they are reversed only on the groundedness plot. Both of these facts are not surprising after seeing Table 3.2, but having a visual tool to discover such facts may be useful for package developers.

The second distinctive feature of these graphs is the behavior of large, ungrounded methods (bottom-right part of the right-hand-side

graph). The "tail" of large methods on the groundedness graphs almost always lies below the 1-level; furthermore, larger methods tend to be less grounded. However, if we switch from groundedness to stability plots, a large portion of the tail jumps to the 1-level. This means larger methods are unlikely to be grounded (as expected, because of the growing number of registers), but they still can be stable and thus efficiently used by other methods. Pluto provides a good example of such a method: its `explore!` method of size 13003 (right-most rectangle on Fig. 3.1, 330 lines in the source code) analyzes Julia syntax trees for scope information with a massive `if/else if/..` statement. This method has a very low chance of being grounded, and it was not grounded on the runs we analyzed. However, the method has a concrete return type annotation, so Julia (and the programmer) can easily see that it is stable.

In the case of the number of gotos and returns, the plots are largely similar to the ones for method size, but they highlight one more package with low groundedness. Namely, the Gen package (aimed at probabilistic inference [Cusumano-Towner et al. 2019]) has the hottest area in the bottom-left corner, contrary to the first general property we identified for the size-based plots. Recall (Tables 3.1 and 3.2) that Gen's groundedness is 14% less than the average on the whole corpus of 760 packages.

3.3.2 Manual Inspection

To better understand the space of stable methods, I performed a qualitative analysis of a sample of stable methods that have either large sizes or many instances.

Many large methods have one common feature: they often have a return type ascription on the method header of the form:

```
function f(...) :: Int
  ...
end
```

These ascriptions are a relatively new tool in Julia, and they are used only occasionally, in my experience. An ascription makes the Julia compiler insert implicit conversions on all return paths of the method. Conversions are user extendable: if the user defines type `A`, they can also add methods of a special function `convert` for conversion to `A`. This function will be called when the compiler expects `A` but infers

a different type, for example, if the method returns `B`. If the method returns `A`, however, then `convert` is a no-op.

Type ascriptions may be added simply as documentation, but they can also be used to turn type instability into a run-time error: if the ascribed type is concrete and a necessary conversion is not available, the method will fail. This provides a useful, if unexpected, way to assure that a large method never becomes unstable.

While about 85% of type-stable methods in the top 10 packages are uninteresting in that they always return the same type, sampling the rest illuminates a clear pattern: the methods resemble what we are used to see in statically typed languages with parametric polymorphism. Below is a list of categories that we identify in this family.

- Various forms of the identity function—a surprisingly popular function that packages keep reinventing. In an impure language, such as Julia, an identity function can produce various side effects. For example, the Genie package adds a caching effect to its variant of the identity function:

```
# Define the secret token used in the app for encryption and
     salting.
function secret_token!(value::AbstractString=Generator.
    secret_token())
  SECRET_TOKEN[] = value
  return value
end
```

- Container manipulations for various kinds of containers, such as arrays, trees, or tuples. For instance, the latter is exemplified by the following function from Flux, which maps a tuple of functions by applying them to the given argument:

```
function extraChain(fs::Tuple, x)
  res = first(fs)(x)
  return (res, extraChain(Base.tail(fs), res)...)
end
extraChain(::Tuple{}, x) = ()
```

- Smart constructors for user-defined polymorphic structures. For example, the following convenience function from JuMP creates an instance of the `VectorConstraint` structure with three fields, each of which is polymorphic:

```
function build_constraint(_error::Function, Q::Symmetric{V,M
    }, ::PSDCone)
```

```
        where {V<:AbstractJuMPScalar,M<:AbstractMatrix{V}}
    n = LinearAlgebra.checksquare(Q)
    shape = SymmetricMatrixShape(n)
    return VectorConstraint(
        vectorize(Q, shape),
        MOI.PositiveSemidefiniteConeTriangle(n),
        shape)
end
```

- Type computations—an unusually wide category for a dynamically typed language. Thus, for instance, the Gen package defines a type that represents generative functions in probabilistic programming, and a function that extracts the return and argument types:

```
# Abstract type for a generative function with return value
    type T and trace type U.
abstract type GenerativeFunction{T,U <: Trace} end
get_return_type(::GenerativeFunction{T,U}) where {T,U} = T
get_trace_type(::GenerativeFunction{T,U}) where {T,U} = U
```

3.4 TAKEAWAYS

The analysis shows that a Julia user can expect mostly stable (74%) and somewhat grounded (57%) code in widely used Julia packages. If the authors are intentional about performance and stability, as demonstrated by the Genie package, those numbers can be much higher. Although the sample of packages is too small to draw strong conclusions, I suggest that several factors can be used by a Julia programmer to pinpoint potential sources of instability in their package. For example, in some cases, varargs methods might indicate instability. Large methods, especially ones with heavy control flow, tend to not be type grounded but often are stable; in particular, if they always return the same concrete type. Finally, although highly polymorphic methods are neither stable nor unstable in general, code written in the style of parametric polymorphism often suggests type stability.

The dynamic analysis and visualization code is written in Julia (and some bash code), and relies on the vanilla Julia implementation. Thus, it can be employed by package developers to study type instability in their code, as well as check for regressions.

4 FORMALIZING TYPE STABILITY AT RUN TIME: JULES

To simplify reasoning about type stability and groundedness, I first define Jules, an abstract machine that provides an idealized version of Julia's intermediate representation (IR) and compilation model. Jules captures the just-in-time (JIT) compilation process that (1) specializes methods for concrete argument types as the program executes, and (2) replaces dynamically dispatched calls with direct method invocations when type inference is able to get precise information about the argument types. It is the type inference algorithm that directly affects type stability and groundedness of code, and thus the ability of the JIT compiler to optimize it. While Julia's actual type inference algorithm is quite complex, its implementation is not relevant for understanding our properties of interest; thus, Jules abstracts over type inference and uses it as a black box.

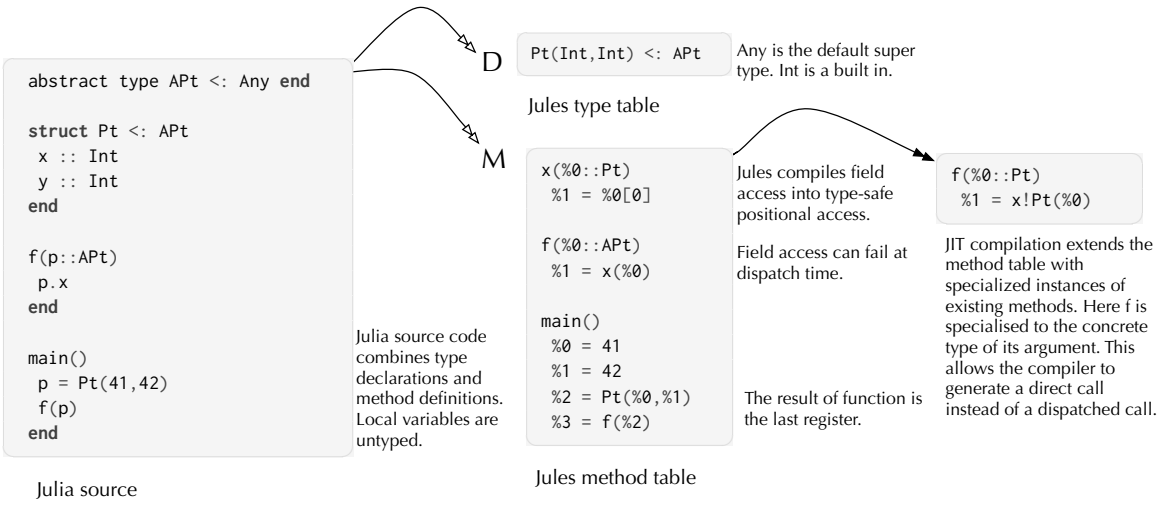

Figure 4.1: Compilation from Julia to Jules

Fig. 4.1 illustrates the relationship between Julia source code, the Jules intermediate representation, and the result of compilation. I do not model the translation from the source to Jules, and simply assume that the front-end generates well-formed Jules code.[1] A Jules program

1 The front-end does not devirtualize function calls, as Julia programmers do not have the ability to write direct method invocations in the source.

consists of an immutable *type table* D and a *method table* M; the method table can be incrementally extended with method instances that are compiled by the just-in-time compiler.

The source program of Fig. 4.1 defines two types, the concrete `Pt` and its parent, the abstract type `APt`, as well as two methods, `f` and `main`. When translated to Jules, `Pt` is added to the type table along with its supertype `APt`. Similarly, the methods `main` and `f` are added to the Jules method table, along with accessors for the fields of `Pt`, with bodies translated to the Jules intermediate representation.

The Jules IR is similar to static single assignment form. Each statement can access values in registers, and saves its result into a new, consecutively numbered, register. Statements can perform a dispatched call `f(%2)`, direct call `x!Pt(%0)`, conditional assignment (not shown), and a number of other operations. The IR is untyped, but the translation from Julia is type sound. In particular, type soundness guarantees that only dispatch errors can occur at run time. For example, compilation will produce only well-formed field accesses such as the one in `x(%0::Pt)`, but a dispatched call `x(%0)` in `f` could fail if `f` was called with a struct that did not have an `x` field. In order to perform this translation, Jules uses type inference to determine the types of the program's registers. We abstract over this type inference mechanism and only specify that it produces sound (with respect to our dynamic semantics) results.

Execution in Jules occurs between *configurations* consisting of both a stack of *frames* F (representing the current execution state) and a *method table* M (consisting of original methods and specialized instances), denoted F, M. A configuration F, M evolves to a new configuration F′, M′ by stepping $\mathsf{F}, \mathsf{M} \rightarrow \mathsf{F}', \mathsf{M}'$; every step processes the top instruction in F and possibly compiles new method instances into M′. Notably, due to the so-called world-age mechanism [Belyakova et al. 2020] which restricts the effect of `eval`, source methods are fixed from the perspective of compilation; only compiled instances changes.

4.1 SYNTAX

The syntax of Jules methods is defined in Fig. 4.2. I use two key notational devices. First, sequences are denoted $\overline{\cdot}$; thus, $\overline{\mathsf{ty}}$ stands for types $\mathsf{ty}_0 \ldots \mathsf{ty}_n$, $\overline{\%\mathsf{k}}$ for registers $\%\mathsf{k}_0 \ldots \%\mathsf{k}_n$, and $\overline{\mathsf{st}}$ for instructions $\mathsf{st}_0 \ldots \mathsf{st}_n$. An empty sequence is written ϵ. Second, indexing is denoted $[\cdot]$; $\overline{\mathsf{ty}}[\mathsf{k}]$ is the k-th type in $\overline{\mathsf{ty}}$ (starting from 0), $\%\mathsf{j}[\mathsf{k}]$ is the k-th

field of %j, and M[m!$\overline{\text{ty}}$] indexes M by method signature where m denotes method name and $\overline{\text{ty}}$ denotes argument types.

ty	::=		type
	|	T	*concrete type*
	|	A	*abstract type*
D	::=		type table
		(T!$\overline{\text{ty}}$ <: A)*	
M	::=		method table
		(⟨m!$\overline{\text{ty}'}$ $\overline{\text{st}}$, $\overline{\text{ty}}$⟩)*	
st	::=		instruction
	|	%i ← p	*int. assignment*
	|	%i ← %j	*reg. transfer*
	|	%i ← T($\overline{\text{\%k}}$)	*allocation*
	|	%i ← %j[k]	*field access*
	|	%i ← %j ? m($\overline{\text{\%k}}$) : %l	*dispatched call*
	|	%i ← %j ? m!$\overline{\text{T}}$($\overline{\text{\%k}}$) : %l	*direct call*

i ∈ ℕ, p ∈ ℤ

Figure 4.2: Syntax of Jules

Types ty live in the immutable type table D, which contains both concrete (T) and abstract (A) types. Each type table entry is of the form T!$\overline{\text{ty}}$ <: A, introducing concrete type T, with fields of types $\overline{\text{ty}}$, along with the single supertype A. Two predefined types are the concrete integer type Int, and the universal abstract supertype Any.

Method tables M contain method definitions of two sorts: first, original methods that come from source code; second, method instances compiled from original methods. To distinguish between the two sorts, the type signature of the original method is stored in every table entry. Thus, table entry ⟨m!$\overline{\text{ty}'}$ $\overline{\text{st}}$, $\overline{\text{ty}}$⟩ describes a method m with parameter types $\overline{\text{ty}'}$ and the body comprised of instructions $\overline{\text{st}}$; type signature $\overline{\text{ty}}$ points to the original method. If $\overline{\text{ty}}$ is equal to $\overline{\text{ty}'}$, the entry defines an original method, and $\overline{\text{st}}$ cannot contain direct calls.[2] Otherwise, $\overline{\text{ty}'}$ denotes some concrete types $\overline{\text{T}}$, and the entry defines a method instance compiled from m!$\overline{\text{ty}}$, specialized for concrete argument types $\overline{\text{T}}$ <: $\overline{\text{ty}}$. Method table may contain multiple method definitions with the same name, but they have to have distinct type signatures.

Method bodies consist of instructions $\overline{\text{st}}$. An instruction %i ← op consists of an operation op whose result is assigned to register %i. An instruction can assign from a primitive integer p, another register %j, a newly created struct T($\overline{\text{\%k}}$) of type T with field values $\overline{\text{\%k}}$, or the result of looking up a struct field as %j[k]. Finally, the instruction may perform a function call. Calls can be dispatched, m($\overline{\text{\%k}}$), where

2 All function calls in Julia source code are dispatched calls.

the target method is dynamically looked up, or they can be direct, $m!\overline{T}(\overline{\%k})$, where the target method is specified. All calls are conditional: %j ? `call` : %l, to allow for recursive functions. If the register %j is non-zero, then `call` is performed. Otherwise, the result is the value of register %l. Conditional calls can be trivially transformed into unconditional calls; in examples, this transformation is performed implicitly.

4.2 DYNAMIC SEMANTICS

Jules is parameterized over three components: method dispatch $\mathcal{D}$, type inference $\mathcal{I}$, and just-in-time compilation *jit*. I do not specify how the first two work, and merely provide their interface and a set of criteria that they must meet (in sections 4.2.2 and 4.2.3, respectively). The compiler, *jit*, is instantiated with either the identity function, which gives a regular non-optimizing semantics, or an optimizing compiler, which is defined in section 4.2.5. The optimizing compiler relies on type inference $\mathcal{I}$ to devirtualize method calls. Type inference also ensures well-formedness of method tables. Method dispatch $\mathcal{D}$ is used in operational semantics.

4.2.1 Operational Semantics

Fig. 4.3 gives rules for the dynamic semantics. Given a type table D as context, Jules can step a configuration F, M to F′, M′, written as $F, M \rightarrow F', M'$. Stack frames F consist of a sequence of environment-instruction list pairs. Thus, $E\ \overline{st} \cdot F$ denotes a stack with environment E and instructions $\overline{st}$ on top, followed by a sequence of environment-instruction pairs. Each environment is a list of values $E = \overline{v}$, representing contents of the sequentially numbered registers. Environments can then be extended as $E + v$, indexed as $E[k]$, and their last value is $\mathit{last}(E)$ if E is not empty.

The small-step dynamic semantics is largely straightforward. The first four rules deal with register assignment: updating the environment with a constant value (Prim), the value in another register (Reg), a newly constructed struct (New), or the value in a field (Field). The remaining five rules deal with function calls, either dispatched $m(\overline{\%k})$ or direct $m!\overline{T}(\overline{\%k})$. Call instructions are combined with conditioning: a call can only be made after testing the register %j, called a guard regis-

$$v ::= p \mid T(\overline{v}) \qquad E ::= \overline{v} \qquad F ::= \epsilon \mid E\ \overline{st} \cdot F$$

$$\boxed{F, M \ \rightarrow \ F', M}$$

$$\textsc{Prim}\ \frac{st = \%i \leftarrow p \qquad E' = E + p}{E\ st\ \overline{st} \cdot F, M \ \rightarrow \ E'\ \overline{st} \cdot F, M} \qquad \textsc{Reg}\ \frac{st = \%i \leftarrow \%j \qquad E' = E + E[j]}{E\ st\ \overline{st} \cdot F, M \ \rightarrow \ E'\ \overline{st} \cdot F, M}$$

$$\textsc{New}\ \frac{st = \%i \leftarrow T(\overline{\%k}) \qquad E' = E + T(E[\overline{k}])}{E\ st\ \overline{st} \cdot F, M \ \rightarrow \ E'\ \overline{st} \cdot F, M}$$

$$\textsc{Field}\ \frac{st = \%i \leftarrow \%j[k] \qquad v = E[j] \qquad E' = E + v[k]}{E\ st\ \overline{st} \cdot F, M \ \rightarrow \ E'\ \overline{st} \cdot F, M} \qquad \textsc{False1}\ \frac{st = \%i \leftarrow \%j\ ?\ m(\overline{\%k}) : \%l \qquad 0 = E[j] \qquad E' = E + E[l]}{E\ st\ \overline{st} \cdot F, M \ \rightarrow \ E'\ \overline{st} \cdot F, M}$$

$$\textsc{False2}\ \frac{st = \%i \leftarrow \%j\ ?\ m!\overline{T}(\overline{\%k}) : \%l \qquad 0 = E[j] \qquad E' = E + E[l]}{E\ st\ \overline{st} \cdot F, M \ \rightarrow \ E'\ \overline{st} \cdot F, M}$$

$$\textsc{Disp}\ \frac{st = \%i \leftarrow \%j\ ?\ m(\overline{\%k}) : \%l \qquad 0 \neq E[j] \qquad \overline{T} = \mathit{typeof}(E[\overline{k}]) \qquad M' = \mathit{jit}(M, m, \overline{T}) \qquad \overline{st'} = \mathit{body}(\mathcal{D}(M', m, \overline{T})) \qquad E' = E[\overline{k}]}{E\ st\ \overline{st} \cdot F, M \ \rightarrow \ E'\ \overline{st'} \cdot E\ st\ \overline{st} \cdot F, M'}$$

$$\textsc{Direct}\ \frac{st = \%i \leftarrow \%j\ ?\ m!\overline{T}(\overline{\%k}) : \%l \qquad 0 \neq E[j] \qquad \overline{st'} = \mathit{body}(M[m!\overline{T}]) \qquad E' = E[\overline{k}]}{E\ st\ \overline{st} \cdot F, M \ \rightarrow \ E'\ \overline{st'} \cdot E\ st\ \overline{st} \cdot F, M} \qquad \textsc{Ret}\ \frac{E'' = E + \mathit{last}(E')}{E'\ \epsilon \cdot E\ st\ \overline{st} \cdot F, M \ \rightarrow \ E''\ \overline{st} \cdot F, M}$$

Figure 4.3: Dynamic semantics of Jules

ter. If the register value is zero, then the value of the alternate register %l is returned (False1/False2). Otherwise, the call can proceed, by Disp for dispatched calls and Direct for direct ones. A dispatched call starts by prompting the JIT compiler to specialize method m from the method table M with the argument types $\overline{T}$ and produce a new method table M′. Next, using the new table M′, the dispatch mechanism $\mathcal{D}$ determines the method to invoke. Finally, the body $\overline{st'}$ of the method and call arguments $E[\overline{k}]$ form a new stack frame for the callee, and the program steps with the extended stack and the new table. Direct calls are simpler because a direct call $m!\overline{T}$ uniquely identifies the method to invoke. Thus, the method's instructions are looked up

in M by the method signature, a new stack frame is created, and the program steps with the new stack and the same method table. The top frame without instructions to execute indicates the end of a function call (RET): the last value of the top-frame environment becomes the return value of the call, and the top frame is popped from the stack.

Program execution begins with the frame ϵ main()[3], i.e. a call to the main function with an empty environment. The execution either diverges, finishes with a final configuration E ϵ, or runs into an error.

We define two notions of error. An *err* occurs only in the DISP rule, when the dispatch function $\mathcal{D}$ is undefined for the call; the *err* corresponds to a dynamic-dispatch error in Julia. A configuration is *wrong* if it cannot make a step for any other reason. The well-formedness criteria below (Subsec. 4.2.4) should rule out *wrong* programs statically, but *err*s may happen at run time.

Definition 4.1 (Errors)**.** *A non-empty configuration* F, M *that cannot step* F, M $\rightarrow$ F′, M′ *has* erred *if its top-most frame,* E $\overline{\mathsf{st}}$*, starts with* %i $\leftarrow$ %j ? m($\overline{\%\mathsf{k}}$) : %l*, where* $\overline{\mathsf{T}}$ *is the types of* $\overline{\%\mathsf{k}}$ *in* E, m $\in$ M*, and* $\mathcal{D}$(M, m, $\overline{\mathsf{T}}$) *is undefined. Otherwise,* F, M *is* wrong*.*

4.2.2 Dispatch

Jules is parametric over method dispatch: any mechanism $\mathcal{D}$ that satisfies the *Dispatch Contract* (Def. 4.2) can be used. Julia's method dispatch mechanism is designed to, given method table, method name, and argument types, return the *most specific method applicable* to the given arguments if such a method exists and is unique. First, applicable methods are those whose declared type signature is a supertype of the argument type. Then, the most specific method is the one whose type signature is the most precise. Finally, only one most specific applicable method may exist, or else an error is produced. Each of these components appears in our dispatch definition. As in Julia, dispatch is only defined for tuples of concrete types.

Definition 4.2 (Dispatch Contract)**.** *The* dispatch *function* $\mathcal{D}$(M, m, $\overline{\mathsf{T}}$) *takes method table* M*, method name* m*, and concrete argument types* $\overline{\mathsf{T}}$*, and returns a method* m!$\overline{\mathsf{ty}}$ $\overline{\mathsf{st}}$ $\in$ M *such that the following holds (we write* $\overline{\mathsf{ty}} <: \overline{\mathsf{ty}'}$ *as a shorthand for* $\forall \mathsf{i}.\, \mathsf{ty}_\mathsf{i} <: \mathsf{ty}'_\mathsf{i}$*):*

1. $\overline{\mathsf{T}} <: \overline{\mathsf{ty}}$*, meaning that* m!$\overline{\mathsf{ty}}$ *is applicable to the given arguments;*

3 Recall that unconditional calls are implicitly expanded into conditional ones.

2. $\forall \mathsf{m}!\overline{\mathsf{ty}'}\,\overline{\mathsf{st}} \in \mathsf{M}.\ \overline{\mathsf{T}} <: \overline{\mathsf{ty}'} \implies \overline{\mathsf{ty}} <: \overline{\mathsf{ty}'}$, *meaning that* $\mathsf{m}!\overline{\mathsf{ty}}$ *is the most specific applicable method.*

4.2.3 Inference

The Julia compiler infers types of variables by forward data-flow analysis. Like dispatch, inference is complex, so it is being parameterized over. For our purposes, an inference algorithm $\mathcal{I}$ returns a sound typing for a sequence of instructions in a given method table, $\mathcal{I}(\lfloor \mathsf{M} \rfloor, \overline{\mathsf{ty}}, \overline{\mathsf{st}}) = \overline{\mathsf{ty}'}$, where $\lfloor \mathsf{M} \rfloor$ denotes the table containing only methods without direct calls. Inference returns types $\overline{\mathsf{ty}}'$ such that each ty'_i is the type of register of st_i. Any inference algorithm that satisfies the Soundness and Monotonicity requirements is acceptable.

Requirement 4.2.1 (Soundness)**.** *If* $\mathcal{I}(\mathsf{M}, \overline{\mathsf{ty}}, \overline{\mathsf{st}}) = \overline{\mathsf{ty}'}$*, then for any environment* E *compatible with* $\overline{\mathsf{ty}}$*, that is, typeof*$(\mathsf{E}_i) <: \mathsf{ty}_i$*, and for any stack* F*, the following holds:*

1. *If* $\mathsf{E}\ \overline{\mathsf{st}} \cdot \mathsf{F}, \mathsf{M} \ \to^*\ \mathsf{F}' \cdot \mathsf{F}, \mathsf{M}'$ *and cannot make another step, then* $\mathsf{F}' \cdot \mathsf{F}, \mathsf{M}'$ *has* erred*.*
2. *If* $\mathsf{E}\,\overline{\mathsf{st}} \cdot \mathsf{F},\ \mathsf{M} \ \to^*\ \mathsf{E}\,\mathsf{E}'\,\overline{\mathsf{st}''} \cdot \mathsf{F},\ \mathsf{M}'$*, then* $\overline{\mathsf{st}} = \overline{\mathsf{st}'}\,\overline{\mathsf{st}''}$ *and typeof*$(\mathsf{E}'_i) <: \mathsf{ty}'_i$*.*

The soundness requirement guarantees that if type inference succeeds on a method, then, when the method is called with compatible arguments, it will not enter a *wrong* state but may *err* at a dynamic call. Furthermore, if the method terminates, all its instructions evaluate to values compatible with the results of type inference. That is, when $\mathsf{E}\ \overline{\mathsf{st}} \cdot \mathsf{F},\ \mathsf{M} \ \to^*\ \mathsf{E}\,\mathsf{E}'\ \epsilon \cdot \mathsf{F},\ \mathsf{M}'$, we have *typeof*$(\mathsf{E}') <: \overline{\mathsf{ty}'}$.

The second requirement of type inference, monotonicity, is important to specialization: it guarantees that using more precise argument types for original method bodies succeeds and does not break assumptions of the caller about the callee's return type. If inference was not monotonic, then given more precise argument types, it could return a method specialization with a less precise return type. As a result, translating a dynamically dispatched call into a direct call may be unsound.

Requirement 4.2.2 (Monotonicity)**.** *For all* $\mathsf{M}, \overline{\mathsf{ty}}, \overline{\mathsf{st}}, \overline{\mathsf{ty}'}$*, such that* $\mathcal{I}(\mathsf{M}, \overline{\mathsf{ty}}, \overline{\mathsf{st}}) = \overline{\mathsf{ty}'}$*,*

$$\forall\, \overline{\mathsf{ty}''}.\quad \overline{\mathsf{ty}''} <: \overline{\mathsf{ty}} \quad\implies\quad \exists \overline{\mathsf{ty}'''}.\ \ \mathcal{I}(\mathsf{M}, \overline{\mathsf{ty}''}, \overline{\mathsf{st}}) = \overline{\mathsf{ty}'''} \ \wedge\ \overline{\mathsf{ty}'''} <: \overline{\mathsf{ty}'}.$$

4.2.4 Well-Formedness

Initial Jules configuration ϵ main(), M is well-formed if the method table M is well-formed according to Def. 4.3. Such a configuration will either successfully terminate, *err*, or run forever, but it will never reach a *wrong* state.

Definition 4.3 (Well-Formedness of Method Table)**.** *A method table is* well-formed *WF*(M) *if the following holds:*

1. *Entry method* main!ϵ *belongs to* M.
2. *Every type* ty *in* M, *except* Int *and* Any, *is declared in* D.
3. *Registers are numbered consecutively from* 0, *increasing by one for each parameter and instruction. An instruction assigning to* %k *only refers to registers* %i *such that* $i < k$.
4. *For any original method* $\langle m!\overline{ty}\,\overline{st},\ \overline{ty}\rangle \in M$, *the body is not empty and does not contain direct calls, and type inference succeeds* $\mathcal{I}(\lfloor M \rfloor, \overline{ty}, \overline{st}) = \overline{ty'}$ *on original methods* $\lfloor M \rfloor$.
5. *Any two methods in* M *with the same name,* $m!\overline{ty}$ *and* $m!\overline{ty'}$, *have distinct type signatures, i.e.* $\overline{ty} \neq \overline{ty'}$.
6. *For any method specialization* $\langle m!\overline{ty'}, \overline{ty}\rangle \in M$, *i. e.* $\overline{ty'} \neq \overline{ty}$, *the following holds:* $\overline{ty'} = \overline{T}$, *and* $\overline{T} <: \overline{ty}$, *and* $\langle m!\overline{ty}, \overline{ty}\rangle \in M$. *Moreover,* $\forall m!\overline{ty}'' \in M.\ \overline{T} <: \overline{ty''} \implies \overline{ty} <: \overline{ty''}$.

The last requirement ensures that only the most specific original methods have specializations, which precludes compilation from modifying program behavior. For example, consider type hierarchy `Int<:Number<:Any` and function `f` with original methods for `Any` and `Number`. If there are no compiled method instances, the call `f(5)` dispatches to `f(::Number)`. But if the method table contained a specialized instance `f(::Int)` of the original method `f(::Any)`, the call would dispatch to that instance, which is not related to the originally used `f(::Number)`. Thus, program behavior would be modified by compilation, which is undesired.

4.2.5 Compilation

Jules implements devirtualization through the $jit(M, m, \overline{T})$ operation, shown in Fig. 4.4. The compiler specializes methods according to the

$$\frac{\begin{array}{c} m!\overline{T} \notin M \qquad \overline{st} = \mathit{body}(\mathcal{D}(M, m, \overline{T})) \qquad \overline{ty'} = \mathcal{I}(\lfloor M \rfloor, \overline{T}, \overline{st}) \\ \overline{ty} = \mathit{signature}(\mathcal{D}(M, m, \overline{T})) \qquad M_0 = M + \langle m!\overline{T}\epsilon, \overline{ty} \rangle \\ \overline{T}\,\overline{ty'} \vdash st_0, M_0 \rightsquigarrow st_0', M_1 \qquad \ldots \qquad \overline{T}\,\overline{ty'} \vdash st_n, M_n \rightsquigarrow st_n', M_{n+1} \\ M' = M_{n+1}[m!\overline{T} \mapsto \langle m!\overline{T}\,\overline{st'}, \overline{ty} \rangle] \end{array}}{\mathit{jit}(M, m, \overline{T}) = M'}$$

$$\frac{m!\overline{T} \in M}{\mathit{jit}(M, m, \overline{T}) = M} \qquad \frac{\begin{array}{c} st = \%i \leftarrow \%j\ ?\ m(\overline{\%k}) : \%l \qquad \overline{T} = \overline{ty}[\overline{k}] \\ M' = \mathit{jit}(M, m, \overline{T}) \qquad st' = \%i \leftarrow \%j\ ?\ m!\overline{T}(\overline{\%k}) : \%l \end{array}}{\overline{ty} \ \vdash\ st, M \rightsquigarrow st', M'}$$

$$\frac{\begin{array}{c} st \neq \%i \leftarrow \%j\ ?\ m(\overline{\%k}) : \%l \\ \text{or} \qquad \overline{T} \neq \overline{ty}[\overline{k}] \end{array}}{\overline{ty} \ \vdash\ st, M \rightsquigarrow st, M}$$

Figure 4.4: Compilation: extending method table with a specialized method instance (top rule) and replacing a dynamically dispatched call with a direct method invocation in the extended table (middle-right)

inferred type information, replacing non-*err* dispatched calls with direct calls where possible. Compilation begins with some bookkeeping. First, it ensures that there is no pre-existing instance in the method table before compiling; otherwise, the table is returned immediately without modification, by the bottom rule. Next, using dispatch, it fetches the most applicable original method to compile. Then, using concrete argument types, the compiler runs the type inferencer on the method's body, producing an instruction typing $\overline{ty'}$. Because the original method table is well-formed, monotonicity of $\mathcal{I}$ and the definition of $\mathcal{D}$ guarantee that type inference succeeds for $\overline{T} <: \overline{ty}$. Lastly, the compiler can begin translating instructions. Each instruction st_i is translated into an optimized instruction st_i'. This translation respects the existing type environment containing the argument types $\overline{T}$ and instruction typing $\overline{ty'}$. The translation $\overline{ty} \ \vdash\ st, M \rightsquigarrow st', M'$ leaves all instructions unchanged except dispatched calls. Dispatched calls cause a recursive JIT invocation, followed by rewriting to a direct call. To avoid recursive compilation, a stub method $\langle m!\overline{T}\epsilon, \overline{ty} \rangle$ is added at the beginning of compilation, and over-written when compilation is done. As the source program is finite and new types are not added during compilation, it terminates. Note that if the original method has concrete argument types, the compiler does nothing.

4.3 TYPE GROUNDEDNESS AND STABILITY

I now formally define the properties of interest, type stability and type groundedness. Recall the informal definition which stated that a function is type stable if its return type depends only on its argument types, and type grounded if every variable's type depends only on the argument types. In this definition, "types" really mean concrete types, as concrete types allow for optimizations. The following defines what it means for an original method to be type stable and type grounded.

Definition 4.4 (Stable and Grounded)**.** *Let* $\mathsf{m}!\overline{\mathsf{ty}}\,\overline{\mathsf{st}}$ *be an original method in a well-formed method table* M*. Given concrete argument types* $\overline{\mathsf{T}} <: \overline{\mathsf{ty}}$*, if* $\mathcal{I}(\lfloor \mathsf{M} \rfloor, \mathsf{m}, \overline{\mathsf{T}}) = \overline{\mathsf{ty}'}$*, and*

1. $\mathit{last}(\overline{\mathsf{ty}'}) = \mathsf{T}'$*, i.e. the return type is concrete, then the method is* type stable for $\overline{\mathsf{T}}$*,*
2. $\overline{\mathsf{ty}'} = \overline{\mathsf{T}'}$*, i.e. all register types are concrete, then the method is* type-grounded for $\overline{\mathsf{T}}$*.*

Furthermore, a method is called type stable (grounded) for a set W of concrete argument types *if it is type stable (grounded) for every* $\overline{\mathsf{T}} \in W$*.*

As all method instances are compiled from some original method definitions, type stabiltiy and groundedness for instances is defined in terms of their originals.

Definition 4.5. *A method instance* $\langle \mathsf{m}!\overline{\mathsf{T}}\,\overline{\mathsf{st}'},\ \overline{\mathsf{ty}} \rangle$ *is called* type stable (grounded)*, if its original method* $\mathsf{m}!\overline{\mathsf{ty}}$ *is type stable (grounded) for* $\overline{\mathsf{T}}$*.*

4.3.1 Full Devirtualization

The key property of type groundedness is that compiling a type-grounded method results in a fully devirtualized instance. We say that a method instance $\mathsf{m}!\overline{\mathsf{T}}\ \ \overline{\mathsf{st}}$ is *fully devirtualized* if $\overline{\mathsf{st}}$ does not contain any dispatched calls. To show that *jit* indeed has the above property, we will need an additional notion, *maximal devirtualization*, which is defined in Fig. 4.5. Intuitively, the predicate $\overline{\mathsf{ty}} \vdash^{\mathcal{D}}_{\mathsf{M}} \mathsf{st}$ states that an instruction st does not contain a dispatched call that can be resolved in table M for argument types found in $\overline{\mathsf{ty}}$. Then, a method instance is maximally devirtualized if this predicate holds for every instruction using $\overline{\mathsf{ty}}$ that combines argument types with the results of type inference.

$$\boxed{\overline{\mathsf{ty}} \vdash^{\mathcal{D}}_{\mathsf{M}} \mathsf{st}}$$

D-NoCall
$$\frac{\mathsf{st} \neq \%\mathsf{i} \leftarrow \%\mathsf{j}\ ?\ \mathsf{m}(\overline{\%\mathsf{k}}) : \%\mathsf{l} \qquad \mathsf{st} \neq \%\mathsf{i} \leftarrow \%\mathsf{j}\ ?\ \mathsf{m}!\overline{\mathsf{T}}(\overline{\%\mathsf{k}}) : \%\mathsf{l}}{\overline{\mathsf{ty}} \vdash^{\mathcal{D}}_{\mathsf{M}} \mathsf{st}}$$

D-Disp
$$\frac{\mathsf{st} = \%\mathsf{i} \leftarrow \%\mathsf{j}\ ?\ \mathsf{m}(\overline{\%\mathsf{k}}) : \%\mathsf{l} \qquad \overline{\mathsf{T}} \neq \overline{\mathsf{ty}}[\overline{\mathsf{k}}]}{\overline{\mathsf{ty}} \vdash^{\mathcal{D}}_{\mathsf{M}} \mathsf{st}}$$

D-Direct
$$\frac{\mathsf{st} = \%\mathsf{i} \leftarrow \%\mathsf{j}\ ?\ \mathsf{m}!\overline{\mathsf{T}}(\overline{\%\mathsf{k}}) : \%\mathsf{l} \qquad \overline{\mathsf{T}} = \overline{\mathsf{ty}}[\overline{\mathsf{k}}] \qquad \mathsf{m}!\overline{\mathsf{T}} \in \mathsf{M}}{\overline{\mathsf{ty}} \vdash^{\mathcal{D}}_{\mathsf{M}} \mathsf{st}}$$

$$\boxed{\overline{\mathsf{ty}} \vdash^{\mathcal{D}}_{\mathsf{M}} \overline{\mathsf{st}}}$$

D-Seq
$$\frac{\overline{\mathsf{ty}} \vdash^{\mathcal{D}}_{\mathsf{M}} \mathsf{st}_0 \qquad \ldots \qquad \overline{\mathsf{ty}} \vdash^{\mathcal{D}}_{\mathsf{M}} \mathsf{st}_n}{\overline{\mathsf{ty}} \vdash^{\mathcal{D}}_{\mathsf{M}} \overline{\mathsf{st}}}$$

$$\boxed{\vdash^{\mathcal{D}} \mathsf{M}}$$

D-Table
$$\frac{\begin{array}{c}\forall \langle \mathsf{m}!\overline{\mathsf{T}}\,\overline{\mathsf{st}'}, \overline{\mathsf{ty}} \rangle \in \mathsf{M}.\quad \overline{\mathsf{T}} \neq \overline{\mathsf{ty}} \ \wedge\ \overline{\mathsf{st}'} \neq \epsilon \\ \Longrightarrow \\ \mathsf{m}!\overline{\mathsf{ty}}\,\overline{\mathsf{st}} \in \mathsf{M} \ \wedge\ \overline{\mathsf{ty}'} = \mathcal{I}(\lfloor \mathsf{M} \rfloor, \overline{\mathsf{T}}, \overline{\mathsf{st}}) \ \wedge\ \overline{\mathsf{T}}\,\overline{\mathsf{ty}'} \vdash^{\mathcal{D}}_{\mathsf{M}} \overline{\mathsf{st}'}\end{array}}{\vdash^{\mathcal{D}} \mathsf{M}}$$

Figure 4.5: Maximal devirtualization of instructions and method tables

Next, let us review the definition from Fig. 4.5 in more details. D-NoCall states that an instruction without a call is maximally devirtualized. D-Disp requires that for a dispatched call, $\overline{\mathsf{ty}}$ does not have precise enough type information to resolve the call with $\mathcal{D}$. Finally, D-Direct allows a direct call to a concrete method with the right type signature: as concrete types do not have subtypes, the $\mathsf{m}!\overline{\mathsf{T}}$ with concrete argument types is exactly the definition that a call $\mathsf{m}(\overline{\%\mathsf{k}})$ would dispatch to. D-Seq simply checks that all instructions in a sequence $\overline{\mathsf{st}}$ are devirtualized in the same register typing $\overline{\mathsf{ty}}$. Here, $\overline{\mathsf{ty}}$ has to contain typing for all instructions st_i, because later instructions refer to the registers of the previous ones. The last rule $\vdash^{\mathcal{D}} \mathsf{M}$ covers the entire table M, requiring all methods to be maximally devirtualized. Namely, D-Table says that for all method instances (condition $\overline{\mathsf{T}} \neq \overline{\mathsf{ty}}$ implies that $\mathsf{m}!\overline{\mathsf{T}}$ is not an original method) that are not stubs ($\overline{\mathsf{st}'} \neq \epsilon$), the

$$\boxed{\overline{\mathsf{ty}} \vdash \mathsf{st}, \mathsf{M} \rightsquigarrow \mathsf{st}', \mathsf{M}'}$$

C-NoDisp
$$\frac{\mathsf{st} \neq \%\mathsf{i} \leftarrow \%\mathsf{j}\ ?\ \mathsf{m}(\overline{\%\mathsf{k}}) : \%\mathsf{l}}{\overline{\mathsf{ty}} \vdash \mathsf{st}, \mathsf{M} \rightsquigarrow \mathsf{st}, \mathsf{M}}$$

C-Disp
$$\frac{\mathsf{st} = \%\mathsf{i} \leftarrow \%\mathsf{j}\ ?\ \mathsf{m}(\overline{\%\mathsf{k}}) : \%\mathsf{l} \qquad \overline{\mathsf{T}} \neq \overline{\mathsf{ty}}[\overline{\mathsf{k}}]}{\overline{\mathsf{ty}} \vdash \mathsf{st}, \mathsf{M} \rightsquigarrow \mathsf{st}, \mathsf{M}}$$

C-Direct
$$\frac{\mathsf{st} = \%\mathsf{i} \leftarrow \%\mathsf{j}\ ?\ \mathsf{m}(\overline{\%\mathsf{k}}) : \%\mathsf{l} \qquad \overline{\mathsf{T}} = \overline{\mathsf{ty}}[\overline{\mathsf{k}}] \qquad \mathsf{m}!\overline{\mathsf{T}} \in \mathsf{M} \qquad \mathsf{st}' = \%\mathsf{i} \leftarrow \%\mathsf{j}\ ?\ \mathsf{m}!\overline{\mathsf{T}}(\overline{\%\mathsf{k}}) : \%\mathsf{l}}{\overline{\mathsf{ty}} \vdash \mathsf{st}, \mathsf{M} \rightsquigarrow \mathsf{st}', \mathsf{M}}$$

C-Instance
$$\frac{\begin{array}{c}\mathsf{st} = \%\mathsf{i} \leftarrow \%\mathsf{j}\ ?\ \mathsf{m}(\overline{\%\mathsf{k}}) : \%\mathsf{l} \qquad \overline{\mathsf{T}} = \overline{\mathsf{ty}}[\overline{\mathsf{k}}] \qquad \mathsf{m}!\overline{\mathsf{ty}'}\,\overline{\mathsf{st}} = \mathcal{D}(\mathsf{M}, \mathsf{m}, \overline{\mathsf{T}}) \qquad \overline{\mathsf{T}} \neq \overline{\mathsf{ty}'} \\ \overline{\mathsf{ty}''} = \mathcal{I}(\lfloor \mathsf{M} \rfloor, \overline{\mathsf{T}}, \overline{\mathsf{st}}) \qquad \mathsf{M}_0 = \mathsf{M} + \langle \mathsf{m}!\overline{\mathsf{T}}\epsilon, \overline{\mathsf{ty}'} \rangle \\ \overline{\mathsf{T}}\,\overline{\mathsf{ty}''} \vdash \mathsf{st}_0, \mathsf{M}_0 \rightsquigarrow \mathsf{st}_0', \mathsf{M}_1 \quad \ldots \quad \overline{\mathsf{T}}\,\overline{\mathsf{ty}''} \vdash \mathsf{st}_n, \mathsf{M}_n \rightsquigarrow \mathsf{st}_n', \mathsf{M}_{n+1} \\ \mathsf{st}' = \%\mathsf{i} \leftarrow \%\mathsf{j}\ ?\ \mathsf{m}!\overline{\mathsf{T}}(\overline{\%\mathsf{k}}) : \%\mathsf{l} \qquad \mathsf{M}' = \mathsf{M}_{n+1} + \langle \mathsf{m}!\overline{\mathsf{T}}\,\overline{\mathsf{st}'}, \overline{\mathsf{ty}'} \rangle\end{array}}{\overline{\mathsf{ty}} \vdash \mathsf{st}, \mathsf{M} \rightsquigarrow \mathsf{st}', \mathsf{M}'}$$

Figure 4.6: Reformulated compilation

body $\overline{\mathsf{st}'}$ is maximally devirtualized according to the typing of the original method with respect to the instance's argument types.

Using the notion of maximal devirtualization, we can connect type groundedness and full devirtualization.

Lemma 4.1 (Full Devirtualization)**.** *If* M *is well-formed and maximally devirtualized, then any type-grounded method instance* $\mathsf{m}!\overline{\mathsf{T}}\,\overline{\mathsf{st}'} \in \mathsf{M}$ *is fully devirtualized.*

Proof. Recall that a method instance is type-grounded if type inference produces concrete typing $\overline{\mathsf{T}'}$ for the original method body $\overline{\mathsf{st}}$, i.e. $\mathcal{I}(\lfloor \mathsf{M} \rfloor, \overline{\mathsf{T}}, \overline{\mathsf{st}}) = \overline{\mathsf{T}'}$. By the definition of a maximally devirtualized table, we know that $\overline{\mathsf{T}}\,\overline{\mathsf{T}'} \vdash^{\mathcal{D}}_{\mathsf{M}} \overline{\mathsf{st}'}$. Since all types in the register typing $\overline{\mathsf{T}}\,\overline{\mathsf{T}'}$ are concrete, by analyzing maximal devirtualization for instructions, we can see that the only applicable rules are D-NoCall and D-Direct. Therefore, $\overline{\mathsf{st}'}$ does not contain any dispatched calls. □

The final step is to show that compilation defined in Fig. 4.4 preserves maximal devirtualization. To simplify the proof, Fig. 4.6 reformulates Fig. 4.4 by inlining *jit* into the compilation relation. The relation does not have a rule for processing direct calls because we compile only original methods. Since the set of method instances is finite, the relation is well-defined: every recursive call to compilation happens for a method table that contains at least one more instance. Every compilation step produces a maximally devirtualized instruc-

tion and potentially extends the method table with new maximally devirtualized instances.

Lemma 4.2 (Preserving Well Formedness)**.** *For any method table* M *that is well-formed* $WF(\mathsf{M})$*, register typing* $\overline{\mathsf{ty}}$*, and instruction* st*, if the instruction* st *is compiled,* $\overline{\mathsf{ty}} \vdash \mathsf{st}, \mathsf{M} \rightsquigarrow \mathsf{st}', \mathsf{M}'$*, then the new table is well-formed* $WF(\mathsf{M}')$*.*

Proof. By induction on the derivation of $\overline{\mathsf{ty}} \vdash \mathsf{st}, \mathsf{M} \rightsquigarrow \mathsf{st}', \mathsf{M}'$. The only case that modifies the method table is C-Instance. Let us analyze M_0 first. Since $\overline{\mathsf{T}} \neq \overline{\mathsf{ty}'}$ and M is well-formed, we know that $\mathsf{m}!\overline{\mathsf{ty}'}$ is an original method and $\overline{\mathsf{T}} \neq \overline{\mathsf{ty}'}$. Furthermore, as dispatch is known to return the most applicable method, $\mathsf{m}!\overline{\mathsf{T}} \notin \mathsf{M}$ and properties (5) and (6) of Def. 4.3 for M_0 are satisfied. Other properties are trivially satisfied because M_0 does not add or modify any original methods, so:

$$WF(\mathsf{M}_0).$$

Therefore, we can apply the induction hypothesis to $\overline{\mathsf{T}}\,\overline{\mathsf{ty}''} \vdash \mathsf{st}_0, \mathsf{M}_0 \rightsquigarrow \mathsf{st}_0', \mathsf{M}_1$ to get $WF(\mathsf{M}_1)$. Proceeding in this manner, we arrive to:

$$WF(\mathsf{M}_{n+1}).$$

As M' only changes the body of the compiled instance $\mathsf{m}!\overline{\mathsf{T}}$ compared to the well-formed M_{n+1}, we get the desired conclusion:

$$WF(\mathsf{M}').$$

□

Lemma 4.3 (Preserving Maximal Devirtualization)**.** *For any well-formed method table* M*, register typing* $\overline{\mathsf{ty}}$*, and instruction* st*, if the method table is maximally devirtualized,* $\vdash^{\mathcal{D}} \mathsf{M}$*, and the instruction* st *is compiled,* $\overline{\mathsf{ty}} \vdash \mathsf{st}, \mathsf{M} \rightsquigarrow \mathsf{st}', \mathsf{M}'$*, then the following holds:*

1. *the resulting instruction is maximally devirtualized in the new table,* $\overline{\mathsf{ty}} \vdash^{\mathcal{D}}_{\mathsf{M}'} \overline{\mathsf{st}'}$*,*
2. *the new table is maximally devirtualized,* $\vdash^{\mathcal{D}} \mathsf{M}'$*,*
3. *any maximally devirtualized instruction stays maximally devirtualized in the new table,* $\forall\, \overline{\mathsf{ty}^{\mathsf{x}}}, \overline{\mathsf{st}^{\mathsf{x}}}.\ \ \overline{\mathsf{ty}^{\mathsf{x}}} \vdash^{\mathcal{D}}_{\mathsf{M}} \overline{\mathsf{st}^{\mathsf{x}}} \implies \overline{\mathsf{ty}^{\mathsf{x}}} \vdash^{\mathcal{D}}_{\mathsf{M}'} \overline{\mathsf{st}^{\mathsf{x}}}$*.*

Proof. By induction on the derivation of $\overline{\mathsf{ty}} \vdash \mathsf{st}, \mathsf{M} \rightsquigarrow \mathsf{st}', \mathsf{M}'$.

- Cases C-NoDisp and C-Disp are straightforward: to show (1), we use rules D-NoCall and D-Disp, respectively. Since the resulting method table M is the same, (2) and (3) hold trivially.

- Case C-Direct. By assumption, we know that $\overline{T} = \overline{ty}[\overline{k}]$ and that $m!\overline{T} \in M$. Therefore, we have (1) by D-Direct:

$$\overline{ty} \vdash^{\mathcal{D}}_{M} \%i \leftarrow \%j \ ? \ m!\overline{T}(\overline{\%k}) : \%l.$$

  The resulting method table M is the same, so (2) and (3) hold.

- Case C-Instance. This is the interesting case where compilation of additional instances happens. First, note that M_0 is the same as M except for a new instance stub $\langle m!\overline{T}\epsilon, \overline{ty'}\rangle$. By assumption, $\vdash^{\mathcal{D}} M$ holds, and D-Table does not impose any constraints on stubs, so $\vdash^{\mathcal{D}} M_0$ also holds. As shown in the proof of Lem. 4.2, M_0 is also well-formed. Therefore, we can apply the induction hypothesis to

$$\overline{T}\,\overline{ty''} \vdash st_0, M_0 \rightsquigarrow st_0', M_1,$$

  and we get that:

  - $\overline{T}\,\overline{ty''} \vdash^{\mathcal{D}}_{M_1} st_0'$,
  - $\vdash^{\mathcal{D}} M_1$, and
  - $\forall\, \overline{ty^x}, \overline{st^x}.\ \ \overline{ty^x} \vdash^{\mathcal{D}}_{M_0} \overline{st^x} \implies \overline{ty^x} \vdash^{\mathcal{D}}_{M_1} \overline{st^x}$.

  The fact that M_1 is maximally devirtualized by (2) and well-formed by Lem. 4.2, lets us apply the induction hypothesis to the next instruction st_1 of the method body $\overline{st}$, and so on. By repeating the same steps, we get that the final table obtained by compiling the body is also maximally devirtualized:

$$\vdash^{\mathcal{D}} M_{n+1}.$$

  Now, let us look at the property (3) that we need to prove. Assuming we have some $\overline{ty^x}$, $\overline{st^x}$ such that $\overline{ty^x} \vdash^{\mathcal{D}}_{M} \overline{st^x}$, we want to show that $\overline{ty^x} \vdash^{\mathcal{D}}_{M'} \overline{st^x}$. First, observe that by case analysis on the derivation of $\overline{ty^x} \vdash^{\mathcal{D}}_{M} \overline{st^x}$, we can show that the property holds in M_0, i.e. $\overline{ty^x} \vdash^{\mathcal{D}}_{M_0} \overline{st^x}$. For an instruction that is not a direct call, the presence of an additional method instance is irrelevant. If an instruction is a direct call to an existing method instance, it stays maximally devirtualized because M_0 does not remove or alter any existing instances of M.

Next, we can apply the fact (3) from the induction on st_0 to conclude that $\overline{\mathsf{ty}^x} \vdash^{\mathcal{D}}_{\mathsf{M}_1} \overline{\mathsf{st}^x}$, and so on, until we get that

$$\overline{\mathsf{ty}^x} \vdash^{\mathcal{D}}_{\mathsf{M}_{n+1}} \overline{\mathsf{st}^x}.$$

Since M' only replaces the body of $\mathsf{m}!\overline{\mathsf{T}}$, by case analysis similar to the above, we conclude

$$\overline{\mathsf{ty}^x} \vdash^{\mathcal{D}}_{\mathsf{M}'} \overline{\mathsf{st}^x},$$

which *proves (3)* for C-Instance. Proceeding with similar reasoning, we can chain facts (3) from all intermediate compilations and apply them to facts (1) of the induction, so we get that

$$\forall \mathsf{i}. \quad \overline{\mathsf{T}}\,\overline{\mathsf{ty}''} \vdash^{\mathcal{D}}_{\mathsf{M}'} \mathsf{st}'_{\mathsf{i}},$$

and thus by D-Seq,

$$\overline{\mathsf{T}}\,\overline{\mathsf{ty}''} \vdash^{\mathcal{D}}_{\mathsf{M}'} \overline{\mathsf{st}'}.$$

Since the compilation does not remove or alter any original methods, $\lfloor \mathsf{M}' \rfloor = \lfloor \mathsf{M} \rfloor$, and thus type inference $\mathcal{I}(\lfloor \mathsf{M}' \rfloor, \overline{\mathsf{T}}, \overline{\mathsf{st}})$ produces the same results as $\mathcal{I}(\lfloor \mathsf{M} \rfloor, \overline{\mathsf{T}}, \overline{\mathsf{st}})$. Therefore, the requirement of D-Table for the method instance $\langle \mathsf{m}!\overline{\mathsf{T}}\,\overline{\mathsf{st}'},\ \overline{\mathsf{ty}'} \rangle$ in M' is satisfied to *prove (2)*.
Finally, we can *prove (1)* for M' by D-Direct because $\mathsf{m}!\overline{\mathsf{T}} \in \mathsf{M}'$.

□

Putting it all together, I have shown that type-grounded methods compile to method instances without dynamically dispatched calls.

Proof. First, observe that the initial table M is maximally devirtualized because it does not contain any compiled method instances. Then, by induction on $\rightarrow^*$, we can show that the dynamic semantics preserves maximal devirtualization of the method table. Namely, by case analysis on one step of the dynamic semantics, we can see that the only rule that affects the method table is Disp; all other rules simply propagate the same maximally devirtualized table.

In the case of Disp, by Lem. 4.3, the new method table M' is also maximally devirtualized. Since every step of the dynamic semantics preserves maximal devirtualization of the method table, so does the multi-step relation $\rightarrow^*$.

If the execution gets to a direct call, the call must have been produced by the JIT compiler. As all compiled code is maximally de-

virtualized, by D-Direct, the method instance $\mathtt{m}!\overline{\mathsf{T}}$ exists, and, by D-Table, it is known to be maximally devirtualized. Finally, as we know by Lem. 4.1, any maximally devirtualized method instance that is type-grounded, is also fully devirtualized, and thus, it does not contain dispatched calls. □

4.3.2 Relationship between Stability and Groundedness

While it is type groundedness that enables devirtualization, the weaker property, type stability, is also important in practice. Namely, stability of calles is crucial for groundedness of the caller if the type inference algorithm analyzes function calls using nothing but type information about the arguments. Since types are the object of the analysis, I call such type inference *context-insensitive*: no information about the calling context other than types is available to the analysis of a function call. A more powerful type inference algorithm might be able to work around unstable methods in some cases, but even then, stability would be needed in general.

As an example, consider two type-unstable callees that return either an integer or a string depending on the argument value,

```
f(x) = (x > 0) ? 1 : "1"
```

and

```
g(x) = (rand() > x) ? 1 : "1"
```

and two calls, `f(y)` and `g(y)`, where `y` is equal to 0.5. If only type information about `y` is available to type inference, both `f(y)` and `g(y)` are deemed to have abstract return types. Therefore, the results of such function calls cannot be stored in concretely typed registers, which immediately makes the calling code ungrounded. If type inference were to analyze the value of `y` (not just its type), the result of `f(y)` could be stored in a concrete, integer-valued register, as only the first branch of `f` is known to execute. However, the other call, `g(y)`, would still have to be assigned to an abstract register. Thus, to enable groundedness and optimization of the client code regardless of the value of argument `x`, `f(x)` and `g(x)` need to be type-stable.

Note that stability is a necessary but not sufficient condition for groundedness of the client, as conditional branches may lead to imprecise types, like in the `f0` example from Fig. 2.3.

In what follows, I give a formal definition of context-insensitive type inference, and show that in that case, *reachable* callees of a grounded

method need to be type stable. The intuition behind the definition is that inferring the type of a function call in the context of a program gives us the same amount of information as inferring the return type of the callee independently. Technical complexities of the definition come from the fact that call instructions are conditional: if the callee is not reachable, its return type does not need to be compatible with the register of the calling instruction.

Definition 4.6 (Context-Insensitive Type Inference[4])**.** *A type inference algorithm* $\mathcal{I}$ *is* context-insensitive*, if, given any method table* M*, register typing* $\overline{\mathsf{ty}}$*, and instructions* $\overline{\mathsf{st}}$ *such that (1) type inference succeeds,*

$$\mathcal{I}(\mathsf{M}, \overline{\mathsf{ty}}, \overline{\mathsf{st}}) = \overline{\mathsf{ty}'},$$

(2) instructions contain a function call $\mathsf{m}(\overline{\%\mathsf{k}})$*,*

$$\overline{\mathsf{st}} = \ldots \mathsf{st}_\mathsf{i} \ldots \quad \wedge \quad \mathsf{st}_\mathsf{i} = \%\mathsf{i} \leftarrow \%\mathsf{j}\ ?\ \mathsf{m}(\overline{\%\mathsf{k}}) : \%\mathsf{l},$$

and (3) the call is reachable by running $\overline{\mathsf{st}}$ *in a compatible environment,*

$$\exists \mathsf{E}.\ \mathit{typeof}(\mathsf{E}) <: \overline{\mathsf{ty}} \quad \wedge \quad \mathsf{E}\ \overline{\mathsf{st}},\ \mathsf{M} \rightarrow^* (\mathsf{E}'\ \overline{\mathsf{st}'}) \cdot (\mathsf{E}''\ \mathsf{st}_\mathsf{i} \ldots),\ \mathsf{M}',$$

where $\overline{\mathsf{st}'}$ *is the body of the callee, that is* $\overline{\mathsf{st}'} = \mathit{body}(\mathcal{D}(\mathsf{M}, \mathsf{m}, \mathit{typeof}(\mathsf{E}')))$*, the inferred type of the calling instruction is no more precise than the inferred return type of the callee:*

$$\mathit{last}(\mathcal{I}(\mathsf{M}, \mathit{typeof}(\mathsf{E}'), \overline{\mathsf{st}'})) <: \mathsf{ty}'_\mathsf{i}.$$

Theorem 4.4 (Type Stability of Callees Reachable from Type-Grouned Code)**.** *If type inference is context-insensitive, then for all type-grounded sequences of instructions* $\overline{\mathsf{st}}$ *where*

$$\mathcal{I}(\mathsf{M}, \overline{\mathsf{T}}, \overline{\mathsf{st}}) = \overline{\mathsf{T}'},$$

any reachable callee in $\overline{\mathsf{st}}$ *is type stable for the types of its arguments.*

Proof. The property follows from the definition of context-insensitivity. Let st_i be a reachable call $\%\mathsf{i} \leftarrow \%\mathsf{j}\ ?\ \mathsf{m}(\overline{\%\mathsf{k}}) : \%\mathsf{l}$. Since $\overline{\mathsf{st}}$ is type-grounded, the inferred type of st_i is some concrete T'_i. Because by

4 It may look strange to apply the term *context-insensitive* to an interface of a static analysis rather than to a concrete analysis. But the requirement I define here limits possible implementations of type inference in a way that would not allow a context-sensitive analysis, and this is the reason I use the term. The reason I believe that context-sensitive analysis cannot satisfy the requirement is because the requirement does not allow to distinguish call sites of methods.

the definition of context-insensitivity, T'_i is no more precise than the inferred return type ty^m of the body,

$$last(\mathcal{I}(\mathsf{M}, typeof(\mathsf{E}'), \overline{\mathsf{st}'})) = \mathsf{ty}^m <: \mathsf{T}'_i,$$

and concrete types do not have subtypes other than themselves, $\mathsf{ty}^m = \mathsf{T}'_i$. Thus, according to the definition of type stability, the method m is type stable for $typeof(\mathsf{E}')$. □

4.4 CORRECTNESS OF COMPILATION

In this section, I prove that evaluating a program with just-in-time compilation yields the same result as if no compilation occurred. I use two instantiations of the Jules semantics, one (written $\rightarrow_{\mathcal{D}}$) where *jit* is the identity, and the other (written $\rightarrow_{\mathsf{JIT}}$) where *jit* is defined as before. The proof strategy is to

1. define the optimization relation $\triangleright$ which relates original and optimized code (Fig. 4.7),
2. show that compilation from Fig. 4.6 does produce optimized code (Thm. 4.7),
3. show that evaluating a program with $\rightarrow_{\mathsf{JIT}}$ or $\rightarrow_{\mathcal{D}}$ gives the same result (Thm. 4.10).

The main Thm. 4.10 is a corollary of bisimulation between $\rightarrow_{\mathcal{D}}$ and $\rightarrow_{\mathsf{JIT}}$ (Lem. 4.9). The bisimulation lemma uses Thm. 4.7 but otherwise, it has a proof similar to the proof of bisimulation between original and optimized code both running with $\rightarrow_{\mathcal{D}}$ (Lem. 4.5). A corollary of the latter bisimulation lemma, Thm. 4.6, shows the correctness of *jit* as an ahead-of-time compilation strategy.

First, Fig. 4.7 defines optimization relations for instructions, method tables, stacks, and configurations. According to the definition of $\mathsf{M} \triangleright \mathsf{M}'$, method table M' optimizes M if (1) it has all the same original methods[5], and (2) bodies of compiled method instances optimize the original methods using more specific type information available to the instance. These requirements guarantee that dispatching in an original and optimized method tables will always resolve to related methods. Optimization of instructions only allows for replacing dynamically

5 $\lfloor \mathsf{M}' \rfloor$ denotes the method table containing all original methods of M' without compiled instances.

$\boxed{\overline{\mathsf{ty}} \vdash_{\mathsf{M},\mathsf{M}'} \mathsf{st} \triangleright \mathsf{st}'}$ $\qquad$ $\boxed{\overline{\mathsf{ty}} \vdash_{\mathsf{M},\mathsf{M}'} \overline{\mathsf{st}} \triangleright \overline{\mathsf{st}'}}$

OI-Refl

$$\frac{}{\overline{\mathsf{ty}} \vdash_{\mathsf{M},\mathsf{M}'} \mathsf{st} \triangleright \mathsf{st}}$$

OI-Direct

$$\frac{\mathsf{st} = \%\mathsf{i} \leftarrow \%\mathsf{j}\ ?\ \mathsf{m}(\overline{\%\mathsf{k}}) : \%\mathsf{l} \qquad \overline{\mathsf{T}} = \overline{\mathsf{ty}}[\overline{\mathsf{k}}] \qquad \mathit{signtr}(\mathcal{D}(\mathsf{M},\mathsf{m},\overline{\mathsf{T}})) = \mathit{o\text{-}signtr}(\mathsf{M}'[\mathsf{m}!\overline{\mathsf{T}}]) \qquad \mathsf{st}' = \%\mathsf{i} \leftarrow \%\mathsf{j}\ ?\ \mathsf{m}!\overline{\mathsf{T}}(\overline{\%\mathsf{k}}) : \%\mathsf{l}}{\overline{\mathsf{ty}} \vdash_{\mathsf{M},\mathsf{M}'} \mathsf{st} \triangleright \mathsf{st}'}$$

OI-Seq

$$\frac{\overline{\mathsf{ty}} \vdash_{\mathsf{M},\mathsf{M}'} \mathsf{st}_0 \triangleright \mathsf{st}'_0 \quad \cdots \quad \overline{\mathsf{ty}} \vdash_{\mathsf{M},\mathsf{M}'} \mathsf{st}_n \triangleright \mathsf{st}'_n}{\overline{\mathsf{ty}} \vdash_{\mathsf{M},\mathsf{M}'} \overline{\mathsf{st}} \triangleright \overline{\mathsf{st}'}}$$

$\boxed{\mathsf{M} \triangleright \mathsf{M}'}$

O-Table

$$\frac{\mathsf{M} = \lfloor \mathsf{M}' \rfloor \qquad \forall \langle \mathsf{m}!\overline{\mathsf{T}}\,\overline{\mathsf{st}'},\ \overline{\mathsf{ty}} \rangle, \langle \mathsf{m}!\overline{\mathsf{ty}}\,\overline{\mathsf{st}},\ \overline{\mathsf{ty}} \rangle \in \mathsf{M}',\ \text{s.t. } \overline{\mathsf{T}} \neq \overline{\mathsf{ty}},\ \overline{\mathsf{st}'} \neq \epsilon.\ \overline{\mathsf{ty}'} = \mathcal{I}(\mathsf{M},\overline{\mathsf{T}},\overline{\mathsf{st}}) \qquad \overline{\mathsf{T}}\,\overline{\mathsf{ty}'} \vdash_{\mathsf{M},\mathsf{M}'} \overline{\mathsf{st}} \triangleright \overline{\mathsf{st}'}}{\mathsf{M} \triangleright \mathsf{M}'}$$

$$\Delta \quad ::= \quad \epsilon \mid \overline{\mathsf{ty}} \cdot \Delta' \quad \text{stack typing}$$

$\boxed{\Delta \vdash_{\mathsf{M},\mathsf{M}'} \mathsf{F} \triangleright \mathsf{F}'}$

O-StackEmpty

$$\frac{}{\epsilon \vdash_{\mathsf{M},\mathsf{M}'} \epsilon \triangleright \epsilon}$$

O-Stack

$$\frac{\overline{\mathsf{ty}} \vdash_{\mathsf{M},\mathsf{M}'} \overline{\mathsf{st}} \triangleright \overline{\mathsf{st}'} \qquad \Delta \vdash_{\mathsf{M},\mathsf{M}'} \mathsf{F} \triangleright \mathsf{F}'}{\overline{\mathsf{ty}} \cdot \Delta \vdash_{\mathsf{M},\mathsf{M}'} (\mathsf{E}\,\overline{\mathsf{st}}) \cdot \mathsf{F} \triangleright (\mathsf{E}\,\overline{\mathsf{st}'}) \cdot \mathsf{F}'}$$

$\boxed{\vdash^{\mathcal{I}}_{\mathsf{M}} \mathsf{F} <: \Delta}$

I-StackEmpty

$$\frac{}{\vdash^{\mathcal{I}}_{\mathsf{M}} \epsilon <: \epsilon}$$

I-Stack

$$\frac{\exists \mathsf{m}!\overline{\mathsf{ty}'}\,\overline{\mathsf{st}_b} \in \mathsf{M}. \qquad \mathsf{E} = \mathsf{E}'\mathsf{E}'' \qquad \mathsf{F} \neq \epsilon \implies \mathsf{F}, \mathsf{M} \to_{\mathcal{D}} \mathsf{E}'\,\overline{\mathsf{st}_b} \cdot \mathsf{F}, \mathsf{M} \qquad \mathsf{E}'\,\overline{\mathsf{st}_b} \cdot \mathsf{F}, \mathsf{M} \to^*_{\mathcal{D}} \mathsf{E}\,\overline{\mathsf{st}} \cdot \mathsf{F}, \mathsf{M} \qquad \overline{\mathsf{T}} = \mathit{typeof}(\mathsf{E}') \qquad \overline{\mathsf{ty}''} = \mathcal{I}(\mathsf{M},\overline{\mathsf{T}},\overline{\mathsf{st}_b}) \qquad \overline{\mathsf{T}} <: \overline{\mathsf{ty}'} \qquad \overline{\mathsf{T}}\,\overline{\mathsf{ty}''} <: \overline{\mathsf{ty}} \qquad \vdash^{\mathcal{I}}_{\mathsf{M}} \mathsf{F} <: \Delta}{\vdash^{\mathcal{I}}_{\mathsf{M}} (\mathsf{E}\,\overline{\mathsf{st}}) \cdot \mathsf{F} <: \overline{\mathsf{ty}} \cdot \Delta}$$

$\boxed{\mathsf{F}, \mathsf{M} \triangleright \mathsf{F}', \mathsf{M}' <: \Delta}$

O-Config

$$\frac{\vdash^{\mathcal{I}}_{\mathsf{M}} \mathsf{F} <: \Delta \qquad \Delta \vdash_{\mathsf{M},\mathsf{M}'} \mathsf{F} \triangleright \mathsf{F}' \qquad \mathsf{M} \triangleright \mathsf{M}'}{\mathsf{F}, \mathsf{M} \triangleright \mathsf{F}', \mathsf{M}' <: \Delta}$$

Figure 4.7: Optimization relation for instructions, method tables, stacks, and configurations

dispatched calls with direct calls in the optimized table. Optimization of stacks ensures that for all frames, instructions are optimized accordingly, and requires all value environments to coincide. The first premise of configuration optimization O-Config guarantees that the original configuration F, M is obtained by calling methods from the original method table M, and bodies of those methods are amenable to type inference. Based on the results of type inference, stacks F, F' need to be related in method tables M, M', and the method tables themselves also need to be related.

As I show below, when run with the dispatch semantics, related configurations are guaranteed to run in lock-step and produce the same result.

Lemma 4.5 (Bisimulation of Related Configurations)**.** *For any well-formed method tables* M *and* M' *(i.e.* $WF(M), WF(M')$ *according to Def. 4.3) where* M' *does not have stubs (i.e. all method bodies* $\neq \epsilon$*), any stacks* F_1, F_1'*, and stack typing* Δ_1 *that relates the configurations,* $F_1, M \triangleright F_1', M' <: \Delta_1$*, the following holds:*

1. *Forward direction:*

$$\begin{gathered} F_1, M \to_{\mathcal{D}} F_2, M \\ \Longrightarrow \\ \exists F_2', \Delta_1'. \quad F_1', M' \to_{\mathcal{D}} F_2', M' \ \wedge \ F_2, M \triangleright F_2', M' <: \Delta_1'. \end{gathered}$$

2. *Backward direction:*

$$\begin{gathered} F_1', M' \to_{\mathcal{D}} F_2', M' \\ \Longrightarrow \\ \exists F_2, \Delta_1'. \quad F_1, M \to_{\mathcal{D}} F_2, M \ \wedge \ F_2, M \triangleright F_2', M' <: \Delta_1'. \end{gathered}$$

Proof. For both directions, the proof goes by case analysis on the optimization relation for configurations and case analysis on the step of execution. By analyzing the derivation of the optimization relation $F_1, M \triangleright F_2, M' <: \Delta_1$, we have three assumptions, HS, HM, and HΔ:

$$\frac{\begin{array}{lll} \text{HI} & \text{HS} & \text{HM} \\ \vdash^{\mathcal{J}}_{M} F_1 <: \Delta_1 & \Delta_1 \vdash_{M,M'} F_1 \triangleright F_1' & M \triangleright M' \end{array}}{F_1, M \triangleright F_1', M' <: \Delta_1} \text{O-Config}$$

The assumptions will be referenced in the proof below.

(1) Forward direction.

To make a step, F_1 has to have at least one stack frame, so HS has the following form:

$$\overline{ty} \cdot \Delta \vdash_{M,M'} (E\ \overline{st_p}) \cdot F \triangleright (E\ \overline{st'_p}) \cdot F',$$

where $F_1 = (E\ \overline{st_p}) \cdot F$, $F'_1 = (E\ \overline{st'_p}) \cdot F'$, and $\Delta_1 = \overline{ty} \cdot \Delta$. Let us analyze the top frame.

(a) If the sequence of instructions is empty, i.e. $\overline{st_p} = \epsilon$, the only possible step for F_1, M is by RET. Therefore, $F = (E'\ st\ \overline{st}) \cdot F_r$ and $F_2 = (E' + last(E)\ \overline{st}) \cdot F_r$. As $\overline{ty} \vdash_{M,M'} \overline{st_p} \triangleright \overline{st'_p}$, it has to be that $\overline{st'_p} = \epsilon$. By analyzing the last premise of HS, $\Delta \vdash_{M,M'} F \triangleright F'$, we know that $F' = (E'\ st'\ \overline{st'}) \cdot F'_r$, and thus F'_1, M' can step by RET accordingly:

$$(E\ \epsilon) \cdot (E' + last(E)\ \overline{st'}) \cdot F'_r, M' \to_{\mathcal{D}} (E' + last(E)\ \overline{st'}) \cdot F'_r, M'.$$

By HI and the last premise of HI, i.e. $\vdash^{\mathcal{J}}_{M} (E'\ st\ \overline{st}) \cdot F_r <: \overline{ty'} \cdot \Delta_r$ where $\Delta = \overline{ty'} \cdot \Delta_r$, combined with $F, M \to^*_{\mathcal{D}} F_1, M \to_{\mathcal{D}} (E' + last(E)\ \overline{st}) \cdot F_r, M$, we can conclude HI′:

$$\vdash^{\mathcal{J}}_{M} (E' + last(E)\ \overline{st}) \cdot F_r <: \overline{ty'} \cdot \Delta_r.$$

By analyzing the derivation O-STACK of $\Delta \vdash_{M,M'} F \triangleright F'$, we know that $\Delta_r \vdash_{M,M'} F_r \triangleright F'_r$ and $\overline{ty'} \vdash_{M,M'} E'\ st\ \overline{st} \triangleright E'\ st'\ \overline{st'}$. It is easy to see that $\overline{ty'} \vdash_{M,M'} E' + last(E)\ \overline{st} \triangleright E' + last(E)\ \overline{st'}$ also holds by OI-SEQ, and thus we can conclude HS′:

$$\overline{ty'} \cdot \Delta_r \vdash_{M,M'} (E' + last(E)\ \overline{st}) \cdot F_r \triangleright (E' + last(E)\ \overline{st'}) \cdot F'_r.$$

Putting it all together, by HI′, HS′, and HM, we get $F_2, M \triangleright F'_2, M' <: \Delta$, i.e.:

$$(E' + last(E)\ \overline{st}) \cdot F_r, M \triangleright (E' + last(E)\ \overline{st'}) \cdot F'_r, M' <: \overline{ty'} \cdot \Delta_r.$$

(b) If the top frame contains a non-empty sequence of instructions, i.e. $\overline{st_p} = st\ \overline{st}$, HS has the following form:

$$(\overline{ty}^{E} ty\ \overline{ty}) \cdot \Delta \vdash_{M,M'} (E\ st\ \overline{st}) \cdot F \triangleright (E\ st'\ \overline{st'}) \cdot F',$$

where $F_1 = (E\ st\ \overline{st}) \cdot F$, $F'_1 = (E\ st'\ \overline{st'}) \cdot F'$, and $\Delta_1 = (\overline{ty}^{E} ty\ \overline{ty}) \cdot \Delta$. By analyzing the derivation O-STACK of HS, we get two assumptions, HOPT for optimization of the top-frame instructions, $\overline{ty}^{E} ty\ \overline{ty} \vdash_{M,M'} st\ \overline{st} \triangleright st'\ \overline{st'}$, and HS′ for optimization of the residual stack, $\Delta \vdash_{M,M'} F \triangleright F'$. The first premise of HOPT, HOPT_0, indicates optimization for

the first instruction, $\overline{\mathsf{ty}}^{\mathsf{E}}\mathsf{ty}\,\overline{\mathsf{ty}} \vdash_{\mathsf{M},\mathsf{M}'} \mathsf{st} \triangleright \mathsf{st}'$. By case analysis on the step $\mathsf{F}_1, \mathsf{M} \to_{\mathcal{D}} \mathsf{F}_2, \mathsf{M}$, we can always find a corresponding step for $\mathsf{F}_1', \mathsf{M}'$:

- Case PRIM. Here, $\mathsf{st} = \%\mathsf{i} \leftarrow \mathsf{p}$, and the configuration steps as:

  $$(\mathsf{E}\ \mathsf{st}\,\overline{\mathsf{st}}) \cdot \mathsf{F}, \mathsf{M} \to_{\mathcal{D}} (\mathsf{E}{+}\mathsf{p}\ \overline{\mathsf{st}}) \cdot \mathsf{F}, \mathsf{M}.$$

  By analyzing HOPT_0, we can see that the first instruction $\mathsf{st}' = \mathsf{st}$ by OI-REFL, and thus the optimized configuration $\mathsf{F}_1', \mathsf{M}'$ has to step by PRIM:

  $$(\mathsf{E}\ \mathsf{st}\,\overline{\mathsf{st}'}) \cdot \mathsf{F}', \mathsf{M}' \to_{\mathcal{D}} (\mathsf{E}{+}\mathsf{p}\ \overline{\mathsf{st}'}) \cdot \mathsf{F}', \mathsf{M}'.$$

  By analyzing HI and recombining its premises with the PRIM step, we get:

  $$\vdash^{\mathcal{I}}_{\mathsf{M}} (\mathsf{E}{+}\mathsf{p}\ \overline{\mathsf{st}}) \cdot \mathsf{F} <: (\overline{\mathsf{ty}}^{\mathsf{E}}\mathsf{ty}\,\overline{\mathsf{ty}}) \cdot \Delta.$$

  Since the rest of the instructions $\overline{\mathsf{st}}, \overline{\mathsf{st}'}$ and stacks F, F' did not change, putting it all together, by O-STACK we get:

  $$(\overline{\mathsf{ty}}^{\mathsf{E}}\mathsf{ty}\,\overline{\mathsf{ty}}) \cdot \Delta \vdash_{\mathsf{M},\mathsf{M}'} (\mathsf{E}{+}\mathsf{p}\ \overline{\mathsf{st}}) \cdot \mathsf{F} \triangleright (\mathsf{E}{+}\mathsf{p}\ \overline{\mathsf{st}'}) \cdot \mathsf{F}',$$

  and thus by O-CONFIG, we have the desired conclusion:

  $$(\mathsf{E}{+}\mathsf{p}\ \overline{\mathsf{st}}) \cdot \mathsf{F}, \mathsf{M} \triangleright (\mathsf{E}{+}\mathsf{p}\ \overline{\mathsf{st}'}) \cdot \mathsf{F}', \mathsf{M}' <: (\overline{\mathsf{ty}}^{\mathsf{E}}\mathsf{ty}\,\overline{\mathsf{ty}}) \cdot \Delta.$$

- Cases REG, NEW, FIELD, FALSE1, and FALSE2 proceed similarly to PRIM.

- Case DISP. Here, $\mathsf{st} = \%\mathsf{i} \leftarrow \%\mathsf{j}\ ?\ \mathsf{m}(\overline{\%\mathsf{k}}) : \%\mathsf{l}$, and the configuration steps as:

  $$(\mathsf{E}\ \mathsf{st}\,\overline{\mathsf{st}}) \cdot \mathsf{F}, \mathsf{M} \to_{\mathcal{D}} (\mathsf{E}'\ \overline{\mathsf{st}_\mathsf{b}}) \cdot (\mathsf{E}\ \mathsf{st}\,\overline{\mathsf{st}}) \cdot \mathsf{F}, \mathsf{M},$$

  where $\mathsf{E}' = \mathsf{E}[\overline{\mathsf{k}}]$, $\overline{\mathsf{T}} = \mathit{typeof}(\mathsf{E}')$, and $\overline{\mathsf{st}_\mathsf{b}} = \mathit{body}(\mathcal{D}(\mathsf{M}, \mathsf{m}, \overline{\mathsf{T}}))$. Note that because F_1, M does make a step by assumption, we know that dispatch is defined for the given arguments. Let's denote the dispatch target in M as $\langle \mathsf{m}!\overline{\mathsf{ty}_\mathsf{o}}\ \overline{\mathsf{st}_\mathsf{b}},\ \overline{\mathsf{ty}_\mathsf{o}}\rangle$. By the definition of dispatch, we know that $\overline{\mathsf{T}} <: \overline{\mathsf{ty}_\mathsf{o}}$. Therefore, by well-formedness of M and monotonicity of $\mathcal{I}$, type inference suc-

ceeds $\mathcal{I}(M, \overline{T}, \overline{st_b}) = \overline{ty'}$ (M contains only original methods, so $\lfloor M \rfloor = M$). Thus, we have:

$$\vdash^{\mathcal{I}}_{M} \ (E'\ \overline{st_b}) \cdot (E\ st\ \overline{st}) \cdot F <: (\overline{T}\ \overline{ty'}) \cdot (\overline{ty}^{E} ty\ \overline{ty}) \cdot \Delta,$$

where $\Delta'_1 = (\overline{T}\ \overline{ty'}) \cdot (\overline{ty}^{E} ty\ \overline{ty}) \cdot \Delta$. Next, let us consider F'_1. There are two possibilities for st'.

1. If HOPT_0 is built with OI-REFL, then $F'_1 = (E\ st\ \overline{st'}) \cdot F'$ where the first instruction is exactly the same dispatch call, i.e. $st' = st = \%i \leftarrow \%j\ ?\ m(\overline{\%k})\ :\ \%l$. The configuration $(E\ st\ \overline{st'}) \cdot F', M'$ could either step by DISP or err if $\mathcal{D}(M', m, \overline{T})$ is undefined. Let us inspect the possibility of the latter. By HM, M' optimizes M, so M' contains the same original method $m!\overline{ty_o}$. Thus, dispatch cannot fail due to the lack of applicable methods. As M' is well-formed, we also know that if there is a specialized method instance, it is for the best original method and ambiguity is not possible. Thus, $\mathcal{D}(M', m, \overline{T})$ succeeds, and F'_1, M' steps by DISP:

   $$(E\ st\ \overline{st'}) \cdot F', M' \rightarrow_{\mathcal{D}} (E'\ \overline{st'_b}) \cdot (E\ st\ \overline{st'}) \cdot F', M'.$$

   There are two possibilities for $\overline{st'_b}$. (a) If M' does not contain a specialization, then $\overline{st'_b} = \overline{st_b}$. By reflexivity of the optimization relation, we then have:

   $$\overline{T}\ \overline{ty'} \vdash_{M,M'} \overline{st_b} \triangleright \overline{st_b}.$$

   (b) If M' contains a specialized method instance $\langle m!\overline{T}\ \overline{st'_b}, \overline{ty_o} \rangle$, then $\overline{st'_b} \neq \epsilon$ by the assumption on M', and thus the desired relation is guaranteed by HM:

   $$\overline{T}\ \overline{ty'} \vdash_{M,M'} \overline{st_b} \triangleright \overline{st'_b}.$$

2. If HOPT_0 is built with OI-DIRECT, then st' is a direct call, i.e. $st' = \%i \leftarrow \%j\ ?\ m!\overline{T'}(\overline{\%k})\ :\ \%l$ where $\overline{T'} = \overline{ty}^{E}[\overline{k}]$, and method instance $m!\overline{T'}$ is in M'. Note that by HI and the soundness of type inference, we know that $\mathit{typeof}(E[\overline{k}]) <: \mathit{typeof}(\overline{ty}^{E}[\overline{k}])$, i.e. $\overline{T} <: \overline{T'}$. Since both are concrete types, and concrete types do not have subtypes other than themselves,

it has to be the case that $\overline{\mathsf{T}'} = \overline{\mathsf{T}}$ and $\mathsf{m}!\overline{\mathsf{T}}' = \mathsf{m}!\overline{\mathsf{T}}$. Thus, F_1', M' calls $\mathsf{m}!\overline{\mathsf{T}}$ and steps by Direct:

$$(\mathsf{E}\ \mathsf{st}'\ \overline{\mathsf{st}'}) \cdot \mathsf{F}',\ \mathsf{M}' \to_{\mathcal{D}} (\mathsf{E}'\ \overline{\mathsf{st}_b'}) \cdot (\mathsf{E}\ \mathsf{st}'\ \overline{\mathsf{st}'}) \cdot \mathsf{F}',\ \mathsf{M}'.$$

By OI-Direct, $\mathsf{m}!\overline{\mathsf{T}}\ \overline{\mathsf{st}_b'}$ specializes $\mathsf{m}!\overline{\mathsf{ty}_o}$. Since M' does not have stubs, we have by HM:

$$\overline{\mathsf{T}}\ \overline{\mathsf{ty}'} \vdash_{\mathsf{M},\mathsf{M}'} \overline{\mathsf{st}_b} \triangleright \overline{\mathsf{st}_b'}.$$

Because the new top frames $\mathsf{E}'\ \overline{\mathsf{st}_b}$ and $\mathsf{E}'\ \overline{\mathsf{st}_b'}$ are related in both cases, and the rest of the stacks did not change, we get that the entire stacks $\mathsf{F}_2, \mathsf{F}_2'$ are related:

$$(\overline{\mathsf{T}}\ \overline{\mathsf{ty}'}) \cdot (\overline{\mathsf{ty}}^{\mathsf{E}}\mathsf{ty}\ \overline{\mathsf{ty}}) \cdot \Delta \vdash_{\mathsf{M},\mathsf{M}'} (\mathsf{E}'\ \overline{\mathsf{st}_b}) \cdot (\mathsf{E}\ \mathsf{st}\ \overline{\mathsf{st}}) \cdot \mathsf{F} \triangleright (\mathsf{E}'\ \overline{\mathsf{st}_b'}) \cdot (\mathsf{E}\ \mathsf{st}'\ \overline{\mathsf{st}'}) \cdot \mathsf{F}'.$$

And thus we get the desired:

$$\mathsf{F}_2,\ \mathsf{M} \triangleright \mathsf{F}_2',\ \mathsf{M}' <:\ \Delta_1'.$$

- Case Direct. This case is not possible because F is obtained by running methods of M, and M consists of only original methods, which do not contain direct calls.

(2) Backward direction. This direction is similar to the forward direction. Because the structure of F_1' matches the structure of F_1, we can always find the corresponding step for F_1, M. The most interesting cases of

$$\mathsf{F}_1',\ \mathsf{M}' \to_{\mathcal{D}} \mathsf{F}_2',\ \mathsf{M}'$$

are Disp and Direct, and some details on these are provided below. In both cases, we have $\mathsf{F}_1 = (\mathsf{E}\ \mathsf{st}\ \overline{\mathsf{st}}) \cdot \mathsf{F}$, $\mathsf{F}_1' = (\mathsf{E}\ \mathsf{st}'\ \overline{\mathsf{st}'}) \cdot \mathsf{F}'$, and $\Delta_1 = (\overline{\mathsf{ty}}^{\mathsf{E}}\mathsf{ty}\ \overline{\mathsf{ty}}) \cdot \Delta$, and we know that configurations step by making a function call. As a result, we get:

$$(\mathsf{E}'\ \overline{\mathsf{st}_b}) \cdot (\mathsf{E}\ \mathsf{st}\ \overline{\mathsf{st}}) \cdot \mathsf{F},\ \mathsf{M} \triangleright (\mathsf{E}'\ \overline{\mathsf{st}_b'}) \cdot (\mathsf{E}\ \mathsf{st}'\ \overline{\mathsf{st}'}) \cdot \mathsf{F}',\ \mathsf{M}' <: (\overline{\mathsf{T}}\ \overline{\mathsf{ty}'}) \cdot (\overline{\mathsf{ty}}^{\mathsf{E}}\mathsf{ty}\ \overline{\mathsf{ty}}) \cdot \Delta,$$

where $\overline{\mathsf{ty}'} = \mathcal{I}(\mathsf{M}, \overline{\mathsf{T}}, \overline{\mathsf{st}_b})$ just like in the forward direction.

- Case Disp. Here, $\mathsf{st}' = \%\mathsf{i} \leftarrow \%\mathsf{j}\ ?\ \mathsf{m}(\overline{\%\mathsf{k}}) : \%\mathsf{l}$, and the configuration steps as:

$$(\mathsf{E}\ \mathsf{st}'\ \overline{\mathsf{st}'}) \cdot \mathsf{F}',\ \mathsf{M}' \to_{\mathcal{D}} (\mathsf{E}'\ \overline{\mathsf{st}_b'}) \cdot (\mathsf{E}\ \mathsf{st}'\ \overline{\mathsf{st}'}) \cdot \mathsf{F}',\ \mathsf{M}',$$

where $\mathsf{E}' = \mathsf{E}[\overline{\mathsf{k}}]$, $\overline{\mathsf{T}} = \mathit{typeof}(\mathsf{E}')$, and $\overline{\mathsf{st}'_\mathsf{b}} = \mathit{body}(\mathcal{D}(\mathsf{M}', \mathsf{m}, \overline{\mathsf{T}}))$. There are two options for $\overline{\mathsf{st}'_\mathsf{b}}$: it is the body of either the most applicable original method or its specialization in M'. Note that HOPT_0 can be built only with OI-REFL, so $\mathsf{st} = \mathsf{st}'$. Because by HM, $\mathsf{M} = \lfloor \mathsf{M}' \rfloor$, M has the same most applicable original method as M', and thus F_1, M steps by DISP. By reasoning similar to the forward direction, we get that the new top frames (as well as entire configurations) are related by the optimization relation.

- Case DIRECT. Here, $\mathsf{st}' = \%\mathsf{i} \leftarrow \%\mathsf{j}\ ?\ \mathsf{m}!\overline{\mathsf{T}}(\overline{\%\mathsf{k}})\ :\ \%\mathsf{l}$, and the argument about HI and the soundness of type inference applies similarly to the case (2) of DISP of the forward direction. As HOPT_0 can be built only by OI-DIRECT, we know $\mathsf{st} = \%\mathsf{i} \leftarrow \%\mathsf{j}\ ?\ \mathsf{m}(\overline{\%\mathsf{k}})\ :\ \%\mathsf{l}$. Furthermore, dispatch is defined in M by OI-DIRECT, and thus F_1, M steps by DISP. By reasoning similar to the forward direction, by HM, we know that the instructions in the new top frames are related, and thus the entire configurations are also related.

□

Theorem 4.6 (Correctness of Optimized Method Table)**.** *For any well-formed method tables* M *and* M' *where (1)* $\mathsf{M} \triangleright \mathsf{M}'$*, (2) table* M' *does not have stubs, and (3)* $\langle \mathsf{main}!\epsilon\ \overline{\mathsf{st}}, \epsilon \rangle \in \mathsf{M}$*,*

$$\epsilon\ \overline{\mathsf{st}}, \mathsf{M} \rightarrow^*_{\mathcal{D}} \mathsf{E}\ \epsilon, \mathsf{M} \quad \Longleftrightarrow \quad \epsilon\ \overline{\mathsf{st}}, \mathsf{M}' \rightarrow^*_{\mathcal{D}} \mathsf{E}\ \epsilon, \mathsf{M}',$$

i.e. program $\overline{\mathsf{st}}$ *runs to the same final value environment in both tables.*

Proof. This is a corollary of Lem. 4.5. By reflexivity of the optimization relation and well-formedness of M, we know:

$$\overline{\mathsf{st}}, \mathsf{M} \triangleright \overline{\mathsf{st}}, \mathsf{M}' <: \overline{\mathsf{ty}},$$

where $\overline{\mathsf{ty}} = \mathcal{I}(\mathsf{M}, \epsilon, \overline{\mathsf{st}})$. Thus, the bisimulation lemma is applicable: if one of the configurations can make a step, so does the other, and the step leads to a pair of related configurations. Since method tables did not change, the lemma can be applied again to these configurations, and so on. If the program terminates and does not err, both sides arrive to final configurations where environments are guaranteed to coincide by O-STACK. □

Next, I show that compilation as defined in Fig. 4.6 produces optimized code in an optimized method table according to Fig. 4.7.

Theorem 4.7 (Compilation Satisfies Optimization Relation)**.** *For any well-formed method tables* M *and* M'*, typing* $\overline{\mathsf{ty}}$*, and instruction* st*, such that*

$$\mathsf{M} \triangleright \mathsf{M}' \ \wedge \ \overline{\mathsf{ty}} \vdash \mathsf{st}, \mathsf{M}' \rightsquigarrow \mathsf{st}'', \mathsf{M}'',$$

it holds that:

1. $\overline{\mathsf{ty}} \vdash_{\mathsf{M},\mathsf{M}''} \mathsf{st} \triangleright \mathsf{st}''$,
2. $\mathsf{M} \triangleright \mathsf{M}''$,
3. *and the optimization relation on* M, M' *is preserved on* M, M''*:* $\forall \overline{\mathsf{ty}^x}, \mathsf{st}^x, \mathsf{st}^y. \quad \overline{\mathsf{ty}^x} \vdash_{\mathsf{M},\mathsf{M}'} \mathsf{st}^x \triangleright \mathsf{st}^y \implies \overline{\mathsf{ty}^x} \vdash_{\mathsf{M},\mathsf{M}''} \mathsf{st}^x \triangleright \mathsf{st}^y.$

Proof. By induction on the derivation of $\overline{\mathsf{ty}} \vdash \mathsf{st}, \mathsf{M}' \rightsquigarrow \mathsf{st}'', \mathsf{M}''$. Similar to the proof of Lem. 4.3 on maximal devirtualization, the most interesting case is C-Instance where compilation of additional instances happens.

- Cases C-NoDisp and C-Disp are straightforward: by reflexivity of the optimization relation, we immediately get $\overline{\mathsf{ty}} \vdash_{\mathsf{M},\mathsf{M}''} \mathsf{st} \triangleright \mathsf{st}$; by assumption $\mathsf{M} \triangleright \mathsf{M}'$ and $\mathsf{M}'' = \mathsf{M}'$, we also have $\mathsf{M} \triangleright \mathsf{M}''$; property (3) also holds trivially because of $\mathsf{M}'' = \mathsf{M}'$.
- Case C-Direct. Here, $\mathsf{st} = \%\mathsf{i} \leftarrow \%\mathsf{j}\ ?\ \mathsf{m}(\overline{\%\mathsf{k}}) : \%\mathsf{l}$ and $\mathsf{st}'' = \%\mathsf{i} \leftarrow \%\mathsf{j}\ ?\ \mathsf{m}!\overline{\mathsf{T}}(\overline{\%\mathsf{k}}) : \%\mathsf{l}$. Since $\mathsf{M}'' = \mathsf{M}'$, we have $\mathsf{M} \triangleright \mathsf{M}''$ by assumption, and (3) holds trivially.
  If $\mathsf{m}!\overline{\mathsf{T}}$ is an original method, then by $\lfloor \mathsf{M}' \rfloor = \mathsf{M}$ (because of $\mathsf{M} \triangleright \mathsf{M}'$) and the properties of dispatch, $\mathsf{m}!\overline{\mathsf{T}}$ has to be the method returned by $\mathcal{D}(\mathsf{M}, \mathsf{m}, \overline{\mathsf{T}})$. Therefore, $\mathit{signtr}(\mathcal{D}(\mathsf{M}, \mathsf{m}, \overline{\mathsf{T}})) = \overline{\mathsf{T}} = \mathit{signtr}(\mathsf{M}''[\mathsf{m}!\overline{\mathsf{T}}]) = \mathit{o\text{-}signtr}(\mathsf{M}''[\mathsf{m}!\overline{\mathsf{T}}])$.
  If $\mathsf{m}!\overline{\mathsf{T}}$ is a compiled instance in M', then by well-formedness of M', we know that its original method is the most applicable method in $\lfloor \mathsf{M}' \rfloor$. Since $\lfloor \mathsf{M}' \rfloor = \mathsf{M}$, we get $\mathit{signtr}(\mathcal{D}(\mathsf{M}, \mathsf{m}, \overline{\mathsf{T}})) = \mathit{o\text{-}signtr}(\mathsf{M}''[\mathsf{m}!\overline{\mathsf{T}}])$.
  Thus, for both original and compiled $\mathsf{m}!\overline{\mathsf{T}}$, we have $\overline{\mathsf{ty}} \vdash_{\mathsf{M},\mathsf{M}''} \mathsf{st} \triangleright \mathsf{st}''$ by OI-Direct.
- Case C-Instance. Here, $\mathsf{st} = \%\mathsf{i} \leftarrow \%\mathsf{j}\ ?\ \mathsf{m}(\overline{\%\mathsf{k}}) : \%\mathsf{l}$ and $\mathsf{st}'' = \%\mathsf{i} \leftarrow \%\mathsf{j}\ ?\ \mathsf{m}!\overline{\mathsf{T}}(\overline{\%\mathsf{k}}) : \%\mathsf{l}$ like in previous case, but M'' is obtained by compiling the body of method $\mathsf{m}!\overline{\mathsf{ty}'}\,\overline{\mathsf{st}}$ that is a dispatch target of $\mathcal{D}(\mathsf{M}', \mathsf{m}, \overline{\mathsf{T}})$ where $\overline{\mathsf{ty}'} \neq \overline{\mathsf{T}}$. Because of the latter condition, we know that $\mathsf{m}!\overline{\mathsf{ty}'}$ has to be an original method in M'. Since $\lfloor \mathsf{M}' \rfloor = \mathsf{M}$, $\mathsf{m}!\overline{\mathsf{ty}'}$ is also the most applicable method in M, so it has to be returned by $\mathcal{D}(\mathsf{M}, \mathsf{m}, \overline{\mathsf{T}})$.

Now, let us consider $M_0 = M' + \langle m!\overline{T}\epsilon, \overline{ty'}\rangle$. Since $m!\overline{T} \notin M'$ and M_0 only adds a new stub of a compiled instance to the well-formed M', we know $WF(M_0)$. Furthermore, because $M \triangleright M'$ and M_0 adds the stub without modifying anything else in M', it is easy to show that all instruction optimizations $\overline{ty^x} \vdash_{M,M'} st^x \triangleright st^y$ are preserved by M_0, that is $\overline{ty^x} \vdash_{M,M_0} st^x \triangleright st^y$, and thus $M \triangleright M_0$. Using the result of type inference on the original body in $\lfloor M' \rfloor$, i.e. $\overline{ty''} = \mathcal{I}(M, \overline{T}, \overline{st})$, we can apply the induction hypothesis to $\overline{T}\,\overline{ty''} \vdash st_0, M_0 \rightsquigarrow st'_0, M_1$, which gives us:

- $\overline{T}\,\overline{ty''} \vdash_{M,M_1} st_0 \triangleright st'_0$,
- $M \triangleright M_1$,
- $\forall \overline{ty^x}, st^x, st^y.\ \overline{ty^x} \vdash_{M,M_0} st^x \triangleright st^y \implies \overline{ty^x} \vdash_{M,M_1} st^x \triangleright st^y$.

The fact that $M \triangleright M_1$ and $WF(M_1)$ by Lem. 4.2, lets us apply the induction hypothesis to the next instruction st_1 of the method body $\overline{st}$, which produces:

- $\overline{T}\,\overline{ty''} \vdash_{M,M_2} st_1 \triangleright st'_1$,
- $M \triangleright M_2$,
- $\forall \overline{ty^x}, st^x, st^y.\ \overline{ty^x} \vdash_{M,M_1} st^x \triangleright st^y \implies \overline{ty^x} \vdash_{M,M_2} st^x \triangleright st^y$.

Now we can apply the latter to $\overline{T}\,\overline{ty''} \vdash_{M,M_1} st_0 \triangleright st'_0$, which gives us $\overline{T}\,\overline{ty''} \vdash_{M,M_2} st_0 \triangleright st'_0$.
Proceeding in this manner, we get:

$$\overline{T}\,\overline{ty''} \vdash_{M,M_{n+1}} \overline{st} \triangleright \overline{st'}$$

and

$$\forall \overline{ty^x}, st^x, st^y.\ \overline{ty^x} \vdash_{M,M'} st^x \triangleright st^y \implies \overline{ty^x} \vdash_{M,M_{n+1}} st^x \triangleright st^y.$$

Finally, let us look at M''. Its only difference from M_{n+1} is the non-stub body $\overline{st'}$ for $m!\overline{T}$. Thus, all $\overline{ty^x} \vdash_{M,M_{n+1}} st^x \triangleright st^y$ are trivially preserved by M'', which gives us

$$\forall \overline{ty^x}, st^x, st^y.\ \overline{ty^x} \vdash_{M,M'} st^x \triangleright st^y \implies \overline{ty^x} \vdash_{M,M''} st^x \triangleright st^y$$

and

$$\overline{T}\,\overline{ty''} \vdash_{M,M''} \overline{st} \triangleright \overline{st'}.$$

The latter lets us conclude $M \triangleright M''$.
Finally, because $\mathit{signtr}(\mathcal{D}(M, m, \overline{T})) = \overline{ty'}$ and $m!\overline{T}$ optimizes $m!\overline{ty'}$ in M'', we get

$$\mathit{signtr}(\mathcal{D}(M, m, \overline{T})) = \mathit{o\text{-}signtr}(M''[m!\overline{T}])$$

and conclude $\overline{ty} \vdash_{M,M''} st \triangleright st''$ by OI-Direct.

□

Finally, using an auxiliary lemma about preserving stub methods during compilation, we can show that the JIT-compilation semantics is equivalent to the dispatch semantics.

Lemma 4.8 (Preserving Stubs)**.** *For any well-formed method tables* M' *and* M''*, typing* $\overline{ty}$*, and instruction* st*, such that*

$$\overline{ty} \vdash st, M' \rightsquigarrow st'', M'',$$

it holds that

$$\{\overline{T} \mid m!\overline{T}\,\epsilon \in M'\} = \{\overline{T} \mid m!\overline{T}\,\epsilon \in M''\},$$

i.e. the set of stubbed method instances is preserved by a compilation step.

Proof. By induction on the derivation of $\overline{ty} \vdash st, M' \rightsquigarrow st'', M''$, similar to the proof of Lem. 4.2 on well formedness.

All cases except for C-Instance are trivial because the method table does not change. For C-Instance, let's denote the set of stubs in M' by S', that is:

$$S' = \{\overline{T} \mid m!\overline{T}\,\epsilon \in M'\}.$$

Since $m!\overline{T} \notin M'$ and $M_0 = M' + \langle m!\overline{T}\epsilon, \overline{ty'}\rangle$, we have $S_0 = S' \cup \{\overline{T}\}$. By applying the induction hypothesis to $\overline{T}\,\overline{ty''} \vdash st_0, M_0 \rightsquigarrow st_0', M_1$, we know that the set S_1 of stubs of M_1 is the same as S_0, and M_1 is well-formed by Lem. 4.2. Proceeding by applying the induction hypothesis to compilation of all st_i, we get that:

$$S_{n+1} = S_0 = S' \cup \{\overline{T}\}.$$

Finally, the only difference between M'' and M_{n+1} is that the stub for $m!\overline{T}$ is replaced by an actual method body. Therefore, we get the desired property:

$$S'' = S_{n+1} \setminus \{\overline{T}\} = S'.$$

□

Lemma 4.9 (Bisimulation of Related Configurations with Dispatch and JIT Semantics)**.** *For any well-formed method tables* M *and* M' *where* M' *does not have stubs, any frame stacks* F_1, F'_1, *and stack typing* Δ_1, *such that* $\mathsf{F}_1, \mathsf{M} \triangleright \mathsf{F}'_1, \mathsf{M}' <: \Delta_1$, *the following holds:*

1. *Forward direction:*

$$\mathsf{F}_1, \mathsf{M} \to_{\mathcal{D}} \mathsf{F}_2, \mathsf{M}$$
$$\Longrightarrow$$
$$\exists \mathsf{F}'_2, \mathsf{M}'', \Delta'_1. \quad \mathsf{F}'_1, \mathsf{M}' \to_{\mathsf{JIT}} \mathsf{F}'_2, \mathsf{M}'' \ \wedge \ \mathsf{F}_2, \mathsf{M} \triangleright \mathsf{F}'_2, \mathsf{M}'' <: \Delta'_1.$$

2. *Backward direction:*

$$\mathsf{F}'_1, \mathsf{M}' \to_{\mathsf{JIT}} \mathsf{F}'_2, \mathsf{M}''$$
$$\Longrightarrow$$
$$\exists \mathsf{F}_2, \Delta'_1. \quad \mathsf{F}_1, \mathsf{M} \to_{\mathcal{D}} \mathsf{F}_2, \mathsf{M} \ \wedge \ \mathsf{F}_2, \mathsf{M} \triangleright \mathsf{F}'_2, \mathsf{M}'' <: \Delta'_1.$$

Furthermore, M'' *is well-formed and does not have stubs.*

Proof. By case analysis on the derivation of optimization $\mathsf{F}_1, \mathsf{M} \triangleright \mathsf{F}'_1, \mathsf{M}' <: \Delta_1$ and case analysis on the step ($\to_{\mathcal{D}}$ for the forward and $\to_{\mathsf{JIT}}$ for the backward direction), similarly to the proof of bisimulation for the dispatch semantics (Lem. 4.5). The only difference appears in cases where $\mathsf{F}'_1, \mathsf{M}'$ steps by Disp: these are the only places where JIT compilation fires and M'' might be different from M'. As an example, we consider only the case of the forward direction where $\mathsf{F}_1, \mathsf{M} \to_{\mathcal{D}} \mathsf{F}_2, \mathsf{M}$ steps by Disp, and HOpt_0 is built by OI-Refl, reusing all the notation from Lem. 4.5. Thus, we have $\mathsf{st} = \mathsf{st}' = \%\mathsf{i} \leftarrow \%\mathsf{j}\ ?\ \mathsf{m}(\overline{\%\mathsf{k}}) : \%\mathsf{l}$, $\mathsf{F}_1 = (\mathsf{E}\ \mathsf{st}\ \overline{\mathsf{st}}) \cdot \mathsf{F}$, $\mathsf{F}'_1 = (\mathsf{E}\ \mathsf{st}\ \overline{\mathsf{st}'}) \cdot \mathsf{F}'$, $\Delta_1 = (\overline{\mathsf{ty}}^{\mathsf{E}} \mathsf{ty}\ \overline{\mathsf{ty}}) \cdot \Delta$, and F_1, M steps as:

$$(\mathsf{E}\ \mathsf{st}\ \overline{\mathsf{st}}) \cdot \mathsf{F}, \mathsf{M} \to_{\mathcal{D}} (\mathsf{E}'\ \overline{\mathsf{st}_\mathsf{b}}) \cdot (\mathsf{E}\ \mathsf{st}\ \overline{\mathsf{st}}) \cdot \mathsf{F}, \mathsf{M},$$

where $\mathsf{E}' = \mathsf{E}[\bar{\mathsf{k}}]$, $\overline{\mathsf{T}} = \mathit{typeof}(\mathsf{E}')$, and $\overline{\mathsf{st}_\mathsf{b}} = \mathit{body}(\mathcal{D}(\mathsf{M}, \mathsf{m}, \overline{\mathsf{T}}))$. $\mathsf{F}'_1, \mathsf{M}'$ can step only by Disp, which triggers JIT compilation. According to the definition of *jit* from Fig. 4.4, there are two possibilities: either $\mathsf{m}!\overline{\mathsf{T}}$ is already in M', in which case $\mathsf{M}'' = \mathsf{M}'$, or $\mathsf{m}!\overline{\mathsf{T}}$ is not in M', in which case the new method instance gets compiled and added to M''.

- In the former case, the proof proceeds exactly as in Lem. 4.5.

- In the latter case, by Thm. 4.7, we know that $\mathsf{M} \triangleright \mathsf{M}''$ and all instruction optimizations on M, M' are preserved for M, M''. Therefore, we know:

$$(\overline{\mathsf{ty}}^{\mathsf{E}}\mathsf{ty}\,\overline{\mathsf{ty}}) \cdot \Delta \vdash_{\mathsf{M},\mathsf{M}''} (\mathsf{E}\,\mathsf{st}\,\overline{\mathsf{st}}) \cdot \mathsf{F} \triangleright (\mathsf{E}\,\mathsf{st}\,\overline{\mathsf{st}'}) \cdot \mathsf{F}'.$$

  Furthermore, as M' is well-formed and does not have stubs by assumption, M'' is also well-formed by Lem. 4.2 and does not have stubs by Lem. 4.8. Reasoning similarly to Lem. 4.5, we can see that the body $\overline{\mathsf{st}_\mathsf{b}}$ of the original method returned by $\mathcal{D}(\mathsf{M}, \mathsf{m}, \overline{\mathsf{T}})$ optimizes to the body $\overline{\mathsf{st}'_\mathsf{b}}$ of the compiled instance $\mathsf{m}!\overline{\mathsf{T}}$. Thus, we get that $\mathsf{F}'_1, \mathsf{M}'$ steps by Disp,

$$(\mathsf{E}\,\mathsf{st}\,\overline{\mathsf{st}'}) \cdot \mathsf{F}', \mathsf{M}' \to_{\mathsf{JIT}} (\mathsf{E}'\,\overline{\mathsf{st}'_\mathsf{b}}) \cdot (\mathsf{E}\,\mathsf{st}\,\overline{\mathsf{st}'}) \cdot \mathsf{F}', \mathsf{M}'',$$

  the resulting configurations are related,

$$(\mathsf{E}'\,\overline{\mathsf{st}_\mathsf{b}}) \cdot (\mathsf{E}\,\mathsf{st}\,\overline{\mathsf{st}}) \cdot \mathsf{F}, \mathsf{M} \triangleright (\mathsf{E}'\,\overline{\mathsf{st}'_\mathsf{b}}) \cdot (\mathsf{E}\,\mathsf{st}'\,\overline{\mathsf{st}'}) \cdot \mathsf{F}', \mathsf{M}'' <: (\overline{\mathsf{T}}\,\overline{\mathsf{ty}'}) \cdot (\overline{\mathsf{ty}}^{\mathsf{E}}\mathsf{ty}\,\overline{\mathsf{ty}}) \cdot \Delta,$$

  and M'' has no stubs.

□

Theorem 4.10 (Correctness ot JIT)**.** *For any original well-formed method table* M *the following holds:*

$$\epsilon\,\overline{\mathsf{st}}, \mathsf{M} \to^*_{\mathcal{D}} \mathsf{E}\,\epsilon, \mathsf{M} \iff \epsilon\,\overline{\mathsf{st}}, \mathsf{M} \to^*_{\mathsf{JIT}} \mathsf{E}\,\epsilon, \mathsf{M}',$$

i.e. program $\overline{\mathsf{st}}$ *runs to the final environment* E *with the dispatch semantics* $\to_{\mathcal{D}}$ *if and only if it runs to the same environment with the JIT-compilation semantics* $\to_{\mathsf{JIT}}$ *.*

Proof. This is a corollary of Lem. 4.9. Be reflexivity of the optimization relation and well-formedness of M, we know: $\overline{\mathsf{st}}, \mathsf{M} \triangleright \overline{\mathsf{st}}, \mathsf{M} <: \overline{\mathsf{ty}}$, where $\overline{\mathsf{ty}} = \mathcal{I}(\mathsf{M}, \epsilon, \overline{\mathsf{st}})$. Since M does not have stubs, the bisimulation lemma is applicable: if one of the configurations can make a step, so does the other, and the step leads to a pair of related configurations such that the lemma can be applied again. If the program terminates and does not err, both sides arrive to final configurations where environments are guaranteed to coincide by O-Stack. □

5 APPROXIMATING TYPE STABILITY STATICALLY

Chapters 3 and 4 consider type stability as it relates to program *execution*: chapter 3 analyzes the state of Julia's virtual machine after running package test suites, and chapter 4 models a type-specializing just-in-time compiler that executes at run time. In this chapter, I set to approximate the property of type stability for arbitrary Julia code statically, without running the code in question. I focus only on type stability but not type groundedness because, as discussed in Chap. 4, groundedness is enabled by calling type-stable API.

5.1 INFERRING TYPE STABILITY VERSUS INFERRING TYPES

Explaining my approach to infer type stability requires a definition of stability. So far, I approached the definition twice: informally in 2.2.2 and formally in 4.3. The original informal definition does not take into account the distinction between concrete and abstract types: using this loophole, any code can be declared type stable because it is always possible to "predict the type of the output" as `Any`. Building upon the formal definition (that does acknowledge the distinction between abstract and concrete types) I provide another informal definition to explain intuitions behind this chapter.

Definition 5.1 (Type Stability, Informally)**.** *A Julia method is called* type stable *if, for any concrete type of the input, it is possible to infer a concrete type of the return value.*

A natural idea for inferring type stability in Julia would be to formulate it as a forward static analysis: being an abstract or concrete type is one bit of information that has a known value at the input (concrete) and should be propagated to the output, possibly changing on the way.

To test the static analysis idea, consider a positive example first: the identity function.

```
function id(x)
  x
end
```

It is straightforward to infer that, given any concrete input type, the return value is also concretely typed: the one bit of information carries over to the result in one step.

However, another example—the increment function—shows that the task quickly becomes unwieldy.

```
function inc(x)
  x + 1
end
```

Concreteness of the result returned by `inc` depends on concreteness of the result of the call to `+`. In turn, the property of the return type of `+` depends on which `+` method Julia will dispatch to at run time. There are about two hundred method implementations of `+` in the standard library alone, and packages add more. Some of those methods are type stable (e.g. `+(::Int64,::Int64)`), and some of them are not (e.g. `+(::Rational{Bool},::Rational{Bool})` [1]). Therefore, to infer the property of interest, in general, we need to predict which methods are selected at run time.

The `inc` example shows that inferring type stability of Julia code requires reasoning about multiple dynamic dispatch, which leads to reasoning about the *types* of intermediate values rather than only the concreteness bit. But if there was a tool for computing type information beforehand, a special-purpose analysis for type stability would not be needed: it suffices to ask the tool for the type of the return value and check if that type is concrete. This observation leads to the following conjecture:

Conjecture 5.1. *Inferring type stability of a Julia method statically is no easier than performing type inference of that method.*

1 The type `Rational{Bool}` represents rational numbers with boolean numerator and denominator. The type is rarely useful but is allowed as a consequence of the `Bool <: Number` subtyping and arithmetic operations on boolean values. In the early history of Julia, the authors decided to count `Bool` as a number type because it can be convenient in some applications.
The reason for the `+(::Rational{Bool},::Rational{Bool})` method to be type unstable is not important, but in a nutshell, Julia has made a questionable design decision about the return type of `+(::Bool,::Bool)`, which in the current implementation is `Int` (see discussion https://github.com/JuliaLang/julia/issues/19168), and when adding two rational numbers with boolean components, depending on the values of the summands, you get back either `Rational{Bool}` or `Rational{Int}`.

A complete type inference algorithm would allow for checking type stability of Julia code. But should type inference be implemented from scratch? There are two reasons to not go this way.

1. It is not clear that inferring types for source-level Julia code without changing anything in the language can yield a meaningful result (more on this see [Chung 2023]). For instance, in the `inc` example, a sound return type cannot be much better than `Any`.

2. Julia already has a built-in type inference engine, which was modeled as a black box in Chap. 4. This engine is used for code optimizations. Thus, analyzing type stability based on a custom type inference algorithm can produce results that diverge from Julia, misleading the users about potential optimizations. This would be of limited usage for Julia users.

5.2 AN ALGORITHM TO INFER TYPE STABILITY

5.2.1 High-Level Description

If our predictions for type stability are to align with the Julia implementation, the analysis should closely model Julia's run-time behavior, as described in Chap. 4. The type-specializing JIT-compiler from Chap. 4 makes optimization decisions based on *concrete input types* with the help of Julia's type inference engine. Therefore, the algorithm for predicting these decisions statically considers all (or as many as possible, see Sec. 5.3) allowed concrete input types of a method. Fig. 5.1 describes this algorithm at a high level.

Let us consider every step of the algorithm described on Fig. 5.1 and explain its meaning using an example. The list below also assigns the numbers to each step in the algorithm.

STEP 1 The input of the algorithm is a Julia method. Methods in Julia are represented by run-time objects of type `Method` and can be manipulated as all other objects (e.g. stored in collections, responding to field accesses, etc.).

For example, consider the `length` method from Julia's standard library. We can get the corresponding `Method` object using the standard `@which` macro applied to an application of the `length` method. This can be done in the Julia REPL (signified by the `julia>` prefix).

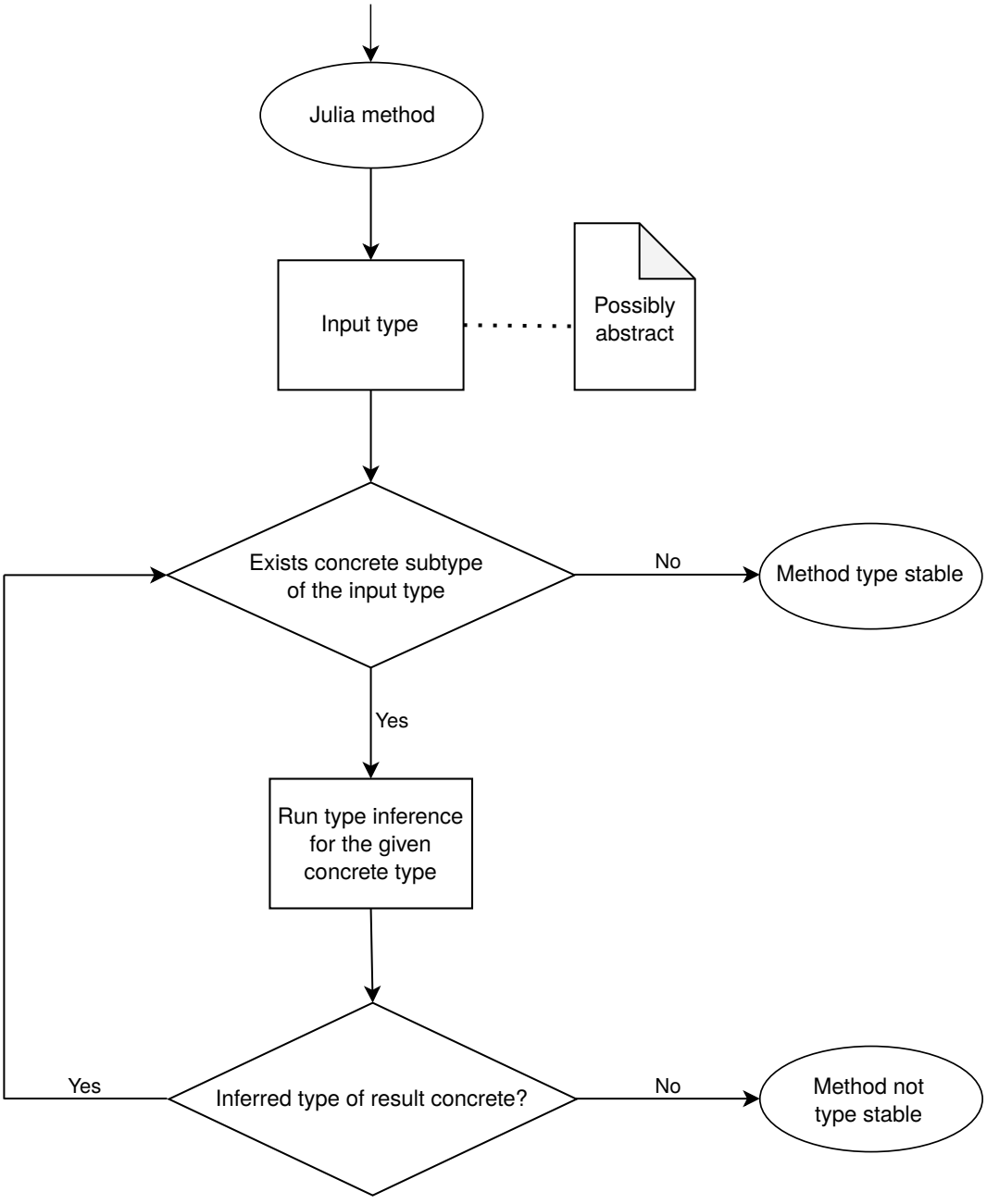

Figure 5.1: Inferring type stability of a Julia method

```
julia> @which length([1,2,3])
length(a::Array) in Base at array.jl:215
```

The output shows that the given `length`-call will dispatch to the method defined in the `Base` module (Julia-speak for the standard library). The output also shows the location of the method in the standard library and, most importantly for us, the signature of the method. In fact, what we see here is a pretty-printed representation of the `Method` object representing a particular Julia method.

STEP 2 The first task of the algorithm is to get the input type of the given method. This is possible through querying the `sig` field of the method object.

Building on the example above, we can get the signature of the `length` method as follows:

```
julia> m = @which length([1,2,3]);

julia> m.sig
Tuple{typeof(length), Array}
```

A signature of a method contains the special singleton function type (`typeof(...)`) as the first component, and the rest is (easy to convert to) the type of the input — an n-tuple. In this example, the type of the input is 1-tuple, consisting of the existential array type `Array{T,N} where T where N` abbreviated simply as `Array`[2].

STEP 3 The input type can be either concrete, which, in Julia, means that there can be no proper subtypes of that type, or abstract. In either case, the choice on the current step will enter the loop at least once, because for concrete input type, the check holds once trivially (e.g. there is exactly one concrete subtype of the concrete type `Int` — it is `Int` itself).

If the input type is abstract, we need a procedure enumerating all concrete subtypes of it. An implementation for this procedure is discussed below (Subsec. 5.2.2), but it suffices to treat is as a black box for now.

In the case of the `length` method, the input type, `Array`, is an existential type and hence abstract. Therefore, the enumeration procedure should have yielded a concrete subtype of `Array`. Assume that the concrete type is `Array{Float64,1}`.

2 A user can always look under the abbreviation using the `dump` function.

STEP 4 Running Julia's type inference for a given method and a given concrete input type is done by calling Julia's standard `code_typed` function. The only issue with the function is that it expects a function object as a part of the input, not a method object. But getting from a method to the corresponding function is possible using the signature field discussed above, and, in particular, the singleton function type contained in the first component of the `sig` field: accessing the single function object using the function type is possible via the `instance` field.

Running type inference for the `length` method and the concrete input type `Array{Float64,1}` could be done as shown on Fig. 5.2. The return value is an array of `CodeInfo` objects that represent the type-annotated method bodies of all methods that a call with the given input type could dispatch to (for a concrete input type and no ambiguities in method definitions, the resulting array always contains exactly one element). Method bodies are transformed into a lower-level intermediate representation, similar to the one discussed in Subsec. 2.2.1 and Chap. 4. In the running example, the method body contains a single call to an intrinsic Julia function that is known to return a value of type `Int64`.

```
julia> code_typed(m.sig.parameters[1].instance,
                  (Array{Float64, 1},),
                  optimize=false)
1-element Vector{Any}:
 CodeInfo(
1 - %1 = Base.arraylen(a)::Int64
+--      return %1
) => Int64
```

Figure 5.2: Running Julia's built-in type inferencer

STEP 5 Concreteness of the inferred return type is checked with the standard Julia `isconcretetype` predicate. In the running example, for the concrete input type `Array{Float64,1}`, the return type of `length` is inferred to be `Int64`, which is a concrete type. Following the second decision element on Fig. 5.1, we get back to the start of the loop and try another concrete subtype of the input type, if there is any.

There are two main auxiliary procedures that the algorithm relies on: enumeration of concrete subtypes of a given input type and type inference over a method with the given input type. The latter can be fully outsourced to Julia itself, while the former requires separate consideration.

5.2.2 Enumerating Concrete Subtypes

5.2.2.1 *Julia's `subtypes` method*

On **STEP 3** of the algorithm (Subsec. 5.2.1), we need to generate a concrete subtype of the input type. This task does not have a direct implementation in Julia's standard library. The closest counterpart that Julia provides is the `subtypes` method to query declared subtypes of a given nominal type. For example,

```
julia> subtypes(Signed)
6-element Vector{Any}:
 BigInt
 Int128
 Int16
 Int32
 Int64
 Int8
```

```
julia> subtypes(AbstractSet)
5-element Vector{Any}:
 Base.IdSet
 Base.KeySet
 BitSet
 Set
 Test.GenericSet
```

The challenge here is that Julia's subtype relation is richer than the nominal type hierarchy. For example, if a method declares the type of its input as

```
Union{AbstractSet, AbstractRange}
```

then the method can be called with an argument of type `Set{Int}`, and, therefore, it should be possible to discover such a type following the algorithm on Fig. 5.1. However, it is not possible with the `subtype` method alone: from `subtypes`' point of view, the union type above has no subtypes:

```
julia> subtypes(Union{AbstractSet, AbstractRange})
Type[]
```

(the query returns an empty array of elements of the type `Type`). Therefore, structural types such as unions, tuples, and existential types, need a special treatment.

The `subtypes` method does not cover the whole subtyping relation not only because it cannot work with structural types but also because it is limited to direct declared subtypes. For example, in the subtype chain `Int <: Signed <: Integer`, `subtypes` reports only `Int <: Signed` and `Signed <: Integer` but not `Int <: Integer`. Thus, we have to build the transitive closure manually.

In the following subsection, I show how to generalize the `subtypes` method to a method I call `direct_subtypes` that produces direct subtypes for a Julia type of any form. Then, the transitive closure mentioned above can be obtained by iterating `direct_subtypes` applications,

starting from the signature of a method we are checking for type stability.

After generating all subtypes of a given type, it is possible to filter for concrete subtypes using the built-in Julia predicate `isconcretetype`. This completes the task of enumerating concrete subtypes and the description of the algorithm as presented on Fig. 5.1.

5.2.2.2 *Enumerating Direct Subtypes*

My goal in this section is to define a general utility generating direct subtypes of a given type, which I call `direct_subtypes`. This is done by case analysis on the existing Julia kinds (i.e., types of types) and expressed with multiple methods of the `direct_subtypes` function. The implementation largely follows from the description of Julia's subtyping relation given in [Zappa Nardelli et al. 2018].

The `direct_subtypes` utility can be defined, for the most part, via multiple dispatch using Julia's kind system. The only special sort of types that does not have a dedicated kind is the tuple type. In particular, although tuple types fall under the `Datatype` kind, the standard `subtypes` method cannot help with tuple types as much as it can with user-defined datatypes. For example, both `Integer` and `Tuple{Integer,Integer}` are datatypes, but `subtypes` returns an empty array of types for the latter.

UNION TYPES Unions in Julia have the kind `Union` and a form of `Union{A,B,C}` with arbitrary number of arguments `A, B, C`. Direct subtypes of `Union{A,B,C}` is the following list of types: `[A,B,C]`.

EXISTENTIAL TYPES Existential types in Julia have the kind `UnionAll` and a form of `t where T`, e.g. `Vector{T} where T`, commonly abbreviated as `Vector`. Every `where`-bound type variable has associated upper and lower bounds with default values `Any` and `Union{}` (the top and bottom types), respectively. Direct subtypes of an existential type are all instantiations of its variable allowed by the bounds. In particular, we need to compute all subtypes of the upper bound, filter out those not satisfying the lower bound, and use the resulting set of types to instatiate the variable with.

For example, consider the `Vector{T} where T<:Signed` type. All subtypes (not only direct or concrete) of `Signed` are: `BigInt`, `Int128`, `Int16`, `Int32`, `Int64`, `Int8`. Therefore, all direct subtypes of the vector type in question are:

```
Vector{BigInt}
Vector{Int128}
Vector{Int16}
Vector{Int32}
Vector{Int64}
Vector{Int8}
```

DATATYPES In general, non-parametric nominal types defined using the `primitive` (e.g. `Int64`) or `struct` (e.g. `Pair`) qualifiers, can be processed using the `subtypes` method (see examples in the previous section).These also include fully-instantiated parametric types (e.g. `AbstractSet{Int}`).

TUPLE TYPES Tuples play a key role in method dispatch because method signatures and arguments are represented with tuple types. As the `subtypes` method cannot work with tuples (for any input tuple, it returns an empty array), tuple types are unfolded according to the following rule:

- direct subtypes of the 0-tuple type is an empty set;
- direct subtypes of a 1-tuple type are 1-tuples of the direct subtypes of its single type argument;
- direct subtypes of an n-tuple type are a Cartesian product of direct subtypes of its first component and direct subtypes of the last $n-1$ components.

  For example, for an abstract type `A` with exactly two declared subtypes `B` and `C`, the set of direct subtypes of `Tuple{A,A}` is

  ```
  Tuple{B,B}
  Tuple{B,C}
  Tuple{C,B}
  Tuple{C,C}
  ```

5.3 SEARCH SPACE AND APPROXIMATION

The algorithm as presented on Fig. 5.1 is an instance of a tree search algorithm: it traverses the subtype tree starting from a given node and tries to find a leaf on the tree that violates a given property. The nodes of the tree are discovered dynamically—using the `direct_subtypes` utility. With the grammar of types and subtype relation as rich as

Julia's, the search may diverge. In this section, I show how to control the search space of the algorithm.

5.3.1 The Problem: Underconstrained Types

There are several groups of abstract types that have "too many" subtypes to explore and, therefore, lead to issues with the subtype tree exploration; I call such types *underconstrained*. Most common groups of underconstrained types are as follows.

1. Nominal types declared close to the top of the subtype lattice have too many (even if finite) concrete subtypes to explore in a reasonable time. The most prominent example is the top type (`Any`) itself. The top type in Julia 1.8 standard library has 567 direct subtypes, most of which are abstract and, therefore, have a number of direct subtypes of their own, etc., so it is not feasible to track the whole tree during the search.

2. Existential types whose variable's upper bound is an underconstrained type. Especially large search space is produced by existentials with `Any` as the upper bound (*unbounded existentials*, for short). Note that out of the 567 direct subtypes of `Any`, quite a few types are unbounded existentials (e.g. `AbstractSet`, `AbstractArray`), which makes the enumeration of all subtypes of `Any` infeasible.

3. The combination of underconstrained existential types and parametric types produces infinite subtrees because any *recursive instantiation* of the parametric type under an underconstrained existential will still be a subtype of the existential. These cannot be explored in finite time, however long. For example, consider the `length` method from before: its input type is an existential type on top of the parametric `Array` type, which, among others, has the following direct concrete subtypes:

   - `Array{Int,1}`
   - `Array{Array{Int,1},1}`
   - `Array{Array{Array{Int,1},1},1}`
   - etc.

Too large (sometimes infinite) search space is the reason why, in general, the algorithm builds only an approximation of the answer to

a type stability inference query. In the rest of the section, I describe three concrete ways to control the search space to avoid looping and excessive search time.

5.3.2 Fuel

The simplest heuristic to limit the number of nodes to explore on the subtype tree is to put an arbitrary upper bound on that number—the idea commonly referred to as *fuel*. The amount of fuel may be determined empirically, for example, taking the maximum number of types explored during a successful type stability check in a corpus of code.

Fuel can be interpreted broadly. The approach currently used in my stability inference algorithm, is to limit the number of types discovered during the tree search. Alternatively, one could limit the number of concrete types explored, or the number of applications of parametric types during one search, etc. However, when picking the strategy to account for the fuel, caution is needed. For example, counting the number of concrete types, and, therefore, the number of actual checks, may seem more appealing, but, in general, the algorithm may diverge before getting to a single concrete type. An example of the issue may be found through any on the corner cases listed in Subsec. 5.3.1, but the issue is especially easy to see with the corner case 3, where a subset of concrete subtypes is produced by an infinite chain of recursive applications of a parametric type.

In the type stability inference algorithm, the fuel parameter is used primarily to detect cases where the search space explodes. This effectively allows for a constructive definition of underconstrained types described informally in Subsec. 5.3.1: if the tree generation for the given type runs out of fuel, the type is considered underconstrained, and the enumeration of subtypes is terminated. In this case, the algorithm can employ other techniques for approximating type stability, as described below.

5.3.3 Sampling Concrete Types

The essence of the issue with diverging search is that we start at a too high point in the subtype lattice and are unable to reach concrete types at the bottom of the lattice. A natural idea to overcome the issue is to start from low points in the lattice instead, and in particular—from

concrete types. Since the number of concrete types is infinite (see, for example, corner case 3 in Subsec. 5.3.1), in order to realize the idea, one needs to collect a sample of concrete types. The sample can then be stored as a *database* that is available at the type-stability inference time.

Regarding the idea of starting from the bottom of the lattice, there are two challenges we need to address.

1. *Which concrete types to put in the database?* We could use some fixed universe of Julia types, such as all types defined in Julia packages. This approach breaks in the presence of parametric types, which require a type argument to become a valid type. This brings back the issue described in corner case 3 of Subsec. 5.3.1.

   As a strategy for collecting concrete Julia types, I propose an approach based on the type-stability tracing framework discussed in Chap. 3. In particular, I run test suites of popular Julia packages and record which concrete types are used to instantiate methods. This way, the sample contains a variety of concrete types relevant to specific packages, including fully instantiated parametric types.

2. *At which moment start using the database?* One obvious idea is, when noticing that fuel runs out during enumeration, to terminate the search and start sampling as if the search was never started. In that case, the types coming from the database should be checked for being in the subtype relation with the method's input type (can be done using Julia's built-in `<:` predicate). The issue is: if the database does not contain (many) such subtypes, we still have the problem with not getting concrete types to check type stability against.

   In practice, the most frequent reason for diverging enumeration of types is the `Any` type — the supertype of all types, which is the default for unannotated method parameters and unbounded existentials. Using this observation, I propose to mold enumeration and sampling as follows. The algorithm switches from the former to the latter when it hits `Any`. In particular, when the enumeration procedure calls `direct_subtypes` on `Any`, the utility returns the types from the database (up to a certain bound if the database is too big). After that, the algorithm will proceed as usual without any other change. Eventually, either the enumeration procedure runs out of fuel, or it finishes with a

success. Overall, this approach seems to fit better with the idea to serve both, unannotated method parameters and unbounded existentials.

There is one technical obstacle to using a sample of types seen elsewhere in the current Julia session. Recovering types from other sessions requires an environment where the name of the type can be successfully resolved. Concretely, to recreate a necessary environment means installing some Julia packages. Note that the obstacle of recreating environments for types does not come up with the top-down search described earlier, because the `subtypes` method applied iteratively, as discussed in Subsec. 5.2.2, always only discovers only the types visible in the current session.

The obstacle can be solved if the database bears enough of the provenance information for every type. Such information makes using the type possible, but it depends whether recreating a necessary environment is desirable. For instance, if network is unavailable, installing extra Julia packages may not be possible. In such cases, the algorithm can filter and employ only the types defined in the currently loaded packages as well as the standard library, which is always available in Julia.

As an example of the sampling approach, I publish a database[3] with concrete Julia types used to instantiate methods during test suite runs in the 10 popular packages listed in Table 3.2. The database contains the necessary type provenance information and can be successfully loaded by the tool.

5.3.4 Type Inference With Abstract Types

So far, we assumed that to determine type stability of a method, it is necessary to run Julia's type inferencer on that method with all possible *concrete* input types. Indeed, this naturally follows from the definition of type stability defined formally in Chap. 4. However, to solve the problem of search space explosion, the assumption can be revised.

A usual definition of type stability (e.g. the formal one I give in Def. 4.4) has the form of an implication: if a condition (a type being concrete) applies to the input type, then the same condition applies to the type of the output. Clearly, relaxing the premise of the implication,

3 https://github.com/prl-julia/julia-type-stability-checker-data/blob/0ef57a6/types-database/types.csv

i.e. the part where we ought to provide a concrete input type, and making no assumption about the input type instead, leads to a stricter statement or, in other words, a subset of the Julia methods that would normally be considered type stable.

Relaxing the definition of type stability as described makes for an easier task and identifies a special group of methods I call *type-constant methods*. Those methods hold two independent properties:

- the type of the input does not impact the type of the outut,
- the output type can be inferred as concrete.

Type-constant methods are a proper subset of all type-stable methods.

The idea of type-constant methods allows another solution for the search space problem. In particular, before trying to solve the potentially hard problem of inferring type stability using concrete input types, the algorithm checks whether the given method is type constant with a single call to the type inferencer.

In order to decide the easier task, the algorithm still needs to run Julia's type inferencer, and for that, some input type should be provided. The obvious candidate for such a type is the one taken directly from the method signature. Such input type, even if abstract, also makes sure that Julia will dispatch the same method that is currently analyzed, and it is the most general (w.r.t. subtype lattice) such type.

Getting back to the running example in Subsec. 5.2.1 with the `length` method, let us call Julia's type inferencer, as shown on **STEP 4** of the algorithm (Fig. 5.2). But instead of the concrete type used in the example, we supply the most general type for that method (i.e. the type provided in the method signature) — the existential `Array` type:

```
julia> code_typed(m.sig.parameters[1].instance,
                  (Array,),
                  optimize=false)
1-element Vector{Any}:
 CodeInfo(
1 - %1 = Base.arraylen(a)::Int64
+--      return %1
) => Int64
```

Hence, the type inferencer is able to infer the concrete `Int64` type, which is what we expect for the output type of a `length` method.

5.4 PARAMETERS AND OUTCOMES OF THE ALGORITHM

After updates as described in the previous section, the algorithm receives two parameters: the amount of fuel to use to traverse the subtype lattice searching for concrete types, and whether to use a type database for sampling (and if so, the database itself as one more parameter).

Possible outcomes of the updated algorithm are as follows.

1. The method is type stable through one of the three possible options.
   - The method is type stable if running Juila's type inference with declared input types computes a concrete type of the output (Subsec. 5.3.4).
   - If the type database is not in use, then all known concrete types acceptable by the method were used to run Juila's type inferencer, and every time the result type computed was concrete.
   - If the type database is in use, then a subset of all known concrete types acceptable by the method were used to run Juila's type inferencer, and every time the result type computed was concrete. The subset is defined by the enumeration procedure, which fills in types from the database every time it hits `Any` (Subsec. 5.3.4).
2. The method is definitely not type stable and a counterexample is provided, i.e. a concrete type that makes the method return a value of type that cannot be inferred as concrete ahead of time.
3. Cannot decide type stability because the algorithm ran out of fuel while running the enumeration (no matter whether it was enhanced with a type database or not).

5.5 EVALUATION

In Chap. 3, I describe a dynamic type stability analysis of a corpus of open-source Julia packages. The analysis is based on executing test suites of the packages and inspecting the resulting method instances collected from the internal state of the virtual machine.

To evaluate the algorithm proposed in the present chapter, I run my implementation of the algorithm[4] to statically infer type stability of methods in the same 10 packages discussed in Chap. 3. The full list of packages is given in Table 3.2, which also serves as a reference point in the discussion below. It is not possible to directly compare the results because the two analyses capture different sets of methods, but some similarities are to be expected.

The numbers for methods grouped by the three possible outcomes (Sec. 5.4) are provided in the Appendix B, and here we present a graphical representation of those numbers —Fig. 5.3. The numbers are computed by running the tool implementing the algorithm without a type database (left column of every pair of columns on Fig. 5.3) and with the database (right column) for each of the 10 packages. Every package name on the figure has the number of methods inspected below it.

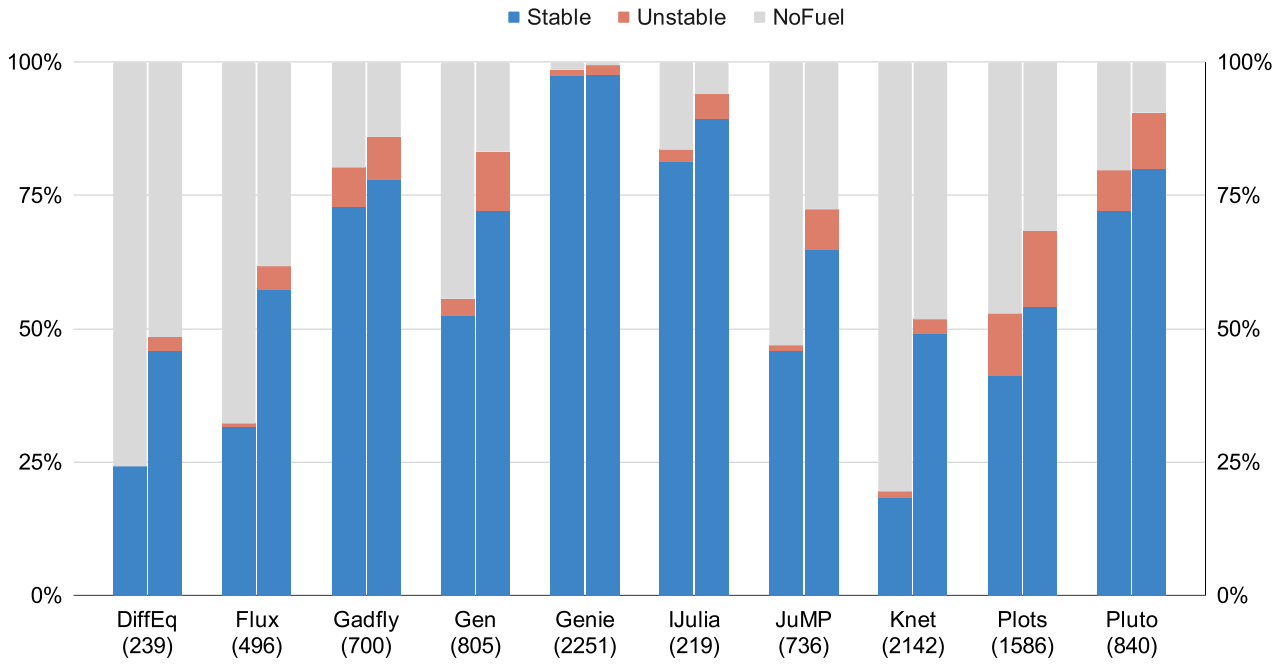

Figure 5.3: Inferring Type Stability for the 10 popular Julia packages: without (left bars) and with (right bars) sampling

On average over the 10 packages, the analysis identified 54% of methods as type stable without sampling, and 69% of methods as type stable with sampling. The last figure comes close to the average 72% of type-stable methods traced via dynamic analysis. As expected,

4 https://github.com/prl-julia/julia-type-stability-checker

our static approach is more conservative in granting the type stability badge because it explores more possibilities for the inputs than what actually occurs in test suites. These numbers also lead us to believe that methods which make the algorithm run out of fuel in the sampling mode are largely type unstable. At least, the sum of such methods and definitely unstable methods (31%) is just over the number of unstable methods identified by tracing in Chap. 3 (28%).

The package having the highest percentage of type-stable methods remains the same as with the dynamic analysis — Genie. This is an encouraging result, as sometimes static analysis tools report too many false positive to be viable in practice. The same is not necessarily an issue with our approach, as the present analysis of Genie shows: a package that does not have type stability issues in practice is also reported as such by the tool.

The least type-stable package, according to the dynamic analysis is Knet, and it is also identifiable on Fig. 5.3. The only difference is that on the figure most of the methods in Knet are marked as *NoFuel* (80%) instead of *Unstable* (the *NoFuel* metrics corresponds to the third of the three possible outcomes listed in Sec. 5.4). This aligns with the observation above that, after sampling is turned on, *NoFuel* metrics should, perhaps, be interpreted as *Unstable*.

Similarly to Knet, DifferentialEquations (called DiffEq for brevity on the figure) holds a substantial number of methods marked *NoFuel* — 76% (and the remaining 24% are *Stable*). This is a significant difference from the results of dynamic analysis where the package comes as 70% stable. A probable reason for this is a considerably different sample of methods analyzed: due to the structural peculiarities of the package, which is organized as several subpackages, the current implementation only processed a portion of methods (namely, those belonging to one subpackage DiffEqBase). The methods analyzed here constitute about 1/5 of methods captured during dynamic analysis.

DifferentialEquations shows the only significant increase (more than 25 percentage points) in the number of methods not identified as stable in comparison to the dynamic analysis. All other packages have this number increased in less than 20 percentage points.

Sampling makes the largest difference for packages that have more *NoFuel*-methods, which is to be expected. For example, 20 or more percentage points increase in the number of stable methods due to sampling is found in least type-stable packages: Knet, DifferentialEquations, Gen, and Flux. Unexpected is that finding counterexamples, i.e. increasing the number of unstable methods, does not always strongly

correlate with the number of *NoFuel*-methods. For example, adding sampling for DifferentialEquations (76% *NoFuel* initially) gives an increase in unstable methods by 3 percentage points, whereas sampling for JuMP (53% *NoFuel* initially, just above the median value of 46%) gives an increase in unstable methods by 7 percentage points. Adding sampling does produce non-neglegible number of counterexamples but, perhaps, could be improved using type databases created specifically to reflect the use cases of a package under analysis.

6 RELATED WORK

Type stability is a consequence of Julia's compilation strategy put into practice. The approach Julia takes is new and simpler than other approaches to efficient compilation of dynamic code.

Attempts to efficiently compile dynamically dispatched languages go back nearly as far as dynamically dispatched languages themselves. Atkinson [1986] used a combination of run-time-checked user-provided types and a simple type inference algorithm to inline methods. Chambers and Ungar [1989] pioneered the just-in-time model of compilation in which methods are specialized based on run-time information. Hölzle and Ungar [1994] followed up with method inlining optimization based on recently observed types at the call site. Rigo [2004] specialized methods on invocation based on their arguments, but this was limited to integers. Similarly, Cannon [2005] developed a type-inferring just-in-time compiler for Python, but it was limited by the precision of type inference. Logozzo and Venter [2010] extended this approach with a more sophisticated abstract interpretation-based inference system for JavaScript.

At the same time, trace-based compilers approached the problem from another angle [Chang et al. 2007]. Instead of inferring from method calls, these compilers had exact type information for variables in straight-line fragments of the program called traces. Gal et al. [2009] describes a trace-based compiler for JavaScript that avoids some pitfalls of type stability, as traces can cross method boundaries. However, it is more difficult to fix a program when tracing does not work well, for the boundaries of traces are not apparent to the programmer.

Few of these approaches to compilation have been formalized. Guo and Palsberg [2011] described the core of a trace-based compiler with two optimizations, variable folding and dead branch/store elimination. Myreen [2010] formalized self-modifying code for x86. Finally, Flückiger et al. [2018] formally described speculation and deoptimization and proved correctness of some optimizations; Barriere et al. [2021] extended and mechanized these results.

The Julia compiler uses standard techniques, but differs considerably in how it applies them. Many production just-in-time compilers

rely on static type information when it is available, as well as a combination of profiling and speculation [Duboscq et al. 2013; Würthinger et al. 2012]. Speculation allows these compilers to perform virtual dispatch more efficiently [Flückiger et al. 2020]. Profiling allows for tuning optimizations to a specific workload [Xu et al. 2009; Ottoni 2018], eliminating overheads not required for cases observed during execution. Julia, on the other hand, performs optimization only once per method instance. This presents both advantages and issues. For one, Julia's performance is more predictable than that of other compilers, as the warmup is simple [Barrett et al. 2017]. Overall, Julia is able to achieve high performance with a simple compiler.

7 CONCLUSIONS

In this dissertation I describe a program property called type stability and the ways it is employed in the Julia programming language. To that end, I make several contributions to facilitate better understanding of the language itself and, more generally, the use of run-time type information in dynamic, just-in-time compiled languages.

First, in Chap. 3, I show that the type stability property is widely exercised in open-source Julia packages. This finding may come as a surprise given that there is no automated tool exists to check the property. Perhaps, much of the Julia code is type stable because it is the most natural way to express algorithms in the language. I show that certain ubiqituos code patterns, e.g. type-constant functions or generic transformations, naturally lead to type-stable code. On the other hand, I point out certain code patterns, especially those coming from traditional object-oriented languages, that produce type-unstable code. These are relatively well known in the Julia community and warned about in the language manual, which helps maintaining high percentage of type-stable code overall. Most encouraging is that Julia packages aimed at performance-critical application have explicit notes about trying to abide by the property of type stability.

Second, my formal model of the Julia JIT in Chap. 4 helps to pinpoint the relationship between type stability and runtime optimizations. The Jules virtual machine recognizes code that I call type grounded and that can only rely on type-stable APIs, and turns it into the most optimized version that I call *full devirtualization*. In practice, not every algorithm can be easily programmed in a type grounded fashion, so the property may be too demanding on the first glance. Yet, about half of the code analyzed in Chap. 3 is in fact type grounded. Conceptually, the idea behind type groundedness is that it provides a radical reference point that facilitates the argument for type stability.

Lastly, in Chap. 5, I build an approach to understanding type stability in terms of the source language and without running the program. The motivation for this task comes from the fact that the formal model in Chap. 4 studies type stability on the level of an intermediate representation inspired by Julia's own, but it is unlikely to be an optimal

model for a casual Julia programmer. The approach in Chap. 5 reuses existing Julia tools like the type inferencer and implementation of the subtyping relation. The idea of reusing the key Julia components ensures that the analysis always agrees with what Julia's optimizer does at run time. The analysis can be run by a package author at the development time or as a part of their continuous integration setup, and does not cost the end-user any performance.

7.1 FUTURE WORK

EVALUATION ON A LARGE-SCALE APPLICATION The evaluation of the type stability approximation approach has been done on a small set of popular Julia packages. It is hard to predict how a package will be used, and, by extension, what type instabilities will matter in practice. Therefore, a better aim for the evaluation may be a Julia application that has a more clear cost promise to the user. Currently, I am working with an industrial company that builds such an application, which consists of about 200K lines of Julia code. The application has a clear set of benchmarks and a constant monitoring of performance regressions. This may be a fruitful setting for tailoring the approximation tool and ideas behind it (for example, the way types database is assembled).

A TOOL FOR FIXING TYPE INSTABILITIES The approach and tool described in Chap. 5 can serve as a bug catching tool if run on every commit in a performance-oriented project to signal any regression in type stability of the code. I think that the bug catching tool can be turned into a bug fixing one. Indeed, there are several recipes in the Julia manual that help to fix type instabilities but none of them were implemented as a tool, to the best of my knowledge. A tool to fix type instabilities may be a good extension for the current tool simply signaling about those instabilities.

GARBAGE CODE COLLECTION While not directly connected to type stability, another problem following from Julia's aggressive approach to type specialization (besides sudden performance regressions due to type instabilities) is code bloating. The formal model of the JIT in Chap. 4 shows how the method table can grow indefinitely during program execution. Although not an issue in the formal setting, it can very well lead to suboptimal performance even in type-stable code. A reasonable extension of the current work relating performance and

type stability would be a study of performance implications of the type specialization strategy.

BIBLIOGRAPHY

Gerald Aigner and Urs Hölzle. 1996. Eliminating Virtual Function Calls in C++ Programs. In *European Conference on Object-Oriented Programming (ECOOP)*. https://doi.org/10.1.1.7.7766 (cited on p. 11)

Robert G. Atkinson. 1986. Hurricane: An Optimizing Compiler for Smalltalk. In *Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*. https://doi.org/10.1145/960112.28712 (cited on p. 74)

Edd Barrett, Carl Friedrich Bolz-Tereick, Rebecca Killick, Sarah Mount, and Laurence Tratt. 2017. Virtual Machine Warmup Blows Hot and Cold. *Proc. ACM Program. Lang.* 1, OOPSLA (2017). https://doi.org/10.1145/3133876 (cited on p. 75)

Aurele Barriere, Olivier Flückiger, Sandrine Blazy, David Pichardie, and Jan Vitek. 2021. Formally Verified Speculation and Deoptimization in a JIT Compiler. *Proc. ACM Program. Lang.* 5, POPL (2021). https://doi.org/10.1145/3434327 (cited on p. 74)

Julia Belyakova, Benjamin Chung, Jack Gelinas, Jameson Nash, Ross Tate, and Jan Vitek. 2020. World Age in Julia: Optimizing Method Dispatch in the Presence of Eval. *Proc. ACM Program. Lang.* 4, OOPSLA (2020). https://doi.org/10.1145/3428275 (cited on p. 27)

Jeff Bezanson, Jiahao Chen, Benjamin 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 (cited on p. 11)

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 (cited on p. 7)

Brett Cannon. 2005. *Localized Type Inference of Atomic Types in Python*. Master's thesis. California Polytechnic State University. (cited on p. 74)

Craig Chambers and David Ungar. 1989. Customization: Optimizing Compiler Technology for SELF, a Dynamically-Typed Object-Oriented Programming Language. In *Proceedings of the ACM Programming Language Design and Implementation Conference (PLDI)*. 146–160. https://doi.org/10.1145/73141.74831 (cited on p. 74)

Mason Chang, Michael Bebenita, Alexander Yermolovich, Andreas Gal, and Michael Franz. 2007. *Efficient just-in-time execution of dynamically typed languages via code specialization using precise runtime type inference*. Technical Report. Donald Bren School of Information and Computer Science, University of California, Irvine. (cited on p. 74)

Benjamin Chung. 2023. *A Type System for Julia*. Ph.D. Dissertation. Northeastern University. (cited on p. 58)

Marco F. Cusumano-Towner, Feras A. Saad, Alexander K. Lew, and Vikash K. Mansinghka. 2019. Gen: A General-purpose Probabilistic Programming System with Programmable Inference. In *Proceedings of the ACM Programming Language Design and Implementation Conference (PLDI)*. https://doi.org/10.1145/3314221.3314642 (cited on p. 23)

L. Peter Deutsch and Allan M. Schiffman. 1984. Efficient Implementation of the Smalltalk-80 System. In *ACM Symposium on Principles of Programming Languages (POPL)*. https://doi.org/10.1145/800017.800542 (cited on p. 11)

Gilles Duboscq, Thomas Würthinger, Lukas Stadler, Christian Wimmer, Doug Simon, and Hanspeter Mössenböck. 2013. An Intermediate Representation for Speculative Optimizations in a Dynamic Compiler. In *Workshop on Virtual Machines and Language Implementations (VMIL)*. https://doi.org/10.1145/2542142.2542143 (cited on p. 75)

Olivier Flückiger, Guido Chair, Ming-Ho Yee, Jan Jecmen, Jakob Hain, and Jan Vitek. 2020. Contextual Dispatch for Function Specialization. *Proc. ACM Program. Lang.* 4, OOPSLA (2020). https://doi.org/10.1145/3428288 (cited on p. 75)

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 (cited on p. 74)

Andreas Gal, Brendan Eich, Mike Shaver, David Anderson, David Mandelin, Mohammad R. Haghighat, Blake Kaplan, Graydon Hoare, Boris Zbarsky, Jason Orendorff, Jesse Ruderman, Edwin W. Smith, Rick Reitmaier, Michael Bebenita, Mason Chang, and Michael Franz. 2009. Trace-based just-in-time type specialization for dynamic languages. In *Programming Language Design and Implementation (PLDI)*. https://doi.org/10.1145/1543135.1542528 (cited on p. 74)

Shu-yu Guo and Jens Palsberg. 2011. The Essence of Compiling with Traces. In *Symposium on Principles of Programming Languages (POPL)*. https://doi.org/10.1145/1926385.1926450 (cited on p. 74)

U. Hölzle and D. Ungar. 1994. Optimizing dynamically-dispatched calls with run-time type feedback. In *Programming Language Design and Implementation (PLDI)*. https://doi.org/10.1145/178243.178478 (cited on p. 74)

Francesco Logozzo and Herman Venter. 2010. RATA: Rapid Atomic Type Analysis by Abstract Interpretation – Application to JavaScript Optimization. In *Compiler Construction (CC)*. https://doi.org/10.1007/978-3-642-11970-5_5 (cited on p. 74)

Magnus O. Myreen. 2010. Verified Just-in-Time Compiler on X86. In *Symposium on Principles of Programming Languages (POPL)*. https://doi.org/10.1145/1706299.1706313 (cited on p. 74)

Guilherme Ottoni. 2018. HHVM JIT: A Profile-Guided, Region-Based Compiler for PHP and Hack. In *Programming Language Design and Implementation (PLDI)*. https://doi.org/10.1145/3192366.3192374 (cited on p. 75)

Artem Pelenitsyn. 2023. Approximating Type Stability in the Julia JIT (Work in Progress). In *Proceedings of the 15th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages* (Cascais, Portugal) *(VMIL 2023)*. ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3563838.3567676 (cited on p. 6)

Artem Pelenitsyn, Julia Belyakova, Benjamin Chung, Ross Tate, and Jan Vitek. 2021. Type Stability in Julia: Avoiding Performance Pathologies in JIT Compilation. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 150 (2021), 26 pages. https://doi.org/10.1145/3485527 (cited on p. 6)

Armin Rigo. 2004. Representation-Based Just-in-Time Specialization and the Psyco Prototype for Python. In *Partial Evaluation and Pro-*

gram Manipulation (PEPM). https://doi.org/10.1145/1014007.1014010 (cited on p. 74)

Thomas Würthinger, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Doug Simon, and Christian Wimmer. 2012. Self-Optimizing AST Interpreters. In *Dynamic Language Symposium (DLS)*. https://doi.org/10.1145/2384577.2384587 (cited on p. 75)

Guoqing Xu, Matthew Arnold, Nick Mitchell, Atanas Rountev, and Gary Sevitsky. 2009. Go with the Flow: Profiling Copies to Find Runtime Bloat. In *Programming Language Design and Implementation (PLDI)*. https://doi.org/10.1145/1542476.1542523 (cited on p. 75)

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 (cited on pp. 6, 8, and 63)

A TYPE STABILITY GRAPHS FROM EMPIRICAL ANALYSIS (SUBSEC. 3.3.1)

This appendix contains graphs similar to the ones described in Subsec. 3.3.1 for all 10 packages discussed in Chap. 3. There are 6 graphs per package: the top two show the relationship between the method size and stability (left) or groundedness (right); the other four graphs connect the two type-related properties with control-flow features: the number of gotos and the number of returns in a method instance.

Note that the bottom four graphs for every package are different from the top two in that they group method instances, not methods. Therefore, the bottom four graphs have all data bins either at level $OY = 0$ or 1, because we always know whether a method instance is stable (grounded) or not. The change comes from the fact that the control-flow features in question depend on compiled code and the way it was optimized: e. g., an `if true` in a method can be optimized away during compilation.

PACKAGE: DIFFERENTIALEQUATIONS

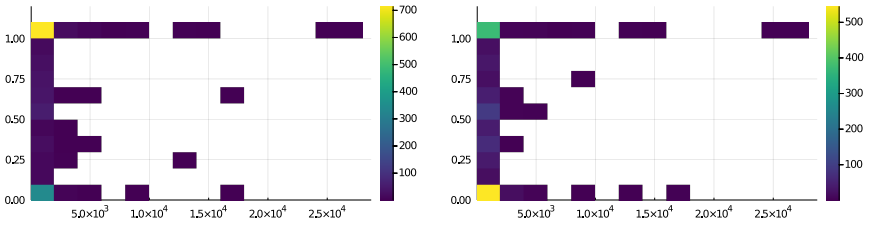

Figure A.1: Stability (left, OY axis) and groundedness (right, OY) by method size (OX)

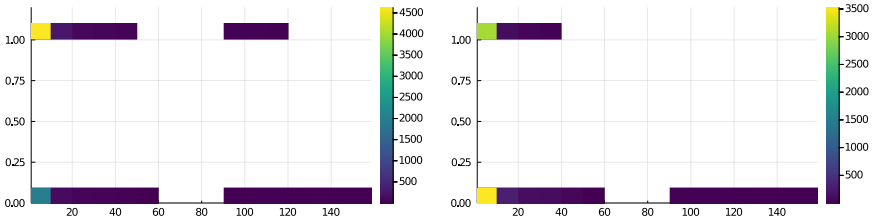

Figure A.2: Stability (left, OY axis) and groundedness (right, OY) by number of gotos in method instances (OX)

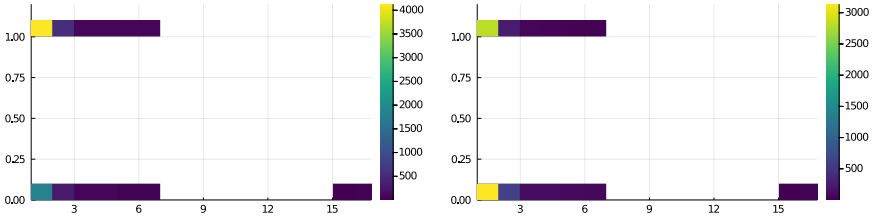

Figure A.3: Stability (left, OY axis) and groundedness (right, OY) by number of returns in method instances (OX)

PACKAGE: FLUX

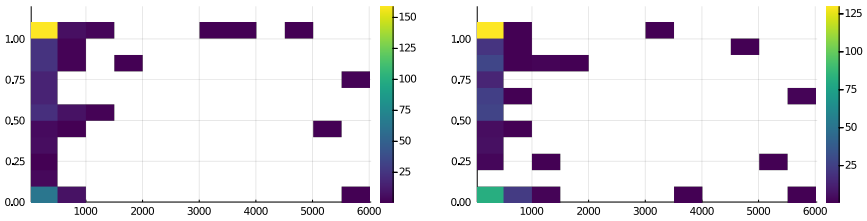

Figure A.4: Stability (left, OY axis) and groundedness (right, OY) by method size (OX)

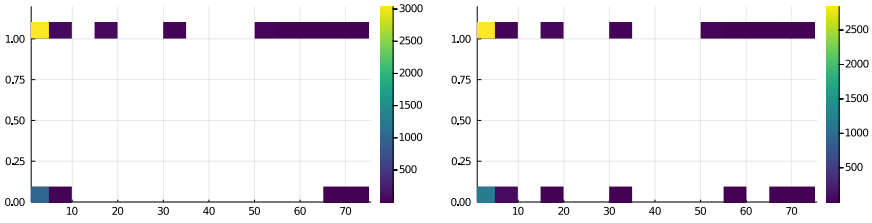

Figure A.5: Stability (left, OY axis) and groundedness (right, OY) by number of gotos in method instances (OX)

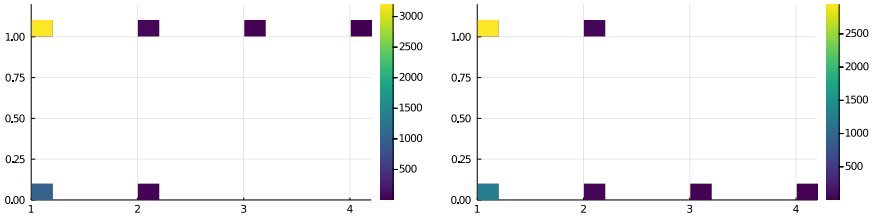

Figure A.6: Stability (left, OY axis) and groundedness (right, OY) by number of returns in method instances (OX)

PACKAGE: GADFLY

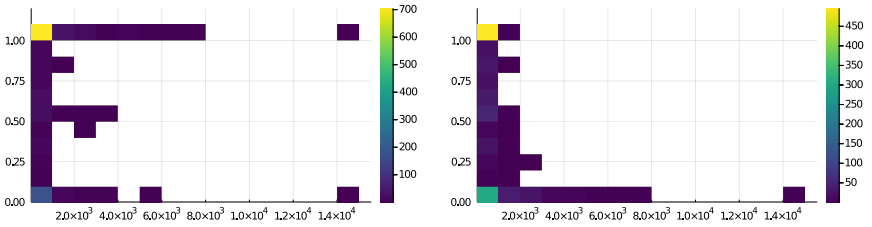

Figure A.7: Stability (left, OY axis) and groundedness (right, OY) by method size (OX)

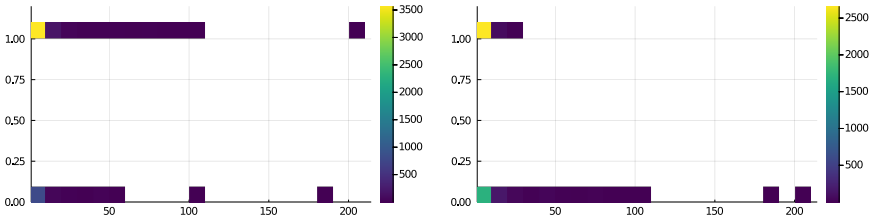

Figure A.8: Stability (left, OY axis) and groundedness (right, OY) by number of gotos in method instances (OX)

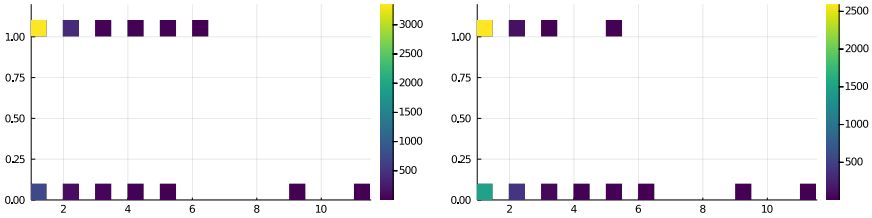

Figure A.9: Stability (left, OY axis) and groundedness (right, OY) by number of returns in method instances (OX)

PACKAGE: GEN

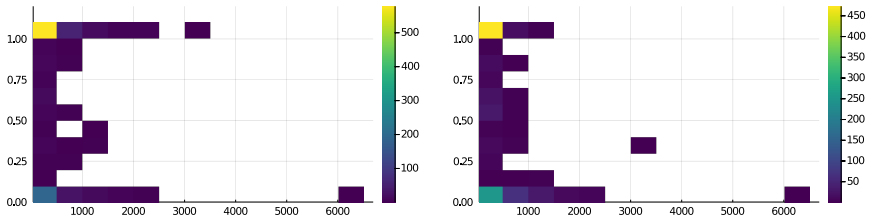

Figure A.10: Stability (left, OY axis) and groundedness (right, OY) by method size (OX)

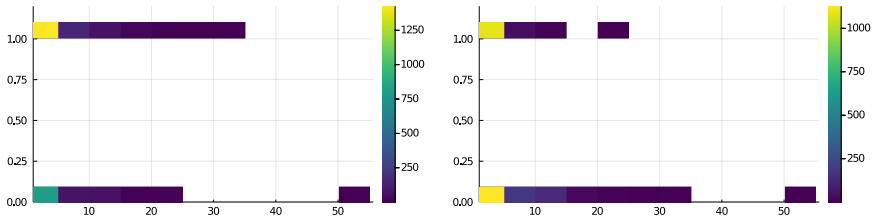

Figure A.11: Stability (left, OY axis) and groundedness (right, OY) by number of gotos in method instances (OX)

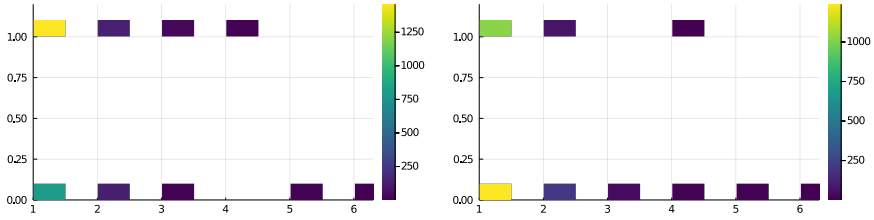

Figure A.12: Stability (left, OY axis) and groundedness (right, OY) by number of returns in method instances (OX)

PACKAGE: GENIE

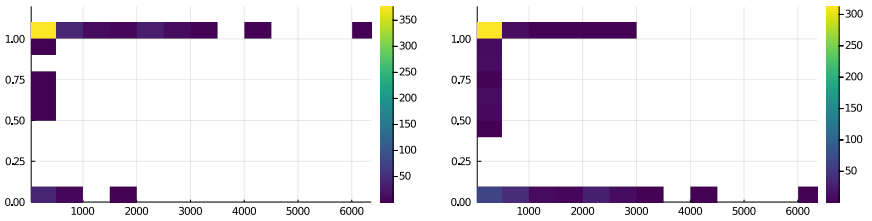

Figure A.13: Stability (left, OY axis) and groundedness (right, OY) by method size (OX)

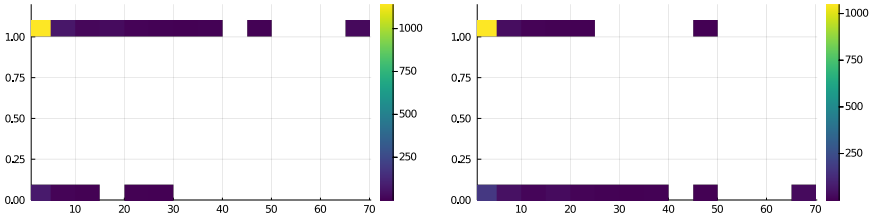

Figure A.14: Stability (left, OY axis) and groundedness (right, OY) by number of gotos in method instances (OX)

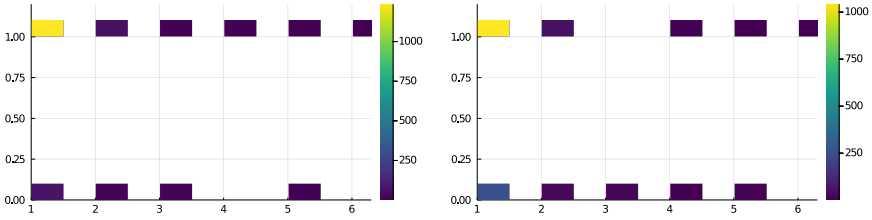

Figure A.15: Stability (left, OY axis) and groundedness (right, OY) by number of returns in method instances (OX)

PACKAGE: IJULIA

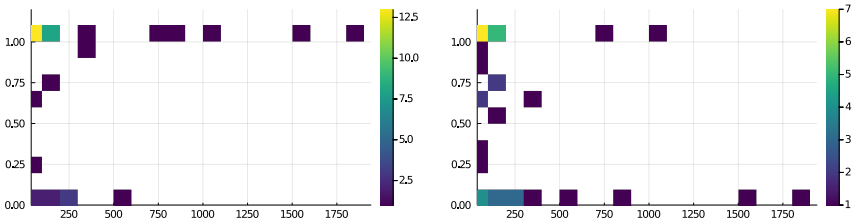

Figure A.16: Stability (left, OY axis) and groundedness (right, OY) by method size (OX)

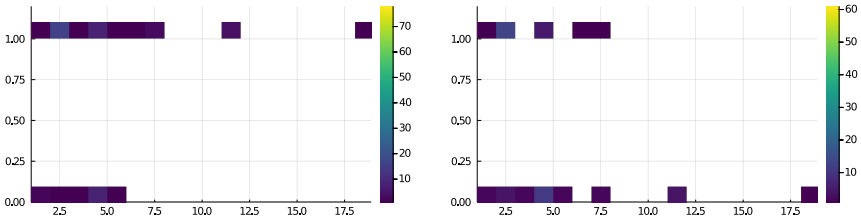

Figure A.17: Stability (left, OY axis) and groundedness (right, OY) by number of gotos in method instances (OX)

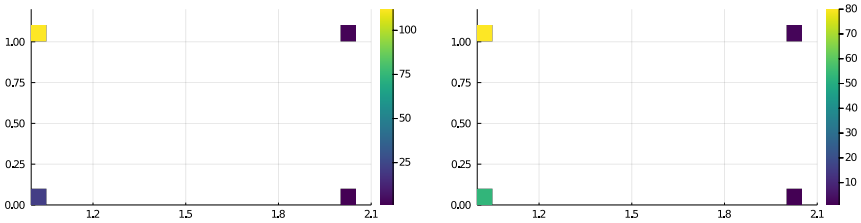

Figure A.18: Stability (left, OY axis) and groundedness (right, OY) by number of returns in method instances (OX)

PACKAGE: JUMP

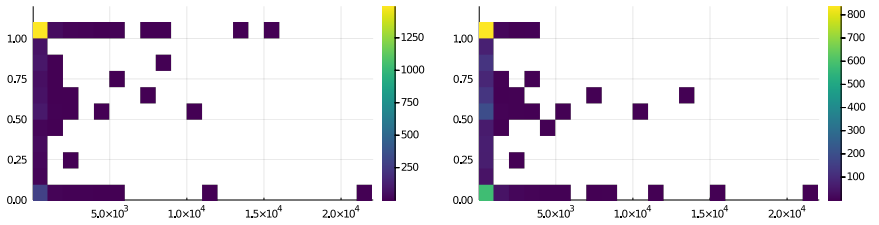

Figure A.19: Stability (left, OY axis) and groundedness (right, OY) by method size (OX)

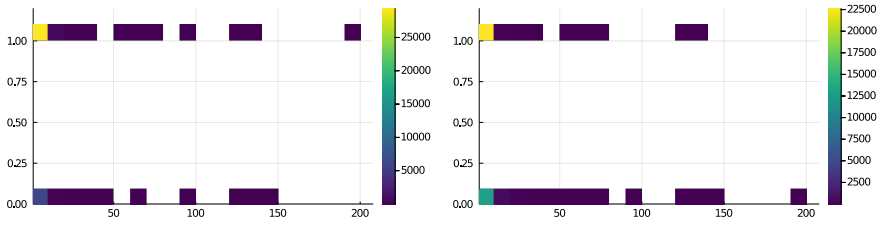

Figure A.20: Stability (left, OY axis) and groundedness (right, OY) by number of gotos in method instances (OX)

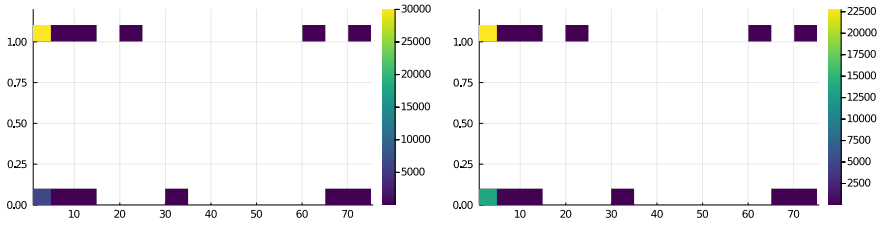

Figure A.21: Stability (left, OY axis) and groundedness (right, OY) by number of returns in method instances (OX)

PACKAGE: KNET

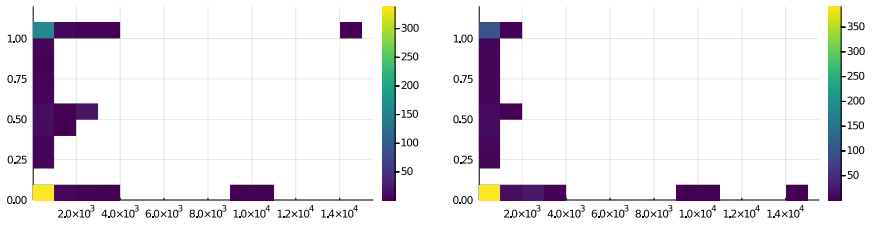

Figure A.22: Stability (left, OY axis) and groundedness (right, OY) by method size (OX)

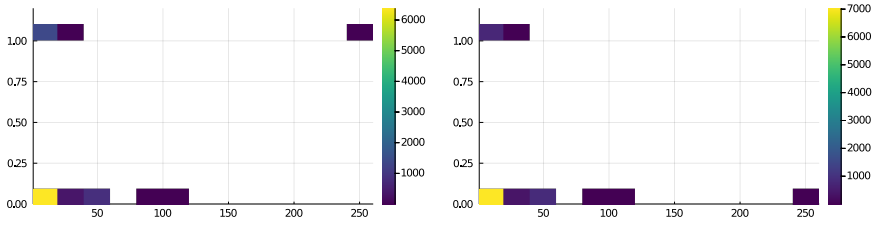

Figure A.23: Stability (left, OY axis) and groundedness (right, OY) by number of gotos in method instances (OX)

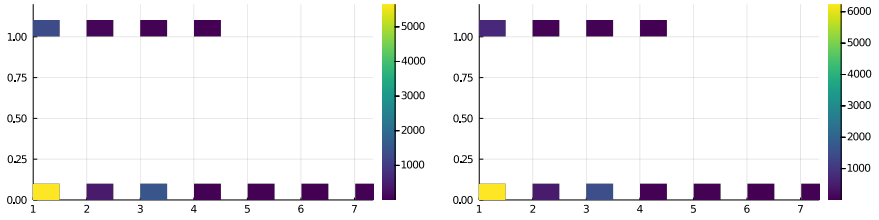

Figure A.24: Stability (left, OY axis) and groundedness (right, OY) by number of returns in method instances (OX)

PACKAGE: PLOTS

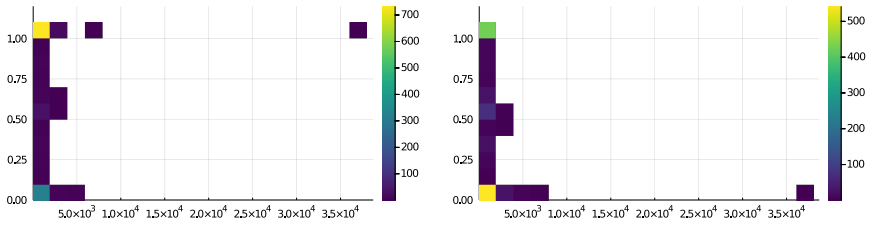

Figure A.25: Stability (left, OY axis) and groundedness (right, OY) by method size (OX)

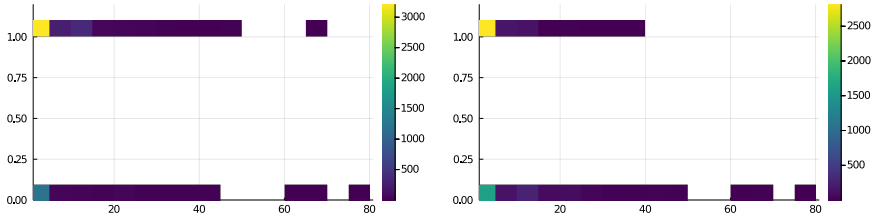

Figure A.26: Stability (left, OY axis) and groundedness (right, OY) by number of gotos in method instances (OX)

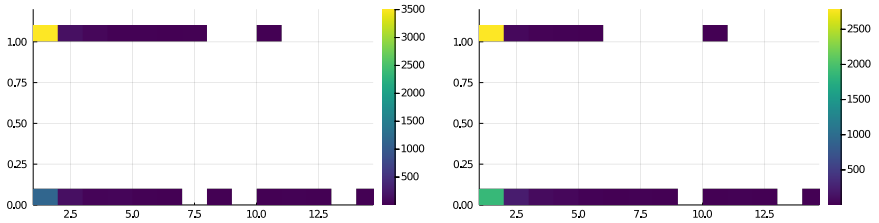

Figure A.27: Stability (left, OY axis) and groundedness (right, OY) by number of returns in method instances (OX)

PACKAGE: PLUTO

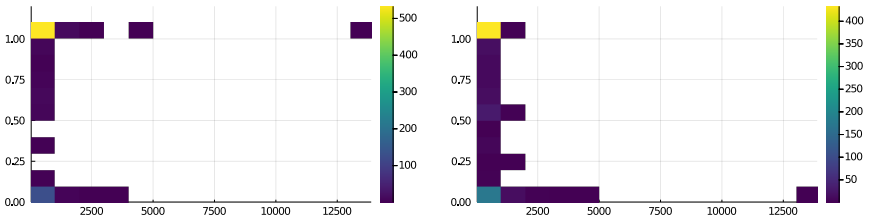

Figure A.28: Stability (left, OY axis) and groundedness (right, OY) by method size (OX)

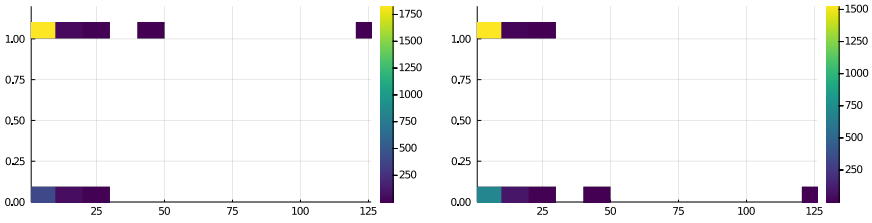

Figure A.29: Stability (left, OY axis) and groundedness (right, OY) by number of gotos in method instances (OX)

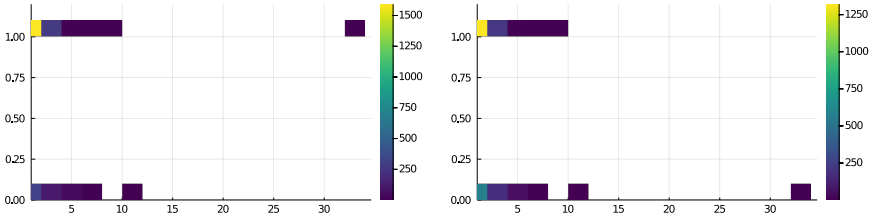

Figure A.30: Stability (left, OY axis) and groundedness (right, OY) by number of returns in method instances (OX)

B DATA FOR EVALUATION OF THE TYPE STABILITY APPROXIMATION ALGORITHM (SEC. 5.5)

The tool implementing the algorithm from Chap. 5 has two technical limitations, in particular:

- The tool relies on Julia's type checker, which may fail on some inputs for unclear reasons (probably a bug). This rarely happens: for the whole corpus it failed on two methods.
- Generic methods in Julia, e.g.

  ```
  function length(v::Vector{T}) where T
  ...
  end
  ```

  are represented internally in a different ways than usual methods, e.g.

  ```
  function length(v::Vector{T} where T)
  ...
  end
  ```

  (notice that the `where` clause is now inside the parenthesis). Currently, the tool is only suited to process the latter form, but not the former. In our corpus, an average package contains 14% of generic methods, which our tool currently cannot process. This limitation will be lifted in a future version of the tool.

Table B.1 contains absolute numbers of the methods processed by the tool. First four numeric columns: Methods, TyChkErr, Generic, MethodsOK are related by the following formula:

$$\text{MethodsOK} = \text{Methods} - \text{TyChkErr} - \text{Generic},$$

which acknowledges the limitations mentioned above. The rest of columns are divided into two groups, three columns each. In a row of each group the numbers sum up to the number from MethodsOK in the same row.

Module	Methods	TyChkErr	Generic	MethodsOK	Stable	Unst	NoFuel	DBStable	DBUnst	DBNoFuel
DiffEq	303	0	64	239	58	0	181	110	6	123
Flux	531	0	35	496	156	4	336	268	20	179
Gadfly	752	0	52	700	510	52	138	546	56	98
Gen	993	0	188	805	422	26	357	581	89	135
Genie	2304	0	53	2251	2193	25	33	2198	38	15
IJulia	220	0	1	219	178	5	36	196	10	13
JuMP	973	2	235	736	337	8	391	476	56	204
Knet	3834	0	1692	2142	391	28	1723	1048	60	1033
Plots	1628	0	42	1586	653	184	749	860	223	503
Pluto	863	0	23	840	606	63	171	671	89	79

Table B.1: Absolute numbers for methods analyzed with the type stability approximation algorithm

Module	Stable	Unst	NoFuel	DBStable	DBUnst	DBNoFuel	StableDiff	UnstDiff
Flux	31	1	68	54	4	36	23	3
Gadfly	73	7	20	78	8	14	5	1
Gen	52	3	44	72	11	17	20	8
Genie	97	1	1	98	2	1	0	1
IJulia	81	2	16	89	5	6	8	2
JuMP	46	1	53	65	8	28	19	7
Knet	18	1	80	49	3	48	31	1
Plots	41	12	47	54	14	32	13	2
Pluto	72	8	20	80	11	9	8	3
Average	54	4	43	69	7	24	15	3
Median	49	2	46	68	6	22	16	2

Table B.2: Percentages for methods analyzed with the type stability approximation algorithm. The last two columns are expressed in percentage points.