Building a TASTy interpreter

Shardul Chiplunkar
February 1, 2023

Abstract

In this report, I describe work towards an interpreter for TASTy, a high-level binary interchange format for Scala 3, motivated by the desire for a specification for the format independent of the compiler. I explain some core implementation details and summarize the current state of the project and avenues for future work, particularly regarding JavaScript interoperability and the Scala.js standard library. I also describe a subsidiary project, TASTyViz, a web-based TASTy visualizer motivated by the absence of human interfaces for TASTy. I conclude by reflecting on the challenges and takeaways of this project.

Introduction

What is TASTy?

The Scala 3 compiler ("Dotty") introduces a new high-level intermediate representation and binary interchange format called *TASTy*, a near-acronym for *Typed Abstract Syntax Trees*. TASTy tries to explicitly represent many kinds of information that is implicit or inferred by the compiler from the Scala[1] source, including types, implicit parameters, scoped name resolution, and method overload resolution. A TASTy file represents the ASTs of the corresponding Scala top-level statements along with enough additional information to straightforwardly compute such inferred properties on demand.

TASTy enables two kinds of "interchange". The first is to better support *separate compilation*: the compiler does not have to recompile a project's dependencies when recompiling the project. Instead, the compiler can use the rich information stored in the dependencies' TASTy files and trust that they typechecked and compiled correctly. The second kind of interchange is between the compiler and other tools that deal with Scala source code such as the Metals language server (for highlighting, navigation, completion, etc. in IDEs), the TASTy-MiMa API compatibility checker, and static analyzers. TASTy makes the parsing and inference results of the compiler available to such tools, that would otherwise have spent a lot of effort duplicating that work, often less accurately than the compiler.

Motivation behind the interpreter

For an interchange format like TASTy to be most useful, there needs to be a convenient API to read the binary format, and a precise specification of its semantics. The tasty-query project in development at the Scala Center serves the former purpose. The latter, until now, was only served indirectly by the Scala compiler itself, insofar as the compiler guarantees that Java bytecode compiled from TASTy will behave the same as Java bytecode compiled from Scala sources that produce the same TASTy. This is unsatisfactory because the compiler only tries to implement the semantics of Scala, which does not guarantee anything about the semantics of intermediate representations like TASTy, and further depends on the JVM to actually execute programs.

Specifying the semantics of TASTy independently of the compiler and the JVM is the primary motivation behind this project to build a TASTy interpreter. The interpreter is intended to be an 'executable specification' in the sense that it produces correct outputs for all valid (syntactically well-formed, well-typed) input programs but promises nothing for invalid input programs. In its ideal finished state, the interpreter would keep the compiler honest about the meaning of TASTy files it outputs.

A secondary motivation for this project comes from the fact that the interpreter is written in Scala.js, for reasons explained in § Design principles, which brings us one step closer to running Scala directly in a browser or other JavaScript (JS) environment. The Scala.js compiler compiles Scala (or rather, a large subset of Scala, with minor semantic differences) into JS, but unfortunately cannot compile itself. So, currently, a JS environment can only indirectly run Scala code by first compiling it with the Scala.js compiler running on the JVM. It seems more feasible to compile to JS only that portion of the Scala.js compiler that lowers Scala to TASTy, and then interpret the resulting TASTy.

[1] In the rest of this report, "Scala" means Scala 3 unless specified otherwise.

TASTy is close enough to the surface-level Scala that a TASTy interpreter could serve as the foundation for Scala debugging tools. Stepping through the execution of a program, inspecting state, adding instrumentation, etc. would likely be easier with a naïve interpreter as opposed to an optimizing compiler that targets the JVM.

TASTy visualization

A particular challenge I encountered during this project was not being able to explore the contents of TASTy files in a human-friendly way. I was not satisfied with the existing console-based pretty-printer included with the Scala compiler (`scalac -print-tasty`). tasty-query provides a great *programmatic* interface to read TASTy but is not designed for human consumption, so its text dumps were also not very useful. Thus, I made my own web-based TASTy visualizer called TASTyViz, described in § TASTyViz.

Design of the interpreter

This section is not intended as documentation of the code but as explanation of its interesting or non-obvious portions. It will not make complete sense without having the code for reference.

Design principles

Two principles characterize the design of the interpreter. The first is that the interpreter should be as close as possible to the conceptual execution model of Scala, as described in the official language specifications. In other words, the interpreter should implement Scala's semantics almost 'by definition'. This means that the interpreter does not perform any optimizations, does not collapse or condense internal representations for efficiency, and does not try to interpret complex language constructs in terms of simpler ones, e.g. by 'compiling them away' and emitting intermediate TASTy for itself to consume later. Of course, although the gap between Scala's semantics and TASTy's semantics is small, it still leaves some room for implementation choices because this is a TASTy interpreter and not a Scala interpreter. In such cases, I tried to pick the most conceptually simple implementation.

The second design principle is to interpret Scala.js (and to support interoperability with JS) instead of Scala, and then naturally, to build the interpreter in Scala.js instead of Scala. For an end user, Scala.js code is nearly identical to Scala code, as is its behavior. But a major implementation difference lies in the fact that Scala code may call arbitrary parts of the Java API whereas Scala.js code may only call the subset of the Java API that has been manually ported to Scala.js. In the former case, an interpreter would have to defer to the JVM to execute Java calls—already a nontrivial engineering challenge—which may then even call back to Scala code, possibly already compiled, posing an even greater challenge for the interpreter to regain control. In the latter case, Scala sources are available for the restricted Java API, which dispatch the most fundamental operations to the underlying JS API. This is much easier to interpret. Consequently, supporting this JavaScript dispatch is essential to the interpreter, because it is also an essential point in the Scala.js story for JS interoperability.

Representations

In the rest of this report, terms in normal typeface used as normal words in text (like 'object') will refer to *concepts* or *language specifications*, while terms in fixed-width typeface (like `ScalaObject`) will refer to *implementations* inside the interpreter.

An environment (synonymous with 'frame') is represented as an instance of `ScalaEnvironment`, with mutable maps for term bindings and type bindings; a parent environment, which is absent only for the top-level environment; and an optional binding for the `this` object. Terms and types inhabit separate namespaces in Scala so the two binding maps are entirely unrelated. The keys for each are TASTy *symbols* (unique per actual entity being referenced) and the values are inside a thin `ScalaBox` wrapper to support mutable variables (`vars` in Scala). The language specifications say little about environments and the environment hierarchy, hence many of the implementation choices are mine.

A method is represented as an instance of `ScalaMethod` with an `apply` method to evaluate its body given evaluated arguments. Note that in Scala, methods are the only entity that can be 'called', and they are not objects. A method must be converted to a function (which is an object of type `FunctionN`) before passing it as an argument to other methods/functions. Calling a function really means calling its `apply` method, and a function is really a thin wrapper around a (possibly anonymous) method. Functions are represented as instances of `ScalaFunctionObjects`. The interpreter also has

`BuiltInMethods` which execute Scala code directly when called.

A class or trait is represented as an instance of `ScalaClass` and has its own environment where *unspecialized* methods (inheriting from `ScalaSpecializable`) are bound. An object is represented as an instance of `ScalaObject` with its own environment. When an object is instantiated, the methods of its class and those the class inherits are *specialized* to that object and bound in the object's environment. A `ScalaClassMethod` is specialized into a `ScalaMethod` that evaluates in the environment of a specific object; a `ScalaClassBuiltInMethod` is specialized into a `BuiltInMethod` that closes over a specific object. The instantiation process is described in greater detail in § How an object is instantiated. A `ScalaObject` also has methods to resolve a field/method access (in Scala, `obj.field`, `this.field`, or just `field`, depending on the context) or a super method access (`super.method` or `super[which].method` when lexically within the object). The latter is surprisingly tricky and is described in § `Super`.

Lastly, the data types `Boolean`, `Int`, `String`, `Unit`, and `Null` are represented as subclasses of `ScalaObject` with corresponding names prefixed with `Scala`. These are placeholder implementations until the interpreter is able to interpret the ported Java API and ultimately dispatch primitive operations to the underlying JS environment. Unfortunately, no progress has been made towards this goal yet, but ideas and possible future directions are described in § Evaluation.

How an object is instantiated

Suppose an object of class `Foo` is to be instantiated. The corresponding TASTy will roughly have the form `Apply(Select(New(Foo), <init>), List(constructor arguments))`. Evaluating the `New` node will create a new `ScalaObject` whose `ScalaEnvironment` points to itself as the `this` object, and has the constructor, `<init>`, bound to a specialized `ScalaClassBuiltInMethod` that was defined while evaluating `Foo`'s definition. Then as expected, `Select` will select this newly bound method, and `Apply` will call it with the provided arguments. The object is considered uninstantiated until this outermost call terminates.

Note that `Foo` may inherit from multiple classes/traits in Scala, but lowering to TASTy produces a *linearization* of the inheritance hierarchy, and in particular, gives one concrete parent class for `Foo` and possibly some mixin classes/traits.

Before presenting a detailed account of the constructor, it will be useful to have a broad (but less precise) picture in mind, as follows. First, some fields of the object are initialized to some default values (which ones specifically will be explained in the next paragraph). Then, the concrete parent class constructor is evaluated, which performs these steps recursively; followed by the mixin constructors, which also perform these steps but do not recurse. Each constructor evaluation is itself of the form `Apply(Select(New(…), <init>), …)`. The `createNewEnvironments` flag on the current `ScalaObject` ensures that all these constructors affect the same `ScalaObject` instead of creating new ones every time a `New` node is encountered. Finally, all the methods and fields are bound, and the object is returned from the constructor.

Below is the full description of how the constructor of a class/trait `C` takes an uninstantiated object and instantiates the `C` aspect of it:

1. Fields that access constructor arguments are bound in the object's environment to the values of those arguments. (They may be needed to call parent constructors, which happens before other fields are bound.)
2. Other fields are initialized to null, false, zero, etc. per their erased type. `lazy vals` are initialized as instances of `ScalaLazyValue` but not computed. Both these initializations are done for fields that are *not overridden* by a class/trait lower in the linearization.
3. The parent constructors are evaluated in the object's environment. Specifically, the concrete parent class constructor is evaluated (which performs these steps recursively), and then the mixins are evaluated (which do not recursively call any constructors). Finally, `createNewEnvironments` is set to true so that step #5 can create new objects not part of the current recursive instantiation.
4. `C`'s methods, represented in the interpreter as `ScalaSpecializables`, are specialized to this object and bound in its environment.
5. Inner classes and non-overridden fields are initialized. New bindings in the object's environment are created for inner classes while the bindings from #2 are updated for fields.
6. `createNewEnvironments` is reset to false, unless

`C` was the 'lowest' class in the linearization, in which case the object is instantiated and `createNewEnvironments` stays true (from #3) so that methods and fields are able to create new objects. In any case, the object is returned from the constructor.

Some possible bugs in this implementation have not yet been addressed. `New` nodes passed as arguments to parent constructors should cause new objects to be created, which they currently don't. Methods should be specialized and bound before calling any parent constructors because a constructor might call a method concretely defined in the child class. These and other bugs could have a smooth resolution if `createNewEnvironments` is changed from a field of `ScalaObject` to a parameter to `evaluate`, toggled appropriately during recursive calls for object instantiation.

Special cases

At the top level, the interpreter is a match-case on TASTy nodes that calls a method to evaluate each node, which can in turn call back to the match-case for inner nodes. `Ident`/`Select` nodes and `Super` nodes are notable departures from this pattern.

`Ident` and `Select` Evaluating an `Ident` or `Select` node produces different results depending on whether the node is being used for its value or for its reference to the value. An environment lookup or an object field access returns a reference `r` of type `ScalaBox[ScalaTerm]`, but by default, a value is expected of type `ScalaObject`. The first step of the conversion is to unwrap the `ScalaBox` via `r.value`. The second step is to coerce the resulting `ScalaTerm` to a `ScalaObject` via `forceValue()`. The concrete subclasses of `ScalaTerm` implement `forceValue` differently:

- A `ScalaObject` returns itself unchanged.
- A `ScalaLazyValue` triggers its deferred computation and returns the resulting `ScalaObject`.
- A `ScalaMethod` or a `BuiltInMethod` evaluates its body with an empty argument list and returns the result. (This is because Scala follows the uniform access principle, and a method by itself is not a value/object.)

There are two circumstances in which the above conversion to a value of type `ScalaObject` does not occur. The first is when evaluating method calls with explicit parentheses or arguments: the method being called should not be evaluated with an empty argument list by `forceValue()`, but should be passed the arguments from the program. The second is when evaluating an assignment statement. Here, the original value of the left-hand side does not matter at all, but the mutable `value` inside the `ScalaBox` is updated to refer to the result of evaluating the right-hand side of the assignment.

`Super` The `Super` node in TASTy is exceptional because it is the only node that does not correspond to a standalone language construct. Scala objects can access super class *methods*[2], but cannot access a super class or 'super object', whatever that means; inside the lexical scope of an object, `super.foo` is syntactically valid, but just `super` is not. Yet TASTy maps the former compositionally to `Select(Super(…), foo)`. Thus, the interpreter does not try to evaluate a `Super` node on its own, but rather only inside `Select` nodes.

Two similar syntaxes for super accesses belie two very different semantics. The first is with an unqualified `super`, as in `super.foo`, which I will call a *dynamic* super access. Its meaning depends on two pieces of information: the linearization of the *run-time class* of the object in the context of which this expression is being evaluated, and the *enclosing class* which lexically contains this expression. Let *LS* be the suffix of the linearization of the run-time class starting right after the enclosing class. Then `super.foo` refers *at run-time* to the first concrete method definition for `foo` among the classes in *LS*. At compile time, `super.foo` statically refers to the first concrete method definition for `foo` among the classes in the linearization of the *enclosing class*.

For instance, consider the following Scala code.

```
class B:
  def foo = 2

trait T:
  def foo = 3

trait U:
  def foo: Int

trait V extends B:
  override def foo = 5
  def bar = super.foo
```

[2] Or super class type members, but I ignore this for now.

```
class A extends B, T, U, V:
  override def foo = 7

val x = A().bar
```

When evaluating `super.foo`, the run-time class is `A` and the enclosing class is `V`. The linearization of the run-time class is `[A, V, U, T, B]` so the suffix starting right after the enclosing class is `[U, T, B]`. Among those, the first concrete definition of `foo` appears in `T`, so that is the one referred to. The value of `x` is thus 3. Note that the enclosing class `V` statically has no relation to `T` and that the symbol `foo` in `super.foo` *statically* refers to the `foo` in `B`, which appears in the linearization of `V`.

The second syntax for super method accesses involves a class/trait qualifier, as in `super[T].foo`, which I will call a *static* super method access. Only the enclosing class that lexically contains this expression is required to resolve its referent. The language specification requires that `T` be a parent of this enclosing class and that it contain a concrete method definition for `foo`. This definition is the one referred to by `super[T].foo` at run-time too.

For instance, consider the following Scala code.

```
trait T:
  def foo = 3

class B extends T:
  override def foo = 2

trait U:
  def foo: Int

trait V extends B:
  override def foo = 5

class A extends B, U, V:
  override def foo = 7
  def barB = super[B].foo
  // def barU = super[U].foo
  // def barT = super[T].foo

val x = A().barB
```

The value of `x` is 2 from the definition of `foo` in `B`. Uncommenting the line with `barU` will give a compilation error because `U` only abstractly declares `foo`. Uncommenting the line with `barT` will give a compilation error because `T` is not a parent of the enclosing class `A`, even though `T` appears in `A`'s linearization.

Static and dynamic super access resolution is supported by the `matchingSymbol` method of `TermOrTypeSymbol` in tasty-query, used inside methods of `ScalaObject`.

A further complication with super accesses is that they can optionally have a class prefix, as in `C.super.foo`, specifying which class to consider as the enclosing class if there are multiple lexically nested enclosing classes. Both static and dynamic super accesses with this prefix will be resolved with reference to `C` as the enclosing class. However, these are not represented with `Super` nodes in TASTy. Instead, an *artifact method* with no body is created in `C`, whose name encodes the prefixed super access, and the original super access is replaced with a reference to this artifact method. The super access has to be resolved at run-time by decoding the name, which happens in the `resolveSuperAccessor` method of `ScalaObject`.

Below is an example of prefixed super accesses.

```
class A:
  def foo = 2

class AA:
  def foo = 3

trait T:
  def foo = 5

class B extends A, T:
  override def foo = 7
  class BB extends AA:
    def barT = B.super.foo
    def barA = B.super[A].foo

val objB = B()
val objBB = objB.BB()
val x = objBB.barT
val y = objBB.barA
```

The value of `x` will be 5 and that of `y`, 2. Here is a purely suggestive depiction of the TASTy for `B`:

```
class B extends A with T:
  override def foo = 7
  artifact def superfoo: Int
  artifact def superfoo$A: Int

  class BB extends AA:
    def barT = this[B].superfoo
    def barA = this[B].superfoo$A
```

`superfoo` and `superfoo$A` are the artifact methods described before. In TASTy, `superfoo` is actually `PrefixedName(super, foo)`, and `superfoo$A` is `PrefixedName(super, ExpandedName($, foo, A))`. `this[B]` is not valid syntax in Scala but `This` nodes in TASTy are qualified with which enclosing class they refer too. So the original super access expression is replaced with, roughly, `Select(This(B), PrefixedName(…))`.

One final special case involves dynamic super accesses from traits, i.e. expressions of the form `super.method` in the body of a trait `T` whose linearization contains at least one class/trait `U` that defines `method`. Statically, `super.method` will refer to the definition in the first such `U`. But note that since `T` is a trait, there will never be objects whose run-time class is `T`, and hence the linearization of `T` will never be relevant. The static referent of `super.method` in `U` may have no relation at all to the run-time referent which depends on the *run-time class*, which may interpose arbitrarily many classes/traits between `T` and `U` in its linearization. Hence, dynamic super accesses from traits are not represented as `Super` nodes in TASTy, but with artifact methods, similar to the prefixed super accesses described earlier.

Here is an example in Scala and then in suggestive TASTy as before.

```
class A:
  def foo = 2

trait T extends A:
  override def foo = 3
  def bar = super.foo


class A extends Object:
  def foo = 2

trait T extends Object with A:
  override def foo = 3
  artifact def
    supercom$example$package$T$$foo: Int
  def bar = this[T]
    .supercom$example$package$T$$foo
```

This time, the artifact method name is `PrefixedName(super, ExpandedName($$, ExpandedName(…), foo))`, where the elided expression consists of nested `ExpandedNames` with single `$` tags that represents the fully qualified name of the enclosing class (here, `com.example.package.T`). The actual method name `foo` appears as the third argument instead of as the second as in the `PrefixedNames` for prefix super accesses. I'm not sure why these inconsistencies exist or if there is a larger pattern with other occurrences of artifact names that I'm not aware of.

Evaluation

The interpreter has a comprehensive implementation of the object model, including nuances of field initialization, multiple inheritance, super method accesses, and nested class definitions. This is implemented in only a few hundred lines of code. I have attempted to write exhaustive tests for everything the interpreter can correctly interpret, checked against both the output of the Scala compiler and my personal manual evaluation of the input programs. The entire project is available on GitHub.

However, the interpreter falls far short of being a usable interpreter for realistic programs, with three main shortcomings. I will describe them in increasing order of how interesting I think they will be from a research perspective.

The first is that the interpreter lacks the 'plumbing' necessary for package declarations, main methods, import statements (for internal code or external libraries), and generally scaling to projects spread over multiple files that shouldn't be maintained in memory all at once. I don't expect any major obstacles here but I also don't expect it to be an insignificant engineering effort.

The second shortcoming is that the interpreter does not handle Scala's expressive type system at all. Type parameters, type members, casting, etc. have no support in the interpreter, and in fact, many design decisions were made without considering types at all. For instance, § `Super` above explains the implementation of super *method* accesses in great detail, but ignores the fact that `super` expressions can also access types. However, this should not be a severe issue, because in TASTy (and lower in the compiler), types are only relevant to the semantics to the extent of their highly simplified *erasure* for the Java runtime.

The third and most severe shortcoming is that the interpreter does not tie into the standard library, or to be specific, the Scala.js standard library along with the subset of the Java standard library that has been ported to Scala.js. Primitive operations like `+` on data types like `String` are implemented as 'mocks' like `ScalaString` within the interpreter with `BuiltInMethods`. However, some operations

delegate to `java.lang.String`, and some ultimately to functions in the underlying JS environment. The interpreter should be able to interpret these delegations. More broadly, this requires implementing JS interoperability features such as native and non-native JS types that allow Scala.js code to access JS APIs and emit JS-accessible APIs. This was one of the design principles for the interpreter but unfortunately I was not able to bring it into practice. I think supporting interoperability with JS and integrating with the standard library are the most interesting directions for future work and also the most consequential for the success of this project.

Other, more minor missing features in the interpreter include support for match-case statements (which probably partially depends on support for types), abstract classes and members, loops, and exceptions. Any of these could make for good starting points for someone looking to further develop the interpreter.

TASTyViz

TASTyViz is a web-based visualizer for the TASTy format. I developed it as a side project during my work on the TASTy interpreter.

Motivation

While working on the TASTy interpreter, I often needed to explore the contents of TASTy files and various properties of TASTy symbols exposed by tasty-query. tasty-query provides a great *programmatic* interface to read TASTy but is not designed for human consumption; the best it can do is produce a long string representation of just the AST. I was also not satisfied with the existing console-based pretty-printer included with the Scala compiler (`scalac -print-tasty`), although it includes a little more information than the string output from tasty-query, as illustrated in Figure 1. I envisioned a visual, reactive interface more like Figures 2 and 3.

This drastic gap in tooling for TASTy was the primary motivation for TASTyViz. TASTyViz would not have been possible without tasty-query, but for the development of any other tooling project that uses TASTy, I think both the programmatic and the human interface are essential. TASTyViz may even encourage such projects to begin with by providing an easy entry point to understanding and working with TASTy.

A secondary motivation was to learn about JS interoperability and web development more broadly in Scala.js. Understanding how to use and write native and non-native JS types was important for the interpreter too, and I wanted to introduce myself to the Scala.js web development ecosystem through a hands-on project.

Design

TASTyViz is built on the MVC pattern (Model-View-Controller). The models are lightweight wrappers around data structures provided by tasty-query. (Often, the model is used only to get the underlying tasty-query data structure, which exposes a lot more functionality.) There are three views of these data: the `PackageView` allows the user to explore the contents of packages similar to a filesystem browser; the `DefTreeView` allows the user to explore a particular TASTy tree containing the definition of a class, field, type, etc.; and the `SymbolInfoView` displays information about TASTy symbols selected by the user, such as their type, TASTy flags, links to enclosing symbols, etc. The controller manages the state of the application and handles user input.

TASTyViz runs entirely client-side. It initializes tasty-query with a classpath containing URLs to source JAR files. These could be anywhere on the network in theory, but are in practice served by the same web server as TASTyViz itself. (Adding a remote directory of TASTy files to the classpath is not yet supported but should be possible.) Among these JAR files are the standard libraries, for some of which TASTy is not available. Once all the JAR files on the classpath are loaded, the main interface launches, allowing the user to browse anything on the classpath for which TASTy is available.

TASTyViz relies on a few external Scala and JS libraries. Of course, tasty-query with a custom classpath loader is used to read TASTy. scalatags is used to generate HTML. scalajs-dom is used to access the JS DOM Fetch API. jQuery is used via jquery-facade. The reactive tree interface is powered by the jQuery plugin jsTree via a custom facade. Lastly, a custom facade allows access to the JS History API.

Evaluation

TASTyViz constitutes significant improvement over the prior state-of-the-art in human interfaces to TASTy. It allows users to navigate the package hierarchy, visually explore a TASTy tree, navigate to related trees, collapse/expand and search nodes,

view additional information about TASTy symbols such as flags and computed types, and share or reload application URLs. There are certainly many ways to improve this nascent project so I encourage others working with TASTy to try using it, report bugs, suggest improvements, and contribute. The entire project is available on GitHub.

As of yet, there aren't really any interesting research questions about TASTyViz. It could form the basis for new TASTy tools or improve the user experience of existing ones, or human interaction research could inform its further development, as the interface has so far been designed solely by my intuition (within the limits of my technical abilities).

I learned through the input of my colleagues Matthieu Bovel (LAMP) and Sébastien Doeraene (Scala Center) that I took a rather old-fashioned approach to web development. This is probably because I have no training in web development, but I had to start somewhere, and this project has convinced me to try to learn to build modern web applications.

Challenges

One of the main challenges I faced during this project came from my depth-first approach in building the core functionality of the interpreter. I chose to try to iron out all the details and edge cases of object instantiation and multiple inheritance, instead of taking a breadth-first approach and implementing the basics of JS interoperability, type parameters and type members, import statements, etc. so that the interpreter could evaluate simple yet realistic complete programs (and simple portions of the standard library) as soon as possible. In the end, I built a solid foundation for the interpreter with a comprehensive implementation of the object model, but the interesting research question of JS interoperability remained untouched. It took three or four careful iterations of a week or two each to get the object model exactly right.

Another challenge, particularly at the start of the project, was to understand the TASTy format and what exactly tasty-query exposed about it. This later motivated TASTyViz as the tool I wished I had. Had I had it from the start, I estimate that my work would have been sped up by a couple weeks. Multiple conversations with Sébastien Doeraene were also essential in understanding TASTy and tasty-query.

A blessing in disguise was the fact that my work on the interpreter uncovered several bugs in tasty-query. This slowed down my work because I had to keep rechecking my code until I shifted sufficient doubt from myself to tasty-query, but prompted many fixes and design improvements in the library.

Lastly, while working on TASTyViz, I found it difficult at first to understand how Scala.js interoperability with JS worked. The Scala.js documentation on the topic was ultimately helpful but still a little disorganized and hard to follow. Plus, apart from understanding the theory, I struggled with writing my own facades and navigating the numerous options for adding JS library dependencies to a Scala.js build, of which the latter I am still not sure I actually understand.

Conclusion

This report describes my work on building a TASTy interpreter, including the background and motivation, the design of the interpreter, and several directions for future work. The interpreter unfortunately did not accomplish all of its original goals but did give rise to a useful subsidiary project, TASTyViz, a web-based TASTy visualizer, also described in this report. I also reflect on the challenges I encountered to try to mitigate them in the future, for myself or others extending this work or similar projects.

My personal motives behind this project, to learn Scala and familiarize myself with the Scala development ecosystem, were certainly achieved through this project. I got a hands-on introduction to Scala, TASTy, Scala.js, and many related tools and technologies, learning a lot more than what I thought I set out to learn. I also became familiar with the work of LAMP and the Scala Center at EPFL which was partly why I joined EPFL as a doctoral researcher and will certainly be useful for my future research. In conclusion, I would like to thank Sébastien Doeraene for being a very helpful mentor throughout this project and paying attention to all my frequent and frustrating issues, and Martin Odersky for providing general mentorship and feedback.

References

"An overview of TASTy." https://docs.scala-lang.org/scala3/guides/tasty-overview.html.

"TastyFormat.scala." *lampepfl/dotty* repository, GitHub, https://github.com/lampepfl/dotty/blob

/9de66c957d79739ec7bf23d237e61f3f60b4c96c/tasty/src/dotty/tools/tasty/TastyFormat.scala.

Martin Odersky and Nicolas Stucki. "Macros: The plan for Scala 3." *Scala Blog*, 2018-04-30, https://www.scala-lang.org/blog/2018/04/30/in-a-nutshell.html.

Sébastien Doeraene. "Implementing Scala.js Support for Scala 3." *The Scala Programming Language* website, 2020-11-03, https://www.scala-lang.org/2020/11/03/scalajs-for-scala-3.html.

"tasty-query." https://github.com/scalacenter/tasty-query/.

Martin Odersky et al. "Scala Language Specification – Version 2.13." https://scala-lang.org/files/archive/spec/2.13/.

"Scala 3 Reference." https://docs.scala-lang.org/scala3/reference/.

```
193:         SHAREDtype 108
195:         SELECT 29 [x]
197:           TERMREFsymbol 168
200:             SHAREDtype 119
202:       TYPEDEF(69) 40 [SimpleFooWithSuperFields]
205:         TEMPLATE(66)
207:           APPLY(12)
209:             SELECTin(10) 38 [<init>[Signed Signature(List(),testinputs.inheritance.S
212:               NEW
213:                 IDENTtpt 35 [SimpleFooSub]
215:                   SHAREDtype 171
218:               SHAREDtype 171
221:           DEFDEF(4) 6 [<init>]
224:             EMPTYCLAUSE
225:             SHAREDtype 55
227:           VALDEF(22) 41 [w]
230:             SHAREDtype 108
232:             APPLY(17)
234:               SELECTin(13) 44 [*[Signed Signature(List(scala.Int),scala.Int) @*]]
237:                 SELECT 29 [x]
239:                   QUALTHIS
240:                     IDENTtpt 40 [SimpleFooWithSuperFields]
242:                       TYPEREFsymbol 202
245:                         SHAREDtype 119
247:               SHAREDtype 108
249:               INTconst 7
251:           DEFDEF(20) 45 [y]
254:             SHAREDtype 108
256:             APPLY(15)
258:               SELECTin(11) 44 [*[Signed Signature(List(scala.Int),scala.Int) @*]]
261:                 SELECT 29 [x]
263:                   QUALTHIS
264:                     IDENTtpt 40 [SimpleFooWithSuperFields]
```

Figure 1: Sample output of `scalac -print-tasty` showing a TASTy tree.

Figure 2: Sample package navigation view in TASTyViz.

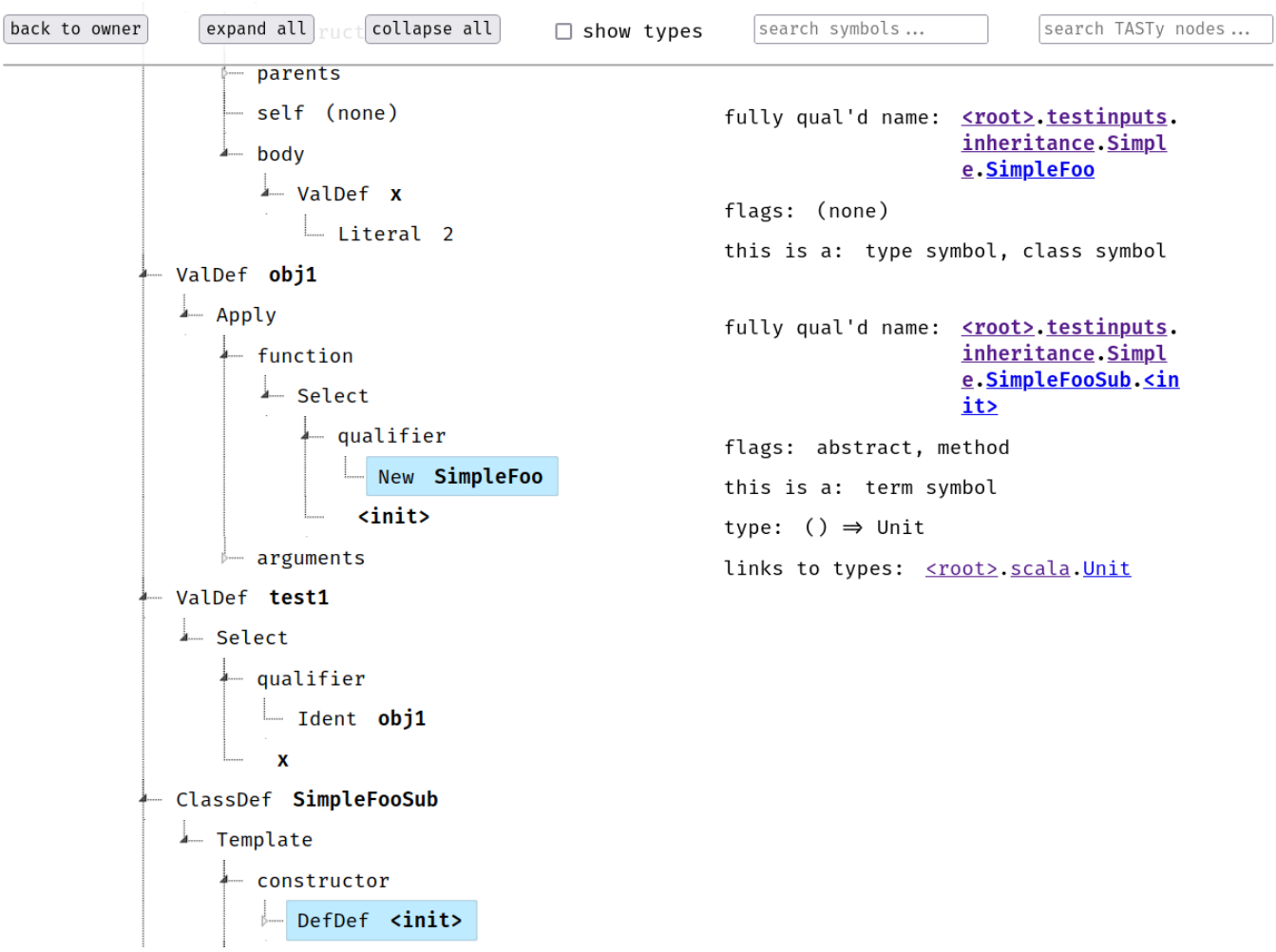

Figure 3: Sample TASTy tree view in TASTyViz.