Formal Semantics and Mechanized Soundness Proof for Fast Gradually Typed JavaScript

by

Ellen Arteca

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2018

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

As dynamic scripting languages are increasingly used in industry in large-scale projects, a need has arisen for more some of the convenient features of statically typed languages. This led to the development of *gradual typing*, a typing paradigm which is a compromise between static and dynamic typing. In gradual typing, programmers can specify type annotations if and when they choose to; then, at compile time, the statically typed sections of code are type checked. Gradual typing also guarantees that any runtime type errors will be caught when they cross the boundary from typed to untyped code, inserting type checks at runtime to ensure this. These runtime checks have the downside of adding significant overhead to the execution time, to the point where Takikawa et al. suggest it is practically untenable, in their paper [19].

Recent work has been done to develop faster implementations of sound gradually typed systems. In this thesis, we consider the work we presented in [15], for a fast gradually typed implementation of JavaScript, a popular dynamic scripting language. This thesis presents the formal semantics of this type system, and provides a mechanized soundness proof using the Coq proof assistant.

Acknowledgements

First of all, I'd like to thank Gregor for being the best advisor I could've asked for – every time we talk I learn something new, and it's fun just hanging out too. I look forward to continuing working with you.

Thanks Prabhakar and Patrick for being on my committee and taking the time to read my thesis. I appreciate the opportunity to learn from your feedback.

Alexi – thanks for everything. You're my best colleague, labmate, travel buddy, and my closest friend.

The rest of the PLG, especially Ifaz, Marianna, and Abel – thanks for good company and conversations. You kept the masters fun.

Par, and Emma – everyone needs friends like you.

This work was partially funded by an NSERC CGSM grant.

Table of Contents

Chapter 1

Introduction

As the scope of dynamic languages increases dramatically past that of basic scripting, partial static type checking has become an increasingly attractive tool to reduce the maintenance burden on large code bases. Users commonly find themselves in the position of using a dynamically typed language in a setting where some of the conveniences of statically typed languages would come in handy.

Gradual typing aims to address this by allowing the programmer to specify type annotations at their own discretion. This has the benefit of allowing modularity: for example, it enables statically typed code to seamlessly interact with dynamically typed libraries. In addition, if the code is being ported from untyped prototype to statically typed production code, gradual typing allows the code to be used throughout the translation process, rather than have it only work when fully statically typed.

On the more theoretical side, Wadler and Findler present some formal work in gradual typing, in which they conclude that if a runtime type error occurs it must be the fault of some untyped code [21]. This result also led to the development of the concept of *blame*, which is essentially a way to track which part of the code caused the type error. In gradual typing, runtime checks are added around the interaction points (the boundaries) between typed and untyped code, to ensure that any runtime type errors can be traced back to their point of origin in the program, and the appropriate section of untyped code can be blamed.

However, like any ideal-sounding solution, there is a downside. In gradual typing, the downside comes in the form of a prohibitive slowdown due to the overhead of these added checks. The slowdown is so bad that a recent paper, [19], suggests that sound gradual typing may be untenable as a concept due to the decrease in efficiency. In response to

this work, researchers have proposed a few approaches to try and minimize the overhead of these checks.

One of these approaches is the companion work to this thesis. [15] presents an approach to making sound gradually typed JavaScript more efficient, by leveraging information already computed in the virtual machine. This is a different implementation of work presented in [14] as SafeTypeScript; while that implementation of sound gradually typed JavaScript was prohibitively slow, the semantic changes in [15] resulted in a much lower overhead cost.

This thesis focuses on the semantics of the system presented in [15]. Now that we have a system whose implementation has been empirically validated, the formal semantics must be developed and the new type system proven to be sound. Some of the basic semantic changes are presented in [15], but only a brief overview, since the focus of that paper was on the implementation.

In this thesis, we present the full formal semantics of the sound gradually typed JavaScript presented in [15]; this is composed of the full grammar, typing judgment, and reduction rules of the language. We mechanized these semantics and proved soundness using the Coq proof assistant. The discussion focuses on the process of constructing the semantics, and what was involved with the accompanying proofs.

This work has some applications and avenues for future work. First and foremost, it is the first mechanized proof of any JavaScript derivative language. This means that anyone working formally over any JavaScript derivative now has a baseline in Coq should they choose to use it. It also means that any modification of the system presented here would be easier to formally prove, as it would just require a modification to this mechanization. There are also many potential applications of the implementation side of this work, or in implementing a similar system in other gradually typed languages which run on a virtual machine.

First, we will discuss the background related to this work.

Chapter 2

Background

Dynamic scripting languages such as JavaScript are seeing increasingly widespread use in industry. Now that there are massive codebases written in these languages, programmers find themselves wanting some features available in other languages that were designed for more large-scale programming. Of specific relevance to this thesis, are types: types are useful for a myriad of reasons, many of which apply in particular to large projects with multiple developers. This project focuses on gradual typing: adding the advantages of types to an untyped scripting language. In particular, we discuss some formal semantics for adding gradual typing to JavaScript, and the soundness proof for this system.

This section discusses the relevant background material in order to properly contextualize this work. Since this work focuses on JavaScript, we first discuss the basics of this language and some of its relevant derivative languages. Then, we go into virtual machines, gradual typing, nominal vs structural typing, and finally the different dynamic semantics.

2.1 A Brief Overview of JavaScript and Derivative Languages

JavaScript was the most widely used language on GitHub in 2017 [1]. It is commonly used in web applications, but it now takes on other roles, even being used in lieu of an object oriented language in some contexts, such as in the Unity game engine. As its use cases grew to larger projects, the potential benefits for the use of types grew too. The self-documenting nature of types, and the additional guarantees present in statically typed

languages are nice features which would be convenient to have in these larger JavaScript projects.

The idea of adding types to JavaScript is not new. TypeScript is a JavaScript derivative language used in industry – it is JavaScript with type annotations. The idea is as straightforward as it sounds: the programmer writes JavaScript code as usual, but now they can add types to their code. The main difference between TypeScript and other more traditional typed languages such as Java is that TypeScript is *optionally typed*. This concept is explained in detail below in Section 2.3.1, but basically it means that the type annotations are not required, and the user can add them if and when they choose to.

Executing a TypeScript program involves first a compilation phase, where the code is translated to the equivalent JavaScript. During this phase, any statically typed sections of code are type-checked, and all type annotations are removed. Then, the resulting JavaScript code can be executed. The benefit of this setup is that the resulting program execution is exactly as fast as the equivalent untyped JavaScript program, since by the time it starts running it is just the JavaScript version.

TypeScript's complete type erasure during compilation also means it does not catch any type errors at runtime. The typing paradigm equivalent to optional typing but where runtime type errors are caught when they are created is called *gradual typing*, and it is also explained in detail in Section 2.3. There was a version of TypeScript developed with these extra checks, presented by [14] as SafeTypeScript. This language is also compiled down to JavaScript during execution, but, unlike TypeScript, type checks are inserted into this JavaScript code to perform runtime type checks.

This thesis presents an alternate implementation of [14]'s SafeTypeScript, with different semantics. The relevant details of SafeTypeScript (henceforth STS) are discussed in relation to the semantics of our system throughout the thesis.

2.2 Virtual Machines and Efficient Implementations of Dynamic Languages

Many modern dynamic languages (such as JavaScript) run on virtual machines (VMs), and many traditionally slow dynamic languages can enjoy far improved performance running on a VM with Just-In-Time (JIT) compilation. JIT compilation allows for speculative optimizations, wherein sections of the code are compiled using information and assumptions from some number of previous runs of a previous section of code in order to perform

optimizations. One classic example of this is profiling with loops: when a loop is flagged as "hot" (i.e. will be executing many times), the JIT compiler enters a tracing phase where it traces and then optimizes the code running inside the loop. To ensure that these speculations are correct, checks are inserted, and, if failed, the code is then recompiled with the correct values.

Of particular relevance to this work is the use of "shapes" in many modern VMs for JavaScript (and other dynamic languages), based originally on work in Self [2]. This shape (commonly also referred to as "map" or "hidden class") represents the layout of field names in the position they occur in the object. In JavaScript, this information is not statically available like it would be in typed languages, and so this idea is very useful: the position in the shape is used to find the correct field value given the name, as the values of each field are positioned in the same order in an array referred to as the store.

Every object with the same fields in the same ordering has the same shape. This fact is then used to avoid a more costly dynamic field lookup by using inline caching – this refers to the shape of the object being cached, and then any field lookups amount to a simple array access at a fixed offset determined by the field's position in the shape.

All the shapes recognized by the VM are stored in a global shape tree, where the root is the empty object, and each child node represents a shape with an additional field. Finding a particular shape in the tree or adding a new one to the tree is trivial, as one simply follows the branches with the addition of every consecutive new field (recall that order of the fields is important, and if fields are not in the same order then the shape is not considered the same).

Since derivatives of JavaScript are structurally typed, the layout of the fields in an object are what make up the type itself. Therefore, the shape of an object has a direct analog to its type; this similarity is what we exploit when using the shapes to store and transmit typing information. Structural typing will be discussed in more detail below, but keep this fact in mind as it explains the motivation for using the shapes for typing information.

2.3 Gradual Typing

When comparing dynamically and statically typed languages, each has some advantages over the other. Dynamic languages offer more flexibility to the programmer: less overhead is required, and coding is typically easier and faster. On the other hand, statically typed languages are generally faster, as they can take advantage of type-based optimizations. The

type annotations also act as a sort of self-documentation, making the code more readable by providing more data on components of the program; and, type annotations also make for better error messages and allow for more errors to be caught at compile time instead of run time.

The goal of creating a paradigm which encompasses the advantages of both styles has led to the development of *gradual typing*. This is the idea that the programmer has the option of using type annotations: in the annotated code, where the types are known statically, type checks can occur at compile time; yet the optionality of the types makes the language fully backwards compatible with the untyped language.

This comes in particularly handy when considering the migration of code from prototype to full version. It is often easier and faster to code an initial prototype in a scripting language. Then, when moving to production version of the program, there are two major benefits to using a gradually typed language.

First, it means that there is no need for a translation to a new language. If the programmer had to rewrite the entire program from scratch to move from a scripting, prototype language to a totally separate statically typed language, this involves a thorough understanding of both languages and the program itself. An erroneous type annotation in the translation due to a misunderstanding of the function of a particular value in the prototype can lead to a cascade of errors which could potentially still result in executable code, while producing the wrong result (depending on the interactions of the types involved, for example using `int` in place of `float`). In addition, during the translation there is always room for error in terms of, for example, library calls which may seem equivalent but use different algorithms which may have divergent results due to accumulation error. Similarly, this applies to rounding error in general. Not all languages store values with the same level of precision or use the same methods to preserve significant figures, and any mismatch could result in different results even when the programs seem identical. This becomes particularly relevant when performing large-scale scientific computations, however, the same problem can apply to security related protocols in more common industry applications. Using the same language for both, with the simple addition of type annotations to ensure type soundness statically and provide more fodder for type-driven runtime optimization, will use the same libraries and low-end implementations for value storage. This guarantees consistency between the prototype and production version, with respect to these factors.

The second major advantage to using gradual typing is that then the translation is no longer "all or nothing". Here, this refers to the idea that with gradual typing, we are no longer constrained to the binary possibilities of either using the prototype or using the entire finished typed product. All intermediate stages of typing the prototype are

valid code in a gradually typed language. This is a major feature, as it allows code to be continued to be used even while being worked on, and allows for more intermediate testing to ensure accuracy during the translation process.

Apart from porting code from untyped to typed, gradual typing is also useful when we consider modularity of programs. The ability to divide the program into unrelated components where each can have its own level of "typed-ness" is very nice in a working environment with many different programmers. There are many instances where a user does not have access to a particular library but they want to make use of the functions provided, and without gradual typing there would be no way for the typed code to interact with these libraries (without creating a foreign function interface, which often have high overhead costs too) – this very true in TypeScript, as there are such a large number of commonly used JavaScript libraries. This is a much more usable system than requiring that the libraries themselves must be typed, especially when there is no guarantee that the user has access to the source code.

Now that gradual typing has been introduced as a concept and motivated, we will present the more specific details about what makes a typing system gradual, and what the difference is between optional and gradual typing.

2.3.1 Gradual vs Optional Typing

The most well-known (or at least, most widely used in today's industry) programming language with optional type annotations is TypeScript, a JavaScript derivative with types as discussed above. TypeScript is not a gradually typed language even though it has optional type annotations. The distinction between gradually and optionally typed is entirely in the presence or absence of runtime type guarantees.

In optionally typed languages such as TypeScript, the type annotations are used during compile time to catch static type errors; if there is a type error it is reported at compile time, but none of the type annotations have any effect on the compiled code if compilation succeeds. The output of a TypeScript compilation is pure JavaScript, such as one would code in JavaScript alone. Therefore, there are no runtime soundness guarantees in a TypeScript program, the same as in the equivalent JavaScript program.

Gradually typed languages add this extra runtime soundness guarantee. Ultimately, this amounts to inserting checks during the compilation phase so that there are type-checks at all the boundaries between typed and untyped code when the code is run, via a code injection of runtime checks. The previously existing gradually typed implementation

of TypeScript was SafeTypeScript as presented in [14]. As discussed above, SafeTypeScript ensures runtime type soundness by using this exact mechanism of adding runtime type checks at boundaries between typed and untyped code found during compilation to JavaScript.

While the additional safety guarantees of gradual typing make it seem like the ideal choice when comparing it to optional typing, this is not the whole picture. Recent work presented by Takikawa et al. in [19] tests the efficiency of performance of a gradually typed system (in this case, Typed Racket) to its fully dynamic equivalent, and found that the slowdowns were on the order of 100x, making the current implementation of Typed Racket so slow as to be practically unusable. The reason for this slowdown is simple: the introduction of the runtime checks. As such, and since the same implementation style is used in most current gradually typed languages, the findings of this paper prompt the authors to pose the question of whether or not sound gradual typing is even a viable option for large-scale use.

Since the publication of [19], there have been several works in response to this, working towards a more viable (i.e. faster) implementation strategy for gradually typed languages. The work presented here is one of these, and will be discussed.

2.3.2 Blame and the Gradual Guarantee

A formal representation of the typing guarantees of gradual typing defines it as a fully qualified typing paradigm. The *gradual guarantee*, formalized by [18], lists a number of requirements for a gradual type system to satisfy. To formally define gradual typing as a typing paradigm, we need a formal representation of the typing guarantees that this setup can ensure.

This guarantee states that "changes to the annotations of a gradually typed program should not change the static or dynamic behavior of the program". In other words, a well-typed program should evaluate to the same result independent of the number of (correct) type annotations present in the program. This is fundamental to the idea of gradual typing: that the actual functionality of the program remains unchanged, with the only effects being on the type checking, in the event of a program without any static or dynamic type errors.

More formally, the gradual guarantee can be stated as follows [18]. Here, when it is said that expression e_1 is "more precisely annotated" than expression e_2, this means that the set of type annotations on e_1 is a superset of those on e_2. Also note that $\vdash e : T$ means expression e has type T (where the type of an expression is the intersection of all types it has been annotated with).

Theorem 1 *Given expression e more precisely annotated than e', and $\vdash e : T$.*

1. *$\vdash e' : T'$ and T is more precise than T'*
2. *If e reduces to a value v, then e' reduces to a value v' which is less precisely annotated than v. If e diverges, then e' also diverges.*
3. *If e' reduces to a value v', then e either reduces to a value v which is more precisely annotated than v', or to some blame location l (in the event of a type error with T). If e' diverges, then e also diverges.*

Once this guarantee has been proven for a system, it signifies that any removal of type annotations will not change the runtime behaviour of the program (i.e. the same result will be produced, and there will not be any new runtime type errors). Note that this guarantee does not go the other way: since there is no way of ensuring that any new type annotations are actually correct, adding new annotations to a well-typed program does not mean that the new program is well-typed. This is equivalent to the third clause in the above theorem: if there was not a static type error and the new program with additional annotations is actually running, then if the original program produces a value the new program will either produce a more precisely typed value or will result in a runtime type error, thus returning a location to blame.

What is blame? Another crucial result in gradual typing is the statement that "well-typed programs can't be blamed", as stated by Wadler and Findler in their paper [21]. In this work, they adapt the concept of *blame* for contracts from [7]: A contract is merely a requirement that must be fulfilled, and authors in [7] define a framework in which contract violations could be *blamed* on some code. For example, a function `f` might have a contract asserting that its argument is a `string`, and returns an integer. If `f` was called with an untyped argument which ended up not being a string (i.e. does not fulfil the `string` contract), then the caller would be blamed for incorrectly using `f`. On the flip side, if `f` was found to return something of type other than integer, it would itself be blamed for not fulfilling its own contract.

In [21], Wadler and Findler apply blame from contracts to the casts in gradual typing. An important result from their work is a simple proof that, if a gradually-typed program goes awry, then blame always lies with the untyped code. This result allows us to make a meaningful statement of soundness for gradual type systems, even in the presence of dynamic code.

2.4 Nominal and Structural Typing

There are two main paradigms used in typing: nominal and structural.

In nominal type systems, types are made manifest through names and declarations, and membership for a given type and the relationship between types are determined by an explicit hierarchy which is declared by the programmer. This is the classic typing style for statically typed object-oriented languages, such as Java. As a demonstrative example, consider the following code snippet in Java-like pseudocode.

```
1 class A { ... }
2 class B { ... }
3
4 A x = new A();
5 B y = new B();
```

In this example, variables `x` and `y` are objects of type `A` and `B` respectively. These variables will have these types for their entire lifetime, and all characteristics (i.e. fields, member functions, interactions with objects of other types) are determined by what is present in the class definitions of their types.

In structural type systems, the structure of an object defines its type, and two objects have the same type if their structure is identical. This style of typing is most common when considering the types of values in dynamically typed languages, such as JavaScript and its derivative languages. It lends itself particularly well to object extension, which is a phenomenon not present in statically typed languages: object extension is where new fields can be added to an object over its lifetime, which means that its shape is not determined at construction. This is not possible in a nominally typed setting, as the structure of the type is determined at construction, but follows quite naturally from a system where the type is just a reflection of the structure.

As a demonstrative example, consider the following code snippet in JavaScript:

```
1 var x = {a : 1, b : 2};
2 var y = {a : 10};
3 // here, x and y have different types
4 y.b = 11;
5 // now, x and y have the same type
```

This example demonstrates object extension: when `x` and `y` are first declared, they do not have the same type as `x` has the extra field `b`. Then, once the object `y` is extended with the field `b`, it gains the same type as `x`, which is a pair of numbers.

This effectively showcases the difference between nominal and structural typing. There is also a difference in how these typing paradigms approach subtyping, and how they interact with each other when working in the same setting, which is showcased below.

2.4.1 Nominal and Structural Subtyping

Checking the subtyping relation between two nominal types is trivial: since the entire inheritance hierarchy must be explicitly declared, statically, in nominal typing, checking type equality or subtyping essentially amounts to a string comparison. Specifically, for nominal types `A` and `B`, if `A` <: `B` then this means that `A` must explicitly extend `B`, either directly or transitively. This requirement for explicitness in turn means that nominal subtype checks can be performed efficiently.

Structural subtyping on the other hand, while simple to explain and understand, is more computationally complex. There are two main approaches to subtyping in a structurally typed setting, which are both described below. Both depend entirely on looking at what fields are present and what their types are.

Depth Subtyping

This approach to subtyping considers the types of the fields which are present. It can be written formally as follows.

$$\frac{\sigma_i <: \tau_i}{\{l_i : \sigma_i \mid i \in \{1, n\}\} <: \{l_i : \tau_i \mid i \in \{1, n\}\}} \quad (\textsc{DepthSubtyping})$$

This rule specifies the conditions for a record type to be a depth subtype of another: all fields must be subtypes of their corresponding field in the other type, and both records must have the same number of fields.

Width Subtyping

This is perhaps the more intuitive approach to subtyping with structural types. Depth subtyping requires that each field present have the same type in both records, however, the record which is a subtype just has more fields. Although less confusing to word, we

still express it formally for more clarity.

$$\frac{}{\{l_i : \sigma_i \mid i \in \{1, n+k\}\} <: \{l_i : \sigma_i \mid i \in \{1, n\}\}} \text{ (WidthSubtyping)}$$

This rule specifies the conditions for a record type to be a width subtype of another: all fields must be the same type of their corresponding field in the other type, and the type which is a subtype must have a greater number of fields. This is the subtyping paradigm used in JavaScript and all of its derivative languages. As such, it is the subtyping strategy used in the work presented here.

Consider the following code snippet, as a demonstrative example:

```
1 var a = {x: 1, y: 2};
2 var b = {x: 3, y: 0, z: "hi"};
```

Here, `b` is a subtype of `a` since both objects have `x` and `y` number fields, but `b` has the additional `z` field which is a string.

Nominal and structural typing both have their relative pros and cons. Though nominal typing makes type checks fast and easy and generally increases readability of the code, the syntactic burden on the programmer is much larger than with structural typing, as the programmer needs to explicitly define all of the types they wish to use. As such, it would be convenient to be able to leverage both of these strategies to achieve the "best of both worlds" typing strategy.

SafeTypeScript already explores the interaction between nominal and structural types: it presents nominal classes, essentially equivalent to a standard nominal class setup, with the added complexity of needing to define the interaction between these new nominal types and the existing structural ones. Corresponding to every nominal type is a structural equivalent, representing the set of member fields and functions in the defined class (and all fields inherited from direct and transitive parent classes). The subtyping rules for the nominal/structural interaction are as follows: for nominal class `C` with corresponding structural type `C_s`, `C` is a subtype of `C_s`, but `C_s` is not a subtype of `C`.

2.4.2 With Regards to Gradual Typing

Many large-scale dynamic languages are structurally typed, simply due to the nature of dynamic coding (for features such as object extension, as discussed above). As such, the

same applies to most gradually typed languages. TypeScript for instance has classes, but these are just a wrapper for the underlying structural type of the class definition.

While structural typing has little-to-no syntactic burden, it complicates subtyping checks computationally, as we can no longer rely on the existence of an explicit named hierarchy, and need to comb over the object and possibly recurse into the object's fields in order to perform such a check. Since gradual typing involves the addition of runtime type checks, the slowdown discussed above is affected by the complexity of these checks.

Therefore, a potential approach to speed up gradual typing is to replace these structural type checks with nominal ones. This would involve building a gradual type system on top of a nominally typed language. In general, gradual typing is developed from dynamic languages, as the main point is to facilitate the transition from untyped to typed code (the transition in the other direction is not very in demand as there is not a practical use case). Dynamic languages are generally structurally typed, so gradual type systems with nominal typing are uncommon.

However, some gradually typed languages have introduced nominal typing [16] [11] to address the performance problem inherent to the runtime checks. With nominal types, runtime checks can be significantly less expensive, as it is not necessary to recursively or lazily check object fields.

In short, nominal typing achieves better performance at the cost of making more work for the programmer (with the additional syntactic burden), and structural typing offers more flexibility at the cost of performance. The work presented here addresses the runtime check slowness in another way (by leveraging information already computed in the VM during JIT compilation to elide runtime checks), while maintaining the structural type system.

The next section discusses various semantics of the runtime type checking, to ensure type soundness.

2.5 Dynamic Semantics

Vitousek et al. developed a gradual type system for Python, presented as Reticulated Python [20]. The authors also detail three approaches to dynamic semantics which ensure runtime soundness of a gradually typed language. Essentially, this amounts to runtime checking of values to ensure that they have the expected type, and that type errors are all caught at expected locations. The three options presented by these authors are monotonic, transient, and guarded semantics, as described below.

2.5.1 Monotonic Semantics

In monotonic semantics, types can only become more specific as the program executes. This is the simplest and most naive way to implement sound runtime type checking: casts are inserted at implicit coercion sites, and each cast permanently alters the type of the object or function in that they must abide by the type they are cast to for the remainder of their lifetime. To put it in other words, once something has been used as one type it must satisfy this type forever, and anything violating this type will be a runtime type error.

This has one specific problem: if an object is being used as a particular type in one scope and then it exits this scope, then it must still satisfy that type even though it is no longer relevant, as shown below.

The simplest case is for function arguments. Consider the following demonstrative example.

```
1 function sum(x : List<number>) {
2     ...
3 }
4 var y : List<Any> = [1, 2];
5 sum(y);
6 y.push("ohno");
```

At first glance, it does not appear that this should cause any errors. When `sum` is called, the `y` list has only numbers, and so the cast to `List<number>` which occurs when the function is called passes with no errors. Then, after the call to `sum`, a string is added to `y` - this should also be fine, since `y` is a list of `Any`. However, since monotonic semantics are being used here, `y` must continue to be a list of numbers for the rest of its lifetime after the call to `sum`. So, this addition of a string violates the `List<number>` type and causes a runtime type error.

However, while monotonic semantics is much more restrictive than guarded or transient, it does have the advantage that it has better performance, as the runtime overhead is lower. We discuss the relative advantages and disadvantages of each system in a summary at the end of this section.

At this point we should note that monotonic semantics are those used in this work. This is described in more detail in Chapter 3, however the reasoning is that it is the simplest and most performance-efficient of the existing dynamic semantics.

2.5.2 Guarded Semantics

Guarded semantics uses casts inserted at implicit coercion sites to catch runtime type errors. A cast between the dynamic (`Any`) type and base types will just return the value, or raise an error if there is a type violation. However, casts between object or function types produce a proxy – essentially, this is a wrapper around the cast which holds blame information in the form of the line at which the cast was added. Then, when the cast object or function is accessed it is done through the proxy, which ensures that it is being used correctly. In general, errors are caught when an object is first misused, typically when an incorrect assignment or mistyped call occurs.

If there are nested casts, there are not also nested layers of proxies: to avoid excessive overhead, a proxy represents a compressed sequence of all the casts on an object or function. The proxy also carries the blame information for every type it must maintain so as to adequately report the source of a type error.

A code example similar to the one above is used to demonstrate where errors are caught in guarded semantics; this also shows the difference between guarded and monotonic.

```
1 var z : List<Any>;
2 function sum(x : List<number>) {
3     z = x
4 }
5 var y : List<Any> = [1, 2];
6 sum(y);
7 z.push("ohno")
```

The example exactly as it was in the monotonic semantics section would not result in any errors in guarded semantics, since the mistyped field is never accessed. However, with the slight modification of the addition of `z`, now there is a runtime type error when the string is pushed to `z`. This error is caused because `z` gains the `List<number>` proxy when it is assigned to `x` in the function, and thus the addition of the string is a violation of the conditions of that proxy.

2.5.3 Transient Semantics

Similar to guarded semantics, transient semantics inserts type checks at implicit coercion sites, however there is no proxy used. Instead of wrapping the object or function being cast in a proxy, transient semantics just checks shallowly that the type is matched and then returns the value. This “shallow” check amounts to checking if the value being cast

is of the correct primitive type: if this is a base type then a mismatch will result in a type error, if it is an object then the check just ensures that it is actually an object.

With just these checks in place, the system would not be sound. So, to ensure soundness, type checks are inserted at the use-sites of fields or variables with non-base types (i.e. function or object types), and at the entry point to function bodies to check the parameter types. Since there is no proxy present, it is up to the object or function itself to check the soundness of its fields.

The result is that one can update an object with an incorrectly typed field as long as there is no primitive type violation. Then, this error will not be discovered until this erroneous field is accessed. This is still sound, as no type error resulting from the field can result unless the field is actually used.

Recall the code example used above in the discussion of guarded semantics. Here there will be no runtime type error with transient semantics, as both the `y` and `z` lists have type `List<Any>` and no information is retained from the cast to `List<number>` local to the `sum` function so adding a string to `z` does not cause any errors.

To recap, we will compare and outline the differences in what the various dynamic semantics have to offer. Monotonic semantics have the fastest execution time, since there is the lowest overhead. However, they also have the most restrictive criteria for type checks to succeed – more potential errors are caught with the monotonic semantics than with the others, due to the type obligations never being removed after they are out of scope. Guarded semantics have the most overhead, due to the presence of the proxies. Their error catching is in the middle between monotonic and transient semantics, considering some cases errors that transient does not, but also more forgiving than monotonic semantics. Transient semantics is the most lenient in terms of catching type errors the latest, while still retaining type soundness, and is middle of the road in terms of overhead load from the checks.

Monotonic semantics are used in the current implementation of HiggsCheck; however, recent work building off of HiggsCheck is introducing a new form of semantics which is less restrictive than monotonic and more so than transient. This is presented in more detail in Section 6.2.

Chapter 3

HiggsCheck

This chapter discusses the work we presented in [15]. This paper presents the implementation paired with the theory presented in this thesis. It provides the bulk of the motivation and some relevant background on the actual semantics work, and is worth presenting in some detail here.

3.1 Is Sound Gradual Typing Dead?

The main barrier currently between gradually typed languages and industry usability is the performance hit caused by the runtime checks necessary for preserving soundness, as discussed in Section 2.3.

These runtime type checks are particularly slow when working over structural types. Many dynamically typed languages are structurally typed, as discussed in Section 2.4; structural typing lends itself well to lazy object construction (i.e. object extension, where the some fields are added to an object after the initial construction). In this case, although it is a much more convenient typing style for these languages, it has the disadvantage of increasing the complexity of type checks from the simple string comparison required with nominal typing to a recursive check over all the fields of an object.

The initial STS implementation also suffered from this slowdown. This is probably a large reason why STS never caught on – industry uses for TypeScript rely on programs being able to run in a reasonable amount of time, thus making STS unusable.

Clearly the optimal system would be a sound gradually typed TypeScript which could run in a reasonable amount of time. The work presented in [15] takes a step towards that,

in implementing alternative semantics for STS, employing a strategy for removing the need for some of the runtime checks previously required. This strategy introduces object contracts, and makes use of the type checks already present in the VM computations. The approach is discussed in detail below.

3.2 SafeTypeScript

As mentioned in Section 2.1, STS is a JavaScript derivative language. Developed and presented by [14], it is the first gradually typed version of TypeScript. The changes to TypeScript implemented by this system just amount to the runtime check insertion required to ensure soundness in a gradually typed system.

The main contributions of [14] are the semantics of STS (the implementation is also provided, but the work in [15] was built off of base TypeScript). These semantics formalize TypeScript and the conditions under which the runtime checks are added, in addition to the changes in the grammar required to include and perform these checks.

The actual semantic differences between the original implementation of STS and the version of the semantics presented here is discussed in detail in Section 4.2 below, since this is the main focus of this thesis. At a glance, however, the main semantic feature of interest with respect to the type checks is the *tag heap*: this is a heap which parallels the execution heap, and stores runtime type information (RTTI) for each element in the heap as it become available. In our version of STS, the tag heap is eliminated, as the typing information is stored on the runtime objects themselves (i.e. on the heap itself, and no type information is required for primitive values), and there is no more need for anything to parallel the heap. The auxiliary functions for checking tags are also completely replaced by new auxiliary functions referencing the contracts.

Now we will discuss how the VM is used to aid in the implementation of this version of STS.

3.3 The Higgs VM

We have already discussed some basic background on VMs in Section 2.2. For the STS implementation, the specific VM used was the Higgs research VM [3] [4], a VM for JavaScript with competitive performance. Higgs implements many modern JavaScript optimizations, as well as a few unique ones which we utilize.

In particular, Higgs implements speculative optimizations through *lazy basic block versioning*. In lazy basic block versioning, each basic block is compiled speculatively with assumptions about the types of data that it accesses, and those assumptions are checked before execution is allowed to proceed into such a speculatively compiled block. Single basic blocks may have multiple compiled versions, corresponding to different sets of assumptions, and execution will choose which block to proceed into through run-time checks. When checks of those assumptions fail, a new version of the basic block and all following basic blocks is built to fit the new assumptions. These assumptions propagate into further blocks, so rechecks are elided. Thus, the compiled code for a function becomes a graph of basic block versions, each representing the compilation of a basic block under a particular set of assumptions, like a decision tree. To avoid code blowup, limits are placed on how large this graph can get, but they are rarely reached in practice.

Consider the following TypeScript code for calculating the sum of a linked list:

```
1  interface Node {
2      next: Node;
3      item: number;
4  }
5  function sum (node: Node ) {
6      if (node.next )
7          return sum(node.next ) + node.item;
8      else
9          return node.item;
10 }
```

A direct compilation of function `sum` to basic blocks, with control flow replaced by gotos, is:

1. `if (!node.next)` **goto** 4
2. `tmp1 = sum(node.next); tmp2 = node.item;`
3. `return tmp1 + tmp2;`
4. `return node.item;`

Blocks 2 and 3 must be split, because the compilation of the + operator depends on the types of its operands, creating implicit control flow for those types to be checked.

Upon the first compilation of this function, as block 1 requires the shape of `node`, the shape seen will be cached, resulting in a compiled procedure as in Figure 3.2. Assuming

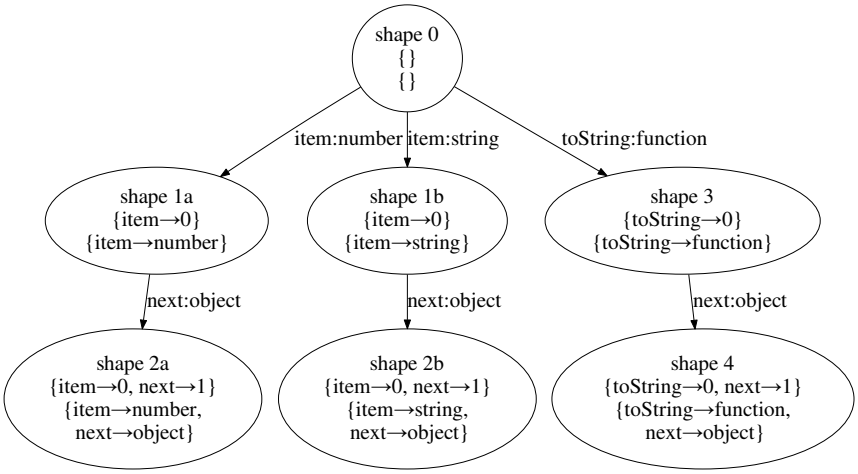

Figure 3.1: Example of Higgs shape tree

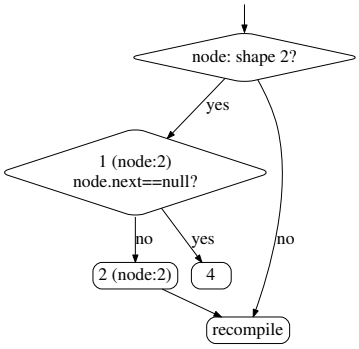

Figure 3.2: First compilation of `sum`.

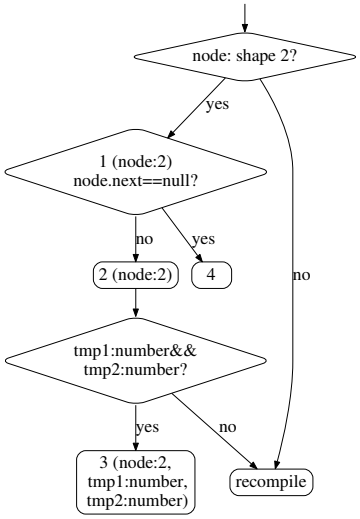

Figure 3.3: Second compilation.

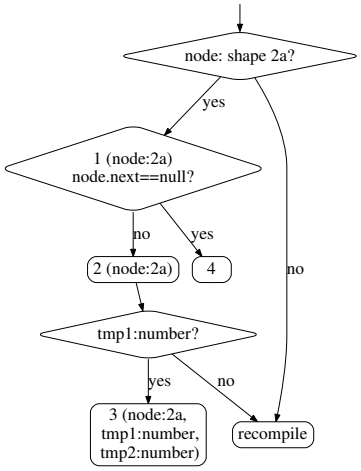

Figure 3.4: With typed shapes.

the function is called with a non-null `node.next`, it will proceed through block 2, and then be compiled again, into a procedure as in Figure 3.3.

Higgs, building on work in TruffleJS [23], extends shapes by encoding member types into them as well. Objects built of the same fields but distinct primitive types of their members will have distinct shapes, and objects with identical layouts and types have the same shape. This is accomplished by extending shapes with a map of names to primitive types. The transitions in this extended shape tree are labeled with both a name and a primitive type. The relevant shape tree for the code example above is shown in Figure 3.1.

For instance, in Figure 3.3, if field `item` is added to an empty object with type `number`, the object's shape is updated to shape 1a. If it's added with type `string`, the object's shape is updated to shape 1b. Because fields are typically mutable, this means the shape can also change when a field is modified: If an object with shape 2a's item field is updated to a string, the object's shape must be updated to 2b. That is, the shape is only a snapshot of the current state of the object, not a guarantee of future state.

The benefit of adding types to shape is twofold: First, as primitive types are available in the shape, it's unnecessary to tag or box values in the heap [22], which removes the need for the tag heap from the original STS implementation (which existed for this purpose, to store a list of types corresponding to their locations in the runtime heap). Objects can thus be laid out more efficiently, avoiding wasting bits on type information that's identical between many objects. Second, speculating on a shape allows the compiler not only to optimize field access, but to eliminate type checks over the accessed field values: The check of the containing object's shape subsumes the check of the accessed value's type. For instance, the `sum` function can be compiled to the procedure in Figure 3.4 with typed shapes, avoiding a check for the type of `tmp2`, which is implied by the shape of `node`. This optimization is important to the goal of making a faster implementation of STS (going from orders of magnitude to an average slowdown of 6.61% across the tested benchmarks [15]): if a tag in the STS tag heap obliges the item field of an object to be a number, and that object's shape is shape 2a, that obligation cannot fail, and so does not need to be checked.

3.4 Object Contracts

This work replaces the tagging system in the original STS implementation by the contract system presented here. As explained above, contracts are a way of storing runtime type information on the relevant object on the heap.

The contract is a set of obligations over a value. For primitive values, the contract is just the type (e.g. `string`, `bool`, etc). For object values, the contract is a list of obligations over the fields and member functions, and the primitive obligation (which lists whether it is an instance of a class, struct, or method). These semantic details are all explained more fully below in Section 4.3. This section also discusses how the contracts are added to objects, and how they are checked. Contracts are added lazily – this means that on contract addition the fields are not recursively parsed, only the primitive contracts are added to each field at this point, and the relevant contracts are added to the fields on access. This is done through auxiliary functions, which are also discussed in the same section.

In terms of the implementation of the contracts, contracts in HiggsCheck are objects in the VM, exposed to the programmer only as opaque integer handles. Every contract in the system has such a handle. Basic, built-in contracts are obtained through a built-in function `contract_for`, which simply maps string contract names to their handles. These contracts are created by the VM during initialization and cannot be changed. The remaining built-in functions create and modify contract objects in the VM and apply them to objects.

Every higher-order contract has a parent contract, and inherits all of the parent's obligations. Each link in this chain adds a single obligation, and the contract is the sum of all of the obligations in this chain. `contract_oblige_member` and `contract_oblige_return` are provided to oblige fields and return values, respectively, and each takes a parent contract as an argument, and returns a new contract. The new contract has all the obligations of the parent contract, plus one new higher-order obligation.

To support the definition of recursive contracts, contracts are mutable by default. An obligation over a field or return may be replaced by `contract_reoblige_member`. `contract_reoblige_member` follows the chain of parent contracts until reaching either an immutable contract or the correct obligation. If the correct obligation is found, it is updated. `contract_freeze` explicitly freezes the contract and all of its parents, and unfrozen contracts are frozen automatically if applied to objects, to assure that live contracts do not change.

For example, `Node`'s contract can be written as:

```
1 var c = contract_for ("object");
2 var cn = contract_for ("number");
3 c = contract_oblige_member (c , "item", cn);
4 c = contract_oblige_member (c , "next", c);
5 contract_reoblige_member (c , "next", c);
6 contract_freeze (c);
```

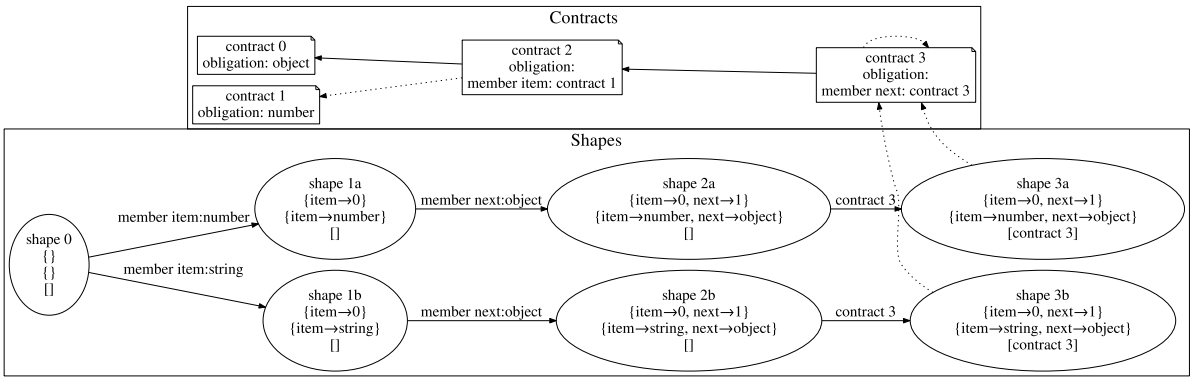

Figure 3.5: Run-time state with shapes and contracts.

The contract `c` at line 4 has its next field obliged to the contract generated at line 3. That is, it is not recursive: Its next field is not required itself to have a next field. Line 5 establishes this contract's necessary recursivity, and line 6 makes it immutable.

Figure 3.5 shows a complete shape tree and contract trees for a system in which objects with shapes 3a and 3b are contractually obliged to behave as `Node`s. Dotted lines in Figure 3.5 indicate contractual obligations, and solid lines indicate the parent/child relationships in the shape and contract trees. An object with shape 3a, for instance, has an `item` field of type `number` and a `next` field of type `object`. Further, objects with shape 3a are contractually obligated to have `item` fields of type `number`, by way of contract 3, and `next` fields which also conform to contract 3, by way of contract 2.

More implementation details are given in the paper [15], but are not included here as they are not fundamental to the formal work presented; the brief overview focusing on type shapes presented here covers the topics most important to this thesis. Next, we will discuss the results of the implementation.

3.5 Benchmark Results

The HiggsCheck implementation of STS was much faster than the original STS implementation by [14], as expected. To test the speed, we used a set of benchmarks including those used to test original STS, and the TypeScript compiler itself. The other benchmarks were ported from the Typed Racket benchmarks in [19], so as to be able to talk specifically about those used to previously conclude that sound gradual typing may be dead.

The numerical results over all the benchmarks and the details on how the tests were performed and recorded can be found in detail in [15]. However, the overall results are presented here to show the improvement made: the slowdown (as compared with the corresponding purely untyped code) for HiggsCheck STS was 7% in the typical case and no worse than 45% in the worst case, which is a vast reduction from the previous state of the art, which can slow down execution 10s or 100s of times. This shows that the contract system developed and coupled with the VM provides a much more reasonable execution time for gradually typed TypeScript programs, thus breathing new hope into the "life" of sound gradual typing.

Since the implementation is viable, and has potential to be used in industrial applications, formalizing the semantics and proving soundness of this type system becomes more relevant. We want to ensure that this system is well understood, can be reasoned about, and actually works.

Chapter 4

Formal Semantics

The work presented here is essentially an alternate implementation of SafeTypeScript [14] (henceforth STS). The primary distinction between this work and the original STS implementation is the introduction of the contract system. In this section, we will explain the semantic differences between our implementation and the original STS work, at both a higher level and in detail. Before we go into the specifics of each language, we will first go over an overview of the languages.

To run an STS program, the code is first compiled down to JavaScript, and then this JavaScript is what gets executed. This is almost the same as how TypeScript programs are run, with the key difference that there is no longer type erasure at runtime, since the gradual typing of STS means that runtime checks must be inserted during the compilation. The method by which these runtime checks are added during compilation is by the use of *auxiliary functions* inserted by the compiler; these are functions which the user does not have access to, that perform the runtime checks as specified. The auxiliary functions present in original STS differ from those in our implementation, and these differences are highlighted below.

The language itself is made up of a *typed grammar* and a corresponding *untyped grammar*. The typed grammar represents the subset of the language that the user has access to, and what is actually considered STS itself. Then, when executing an STS program, the resulting generated JavaScript conforms to the untyped grammar. This grammar is not just the grammar of JavaScript: it also includes all the auxiliary functions required for the type check generation.

In terms of the operational semantics, STS has a typing judgment and a set of reduction rules. The typing judgment is a bit different than in most languages, as it is used to

represent the compilation from the typed to untyped grammar. This is explained in detail below.

The reduction rules correspond to the actual execution of the program, and therefore are defined over the untyped grammar as well. There are reduction rules for the auxiliary functions, to show what actions they perform at runtime and how this affects the running environment.

4.1 Runtime Soundness Checks in Original STS

As stated above, the difference between STS and TypeScript lies in the soundness guarantees: TypeScript is an optionally typed language, while original STS and HiggsCheck STS are both gradually typed. To add the gradual guarantee of runtime soundness, typechecks must be introduced to the system which are not present in TypeScript.

The naive approach to ensuring runtime type soundness when using monotonic semantics would be to add a runtime check at every boundary between typed and untyped code.

For example, consider the following code snippet in TypeScript:

```
1 var uPoint = {x:1, y:2};
2 var tPoint : Point;
3 tPoint = uPoint; // assigning untyped value to a typed value
```

Here we see that the boundary between typed and untyped code is being crossed in the assignment of `uPoint` to `tPoint` at line 3. `tPoint` is a `Point`, and thus when `uPoint` is assigned to it, it must also satisfy the `Point` type. However, since no type is stated for `uPoint`, at its definition on line 1 it cannot be checked statically. So, to ensure runtime type soundness a sound gradually typed system must add a check to see if `uPoint` actually fits the type `Point`.

The STS implementation presented in [14] follows this naive implementation approach. During the compilation from STS to JavaScript, checks are added which run during the execution of the JavaScript. These checks use auxiliary functions and a tag heap. The tag heap is a heap parallel to the regular execution heap which stores the current known type information about each object in the heap. This amounts to the runtime type information (RTTI) for each object.

The idea of the tag heap is to act as a runtime repository for the type information of objects. Then, when a runtime type check is induced (on the boundary between typed

and untyped code, as explained above) the checks reference the relevant entries in the tag heap.

Once an object has been created, it can be considered as having various different types, depending on its circumstances, via the power of casts. This includes explicit casts, or implicit casts if the object is passed into a function with typed arguments.

A simple example of an implicit cast is an object which is passed in as a parameter to a typed function. Consider the following code snippet.

```
1 typedFun(x: B) {
2   ...
3   ...
4 }
5 var b : A; // at this point, b has type A
6 typedFun( b); // during the call to typedFun, b gains type B
```

When `b` is passed into `typedFun`, it is implicitly cast to type `B`. Since STS has monotonic semantics, once the new type information is gained, it is never removed – the type of `b` can only get more specific, and at every point must satisfy all of the types it is tagged with. Therefore, at the end of this code snippet, `b` is obliged to correspond to type `A`$\cap$`B`.

As such, the type of a value is not fixed over its lifetime, and is prone to evolution over the course of the program. Therefore, it is natural that there needs to be a system in place to update the type of a value, so the RTTI is always current. Since type information in STS is stored in the form of tag information in the tag heap, this type update amounts to updating the tag information for a particular location in the tag heap.

This is where *auxiliary functions* come into play: these are injected functions which the user does not have access to, and which are inserted into the compiled code (in this case, the JavaScript compiled from the STS source). There are three primary auxiliary functions central to the tagging and checking as described in the STS semantics. The actual details of these are not important, but we present the idea of each one to draw parallels to the auxiliary functions we add in our own implementation.

The two functions involved with updating the RTTI are `shallowTag` and `checkAndTag`. `shallowTag` updates the tag of the specified location with the specified type. It is named `shallowTag` since it does not actually check anything apart from the compatibility of the tags being combined; there is no recursion into the structure of the object (if the value being tagged is an object). Then, it inserts checks for each object field, to be evaluated lazily on access. `checkAndTag` is similar to `shallowTag` in that it updates the RTTI of a location on the tag heap; the key difference is that while `shallowTag` does not recurse

into the structure of the object, `checkAndTag` goes through all layers of the object to catch potential type errors. Finally, all the tag updates are performed through the use of the `comb` function – this combines two tags into one intersection type (or returns error given incompatible types).

Apart from these main auxiliary functions, there are a few more in STS, namely `read` and `write`, which are used when reading from or writing to a field of an object. These functions ensure that the field being accessed is actually present on the object it is being called on, and then calls the appropriate auxiliary function for updating the RTTI. There is a similar auxiliary function for calling a method with a list of arguments, `callMethod`, which checks that the method exists and performs the appropriate tagging (in this case, a `checkAndTag` on each of the method arguments with the type according to the method definition, and a `shallowTag` corresponding of the whole method call to the return type). These functions are essentially just wrappers for the more fundamental ones listed above.

The reasoning behind using `shallowTag` vs `checkAndTag` in instances in STS is not due to some nice theoretical result; in the STS paper the authors describe their choices in this regard as just those minimizing the runtime overhead, as verified by some empirical testing (in some cases eager checking is more efficient than lazy checking). In fact, only `shallowTag` is necessary to preserve soundness; if every tag operation was replaced with `shallowTag`, this would just perform all the checks lazily. Rather than failing on a `checkAndTag`, an error be raised when the relevant field was accessed; similarly, if all tagging was replaced with `checkAndTag`, errors would be caught earlier, at a performance cost.

This fact that soundness is preserved even if only `shallowTag` is used is also very relevant to the system implemented in HiggsCheck STS. The contract system implemented there employs the delayed check strategy. This is described in the following section.

4.2 Key Semantics of HiggsCheck

In essence, the semantics of STS with HiggsCheck are equivalent to those of the original STS. The majority of the grammar rules remain unchanged, and many of the type compilation and reduction rules are also the same. The key differences will be highlighted below, and the entire semantics is included in Appendices A and B.

The typed grammar is displayed below. The majority of the rules are identical to those in the original STS implementation, albeit with some small changes in notation. The primary change was the addition of expression rules for the auxiliary functions added, and this is discussed below.

One note about the types is that although Top and Bottom are not explicitly stated there are types which are equivalent to these. The **void** type effectively acts as the Bottom type, in that when `undefined` (i.e. the error literal) is assigned a type, it is this type. The dynamic type `Any` (i.e. the type assigned to all untyped terms) acts as Top, in that every other type is a subtype of `Any`, and can be cast to it without errors.

Field name	f	::=	*field name identifier*	
Method name	m	::=	*method name identifier*	
Class name	C	::=	*class name identifier*	
Interface name	I	::=	*interface name identifier*	
Primitive type	t_c	::=	**number**	*number type*
		|	**string**	*string type*
		|	**bool**	*boolean type*
		|	**void**	*void type*
Type	t	::=	t_c	*primitive type*
		|	$\{M; F\}$	*struct type*
		|	`Any`	*dynamic type*
		|	C	*nominal class type*
		|	I	*nominal interface type*
Field decl	F	::=	$\overline{(\ f, t)}$	*field declaration*
Meth decl	M	::=	$\overline{(\ m, t_m)}$	*method declaration*
Meth type	t_m	::=	$\overline{(\ t)}\ , t$	*method type*
Literals	v_c	::=	*number literal*	
		|	*string literal*	
		|	*boolean literal*	
		|	**undefined**	
Location	l	::=	`nil`	*invalid location*
		|	n	$n \in \mathbb{N}$
Var	x	::=	*variable identifier*	

Interface defn	$\widehat{I}$	::=	`baseI`	*root interface*
		|	`interface` I `implements` $\overline{I_i}$ $\{M; F\}$	*I with body*
Class defn	$\widehat{C}$	::=	`baseC`	*root class*
		|	`class` C `extends` C'`implements` $\overline{I_i}$ $\{\widehat{M}; F\}$	*C body*
Meth Defn	$\widehat{M}$	::=	$\overline{(\ m, t_m)}$; $\overline{(\ \overline{(\ x, t)}\ : t \{s;$ `ret` $e\})}$	*method defn*
Field defn	$\widehat{F}$	::=	$\overline{(\ f, t)}$; $\overline{e}$	*field defn*
Val	v	::=	l	*location*
		|	$t_c\ v_c$	*primitive*

Exp	e	::=	v	*value*
		|	x	*variable*
		|	$\{\widehat{M}; \widehat{F}\}$	*struct*
		|	`new` $C(\ \overline{e_i})$	*new class*
		|	$e.m(\ \overline{e_i})$	*method call*
		|	$e_1[\ e_2]\ (\ \overline{e_i})$	*dynamic method call*
		|	$e.f$	*field access*
		|	$e_1[\ e_2]$	*dynamic field access*
		|	$< t > e$	*explicit cast*
		|	$\{l;\ \overline{\widehat{M}};\ \widehat{F};\ \overline{t}\}$	*rep'n of runtime object*
		|	`contract_assign`$(\ e, t)$	*assign type contract t to e*
		|	`check_contract`$(\ e, f)$	*check f field contract on e*
		|	`check_contract`$(\ e, m)$	*check m method contract on e*

Stat	s	::=	e	*expression*
		|	`skip`	*no op*
		|	$s_1\ ;\ s_2$	*sequence of statements*
		|	`var` $x : t := e$	*variable assignment*
		|	$x := e$	*variable reassignment*
		|	$e_1.f := e_2$	*field assignment*
		|	$e_1[\ e_2]\ := e_3$	*dynamic field assignment*
		|	`if` $(\ e)\ \{s_1\}$ `else`$\{s_2\}$	*if statement*

The grammar above represents the typed grammar of our implementation of STS. During the compilation to JavaScript we see the addition of the contracts and subsequent

checks (representing the type checks required to make the language gradually typed). Parts of the grammar only relevant during the untyped runtime are described later in this section.

The key semantic differences between the original STS and the grammar presented above is the inclusion of expression rules for the `check_contract` and `contract_assign` auxiliary functions. The grammar presented here represents the entire grammar of the language, of which the programmer can only code with a subset, since the auxiliary functions are included in the grammar but the programmer cannot directly use these. The reason for the auxiliary functions being unavailable is that their inclusion would allow for the programmer to add their own type checks wherever and whenever they chose; this in turn would add restrictions to the typing of the program which may not agree with the actual typing of the code. In particular, the user could add a check for a type which is not satisfied, thus resulting in a type error when in fact the type violation was due to an incorrect check added by the user; this would in turn compromise the validity of any claims made about contracts representing the type annotations.

The expression for the typed representation of a runtime object is also not available to the programmer. It would not make sense for the user to have access to the runtime heap – the backend functionality of the runtime environment remains opaque to the programmer. This rule exists in order to allow for typing of the rule for reduction to something to add to the runtime heap (to allow for preservation). Its use in the proof is discussed in more detail in Chapter 5.

4.2.1 HiggsCheck STS Auxiliary Functions

This section explains the functionality of the HiggsCheck STS auxiliary functions in detail. These are the `contract_assign` and `check_contract` functions, which are included in the grammar as expressions.

First we will discuss the `contract_assign` function. This function is added at a cast point in the code, either an explicit or implicit cast. The explicit cast is the simplest case, in which an expression is cast to a user-specified type, taking the form $< t > e$ for some expression e and type t. Implicit casts occur at points where an expression is considered to be of a particular type by its context in the program. The simplest example of this is when an expression is passed in as an argument to a typed function. This results in an implicit cast to the specified type of the corresponding parameter of the function. For example, consider the code snippet below.

```
1 class Point {
2         x:number;
3         y:number;
4 }
5
6 var p = {x:1, y:2};
7
8 function fun(q : Point) { ... }
9 fun(p);
```

In the call to function `fun` at line 9, variable `p` is cast to type `Point`, since it is the actual parameter for the `q` formal parameter.

Another example of an implicit cast location would be expression assignment to a typed variable (or field in a typed object). In these cases, the assignment requires that the expression satisfy the type it is declared to have.

During the compilation phase from typed to untyped code (i.e. from STS to JavaScript), a `contract_assign` call is inserted at these implicit and explicit cast locations, as a wrapper for the expression being cast. At runtime, this function has the following behaviour (for `contract_assign`(e, t) with expression e and type t), once e has been reduced to a value:

1. Check that the type of e is the same primitive type as t. This amounts to either both of them being object types (in which case the primitive type is just object), or, if e is a base type, then the types must exactly match. If there is not a match, a runtime type error occurs.

2. If t and e had corresponding base types, then the check is complete, and code execution can continue on after the check.

3. If t was an object type and e corresponded to an object, then the primitive contract check has passed. At this point, a contract representing t is added to e. This involves converting t into a sequence of corresponding obligations, through a call to the function `toContract` which will be described below.

4. Adding the contract to e is done once e has been fully reduced to a runtime object on the heap, adding to the list of contracts present.

Note that there is no recursive checking of the contracts. This means that there could be type errors which are not caught immediately. Errors in fields (or components of objects nested in their structure) are caught once those components are accessed.

This lazy checking is implemented through the use of the other primary auxiliary function, `check_contract`. While `contract_assign` is inserted during the compilation of a cast, `check_contract` is added during the compilation of object member accesses, to either a field or a function. At runtime, this function has the following behaviour (for `check_contract`$(e, m \mid f)$, for expression e and either field f or member function m), once e has been reduced to a value:

1. If e is a primitive value, then this returns a type error, since any member access to a primitive is impossible.
2. If e is reduced to an object, then first there is a check to see if the member being checked is actually present. If not, then there is a type error as the object shape does not match its requirements.
3. If the member is present on the object, then the list of contracts is parsed to see if there are any obligations on this member. If not, then the work of `check_contract` is done, and the value of the field is returned.
4. If there are obligations on the member being checked, then these obligations are applied to the member with calls to `contract_assign`. The functionality of this is as described above.

The `check_contract` assigning contracts is how the recursive checking of object fields is done lazily, while still ensuring that a check is always done for anything typed, when this check becomes relevant. In other words, we are sure that if there is a type error, it will be caught when the erroneous field/value is used.

Consider the following example, a slight variation on the code example above:

```
1  class Point {
2          x:number;
3          y:number;
4  }
5
6  var p = {x:1, y:"ohno"};
7
8  function fun(q : Point) {
9      console.log("Hi");
10 }
11 fun(p);
12
13 console.log(p.y);
```

The latter part of this code gets compiled to the following JavaScript:

```
1 function fun(q : Point) {
2     contract_assign(q, Point);
3     console.log("Hi");
4 }
5 fun(p);
6
7 console.log(check_contract(p, y));
```

In this example, we see that the `p` variable gains the `Point` contract when `fun` is called with it. It is clear that `p` is not actually a point, since its `y` field is a string. However, the primitive contract is satisfied, in that `p` is an object and it is being checked with the `Point` type which is also an object. So, there is no type error at the function call, since the function body does not access the `y` field of `p`.

Then, when the print statement `console.log(p.y)` is called, this access to the `y` field results in the insertion of a `check_contract` call. This parses over the obligations present on `p` to see if there is one for the field `y`, since this is the field being accessed. Since `p` has a `Point` contract, it has the obligation that the `y` field must be a number. This obligation on the type of the field is applied as a contract on `y` through a call to `contract_assign`, and at this point the primitive type of the value in `y` is checked to see if it satisfies its contract. Here, the check finds that the value in `y` is a string, while the contract requires it to be a number. So, a runtime type error occurs at this point, attributed to the initial site where the contract was added. Note how although the field was causing a type error since the cast, it was only caught when the field was used, in the lazy checking style.

4.3 Runtime Semantics

As discussed, the typed language is type-reduced down to an untyped version of the language during compilation. This untyped language must also have a grammar, but the actual language rules are the same. The reason behind this is discussed in detail in Section 5.5.1, as it is closely related to the proof style required in Coq. But ultimately, the result is that the runtime language grammar is redundant and thus not included here, since all the expressions look the same as their untyped equivalents.

The only addition to the runtime language is the existence of an **error** value. Written as ϵ, the error value indicates that there has been an error during reduction – usually this is some sort of type error discovered during runtime type checking. The functionality of

error is discussed in detail below in terms of the reduction rules, which is where it becomes relevant.

There is also some language functionality which is only present at runtime – this forms the runtime grammar which is not part of the language, instead representing the runtime environment. This grammar is included below.

Signature	S	$::=$	$\widehat{I} \mid \widehat{C}$	*interface or class defn*
		$\mid$	S_1, S_2	*recursive case*
Local store	L	$::=$	$x \mapsto l$	*var position on heap*
		$\mid$	L_1, L_2	*recursive case*
Runtime heap	H	$::=$	$l \mapsto O$	*runtime object*
		$\mid$	$l \mapsto v$	*primitive value*
		$\mid$	H_1, H_2	*recursive case*
State	C	$::=$	$S; H; L$	

Primarily, we see the *state*, which contains the runtime heap, the local store, and the signature. Note that the signature is the same as that present in the typing environment, but it is included here in order to be able to access the contents of a class definition during runtime when a class constructor is called. In particular, calling a method on an object requires an access to the signature to retrieve the code for the method body.

The runtime heap is represented as a list of (location, heap contents) pairs, where the location is the position in the heap (i.e. the index), and the heap contents is either a runtime object or a primitive value. The local store is a list of the locations on the heap corresponding to the variables in the program. In other words, this is a list so the value of a variable can be retrieved. Both of these definitions are as normal for the heap and store.

In addition to the state, which is the runtime environment required for each reduction rule, the other terms seen only at runtime are the runtime objects themselves, and their component parts. The runtime objects are made up of their location on the heap, their list of methods and fields, and their list of contracts. The semantics is given below.

Runtime obj	O	$::=$	$\{l;\ \overline{\widehat{M}};\ \{\overline{f_e}, \overline{f_t}\};\ \overline{c}\}$

These are the heap contents which are not primitive values. The user cannot directly access a runtime object – they can interact with an object on the heap via the location,

but not remove it and store it in their own variable. As such, there is no expression which will contain an instance of a runtime object, since all references to objects are indirect.

Runtime objects are made up of various component parts. At first glance, they appear to be the same as the expression for object contents, but there are two key differences. The first difference is a bit subtle: object representations are defined as having method definitions and field definitions. However, in the runtime objects, there is a restriction that the expressions in the field definitions must be *values* (i.e. heap locations or primitive values), and not more complex expressions which can be reduced. This can be seen in the reduction rules of the object representation expressions; only once the field expressions have been reduced to values is the corresponding runtime object created and added to the heap.

The second difference is the introduction of the contracts. Each runtime object has a list of contracts; each time it is expected to have a particular type, the corresponding contract is added to its list. The contract itself is made up of a list of *obligations*.

Prim shape	p_s	::=	t_p	*primitive type*
		|	m	*method name*
		|	C	*class name*
		|		*struct shape*
Obligations	k	::=	p_s	*primitive obl*
		|	f_n, t	*field obl*
		|	m, t_m	*member fct obl*
		|	t	*fct arg obl*
		|	t	*fct return obl*

As we can see, the obligations essentially amount to sub-contracts on particular aspect of the object. There is the primitive obligation, which specifies the primitive shape of the value. In the case of a runtime object, this specifies that the object is actually an object, or a function (note that the primitive obligation for a primitive value is its primitive type). Then, in the case of an object, there is also an obligation for each of its member functions and fields. Functions also have obligations for their argument and return types, respectively.

When an object is designated with a particular type, the corresponding contract (i.e. list of obligations) is added to its contract list. This list of obligations is determined by calling the `toContract` method on the type, which converts a type into its corresponding contract. The behaviour of this function is described below; the actual implementation of the function in Coq is included in the attached Coq file, and is also described in Chapter 5.

The following describes the `toContract` function over each type.

- The `Any` type: returns the empty contract
- Primitive type: returns a primitive obligation equivalent to this type
- Struct type: returns a struct contract equivalent to the list of method contracts and field contracts for the component methods and fields
- Interface type: we can't have an interface object, so this is not a possibility for contracts
- Class type: this is the same as for the struct type case, with the added contract of the nominal type of the class as the primitive contract. The added complication in the class conversion to a contract is that the body of the class is not present in the constructor expression, just the name – so, to get the list of methods and fields in the class, the signature S must be accessed. In the Coq, there are various search functions implemented to find the contents of a class in the signature.

The key thing to note from this `toContract` function is that the evaluation is *lazy*. In this context, lazy contract evaluation means that the contracts are not recursively added through the structure of an object. Instead, only the primitive contract and the list of primitive obligations over the fields are added. Then, when a field or member function is accessed the appropriate contract is applied. In other words, the contracts are applied to the members when they become relevant. This is why errors in field assignment are only caught when the field is accessed, if it is not a shallow primitive error.

Now that we have outlined the component parts of the runtime grammar, we can proceed to discussing how a program is translated from the typed language to the untyped runtime language.

4.4 Type reductions

In most language formalizations, there is a *typing judgment* which, for each expression or statement in a language, shows that it can be obtained from the typing environment with a particular type.

In STS, we have a slightly different setup: since the distinction is made between the typed and untyped languages, the typing judgment serves more as a "translation" or

"compilation" from the user's typed language to the runtime untyped language (here, from STS to JavaScript), in addition to its role as a traditional typing judgment. Thus, rather than having the typing relation be from typed expression to typed expression, in our case it is from typed term to untyped term. This transformation where the typing relation changes languages results in some unexpected quirks in the Coq formalization – these are discussed in detail in Section 5.5.

The general formula for the STS typing relation is given by:

$$T \vdash k : t \hookrightarrow_k k'$$

This specifies that in the typing environment T, typed term k with type t is compiled to untyped term k' during type compilation. In STS there is a typing compilation for expressions, statements, field definitions, method definitions, and lists of expressions.

Most of the rules in the typing compilation are the same as in the original STS formalization. This is unsurprising, as the main change to the type system was in the addition of the object contracts, by way of the auxiliary functions described above. Thus, the main interesting typing rules are those which result in the addition of one of the auxiliaries to the term on type compilation. In the body of the thesis we go through these typing rules; the complete judgment is included for reference in Appendix A.

The first of these rules we will discuss is the most obvious: the typing compilation of a contract assignment expression.

$$\frac{T\ (e, t'') \hookrightarrow_e e' \qquad t'' <: t'}{T\ (\texttt{contract\_assign}(e, t), t') \hookrightarrow_e \texttt{contract\_assign}(e', t)}\ (\text{T\_ContAssign})$$

As expected, this rule states that given a contract assignment over a typed expression e, if e type compiles to untyped expression e', then the typed contract assignment over e type compiles to an untyped contract assignment over e'. One might be wondering how there is a typed expression for contract assignment, when contract assignment is a purely runtime aspect to the language. This is due to the required parallel between the typed and untyped languages, as discussed above.

Now we will discuss the rules where calls to `contract_assign` are inserted during type compilation.

$$\frac{(S, \Sigma, \Gamma)\ (e, t') \hookrightarrow_e e' \qquad \widehat{C} = getClass(S, C_n) \qquad F = getFieldDeclFromClass(\widehat{C}) \qquad F = \overline{f_e}, \overline{f_t} \qquad \overline{t} <: \overline{f_t}}{(S, \Sigma, \Gamma)\ (\texttt{new}\ C_n(\ \overline{e}),\ \mathrm{C}_n) \hookrightarrow_e \texttt{contract\_assign}(\ \texttt{new}\ C_n(\ \texttt{CA\_list}(\ \overline{e}, \overline{t})\ )\ )}\ (\text{T\_New})$$

The first instance where contracts are signalled to be added during type compilation is in the instantiation of a object of a specified class. There is a contract assignment on the return value of the constructor, to immediately add the nominal contract (i.e. the primitive contract stating that it is an object and specifying the name of its class) to the object. There is also a contract assignment on each of the arguments to the constructor, to their specified types (this is the `CA_list` function, `contract_assign` for a list of expressions and parallel list of types they should correspond to).

There is a `contract_assign` call added during type compilation of a method call.

$$\frac{\begin{array}{cc} T\ (e, t') \hookrightarrow_e e' & (T\ (\overline{e}, \overline{t'}) \hookrightarrow_{\overline{e}} \overline{e'} \\ \overline{t'} <: \overline{t} & (m_n) \in t' \end{array}}{T\ (e.m_n(\overline{e}, t) \hookrightarrow_e e'.m_n(\texttt{CA\_list}(\overline{e}, \overline{t}))} \qquad (\text{T\_SMCall})$$

Here we see a contract addition on each of the parameter values as they are passed in, corresponding to the respective types as given by the method definition retrieved from the type of e'. This is very similar to the above call to a class constructor.

The rest of the typing rules are almost the same as those given in the STS formalization. Recall that they are given in Appendix A, and they are also visible in their Coq representation in the proof code attached.

One might notice that no typing compilation rules highlighted above mention the `check_contract` auxiliary function. This is simply because no calls to `check_contract` are generated during type compilation. These are all generated during the runtime stage, given by the reduction rules and discussed in the next section.

4.4.1 Subtyping

The subtyping relation is the same as in the original STS formalization. We added some rules to the Coq formalization which were not present in the original paper. These were probably just left out since they were trivial. In particular, the transitivity subtyping rule was excluded in the paper formalization but required for the Coq. Recall that the transitivity subtyping rule states that given types A, B, C:

$$A <: B \ \wedge \ B <: C \implies A <: C$$

In terms of the formalization of STS specifically, the only noteworthy facts about the subtyping relation are that:

1. STS uses **width** subtyping

2. There is an interaction between structural types and the nominal classes in STS

Both of these have been discussed above in the description about STS and how it works.

4.5 Reduction Rules

While the typing compilation formalizes the translation of STS into JavaScript, the reduction rules formalizes the execution of the code: the reduction of every allowed term, and the eventual reduction of a whole program into a value (either an allowed value or ϵ error if there is a problem encountered during execution). The general setup for the reduction rules is the same as in general language formalizations, as they map untyped terms to untyped terms, under a particular runtime context.

This general form for reduction rules is given by

$$C/k \longrightarrow_k C'/k'$$

reducing untyped term k to untyped term k', with runtime states C to C'. Recall that the runtime state is composed of the heap, local store, and signature, as detailed above in the runtime-only grammar.

For STS, there are reduction rules over expressions, statements, and lists of expressions. Similarly to the typing relation, the vast majority of the reduction rules in HiggsCheck STS are the same as those presented in the original STS paper. The main differences are in the removal of rules containing the original auxiliary functions (`checkAndTag`, `shallowTag`, etc), and the addition of rules for the new auxiliary functions.

There are many more rules in our formalization than there are listed in the original STS. Most of the new ones, excluding the ones introduced to address the new auxiliary functions, are added either because they are trivial rules which were just not included in the original STS paper but were present implicitly in their formalization, or due to the introduction of ϵ error. Since we have error as a value, and incorrect terms stepping to error instead of getting "stuck" as is the standard approach, rules need to be introduced for all the possible terms which can step to error.

Error rules are all straight-forward: in general, we have a reduction where if some part of the term reduces to error then the whole term reduces to error. For example, consider

the reduction rule for the variable definition statement if the assigned expression reduces to error:

$$\frac{C/e \longrightarrow_s C/\epsilon}{C/\texttt{var}\ x : t := e \longrightarrow_s C/\epsilon} \quad (\text{S\_VarDef\_canRed\_ERR})$$

Here we see that if the expression e reduces to error, then the whole statement reduces to error. All of the error reduction rules are like this, where some component expression or statement reduces to error and causes the entire term to reduce to error.

Although this does inflate the number of reduction rules in the system, the resulting error rules are trivial. We will see in the next section how reasoning formally (particularly in regards to the Preservation proof) about these rules is simple as well.

Now we will discuss the reduction rules related to the new auxiliary functions. Consider first the reduction of `contract_assign` expressions. Recall that these take on the form $\texttt{contract\_assign}(e, \tau)$, where e is an expression and τ is a type.

The rule where the interior expression e can be reduced (i.e. is not yet a value) is trivial. In this case, the contract assignment wrapper remains unchanged, and the interior expression is reduced.

$$\frac{C/e_1 \longrightarrow_e C'/e_1'}{C/\texttt{contract\_assign}(e_1, \tau) \longrightarrow_e C'/\texttt{contract\_assign}(e_1', \tau)} \ (\text{E\_contAssign\_canRed})$$

Contract assignment over error is similarly trivial: assigning a contract to the error value just results in the error value.

$$\frac{}{C/\texttt{contract\_assign}(\epsilon, t) \longrightarrow_e C/\epsilon} \quad (\text{E\_contAssign\_ERR})$$

Now consider contract assignment over primitive values. In these cases, the contract assignment is equivalent to a simple type check to see if the primitive type of the value matches that the contract is trying to assign. No lists of contracts are stored on primitives, and the type check is trivial since the type information is carried along with the primitive values.

$$\frac{t = \mathit{primType}\ t_p}{C/\texttt{contract\_assign}(t_p\ p, t) \longrightarrow_e C/t_p\ p} \ (\text{E\_contAssign\_onPrim})$$

In the case where the contract assignment succeeds (i.e. the types match), the expression reduces to just the primitive value.

$$\frac{t \neq \mathit{primType}\ t_p}{C/\texttt{contract\_assign}(t_p\ p,\ t) \longrightarrow_e C/\epsilon}\ (\text{E\_CONTASSIGN\_ONPRIM\_ERR})$$

Then, in the case where the types do not match, the expression reduces to error. This mismatch could be the contract assignment supplying the incorrect primitive type (i.e. `string` where `bool` was expected, etc), or it could be in supplying a non-primitive type, in which case it immediately fails.

Now consider the final case, of contract assignment over locations in the heap. There are 3 potential cases when assigning over a location. First, there is the possibility that the location being referred to does not actually correspond to something on the heap.

$$\frac{\mathit{findHeapContsAtLoc}(H,\ l_1) = \texttt{nulObj}}{(S; H; L)/\texttt{contract\_assign}(l_1,\ t) \longrightarrow_e (S; H; L)/\epsilon}\ (\text{E\_CA\_ONLOC\_BADLOC})$$

Here the expression is reduced to error, since a location which does not exist in the heap is being accessed.

If the location is there, then it can map to either a primitive value or an object on the heap. In the primitive case, no change will be made to the heap, since primitive values do not store their contracts. So, this can be reduced to just a regular contract assignment over a primitive value, essentially dereferencing the location and returning the corresponding value in the heap.

$$\frac{\mathit{findHeapContsAtLoc}(H,\ l_1) = t_p\ p}{C/\texttt{contract\_assign}(l_1,\ t) \longrightarrow_e C/\texttt{contract\_assign}(t_p\ p,\ t)}\ (\text{E\_CONTASSIGN\_ONLOC\_PRIM})$$

If the contract assignment is over a location which points to a runtime object, then the contract must be added to the object. This is a two-step process: first, we check to ensure that the primitive obligation is met. This amounts to checking that the type being assigned via contract is actually an object type. If this is not the case, then this expression reduces to error, as there cannot be a primitive shape obligation on an object (i.e. an object can never typecheck as a `string`, for example).

$$\frac{findHeapContsAtLoc(H,\, l_1) = \{l;\ \widehat{M};\ \widehat{F};\ \overline{c}\} \qquad isObjType(t) = \texttt{false}}{C/\texttt{contract\_assign}(l_1,\, t) \longrightarrow_e C/\epsilon}\ (\text{E\_CA\_onLoc\_obj\_badType})$$

If the type being assigned is actually an object type, then the contract corresponding to this type must be assigned to the object. This is done via the lazy addition style discussed above.

$$\frac{findHeapContsAtLoc(H,\, l_1) = \{l;\ \widehat{M};\ \widehat{F};\ \overline{c}\} \qquad isObjType(t) \qquad H' = updateObjWithContract(H,\, l_1,\, t)}{(S; H; L)/\texttt{contract\_assign}(l_1,\, t) \longrightarrow_e (S; H'; L)/l_1}\ (\text{E\_CA\_onLoc\_runObj})$$

The specific implementation details of *updateObjWithContract* are not given here, but this just adds the contract lazily. The high-level version of this function behaviour is that it takes the type t specified and converts it to its corresponding contract through the use of the `toContract` function described earlier; then, the object in the heap H at location l_1 is accessed, the contract computed is added to its contract list, and the heap is updated so l_1 contains the new object (i.e. the same object, but with the extra contract). For the full implementation details, consult the attached Coq code.

Now that we have discussed the `contract_assign` auxiliary function, we can delve into the reduction rules for `check_contract`. The style of these is very similar to the reduction rules for `contract_assign`. Recall that this auxiliary function checks to see if a member accessed (i.e. a field or member function) is actually present on the object being accessed on, and, if so, applies the contract as specified in the contract list on the object. Recall also that these calls take on the form $\texttt{contract\_assign}(e, s)$, where e is an expression and s is either a method name or a field name.

The rule where the interior expression e can be reduced (i.e. is not yet a value) is trivial, and almost identical to that for `contract_assign`. Again, the contract check wrapper remains unchanged, only the interior expression is reduced.

$$\frac{C/e_1 \longrightarrow_e C'/e_1' \qquad e_1' \neq \epsilon}{C/\texttt{check\_contract}(\texttt{e}_1,\, \texttt{s}) \longrightarrow_e C'/\texttt{check\_contract}(e_1',\, s)}\ (\text{E\_checkCont\_canRed})$$

Checking a contract over an expression which reduces to error is similarly trivial: checking over an error just results in the error value.

$$\frac{C/e_1 \longrightarrow_e C'/e_1' \qquad e_1' = \epsilon}{C/\texttt{check\_contract}(\texttt{e}_1,\, \texttt{s}) \longrightarrow_e C'/\epsilon}\ (\text{E\_checkCont\_canRed\_ERR})$$

Now consider checking contracts over non-error values. Contract check is only done over objects, since it relates exclusively to member accesses; therefore, calling `check_contract` over any value not a runtime object should result in an error.

Since the only way to access a runtime object is by way of a heap access, contract checking over any value not a heap location should result in an error.

$$\frac{value(e_v) \qquad \nexists l \mid e_v = l}{C/\texttt{check\_contract(e}_\texttt{v}\texttt{, s)} \longrightarrow_e C/\epsilon} \text{(E\_checkCont\_notLoc)}$$

However, just being a heap location does not mean that the value referenced is necessarily a runtime object, since there are also primitive values on the heap.

$$\frac{value(e_v) \qquad \exists l \mid e_v = l \qquad findHeapContsAtLoc(H, l) = t_p\, p}{\{S; H; L\}/\texttt{check\_contract(e}_\texttt{v}\texttt{, s)} \longrightarrow_e \{S; H; L\}/\epsilon} \text{(E\_checkCont\_loc\_prim)}$$

In the case where the heap location points to a primitive value, this also results in an error when calling `contract_check`.

The last error case is when the member being accessed is not present on the object being referenced. An example case for this would be attempting to access the `z` field of a 2D point which only has `x` and `y` fields.

$$\frac{\begin{array}{c} value(e_v) \qquad \exists l \mid e_v = l \qquad findHeapContsAtLoc(H, l) = obj \\ \texttt{None} = getMemberFromRunObj(obj, s) \end{array}}{\{S; H; L\}/\texttt{check\_contract(e}_\texttt{v}\texttt{, s)} \longrightarrow_e \{S; H; L\}/\epsilon} \text{(E\_checkCont\_loc\_noMember)}$$

If `None` is returned from *getMemberFromRunObj*, this means that there is just no member of that name present on the object, and thus this access should also result in an error. Note that in the Coq formalization we make a distinction between this rule for accessing methods or accessing fields, although the process is exactly the same – we have chosen to write the simpler version here since it is shorter.

The interesting cases for `check_contract` are when there are no errors, and the field or method accessed is actually present on the object being referenced.

$$\frac{value(e_v) \qquad \exists\, l \mid e_v = l \qquad findHeapContsAtLoc(H,\, l) = obj \qquad e = getMemberFromRunObj(obj,\, s) \qquad \bar{t} = getTypesFromRunObj(obj,\, s)}{\{S; H; L\}/\texttt{check\_contract}(\texttt{e}_\texttt{v},\, \texttt{s}) \longrightarrow_e \{S; H; L\}/applyListOfContsFromTypes(S, \bar{t}, e)}$$
(E_CHECKCONT_OBJ_MEMBER)

This rule specifies the behaviour if a member access on an object actually succeeds: the list of member obligations (returned as a list of types the member should correspond to) is retrieved from the object. Then, this list of types is applied as a sequence of contracts to the expression returned from the field access. In practice, this is a chain of nested `contract_assign` function applications.

There is a difference in contracts between methods and fields; however, the method call-specific contract assignments are all added during the type compilation, and therefore at the reduction rule point they are already part of the code and do not need to be added again. Therefore, checking the type itself is sufficient to add here.

The complete set of reduction rules (including those which are identical to those in the original STS) are included for reference in Appendix B.

Chapter 5

Proofs in Coq

In this chapter we will discuss the formalization and mechanization of the semantics and proofs for the HiggsCheck STS type system. The semantics were formalized by hand, but then mechanized so as to be used in the relevant proofs. All mechanization was done using the Coq proof assistant.

The goal of this chapter is to go over the most important details in terms of the mechanization process, and detail the outlines of the proofs. The proofs discussed here are Progress and Preservation; these are described in detail later in the chapter, but the point of proving these is that together they prove type soundness of the system. These are the classic proofs to show when proving a type system.

Also note that the proof explanations detailed here are an overview: the entire proofs, in all their gory details, are included in the attached Coq file, hosted at https://bitbucket.org/emarteca/higgschecksts/src/master/.

5.1 Intro to Coq

Coq is a proof assistant. This language is commonly used for the formalization and mechanized proving of language properties. Many other languages have propreties proven in Coq; in this thesis, we use some of the techniques presented in the Coq formalization of Featherweight Java, given by [9]. The relevant details of this formalization are discussed below.

Coq is a functional language implemented using OCaml. The syntax and language constructs are quite different than what someone with an imperative background would

expect. One major difficulty when using Coq to prove theorems about a programming language is determining a good representation of this language in order to make the proof structure as simple as possible.

The grammar of HiggsCheck STS is represented in Coq as a sequence of *mutual inductives*. In Coq, this means the language is being represented in terms of the abstract syntax tree (AST) of a HiggsCheck STS program, with tree nodes of different types; these nodes are mutually recursive since any node may have children of other node types. In terms of the grammar, this means the terms are defined mutually recursively. As an example, consider the following subset of the HiggsCheck STS grammar written out in Coq:

```
1  Inductive fieldDecl : Type :=
2    | fDecl : list (fieldName * ty) -> fieldDecl
3
4  with methDecl : Type :=
5    | mDecl : list (methName * methType) -> methDecl
6
7  with methType : Type :=
8    | methType_ty : list ty -> ty -> methType
9
10 with ty : Type :=
11   | prim_dTyp       : primType -> ty
12   | mf_dTyp         : methDecl -> fieldDecl -> ty
13   | Any             : ty
14   | inter_dTyp      : interName -> ty
15   | class_dTyp      : className -> ty.
```

This corresponds to the grammar for the field declarations, method declarations, method type, and type. The `with` notation specifies that the next statement will be part of the mutually recursive grammar. The names after all the `Inductive` or `with` specify the non-terminals which make up the grammar. The vertical bars separate alternative constructors for a node type, with the constructor name before the colon and the types of arguments and result after it. Although the format might not be what we first think of when we think of a language grammar, it is easy to translate to the more traditional form when writing grammars by hand.

Functions in Coq are given as either `Definition` (non-recursive function), or `Fixpoint` (recursive function). As one example of a function, consider the Coq version of the `toContract` explained in the last chapter. Recall that this function converts a type to its corresponding contract. This works by pattern matching over the type passed in, and determining the contract depending on the type of type (i.e. primitive, struct, etc).

```
1  Definition toContract (S: signature) (t: ty): beta_contracts :=
2    match t with
3    | Any => empty_cont
4    | prim_dTyp p => lobls_cont [prim_obl (primitive p)]
5    | mf_dTyp mD fD => lobls_cont ( [(prim_obl structS)] ++
6      ((getBOblsMethDecl mD) ++ (getBOblsFieldDecl fD)))
7    | inter_dTyp iN => empty_cont (* can't have an interface object *)
8    | class_dTyp cN => lobls_cont ( [(prim_obl (class cN))] ++
9      ((getBOblsMethDecl (getMethDeclFromDefn
10                        (getMethods (getClass S cN)))) ++
11     (getBOblsFieldDecl (getFieldDeclFromOptionDecl
12                        (getFields (getClass S cN))))))
13   end.
```

The function is comparing the input parameter `t`, the type, with the various potential forms a type can take. Then, depending on what form the type has taken, the action taken by `toContract` differs.

This function contains many calls to external functions, such as `getBOblsFieldDecl`. Their implementations are not included here, but what they do is basically implied from the name (where `BObls` means obligations). `getBOblsFieldDecl` computes the field obligations from a field declaration `fD` passed in, and similarly for `getBOblsMethDecl`. `((getBOblsMethDecl (getMethDeclFromDefn (getMethods (getClass S cN))))` gets the method obligations from the methods in a nominal class named `cN`. The bloat here comes from the need to access the signature `S` to retrieve the class definition, so as to be able to extract the methods and fields.

In terms of actually proving things in Coq, this is generally done by deconstructing terms in some way. In this thesis, most of the proofs are done by induction over terms in the proof statement. We do not go into detail about how to prove things in Coq here, as this is a very complex topic – the proof strategies relevant to this thesis are described when the proofs are described below.

5.2 Advantages of Mechanization

There are many advantages to a mechanization of a language formalization and the associated type system proofs. First and foremost, a mechanization verifies that the proof is indeed correct (for the grammar and semantics specified; correctness for the language

assumes that this language representation in Coq is correct). When formalizing the semantics of a large language (and this applies to most languages which are used in practice and are not just toy languages which exist purely for demonstrative purposes), there are so many features and quirks to the grammar that it is very easy to miss something when doing the proofs, or even in writing out the semantics. Of particular relevance to this work is the paper formalization of the original implementation of STS.

Their semantics alone take up five pages as an appendix to the STS technical report; this is excluding any of the proofs, and counting only the grammar, operational semantics, and typing judgment. To say this is dense is an understatement. As such, reading and understanding the entire formalization is quite a time commitment, and this leads to less rigorous checking by reviewers.

This is very clear in the STS case, as there were enough typos in the grammar and semantics to cause much head-scratching on our end and wasted time in reasoning out the true intentions. All of these typos were found after weeks of working closely with the grammar; it was not the fault of the reviewers or the authors for not noticing these small errors, however presenting the formalisms only on paper makes typographical errors almost inevitable.

With a mechanization, the semantics presented must be guaranteed to satisfy certain properties, and thus the issue of missing reduction rules or typos in the grammar is greatly reduced.

An additional advantage of mechanization is the reusability aspect. Consider in particular the slight modification or feature addition to an existing language, which is a very common practice in PL. Given a mechanized proof for the original language, the modifications could be added to the grammar and then the proof compiled to quickly see "what breaks". This usefulness is twofold: for one, this is a fast and efficient way to see what the effect of the new changes are on the formal aspects of the soundness of the language. In addition, it quickly isolates the exact changes which need to be made to the reduction rules and typing judgment following the changes to the grammar.

5.3 Pre-existing Mechanized Proofs

Starting a mechanized formalization from scratch is a daunting task. To inspire us in terms of technique, we looked at one previously existing formalization: the mechanized proof of Featherweight Java presented in [9], where they also provided their entire Coq setup for this proof.

5.3.1 Key Differences between TypeScript and Featherweight Java

Although the Featherweight Java (FWJ) proof was a good inspiration as to how setup the language formalization, and provided ideas of how to set up some proofs, there were also some major differences between our language and theirs. Most importantly, Featherweight Java uses a purely nominal type system. As such, all type checking can be done quickly and easily; subtype checks amount to just string comparisons. As discussed in Section 2.4.1 structural type checking is a much more difficult process. This applies not only to the implementation, but also to the formalization: the conditions for a type check to be satisfied in a nominal type system are much simpler to verify.

For instance, for a nominal type to be a subtype of another, it must be explicitly declared as such when the type is created. The following code sample shows an example of the declaration of a nominal subtype in Java-like pseudocode.

```
1 class  A {
2     int x;
3 }
4
5 class B extends A {
6     String y;
7 }
```

In this example class `B` is a subtype of `A`. The entire inheritance hierarchy is explicitly declared, and any object of type `B` has both a string field `y` and an integer field `x`. Therefore, runtime subtype checks are simple since they just check the existing inheritance hierarchy which was computed statically. There is no need to descend into the structure of the object to check for any fields being present: they are guaranteed to be present due to the type of the object.

However, STS (as with all typed JavaScript derivative languages) is structurally typed. This means that the type of an object depends entirely on its structure, rather than the other way around as in nominal typing. This also means that the subtyping relation between any two types is dependent on the structure of these types. The subtyping in STS is **width subtyping**. This was presented in Section 2.4.1, but as a quick reminder, with width subtyping, record `A` is a subtype of record `B` if `A` has all the same fields as `B`, with the same types, and potentially more fields. In other words, the set of *(fieldName, type)* pairs in `B` is a subset of that of `A`.

The result is that the subtyping in STS is fundamentally different and much more complex than the subtyping in FWJ.

Another key difference is due to the language paradigms. FWJ is a statically typed language: all type checking is done at compile time. As such, there is a more traditional formal setup with the typing judgment stating the type for any expression in the language, and the reduction rules specifying reduction from one expression to the next. In contrast, STS is a gradually typed language, with multiple phases of "compilation". There is the typing reduction stage, where any statically typed code is type checked, and runtime type checks are inserted at appropriate locations. This type reduction is equivalent to the typing judgment. However in general the typing judgment goes from an (expression, type) pair to an expression (as such, typing the expression). In STS, there are two phases of the language, the typed language and the runtime language; so, the typing judgment (in this case called the typing reduction) goes from (typed expression, type) pair to an untyped expression. Then, the STS reduction rules go from untyped expression to untyped expression. This is explained in the introduction to Chapter 4. The different stages of the grammar, and the parallels which must be semantically drawn between them, is very different from the setup in FWJ.

However, there are also many similarities as the general soundness proof formula is the same for every case. The key similarities are discussed in the next section, in addition to the other details about the formalization.

5.4 HiggsCheck STS formalization

The general process of formalizing the semantics of a language (specifically here, this version of STS) can be detailed as follows.

1. Formalizing the grammar, to include the new features
2. Figuring out the new reduction rules and typing judgment
3. Writing all of this out in Coq
4. Proving soundness of the system

Each of these steps is explained in more detail in the section below.

5.4.1 Mechanizing the Semantics

First, the grammar and semantics must be formalized. In this case this step was fairly straightforward, as we were applying a modification to the existing STS language, which was available to us. So the first step was figuring out what changes would have to be made to the original STS grammar. This mainly involved the addition of the expressions for the primary auxiliary functions, `check_contract` and `contract_assign`, as explained in Chapter 4.

Once the grammar has been formalized, the typing judgment and reduction rules must also be written out. The typing judgment is how the type of an expression in the language is specified. In STS, rather than having a typing judgment just for expressions, we have a typing judgment for expressions, statements, field definitions, method definitions, and lists of expressions. This is due entirely to the complicated nature of the language, meaning that the typing of one of these productions is not possible without the typing of these others. The typing judgment is defined mutually inductively, and was given and described in more detail in Chapter 4.

Ultimately, when a program runs, the end result will either be a value or an error in execution. The rules by which expressions are reduced into values are the reduction rules. Similarly to the typing judgment, in this system we have a mutually inductive set of reduction rules, where untyped expressions, statements, and lists of expressions are reduced.

The proof for type soundness is made up of two major subproofs: progress and preservation. The process to prove these is usually fairly standardized, following the conventions presented in [13], given a particular language formalization. Each of these, and the approach used to prove them, is described below.

5.4.2 Progress

The general statement of progress is as follows, as presented by [13], for the simply typed lambda calculus (STLC):

Theorem 2 : Progress (STLC)*: closed, well-typed terms are not stuck: either a well-typed term is a value, or it can be reduced.*

This can be written formally as: $\forall e, T : \quad \vdash e \in T \implies value(e) \vee \exists\, e', e \to e'$

As suggested by the name, STLC is a simplified generalization of most programming languages which are used in practice. It is a great tool for instructional purposes, and many of the approaches can be applied to more complex language formalizations. However, the added language features in many formalizations of actual languages add complexity to the style of these proofs.

In the STLC, reduction rules are from term to term, as the grammar is uncomplicated enough to not require mutual recursion of nonterminals. In STS, as shown in Section 4.2, the grammar has various mutually recursive nonterminals for which reduction rules need to be defined, *expressions*, *statements*, and *lists of expressions*. Therefore, the statement of progress is no longer as simple as "well-typed terms are either a value, or can be reduced" – we need a progress statement which includes this condition for all nonterminals for which reduction is defined.

Another difference between the STLC and STS is in the definition of a term being "stuck". In the STLC, a term being stuck refers to there not being a reduction rule corresponding to it. In other words, a term being stuck means that it is not a value, but there is no way for it to be reduced given the semantics of the language – this corresponds to an error. In the grammar presented here, rather than representing a stuck term by just not including a reduction rule for the cases where the result is an invalid term in the language, we have these cases step to error, written as ϵ. This is explained in detail in Section 2.5.

The result is that there are many more reduction rules than if we had just not included those where the result would step to error. However, we decided that it would be more explicitly clear if the reduction rules were all included, and stepping to error is a transparent way of showing that there was a problem with code being run while still ensuring that every program returns a value. It also has the added benefit of greatly simplifying the proof for preservation, which will be discussed in the next section.

The statement for progress in STS can be written formally as follows:

Theorem 3 : Progress*:*

$$(\ \forall T, s, s' :\ T \vdash s \hookrightarrow_s s' \implies \forall C :\ (value(s') \vee (s' = \texttt{skip})) \vee (\exists\, C', s'' :\ C/s' \longrightarrow_s C'/s'') \quad \wedge$$

$$(\ \forall T, e, \tau, e' :\ T \vdash e : \tau \hookrightarrow_e e' \implies \forall C :\ value(e') \vee (\exists\, C', e'' :\ C/e' \longrightarrow_e C'/e'') \quad \wedge$$

$$\big(\ \forall T, \overline{e}, \overline{\tau}, \overline{e'} :\ T \vdash \overline{e} : \overline{\tau} \hookrightarrow_{\overline{e}} \overline{e'} \implies \forall C :\ expListAllValues(\overline{e'}) \vee \big(\exists\, C', \overline{e''} :\ C/\overline{e'} \longrightarrow_{\overline{e}} C'/\overline{e''}\big) \quad \wedge$$

$$\big(\ \forall T, \overline{f_e}, \overline{f'_e}, \overline{f_{nt}} :\ T \vdash \big(\overline{f_e}, \overline{f_{nt}}\big) \hookrightarrow_{\overline{f_d}} \big(\overline{f'_e}, \overline{f_{nt}}\big) \implies \forall C :\ fieldIsVal(\big(\overline{f'_e}, \overline{f_{nt}}\big)) \vee \big(\exists\, C', \overline{f''_e} :\ C/\overline{f'_e} \longrightarrow_{\overline{e}} C'/\overline{f''_e}\big) \quad \wedge$$

$$\big(\ \forall T, \overline{m_d}, \overline{m'_d} :\ T \vdash \overline{m_d} \hookrightarrow_{\overline{m_d}} \overline{m'_d} \implies methDefnIsValue(\overline{m'_d})\big)$$

This theorem statement is very unwieldy, but it is actually fairly understandable once it is broken down into its component parts, each of which are very similar to the initial statement for the STLC. The first condition is progress over statements. If statement s type-reduces to untyped statement s', then either s' is a value or `skip`, or there exists another untyped statement s'' such that s' steps to s''.

The second condition is progress over expressions. If expression e with type τ type-reduces to untyped expression e', then either e' is a value or there exists another untyped expression e'' such that e' steps to e''. Notice how `skip` is not an option for e' here, since `skip` is a statement and not an expression.

The third condition is progress over lists of expressions. If a list of expressions $\overline{e}$ with types $\overline{\tau}$ type-reduces to a list of untyped expressions $\overline{e'}$, then either every $e'_i \in \overline{e'}$ is a value, or at least one e'_i can be reduced (i.e. there exists a new list of expressions $\overline{e''}$ such that $\overline{e'}$ steps to $\overline{e''}$).

The fourth statement is progress over field definitions. The notation here is a bit different, but the idea is the same. If a field definition $\big(\overline{f_e}, \overline{f_{nt}}\big)$ type-reduces to an untyped field definition $\big(\overline{f'_e}, \overline{f_{nt}}\big)$, then either all the field expressions $\overline{f'_e}$ are values, or this list of

expressions can be reduced. Note that when the field definition is type-reduced, the list $\overline{f_{nt}}$ remains unchanged. This is the expected behaviour – it would be very strange if the names or declared types of object fields changed as the field expressions were reduced. Although the types of the expressions may become more refined as the fields are reduced, this does not affect the declared types, which are those included in the `(fieldName, type)` list $\overline{f_{nt}}$.

The last statement is progress over method definitions. This case is different than the rest, as there is no reduction implication. In JavaScript (and its derivatives), functions are objects – as such, considering them as irreducible values makes sense. Once a method is called, then the expression making up the method body is reduced with the relevant parameter values substituted in, however it would not make sense to reduce this beforehand. The *methDefnIsValue* always returns `True`, as methods are always considered values – this may seem redundant, but is required in order to be able to use the combined scheme. The function name was included in the proof to make it more symmetrical with the other cases, and to make it more readable.

Together these five clauses make up progress, defining it over every term which has a corresponding type reduction scheme. Notice that there is no mention of a term being stuck, as the concept of stuck is not applicable in this grammar. There is no mention of error in the progress proof, as $value(\epsilon) = \texttt{true}$, and so if something steps to ϵ it still satisfies the statement of progress.

The Proof

The recipe for proving progress is also detailed in [13] for the STLC. The general idea is to induct over the typing judgment, and then prove progress for each subcase, i.e. each rule in the typing judgment. We adapted the proof for STLC, and encountered some complications, described below.

The first complication arose in performing the induction over the typing judgment. Since the typing judgment is defined over five terms, all of which are mutually recursive, in order to properly induct we need to perform a mutual induction. We drew inspiration from the approach used in proving progress for FWJ, as they also had a mutually recursive typing judgment.

In Coq, induction over a mutually recursive statement can be done via a combined scheme. This just specifies which inductives should be included, and then applying the scheme inducts over all its components; this will produce an inductive hypothesis which includes all of those inductives in the scheme. This description is not very clear without

an example; so, we have included the combined scheme for the typing judgment below for clarity.

```
Scheme typing_s_ind := Minimality for type_reduction Sort Prop
with typing_e_ind := Minimality for type_reduction_exps Sort Prop
with typing_es_ind := Minimality for type_reduction_exps_list
                        Sort Prop
with typing_f_ind := Minimality for type_reduction_fieldDefns
                        Sort Prop
with typing_m_ind := Minimality for type_reduction_methDefns
                        Sort Prop.

Combined Scheme typing_mutind from typing_s_ind, typing_e_ind,
                typing_es_ind, typing_f_ind, typing_m_ind.
```

This code shows the creation of the combined scheme for the typing judgment. It is clear that all of the components of this scheme correspond to what the typing judgment is defined over, i.e. statements, expressions, lists of expressions, field definitions, and method definitions. This is the same approach that the FWJ Coq proof took, since they also had the mutually recursive typing judgment inductives.

In terms of the actual Progress proof itself, the approach was also quite similar that taken when proving FWJ. The theorem statement in Coq is as shown above in the written Theorem 3, although arguably a bit more readable than the math version.

```
Theorem progress :
      ( forall TypCont s s',
        type_reduction TypCont s s' ->
        forall C, ((value s') \/ (s' = beta_s_skip)) \/
            (exists C' s'',
                reduction (C, s') (C', s'')))
  /\  ( forall TypCont et e',
        type_reduction_exps TypCont et e' ->
        (value (beta_s_exp e')) \/
            forall C, (exists C' e'',
                reduction_exps (C, e') (C', e'')))
  /\  ( forall TypCont lett le',
        type_reduction_exps_list TypCont lett le' ->
        (expListAllValues le') \/
            forall C, (exists C' le'',
                ListReduction (C, le') (C', le'')))
  /\  ( forall TypCont lfds lfds',
```

```
18          type_reduction_fieldDefns TypCont lfds lfds' ->
19          (forall lfnts lfes',
20           (beta_fDefn lfnts lfes') = lfds' ->
21           ((beta_field_is_value lfds') \/
22              forall C, (exists C' lfes'',
23                  ListReduction (C, lfes') (C', lfes'')))))
24    /\ ( forall TypCont lmds lmds',
25          type_reduction_methDefns TypCont lmds lmds' ->
26          (methDefnIsValue lmds')).
```

This proof statement is deceptively long – in fact, most of the space is taken up by the wordiness of all the variable and function names as compared to the compact math notation. In reality, it is the same five statements: progress over each term the typing judgment is defined over.

Then, to do the proof, we start by applying the combined scheme `typing_mutind`. At this point we have one case for each of the typing rules. At a high level, the approach to solving each of the cases is largely the same: first we introduce all the hypotheses, and inversion over the hypothesis which states that the term in question is either a value or can be reduced (in most of the proofs the autogenerated name is `H0`). This inversion results in two subcases: one where the term is a value, and one where it is reducible. In general, the value case is the harder to solve, since we can't just use the inductive hypothesis, but the formula is generally the same for all cases, in destructing some sub-category of value relevant to the rule and doing an extended case analysis (for example, destructing whether a type is a primitive type or not in the case for typing primitive values).

As a demonstrative example of the proof style for the subcases, consider the proof of the subcase for the `T_VDef` typing rule, corresponding to variable definition. The typing rule is written out below:

$$\frac{T\ (e, t') \hookrightarrow_e e' \qquad e' \neq \epsilon \qquad t' <: t}{T\ (x : t := e) \hookrightarrow_s x : t := e'} \tag{T\_VDef}$$

Then, the corresponding progress proof for this typing rule is as follows:

```
1  - intros. right. inversion H0.
2    + destruct C. inversion H3.
3      * (* is primval *)
4        destruct t'.
5        destruct (prim_ty_eq_dec pt p0); destruct p; destruct pt;
6        try (var_econ 2; eapply beta_E_VarDef_prim_badType;
```

```
7                                                    eauto; fail);
8       try (var_econ 2; eapply beta_E_VarDef_prim_ok;
9                                              eauto; fail).
10
11      var_econ 2. eapply beta_E_VarDef_prim_objType.
12      eapply ifObjSubtypeObjType with (oT:=(mf_dTyp m f))
13            (S:=S); eauto.
14
15      var_econ 2. eapply beta_E_VarDef_prim_objType.
16      eapply ifObjSubtypeObjType with (oT:=Any) (S:=S); eauto.
17
18      var_econ 2. eapply beta_E_VarDef_prim_objType.
19      eapply ifObjSubtypeObjType with (oT:=(inter_dTyp i))
20            (S:=S); eauto.
21
22      var_econ 2. eapply beta_E_VarDef_prim_objType.
23      eapply ifObjSubtypeObjType with (oT:=(class_dTyp c))
24            (S:=S); eauto.
25    * (* is loc *)
26      remember (findHeapContsAtLoc b l) as tg.
27      inversion Heqtg.
28      destruct l. var_econ 2. eapply beta_E_VarDef_loc;
29        eauto. eauto.
30  + edestruct (e_C_redErr C e').
31    * destruct e0. exists x0. var_econ 1.
32      eapply beta_E_VarDef_canRed_ERR.
33      eapply beta_E_e_steps. eauto.
34    * edestruct H3. edestruct H4. var_econ 2.
35      eapply beta_E_VarDef_canRed. eauto.
36      unfold not in n. unfold not. intros. subst.
37      apply n. exists x0. eauto.
```

The gist of this is that we have are considering the various cases for the expression e. If e is a value, then we divide it into the cases of either being a primitive value, which is the first * case, or a location, which is the second * case. Then, in each of these cases, we break down some subproperties until we can apply some appropriate reduction rule. For example, in the primitive value * case, we destruct the type, as there are different reduction rules for variable definition depending on what type is used.

The rest of the subcases are quite similar in approach. This entire proof can be examined in detail in the attached Coq file.

5.4.3 Preservation

Preservation is the other main property which must be proven which, together with progress, proves type soundness of the system in question. It is the companion property to progress, with a complementary statement and proof style. Intuitively, preservation means that the reduction relation preserves types.

The general statement of preservation is as follows, as presented by [13] , for the STLC:

Theorem 4 : Preservation*: if a closed term e has type T and takes a step to some other term e′, then e′ is also a closed term with type T.*

This can be written formally as: $\forall e, e', T : \vdash e \in T \implies e \to e' \implies \vdash e' \in T$

The STS statement of preservation, similar to that of progress, is a bit more complicated, but essentially the same in setup. The first main difference is that rather than having just one reduction relation from term to term, STS has reductions over statements and lists of expressions too. Therefore, the theorem statement has to have a statement for each of these cases.

Preservation for HiggsCheck STS is expressed as follows in Coq:

```
1  Theorem preservation :
2      (forall p p', reduction p p' ->
3      (forall S Sigma ss s s' Sig H X L Sig' H' X' L',
4      ((bstate_state Sig H X L), s) = p ->
5      ((bstate_state Sig' H' X' L'), s') = p' ->
6      s' <> (beta_s_exp (beta_e_value beta_v_ERROR)) ->
7      type_reduction (typConx S Sigma []) ss s ->
8       heap_store_well_typed Sigma H ->
9        ( exists ss' S' Sigma',
10             ST_Extends Sigma' Sigma /\
11             Sig_Extends S' S /\
12             type_reduction (typConx S' Sigma' []) ss' s'))))    /\
13
14     (forall p p', ListReduction p p' ->
15     (forall S Sigma es es' Sig H X L Sig' H' X' L' ts ess,
16      ((bstate_state Sig H X L), es) = p ->
17      ((bstate_state Sig' H' X' L'), es') = p' ->
18      ListReduction ((bstate_state Sig H X L), es)
19                    ((bstate_state Sig' H' X' L'), es') ->
```

```
20      ~ (errorInListOfExps es') ->
21      type_reduction_exps_list (typConx S Sigma []) (ess, ts) es ->
22       heap_store_well_typed Sigma H ->
23        (exists ess' S' Sigma' ,
24           ST_Extends Sigma' Sigma /\
25           Sig_Extends S' S /\
26           type_reduction_exps_list (typConx S' Sigma' [])
27                                    (ess', ts) es'))) /\
28
29
30      (forall p p', reduction_exps p p' ->
31      (forall S Sigma t ee e e' Sig H X L Sig' H' X' L',
32      ((bstate_state Sig H X L), e) = p ->
33      ((bstate_state Sig' H' X' L'), e') = p' ->
34      e' <> (beta_e_value beta_v_ERROR) ->
35      type_reduction_exps (typConx S Sigma []) (ee, t) e ->
36      heap_store_well_typed Sigma H ->
37        ( exists ee' t' S' Sigma',
38            ST_Extends Sigma' Sigma /\
39            Sig_Extends S' S /\
40            subtyping S' t' t /\
41            type_reduction_exps (typConx S' Sigma' []) (ee', t') e'))).
```

Similar to the Coq statement for Progress, this proof statement is long and hard to read, but also in some ways easier to read than the equivalent math statement given above, as the names are all longer and more explicit. There are some quirks to the statement as compared with those in the math version.

In particular, the first 4 lines of each condition are not present in the math versions. While it would make sense to just collapse

```
1 (forall p p', reduction p p' ->
2 (forall S Sigma ss s s' Sig H X L Sig' H' X' L',
3 ((bstate_state Sig H X L), s) = p ->
4 ((bstate_state Sig' H' X' L'), s') = p' ->
```

into one line stating reduction from `p` to `p'` with the states explicitly written out in their components (S, Σ, Γ), this is not possible due to the finicky nature of Coq. While using the combined scheme over reduction, the style of the statement must match exactly; and, since reduction rules are defined over states and not all the components of the state, so, too, must the statement be defined in terms of the states.

The proof itself follows a similar approach to that taken by FWJ. [13] presents a recipe for proving preservation in the general case, which involves proving a particular sequence of lemmas. Sadly, this recipe was not applicable to this project, due to the fact that the method body is stored in an external store, and is not available in the expression of the method call. In the formalization of HiggsCheck STS, the same as in original STS, the method body is available in the signature (S in the runtime state) and can be looked up via the method name.

The FWJ formalization has a similar setup to ours in terms of the method body storage, and so we instead followed their recipe for the preservation proof. This style was essentially to create a combined scheme for reduction (almost identical to the combined scheme for the typing relation), apply that, and then solve a case for each of the reduction rules. In general the cases here were much more straight-forward than those in the proof for progress. Although there were many more cases (90+ reduction rules means 90+ cases), the proof for preservation was easier since each case was both very similar and very short.

Many of the cases in preservation were trivial: since in the proof statement there is a clause that the term is not an error, every reduction rule which is defined over error (i.e. where a component of a statement or expression is an error and so the entire expression/statement reduces to error) is solved by a simple contradiction, as in these cases there exist hypotheses both that the term is an error and is not an error.

In the cases where there was no trivial contradiction with error, the general recipe was also simple and basically the same over every case. First we inversion over the hypotheses resulting from the long form state equalities (`(( bstate_state Sig H X L), s) = p`, etc). In the cases where we have an error contradiction, this is where it appears. In the non-error cases, at this point we can inversion over the type reduction hypothesis. Then, the rest of the proof is just stating which term can satisfy the type reduction; this is usually very clear, we just need to find the corresponding typed term to the untyped term on the right hand side. The only complication is in dealing with the signature and store typing being extended. We have companion lemmas for these, to show that the type relation is preserved over extended parts of the typing context, which are trivially applied in the preservation proof. These are discussed in the next section.

Consider the proof for the subcase of preservation for the first reduction rule, `S_Val`. The reduction rule is as follows:

$$\frac{}{C/v \longrightarrow_s C/\texttt{skip}} \quad \text{(S\_Val)}$$

Then, the Coq proof for this subcase is as follows.

```
- inversion H2. inversion H3. subst. clear H2 H3.
  inversion H5; subst;
  econstructor; exists S, Sigma; eauto using Sig_Extends_refl,
                                                ST_Extends_refl.
```

This just follows the recipe described above: we can prove the subcase trivially using the weakening lemmas.

5.4.4 Companion Lemmas

In order to prove the main results of Progress and Preservation, many intermediate results were required. These can essentially be divided into two categories: those which were required as an actual part of a proof (useful lemmas), and those which were required for some trivial result for which Coq required a formal proof.

Useful Lemmas

Most of the lemmas proving subresults are relevant to the Preservation proof. In particular, we have the *weakening* results, relevant to the extension of the signature and store typing. The statements for weakening are as follows:

Lemma 1 : Weakening*:*

$$\forall\, S, \Sigma, \Gamma, S', \Sigma', k_t : \\ S' \,\texttt{extends}\, S \implies \Sigma' \,\texttt{extends}\, \Sigma \implies (S, \Sigma, \Gamma)\, k_t \hookrightarrow_k k \implies \\ (S', \Sigma', \Gamma)\, k_t \hookrightarrow_k k$$

The actual statement is very intuitive: if new classes or interfaces are added to the signature, and/or new (heap location, type) pairs are added to the store typing, then this has no effect on what was previously type-reducible. In other words, adding new types does not break the typing of anything which was already typed.

This lemma is stated in its most general form, with a type reduction from typed k_t to k; in STS, weakening is stated over all nonterminals for which a typing reduction is defined (i.e. statements, expressions, lists of expressions, field definitions, and method definitions).

Proving the weakening lemmas in Coq is done by induction over the typing judgment. Again, like in Progress, since the typing relation is mutually recursive this requires a mutual

inductive scheme to reflect this. We can reuse the same one as for proving Progress. Then, each case analysis amounts to a trivial application of the relevant typing rule.

Once the weakening lemmas have been proven, we still need to prove some results relevant to signature and store typing extension. This amounts to showing that other properties (not just the typing relation) are preserved over extension.

With regards to the signature, this list includes class name lookup, interface name lookup, and all other functions which involve reading from the signature and are included in the typing relation. A complete list is available in the Coq code attached. All the proofs follow the exact same format, and can also be followed along in the Coq provided. Consider the following proof as an example.

```
1  Lemma sig_extends_lookup_cn : forall cN S S' cD,
2    (Some cD) = (getClass S cN) ->
3    Sig_Extends S' S ->
4    (Some cD) = (getClass S' cN).
5  Proof.
6    intros. induction H0.
7    - subst. inversion H.
8    - simpl in *. eauto.
9    - destruct cd.
10     + simpl in H. eapply IHSig_Extends in H. simpl. eauto.
11     + simpl in *. destruct (beq_cName cN c); eauto.
12 Qed.
```

The above code snippet is the Coq proof for class definition lookup in the signature being preserved over signature extension. The basic recipe for the proof is inducting over the generated hypothesis that `Sig_Extends S' S`, i.e. S' `extends` S, which in this proof is named `H0`. Then, the subcases generated are for the 3 constructors for the signature inductive: the nil signature (i.e. the empty signature), and the constructors adding a class name and an interface name, and dealing with these is fairly trivial, as shown in the proof above.

The proof style for store typing extension itself is slightly different than for signature extension, since the store typing is a list rather than an inductive. The induction is done over Σ itself, rather than over a generated hypothesis like we had for the signature extension proofs.

Consider the following proof as an example:

```
1 Lemma ST_extends_app : forall Sigma lt,
2   ST_Extends (Sigma ++ [lt]) Sigma.
3 Proof.
4   intros. induction Sigma; intros; eauto.
5   simpl. eauto.
6 Qed.
```

This is a proof that adding a new element (i.e. (heap location, type) pair) to the store typing results in an extension of the original store typing. In other words, adding to the list which is the store typing does not change or break anything which was previously in the list.

These lemmas are part of the recipe for showing weakening and propagating its results throughout the relevant sections of the Preservation proof. Similar lemmas are the norm in type system soundness proofs. The style used here (in particular, the mutual induction scheme needed for the mutually recursive reduction rules and typing judgment) is inspired from the proof techniques used for these lemmas in the FWJ proof [9].

Coq-placating Lemmas

While there are many lemmas described above which show pertinent results for the various proofs, there are also some lemmas which described more trivial results that Coq required convincing for. These lemmas are labelled as such in the attached Coq file. Had the proofs been done on paper, these lemmas would not have been required, since the results are obvious.

We will not discuss all of these lemmas in detail, as the results are not very important for the work at hand. However, it is worth noting that the vast majority of these involve proving some trivial result over lists; Coq appears to have a hard time automatically proving things when lists are present. Consider the following lemma as a demonstrative example.

```
1 Lemma noErrInListNoErr : forall e es,
2   ~ (errorInListOfExps (e :: es)) ->
3   e <> (beta_e_value beta_v_ERROR).
4 Proof.
5   unfold not. intros. subst.
6   unfold errorInListOfExps in H. contradiction.
7 Qed.
```

The above lemma is proving the result that, if there is a list of expressions with a non-zero length (i.e. there is a first element), and this list contains no errors, then the first element in the list is not an error. There are many such lemmas proving results about lists of expressions.

The one "Coq-placating" lemma which is not related to lists is a proof about object subtyping. We show this lemma below.

```
1  Lemma ifObjSubtypeObjType :
2    forall S oT t,
3      (isObjType oT) /\ (subtyping S oT t) ->
4      (isObjType t).
5  Proof.
6    intros. inversion H.
7    induction H1.
8    - apply IHsubtyping2; eauto.
9    - destruct tau; eauto.
10   - inversion H0.
11   - unfold isObjType. trivial.
12   - destruct t2; try (inversion H2; fail). eauto.
13   - destruct t2; try (inversion H2; fail). eauto.
14   - destruct t2; try (inversion H2; fail). eauto.
15   - destruct t2; try (inversion H2; fail). eauto.
16 Qed.
```

This lemma proves the result that if type oT is an object type, and $oT <: t$, then t is also an object type. This result is also trivial: of course, an object type cannot be a subtype of a non-object type. One important thing to note about this proof is that the dynamic `Any` type returns `True` from the `isObjType` check; it essentially takes the role of Top. While it could be pointed out that a term with `Any` type is not necessarily an object, in this case it makes sense to return `True` when checking if something with the dynamic type is an object, because otherwise this lemma would be false and `Any` could no longer be considered Top.

5.5 Quirks about this Mechanization

When adapting the original STS semantics to properly model the our new version, the main consideration was how to properly integrate the contracts into the pre-existing rules, and how to formally express the new auxiliary functions. Since the semantics were first

developed on paper, at first these were the only considerations required. However, when mechanizing them, some complications arose which had to be resolved by changes to the grammar. These are detailed below.

5.5.1 One-to-one Correspondence between Typed and Untyped

Running an STS program involves a two-step process: first, the STS code is compiled into JavaScript, and then the JavaScript is run. The JavaScript produced is untyped, and contains any auxiliary functions (i.e. checking or adding contracts) which were generated during the compilation, and which the user has no access to. On paper, this draws a very clear divide between the user language available to the programmer, and the intermediate language generated by the compiler.

However, the line between typed and untyped languages becomes blurred when formalizing it in Coq. The issue arises when proving Preservation: this proof requires that for each untyped expression involved in a reduction rule, we can also get it out of the typing reduction. However, there are some untyped expressions which have no typed equivalent, namely the auxiliary functions. While there are reduction rules for `contract_assign` and `check_contract`, there are no typed expressions which produce these directly.

This would probably be possible to justify verbally were the proof done on paper, but with Coq this needs to be justified formally. The solution to this issue is to introduce rules to the typed grammar which type-reduce to the expressions which previously only made an appearance in the untyped grammar. The result is that there is essentially a one-to-one correspondence required between the untyped and typed grammar. Then, as specified in Chapter 4, the typed language is actually a superset of the language made available to the user.

5.5.2 Avoiding Infinite Contract Addition

There are various typing reductions which result in the addition of a contract auxiliary function. For example, when there is an explicit cast, a contract assignment is added during type compilation (assigning the type specified in the cast as a contract to the expression in the cast). The typing rule for casts is as follows:

$$\frac{T\ (e, t) \hookrightarrow_e e' \qquad t <: t'}{T\ (<t'>e,\ t) \hookrightarrow_e <t'>\ \texttt{contract\_assign}(e', t)} \qquad \text{(T\_Cast)}$$

However, if this was the only typing rule for cast, then it would be impossible to get a cast expression without a contract assignment out of the typing relation. This causes a problem in preservation, where any cast expression is considered to be gotten out of the typing relation with a contract added, thus resulting in essentially infinite contract addition around the object of the cast.

The solution is to create a typing rule for "not the first time this cast is encountered", i.e. a typing rule for a cast where we know the contract has already been added. To do this, we have included a two-step expression to cast, the `e_cast` and `e_FTcast` expressions, where the `FT` indicates "first time". This `e_FTcast` expression is identical in composition to the `e_cast` rule, in that it is composed of a type `t` and an expression `e`, in the form `<t> e`. When the `e_FTcast` expression is found from the typing relation, this is where we add the contract, given by the typing rule above, `T_Cast`. Then, once it is not the first time the cast is encountered, we use a typing rule without adding a contract:

$$\frac{T\ (e, t) \hookrightarrow_e e' \qquad t <: t'}{T\ (< t' > e,\ t) \hookrightarrow_e < t' > e'} \qquad \text{(T\_Cast\_notFirst)}$$

Because of the required one-to-one correspondence between the typed and untyped languages, we also need an untyped expression corresponding to `e_FTcast`. Then, in the reduction rules, the untyped `e_FTcast` is just reduced directly to the corresponding `e_cast`, in the following rule:

$$\frac{}{C/\texttt{e\_FTcast} < t > e \longrightarrow_e C/\texttt{e\_cast} < t > e} \qquad \text{(E\_FTCast\_DH)}$$

This reduction is not conditional; its only purpose is to provide a means for the typing relation to add a contract to `e_FTcast` expressions. This rule acts essentially as a switch, to label a cast as having already been "seen" once by the typing relation, so the contract will not be added again.

Then, there are the regular reduction rules for the cast expressions, which involves actually reducing the expression within the cast until it becomes either ϵ or a value. These reduction rules are shown all included in Appendix B.

One thing to note, of some interest: The first time the cast expression passes through one of the rules after having gotten through `e_FTcast`, the expression within the cast will be wrapped in a `contract_assign` auxiliary function call. Therefore, the first actual reduction on a cast will always be a reduction of the `contract_assign` call. However, there is no need to take any of this into account in the reduction rules, since the general

reduction rule with a reducible inner expression covers this case in addition to the more general case.

Now we have formally discussed the mechanization of the HiggsCheck STS in Coq. This concludes the work presented in this thesis: the entire Coq formalization is available in the attached Coq file for further examination.

At this point we will conclude with a recap of the work presented, and some insight into potential future work on this project.

Chapter 6

Conclusions and Future Work

In summary, this thesis presents the formal semantics and a mechanized proof of soundness for the implementation of STS presented in [15]. This required many modifications to the grammar and semantics presented for the original STS presented by [14]. All of the modifications to the language and new semantics required are described in detail through the thesis, in addition to the Coq work required to prove progress and preservation of the system.

In terms of applicability, this is the first mechanization of a derivative of JavaScript, and, as such, could serve as a base for future mechanizations of other derivative languages.

There are many potential applications to the object contract system developed and presented in this work on the implementation side as well.

A fast implementation of a typesound TypeScript is inherently useful, as it is a direct improvement on the existing optionally typed TypeScript. The implementation of gradual typing and a functioning blame tracking system would greatly improve error tracking and debugging in TypeScript. As explained, the original implementation of STS was not used because of its inefficiency – the increase in execution time made it unusable. However, our implementation is much faster and as such taking a small performance hit as a tradeoff for more information and better debugging is a more plausible choice.

6.1 Current Related Work

Since the publication of [15], there have already been some papers which reference its implementation work done. Chung et al. [5], focus on multiple potential implementation

strategies for gradual typing. They implement gradual typing over a toy system with a variety of dynamic semantics, including guarded (in their words *behavioural*), transient, and optional (as in optional typing). They also explore *concrete* dynamic semantics, which are those used by Muehlboeck and Tate [11]. Their objective was to compare the different dynamic semantics in the same framework, so as to have a common ground to compare them with, in particular with regards to the various typing guarantees. Chung et al. also did similar work with monotonic semantics, in their paper [6], presented at a workshop cohosted with the same conference as they presented [5].

There continues to be work in making gradual typing more efficient: another related paper is that presented by Kuhlenschmidt et al. [8], introducing an ahead-of-time compiler called Grift. Like our work, this attempt at speeding up gradual typing also uses monotonic semantics.

The recent work most related to [15] is [17]. This paper presents another approach to eliding redundant runtime checks in a language running on a virtual machine, by adding shallow structural type checks to the Truffle interpreter for Grace, in yet another step towards faster sound gradual typing. This paper also uses some of our benchmarks to test their system.

6.2 Future Work

There are also many other potential applications of this work. In light of the results by [11] that show that gradual typing is much faster over a nominal type system (as expected, due to the simplicity of nominal type checks), there is work on creating a nominal type system to overlay over the existing HiggsCheck STS, to take advantage of this for further speedups. There is also the beginnings of work on a project which would implement HiggsCheck STS with semantics more forgiving than the monotonic semantics, based around leveraging the garbage collector.

In terms of working off of the implementation side of this project, the object contract semantics using information in the virtual machine could be implemented over any gradually typed language running on a VM. Since it is fast in TypeScript, the same strategy should apply to other such languages. Similarly, porting this implementation to VMs other than the Higgs research VM would also be useful avenues for further work. In fact, we are currently looking at porting this to the V8 JavaScript VM, which is very widely used.

The theory presented in this paper has less practical applications to actual programmers, but, as the first mechanized proof of a JavaScript derivative language, this proof can

serve as the baseline for future work in similar avenues. For example, proving soundness over any language derivative of HiggsCheck STS will be much easier now, as it would amount to just a modification of the existing mechanized proof. Even another JavaScript derivative language could still benefit from this proof as inspiration for how to prove soundness over their system, although more modifications would be required. Changing the operational semantics or the grammar of the language and then observing how this breaks the proof in Coq would give good insight as to how any changes would propagate through the functionality of the language, and how a particular modification would affect soundness.

Appendices

Appendix A

Complete Typing Judgment

This appendix is the complete typing judgment for this system. Appendix B details the reduction rules. The list of typing rules here includes those highlighted in the main text.

Recall that there is a typing judgment for statements, expressions, lists of expressions, field declarations, and method declarations.

First we consider the typing rules for expressions.

$$\frac{t = \Sigma[l]}{(S, \Sigma, \Gamma)(l, t) \hookrightarrow_e l} \quad \text{(T\_Loc)}$$

$$\frac{t = \Gamma[x]}{(S, \Sigma, \Gamma)(x, t) \hookrightarrow_e x} \quad \text{(T\_Env)}$$

$$\frac{}{(S, \Sigma, \Gamma)((t_p p), t_p) \hookrightarrow_e (t_p, p)} \quad \text{(T\_Const)}$$

$$\frac{T\ \widehat{F} \hookrightarrow_f \widehat{F}' \qquad T\ \widehat{M} \hookrightarrow_m \widehat{M}'}{T\ (\{\widehat{M}, \widehat{F}\}, getStructTypeFromDefns(\ \{\widehat{M}, \widehat{F}\})\ ) \hookrightarrow_e \{\widehat{M}', \widehat{F}'\}} \quad \text{(T\_Rec)}$$

$$\frac{(S, \Sigma, \Gamma)\ (e, t') \hookrightarrow_e e' \qquad \begin{array}{c} \widehat{C} = getClass(S, C_n) \qquad F = getFieldDeclFromClass(\widehat{C}) \\ F = \overline{f_e}, \overline{f_t} \qquad \overline{t} <: \overline{f_t} \end{array}}{(S, \Sigma, \Gamma)\ (\texttt{new}\ C_n(\ \overline{e}),\ \mathrm{C}_n) \hookrightarrow_e \texttt{contract\_assign}(\ \texttt{new}\ C_n(\ \texttt{CA\_list}(\ \overline{e}, \overline{t})\ )\ )} \quad \text{(T\_New)}$$

$$\frac{T\ (e, t') \hookrightarrow_e e' \qquad (T\ (\overline{e}, \overline{t'}) \hookrightarrow_{\overline{e}} \overline{e'} \\ \overline{t'} <: \overline{t} \qquad (m_n) \in t'}{T\ (e.m_n(\overline{e}, t) \hookrightarrow_e e'.m_n(\texttt{CA\_list}(\overline{e}, \overline{t}))} \quad \text{(T\_SMCall)}$$

$$\frac{T\ (e, t'') \hookrightarrow_e e' \qquad t'' <: t'}{T\ (\texttt{contract\_assign}(e, t), t') \hookrightarrow_e \texttt{contract\_assign}(e', t)} \quad \text{(T\_ContAssign)}$$

$$\frac{T\ (\overline{f_e}, \overline{f_t}) \hookrightarrow_{\overline{e}} \overline{f'_e}}{T\ (\ l; \overline{\widehat{M}}; \overline{f_e}, \overline{f_t}cs, getStructTypeFromDefns(\ \overline{\widehat{M}}, \overline{f_e}, \overline{f_t})\ )\ \hookrightarrow_e l; \overline{\widehat{M}}; \overline{\widehat{M}}} \quad \text{(T\_ObjNotation)}$$

$$\frac{T\ (e, t) \hookrightarrow_e e' \qquad t <: t'}{T\ (< t' > e,\ t) \hookrightarrow_e < t' > \texttt{contract\_assign}(e', t)} \quad \text{(T\_Cast)}$$

$$\frac{T\ (e, t') \hookrightarrow_e e' \qquad T\ (e.s,\ t) \hookrightarrow_e e'.s}{T\ (\texttt{check\_contract}(e,\ s), t) \hookrightarrow_e \texttt{check\_contract}(e',\ s)} \quad \text{(T\_CheckCont)}$$

$$\frac{T\ (e, t') \hookrightarrow_e e'}{t\ (e.f, t_f) \hookrightarrow_e e'.f} \quad \text{(T\_FldRd)}$$

Next, we consider the typing judgment for lists of expressions. Here we have two rules, one for the empty list (base case), and one for a list with a non-zero number of elements (recursive case).

$$\frac{}{T\ ([\,], [\,]) \hookrightarrow_{\overline{e}} [\,]} \quad \text{(T\_ListExp\_Empty)}$$

$$\frac{T\ (e, t) \hookrightarrow_e e' \qquad T\ (\overline{e}, \overline{t}) \hookrightarrow_{\overline{e}} \overline{e'}}{T\ (e :: \overline{e}, t :: \overline{t}) \hookrightarrow_{\overline{e}} e' :: \overline{e'}} \quad \text{(T\_ListExp\_Cons)}$$

The typing judgment for methods and fields are quite similar, and similar in setup to the lists of expressions, in that we consider the base case and recursive case. First, consider the typing judgment for fields.

$$\frac{}{T\ ([\,], [\,]) \hookrightarrow_{fd} ([\,], [\,])} \quad \text{(T\_FDs\_Empty)}$$

$$\frac{T\ (e_1, t_1') \hookrightarrow_e e_1' \qquad t_1' <: t_1 \qquad T\ (\overline{f_{nt}}, \overline{f_e}) \hookrightarrow_{fd} (\overline{f_{nt}}, \overline{f_e'})}{T\ ((f_n, t_1) :: \overline{f_{nt}}, e_1 :: \overline{f_e}) \hookrightarrow_{fd} ((f_n, t_1) :: \overline{f_{nt}}, (\texttt{contract\_assign}(e_1', t_1)) :: \overline{f_e'})} \quad \text{(T\_FDs\_Cons)}$$

And now, for methods:

$$\frac{}{T\ ([\,], [\,]) \hookrightarrow_{md} ([\,], [\,])} \quad \text{(T\_MDs\_Empty)}$$

$$\frac{T'\ (e_1, t_1') \hookrightarrow_e e_1' \qquad T\ s_1 \hookrightarrow_s s_1' \qquad T\ (\overline{m_{nt}}, \overline{m_c}) \hookrightarrow_{md} (\overline{m_{nt}}, \overline{m_c'}) \qquad T' = T, \Gamma \text{ updated with } \overline{v}}{T\ ((m_n, t_1) :: \overline{m_{nt}}, \{\overline{v}, t_1, s_1, e_1\} :: \overline{m_c}) \hookrightarrow_{md} ((m_n, t_1) :: \overline{m_{nt}}, \{\overline{v}, t_1, s_1', e_1'\} :: \overline{f_e'})} \quad \text{(T\_MDs\_Cons)}$$

Finally, we have the type reductions for statements. These are very similar in format to the type reductions for expressions.

$$\frac{T\ (e, t) \hookrightarrow_e e'}{T\ e \hookrightarrow_s e'} \quad \text{(T\_Exp)}$$

$$\frac{}{T\ \texttt{skip} \hookrightarrow_s \texttt{skip}} \quad \text{(T\_Skip)}$$

$$\frac{T\ (e,\ t') \hookrightarrow_e e' \qquad e' \neq \epsilon \qquad t' <: t}{T\ (x : t := e) \hookrightarrow_s x : t := e'} \quad \text{(T\_VDef)}$$

$$\frac{T\ (x, t) \hookrightarrow_e x \qquad T\ (e, t') \hookrightarrow_e e' \qquad t' <: t}{T\ x = e \hookrightarrow_s x = e'} \quad \text{(T\_VAssign)}$$

$$\frac{T\ (e_1, t_1) \hookrightarrow_e e_1' \qquad T\ (e_2, t_2) \hookrightarrow_e e_2' \qquad t_1 <: t_2}{T\ e_1.f = e_2 \hookrightarrow_s e_1'.f = e_2'} \quad \text{(T\_SFAssign)}$$

$$\frac{T\ (e, t) \hookrightarrow_e e' \qquad T\ s_1 \hookrightarrow_s s_1' \qquad T\ s_2 \hookrightarrow_s s_2'}{T\ \texttt{if}(e)\ s_1\ \texttt{else}\ s_2 \hookrightarrow_s \texttt{if}(e')\ s_1'\ \texttt{else}\ s_2'} \quad \text{(T\_If)}$$

$$\frac{T\ s_1 \hookrightarrow_s s_1' \qquad T\ s_2 \hookrightarrow_s s_2'}{T\ s_1;\ s_2 \hookrightarrow_s s_1';\ s_2'} \quad \text{(T\_Seq)}$$

Appendix B

Complete Reduction Rules

This appendix is a complete list of all the reduction rules in this system. Appendix A details the typing judgment. The list includes those highlighted in the main text.

Recall that there are reduction rules for statements, expressions, and lists of expressions. Recall also that the reduction rules are only for the subset of the language which exists at runtime, i.e. the untyped language.

First, the reduction rules from statement to statement. These reduction rules take the form:

$$\frac{conditions}{C/s \longrightarrow_s C'/s'}$$

Here, recall that s and s' are untyped statements, and C and C' are runtime states. The state is an amalgamation of the components of the runtime environment. Specifically,

$$C ::= S; H; L$$

Where S is the signature (list of existing nominal types and their definitions), H is the runtime heap, and L is the variable store.

So now we will detail the reduction rules for statements.

Value:

$$\frac{v \neq \epsilon \qquad value(v)}{C/v \longrightarrow_s C/\texttt{skip}} \qquad \text{(S\_Val)}$$

Sequences:

$$\frac{}{C/\texttt{skip}; s \longrightarrow_s C/s} \quad \text{(S\_Seq\_Skip)}$$

$$\frac{C/s_1 \longrightarrow_s C'/s_1' \qquad s_1' \neq \epsilon}{C/s_1; s_2 \longrightarrow_s C'/s_1'; s_2} \quad \text{(S\_Seq\_canRed)}$$

$$\frac{C/s_1 \longrightarrow_e C'/\epsilon}{C/s_1; s_2 \longrightarrow_s C/\epsilon} \quad \text{(S\_Seq\_canRed\_ERR)}$$

$$\frac{}{C/\epsilon; s \longrightarrow_s C/\epsilon} \quad \text{(S\_Seq\_ERR)}$$

Variable definition:

$$\frac{C/e \longrightarrow_e C'/e' \qquad e' \neq \epsilon}{C/\texttt{var}\ x : t := e \longrightarrow_s C'/\texttt{var}\ x : t := e'} \quad \text{(S\_VarDef\_canRed)}$$

$$\frac{C/e \longrightarrow_s C/\epsilon}{C/\texttt{var}\ x : t := e \longrightarrow_s C/\epsilon} \quad \text{(S\_VarDef\_canRed\_ERR)}$$

$$\frac{primTypeMatches(p,\ t) \qquad H' := C.H[C.L(x) \mapsto (p, t)] \qquad C' := C.S; H'; C.L}{C/\texttt{var}\ x : t := t\ p \longrightarrow_s C'/\texttt{skip}} \quad \text{(S\_VarDef\_Prim\_OK)}$$

$$\frac{!\,primTypeMatches(p,\ t)}{C/\texttt{var}\ x : t := t\ p \longrightarrow_s C/\epsilon} \quad \text{(S\_VarDef\_Prim\_badType)}$$

$$\frac{isObjType(t)}{C/\texttt{var}\ x : t := t\ p \longrightarrow_s C/\epsilon} \quad \text{(S\_VarDef\_Prim\_objType)}$$

$$\frac{}{C/\texttt{var}\ x : t := \epsilon \longrightarrow_s C/\epsilon} \quad \text{(S\_VarDef\_ERR)}$$

Variable update:

$$\frac{C/e \longrightarrow_e C'/e' \qquad e' \neq \epsilon}{C/x ::= e \longrightarrow_s C'/x := e'} \quad \text{(S\_VarUpd\_canRed)}$$

$$\frac{C/e \longrightarrow_s C/\epsilon}{C/x := e \longrightarrow_s C/\epsilon} \quad \text{(S\_VarUpd\_canRed\_ERR)}$$

$$\frac{C.L(x) \neq \texttt{nil} \qquad H' := C.H[C.L(x) \mapsto (p,t)] \qquad C' := C.S; H'; C.L}{C/\texttt{var}\ x : t := t\ p \longrightarrow_s C'/t\ p} \quad \text{(S\_VarUpd\_Prim)}$$

$$\frac{}{C/x := \epsilon \longrightarrow_s C/\epsilon} \quad \text{(S\_VarUpd\_ERR)}$$

$$\frac{value(v) \qquad C.L(x) = \texttt{nil}}{C/x : t := v \longrightarrow_s C/\epsilon} \quad \text{(S\_VarUpd\_BadLoc)}$$

Static field update:

$$\frac{C/e_1 \longrightarrow_e C'/e_1' \qquad e_1' \neq \epsilon}{C/e_1.f := e_2 \longrightarrow_s C'/e_1'.f := e_2} \quad \text{(S\_SFldUpd\_firstRed)}$$

$$\frac{value(v) \qquad \nexists\, l \in C.H \mid v = l}{C/e_1.f := e_2 \longrightarrow_s C/\epsilon} \quad \text{(S\_SFldUpd\_firstBadVal)}$$

$$\frac{C/e \longrightarrow_e C'/e' \qquad e' \neq \epsilon}{C/l.f := e \longrightarrow_s C'/l.f := e'} \quad \text{(S\_SFldUpd\_canRed)}$$

$$\frac{updateObjFieldWithVal(C.H,\ f,\ l,\ (t\ p)) \qquad C' := C.S; H'; C.L}{C/l.f := t\ p \longrightarrow_s C'/t\ p} \quad \text{(S\_SFldUpd\_prim)}$$

$$\frac{updateObjFieldWithVal(C.H,\ f,\ l_1,\ l_2) \qquad C' := C.S; H'; C.L}{C/l_1.f := l_2 \longrightarrow_s C'/l_2} \quad \text{(S\_SFldUpd\_loc)}$$

Dynamic field update:

$$\frac{C/e \longrightarrow_e C'/e'}{C/l[v] := e \longrightarrow_s C'/l[v] := e'} \quad \text{(S\_DFldUpdLit\_canRed)}$$

$$\frac{f := toString(v) \qquad updateObjFieldWithVal(C.H,\ f,\ l_1,\ l_2) \qquad C' := C.S; H'; C.L}{C/l_1[v] := l_2 \longrightarrow_s C'/l_2} \quad \text{(S\_DFldUpdLit\_loc)}$$

$$\frac{f := toString(v) \qquad updateObjFieldWithVal(C.H,\ f,\ l,\ (t\ p)) \qquad C' := C.S; H'; C.L}{C/l_1[v] := t\ p \longrightarrow_s C'/t\ p} \text{(S\_DFLDUPDLIT\_PRIM)}$$

$$\frac{}{C/l_1[v] := \epsilon \longrightarrow_s C/\epsilon} \text{(S\_DFLDUPDLIT\_ERR)}$$

If statements:

$$\frac{v = \textbf{bool}\ \texttt{true}}{C/\texttt{if}(v)\{s_1\}\ \texttt{else}\{s_2\} \longrightarrow_s C/s_1} \text{(S\_IF\_BOOLVAL\_TRUE)}$$

$$\frac{v = \textbf{bool}\ \texttt{false}}{C/\texttt{if}(v)\{s_1\}\ \texttt{else}\{s_2\} \longrightarrow_s C/s_2} \text{(S\_IF\_BOOLVAL\_FALSE)}$$

$$\frac{value(v) \qquad \nexists\ b \mid v = \textbf{bool}\ b}{C/\texttt{if}(v)\{s_1\}\ \texttt{else}\{s_2\} \longrightarrow_s C/\epsilon} \text{(S\_IF\_BADVAL)}$$

$$\frac{C/e \longrightarrow_e C'/e' \qquad e' \neq \epsilon}{C/\texttt{if}(e)\{s_1\}\ \texttt{else}\{s_2\} \longrightarrow_s C'/\texttt{if}(e')\{s_1\}\ \texttt{else}\{s_2\}} \text{(S\_IF\_CANRED)}$$

$$\frac{C/e \longrightarrow_e C/\epsilon}{C/\texttt{if}(e)\{s_1\}\ \texttt{else}\{s_2\} \longrightarrow_s C/\epsilon} \text{(S\_IF\_CANRED\_ERR)}$$

Expression:

$$\frac{C/e \longrightarrow_e C'/e'}{C/e \longrightarrow_s C'/e} \text{(S\_E\_STEPS)}$$

Method call:

$$\frac{\begin{array}{ccc} value(e) & \exists\ l \in C.L \mid e = l & !\ errorInListOfExps(\overline{v}) \\ O := C.H[l] & \overline{vt}, t, s, e_1 := getMethDefnFromRunObj(O, m) & \end{array}}{C/e.m(\overline{v}) \longrightarrow_s C/subst\_stat((\overline{vt}, \overline{v}), s)} \text{(S\_SMCALL\_M\_S)}$$

Now, the reduction rules from expression to expression. These reduction rules take the form:

$$\frac{conditions}{C/e \longrightarrow_e C'/e'}$$

Here, recall that e and e' are untyped expressions, and C and C' are runtime states (same as for the statement reduction rules above).

Object Literals:

$$\frac{C/\overline{f_e} \longrightarrow_{\bar{e}} C'/\overline{f'_e} \qquad !\, errorInListOfExps(\overline{f'_e})}{C/\{\widehat{M}; (\overline{f_{nt}}, \overline{f_e})\} \longrightarrow_e C'/\{\widehat{M}; (\overline{f_{nt}}, \overline{f'_e})\}} \text{ (E\_objLit\_canRed)}$$

$$\frac{C/\overline{f_e} \longrightarrow_{\bar{e}} C'/\overline{f'_e} \qquad errorInListOfExps(\overline{f'_e})}{C/\{\widehat{M}; (\overline{f_{nt}}, \overline{f_e})\} \longrightarrow_e C'/\epsilon} \text{ (E\_objLit\_canRed\_ERR)}$$

$$\frac{fieldIsVal(\widehat{F})}{C/\{\widehat{M}; \widehat{F}\} \longrightarrow_e C'/\{length(C.H);\ [\widehat{M}];\ \widehat{F};\ [\,]\}} \text{ (E\_objLit\_isVal)}$$

$$\frac{objIsVal(\{l;\ \widehat{M};\ \})}{C/\{\widehat{M}; (\overline{f_{nt}}, \overline{f_e})\} \longrightarrow_e C'/\epsilon} \text{ (E\_objLit\_redToRunObj)}$$

Contract Assign:

$$\frac{C/e_1 \longrightarrow_e C'/e'_1}{C/\texttt{contract\_assign}(e_1,\, \tau) \longrightarrow_e C'/\texttt{contract\_assign}(e'_1,\, \tau)} \text{ (E\_contAssign\_canRed)}$$

$$\frac{t = primType\ t_p}{C/\texttt{contract\_assign}(t_p\ p,\, t) \longrightarrow_e C/t_p\ p} \text{ (E\_contAssign\_onPrim)}$$

$$\frac{t \neq primType\ t_p}{C/\texttt{contract\_assign}(t_p\ p,\, t) \longrightarrow_e C/\epsilon} \text{ (E\_contAssign\_onPrim\_ERR)}$$

$$\frac{}{C/\texttt{contract\_assign}(\epsilon,\, t) \longrightarrow_e C/\epsilon} \text{ (E\_contAssign\_ERR)}$$

$$\frac{findHeapContsAtLoc(H,\, l_1) = t_p\ p}{C/\texttt{contract\_assign}(l_1,\, t) \longrightarrow_e C/\texttt{contract\_assign}(t_p\ p,\, t)}\ (\text{E\_contAssign\_onLoc\_prim})$$

$$\frac{\begin{array}{c} findHeapContsAtLoc(H,\, l_1) = \{l;\ \widehat{M};\ \widehat{F};\ \overline{c}\} \\ !\ isObjType(t) \end{array}}{C/\texttt{contract\_assign}(l_1,\, t) \longrightarrow_e C/\epsilon}\ (\text{E\_CA\_onLoc\_obj\_badType})$$

$$\frac{\begin{array}{c} findHeapContsAtLoc(H,\, l_1) = \{l;\ \widehat{M};\ \widehat{F};\ \overline{c}\} \\ isObjType(t) \qquad H' = updateObjWithContract(H,\, l_1,\, t) \end{array}}{(S; H; L)/\texttt{contract\_assign}(l_1,\, t) \longrightarrow_e (S; H'; L)/l_1}\ (\text{E\_CA\_onLoc\_runObj})$$

$$\frac{findHeapContsAtLoc(H,\, l_1) = \texttt{nulObj}}{(S; H; L)/\texttt{contract\_assign}(l_1,\, t) \longrightarrow_e (S; H; L)/\epsilon}\ (\text{E\_CA\_onLoc\_badLoc})$$

Check Contract:

$$\frac{C/e_1 \longrightarrow_e C'/e_1' \qquad e_1' \neq \epsilon}{C/\texttt{check\_contract}(\texttt{e}_1,\, \texttt{s}) \longrightarrow_e C'/\texttt{check\_contract}(e_1',\, s)}\ (\text{E\_checkCont\_canRed})$$

$$\frac{C/e_1 \longrightarrow_e C'/e_1' \qquad e_1' = \epsilon}{C/\texttt{check\_contract}(\texttt{e}_1,\, \texttt{s}) \longrightarrow_e C'/\epsilon}\ (\text{E\_checkCont\_canRed\_ERR})$$

$$\frac{value(e_v) \qquad \nexists\, l \mid e_v = l}{C/\texttt{check\_contract}(\texttt{e}_\texttt{v},\, \texttt{s}) \longrightarrow_e C/\epsilon}\ (\text{E\_checkCont\_notLoc})$$

$$\frac{value(e_v) \qquad \exists\, l \mid e_v = l \qquad findHeapContsAtLoc(H,\, l) = t_p\, p}{\{S; H; L\}/\texttt{check\_contract}(\texttt{e}_\texttt{v},\, \texttt{s}) \longrightarrow_e \{S; H; L\}/\epsilon}\ (\text{E\_checkCont\_loc\_prim})$$

$$\frac{\begin{array}{c} value(e_v) \qquad \exists\, l \mid e_v = l \qquad findHeapContsAtLoc(H,\, l) = obj \\ \texttt{None} = getMemberFromRunObj(obj,\, s) \end{array}}{\{S; H; L\}/\texttt{check\_contract}(\texttt{e}_\texttt{v},\, \texttt{s}) \longrightarrow_e \{S; H; L\}/\epsilon}\ (\text{E\_checkCont\_loc\_noMember})$$

We also have reduction over lists of expressions. These are inspired by the same approach taken by the FWJ mechanization for their reduction over lists of expressions.

$$\frac{C/e \longrightarrow_e C'/e'}{C/(e :: \overline{e}) \longrightarrow_{\overline{e}} C'/(e' :: \overline{e})} \quad \text{(LR\_ONE)}$$

$$\frac{C/\overline{e} \longrightarrow_{\overline{e}} C'/\overline{e'}}{C/(e :: \overline{e}) \longrightarrow_{\overline{e}} C'/(e :: \overline{e'})} \quad \text{(LR\_CONS)}$$

References

[1] The State of the Octoverse 2017. https://octoverse.github.com/.

[2] Craig Chambers, David Ungar, and Elgin Lee. An efficient implementation of SELF a dynamically-typed object-oriented language based on prototypes. *ACM Sigplan Notices*, 24(10):49–70, 1989.

[3] Maxime Chevalier-Boisvert and Marc Feeley. Simple and effective type check removal through lazy basic block versioning. *arXiv preprint arXiv:1411.0352*, 2014.

[4] Maxime Chevalier-Boisvert and Marc Feeley. Interprocedural Type Specialization of JavaScript Programs Without Type Analysis. *arXiv preprint arXiv:1511.02956*, 2015.

[5] Benjamin Chung, Paley Li, Francesco Zappa Nardelli, and Jan Vitek. Kafka: Gradual typing for objects. *LIPIcs-Leibniz International Proceedings in Informatics*, 109, 2018.

[6] Benjamin Chung and Jan Vitek. Monotonic gradual typing in a common calculus. *Proceedings of 20th Workshop on Formal Techniques for Java-like Programs (FTfJP18)*, 2018.

[7] Robert Bruce Findler and Matthias Felleisen. Contracts for higher-order functions. *ACM SIGPLAN Notices*, 37(9):48–59, 2002.

[8] Andre Kuhlenschmidt, Deyaaeldeen Almahallawi, and Jeremy G Siek. Efficient gradual typing. *arXiv preprint arXiv:1802.06375*, 2018.

[9] Julian Mackay, Hannes Mehnert, Alex Potanin, Lindsay Groves, and Nicholas Cameron. Encoding Featherweight Java with assignment and immutability using the Coq proof assistant. *Proceedings of the 14th Workshop on Formal Techniques for Java-like Programs*, pages 11–19, 2012.

[10] Microsoft. Typescript – language specification version 1.8. Technical report, January 2016.

[11] Fabian Muehlboeck and Ross Tate. Sound gradual typing is nominally alive and well. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):56, 2017.

[12] Benjamin C Pierce. *Types and programming languages*. MIT Press, 2002.

[13] Benjamin C Pierce, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, and Brent Yorgey. Software foundations. *Webpage: https://softwarefoundations.cis.upenn.edu/plf-current/toc.html*, 2010.

[14] Aseem Rastogi, Nikhil Swamy, Cedric Fournet, Gavin Bierman, and Panagiotis Vekris. Safe and efficient gradual typing for TypeScript. *POPL*, 2015. doi:10.1145/2676726.2676971.

[15] Gregor Richards, Ellen Arteca, and Alexi Turcotte. The VM already knew that: leveraging compile-time knowledge to optimize gradual typing. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):55, 2017.

[16] Gregor Richards, Francesco Zappa Nardelli, and Jan Vitek. Concrete types for TypeScript. *LIPIcs-Leibniz International Proceedings in Informatics*, 37, 2015.

[17] Richard Roberts, Stefan Marr, Michael Homer, and James Noble. Shallow types for insightful programs: Grace is optional, performance is not. *arXiv preprint arXiv:1807.00661*, 2018.

[18] Jeremy G Siek, Michael M Vitousek, Matteo Cimini, and John Tang Boyland. Refined criteria for gradual typing. *LIPIcs-Leibniz International Proceedings in Informatics*, 32, 2015.

[19] Asumu Takikawa, Daniel Feltey, Ben Greenman, Max S. New, Jan Vitek, and Matthias Felleisen. Is sound gradual typing dead? *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 456–468, 2016.

[20] Michael M Vitousek, Andrew M Kent, Jeremy G Siek, and Jim Baker. Design and evaluation of gradual typing for Python. *ACM SIGPLAN Notices*, 50(2):45–56, 2014.

[21] Philip Wadler and Robert Bruce Findler. Well-typed programs cant be blamed, 2009.

[22] Andreas Wöß, Christian Wirth, Daniele Bonetta, Chris Seaton, Christian Humer, and Hanspeter Mössenböck. An object storage model for the Truffle language implementation framework. *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java platform: Virtual machines, Languages, and Tools*, pages 133–144, 2014.

[23] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. One VM to rule them all. *Proceedings of the 2013 ACM International Symposium on new ideas, new paradigms, and reflections on programming & software*, pages 187–204, 2013.