No Zombie Types: Liveness-Based Justification For Monotonic Gradual Types

by

Yangtian Zi

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, 2021

Author's Declaration

Statement of Contributions

Yangtian Zi is the sole author for Chapters 1 to 5 and 7 under supervision of Dr. Gregor Richards.

The original HiggsCheck [14] semantic rules presented in Chapter 6 and Appendix A are works of Ellen Arteca, presented in her Master's thesis [5]. The modified semantic rules presented in Chapter 6 are co-authored by Ellen Arteca and Yangtian Zi. All of the prose descriptions for the semantic rules in Chapter 6 are the sole work of Yangtian Zi.

Abstract

Gradual type systems with the monotonic dynamic semantics, such as HiggsCheck [14] implementing SafeTypeScript [13], are able to achieve decent performance, making them a viable option for JavaScript programmers seeking run-time-checkable type annotations. However, the type restrictions for objects in the monotonic dynamic semantics are, as the name suggests, monotonic. Once a typed reference is defined or assigned to refer to an object, the contract carrying the type obligation of the reference is part of the object for the remainder of execution. In some cases, such contracts become “zombies”: the reference that justifies a contract is out of scope, yet the object still retains the type obligation.

In this thesis, we propose a novel idea of contract liveness and its implementation. Briefly speaking, contracts must be justified by live stack references defined with associated type obligations. Our implementation, taking inspiration from how garbage collectors approximate object liveness by reachability of objects, approximates contract liveness by reachability of contracts. Then, to achieve a much closer approximation to contract liveness, we introduce a poisoning process: we nullify the stack references justifying the violated contract, and associate the location that triggered the contract violation with a poisoned reference for blame.

The implementation is compared with the original implementation of HiggsCheck. The comparison shows our system is fully compatible with code that raised no errors, with a small performance penalty of 8.14% average slowdown. We also discuss the performance of the contract removal process, and possible worst cases for the liveness-based system. Also, the semantics of HiggsCheck SafeTypeScript is modified to formalize the liveness-based type system. Our work proves that relaxations of contractual obligations in a gradually typed system with the monotonic semantics are viable and realistic.

Acknowledgements

First and foremost, I would like to thank my supervisor, Dr. Gregor Richards, not only for his guidance in both academic research and prose composition but also for being a great friend.

I would like to also thank Ellen Arteca, for without her help in formal semantics, this work would not be complete.

Thank you, Dr. Peter Buhr and Dr. Werner Dietl, for agreeing to read my thesis and providing valuable feedback.

I would like to extend my gratitude for friends in the Programming Languages Group and other people in University of Waterloo for the support. More specifically, I would like to say thank you to Dr. Marianna Rapoport, Abel Nieto, Dr. Nomair Naeem, Dr. Gang Lu, Dr. Prabhakar Ragde and Alexi Turcotte.

During my graduate career in University of Waterloo, there are a few people who helped me to discover my passion for teaching: Dr. Carmen Bruni, Dan Wolczuk, Dr. Sandy Graham and Dr. Graeme Turner. Thank you!

I would like to thank my friends who supported me throughout my graduate career: Josh Yang, Dylan Cheng, Luke Yao, Sherry Zhu, Yao Lu, Linqing Liu, Yuan Chen, Zhongling Wang and many more.

Last but not least, I would like to thank my parents, Jian Zi and Quanfang Yang, and my relatives, for their immense encouragement and assistance in completing this degree.

Dedication

I dedicate this thesis to 訾应举 (Yingju Zi), my grandfather, the first person who obtained a university degree in my family.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Most programming languages need a notion of types to distinguish one piece of binary data from another. In statically typed languages, variable types are checked during compilation and seldomly checked during run-time. Code generation in such languages is often based on an assumption that a variable is some given type, and that assumption is based on the results of type checking. Alternatively, variables in dynamically typed languages usually are not type-checked before the program is run; types are still checked, but they are checked at run-time, i.e., at the time of use. One major implication of this dynamic checking is that programs in dynamically typed languages are more amenable to be compiled and executed by Just-In-Time compilers. Such compilers often assign to each value a tag, a run-time value indicating the type of the object, before generating the code that uses values.

With the proliferation of the software ecosystem based on the Web, dynamically typed languages such as Ruby and JavaScript have become increasingly popular among programmers. However, programs in dynamically typed languages are often prone to errors related to types and challenging to debug. More specifically, some type errors, which would typically be caught before compilation in a static language, could be detected much later in the execution path of a dynamic program, or not detected at all. For example, while programs in statically typed languages often check for types of function arguments at function calls, programs in dynamically typed languages do not automatically check the value of the parameter. As a consequence, type errors in dynamically typed languages usually are not detected during function calls but often somewhere later in the program execution.

Erroneous execution of code might lead to unintended side effects. However, dynamically typed languages are still widely used due to their flexibility of the type system and straightforward syntax, which makes them a good choice of language for building product prototypes and minimum viable products, the products ready for users with minimal features.

Programmers who prefer dynamically typed languages, yet would like to add type annotations, could use optionally typed languages. Furthermore, those who wish to make sure type annotations in the programs are checked could use gradually typed languages. For example, JavaScript developers might turn to TypeScript [3] if they wish to add type annotations. However, TypeScript does not check its type annotations at run-time; it is optionally typed, i.e., no type checks are actually checked at run-time. This phenomenon implies that while static code in TypeScript can still be type checked, but there is no type checking for dynamic code. SafeTypeScript [13], a gradually typed language, should be used instead if those developers want to have their type annotations checked in dynamic code.

Type annotations in a gradually typed language are enforced in static parts of the program, just like static languages. However, the design goal of gradually typed languages is that statically and dynamically typed parts of the program can interact. More specifically, type annotations in the static part of the program can become obligations, to which the dynamic part of the code has to adhere. Different gradually typed languages have different definitions regarding how such obligations are checked, and the definitions are encoded in their respective operational semantics. Among these semantics, the most popular ones in the gradual typing literature are called the transient, guarded, and monotonic [19] dynamic semantics.

The transient semantics is the least strict, since type obligations follow the specific location of access and are never passed around. The guarded semantics is in between the transient and monotonic semantics in terms of strictness, for there are possibilities for type obligation to pass from one reference to other, in the case of reference assignment. The monotonic semantics is the most strict, since the type obligation follow the object; in other words, each time an object flows through typed code, the type obligations defined on references which referred to the object will be part of the object. Figure 1.1 summarizes the relationship of these semantics of gradual typing, in terms of sets of type errors they detect.

The monotonic semantics, which SafeTypeScript adopts, is able to achieve decent performance in the HiggsCheck implementation [14]. However, building type obligations in the objects, the key characteristics of this semantics, implies that type obligations obtained through references are permanent to the object once assigned. This property could produce counter-intuitive behavior to programmers. Consider the following example:

```
function sort(lst: List<number>) {
    ...
```

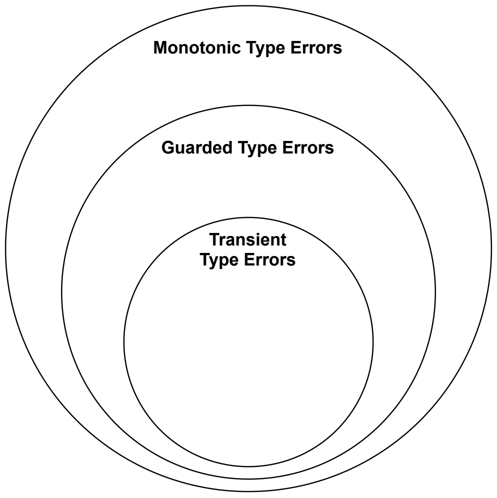

Figure 1.1: Venn diagram showing relationships in different semantics of gradual typing.

```
3 }
4 List<any> nums = {1,2,3,4};
5 sort(nums);
6 nums.first.elem = "discarded";
```

The function `sort()` has given the obligation that every element in the list must be a number. In HiggsCheck, the program does not pass the run-time type check at line 6. This behavior might appear strange: on line 5, there is a parameter assignment of a list having type `List<any>`, which is not compatible with the expected type `List<number>`. However, this hidden assignment in the function call succeeds in HiggsCheck due to the laziness property of contract checking, which we will explain in section 2.4.2. Instead, the type check fails at line 6, due to the list contains the type obligations of `List<number>` after the call to `sort()`. Effectively, the type of the list has been changed by the function `sort()`, and there is no way to remove this obligation, despite the fact that the context (i.e., execution of function `sort()`) for this obligation no longer exists. Type obligations like this are hence considered "zombies": they are no longer relevant, yet still part of an object's type obligations.

By extension, the monotonic semantics preserves these zombie type obligations in objects. This relevance, putting it more specifically, is the liveness of the references which defined the type obligations. This observation provides us a new angle regarding how types are defined in a gradually typed system: the type obligations of an object are defined by the live references that refer to them. We could alternatively say that live references justify the type obligations in objects. This thesis presents a liveness-based type system based on HiggsCheck. The key differences between a traditional monotonic gradually typed system and the proposed system is introduced. We discuss how type checking is done in the monotonic semantics, how the liveness-based gradually typed system changes the monotonic system, and what the costs are. In addition, we will present a modified semantics for the liveness-based system derived from the semantics of HiggsCheck.

Chapter 2

Background

The work presented in this thesis is based on HiggsCheck, an implementation of SafeTypeScript. SafeTypeScript is a gradually typed system using a monotonic dynamic semantics. This chapter will introduce these concepts and related topics.

2.1 Gradual Typing

Gradual typing, introduced by Taha and Siek [15], is a term to describe any type system that accepts typed code, untyped code, and any program in between [16]. That is, a programming language with a gradual type system usually inherits characteristics of both dynamically and statically typed languages. Note that the definition is broad: it includes optionally typed systems where type annotations are not checked at run-time; we will confine our discussion of gradual typing to the systems where there are run-time checks. A gradually typed language could derive from a dynamic language, hence having a semantics that is more akin to dynamic languages, such as SafeTypeScript [13]; alternatively, a gradually typed language could be based on a static language and with semantics closer to a static language, such as Groovy [1].

For example, the following function definition is valid in SafeTypeScript:

```
function trace(exn, message: string) : string {
    return exn + " threw " + message;
}
```

In this example, the variable `exn` is untyped, but `message` and the return values are typed and have type `string`. In gradually typed systems, untyped variables are treated as having the type `any`.

Implicit type checks might be performed at run-time. The following example checks if `lst` is an array or array-like object at run-time:

```
function access(lst: any[], index : number){
   return lst[index];
}
```

One problem with interactions between typed and untyped code is type compatibility of function calls. Typically, in a traditional statically typed language, subtyping is used to match arguments and parameters of functions. However, using subtyping relations in gradually typed languages poses a problem: interoperation between typed and untyped code would break. Gradually typed languages use `any` for untyped entities, and interfacing typed code to untyped code would make `any` both the top and bottom type of the type system, creating a cycle. For the sake of practicality, gradually typed languages do not use subtyping relations when checking types; instead, they use a compatibility relation [15][1], in which `any` is compatible with any type. For example, the following code would pass the static type check, and only raise an error in gradually typed systems if `y`'s run-time type is not `number`:

```
// input() gets the first token from input stream.
// Return the token as a number if the token can be parsed into one,
   string otherwise.
function input() : any { ... }

function factorial(x : number) : number { ... }

var y : any = input();
var fac_y = factorial(y);
```

It was not until 2015 that Siek et al. [16] stated a more concrete goal and definition for gradual typing: a gradually typed language should satisfy the Gradual Guarantee [16]. The Gradual Guarantee describes three characteristics a gradual language should have:

- A well-typed program remains well-typed if any of the type annotations are removed;
- A correct program remains correct if any of the type annotations are removed;
- Adding type annotations only causes errors if they are incorrect.

Along with the Gradual Guarantee, Siek et al. also introduced the Gradually Typed Lambda Calculus (GTLC) [16]. It augments the Simply Typed Lambda Calculus (STLC) by adding a set of casting-based gradual type and semantic rules. In summary, GTLC

[1]Also called consistency relation.

introduces a new type $*$ which denotes the “dynamic type” `any`, which is consistent with any type. Every function application $f\ x$ is modified so casts are inserted into f and x. Casts for functions are split into two parts: one part for the argument(s) and one part for the return value.

The variant of gradual typing mentioned above is said to employ a guarded dynamic semantics, as named by Vitousek et al. [19]. Objects in the guarded semantics can be modeled in message-passing style, where field access is also a method call on the object, which in turn can be modeled by applying a function on the object and a symbol representing the field. A reference-to-reference assignment will pass the type obligations of the right hand side reference to the reference being assigned. Type checks in the guarded semantics are often done through a “chaperon” object or “proxy” [19] that acts as an access guard. Proxies represent not a single cast but a compressed series of casts [19]; this is to improve the efficiency of nested casts.

In statically typed code, the type for data does not change unless explicit casting is performed. If there is a type violation, programmers will refer to the type definition for the entities involved to check for type inconsistencies. Data in gradually typed systems, on the other hand, does not always obtain type obligations from their definition. Type obligations might be obtained much earlier in the program execution and carried along. For example, in the guarded semantics, the following assignments pass the type obligation defined by class `A` to reference `d`:

```
var a : A = new A();
...
var b = a;
...
var c = (function (x) {return x})(b);
...
var d = c;
```

If there is a type violation on the obligation of class `A` in `d`, the definition of `a` should be considered the culprit. Thus, there is a need to separately record the location of type obligation acquisition in gradually typed systems. In GTLC, each cast also carries a unique label referring to the application that created the cast; those labels serve the purpose of recording the cast creation location in case a cast fails. Such labels serve to locate blame, i.e., the culprit where a type violation originated, and as a marker of the positions where the type obligations are acquired in gradually typed systems.

Unlike the guarded semantics, which places the checks in references, the monotonic semantics will check the consistency of such type information during each field read and write from the run-time type information in the object. Recall the example shown in the introduction:

```
1 function sort(lst: List<number>) {
2     ...
3 }
4 List<any> nums = {1,2,3,4};
5 sort(nums);
6 nums.first.elem = "discarded";
```

The type check occurs at the line 6 will test the assignment against type obligations of both `List<any>` and `List<number>`, despite the fact that the reference of access, `nums`, is defined as `List<any>`. The list object did not obtain its type obligation for `List<number>` until the program execution reached the call to `sort`, so the blame for `List<number>` being obliged on the list object would be the assignment of the parameter reference `lst` in the function `sort`.

Vitousek et al. also introduced the transient dynamic semantics [19], which places the obligation on the code that accesses the field itself. The check will only be performed at the time objects are accessed through typed references. In the above example, since `lst` is defined as an `List<any>`, the access on line 6 will not fail in the transient semantics. The transient dynamic semantics does not have a definition for blame, but it is possible to implement blame in the style of the monotonic dynamic semantics with the casting semantics of the transient dynamic semantics[18].

The following example shows how the three semantics compare to each other. Consider the following code:

```
class A { elem : any };
class B { elem : number };

function f(obj : B) {
    obj.elem = 0;
    return obj;
}

var a = new A();
var a1 = a;
var b = f(a);
b.elem = 2;
var c = b;
c.elem = 3;
a1.elem = 4;
```

In this example, the only access of `elem` through the parameter `obj` is checked against the obligation `{ elem : number }` in the transient type system, since they are the references that have explicitly defined their type definition. In addition to `a` and `obj`, accesses of `elem` through `b` and `c` are all checked against that obligation in the guarded type system since the proxy objects are passed around in definitions, assignments, and return values. In the monotonic dynamic semantics, all of the accesses, including access of `elem` through `a1`, are checked, since the object referred to by `a1` has obtained that obligation in the call `f(a)`.

2.2 Just-In-Time Compilation and Dynamically Typed Languages

A compiler is a program that translates a program in one programming language (the source language) into a machine language or another programming language (the target language). A Just-In-Time (JIT) compiler accomplishes not only exactly what a conventional compiler would do, but also executes the compiled code, which is stored in memory. A compiler that is not Just-In-Time is usually referred to as Ahead-of-Time (AOT), that is, an AOT compiler will compile to target code first, without executing it. The JIT architecture enables non-conventional code optimizations based on profiling information of previously executed code sections. A well-implemented JIT compiler can interleave compilation and execution, so that the newly compiled code can make use of such profiling information. There are even some JIT compilers that execute the code and compile upcoming code in parallel [12].

An AOT compiler makes use of type information included in the language to perform optimizations and analysis. Dynamically typed languages, due to their lack of type information, have to collect this information through profiling at run-time. Hence, they would naturally benefit more from a JIT architecture than statically typed languages. A typical implementation of a dynamic language with JIT compilation includes optimizations such as inline caches [11] and shape trees [20].

Usually, despite the lack of types in the language, a JIT compiler keeps its own record of what the type of an entity is. The "types" assigned here are called tags, which are stored in the memory alongside values or pointers to values and is read when type information is needed. For objects, we need to further distinguish them by their fields, using hidden classes [4] or shapes [20]. For example, for the expression `var + var2`, a naive approach to compile this expression would enumerate all possible types of `var` and `var2`:

```
if (is_int(var)) {
    if (is_int(var2)) {
        add_int_int(var, var2);
    } else if (is_string(var2)) {
        add_int_string(var, var2);
    } else if (...) {
        ...
    }
} else if (is_string(var)) {
    ...
}
...
```

One way we could optimize this code is to have a global or per-value cache that records the recently seen type of value. Inline caches [10] take this idea further by storing recently seen tag information of a given value in the code data structure, so the code generation step can make assumptions based on the tag of the value given the recent type tag seen for the values. Code generation may make use of these assumptions to compile code that could potentially be more efficient, if the assumption holds true. Suppose based on previous code execution that `var` and `var2` have the type `int` in the inline cache. Then, the generated code could look like:

```
if (is_int(var) && is_int(var2)){
    add_int_int(var, var2);
} else {
    // Type assumption failed.
    add_recompile(var, var2);
}
```

This code is likely faster, since in most program executions, the tag of a variable never changes. When the type assumptions fail, the code falls into the else case, which handles all other pairs of types. Note that since a JIT compiler can interleave compilation and execution, instead of running the already compiled code, the else case could be just a prompt to go back to compilation mode to compile the actual code for the else case. A variation of this technique, the polymorphic inline cache [11], stores multiple recently seen tags in a store site. Suppose that, in the above example, we see the types of `var` and `var2` now have type of `int` and `string` respectively. The generated code could look like this in a polymorphic inline caching system:

```
if (is_int(var) && is_string(var2)){
    string_concat(var, var2);
}
else if (is_int(var) && is_int(var2)){
    add_int_int(var, var2);
} else {
    // Type assumption failed.
    add_recompile(var, var2);
}
```

Object-oriented languages have objects that contain fields and methods. In implementations of dynamic languages, we need to also have a data structure that keeps the locations of fields and methods within an object. Clearly, having a per-object record would be a

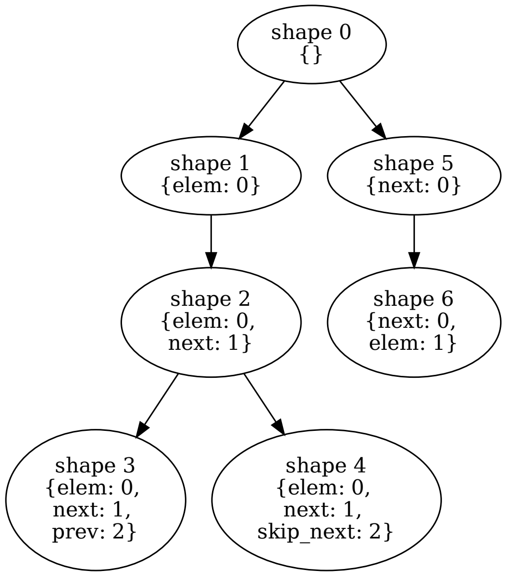

Figure 2.1: A shape tree.

waste of memory, as many object share the same list of fields and methods. A solution commonly seen in JIT compilers implementing such languages is to assign each object a shape that represents the arrangement of fields and methods in the object. Since such languages often allow new fields to be added to existing objects, different shapes are organized into a tree. The shape tree starts with the empty shape, which represents empty objects. Each time there is a new shape created by extending an existing object with an additional field, the shape tree is grown by adding a child to the node that represents the shape of that existing object. As illustration, a sample shape tree is presented in Figure 2.1. In this shape hierarchy, shapes 2, 3 and 4 could represent a simple linked list, a doubly linked list, and a skip list respectively. Due to the incremental nature of the shape tree, shape 2 and shape 6 are considered different shapes, despite having the same fields and thus interface; the only thing that differs is the index of fields within the object.

Here is another example, illustrating the incremental nature of the shape tree. Suppose we have an object that is defined as follows:

```
var obj = { a: 1, b: 2 };
```

Later the object has an additional field `c` added in, and the shape tree is changed to include

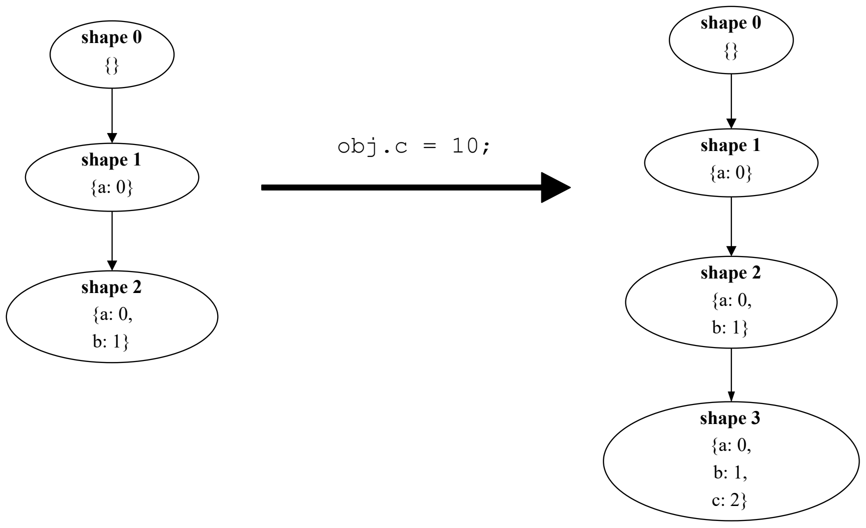

Figure 2.2: A shape tree increment with a field addition.

the new shape `{a, b, c}`, as shown in Figure 2.2.

Suppose we create another object `obj2`, this time only having one member `c`. The shape tree is going to create a new branch from the empty shape, as shown in Figure 2.3.

One important observation is that one can generalize the concept of inline caching for not only tags but other run-time properties as well. As a result, inline caches usually also cache the shape, in the case that the tag cached is an object. This type of optimization is referred to as speculative compilation, or simply speculation.

There are many other JIT compilation techniques that are widely seen in JIT compilers like On-Stack Replacement and Tiering; however, they are not discussed here, as they are not relevant to the work presented.

2.3 SafeTypeScript

TypeScript is a typed variation of JavaScript that has recently gained popularity. Its goal is to bring types to JavaScript programmers in order to provide stronger assurance that their programs are correct. However, TypeScript is intentionally unsound, since its goals are to be compatible with JavaScript and compile to it. It is an optionally typed language.

SafeTypeScript [13] is a "safe" compilation mode for TypeScript. SafeTypeScript enforces stricter type checking in TypeScript; in addition, SafeTypeScript inserts code in

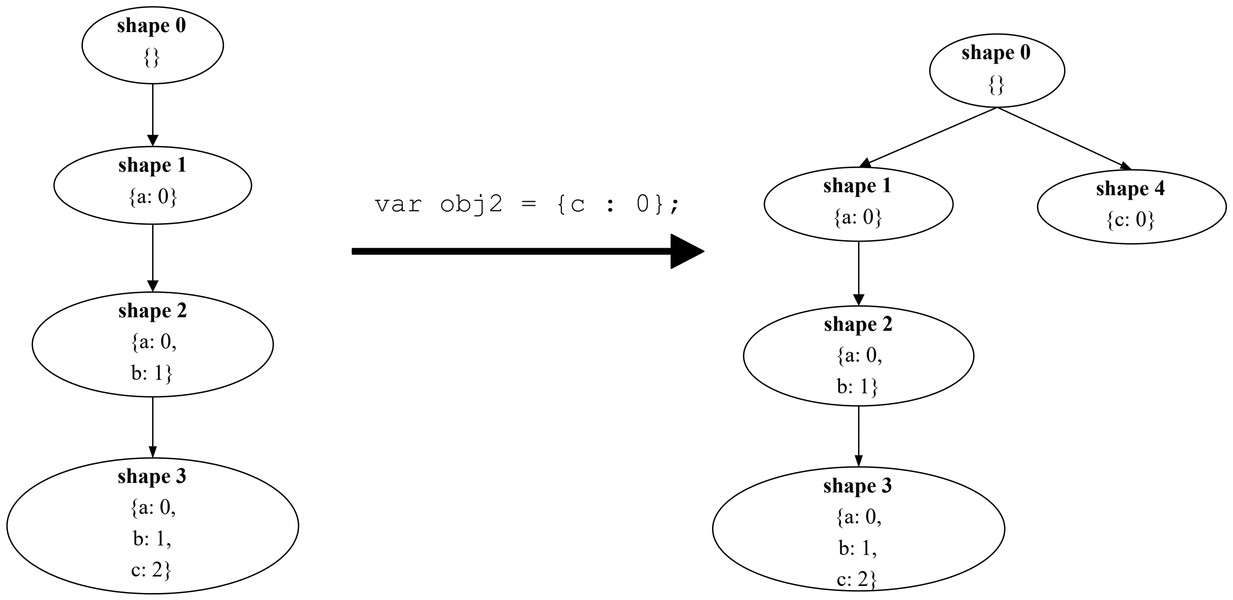

Figure 2.3: A shape tree increment with an new object.

the generated JavaScript that checks for type errors that arise at run-time. These checks make SafeTypeScript a gradually typed language; more specifically, it employs a monotonic dynamic semantics. However, the additional type checking reduces performance over equivalent JavaScript code.

2.4 Intrinsic Object Contracts

2.4.1 Higgs

Higgs [7] [9] is a JIT compiler for JavaScript for research purposes. It is known in academia due to its relatively light implementation compared to industry-grade JIT compilers. Higgs is built on a few new ideas in the implementation of dynamic languages, notably typed shapes [8] and lazy basic block versioning [7].

As mentioned in the previous section, a shape tree keeps track of all the seen object layouts in a program execution. Higgs makes use of typed shapes, a system in which objects are considered to be the same shape if all the fields are added in the same order and their types match. The shape tree's children are created once a new pair of name and

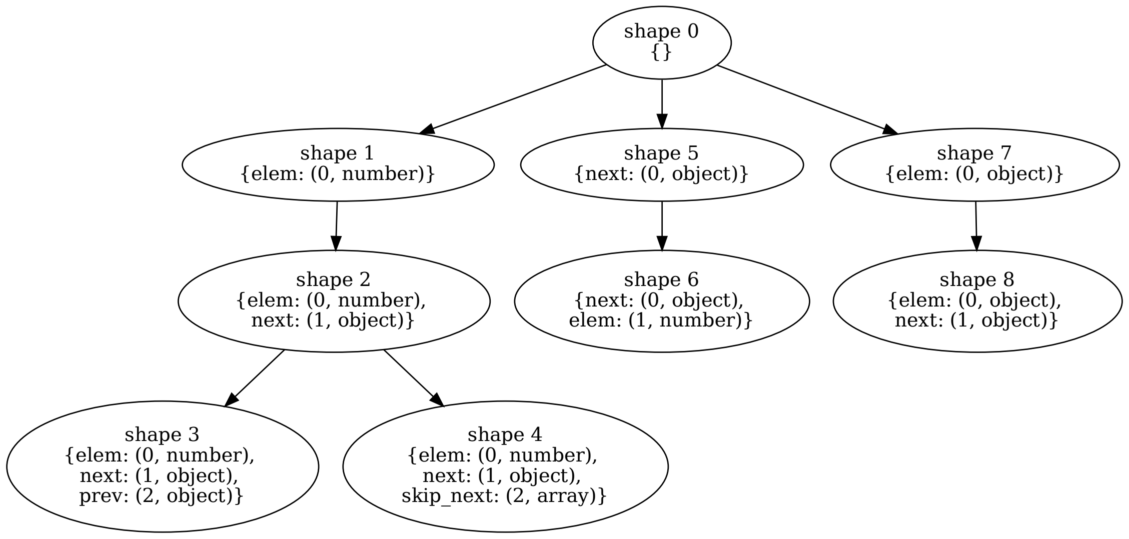

Figure 2.4: A typed shape tree. The first item of the pair is the index. The second item is the tag of the value.

type is seen. Figure 2.4 illustrates the object shape hierarchy described in Figure 2.1 under a typed shape system. Note that shape 2 in Figure 2.1 represents a linked list for all kind of types of `elem`, but a typed shape system would need a separate hierarchy for each kind of node holding elements with different tags. This difference is illustrated in shapes 7 and 8. Typed shapes are a description of the fields and its tags the compiler has seen in the execution of the program so far, but are not a promise for the future; an object with shape 2, upon the assignment of `elem` to an object, will change to shape 8.

Lazy basic block versioning is the key technique Higgs uses to perform speculative compilation. Essentially, each basic block contains small snippet of code in an intermediate representation. A basic block contains the general logic of the code, but the actual assembly code is generated from basic block versions. Each version carries assumptions regarding values, such as known tags, shapes, and register allocation data, obtained from previous contexts. Before entering a basic block, the assumptions are checked against run-time information. If the assumptions match the run-time contexts in the basic block version, Higgs proceeds to execute the code. Otherwise, Higgs will generate and compile another version of the basic block matching the current context. The generation of new block versions is on-demand, due to the possibly enormous number of versions possible for a basic block.

Higgs' memory management is basic. The stack is tagged—every value on the stack will have a tag value associated with it. The heap is divided in two spaces of equal size, and the garbage collection algorithm is Cheney's algorithm [6], the classic algorithm for a copying garbage collector. The garbage collector roots are just the references on the stack. Closures are implemented as objects in Higgs, so they do not need to be specifically treated for garbage collection purposes.

2.4.2 HiggsCheck

HiggsCheck [14] is a re-implementation of SafeTypeScript. The design of this alternative implementation is to build the type obligations, which are manifested as run-time checks in the original implementation, into the shape of the objects. The obligations in this case are referred to as intrinsic object contracts [14] (or simply contracts), since the obligations are part of the shape, which is intrinsic in the object. HiggsCheck consists of two components: a modified Higgs JavaScript compiler to implement intrinsic object contracts [14], and a modified TypeScript compiler in order to generate the runtime functions that add contracts to objects.

Contracts are run-time values built into the shape of an object but not indicating the presence of a field; they serve as an indicator of the set of type obligations for fields of the object. There are two kind of contracts: the first kind, the primitive contracts, simply represents the requirement that a field should be of some primitive type, and the higher-order contracts, which specifies that a field should reference an object, and the contract that object should follow. The contracts also follow a hierarchy: the child contract is seen as also having the obligations of its parent contracts. The contracts are checked only at field accesses. Similarly to shape trees, we can also arrange the hierarchy of contracts into a tree. To illustrate how contracts work, consider the Figure 2.5. We can see that contract 3 has both the obligations of contracts 2 and 0 through parent contracts. In addition, since contract 2 has, outside of the hierarchy, also expressed that the member `elem` must exist and must be a number, contract 3 also contains this obligation. The obligation a contract defines can also be recursive: contract 3 explicitly defines that the field `next` should follow all obligations of contract 3 itself. Essentially, contract 3 is the contract for a node of `List<number>`.

Contracts are added to the shape of the object. Adding a contract to an object is thus considered a shape change. In Figure 2.6, shape 3 is the shape for a `List<number>` in SafeTypeScript with HiggsCheck.

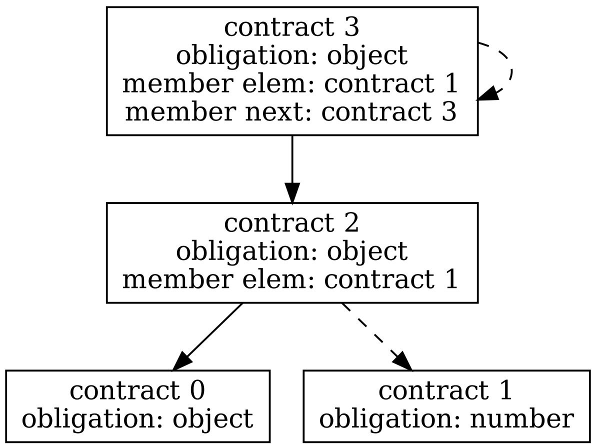

Figure 2.5: A contract tree. The dashed arrows represent the explicitly included obligations, and the solid arrows represent the parent-child relationship.

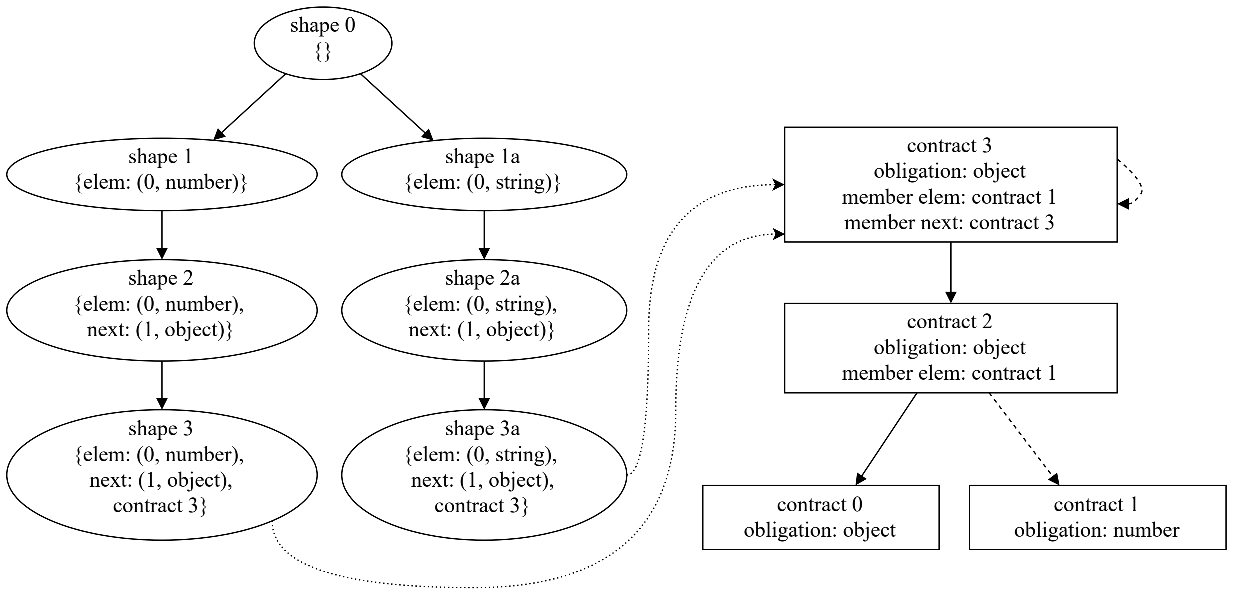

Figure 2.6: Shapes with contracts.

Shapes like 3a are contradictory at first glance: the shape indicates that `elem` is a string, but the contract indicates that `elem` must be a number. Such shapes are allowed to exist in HiggsCheck due to the fact that the contracts are checked during field access, so an object could have a shape that contains conflicting contracts. This design also allows the commonly-seen TypeScript paradigm of converting objects of type A to type B by adding fields or changing their types after the object has been initialized [14]. As an example, the conversion of an object of shape 3a to the non-self-conflicting shape 3 would be possible by just assigning `elem` with a number.

As mentioned in the previous section, code compilation in Higgs relies on the previous context, including known shape information. In HiggsCheck, contracts are considered part of shapes, so Higgs can naturally make a lot of compile time optimizations with regards to contract checks. For example, consider the following code for the implementation of `getProp`, which produces the value of the field `prop` of an object `obj`, in HiggsCheck [14]:

```
1  function getProp(obj, prop) {
2      while (!capture_shape(obj)) {}
3      var val = obj_get_prop(obj, prop);
4      if (contract_can_fail(obj, prop, val)) {
5          while (!capture_shape(val)) {}
6          if (contract_can_fail(obj, prop, val)){
7              contract_check(obj, prop, val);
8          }
9      }
10     return val;
11 }
```

The code at line 2, 4, 5 and 6 can be partially or completely evaluated at compile-time. More specifically: lines 2 and 5 capture the shape of `obj` and `val` respectively. `capture_shape` is Higgs' implementation of an inline cache. If the shape of `obj` is already known, then no code is emitted for it; otherwise, the code generated would be a comparison of the shape of `obj` with the cached value. If there is no code emitted or the shape of `obj` is the cached shape, the code proceeds to the following line. In the case of an unexpected shape, a recompilation of the code will be triggered.

`contract_can_fail(obj, prop, val)` in line 4 and 6 checks if a contract has the potential to fail. A contract can fail for three reasons: either the shape of `obj` or the tag of `val` is unknown to the context, the tag of `val` does not match one of the primitive obligations on `prop`, or the shape of `val` is not known in the case the obligation on `prop` is a higher-order one. All of those checks are performed at compile-time. In addition to checking for potential failures, `contact_can_fail` also emits code that applies higher-order contracts in `obj` on `prop` to `val`.

Finally, `contract_check` is the function that checks the run-time field values against the run-time contract value in the object. In fact, both `contract_can_fail` and `contract_check` perform contract checking, except that the former checks the compile-time assumptions and the latter checks the run-time state. For both functions, primitive contracts are checked against the tag of the value `val` immediately. Higher-order contracts are not immediately fully checked; instead, the contract for the type obligation of `obj`'s field `prop` is applied to `val`. Lines 5 and 6 are not needed in terms of correctness, but serve as an optimization: if there is a higher-order contract applied to `val`, the shape of `val` needs to be checked.

This implementation detail implies that the contracts are lazily applied to fields: for objects that have not been explicitly assigned a contract (e.g.,the subsequent nodes in a linked list), the contract is not applied to their shape unless the field referencing them is contract checked. Figure 2.7 illustrates two possible internal states of contract-applied objects, both representing the same user-observable state, using a linked list `lst` as an example. `lst` is defined to have contract `List<number>`; that is, every node object should be regarded as having the type obligation of `List<number>`. More specifically, every node object is obliged to have an element with type `number`, and have a field `next` with type `List<number>`. However, HiggsCheck lazily applies contracts to objects; therefore, when `lst` is created, only the head node object of the list have the contract in its shape.

Now, suppose the element from the third node of `lst` is accessed through the head node. The access proceeds as follows:

1. The second node `lst.next` is accessed from the head node. The head node object has contract `List<number>`, and the contract has an entry indicating the object's `next` field should have contract `List<number>`. Contract `List<number>` is thus applied to the second node.

2. The third node `lst.next.next` is accessed from the second node. In a way similar to step 1, contract `List<number>` is applied to the third node, since the contractual obligation for `List<number>` is present in the second node.

3. The element in the third node is checked against the primitive contract for `number`. If the check succeeds, the value is produced; otherwise an error is raised.

The access itself did not change the state of `lst` from the user perspective; however, the internal state of the object has changed: the contract is now applied to the first three nodes of the list.

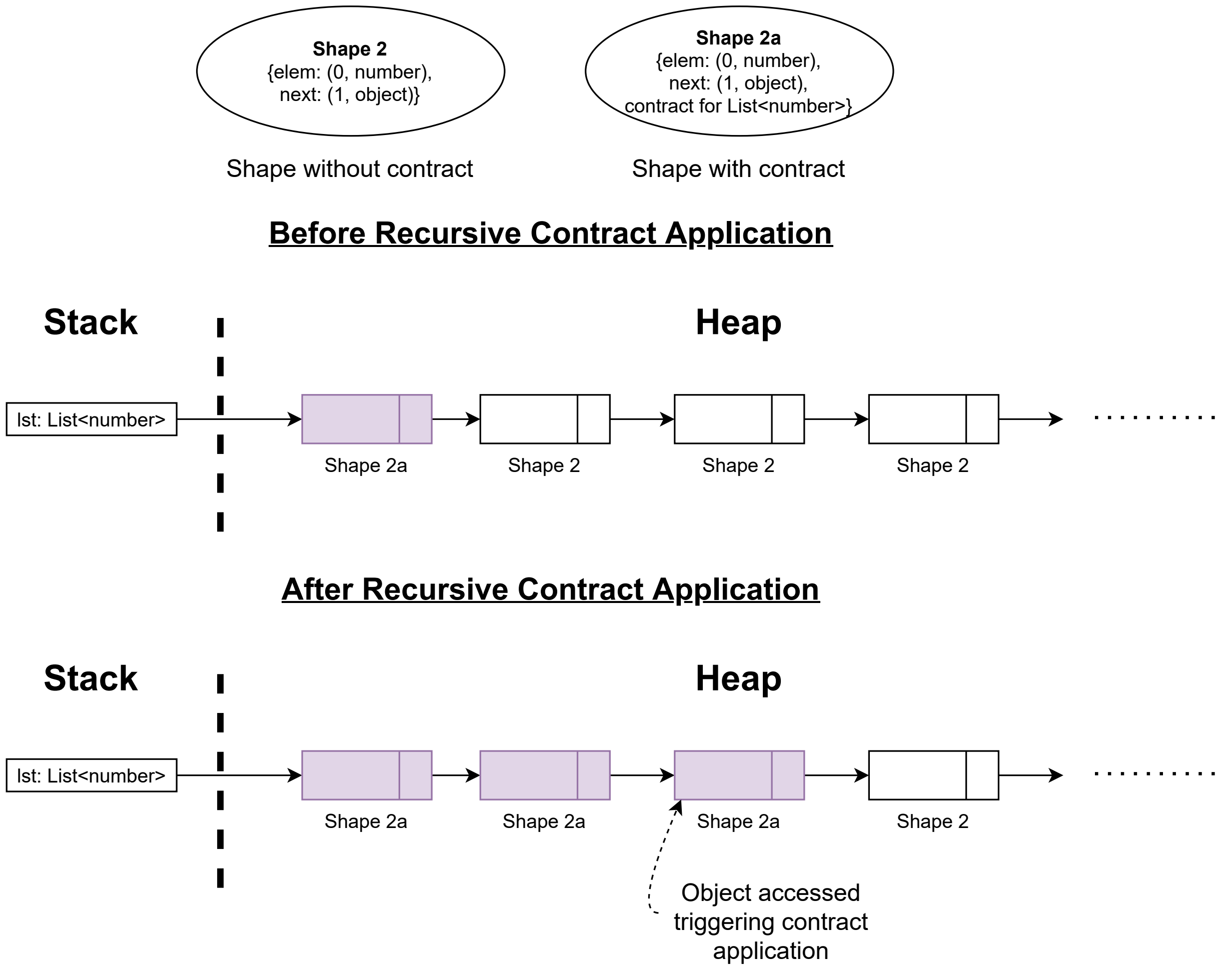

Figure 2.7: Possible contract states in program execution.

The method presented above is also generalized into other run-time functions associated with contracts. Ultimately, HiggsCheck builds the obligations into the shape, which is cached as part of the run-time information available at compile-time. As a result, HiggsCheck can eliminate a lot of unnecessary run-time checks. HiggsCheck achieved an average slowdown of 7%, and 45% in the worst case, compared to 22x slowdowns from the original implementation [14]. However, the original implementation of SafeTypeScript and HiggsCheck are both implementing the monotonic dynamic semantics, hence both inherit all of its limitations, which we will investigate in detail in the next chapter.

Chapter 3

Rechability-Based Obligation Removal

3.1 Motivation

To motivate the discussion about our proposed type system, we will formalize the example mentioned in the previous chapter. Consider the definition of a `List<T>` that supports indexing and literal initialization syntax:

```
class List<T> {
    elem: T;
    next: List<T>;

    // constructors and methods omitted
}

var l1 : List<any> = new List<any>();
var l2 : List<number> = {1,2,3};
```

Also, if a reference is defined as typed, we would say that the reference justifies the obligation (or the contract) carried by the type annotation. In the above example, we would say that `l1` justifies the contract for `List<any>` on the created object.

Here is a concrete example:

```
function foo(nums : List<number>) {
    var n: number = getTime();
    console.log(n + " nums: " + nums[0]);
}

```

```
6 var a : List<any> = {1,2,3};
7
8 foo(a);
9 a[0] = "hello!"
```
Listing 3.1: An error in the monotonic semantics.

In this example, the assignment at line 9 will fail in the monotonic semantics; the definition of `foo` ensures that the list `a` possesses the obligation that the elements in the list must be numbers, even after returning from `foo`. However, the reference `nums` which justified this obligation is defined as a parameter of `foo`, and hence is already out of scope at the time type checking fails. Although this is the defined monotonic behavior, a programmer may find the change of type by a function call surprising: `foo` does not appear to mutate any state, nor does it obviously need to be a mutating function, but the type obligations for the parameter mutated the type obligation of the object to which it refers. After calling `foo`, the object referred to by `a` now has type obligations of `List<number>`. In this case, a "type relaxation" could be possible, since the justifying reference does not exist anymore.

Formally, instead of saying some data "does not exist", such data is said not to be live. Liveness is the minimum lifespan of a value required for correct execution of the program. In this example, the stack frame of the function call in line 8, which hosted the contract justifying reference, has been popped. The contract-justifying reference, `nums`, is no longer live after the stack reference have been popped. Note that the converse of this statement is not true: a reference might still be in a stack frame, yet it is not live; this case will be discussed in detail in section 3.4. The key observation here is that we could remove obligations that none of the live references justify.

In this chapter, we are going to define the notion of live and reachable contracts, present a system where types are justified by reachable contracts, and describe when the system will be applied. We will discuss the concept first, followed by the implementation.

3.2 Approximating Liveness Using Reachability

The first question we ask ourselves is: how do we check for liveness? Unfortunately, determining the liveness information of a reference at any given code location is impossible; to do so would require predicting the future. Even the best liveness analysis tools are unable to provide liveness information for certain code, e.g., branches based on external data such as user input. The following example shows that there is no way for any compiler to predict if the reference `lastSeen` is live or not.

```
if (input() == 0){
    print(lastSeen[0])
}
```

One way we could approach this problem is to look for existing systems for inspiration. Garbage collectors use an overestimation of liveness of variables to determine liveness of objects on the heap; in fact, it assumes all of the objects that can be reached from references in root objects and the stack might be live. This property is called reachability. A reference `ref` is said to be reachable if one of the following two conditions is met:

- `ref` is on the stack.
- `ref` is the field of some object[1], and the object is referred to by a reachable reference.

This definition is recursive, in a way that every reachable reference must have a chain of references from the stack to itself. Consider the following example in JavaScript:

```
var obj = {a : {b : {c : {d : 0}}}};
```

We say that `obj` is reachable since it is defined on the stack. Consequently, `a` is reachable, since it is the field of an object `{a : ...}` which is referred to by `obj`. It is then evident that references `b` and `c` are both reachable. In addition, we say that `a` is directly referred to by `obj`, and `b` and `c` are indirectly referred to by `obj`. To access `b` from the stack reference `obj`, one must access `a`, a reference on the heap. This means `b` and `c` are reachable from the stack by the reference chain `obj` $\rightarrow$ `a` $\rightarrow$ `b` and `obj` $\rightarrow$ `a` $\rightarrow$ `b` $\rightarrow$ `c`, respectively. Extending our definition, an object referred to by a reachable reference is a reachable object.

Direct reference and indirect reference are not mutually exclusive: a reference or an object can be both directly and indirectly referred to by stack references at the same time. The following code shows the object referred to by `obj2` being an example:

```
var obj2 = {a : 0};
var obj = {b : obj};
```

Our reachability-based type system is in fact very simple: the object's type is the intersection of types from contractual obligations of its reachable references. We say that these references justify the type of the object. A reference `ref` is said to have contractual obligation of type T (or simply have contract T) if one of the following two is true for the reference:

- The reference is on the stack and is defined to have type T; or

[1]This fact implies that `ref` is on the heap.

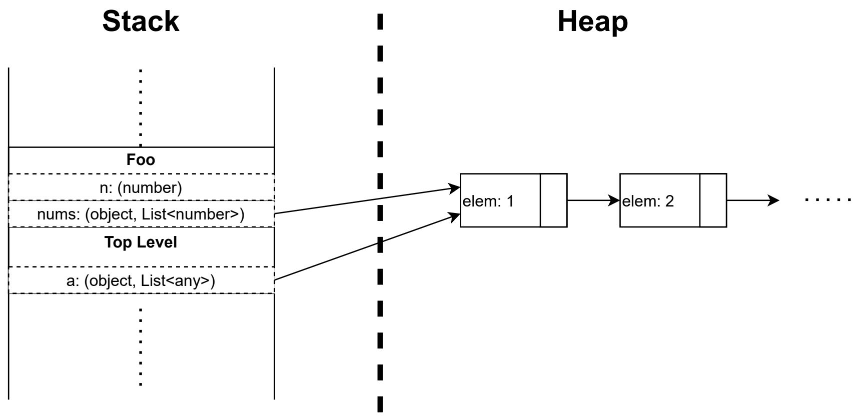

Figure 3.1: State of memory in Listing 3.1.

- an object with field `ref` has a contractual obligation of type U that contains the obligation `ref` : T.

An object is said to have contract T if there is a reachable reference having contract T referring to the object. For example, in Listing 3.1, in function foo, the object referred to by `a` should have both type obligations for `List<int>` and `List<any>`, which resolves to `List <int>`. After the call, at line 8, this object should only have type obligations for `List<any>`.

Before explaining the details of the design, we need to make sure the stack is aware of the types the program has assigned to references. This step is done by adding an element to a stack entry, which keeps track of the contract assigned to the object referred to by the stack reference. Now, a stack entry is a 3-tuple, consisting of the value, the tag, and, if applicable, the contract of the object referred. Contracts will only be assigned to references with object as their tag. Figure 3.1 shows the state of memory in Listing 3.1 with contracts on the stack added in.

A possible design for this type system is to undo the contract application for a function at the end of its scope (i.e., when the stack frame is popped); however, this solution does not scale. In the case that the contract is applied only to the top level object, this algorithm would be sufficient. Closures are implemented as objects in Higgs, so they are automatically taken into account. However, there is a possibility that the contract has been applied to some of the non-directly-reachable objects due to a contract check. In fact, there is no way to easily know the set of contract justifying objects for any object. Static points-to analyses yield inaccurate results, leaving the only option of scanning the whole heap during program execution, which is an extremely expensive operation. Here, we take the assumption that calling and returning from a function is frequent, so heap scanning for each function return will put a prohibitive toll on program execution time.

Recall that the design of the contract system is based on lazy contract application. Contracts are applied on objects only if it is directly referenced from the stack or if a field referencing the object is accessed. Contract checking will be performed if the fields with non-object tags are accessed, since contract checking for objects is potentially expensive. To check if an object is following contractual obligations, it is necessary to check the contractual obligation for all of its members, which might be objects themselves. This approach ensures that contracts checks occur only on an as-needed basis.

Back to the initial principle: we want the contract removal process to happen as infrequently as possible. To make contract checks for objects occur as infrequently as possible, the fields are checked on-demand at the time of access. Following the virtue of laziness, this can be done at the time when a contract violation is about to happen in the monotonic semantics. Consider this hypothetical problem: in SafeTypeScript, when an object flows into more strictly-typed code, the stricter obligation is usually given by the (potentially unsafe) downcasting of the object, so it can interface to the more strictly-typed code. Consider the following class definitions:

```
class Foo {
    x: any;
    y: any;
}

class Point {
    x: number;
    y: number;
}
```

In this example, `Foo` is the less strict obligation and `Point` is the stricter obligation. The following code illustrates the downcast from `Foo` to `Point`:

```
function manhattan_distance(a: Point, b: Point) {
    return abs(a.x - b.x) + abs(a.y - b.y);
}

var objFoo : Foo = new Foo({x: 0, y: 0});
var pt : Point = new Point({x: 3, y: 4});
var distance = manhattan_distance(objFoo, pt);
```

In this example, the object referred to by `objFoo` is downcasted to `Point`, when it flows into the `manhattan_distance` function. The downcast is safe within the function, since the actual type of members `x` and `y` of `objFoo` also conform to the type obligation for `Point`. However, accessing the `Foo` object via `objFoo` after this function call in a way that violates the type obligations for `Point` results in an error, as this object now contains the contract `Point`:

```
objFoo.x = "hello"; // ERROR
```

The contract violation often means that we are attempting to upcast an object that has previously been downcasted, so the object can resume its original, less strict obligations. We assume that the transition to a more strict obligation and back to a less strict obligation for any object occurs very rarely, and so it follows that the point of the contract violation is a good candidate for the contract removal process to happen.

Since each type in SafeTypeScript has a unique contract, in the case there is a contract violation, it suffices to find all stack references with the contracts that directly or indirectly justifies the obligation for the object involved in order to reaffirm the type of that object. The following example illustrates both such references.

```
1  class A {
2      b : B
3      constructor (b : B) {
4          this.b = b;
5      }
6  }
7  
8  class B {
9      ...
10 }
11 
12 var objB : B = new B();
13 var objA : A = new A(objB);
```

There are two paths of contract justification for the object created on line 12:

1. Via `objB`, which is a direct justification.
2. Via `objA`, which is an indirect justification. `objA` is justifying the contract of an object `A`, which in its own contractual obligation states that its field `b` must have contractual obligation for `B`.

Both of the contract justification paths can be seen in Figure 3.2. If the object created in line 12 needs to remove the type obligation in the object itself, both contract justification paths needs to be broken. This condition will only be met if both `objB` and `objA` have gone out of scope.

The above example showed the possibility that type obligations in objects can be justified by any stack references regardless of the contractual obligation of that reference defined

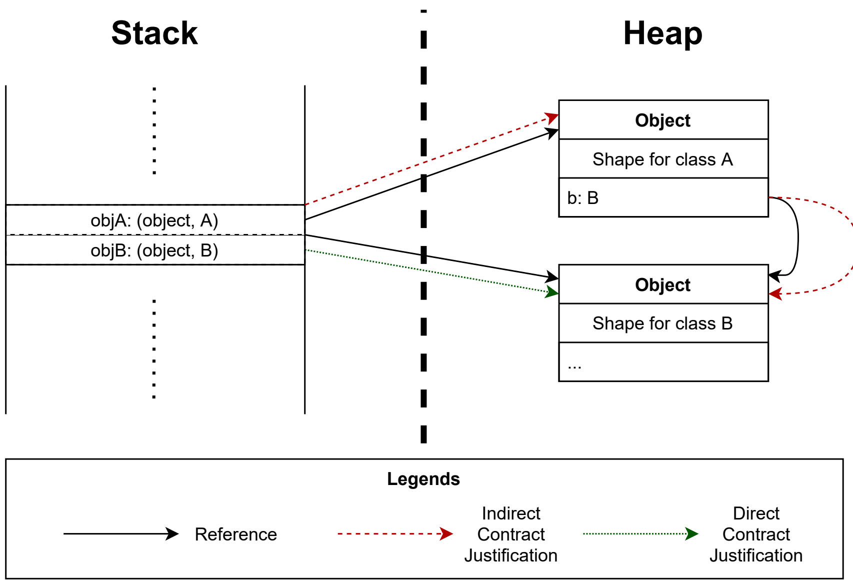

Figure 3.2: Direct and indirect contract justification.

in the code. Therefore, the whole heap must be scanned so that all objects justifying the violating contract will be checked, due to the possibility of indirect references.

In summary, to approximate contract liveness by reachability, we need to reaffirm the contracts that are reachable for the whole heap when a contract violation occurs.

3.3 Implementation

3.3.1 Changing Contracts In An Object

The monotonic semantics builds contracts into the shapes of objects, so we need to have a mechanism to change their type obligations. More specifically, there needs to be a mechanism to change the contracts in objects to a provided list of contracts, which is expected to be the set of contracts justified by reachable references. Shapes in HiggsCheck are built by adding either fields (and their tag information) or contracts into the existing shape. In Figure 3.3, shape 4 is built by adding the field elem, next, contract 3, and prev into an empty shape, in this order. The contract changing process is just rebuilding the shape of the object in the order the fields are inserted, followed by contracts. This process

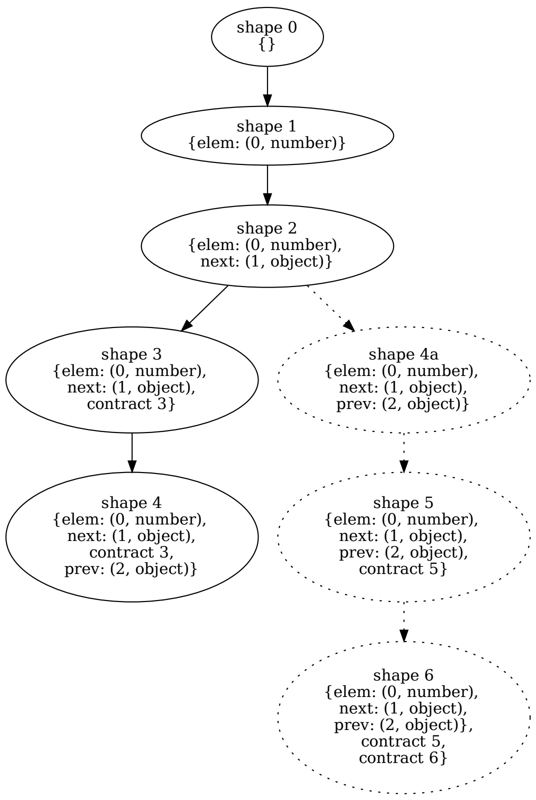

Figure 3.3: Changing contracts in shapes.

might create a new shape. In Figure 3.3, shape 4a is derived from shape 4 if an object in shape 4 removes its only contract (i.e., no justified contract for it). Shape 6 is the result if the same object has contract 5 and 6 by the contract justification process.

The following would be a possible implementation:

```
function change_contract_from_obj(obj, justified_contracts) {
    var new_shape = get_empty_shape();

    var shapes_seen = new Stack();

    for (var curr_shape = obj.shape;
         curr_shape != EMPTY_SHAPE;
         curr_shape = curr_shape.parent()) {

```

```
        if (curr_shape is not a contract) {
            shapes_seen.push_back(curr_shape);
        }
    }

    while (!shapes_seen.empty()) {
        new_shape = new_shape.get_next_shape(shapes_seen.back());

        shapes_seen.pop_back();
    }

    for (var contract: justified_contracts) {
        new_shape = new_shape.get_next_shape(contract);
    }

    obj.shape = new_shape;
}
```

The rebuilding of the shape takes a stack to remember all of the field-and-tag-pair shapes (i.e., no existing contract shapes), with the first shape inserted being the current shape, representing the most recent field-and-tag pair or contract into `obj`. A new shape for `obj` is created out of the shapes from the stack, starting from the most recently inserted field-and-tag pair. `get_next_shape` method of shapes returns an existing shape on the shape tree or a new shape if one does not exist already. After the shapes from the field-and-tag pair stack are emptied, the justified contracts are written into the new shape. Then, the new shape for `obj` is built, and the shape change is written into `obj`.

In the context of Figure 3.3, if `new_shape` is shape 1 and the `shapes_seen.back()` produces a shape that contains the field-and-tag pair `next: (1, object)`, `get_next_shape` returns shape 2, instead of creating a new identical shape. The subsequent field-and-tag pair insertion will be `prev: (2, object)`. Since there is no shape representing the new state of the object yet, shape 4a is created, added to the shape tree, and then returned by `get_next_shape`. After the field-and-tag pair insertion completes, contract 5 and 6 are inserted into `new_shape`, and at this time `new_shape` is shape 6 in the figure. At last, the new shape is assigned to the object.

3.3.2 Heap Scanning for Contracts

Scanning memory for contracts might seem to be a lot of work, but we already have a component in Higgs that scans all of memory: the garbage collector. Inspired by this idea, the first step for contract justification is a garbage collection cycle. After that, we need to obtain a list of all objects with the violated contract that either directly or indirectly refer to the object involved in the contract violation. This step is done through two parts: first, during the stack scanning in garbage collection, the garbage collector records all of the objects with their contracts directly justified by the stack. For each justified object, we keep track of the justifying objects or references per contract justified. Given the initial sets of objects with justified contracts from the stack, we expand this set by scanning the fields of existing objects for references that have obligations associated with them. After finding all justification relationships, we just need to write the justified contract into the shape of each object, analogous to garbage collection.

Our logic can be summarized into the following pseudo-code:

```
function gc_collect_with_contract_justification(vm) {
    Normal garbage collection, with contract justifications
    directly from the stack recorded

    // Find all justified contracts for all objects in the heap
    do {
        for each object that has a justified contract {
            for each justified contract found for the object {
                for each (field, obligation_on_field) in contract {

                    if field is an object {
                        record a new justifying entry:
                            (object.field has
                                obligation_on_field justified by object)
                    }

                }
            }
        }
    } while no new justified contracts has been found

    for each object and their justified_contracts {
        change_contract_from_obj(object, justified_contracts);
    }
}
```

Last but not least, the `contract_check` function, for checking contracts, will now proceed as follows: first perform the check, and if it fails, perform the garbage collection, and check the contracts once again to see if the contract violation has been cleared.

3.4 Unexpected Behavior

This implementation indeed removes undesired contracts in the case of a contract violation; however, contract invalidation has a surprising behavior. Consider the following example:

```
var lastSeen : List<int> = null;

function f(nums : List<int>) {
    lastSeen = nums;
}

var a : List<any> = {1,2,3};

f(a);
a[0] = "hello"; // contract failure
```

Listing 3.2: An example showing reachable references justifying contracts in a contract violation.

In the above example, the function `f` is defined to include a global variable `lastSeen`, which carries the type obligation of `List<int>`. The call `f(a)` does nothing except to assign the value of `a` to `lastSeen`. In the scope of function `f(a)`, `a` has the contracts of `List<any>` and `List<int>`. The following assignment operation immediately violates the latter obligation by assigning a string to the first element of `a`. Our system detects the contract error, and invokes the garbage collector with the goal of attempting to removing contracts that are no longer reachable, thus possibly removing the contract representing the obligation of `List<int>`. The attempt will fail due to the variable `lastSeen`, with a contract of `List<int>`, referencing to the same object as `a`. The assignment would then raise an error. However, in this example, it would be okay to remove the contract being violated when the error is about to occur, as the variable `lastSeen` is never used.

The above example illustrates an issue with respect to the system we just described: the implementation of the contract removal process depends on garbage collection to tell us which contracts are live. Recall that the liveness of an object or a variable is its minimal lifespan needed so the program will execute correctly. The garbage collector, in fact, does not compute the liveness of objects; it computes the reachability of objects in the heap, based on variables (i.e., references) that are on the stack. Note that not all variables on the stack are live at the time of garbage collection. In the example above, `lastSeen`, with respect to referring to the object `a`, is no longer live after the call `f(a)`, because `lastSeen` will no longer be accessed under the context of referring to `a`. Reachability is an over-approximation of liveness. If an object is live, it must be reachable, but the converse is not true: a reachable reference may not be live. Reachable-yet-not-live references exists, as there are a path from stack references that are not live to those references. Following the same logic, reachable-yet-not-live contracts exist, as there are stack references that are not live justifying those contracts.

`lastSeen` is still on the stack at the time of the contract-failure-induced garbage collection, so our system will still justify the contract `List<int>` from this reference. Yet, in this case, the removal of this contract would be perfectly reasonable, since contracts that are no longer live should not contribute any type information to the objects (including closures) it directly or indirectly refers to. This means in the semantic model we would have to model reachability, which is managed by the garbage collector. However, the concept of reachability should not be exposed to the semantics, since a garbage collector should be transparent in the semantics of the language. Our goal is to model liveness of contracts, not reachability. In the next chapter, we will propose a modification of the implementation to take account of references that are no longer live, yet still reachable.

Chapter 4

Poisoning

Who controls the past controls the future: who controls the present controls the past.

– George Orwell, *Nineteen Eighty-Four*

The last chapter presented a way to approximate contract liveness by reachability. However, we also showed that reachable-yet-not-live stack references still act as certification of contracts. Although the system was sufficient, we have not yet achieved honest-to-goodness liveness for contracts. This chapter will discuss an additional method, reference poisoning, which allows us to bypass this restriction by changing the program execution state.

4.1 Rationale

As mentioned in the previous chapter, it is impossible to compute liveness, which requires predicting the future. What can be done instead, by applying the principles of branch prediction and inline caches, is to assume a path first, and recover from the falsely-assumed state later if the assumption is wrong. In this case, we are taking the assumption that if an object is accessed with a reference and a contract violation occurs, the references justifying the contract in question are always no longer live. Broadly speaking, we are changing the state of the program, so the contract being justified in the contract violation is no longer justified. More concretely, any root (i.e., stack) reference justifying the violated contract will not be considered live and will be an “unreference”, i.e., invalidated. We call this operation reference poisoning. A poisoned reference is equivalent to a null reference in semantics when it is evaluated. Note that the reference in the contract violation itself might also be poisoned, if this reference is also justifying the violated contract. Consequently, in the case of a contract violation, the contract-violating operation will be allowed to proceed, so long as the reference involved in the contract violation is not poisoned.

Recall the example presented in the last chapter, Listing 3.2. Program execution remains the same until the contract violation at line 14. In the system with reference poisoning, `lastSeen` will be poisoned when the contract reaffirmation is invoked. The program is then allowed to proceed with `lastSeen` poisoned, until it is potentially reassigned later to another object. Suppose Listing 3.2 is modified in function `f`, and the code is run in a system with poisoning:

```
function f(nums: List<int>) {
    if (lastSeen){
        print(lastSeen[0]);
    }
    lastSeen = nums;
}
```

This time, the error will not occur at the assignment `a[0] = "hello"`. Instead, the operation will be allowed to proceed, with `lastSeen` poisoned. Note that the error occurs not at the point of contract removal, but at the time any of the poisoned variables is accessed. That means that any programmer attempting to debug any contract related run-time errors now needs three code locations: where the contract is initially assigned to the object, where the obligation associated with the contract is violated, and where the poisoned variable is accessed. Since the error happens at the access site of the poisoned variable, there is no need to remember this location. Furthermore, the contract information carried by the objects already contains blame information pointing to the code location where the contract was initially given to an object. What's left for the implementation is to keep track of the locations of contract violation at the time they happen, and to associate them with the poisoned references.

At this time, we have a system that enforces the liveness of contracts by enforcing contract justification from the stack references, in a reactive way: programs that do not have a contract violation will run without any changes to the execution state. When there is a contract violation, instead of immediately raising an error, our system will poison the stack references that justify the violated contract for the object involved.

4.2 Implementation

The implementation of poisoning needs to perform two tasks during a contract violation, mentioned above:

- Mark stack references that contain the contracts justifying the violation as poisoned.
- Record the location of contract violations and assignments, and associate them with poisoned references.

To mark the stack references, we just need to scan the stack after contract collection, and replace the value of stack references with the offending contract to null. This would functionally be correct to achieve the first goal. The second goal is to retain the contract violation location and associate it with the poisoned references. Code locations are relatively easy to obtain from the basic blocks in Higgs; the question is where to store them. Luckily, the above poisoned references are nulled, so they only have one value, `null` in terms of interfacing with the rest of the code. This design means that the actual value slot for a poisoned reference is just a word-sized location free to take over. To make use of this slot, poisoned values are given a memory address to the string representing the contract violation location. So far, our poisoned stack references will look like a value with the object tag, an associated contract, and a pointer to a string as value. This leaves a few opportunities where the compiler could misinterpret the poisoned references and treat them as actual objects. To get around this, a new tag for poison is created to distinguish poisoned references from other references.

Note that unlike the traditional implementations of garbage collectors, the garbage collector will change the state of the stack as perceived by the program in our liveness-based system with poisoning. This could invalidate Higgs' run-time contexts since we are directly modifying the values themselves. Hence, we must, as an overapproximation, invalidate all of the contexts for the current execution.

Here is the contract collection algorithm, now with poisoning:

```
function gc_collect_with_contract_justification(vm, obj_failed,
    offending_contract, offending_location) {
     Normal garbage collection, with contract justifications
     directly from the stack recorded

     // Find all justified contracts for all objects in the heap
     do {
          for each object that has a justified contract {
               for each justified contract found for the object {
                for each (field, obligation_on_field) in contract {

                    if field is an object {
                        record a new justifying entry:
                            (object.field has
                                obligation_on_field justified by object)
                    }

                }
            }
        }
    } while no new justified contracts has been found

    for each object and their justified_contracts {
        if (object == obj_failed) {
            remove offending_contract from justified_contracts
        }
        change_contract_from_obj(object, justified_contracts);
    }

    for each reference in the stack {
        if (reference justifies offending_contract for the obj_failed){
            reference.value = offending_location
            reference.tag = POISON
        }
    }

    invalidate all run-time contexts
}
```

4.3 Comparison with existing semantics

The description of the system is now complete. However, no discussion would be complete without a comparison with existing systems.

We will use the following example to motivate the discussion:

```
var m: List<any>;
var p: List<int>;

function printer(lst: List<any>) {
    print(lst[0]);
}

```

```
function foo(x: List<int>) {
    printer(x);
    m = x;
    p = x;
}

var l: List<any> = {1, 2, "hello"};
foo(l);
```

In this example, we have a reference of the list `l` with type `List<any>` with the first and second elements as `int`s and third one as a `string`. Then, the function `foo` is called with the list `l`. `foo` will in turn call `printer` with argument `x`, the list, which prints the first element of that list. Then `foo` will assign `x` to two "global" references `m` and `p` with type `List<any>` and `List<int>`, respectively. As reminder, downcasting assignments, such as passing `l`, an object with type `List<any>`, to `x`, a parameter reference expecting `List<int>`, is valid in HiggsCheck; contracts are checked lazily, so the type inconsistency in `l[2]` will only be detected if this element is accessed.

In JavaScript, we can recover from any errors (even type errors!) using exception handling, and SafeTypeScript (hence HiggsCheck) inherits this property. To keep our discussion of the semantics using shorter examples, we also assume that if any given line is reached, the previous lines have always been executed per the dynamic semantics of JavaScript, unless described otherwise. To keep our examples succinct, the error handling code is not shown.

In the following discussion, we will present different snippets of code that would run right after the above to illustrate the differences of the proposed dynamic semantics to existing ones. We will start with the monotonic semantics and the transient semantics first; then we will compare our proposed system with the guarded dynamic semantics.

4.3.1 Comparison With The Monotonic And Transient Semantics

We will start with a comparison of our liveness-based dynamic semantics with the monotonic dynamic semantics.

```
l[0] = "world";
printer(p);
p[0] = 1;
```

In this example, if the code is to run in the monotonic dynamic semantics, it would fail at `l[0] = "world";`, since the object referred to by `l` already contains the contract of

`List<int>` from the call to `foo`. The liveness-based semantics will not fail at the first line; however, `p` will be poisoned, which causes the second line to fail, since it's using a poisoned reference. For comparison, `m` is not poisoned since it does not justify the contract `List<int>`, which is violated by the assignment. The monotonic semantics, on the other hand, will never fail when executing the second and third line, regardless of whether or not a contract violation occurred previously.

In summary, our system relaxes the restriction of using a reference pointing to an object that has flowed through strictly typed code, at the cost of rendering all of references justifying the violated type obligations invalid. What we observed here is that less strictly typed code could proceed after the object passed through a more strictly typed code under the liveness-based system, unlike a system following the monotonic semantics. Based on this observation, our liveness-based semantics is less strict than the monotonic semantics.

The transient semantics only fails if an object fails to conform to its obligation at the access site. That is, the check will only be performed at the line of code where a read or write is performed, and the type used is the declared type of the reference. There is no run-time type checks in the objects in the transient semantics. Looking at the example above, a type system with the transient semantics would undergo these type checks:

- In `l[0] = "world"`, checking if `"world"` is consistent with `any`, given that `l` has type `List<any>`.
- In the call `printer(p)`, checking if `p` is consistent with `List<any>`.
- In `p[0] = 1`, checking if `1` is consistent of the member type of `p`, being `int`.

None of the above type checks fail. A program written in the transient semantics will only fail if there is a type violation when there is a direct violation to the type annotation. Indirect type obligation violations exist in the form of using an untyped or less strictly typed reference to change the state of the object. One example is the assignment `p[0] = "hello"`. As a result, our liveness-based semantics is more strict than the transient dynamic semantics.

The transient dynamic semantics is designed to only check for the expected type of the object at the time of dereference. There is no way to know when an object has been casted to some types in a system with the transient semantics. Hence, it is impossible to implement blame information in the transient semantics. Our liveness-based system can support blame, and HiggsCheck already implements blame information; furthermore, we are able to record the "poisoning blame" information which represents the location of contract failure that triggered the poisoning.

4.3.2 Comparison With The Guarded Semantics

The guarded dynamic semantics is similar to the transient dynamic semantics in terms of the location where checks are created. However, unlike the transient dynamic semantics, in which the checks are at access sites of typed references, the guarded dynamic semantics passes checks through reference assignments. In the guarded semantics, checks are accomplished by creating proxy objects; a reference-to-reference assignment does not assign the original object to which the right-hand-side reference refers, but the proxy object. The following example illustrates the differences between the guarded semantics and our system:

```
l[0] = "world";
m[0] = "world";
printer(p);
p[0] = 1;
```

The assignment `l[0] = "world"` does not give an error under either guarded or our liveness-based semantics. The assignment `m[0] = "world"`, however, will throw an error in the guarded semantics, since the assignment `m = x` in the `foo(l)` call has passed the check contained in `x` to `m`. Similar to the case mentioned above in subsection 4.3.1, the assignment of `m[0]` also poisons `p` in our liveness-based semantics (if not poisoned already by the first line); this will cause both subsequent lines of code to fail under the liveness-based semantics, again due to the fact that `p` is poisoned. However, under the guarded semantics, only the call to `printer(p)` will fail, due to the fact that `lst` inherited the obligation in `p` (i.e., `List<int>`) through parameter passing. The assignment `p[0] = 1` will not fail under the guarded semantics, in a similar way that this does not fail in the transient dynamic semantics.

Our goal is to closely approximate contract liveness; the guarded semantics would align with our liveness-based system in some cases, as seen above with the behavior of both systems in the first line. Nonetheless, if the more strictly typed code ever escapes into the continuation of execution through either reference-to-reference assignments or return values, the guarded semantics will regard objects as still more strictly typed, and raise an error if those types are violated. As shown in the above example, there are cases where the liveness-based system raises an error due to the effect of poisoning even if the type obligation is respected on the read or write itself, while the guarded semantics lets the operation proceed. Therefore, our liveness-based system is neither more nor less strict than the guarded dynamic semantics, as there are cases where the liveness-based system raised an error but the guarded semantics did not, and vice-versa.

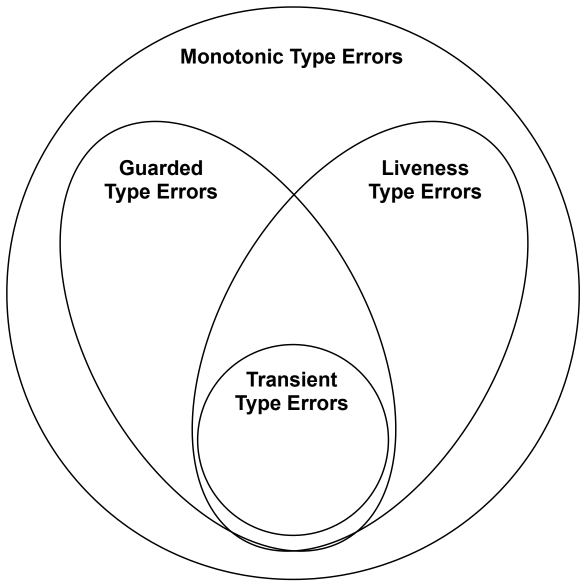

Figure 4.1: Venn diagram showing relationships of our liveness-based system with other semantics for gradual typing.

4.3.3 Summary

We have shown that the gradually typed system based on liveness is more restrictive than transient, yet less restrictive than monotonic. Furthermore, our liveness-based system agrees with the guarded semantics to raise type error in some cases, but also disagrees with it in some other cases. To summarizes this information, we added our liveness-based semantics into Figure 1.1, obtaining Figure 4.1. Still, it is worth noting that our system does involve reference liveness and its extension, contract liveness, which is a completely new dimension. None of the existing semantics take liveness of references, objects or type obligations into account; they are only different in terms of how references or objects obtain type checks throughout program execution.

Chapter 5

Evaluation and Results

The liveness-based system, just like all new software systems, needs to be evaluated. However, it is challenging to propose a suitable method to produce a fair evaluation: The proposed type system is one-of-a-kind, and there is very little meaningful existing code that would be readily available for use under the proposed system: Programs that raise type errors in gradually typed systems using the monotonic semantics are extremely rare. SafeTypeScript code is written to not have type errors under a monotonic typed system, and retrofitting JavaScript programs requires complete inspection of code so meaningful type annotations can be inserted into them. Nevertheless, this chapter attempts to evaluate the implementation and provide an analyses of the performance.

5.1 Performance on Monotonic Code

When there is no contract violation in the monotonic semantics, the liveness-based system should function exactly like HiggsCheck. An expectation for the implementation is that the contract collection and poisoning system should not increase the time spent running the code that raises no errors under the monotonic dynamic semantics. In order to evaluate this criterion, the precursor system, which is implemented by Richards et al. [14], serves as the baseline for the comparison. The benchmarks will be run with the original implementation and the system which we implemented. The source code of the original implementation has been minimally modified in order to allow compilation with the newest compiler for D[1], the language in which both systems are written.

[1]At the time of writing, the newest version for DMD compiler, which we use, is version 2.097.0 .

	Original Implementation		**Liveness-Based System**		**Slowdown**
	Mean	**Std. Dev.**	**Mean**	**Std. Dev.**	
crypto	112	1.944	135.6	2.011	21.07%
navier-stokes	167.9	2.470	189.3	2.497	12.75%
raytrace	72.1	1.449	78.3	1.889	8.60%
richards	108	1.886	110.6	2.119	2.41%
splay	648.1	8.103	666.4	5.562	2.82%
gregor	3483.9	45.975	3728.8	230.808	7.03%
morse	158	2.582	167.8	3.259	6.20%
sieve	828.5	12.349	804.9	14.693	-2.85%
snake	3649	41.891	3957.7	51.848	8.46%
suffixtree	1093.2	16.267	1175.2	19.820	7.50%
tetris	1117.1	14.731	1346.3	27.845	20.52%
tsc	43296.8	388.557	44665.4	546.771	3.16%
Average					8.14%

Table 5.1: Performance results for comparing the original HiggsCheck implementation and the liveness-based system.

The benchmarks used will be the ones used in the original HiggsCheck paper by Richards et al. [14], and consist of benchmark programs from the original implementation of SafeTypeScript [13], programs adopted from benchmarks used by Takikawa et al. [17], and the TypeScript compiler (`tsc`) itself. Only the version with intrinsic object contracts is included. The pessimal configuration listed in the HiggsCheck paper is not used, as the aspect of pessimal configuration is not relevant in our work. Each benchmarks has a warm-up phase to maximize the capabilities of JIT optimizations. Also, we raised the stack size of our liveness-based system from 2MB to 16MB, to accommodate the need of adding a contract value for each stack slot.

The benchmarks were performed on a machine with an AMD Ryzen 5800X CPU at 3.8GHz for base clock, 32GB RAM and Arch Linux with kernel `5.10.42-1-lts` [2]. Each of the benchmarks is run 10 times. The mean and standard deviation are recorded for each setup and the slowdown for each of the benchmarks is recorded. The results are shown in Table 5.1 The unit for the average run time in Table 5.1 above is milliseconds.

[2]The benchmarks were also run on another machine with AMD Opteron(TM) 6274 at 1.4GHz with 64GB memory, running Ubuntu 20.04 LTS. However, the results were deemed abnormal, for the baseline performance was enormously worse than on similarly-specced machines. The cause of this behavior is considered future work.

As the figures show, the majority of the benchmarks run with less than 10% slowdown. The benchmarks that achieve less optimal performance are navier-stokes, tetris and crypto, with the worst being crypto with 21.07% slowdown.

The results are promising in the sense that most of the benchmark programs run without too much slowdown; however, the worse performance in some benchmarks hint of the possibility of pessimal configuration that could introduce a worse slowdown. This slowdown could be caused by the extra instructions and the extra space required tracking contracts on the stack; in fact, the existence of inline caching means that contract tracking across contexts might load and unload contracts on and off the cache. The liveness-based system will take a harsher penalty on inline cache invalidations since this loading and unloading of contracts is necessary. The increase of the stack size could also contribute to the slowdown; more code is needed to manage the stack, and the stack may not fit into the first-level cache.

5.2 Performance On Code With Poisoning

The design goal of the liveness-based type system is to make sure that contracts are actually justified by liveness on the basis of a monotonic semantics configuration. As discussed in the previous section, the recovery from a contract violation, which is the direct consequence of using a lazy approach, comes with an expensive cost of a full scan of memory. We are curious about the performance impact of the work presented in the paper under different circumstances. This section attempts to provide the analysis by investigating the performance of some specific cases.

5.2.1 Poisoning At A Distance

One of the cases we are inspecting is the performance impact of poisoning at a different point of program execution: How is the performance of poisoning in the middle of execution of the code with lots of execution contexts (e.g.,stack frames) compared to poisoning after the execution of such code?

To evaluate this, we used the **gregor** benchmark [17] with a definition of a basic linked list inserted. To create a test in which the defined linked list is poisoned after the benchmark, the following code is inserted around the benchmark function:

```
var lst_poison = new List<any>(...);
benchmark();
violate_contract(lst_poison);
if (typeof lst_poison !== "poison"){
```

Poison Location	Mean	Std. Dev
Original `gregor` (no added code)	3728.8	230.808
After the benchmark	4215.6	86.558
During the benchmark	4288.6	64.234
Slowdown (after vs. during)	1.70%	

Table 5.2: Performance comparison of poisoning at a distance

```
    throw new Error();
}
```

The version which poisoning happens during the benchmark is similar to the code snippet above, except the call to `violate_contract` which is moved to a deeper stack frame inside `benchmark()`, with a guard variable to ensure that `violate_contract` is executed only once. We expect that poisoning references during the benchmark is going to be slower than poisoning after the benchmark, as the reachable memory space should shrink after the benchmark.

These benchmarks are run 10 times each, and their average running time is recorded. The performance results of those two versions is shown in Table 5.2, with the original `gregor` results listed for reference. The first column is the average running time for both configurations. The third row indicates the slowdown of poisoning during the benchmark versus poisoning after the benchmark.

As shown in the table, the mean execution time for poisoning during the benchmark is 1.7% slower than poisoning after the benchmark, which we consider as not significant.

5.2.2 Recursive Stack Frames

To further evaluate the correctness and performance of poisoning, testing is performed on recursive functions. The test is helpful to know the performance of poisoning in relation to the size of the stack. We designed the following benchmark for this purpose:

```
class A {
    a : string;
}

function violateContract(a: any): number {
    a.a = 1;
    return 0;
}
```

RECURSE_TIMES	Mean Execution Time
32	24.9
64	25.4
128	26.5
256	29.8
512	40.8
1024	83.7
2048	252.6
4096	896.1
8192	3486.0
16384	13695.7

Table 5.3: Performance results of recursive stack frames.

```

function foo(objA : A, n: number): number {
    if (n > 0) {
        foo(objA, n - 1);
    } else {
        return violateContract(objA);
    }

    if (typeof objA != "poison"){
        throw new Error("Fail: expected objA to be poison");
    }
    return 0;
}

// Warmup and time recording code omitted

var obj = new A();
obj.a = "hello world";
var retCode = foo(obj, RECURSE_TIMES);
```

In this example, we have a recursive function `foo` that contains a reference `objA`, a reference to an object of class `A`, and `n`. The class `A` contains a definition of string `a`. `foo` in the recursive case will call `foo` once again with `n` decremented by one. In the base case, this function will call `violateContract`, which violates the contract for the member `a`. `foo` recurses `RECURSE_TIMES` times; before the execution of the program, `RECURSE_TIMES` is replaced by the actual number of recursions performed.

The performance result is shown in the Table 5.3. The first column is the value of

RECURSE_TIMES, and the second column is the corresponding mean execution time of the benchmark out of 10 runs. The set of data correlates to a quadratic relation with $R^2 = 1$. This correlation is due to our heap scanning algorithm being quadratic; this inefficiency is to avoid adding bookkeeping values in the heap or using extra storage. However, we believe the performance of the liveness-based system can be improved further by changing to a faster or a more space efficient algorithm, or performing implicit contract relation detection in the garbage collection process.

5.3 Slowdowns Related To Iteration

The liveness-based type system we implemented reduces the number type errors relative gradually typed languages with the monotonic semantics, hence accepts more programs. We also stated that the flexibility comes with a cost of a garbage collection cycle and a contract reaffirmation cycle, which is somewhat expensive. This expense in time could be amplified if a contract failure happens at the wrong time. More specifically, repeated contract violations in a loop will result in repeated poisoning. Consider the following example:

```
1  function mean(lst: List<number>) : number { ... }
2
3  for (i = 0; i < n; i++) {
4      var lst = new List<any>();
5      var input: number = parse_input();
6      while (input !== undefined) {
7          lst.insert_back(input);
8          input = parse_input();
9      }
10     var lst_mean = mean(lst);
11     lst.insert_front("benchmark_results");
12     lst.insert_back(lst_mean);
13     csv_file.output(lst);
14 }
```

This example shows a program that accepts some input, stores it in a list lst, and uses lst as the output list for the destination csv_file. Since mean takes a list of numbers as its argument, the object referred to by lst has the contract List<number> when this object is passed into mean, and retains the contract after returning from it. The contract will then be removed as part of the liveness check, due to a contract violation at line 11. As the whole process of contract assignment and violation occurs in a loop, the contract collection will run at each iteration of the for loop. This will lead to a significant performance reduction.

In this example, there are a number of ways to avoid repeated poisoning: create a new list for storing the string and numerical data together instead of using `lst`, or relax the type obligation for the parameter in `mean`. Readers familiar with the monotonic dynamic semantics may point out that those are exactly the workarounds for a program running in the original SafeTypeScript; the message here is that our attempt at making monotonic-semantics-based programs more intuitive comes at a cost that might be prohibitive if it happens frequently, and idioms for writing programs in the monotonic dynamic semantics still apply.

5.4 Performance Comparison With Other Semantics

Our performance discussion with the liveness-based system and the monotonic semantics is very detailed, since our implementation is directly on top of a system with the monotonic semantics. We are not presenting a similar study here for the guarded and transient dynamic semantics since there is no equivalent TypeScript-derived system available to benchmark. Instead, we will try to discuss potential comparison of performance theoretically.

Our system is built on the works of HiggsCheck [14], implementing the monotonic semantics. HiggsCheck is capable of using run-time information, which the JIT compiler collects, to optimize dynamic type checks. However, a transient type system is unable to utilize most of the JIT profiling information for optimization, since the transient semantics does not insert run-time values in objects. The guarded semantics, however, is expected to be the worst performer, due to its use of proxy objects; allocation of proxy objects will blow up very fast in terms of execution time and memory overhead.

Another consequence of our implementation being based on HiggsCheck is that our system suffers from degradation of performance if the code is typed. In fact, the more type annotations a program has, the worse the performance gets in our system. This phenomenon is causes by the need to assign contracts on every object directly referenced in the code. The transient semantics also has a worse performance over the increase of number of type annotations; however, the performance impact is caused by an increase in the numbers of type checks. In fully typed code, there is usually no downcasts or casts to `any`. This phenomenon implies that the guarded semantics does not need to create any proxies and performs well in fully typed code. On the other hand, interleaving typed and untyped code makes a system with the guarded semantics repetitively pack objects into proxies in typed code and then perform the cast (i.e., unpack the object) in untyped code. As a result, the performance of the guarded semantics degrades when there are many interleaved sections of typed and untyped code.

Chapter 6

Formal Semantics

The formal semantics of the liveness-based contract system mostly follows the semantics of the SafeTypeScript implementation outlined in *Formal Semantics and Mechanized Soundness Proof for Fast Gradually Typed JavaScript* by Arteca [5], implementing the monotonic dynamic semantics. This chapter will highlight the changes to the semantics to support the changes described in this thesis, being a liveness-based system on top of the monotonic semantics. The complete semantic rules are available in Appendix A.

6.1 Rationale

Compared to the original implementation of HiggsCheck, the modification of the semantics should reflect the changes implemented in our liveness-based system. In particular, a few aspects of the system must be covered:

- Instead of adding contracts to the object themselves, there must be a mechanism to keep track of the live references from the stack. The contract of an object should consequently be defined by the contract of the live references that refer to the object.
- Contract checking inspects the contracts from the set of live references. If this operation succeeds, the program proceeds normally; otherwise, the live reference that carry the contract for which an obligation is violated are modified to a poison value.
- The poison value is able to associate with the contract that failed and the location of the code in which the reference is poisoned. Trying to access the poisoned value yields an error associated with this information.

Although the removal of contracts is performed lazily in the implementation, there is no need to keep this laziness in the semantics, so long as the behavior is indistinguishable to the user. Following the same principle, garbage collectors are generally not modeled in semantics. Thus, despite the fact that most of the implementation is done in the garbage collector, there no need to model it in the semantics.

An alternative way to approximate liveness is needed, since liveness can not be directly modeled, as explained previously. The approach taken here is to approximate the liveness of contracts by a stack-like data structure. The contracts are pushed onto the stack when a new reference with contracts is defined, and removed when the contracts are invalidated as a result of contract check failure. This data structure is inserted as a stack, but with arbitrary removal order. A contract check will then be checking for the associated locations on the contract stack for the presence of the obligations.

6.2 Basic Definitions

HiggsCheck's semantics [5] is a re-implementation of SafeTypeScript's semantics [13]. The liveness-based system is not very different from HiggsCheck's, with only a few differences in interaction with objects. Here are the relevant parts of the semantics, with modification to model the liveness-based dynamic semantics.

$$\text{Runtime Object} \quad O \quad ::= \quad \{l;\ \widehat{M};\ \{\overline{f_e}, \overline{f_t}\};\ \overline{l_k}\}$$

Objects in the semantics of HiggsCheck are modeled by the runtime object, comprising the location of the object, method and field declarations, and a list of locations of the contracts on the contract stack.

$$\begin{array}{llll} \text{Contract stack} & K & ::= & X \\ & & \mid & K_1, K_2 \\ \text{Context} & X & ::= & l_k \mapsto \{c, l_b\} \\ & & \mid & X_1, X_2 \end{array}$$

The contract stack is defined to contain contract contexts X, which itself contains one or more pairs consisting of contract c and poison blame location label l_b. The context is used to separate contracts defined in different scopes.

$$\begin{array}{llcl}
\text{Runtime heap} & H & ::= & l \mapsto O \\
 & & | & l \mapsto v \\
 & & | & l \mapsto w \\
 & & | & H_1, H_2 \\
\text{Poison} & w & ::= & \textit{poison label}
\end{array}$$

The runtime heap allows a location l to map to either a runtime object O or a primitive value v. For the liveness-based system, we add the poison value w to the possible values in the runtime heap, which contains a poison label representing blame information.

Last but not least, the contract stack K needs to be added onto the execution state:

$$\text{State} \quad C \quad ::= \quad S; H; L; K$$

6.3 Reduction Rules For The Untyped Language

Most of the proposed changes are in the untyped language, as the untyped language is where most of the contract-related operations take place. As mentioned, it is needed to change how contracts are assigned to objects. The function for the contract check in the semantics of HiggsCheck, `contract_assign`, needs to be redefined. `contract_assign` accepts a value v and a type t. If v is an object, `contract_assign` adds the contract to v, or checks the contract immediately if v is primitive. The latter case will only happen at the time a contract is being checked against an object that refers to v through one or more reference accesses, due to the fact that contract assignment and checking is done lazily.

Previously, the rule for contract assignment, E_CA_onLoc_runObj, adds the contract onto the runtime object.

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

The updated rule for the liveness-based system defines the contract assignment on a heap location l_1. In particular, `contract_assign` will add the contract c, with the poison label l_b, onto the current context of the contract stack, and modify the runtime object to include the location of the contract entry on the contract stack. Note that the function `contract_assign` in the semantics is different than the contract assignment function in the HiggsCheck implementation: this semantics meta-function is used for both contract assignment and checking. Hence, l_b is created for each contract check, satisfying our requirement regarding poisoning blame location.

$$\frac{\begin{array}{c} C.H[l_1] = \{l;\ \widehat{M};\ \widehat{F};\ \overline{l_k}\} \\ \mathit{isObjType}(t) \qquad c = \texttt{toContract}(t) \qquad l_c = C.K.\mathit{length} \\ X = C.K.\mathit{top}() \qquad X' = X[l_c \mapsto \{c, l_b\}] \qquad K' = K \lhd X \\ H' = C.H[l_1 \mapsto \{l;\ \widehat{M};\ \widehat{F};\ \overline{l_k}, l_c\}] \end{array}}{C/\texttt{contract\_assign}(l_1,\ t, l_b) \longrightarrow_e C \lhd H', K'/l_1} \text{(E\_CA\_onLoc\_runObj)}$$

The rule for primitives of the original HiggsCheck is straightforward: if the type of the value and the expected type matches, produce the value; otherwise, an error is raised.

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

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

In the liveness-based system, if l_1 points to a value that is of a primitive type, that means we need to check the obligation right away. If t is some primitive type t_p and the value on the heap indeed has type t_p, the check is successful and the original value is returned:

$$\frac{C.H[l_1] = t_p\ p \qquad t = \mathit{primType}\ t_p}{C/\texttt{contract\_assign}(l_1,\ t, l_b) \longrightarrow_e C/l_1} \text{(E\_contAssign\_onLoc\_prim)}$$

If the check fails or t is an object type, the heap location of the contract assignment is immediately poisoned:

$$\frac{C.H[l_1] = t_p\ p \qquad t \neq \mathit{primType}\ t_p \qquad H' = C.H[l_1 \mapsto l_b]}{C/\texttt{contract\_assign}(l_1,\ t, l_b) \longrightarrow_e C \lhd H'/l_1}$$
(E_contAssign_onLoc_prim_ERR)

The other error case, where t is a primitive type and l_1 is an object, also needs to be handled. Originally in HiggsCheck, an error will be raised immediately:

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

Now we poison the heap location:

$$\frac{\begin{array}{c} C.H[l_1] = \{l;\ \widehat{M};\ \widehat{F};\ \overline{l_k}\} \qquad isObjType(t) = \texttt{false} \\ H' = C.H[l_1 \mapsto l_b] \end{array}}{C/\texttt{contract\_assign}(l_1,\, t, l_b) \longrightarrow_e C \lhd H'/l_1}\ (\text{E\_CA\_}\textsc{onLoc\_obj\_badType})$$

Next, the way a contract is checked on an object needs to be changed as well. Originally, for a member access, the list of obligations is retrieved from the runtime object, then each obligation is tested against the retrieved member value via `contract_assign`. This test is done through a helper function *applyListOfContsFromTypes*.

$$\frac{\begin{array}{c} value(e_v) \qquad \exists\, l \mid e_v = l \qquad findHeapContsAtLoc(H,\, l) = obj \\ e = getMemberFromRunObj(obj,\, s) \qquad \overline{t} = getTypesFromRunObj(obj,\, s) \end{array}}{\{S; H; L\}/\texttt{check\_contract}(\texttt{e}_\texttt{v},\, \texttt{s}) \longrightarrow_e \{S; H; L\}/applyListOfContsFromTypes(S, \overline{t}, e)}$$
$$(\text{E\_}\textsc{checkCont\_obj\_member\_old})$$

In the liveness-based system, given some value v which should be referring to an object and some field s, checking the contracts is just applying all of the contracts that are live based on the contract stack. In the spirit of HiggsCheck, the contracts are lazily applied to the fields of the object.

$$\frac{\begin{array}{c} value(v) \qquad \exists\, l \mid v = l \qquad C.H[l] = \{l;\ \widehat{M};\ \widehat{F};\ \overline{l_k}\} \\ m = getMemberFromRunObj(\widehat{M}; \widehat{F}, s) \\ \overline{t} = \overline{(l_b, \overline{t_j})} \qquad l_i \in \overline{l_k} \cap (\bigcup_{X_i \in C.K} X_i) \qquad (c_k, l_b) = X_k[l_i] \\ (s, t_j) \in c_k \qquad X_k \in C.K \end{array}}{C/\texttt{check\_contract}(v,\, s) \longrightarrow_e C/propagateContract(\overline{t}, m)}$$
$$(\text{E\_}\textsc{checkCont\_obj\_member})$$

Here, a helper function *propagateContract* is called to achieve the goal of assigning all applicable contracts on m, the member value of the object. The other argument $\overline{t}$ is a list of pairs; more specifically, it is a grouping of all type obligations from every "live" contract imposed onto the field s, indexed in the first element of the pair by their poison label l_b.

What *propagateContract* needs to do is simply apply every type obligation $t \in \overline{t_j}$ for all $\overline{t_j}$. This contract application is achieved through another helper function, *assignContractLstTypes*.

$$\frac{e' = assignContractLstTypes(\overline{t_j}, e, l)}{C/propagateContract((l, \overline{t_j}) :: \overline{t},\ e) \longrightarrow_e C/propagateContract(\overline{t}, e')} \quad (\text{E\_propCont\_rec})$$

$$\frac{e' = assignContractLstTypes(\overline{t_j}, e, l)}{C/propagateContract((l, \overline{t_j}),\ e) \longrightarrow_e C/e'} \quad (\text{E\_propCont\_base})$$

$$\frac{e' = \texttt{contract\_assign}(e, t_1, l_b)}{C/assignContractLstTypes((t_1 :: \overline{t_j}), e', l_b) \longrightarrow_e C/assignContractLstTypes(\overline{t_j}, e', l_b)} \quad (\text{E\_contTypeList\_rec})$$

$$\frac{e' = \texttt{contract\_assign}(e, t_1, l_b)}{C/assignContractLstTypes((t_1), e, l_b) \longrightarrow_e C/e'} \quad (\text{E\_contTypeList\_base})$$

There is one thing left to do: make sure accessing a poison label yields an error.

$$\frac{C.H[l] = w}{C/l \longrightarrow_e C/\epsilon} \quad (\text{E\_poison\_access})$$

6.4 Typing Judgement

In HiggsCheck, the typing judgement determines how a typed term becomes an untyped term with the necessary setup so that the contract can be embedded into the object. The only typing judgement rules to change for the liveness-based system are the ones that assign contracts to objects, and the only change needed is to pass the poison label into `contract_assign`. Here are the typing judgement rules for `new` expression, cast and field declaration:

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

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

$$\frac{\begin{array}{c} T\ (e_1, t_1') \hookrightarrow_e e_1' \qquad t_1' <: t_1 \\ T\ (\overline{f_{nt}}, \overline{f_e}) \hookrightarrow_{fd} (\overline{f_{nt}}, \overline{f_e'}) \qquad l_b = \text{poison label} \end{array}}{T\ ((f_n, t_1) :: \overline{f_{nt}}, e_1 :: \overline{f_e}) \hookrightarrow_{fd} ((f_n, t_1) :: \overline{f_{nt}}, (\texttt{contract\_assign}(e_1', t_1, l_b)) :: \overline{f_e'})} \quad (\text{T\_FDs\_Cons})$$

6.5 Closing Thoughts

In short, we have modeled the two aspects of our liveness-based system in the semantics: reachability-based types are implemented using the contract stack, and poisoned references are implemented using a poisoning label on the heap, replacing the original reference value.

From intuition, our liveness-based system satisfies the Gradual Guarantee, since it is derived from an existing system [5] satisfying the Gradual Guarantee and strictly more restrictive than the transient dynamic semantics, which also satisfies the Gradual Guarantee [18]. However, no formal statement can be made at this time, as the proof for our claim is considered future work and yet to be completed.

Chapter 7

Conclusion and Future Work

This thesis presented an implementation that defines gradual types based on liveness. In addition, we also show the modifications required for the monotonic dynamic semantics to model the liveness-based type system. We have shown that our system is compatible with existing code written for a monotonically gradually typed system. Furthermore, our system imposes only a small performance penalty. We have compared our system with existing gradually typed systems in both behavior and performance.

The idea of poisoning presented in this thesis is entirely new; in fact, in existing implementations, it is highly unusual that a type error in the code modifies the run-time state, let alone invalidates all other references that justify the type for the object. However, the program is allowed to proceed from a contract violation only if the violated contract is not justified by the object involved in the contract violating access. Thus, such underestimation of contract liveness does not hinder the correctness of the type system.

As explained, recovering from a contract violation is costly due to the need for contract scanning of all reachable values. There are a few possible paths to reduce the number of occurrences of contract violations. For example, if it is confirmed that a reference with contractual obligations of a class A never escapes the context of some scope, we can immediately remove the contract of class A when that scope has finished execution. One way to obtain such confirmation is through static analyses such as escape analysis. In addition, Higgs has frameworks for approximating liveness of variables; despite that, we have not made use of this system in the work presented in this thesis. In the future, we may investigate ways of incorporating static analyses with our system, with the goal of improving the performance.

During the introduction of our implementation, we explained the necessity of associating contract values with stack references. This modification triggered another modification for how inline caching is implemented in Higgs: now we need to put the contracts on the cache to make sure the contract is properly stored. This change presents a new opportunity for Just-In-Time compilation optimization: is there some way to make use of this in-cache contract information to further optimize code generation, just like in-cache tag information?

Liveness-based type systems are rarely considered in academic literatures due to the impossibility of keeping liveness information in program execution. We present a new approach for implementing liveness in a lazy way. Types are added monotonically onto objects; the only time that types are justified through liveness is when a type error is encountered under the monotonic typing scheme. One potential topic for future investigation is the possible use of the technique presented here to other gradually-typed systems, or broadly, any type system with run-time type information.

In this thesis, we presented the possibility of recovering from what would be a run-time type error in a system employing the monotonic dynamic semantics. Although such errors can occur, they are mostly hypothetical, without any observations in real code. One question we can ask here is how applicable our liveness-based type system is in day-to-day code. To answer this question, a user study needs to be conducted. Such a user study would inspect a large quantity of dynamic code with type annotations (e.g.,Python code with type annotations or TypeScript code). The goal is to ascertain the frequency of liveness-based type obligation relaxation required to complete execution if the program is to be run under a monotonically gradual type system.

We have described the semantics for the system on the basis of the semantics for HiggsCheck [5]. However, unlike this previous work, we did not prove any standard theorems. We believe that our liveness-based system is sound by satisfying the soundness property, including the Gradual Guarantee, yet are unable to make the claim due to lack of formal proof. We will be looking at integrating the rules into the existing mechanized proof for progress and preservation for HiggsCheck [5], in order to prove our modification to HiggsCheck keeps the soundness of the type system.

Ultimately, we explored integrating the idea of liveness of references and objects in the monotonic semantics of gradual typing. Discussions of liveness in type system are few and far between. We hope this thesis serves as an attempt to begin the study of other techniques by which liveness can improve type systems.

References

[1] The Apache Groovy programming language. groovy-lang.org.

[2] Documentation - type compatibility. www.typescriptlang.org/docs/handbook/type-compatibility.html.

[3] TypeScript. www.typescriptlang.org.

[4] Wonsun Ahn, Jiho Choi, Thomas Shull, María J. Garzarán, and Josep Torrellas. Improving JavaScript performance by deconstructing the type system. page 496–507, 2014.

[5] Arteca, Ellen. Formal semantics and mechanized soundness proof for fast gradually typed JavaScript. Master's thesis, University of Waterloo, 2018.

[6] C. J. Cheney. A nonrecursive list compacting algorithm. *Commun. ACM*, 13(11):677–678, November 1970.

[7] Maxime Chevalier-Boisvert and Marc Feeley. Simple and effective type check removal through lazy basic block versioning. *CoRR*, abs/1411.0352, 2014.

[8] Maxime Chevalier-Boisvert and Marc Feeley. Extending basic block versioning with typed object shapes. *CoRR*, abs/1507.02437, 2015.

[9] Maxime Chevalier-Boisvert and Marc Feeley. Interprocedural type specialization of JavaScript programs without type analysis. *CoRR*, abs/1511.02956, 2015.

[10] L. Peter Deutsch and Allan M. Schiffman. Efficient implementation of the Smalltalk-80 system. page 297–302, 1984.

[11] Urs Hölzle, Craig Chambers, and David Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. pages 21–38, 1991.

[12] Ross McIlroy. Background compilation. https://v8.dev/blog/background-compilation, Mar 2018.

[13] Aseem Rastogi, Nikhil Swamy, Cédric Fournet, Gavin Bierman, and Panagiotis Vekris. Safe & efficient gradual typing for TypeScript. *SIGPLAN Not.*, 50(1):167–180, January 2015.

[14] Gregor Richards, Ellen Arteca, and Alexi Turcotte. The VM already knew that: Leveraging compile-time knowledge to optimize gradual typing. *Proc. ACM Program. Lang.*, 1(OOPSLA), October 2017.

[15] Jeremy Siek and Walid Taha. Gradual typing for functional languages. *Scheme and Functional Programming*, Janurary 2006.

[16] Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, and John Tang Boyland. Refined Criteria for Gradual Typing. 32:274–293, 2015.

[17] Asumu Takikawa, Daniel Feltey, Ben Greenman, Max S. New, Jan Vitek, and Matthias Felleisen. Is sound gradual typing dead? *SIGPLAN Not.*, 51(1):456–468, January 2016.

[18] Michael Vitousek, Cameron Swords, and Jeremy Siek. Big types in little runtime: open-world soundness and collaborative blame for gradual type systems. *ACM SIGPLAN Notices*, 52:762–774, 05 2017.

[19] Michael M Vitousek, Andrew M Kent, Jeremy G Siek, and Jim Baker. Design and evaluation of gradual typing for Python. pages 45–56, 2014.

[20] 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. page 187–204, 2013.

APPENDICES

Appendix A

Full Semantics

This complete semantics for HiggsCheck with liveness-based types is derived from the original semantics of HiggsCheck SafeTypeScript [5].

A.1 Grammar

A.1.1 Typed Grammar

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*
		$\mid$	**string**	*string type*
		$\mid$	**bool**	*boolean type*
		$\mid$	**void**	*void type*
Type	t	$::=$	t_c	*primitive type*
		$\mid$	$\{M; F\}$	*struct type*
		$\mid$	`Any`	*dynamic type*
		$\mid$	C	*nominal class type*
		$\mid$	I	*nominal interface type*
Field decl	F	$::=$	$\overline{(\ f, t)}$	*field declaration*
Method decl	M	$::=$	$\overline{(\ m, t_m)}$	*method declaration*
Method type	t_m	$::=$	$\overline{(\ t)}\ , t$	*method type*
Literals	v_c	$::=$	*number literal*	
		$\mid$	*string literal*	
		$\mid$	*boolean literal*	
		$\mid$	**undefined**	

Location	l	$::=$	`nil`	*invalid location*
		$\mid$	n	$n \in \mathbb{N}$
Variable	x	$::=$	*variable identifier*	
Interface defn	$\widehat{I}$	$::=$	`baseI`	*root interface*
		$\mid$	`interface` I `implements` $\overline{I_i}\ \{M; F\}$	*I with body*
Class defn	$\widehat{C}$	$::=$	`baseC`	*root class*
		$\mid$	`class` C `extends` C'`implements` $\overline{I_i}\ \{\widehat{M}; F\}$	*C body*
Method Defn	$\widehat{M}$	$::=$	$\overline{(\ m, t_m)}\ ;\ \overline{(\ \overline{(\ x, t)}\ : t\,\{s;\ \texttt{ret}\ \ e\})}\}$	*method defn*
Field defn	$\widehat{F}$	$::=$	$\overline{(\ f, t)}\ ;\ \overline{e}$	*field defn*
Value	v	$::=$	l	*location*
		$\mid$	$t_c\ v_c$	*primitive*

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

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

A.1.2 Runtime Grammar

Runtime obj	O	$::=$	$\{l;\ \overline{\widehat{M}};\ \{\overline{f_e}, \overline{f_t}\};\ \overline{l_k}\}$	
Prim shape	p_s	$::=$	t_p	*primitive type*
		$\mid$	m	*method name*
		$\mid$	C	*class name*
		$\mid$		*struct shape*
Obligations	k	$::=$	p_s	*primitive obl*
		$\mid$	$f_n,\ t$	*field obl*
		$\mid$	$m,\ t_m$	*member fct obl*
		$\mid$	t	*fct arg obl*
		$\mid$	t	*fct return obl*
Contract stack	K	$::=$	X	*context*
		$\mid$	K_1, K_2	*recursive case*
Context	X	$::=$	$l_k \mapsto \{c, l_b\}$	*contract and poison/blame label*
		$\mid$	X_1, X_2	*recursive case*
Runtime heap	H	$::=$	$l \mapsto O$	*runtime object*
		$\mid$	$l \mapsto v$	*primitive value*
		$\mid$	$l \mapsto w$	*poison value*
		$\mid$	H_1, H_2	*recursive case*
Poison	w	$::=$	*poison tag*	
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*
State	C	$::=$	$S; H; L; K$	

A.2 Typing Judgement

A.2.1 Expressions

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

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

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

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

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

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

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

$$\frac{T\ (\overline{f_e}, \overline{f_t}) \hookrightarrow_{\overline{e}} \overline{f_e'}}{T\ (\ l; \overline{\widehat{M}}; \overline{f_e}, \overline{f_t} cs, getStructTypeFromDefns(\ \overline{\widehat{M}}, \overline{f_e}, \overline{f_t})\ )\ \hookrightarrow_e l; \overline{\widehat{M}}; \overline{\widehat{M}}}$$
(T_ObjNotation)

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

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

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

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

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

A.2.2 Fields and Methods

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

$$\frac{\begin{array}{c} T\ (e_1, t_1') \hookrightarrow_e e_1' \qquad t_1' <: t_1 \\ T\ (\overline{f_{nt}}, \overline{f_e}) \hookrightarrow_{fd} (\overline{f_{nt}}, \overline{f_e'}) \qquad l_b = \text{poison label} \end{array}}{T\ ((f_n, t_1) :: \overline{f_{nt}}, e_1 :: \overline{f_e}) \hookrightarrow_{fd} ((f_n, t_1) :: \overline{f_{nt}}, (\texttt{contract\_assign}(e_1', t_1, l_b)) :: \overline{f_e'})} \qquad \text{(T\_FDs\_Cons)}$$

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

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

A.2.3 Statements

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

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

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

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

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

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

$$\frac{T\ s_1 \hookrightarrow_s s_1' \qquad T\ s_2 \hookrightarrow_s s_2'}{T\ s_1;\ s_2 \hookrightarrow_s s_1';\ s_2'} \tag{T_Seq}$$

A.3 Reduction Rules

A.3.1 Rules for Statements

Given metavariables s and C over untyped statements and runtime states respectively, define the following relation as "statement s with runtime state C will step to s' with runtime state changed to C'".

$$C/s \longrightarrow_s C'/s'$$

The runtime states is defined as a combination of four entities:

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

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

Value:

$$\frac{v \neq \epsilon \qquad value(v)}{C/v \longrightarrow_s C/\texttt{skip}} \tag{S_Val}$$

Sequences:

$$\frac{}{C/\texttt{skip}; s \longrightarrow_s C/s} \tag{S_Seq_Skip}$$

$$\frac{C/s_1 \longrightarrow_s C'/s_1' \qquad s_1' \neq \epsilon}{C/s_1; s_2 \longrightarrow_s C'/s_1'; s_2} \tag{S_Seq_canRed}$$

$$\frac{C/s_1 \longrightarrow_e C'/\epsilon}{C/s_1; s_2 \longrightarrow_s C/\epsilon} \tag{S_Seq_canRed_ERR}$$

$$\frac{}{C/\epsilon; s \longrightarrow_s C/\epsilon} \tag{S_Seq_ERR}$$

Variable definition:

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

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

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

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

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

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

Variable update:

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

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

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

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

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

Static field update:

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

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

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

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

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

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

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

Dynamic field update:

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

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

$$\frac{f = toString(v) \qquad H' = updateObjFieldWithVal(C.H,\ f,\ l,\ (t\ p))}{C/l_1[v] := t\ p \longrightarrow_s C \lhd H'/t\ p} \quad (\text{S\_DFldUpdLit\_prim})$$

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

If statements:

$$\frac{v = \textbf{bool}\ \texttt{true}}{C/\texttt{if}(v)\{s_1\}\ \texttt{else}\{s_2\} \longrightarrow_s C/s_1} \qquad (\text{S\_I}\textsc{f\_boolV}\textsc{al\_true})$$

$$\frac{v = \textbf{bool}\ \texttt{false}}{C/\texttt{if}(v)\{s_1\}\ \texttt{else}\{s_2\} \longrightarrow_s C/s_2} \qquad (\text{S\_I}\textsc{f\_boolV}\textsc{al\_false})$$

$$\frac{value(v) \qquad \nexists\, b \mid v = \textbf{bool}\ b}{C/\texttt{if}(v)\{s_1\}\ \texttt{else}\{s_2\} \longrightarrow_s C/\epsilon} \qquad (\text{S\_I}\textsc{f\_badV}\textsc{al})$$

$$\frac{C/e \longrightarrow_e C'/e' \qquad e' \neq \epsilon}{C/\texttt{if}(e)\{s_1\}\ \texttt{else}\{s_2\} \longrightarrow_s C'/\texttt{if}(e')\{s_1\}\ \texttt{else}\{s_2\}} \qquad (\text{S\_I}\textsc{f\_canR}\textsc{ed})$$

$$\frac{C/e \longrightarrow_e C/\epsilon}{C/\texttt{if}(e)\{s_1\}\ \texttt{else}\{s_2\} \longrightarrow_s C/\epsilon} \qquad (\text{S\_I}\textsc{f\_canR}\textsc{ed\_ERR})$$

Expression:

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

Method call:

$$\frac{\begin{array}{c} value(e) \qquad \exists\, l \in C.L \mid e = l \qquad \nexists\, v_i \in \overline{v} \mid v_i = \epsilon \\ C.H[l] = \{l, \widehat{M}; \widehat{F}; \overline{l_k}\} \qquad \overline{vt}, t, s, e_1 = getMethDefnFromRunObj(\widehat{M}, m) \end{array}}{C/e.m(\overline{v}) \longrightarrow_s C/subst\_stat((\overline{vt}, \overline{v}), s)}$$

$$(\text{S\_SMC}\textsc{all\_M\_S})$$

A.3.2 Rules For Expressions

Given metavariables e and C over untyped expressions and runtime contexts, define the following expression as "expression e with runtime state C will step to e' with runtime state changed to C'".

$$C/e \longrightarrow_e C'/e'$$

Object literals:

$$\frac{C/\overline{f_e} \longrightarrow_{\overline{e}} C'/\overline{f'_e} \qquad \nexists\, e_i \in \overline{f'_e} \mid e_i = \epsilon}{C/\{\widehat{M}; (\overline{f_{nt}}, \overline{f_e})\} \longrightarrow_e C'/\{\widehat{M}; (\overline{f_{nt}}, \overline{f'_e})\}} \quad (\text{E\_objLit\_canRed})$$

$$\frac{C/\overline{f_e} \longrightarrow_{\overline{e}} C'/\overline{f'_e} \qquad \exists\, e_i \in \overline{f'_e} \mid e_i = \epsilon}{C/\{\widehat{M}; (\overline{f_{nt}}, \overline{f_e})\} \longrightarrow_e C'/\epsilon} \quad (\text{E\_objLit\_canRed\_ERR})$$

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

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

Contract assignment:

$$\frac{C/e_1 \longrightarrow_e C'/e'_1}{C/\texttt{contract\_assign}(e_1, \tau, l_b) \longrightarrow_e C'/\texttt{contract\_assign}(e'_1, \tau, l_b)}$$
(E_contAssign_canRed)

$$\frac{C.H[l_1] = t_p\ p \qquad t = primType\ t_p}{C/\texttt{contract\_assign}(l_1, t, l_b) \longrightarrow_e C/l_1} \quad (\text{E\_contAssign\_onLoc\_prim})$$

$$\frac{C.H[l_1] = t_p\ p \qquad t \neq primType\ t_p \qquad H' = C.H[l_1 \mapsto l_b]}{C/\texttt{contract\_assign}(l_1, t, l_b) \longrightarrow_e C \lhd H'/l_1}$$
(E_contAssign_onLoc_prim_ERR)

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

$$\frac{C.H[l_1] = \{l;\ \widehat{M};\ \widehat{F};\ \overline{c}\} \qquad \mathit{isObjType}(t) = \texttt{false} \\ H' = C.H[l_1 \mapsto l_b]}{C/\texttt{contract\_assign}(l_1,\, t, l_b) \longrightarrow_e C \lhd H'/l_1} \text{(E\_CA\_onLoc\_obj\_badType)}$$

$$\frac{C.H[l_1] = \{l;\ \widehat{M};\ \widehat{F};\ \overline{l_k}\} \\ \mathit{isObjType}(t) \qquad c = \texttt{toContract}(t) \qquad l_c = C.K.\mathit{length} \\ X = C.K.\mathit{top}() \qquad X' = X[l_c \mapsto \{c, l_b\}] \qquad K' = K \lhd X \\ H' = C.H[l_1 \mapsto \{l;\ \widehat{M};\ \widehat{F};\ \overline{l_k}, l_c\}]}{C/\texttt{contract\_assign}(l_1,\, t, l_b) \longrightarrow_e C \lhd H', K'/l_1} \text{(E\_CA\_onLoc\_runObj)}$$

$$\frac{C.H[l_1] = \texttt{nulObj}}{C/\texttt{contract\_assign}(l_1, t, l_b) \longrightarrow_e C/\epsilon} \text{(E\_CA\_onLoc\_badLoc)}$$

Contract checking:

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

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

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

$$\frac{\mathit{value}(e_v) \qquad \exists\, l \mid e_v = l \qquad C.H[l] = t_p\, p}{C/\texttt{check\_contract}(\texttt{e}_\texttt{v},\, \texttt{s}) \longrightarrow_e C/\epsilon} \text{(E\_checkCont\_loc\_prim)}$$

$$\frac{\mathit{value}(e_v) \qquad \exists\, l \mid e_v = l \qquad C.H[l] = \{l;\ \widehat{M};\ \widehat{F};\ \overline{l_k}\} \\ \texttt{None} = \mathit{getMemberFromRunObj}(\widehat{M}; \widehat{F},\, s)}{C/\texttt{check\_contract}(\texttt{e}_\texttt{v},\, \texttt{s}) \longrightarrow_e C/\epsilon} \text{(E\_checkCont\_loc\_noMember)}$$

$$\frac{\mathit{value}(e_v) \qquad \exists\, l \mid e_v = l \qquad C.H[l] = \{l;\ \widehat{M};\ \widehat{F};\ \overline{l_k}\} \\ e = \mathit{getMemberFromRunObj}(\widehat{M}; \widehat{F},\, s) \qquad q = \overline{l_k} \cap \text{ all } X_i \in C.K \\ \overline{t} = \overline{(l_b, \overline{t_j})} \qquad l_i \in q \qquad (c_k, l_b) = X_k[l_i] \qquad (s, t_j) \in c_k \qquad X_k \in C.K}{C/\texttt{check\_contract}(\texttt{e}_\texttt{v},\, \texttt{s}) \longrightarrow_e C/\mathit{propagateContract}(\overline{t}, e)}$$
(E_checkCont_obj_member)

Poisoning-related auxiliary functions:

$$\frac{e' = assignContractLstTypes(\overline{t_j}, l, e)}{C/propagateContract((l, \overline{t_j}) :: \overline{t},\ e) \longrightarrow_e C/propagateContract(\overline{t}, e')} \quad (\text{E\_propCont\_rec})$$

$$\frac{e' = assignContractLstTypes(\overline{t_j}, l, e)}{C/propagateContract((l, \overline{t_j}),\ e) \longrightarrow_e C/e'} \quad (\text{E\_propCont\_base})$$

$$\frac{e' = \texttt{contract\_assign}(e, t_1, l_b)}{C/assignContractLstTypes((t_1 :: \overline{t_j}),\ l_b) \longrightarrow_e C/assignContractLstTypes(\overline{t_j}, e', l_b)} \quad (\text{E\_contTypeList\_rec})$$

$$\frac{e' = \texttt{contract\_assign}(e, t_1, l_b)}{C/assignContractLstTypes((t_1),\ l_b) \longrightarrow_e C/e'} \quad (\text{E\_contTypeList\_base})$$

$$\frac{C.H[l] = w}{C/l \longrightarrow_e C/\epsilon} \quad (\text{E\_poison\_access})$$

List of expressions:

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

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