The VM Already Knew That

Leveraging Compile-Time Knowledge to Optimize Gradual Typing

GREGOR RICHARDS, University of Waterloo, Canada, Canada
ELLEN ARTECA, University of Waterloo, Canada, Canada
ALEXI TURCOTTE, University of Waterloo, Canada, Canada

Programmers in dynamic languages wishing to constrain and understand the behavior of their programs may turn to gradually-typed languages, which allow types to be specified optionally and check values at the boundary between dynamic and static code. Unfortunately, the performance cost of these run-time checks can be severe, slowing down execution by at least 10x when checks are present. Modern virtual machines (VMs) for dynamic languages use speculative techniques to improve performance: If a particular value was seen once, it is likely that similar values will be seen in the future. They combine optimization-relevant properties of values into cacheable “shapes”, then use a single shape check to subsume checks for each property. Values with the same memory layout or the same field types have the same shape. This greatly reduces the amount of type checking that needs to be performed at run-time to execute dynamic code. While very valuable to the VM’s optimization, these checks do little to benefit the programmer aside from improving performance. We present in this paper a design for *intrinsic object contracts*, which makes the obligations of gradually-typed languages’ type checks an intrinsic part of object shapes, and thus can subsume run-time type checks into existing shape checks, eliminating redundant checks entirely. With an implementation on a VM for JavaScript used as a target for SafeTypeScript’s soundness guarantees, we demonstrate slowdown averaging 7% in fully-typed code relative to unchecked code, and no more than 45% in pessimal configurations.

CCS Concepts: • **Software and its engineering** → **Just-in-time compilers**; **Runtime environments**;

Additional Key Words and Phrases: Gradual typing, run-time type checking

ACM Reference Format:
Gregor Richards, Ellen Arteca, and Alexi Turcotte. 2017. The VM Already Knew That: Leveraging Compile-Time Knowledge to Optimize Gradual Typing. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 55 (October 2017), 27 pages. https://doi.org/10.1145/3133879

1 INTRODUCTION

Dynamic languages provide flexibility, and modern implementations of dynamic languages are quite efficient. However, the dynamic nature of these languages makes it difficult for a programmer to statically constrain their behavior. For example, a function may be written in anticipation of a number as an argument, but can be called with an object, which will result in a runtime type error.

Programmers wishing to understand or constrain run-time behavior in dynamic languages have the option to use gradually-typed programming languages. Gradually-typed languages are

Authors’ addresses: Gregor Richards, School of Computer Science, University of Waterloo, Canada, 200 University Avenue West, Waterloo, Ontario, N2L 3G1, Canada, gregor.richards@uwaterloo.ca; Ellen Arteca, School of Computer Science, University of Waterloo, Canada, 200 University Avenue West, Waterloo, Ontario, N2L 3G1, Canada, earteca@uwaterloo.ca; Alexi Turcotte, School of Computer Science, University of Waterloo, Canada, 200 University Avenue West, Waterloo, Ontario, N2L 3G1, Canada, aturcotte@uwaterloo.ca.

This work is licensed under a Creative Commons Attribution 4.0 International License.

© 2017 Copyright held by the owner/author(s).
2475-1421/2017/10-ART55
https://doi.org/10.1145/3133879

typically compatible extensions of existing dynamically-typed languages, which allow typed and untyped code to interact while still providing some type soundness guarantees. Typically, run-time checks are used to verify types when values come from dynamic sources, and so can't be trusted. Implementing these checks faces a semantic mismatch, however: Types may be arbitrarily complex, and indeed recursive, and thus cannot be checked quickly. Some gradually-typed languages resolve this by restricting the type system in such a way that checks are always fast [Richards et al. 2015] [Bloom et al. 2009], but this naturally restricts the expressibility of types. Others work by delaying checks until a first-order, easily checkable value is obtained.

In a typical implementation, first-order type checking is imposed on higher-order values through run-time contracts. Run-time contracts interpose on values to check all accesses, guaranteeing that the constraints of the specified types are not violated. Checking has proved to be a major overhead in gradually typed languages [Takikawa et al. 2016]; depending on where checking is necessary, the addition of contracts can slow down the execution of software by 100 times. Other implementation techniques have been used, which can move the cost of checking, but slowdowns of 10x [Vitousek et al. 2014] to 22x [Rastogi et al. 2015] are still reported when static and dynamic code interacts. These slowdowns have led Takikawa et al. [2016] to ask, “is sound gradual typing dead?” The goal of this work is to demonstrate that with some help from the virtual machine, sound gradual typing with usable performance is still practical.

Modern virtual machines (VMs) for dynamic languages optimize based on *speculation*: If a particular value was seen once, it is likely that similar values will be seen in the future. This is possible because of just-in-time (JIT) compilation, which blurs the line between compile-time and run-time, and allows the compiler to utilize run-time information. Code may be compiled or recompiled once crucial information is known. The code generated by JITs checks that certain properties of values are similar to previously-seen values, and if they are, runs branches optimized to those assumptions.

In particular, dynamic language VMs optimize access to object fields by giving every object a “shape”, and assuring that similar objects have similar shapes. All objects with the same shape have the same fields, and are laid out identically in memory, so optimizing based on the shape allows such objects to be accessed efficiently.

Speculative JIT optimizations and gradual type checks are often checking similar properties, but they are effectively unaware of each other. In the best case, the VM may be able to speculate based on contract success, but nothing more. The goal of this work is to improve the performance of run-time contracts for gradually-typed programming languages by leveraging the existing speculative optimization framework in JITs, and taking advantage of the redundancies between these systems.

This paper illustrates the design of a contract system sufficient for the soundness checks required by SafeTypeScript, a gradually-typed language based on JavaScript, and a VM which implements these contracts with optimizations to substantially reduce the cost of run-time type checking. We call this style of contracts *intrinsic object contracts*, and implement it in the Higgs [Chevalier-Boisvert and Feeley 2016] [Chevalier-Boisvert and Feeley 2015] JavaScript research virtual machine, as HiggsCheck. We demonstrate the efficacy of this system by reimplementing SafeTypeScript [Rastogi et al. 2015] to provide its guarantees with our contract system, and show using SafeTypeScript's benchmarks that we achieve slowdown of 7% in the common case, and no worse than 45% in intentionally pessimal configurations, as compared to 22x slowdowns in the original implementation of SafeTypeScript. The presented implementation is on a JavaScript-based language on a JavaScript virtual machine, but the technique does not depend deeply on JavaScript's semantics.

The contributions of this paper are

- the design of an intrinsic contract system at the VM level,

- a technique for reducing the overhead of checking of such contracts,
- the implementation of that system on the Higgs virtual machine,
- the implementation of SafeTypeScript's soundness guarantees on these contracts,
- an evaluation, demonstrating that the overhead of this contract system is practical.

The complete source code of this system is available at http://plg.uwaterloo.ca/~dynjs/higgscheck/.

2 BACKGROUND AND GOALS

We will demonstrate the goals of this system through a simple running example. Consider the following JavaScript code, which sums the values in a list of numbers:

```
function sum(node) {
  if (node.next)
    return sum(node.next) + node.item;
  else
    return node.item;
}
```

It is evident from this code that `node` is expected to be an object, with fields `next` as a reference to a similar object or `null`, and field `item` as a number. Called with the object `{next: {next: null, item: 40}, item: 2}`, `sum` will return 42. However, due to the dynamic nature and peculiar semantics of JavaScript, in practice this function will do something—albeit perhaps something unpredictable—with many input objects. For instance, called with `{next: 42}`, which is obviously not a node in a list, `sum` will return `NaN` (Not a Number), a consequence of the odd behavior of accessing and performing arithmetic with nonexistent fields.

2.1 Optional and Gradual Typing

A programmer wishing to make their intents clearer may instead use TypeScript [Bierman et al. 2014], a language which extends, and compiles into, JavaScript, adding type annotations. In TypeScript, this function and its expected type can be expressed as:

```
interface Node {
  next: Node;
  item: number;
}
```

```
function sum(node: Node) {
  if (node.next)
    return sum(node.next) + node.item;
  else
    return node.item;
}
```

Now, the call `sum({next: 42})` will fail to compile, with an appropriate type error: Argument of type `'{ next: number; }'` is not assignable to parameter of type `'Node'`. However, TypeScript is a strictly compile-time type checker, and is designed to interact with untyped code, so it's trivial to work around this type restriction by either calling `sum` from untyped code, or using the special "*any*" type to bypass checking. Indeed, the JavaScript code above is the compiled output of this TypeScript snippet. This style of typing is often called *optional typing*, which implies that the types are, of course, optional, but also that there are no run-time guarantees that the expressed types are correct. TypeScript is intentionally unsound, allowing unchecked downcasts and covariant function overloading, but it has a sound core, and fully-typed programs written in that sound core are type sound [Bierman et al. 2014].

Gradually typed languages regain some of the assurance of statically typed languages, while maintaining the flexibility of interacting with dynamic code. They define a distinct *consistency relation*, by which a type may be consistent even if it is not a subtype, and the consistency relation is used to allow dynamic and static code to interact. Typically, a single dynamic type (in TypeScript,

any) is consistent with all types, and structural or higher-order types must be consistent in their type members. In terms of dynamic semantics, the type correctness of this relationship is enforced by injecting unsound casts, and implementing those casts as run-time checks. The standard criteria by which their correctness is measured is that adding types does not affect behavior in the absence of unsound casts, and in the presence of unsound casts, errors are always the fault of dynamic code [Siek et al. 2015a] [Wadler and Findler 2009].

The caveat is how one performs run-time checks with higher-order types. Languages such as StrongScript [Richards et al. 2015] and Thorn [Bloom et al. 2009] accomplish this by restricting the type system to nominal, Java-like classes, such that all type checks can be performed eagerly, or by allowing the programmer to explicitly specify whether checking is desired [Wrigstad et al. 2010], avoiding behavioral changes when checking is not performed. More general approaches, such as Typed Racket [Tobin-Hochstadt and Felleisen 2008], Reticulated Python [Vitousek et al. 2014] and, relevant to this work, SafeTypeScript [Rastogi et al. 2015], perform their checks lazily: When a first-order, primitive value is expected, it is checked immediately, but other checks are delayed. This approach allows for higher-order types, such as structural object types and first-class functions, in a way that nominal types cannot, and does not typically interfere with the behavior of the underlying dynamic language except to introduce checks.

Three principle techniques have emerged for delayed run-time checks in gradually-typed languages, each with their own benefits and disadvantages. In each case, if a value undergoes an unsound cast to a first-order, primitive type, it is simply checked eagerly, but casting to a higher-order type invokes some form of run-time protection.

Typed Racket and Reticulated Python (in one configuration) use wrappers, or proxies. These wrappers implement a general contract, which could notionally verify any property expressible in the host language, but are typically used for types. The relationship between contracts and types has been explored for both objects [Findler and Felleisen 2001] and functions [Findler and Felleisen 2002]. In a wrapper-based gradually-typed system, every time a value is obtained which may not have the correct type, it is wrapped in a function or object which protects it and checks accesses. This wrapper implements a run-time contract, guaranteeing that the value's behavior matches the specified type. For objects, for instance, each field is implemented as an accessor function. If the field's type is itself higher-order, this accessor wraps it; if it is first-order, the accessor checks it. Further, these wrappers contain blame information, i.e., a label of where the wrapper itself was applied. Because checks are delayed, if a check fails, the blame information stored in the wrapper will ultimately tell the programmer where the incorrect cast was performed, rather than the point where the error occurred.

Using the wrapping technique puts the cost of both allocation of wrappers and checking on both statically-typed and dynamically-typed code, for any value which has been downcast. Thus, fully-typed code incurs no cost, and fully-untyped code also incurs no cost, but mixed code can have severe slowdown: When static and dynamic code is mixed, Typed Racket reports slowdowns up to 100x [Takikawa et al. 2016].

Reticulated Python introduced two other semantics: *transient checks* and *monotonic objects*. The technique of transient checks simply explicitly checks primitive types when they're expected. That is, no check would be performed to verify that an object is a Node (except perhaps that it is an object), but if the item member of a Node-typed variable is accessed, its type is checked. Transient checking cannot preserve blame information, as higher-order downcasts have no run-time behavior.

It's likely that transient checks could be made to work well with the VM, but if they're purely superficial, they also cause significant slowdowns. Reticulated Python reports slowdowns of 10x in this configuration [Vitousek et al. 2014].

Reticulated Python's third mode, monotonic objects, assures that values in the heap which undergo unsound casts may only monotonically become more precisely typed and their behavior more constrained. So long as objects monotonically become more precisely typed, static code can be free from performing checks: Static code accesses members directly, assured that they will be correctly typed, while dynamic code is redirected to perform checks, thus not sullying the objects that static code expects. Thus, a direct implementation of monotonic objects makes static code very efficient, but at a severe cost to dynamic code. The monotonic semantics has been shown to be sound and correct with respect to blame [Siek et al. 2015b]. Reticulated Python reports slowdowns of 3.4x of monotonic objects over transient checks, so approximately 34x of monotonic objects over unchecked code.

SafeTypeScript's technique, though it carries the types in a separate "tag heap", has the same semantics: When objects pass through an unsound cast, it is shallowly checked and tagged in the heap, and all unsound (i.e., dynamic) updates have their values checked against the tag. The tag may be made more precise, but never less precise. The tag corresponds to a resolved structural object type, which in TypeScript is a list of typed fields, and a list of a call signatures, and we thus design our system of contracts to be sufficient for these types.

SafeTypeScript reports small slowdowns, on average 6.5%, in fully-typed code, but in mixed code, the cost of maintaining type soundness is on average 22x for one suite of benchmarks.

Our work implements the syntax of SafeTypeScript (and thus TypeScript) and the semantics required by SafeTypeScript's tag heap, but does not avoid checks in fully-static code, for reasons of polymorphism which will be discussed in later sections. We highly optimize dynamic soundness checks by taking advantage of existing checks in the infrastructure of the language implementation, substantially reducing the cost on dynamic code. Our implementation reports in Section 6 similar slowdowns to the original implementation of SafeTypeScript's tag heap for fully typed code, 6.61% in the average case, but our optimization brings the performance to a slowdown of at worst just 44.8% in intentionally pessimal mixes of typed and untyped code.

2.2 Just-in-Time Compilation

Modern virtual machines (VMs) for dynamic languages use Just-in-Time (JIT) compilation. This allows for *speculative optimizations*, which optimize code based on values seen at run-time: The program is allowed to run until a particular function or block must be compiled, and then it is compiled under the assumption that the values it interacts with will be similar to the ones previously seen. Checks are inserted to validate these assumptions, and code is recompiled if those checks are violated.

In particular, deriving from work in Self [Chambers et al. 1989], modern implementations of JavaScript and similar dynamic languages represent objects as a *shape* and a *store*. The store is a simple array containing the values of every field in the object, but not their names. The shape (in other contexts called "map" or "hidden class") is a table mapping field names to their position in the store. Every object with the same fields and same layout of those fields in the store refers to the same shape. This fact is used to avoid dynamically looking up fields in a technique which in JavaScript is called *inline caching*: The shape of a seen object is cached, and the surrounding function or block is compiled with knowledge of that seen shape, and thus the lookup consists only of accessing a known offset in an array.

To assure that every object with the same layout has the same shape, shapes are stored in a global tree. Empty objects all share the same empty shape, and to add a field to an object, one follows the transition in that tree from the object's current shape labeled with the field being added. For instance, in Figure 1, if field `item` is added to an empty object (i.e., an object with shape 0), the object's shape is updated to shape 1 and the value is stored at offset 0 of the store. If `next` is

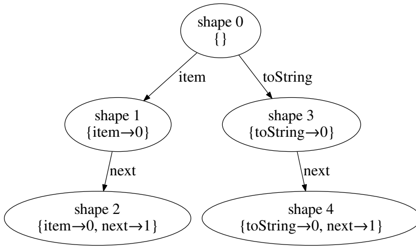

Fig. 1. Basic shape tree.

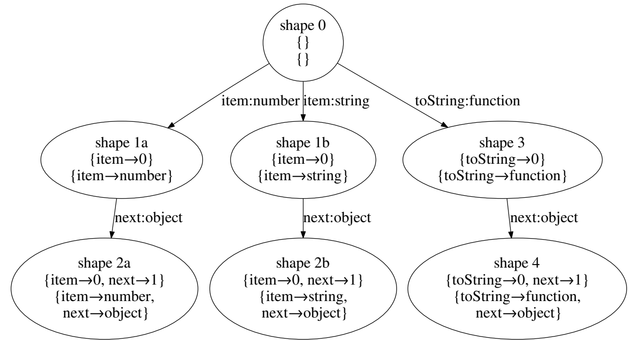

Fig. 2. Higgs shape tree.

then added to the object, its shape will be updated to shape 2. If no transition exists for a necessary addition to an object, a new shape is created and added to the shape tree to satisfy the request. Thus, in our previous example, `Node`s will have[1] shape 2.

This work builds on the Higgs virtual machine [Chevalier-Boisvert and Feeley 2016] [Chevalier-Boisvert and Feeley 2015], a research JavaScript virtual machine with competitive performance. Higgs implements many modern JavaScript optimizations, as well as a few unique ones which we utilize.

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

For instance, consider our `sum` function. A direct compilation of this function to basic blocks, with control flow replaced by `goto`s, is:

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

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

Upon the first compilation of this function, as block 1 requires the shape of `node`, the shape seen will be cached, resulting in a compiled procedure as in Figure 3. Assuming the function is called with a non-null `node.next`, it will proceed through block 2, and then be compiled again, into a procedure as in Figure 4.

Higgs, building on work in TruffleJS [Würthinger et al. 2013], extends shapes by encoding member types into them as well. Objects built of the same fields but distinct primitive types of

[1] This assumes that they are built in a particular order, as in JavaScript VMs the store is never rearranged.

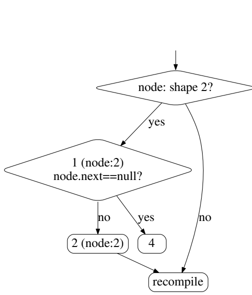

Fig. 3. First compilation of sum.

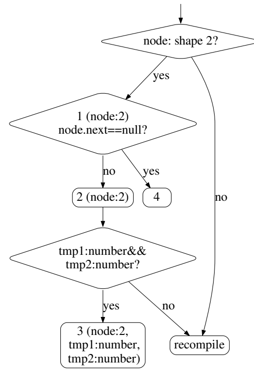

Fig. 4. Second compilation.

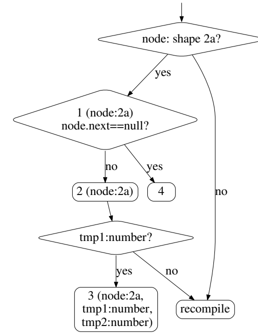

Fig. 5. With typed shapes.

their members will have distinct shapes, and objects with identical layouts and types have the same shape. This is accomplished by extending shapes with a map of names to primitive types. The transitions in this extended shape tree are labeled with both a name and a primitive type. For instance, in Figure 2, if field `item` is added to an empty object with type number, the object's shape is updated to shape 1a. If it's added with type `string`, the object's shape is updated to shape 1b. Because fields are typically mutable, this means the shape can also change when a field is modified: If an object with shape 2a's `item` field is updated to a string, the object's shape must be updated to 2b. That is, the shape is only a snapshot of the current state of the object, not a guarantee of future state.

The benefit of adding types to shape is twofold: First, as primitive types are available in the shape, it's unnecessary to tag or box values in the heap [Wöß et al. 2014]. Objects can thus be laid out more efficiently, avoiding wasting bits on type information that's identical between many objects. Second, speculating on a shape allows the compiler not only to optimize field access, but to eliminate type checks over the accessed field values: The check of the containing object's shape subsumes the check of the accessed value's type. For instance, `sum` can be compiled to the procedure in Figure 5 with typed shapes, avoiding a check for the type of `tmp2`, which is implied by the shape of `node`. This optimization is important to our goal: If a tag in SafeTypeScript's tag heap obliges the `item` field of an object to be a number, and that object's shape is shape 2a, that obligation cannot fail, and so does not need to be checked.

Lazy basic block versioning is in many ways similar to tracing [Gal et al. 2009] and even meta-tracing [Bolz et al. 2009], and all of the optimizations of contracts discussed herein could be applied equally to those contexts. That is, our performance improvement comes from careful use of speculation, not from the particular implementation of speculation in Higgs.

2.3 Efficient Contracts

The checks present in dynamically typed programming languages, and thus the guards and checks present in speculatively-compiled dynamic code, are fundamentally similar to the run-time contracts generated by gradually-typed languages. The principle difference is simply that the properties a JIT is checking are not requested by the user, but by speculative compilation, and thus violating such a guard is not an error, merely a cause for recompilation. We implement in HiggsCheck, an extension of Higgs, a system of *intrinsic object contracts*, which integrates with the shape tree to make type contracts as intrinsic a part of an object's run-time implementation as its memory layout. Shapes in HiggsCheck encode the layout, types, and contractual obligations of object members,

and the compiler is aware of these obligations. The contracts themselves, while not as general as true pre- and post-condition contracts, correspond to structural object types, and delay type checks of fields and function returns until the relevant field access or function call is performed, similarly to wrappers. Whenever a field of an object with contracts is read, a check is incurred, and if the contract stipulates that the value itself have a higher-order type, then a contract is applied to the value. Contracts may be added but never removed, and so values are monotonic, and may only become more restrictive—more precisely typed—during the execution of a program as they are used with contracts.

In the remainder of this paper we discuss the implementation of contracts and how they bridge the gap between the VM and gradually-typed languages. In Section 3 we discuss the design of intrinsic object contracts in isolation. Section 4 discusses how these contracts can be implemented efficiently, and Section 5 how they are used in the reimplementation of SafeTypeScript.

3 CONTRACTS

In this section we discuss the features and basic implementation of the contracts provided by HiggsCheck, independent of their optimizations and the direct use of these contracts by gradually-typed languages. We call this style of contracts *intrinsic object contracts*, as the contracts are intrinsic to the protected objects. In particular, we focus on how its design differs from other underlying systems of higher-order checks.

3.1 Design

The behavior of intrinsic object contracts in HiggsCheck is semantically simple: Values may be tested against contracts, and those contracts describe a number of obligations the values must adhere to. These obligations are intended to support gradual typing, and so are to type-related properties that are useful for languages and supported by the VM. That is, a contract is a set of typing obligations over a value, and a value may be tested against a contract to assure that it conforms to those obligations.

We divide the supported obligations into basic obligations, which can be checked quickly and in constant time for any value, and higher-order obligations, which cannot. Higher-order obligations correspond approximately to higher-order types, while basic obligations correspond to first-order, or primitive, types. We call a contract with only basic obligations a basic contract, and a contract with at least one higher-order obligation a higher-order contract. The empty contract, i.e. the contract with no obligations at all, is also a basic contract. When a value is tested against a contract, conceptually it is tested against all of that contract's obligations.

The basic obligations are to the primitive run-time types and `null`ness. For instance, the obligation "number" corresponds naturally to the JavaScript type number. These basic obligations are trivial to check with no contract system at all, as JavaScript's built-in `typeof` operator is sufficient.

Higher-order obligations are requirements over object fields and function return values. We focus first on object fields. These obligations require that the values yielded by accessing a field complies to a given contract, and thus incur testing of the value against the contract whenever it is accessed. A higher-order contract with several field obligations is equivalent to a structural object type, with each obligation referring to the type of a particular field. For instance, a contract can be built for the `Node` type as follows: Contract α's sole obligation is "number". Contract β, the contract for `Node`, has three obligations: The basic obligation "object", the higher-order obligation that field `item` be tested against contract α, and the obligation that field `next` be tested against contract β.

To support array indices, which in JavaScript are simply fields with numeric names, a contract may have an obligation over *all* numeric fields. So, to support arrays of `Node`s, we may create a

contract γ with two obligations: The basic obligation "object", and the higher-order obligation that all fields with numeric name be tested against contract β.

Because these obligations allow the recursive testing of contracts, they cannot be checked eagerly. Even in the case that they're not recursive, they cannot be checked quickly, and because objects in JavaScript are typically mutable, a check of such a higher-order contract would only be a momentary guarantee. As such, testing a value against such a contract checks only the basic obligations immediately, and delays testing of higher-order obligations until the relevant field is accessed. Testing an object `x` against contract β, our contract for `Node`, for instance, will immediately verify that `x` is an object, but will only test that `x.item` is a number when, and if, `x.item` is accessed.

Higher-order obligations may additionally restrict function returns. For instance, a contract δ may have the obligation that return values comply to contract α above. If a function is tested against δ, it cannot be checked eagerly, and so instead the system asserts that the function's return will be tested in all future calls. This is a simple obligation not sufficient for more sophisticated functional programming, and the reasons for this limitation and possible enhancements to support functional programming are discussed in Section 4.3.

When an object or function (in JavaScript, functions are objects) is tested against a higher-order contract, assuming all basic obligations succeed, the higher-order obligations are imposed on the object through delayed checks. To assure that these delayed checks happen, the contract is *applied* to the object. Objects in this system may have any number of contracts applied to them, and each object retains a list of applied contracts for its lifetime. This contract application is what assures that future accesses will be tested against the higher-order obligations: When a field is read from an object with applied contracts, if any of its contracts have obligations over that field, that obligation's contract is tested against the value seen at run-time. As higher-order obligations each test a value against a contract, this contract application may occur recursively. For instance, if an object `x` has the contract for `Node` applied, an access to `x.next` will be tested against the `Node` contract, applying it if applicable. If an object is tested against a contract which has already been applied, it isn't applied a second time.

Contracts are monotonic: While an object may always have more contracts applied, and thus gain more specific type information and a more restrictive API, it may never lose contracts. It is possible to add incompatible contracts, for instance `Node` along with a contract that demands that `item` be a string, in which case accessing such contradicting fields will inevitably cause a contract violation. They are independent of JavaScript's particular system of inheritance, and treat objects as black boxes. Thus, if it happens that the value of a field is fulfilled by accessing a prototype, or by using an accessor function, it is checked the same as a simple, local field.

Built-in functions, intended to be used by the implementation of a gradually-typed language, are provided to build contracts. Basic contracts for the primitive types (objects, functions, strings, numbers and booleans) and the empty contract are built in, and other contracts are built by extending them through further built-in functions which add field, index and function return obligations.

3.2 Blame

When a contract is applied to an object, an actual violation associated with that contract may occur later in the execution of the program, when a primitive value is obtained. With wrapper-based systems, the wrapper object carries with it blame information, such as the code location of the cast which caused a contract to be added, and with monotonic objects, the blame is stored in the object itself. Intrinsic contracts in HiggsCheck contain similar blame information: A contract test may provide an additional argument which represents the relevant blame label, typically a code location, and that blame will be reported if the contract's obligations are violated on the object.

When testing the obligations of a contract requires testing another value against another contract, the blame information from the parent contract is passed on to the child.

This gives intrinsic object contracts support for blame as in Findler and Felleisen [2002]. As contracts are only applied once, an object cannot accrue multiple blame labels for a single contract, and so-called "threesomes" [Siek and Wadler 2010] naturally collapse: We do not re-apply a contract if only its blame has changed. Given a fixed number of higher-order types in a program, there are thus a fixed number of contracts that an object may ever acquire. However, only the first blame label applied is kept.

The creation of useful blame information itself is left to the contract-utilizing code. It is assumed that contracts will usually be used by code generated from compilation of gradually-typed programming languages, and so it is the role of that compiler to provide useful information. The compiler described in Section 5 simply uses code lines as blame labels, but using e.g. stack traces is an alternative—albeit slower—option.

3.3 Discussion

The design of our contract system was intended to satisfy two requirements: That it be usable for the contracts usually needed by gradually-typed object-oriented programming languages, and that it be implementable in an optimal way. Relative to wrapper-based contracts, our contracts are less easily generalizable to other kinds of checks, but are implementable with a significantly reduced performance penalty, as demonstrated in Section 6.

Our contracts implement the semantics of monotonic objects, and so all references to an object have the same contractual obligations. In wrapper-based systems, references derived from the one to which the wrapper was applied will conform, and others will not. Both semantics have advantages, and both have been demonstrated to be correct, but monotonic semantics do mean that objects cannot be "chameleons": They cannot change their behavior in non-backwards-compatible ways once contracts have been applied.

Unlike wrapper-based implementations, there is no question of object identity, and no possibility of distinct identities for the same object, as no stand-in is ever created for an object. The application of contracts does not affect an object's identity or, equivalently, it affects the identity of all references to the object in the same way.

As intrinsic object contracts are a new language-level feature, backwards compatibility is a problem. Reticulated Python demonstrates [Vitousek et al. 2014] that monotonic objects are implementable through accessor functions, and these would be sufficient to implement intrinsic object contracts as well with no changes to the VM, but such implementations are slow. As contracts do not change the behavior of programs except to raise contract violation errors, the behavior of a correct program is not altered by removing its contracts. Thus, a simple shim library which implements the contract built-in functions with no behavior can be used to support standard implementations (with no checking), while still using contracts on supporting implementations.

4 IMPLEMENTATION

Using a similar technique to Reticulated Python's monotonic objects, it would be possible to create a correct implementation of the contracts presented in Section 3 with pure JavaScript[2]. The requisite checking of all values, however, would be untenably slow. HiggsCheck implements contracts at the VM level, and uses compile-time knowledge to optimize them. In this section we describe the

[2]If implemented at the VM level, contracts will be naturally resistant to malicious tampering, which may be impossible in an implementation in pure JavaScript.

implementation first of intrinsic object contracts, and second of the optimizations to avoid contract checks where possible.

4.1 Contract System

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

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

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

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

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

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

Recall that JavaScript VMs encode object shapes into a tree separate from the objects themselves. This means that two objects which have the same members, with the same layout, share a referentially identical shape. Higgs, further, encodes member types into these shapes. HiggsCheck extends them in one additional way: Contracts applied to an object become part of its shape as well. As well as maps of member names to offsets within the store and to types, shapes contain an array of contracts applied. Applying a contract to an object thus becomes a new kind of transition in the shape tree. Only objects for which the layout, member types and contracts applied are identical share the same shape, and as contract application is a transition, contracts must be applied in the same order to share a shape as well.

This relationship is crucial: When a contract is applied to an object, if the same contract has been assigned to another object with the same shape, nothing is allocated. The shape of the current object is updated, and the contract-object relationship is encoded into that shared shape. This is the first major optimization, as the number of allocations is equal to the number of contract-*shape* relationships, instead of the presumably much larger number of contract-object relationships.

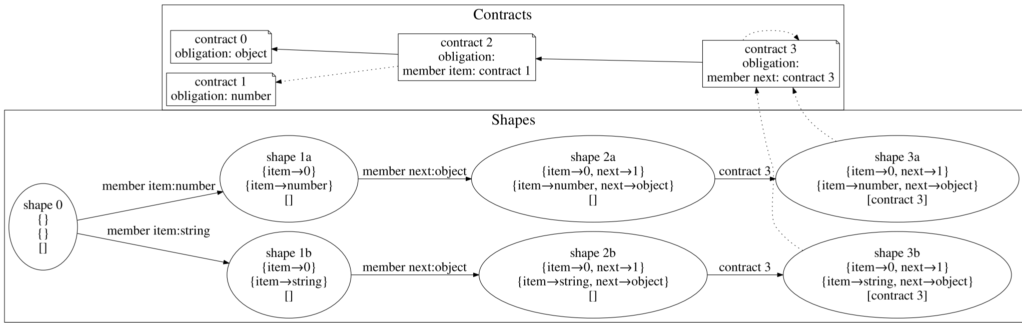

Fig. 6. Run-time state with shapes and contracts.

Adding contracts to the shape tree can increase polymorphism. Inline caches work by checking whether the shape of an object is identical to a shape seen in the past; if otherwise-identical objects which differ only in contracts have different shapes, then this inline cache must check more possible shapes, and thus take more time. As such, the mixing of typed and untyped code is expected to increase polymorphism. This is the primary cause of slowdown in the benchmarks in Section 6.

In HiggsCheck, the blame is also stored as part of the shape tree. That is, a transition which adds a contract is for a particular contract-blame tuple, and if a contract violation occurs, the blame information is found in the shape. This allows objects which share identical blame information to avoid duplicating it, and works well for simple benchmarks, but may not scale well to larger programs, as objects could acquire contracts through various code paths. An alternative is to store the blame information in the object itself, in which case each contract transition in the shape must also reserve a position in the store, where the blame information can be saved.

Figure 6 shows a complete shape tree and contract trees for a system in which objects with shapes 3a and 3b are contractually obliged to behave as `Node`s. Dotted lines in Figure 6 indicate contractual obligations, and solid lines indicate the parent/child relationships in the shape and contract trees. An object with shape 3a, for instance, has an `item` field of type number and a `next` field of type object. Further, objects with shape 3a are contractually obligated to have `item` fields of type number, by way of contract 3, and `next` fields which also conform to contract 3, by way of contract 2. It may seem unusual that shape 3b exists: Such objects have `item` fields of type string, but are obliged to have `item` fields of type number.

There are two reasons to allow such seeming contradictions: First, the design of our contracts is to check only on access to fields with obligations, and so a contract can be applied but never checked at all. Second, updates can bring the object into conformance, and types in TypeScript are often used before the object has actually been updated to conform. For instance, in the TypeScript compiler itself, types are stored abstractly and then eventually “resolved” with their fields, transforming an object from type `Type` to type `ResolvedType`:

```
function setTypeMembers(type: Type, members: SymbolTable, [...]): ResolvedType {
  (<ResolvedType>type).members = members;
  [...]
  return <ResolvedType>type;
}
```

If this is implemented with contracts, when the `ResolvedType` contract is initially applied to `type`, it does not yet have a `members` field. However, this is immediately fixed, and so raising a contract violation would be unhelpful.

When fields are read from or written to objects and the object shape has contracts, or when a function is called and its shape has contracts, their obligations are checked. If the obligations are higher-order, the shape of the read field or retrieved value may itself need to be changed, and it inherits the blame from the parent object.

Finally, values may be tested against contracts with `test_contract`. `test_contract` simply takes a value to be checked, a contract, and blame information, and performs the appropriate check: If the contract has any primitive obligations, they are checked immediately. If it has any higher-order obligations, it is applied to the checked object with the specified blame. After creating contracts, `test_contract` is the only function a gradually-typed programming language compiler needs to use to perform its run-time checks.

A contract is applied to an object as described in the following pseudocode:

```
function apply_contract(object, contract, blame):
  shape := object.shape
  if shape.hasContract(contract):
    # Nothing
  else if shape.hasChild(contract, blame):
    # That is, there is an existing child shape with this contract and blame
    object.shape := shape.child(contract, blame)
  else:
    newShape := shape.clone()
    newShape.contracts.add(contract, blame)
    shape.addChild(contract, blame, newShape)
    object.shape := newShape
```

When fields are changed that force shape changes, an equivalent shape with equivalent contracts must be found or created, just as when a type is changed in a Higgs shape tree, a shape with the correct types must be found.

When an object has contracts applied, those contracts are checked every time a member of that object is accessed or, if the object is a function, when it is called. This checking is transparent to the program, and objects with contracts are semantically identical to objects without contracts except that these accesses may fail.

4.2 Optimization of Checks

The design of contracts in HiggsCheck lends itself to speculative optimizations. Contracts and shapes, while serving different purposes, are partially redundant: If a shape has contracts applied, and the types specified in each basic obligation are supertype of those specified by the shape, those basic obligations cannot be violated. It is thus unnecessary to check such obligations: the run-time checks are subsumed by the inline cache.

Object field access in JavaScript is implemented using inline caching. In Higgs, this is implemented using an inlined JavaScript function, the core of which captures the object's shape, then performs the property access. In other systems this is done explicitly by the JIT compiler, inlined implicitly through tracing; while we will discuss these optimizations in the context of Higgs, the important consideration is speculation: what is computed at compile-time and what is computed at run-time. Higgs' property accessors, simplified for presentation, are:

```
1 function getProp(obj, prop) {
2   while (!capture_shape(obj)) {}
3   return obj_get_prop(obj, prop);
4 }
```

```
1 function setProp(obj, prop, val) {
2   while (!capture_shape(obj)) {}
3   obj_set_prop(obj, prop, val);
4 }
```

The loop on line 2 uses the built-in function `capture_shape` to implement an inline cache. In this section, code sample lines highlighted in grey are implemented partially or entirely at compile-time, and thus the code emitted and executed at run-time corresponds to part, or in some cases none, of the original code. `capture_shape`'s semantics are unusual: It implements inline caching by checking the shape of `obj` against a hidden cache. If the shape is already known at compile-time, then `capture_shape` evaluates to `true`, and so no code whatsoever is emitted for line 2. Otherwise, the shape of `obj` during the particular execution is captured into a cache, and the generated code is a check of the shape against that cached value. Since Higgs' fundamental design is to recompile basic blocks under different sets of assumptions, this means that further basic blocks will be compiled with a known shape for `obj`, and the "else" branch of that check will be a recompilation to capture a new shape. Thus, line 3, the internal operation which actually gets/sets the field, is always compiled with full knowledge of the shape of the object. `obj_set_prop` may change the shape, but as the shape is known at compile-time, the full change is known, and so the shape change itself is a simple assignment of a static value. Because this function is always inlined, further blocks in the containing function are also compiled with knowledge of the object's shape.

To implement contracts, this field access is extended to perform checks. We extend the built-in property retrieval function with two new built-in functions:

```
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     contract_check(obj, prop, val);
6   return val;
7 }
```

```
1 function setProp(obj, prop, val) {
2   while (!capture_shape(obj)) {}
3   obj_set_prop(obj, prop, val);
4   if (contract_can_fail(obj, prop,
                                val))
5     contract_check(obj, prop, val);
6 }
```

Conceptually, `contract_can_fail` and `contract_check` are both checking contractual obligations for the property `prop` in the object `obj` with the value `val`. The difference is when the checks are performed: `contract_can_fail` uses only information available at compile-time, and is evaluated at compile-time. In contrast, `contract_check` is just a thin wrapper around `test_contract`, determining which contract(s) need to be tested against `val` due to obligations on `obj`, and calling `test_contract` for each.

`contract_can_fail` evaluates to either `true` or `false`, and emits for execution at run-time at most a change to the shape of `val` to make its assertions true. If `contract_can_fail` evaluates to `false`, the entire `if` branch is trivially removed; thus, when contracts cannot fail, no call to `contract_check` is emitted, and the generated code is identical to that of the original `getProp`. `contract_can_fail` uses the following logic:

```
function contract_can_fail(obj, prop, val):
  # Note that obj and val are static references, not instances. Their type and
  # shape might be known due to previous capturing.
  if !shapeIsKnown(obj):
    return true
  shape := obj.knownShape()
  if !shape.hasContracts():
    return false
  obligations := shape.obligationsForProperty(prop)
  if obligations is empty:
    return false
  for each obligation in obligations:
    if obligation is basic:
```

```
      if !val.typeIsKnown() or val.knownType() != obligation.type:
        return true
    else:
      if !val.shapeIsKnown():
        return true
      valShape := val.knownShape()
      if !valShape.hasContract(obligation.contract):
        # We cause a contract application at run-time
        emit: apply_contract(val, obligation.contract, obligation.blame)
  return false
```

That is, if there are no relevant obligations or all obligations are known to be followed, `contract_can_fail` evaluates to `false` at compile-time. If insufficient information is known, it evaluates to `true`. If sufficient information is known but `val` is known not to have a necessary contract, the contract application is emitted, and thus run at run-time, and `false` is returned at compile-time.

Since this function is compiled speculatively, the shape of `obj` is known, which includes both the type of fields of `obj` and contracts over `obj`. This means that primitive contracts over fields will be evaluated at compile-time at no run-time cost. However, if the field assigned or retrieved is an object, and `obj` has higher-order obligations over that field, its shape is needed to check whether `obj`'s obligations have been met. Thus, as a further optimization, we capture the shape of `val` and call `contract_can_fail` a second time using that captured shape:

```
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    return val;
10 }
```

```
1  function setProp(obj, prop, val) {
2    while (!capture_shape(obj)) {}
3    obj_set_prop(obj, prop, val);
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    return val;
10 }
```

Like any other captured shape, the shape of `val` may be used after this accessor, and so if `val` was to be accessed as an object anyway, we've merely moved back a check that would have occurred later. In the worst case, `val`'s shape check is not redundant with any other checks, and so contract checking incurs one additional check, and possibly one shape change, if `contract_can_fail` requires it. With the shape of `val` known, and `contract_can_fail` free to update the shape with a contract if necessary, the only case in which `contract_check` can be reached is if contracts definitely will fail, or if the shape of `obj` or `val` cannot be captured.

Speculative compilation assures that most of the cost of `getProp` is at compile-time, not run-time. The basic blocks of `getProp`, with control code replaced by gotos, are:

(1) `if (!capture_shape(obj))` **goto** `1`
(2) `val = obj_get_prop(obj, prop); if (!contract_can_fail(obj, prop, val))` **goto** `6`
(3) `if (!capture_shape(val))` **goto** `3`
(4) `if (!contract_can_fail(obj, prop, val))` **goto** `6`
(5) `contract_check(obj, prop, val);`

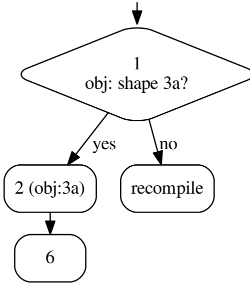

Fig. 7. Compilation of getProp for shape 3a, property item.

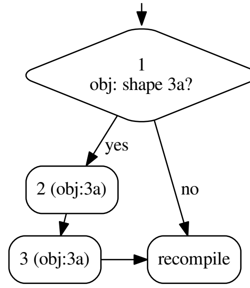

Fig. 8. First compilation of getProp for shape 3a, property next.

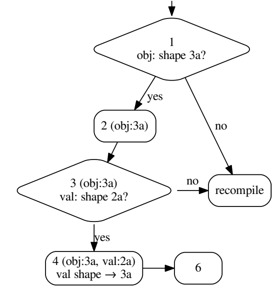

Fig. 9. Final compilation of getProp for shape 3a, property next with shape 2a.

(6) `return val;`

For instance, consider the compilation of `getProp` when called for the `item` member of an object with shape 3a from Figure 6. Upon the first compilation, the shape of `obj` will be captured, so block 1 is compiled into a simple check of that cached shape. Access to that object's `item` field cannot fail if that object has shape 3a: While shape 3a does have a contractual obligation over `item`, the obligation is that `item` be a number, and shape 3a defines it as a number. Thus, the `contract_can_fail` in block 2 can be evaluated to `false` with no further recompilation, resulting in a procedure as in Figure 7.

When accessing `next`, because the shape of `obj.next` will be unknown, the first `contract_can_fail` is resolved to `false` at compile-time. Thus, upon the first compilation, the compiler will reach block 3, as in Figure 8. `val`'s shape will be captured upon the first compilation, at which point blocks 4 and 6 can be compiled. If the captured shape of `val` does not have the needed contract applied, e.g. shape 2a, then the behavior of block 4 will be to transform `val`'s shape to one which does, e.g. shape 3a. The resulting procedure is shown in Figure 9.

`test_contract`, the language's front-end to checking and applying contracts, is also implemented as an inline function, and also uses an inline cache:

```
1  function test_contract(val, contract, blame) {
2    while (!capture_shape(val)) {}
3    while (!capture_contract(contract)) {}
4    if (!test_contract_primitives(val, contract))
5      throw new ContractError(blame);
6    if (contract_is_higher_order(contract))
7      obj_apply_contract(val, contract, blame);
8  }
```

`capture_contract` is equivalent to `capture_shape`, but captures the exact contract bound to `contract`. If contracts are always stored in `const` variables, no code is emitted to capture them. Lines 4–5 test the primitive obligations of the contract, throwing an exception if they are violated. If sufficient information about `val` is known to test the primitive obligations at compile-time, they are, and no code is evaluated for the `if`. If the contract is higher-order, `obj_apply_contract` performs the actual shape change, using the captured shape such that both the from- and to-shape are known. Since the contract is known at compile time from line 3, `contract_is_higher_order` is always evaluated at compile-time.

Now we may consider a function which uses contracts. Consider a function which adds the `item` field of a node to a constant value, using contracts to assure its argument type:

```
function add5(ln) {
  test_contract(ln, Node, "blame");
  return ln.item + 5;
}
```

Bearing in mind that the contract test and property access will automatically inline the functions described above, the compilation of this function to basic blocks, with control code replaced by gotos, is:

(1) `if (!capture_shape(ln))` **goto** `1`
(2) `if (!capture_contract(Node))` **goto** `2`
(3) `if (test_contract_primitives(ln, Node))` **goto** `5`
(4) `throw new ContractError("blame");`
(5) `if (!contract_is_higher_order(Node))` **goto** `7`
(6) `obj_add_contract(ln, Node, "blame");`
(7) `if (!capture_shape(ln))` **goto** `7`
(8) `val = obj_get_prop(ln, "item"); if (!contract_can_fail(ln, "item", val))` **goto** `12`
(9) `if (!capture_shape(val))` **goto** `9`
(10) `if (!contract_can_fail(ln, "item", val))` **goto** `12`
(11) `contract_check(ln, "item", val);`
(12) `return val + 5;`

Upon the first execution of this function, block 1 will invoke a recompilation to capture the shape of `ln`. Assuming that `Node` is a constant value, it will not need explicit capturing, so block 2 will emit no code. Blocks 3 and 5 are checks of information known at compile-time, and so will also emit no code. Block 6 performs a known shape change, and block 7 depends only on known information, emitting no code. Then, blocks 8–11 are generated similarly to Figure 7, and block 12 performs integer arithmetic, resulting in the procedure in Figure 10.

If this function is called again with an object of a different shape, recompilation will be forced. For instance, if it is called with `ln` having shape 2b, the block versions will be generated as in Figure 11. Blocks 2 and 3 are the same, but block 4 now retrieves a string, and as `contract_can_fail` is true, proceeds to block 5. At run-time, of course, block 11 will throw an exception, as the contract has been violated.

Contracts over function returns and array indices are checked similarly through similar inlined functions. Like with field contracts, these checks are two-stage, possibly checking the type and shape of the retrieved value to allow `contract_can_fail` to be evaluated at compile-time. If the

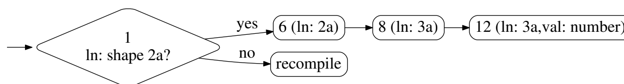

Fig. 10. add5 basic blocks after first compilation.

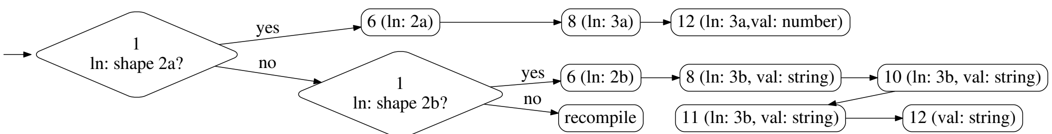

Fig. 11. add5 basic blocks after second compilation.

value is used later in the function, later checks are elided, and so the cost of run-time checking is similarly made redundant with existing checks.

4.3 Summary and Discussion

The core optimization allowed by intrinsic object contracts is simply the relationship between contracts and objects: Many objects may refer to the same contract, via their shape, and the objects needn't be deeply modified or wrapped to do so. This is fundamental to the design, causing intrinsic object contracts to incur far fewer allocations than wrapper-based solutions and less severe cost of applying contracts than other monotonic solutions.

With existing solutions, performance depends on functions being inlined or checks matching those performed by the JIT. Implementing contracts at the VM level allows HiggsCheck to use them in a reliable, predictable way.

The remaining optimizations are the result of the relationship between inline caching of object shapes and integrating shapes with contracts. Since inline caching allows speculative compilation of code with object shapes intact, integrating contracts into object shapes allows the speculative compilation of code with known contracts. In the context of typed object shapes, this makes contract checks mostly redundant, but even without typed object shapes, it would allow the VM to perform the relevant checks as part of its own speculative optimizations.

While HiggsCheck does support contracts over function returns, we have found them insufficient in practice. The problem arises from polymorphic functions, e.g. `function id(x) { return x; }`. Although the type of this function is easily defined, `T → T`, no intrinsic object contract as currently defined can encompass that relationship. The problem of polymorphic contracts has been explored [Guha et al. 2007], and is, at the most basic level, solved by parameterizing contracts in such a way that the type parameter can be removed from generated results. This is difficult to rectify with the monotonic semantics, but in order for interesting, higher-order functions to be supported, we will need to support such polymorphic contracts. A possible solution to this problem is modifying the semantics of function calls to specify explicitly what types the arguments have in their local context, so that contract parameters can be applied in a strictly-local setting, but it is unclear how this information should be communicated, and so it has not yet been implemented. Reticulated Python encountered a similar problem with monotonic functions in Python [Vitousek et al. 2014], and solved it by using wrappers for functions instead of monotonicity. In practice we bypass contracts on function returns, instead falling back on explicit checks.

5 SAFETYPESCRIPT

We have reimplemented SafeTypeScript [Rastogi et al. 2015], using a later version of TypeScript (2.2.1) and intrinsic object contracts, to be both an impetus for their implementation and a demonstration of their ability. SafeTypeScript's checks arise from a "tag heap", which stores an association between values in the runtime heap and types those values have been expected to adhere to.

For instance, if an object from dynamic code reaches a variable with type `Node`, the tag heap will store that association, and if the member `item` is read from that object, it is verified to be a number. If the member `next` is read, it will be tagged as a `Node` in the tag heap. This tag heap has a direct analog in our intrinsic object contracts: Every object in our system has a (possibly empty) list of contracts, each of which corresponds to a type the object has been associated with. Thus, we don't consider this language to be a new, SafeTypeScript-based language, but simply a reimplementation of SafeTypeScript with a different method for implementing the tag heap. We describe in this section how SafeTypeScript's semantics map to our implementation.

Values are either primitive or objects. Types of primitives can be checked directly using the JavaScript `typeof` operator, and checks are performed as such both in our implementation and in

the original implementation of SafeTypeScript. However, objects are more complex: they can be updated at runtime, and thus functionality must exist to assure that they remain consistent with their tags.

SafeTypeScript's semantics describe three main auxiliary functions involved with tagging and checking:

- `comb`: This function combines two tags into one intersection type, or returns an error if this is not possible. This is used in object tag updates, as each object can only have one tag, and it must be able to adapt to reflect changes to the object during runtime. As our objects may have any number of contracts, `comb` is implicit in applying multiple contracts to an object.
- `shallowTag`: This function updates the tag of the specified location with the specified type. It combines the old tag with the new tag through `comb`, and returns this new tag. `shallowTag` is implemented as an implicit part of `test_contract` and by the contract checking performed in `getProp` and `setProp`.
- `checkAndTag`: Checks the primitive type of a value, and tags it with higher-order types. It is of course implemented through `test_contract`. `checkAndTag` checks the primitive type of fields eagerly, while `test_contract` does not, but this recursivity is not necessary for SafeTypeScript's soundness properties, as the tags are checked while reading and writing fields regardless.

SafeTypeScript's typing judgment adds tags in the following scenarios at compile-time: Variable/-parameter definition (including `this`), variable assignment, field assignment, class constructors and class method calls. Its implementation avoids doing so explicitly in circumstances where no unsound cast could have occurred, and fully-static code thus incurs little run-time overhead. Intrinsic object contracts have a different set of concerns: Run-time checks are expected to be inexpensive or even free, but if some objects have contracts and some objects do not, inline caches will become unnecessarily polymorphic. We thus add contracts at run-time for any non-strictly-local value in all of these situations, plus all accesses to global variables not defined in the JavaScript language specification, regardless of whether an unsound cast has occurred.

Run-time checks are performed during reads and writes to object fields and method calls, and these are naturally covered by HiggsCheck's contract checks. The following rules, A-ReadLit and A-WriteLit, are included from the run-time typing specifications of SafeTypeScript for reading from a location in the runtime heap, respectively:

$$\frac{f = \texttt{toString}(\,cv) \quad \tau'' = \texttt{comb}(\,\texttt{tag}_{C.T}(l),\ t) \quad f : \tau' \in \texttt{fields}_S(\,\tau'') \vee (\,f \notin \texttt{fields}_S(\,\tau'') \wedge \tau'' = \texttt{any}) \quad \tau' <:_{C.S} \texttt{any} \rightsquigarrow \delta}{C; \texttt{read}(\,l, t, cv) \rightarrow C; \texttt{shallowTag}(\,l[\,cv]\,\delta)}$$

$$\frac{f = \texttt{toString}(\,cv) \quad \tau'' = \texttt{comb}(\,\texttt{tag}_{C.T}(l),\ t) \quad f : \tau' \in \texttt{fields}_S(\,\tau'') \vee (\,f \notin \texttt{fields}_S(\,\tau'') \wedge \tau'' = \texttt{any})}{C; \texttt{write}(\,l, t, cv, v_3, t_3) \rightarrow C; l[\,v_2] := \texttt{checkAndTag}(\,v_3, t_3, \tau')}$$

These (in addition to A-CallMLit, for method calls but not shown here) are the central run-time rules in SafeTypeScript which tag locations in the heap. Notice the second precondition in each case, $\tau'' = \texttt{comb}(\ \texttt{tag}_{C.T}(l),\ t)$, serves to combine the previous tag for the location, and the current tag to tag with. The checks in HiggsCheck are implemented by `getProp` and `setProp`, respectively, and `comb` arises from objects having a list of contracts rather than a single tag: The combined tag is simply the intersection of the list of contracts.

Complex types are easily handled: Contracts are always generated for fully resolved types, so union types, intersection types and generics are all supported implicitly. For instance, a contract will be generated for `List<number>`, and used whenever that resolved generic type is specified. When only `List<T>` is specified, it is resolved with the lowest constraint on `T`, typically `any`, as `List<any>`.

Blame is not considered in the original SafeTypeScript implementation, however here it is tracked simply by the input line which generated the contract application.

Consider our Node example. A contract is generated for Node, resulting in the following JavaScript code when compiled:

```
const c1 = contract_for("number");
function c2f() {
  var co = contract_for("object");
  var c = co;
  c = contract_oblige_member(c, "next", co)
  c = contract_oblige_member(c, "item", c1)
  contract_reoblige_member(c, "next", c);
  return c;
}
```

```
const c2 = c2f();
function sum(l) {
  test_contract(l, c2, "sum.ts:6");
  // (otherwise unmodified)
}
```

The contract is generated then applied to a constant variable to assure that it is always known in `test_contract`, and the remainder of the `sum` function requires no change, as the obligations of `l` will naturally be checked whenever it's used.

Note that TypeScript, even in fully-typed programs, is not sound due to covariant method subtyping rules [Bierman et al. 2014]. SafeTypeScript and our reimplementation thus introduce some checks where the language, in essence, introduces implicit downcasts. If one adopts the sound core of TypeScript's type system and writes programs that are sound in that context, neither the original nor our reimplementation will ever raise type errors at run-time.

6 PERFORMANCE

We measure the performance of HiggsCheck's intrinsic object contracts by comparing the execution time of a suite of benchmarks compiled with and without contracts, as well as by examining how and how many checks are eliminated from the compiled code. Each benchmark has no untyped code. Note that our compiler does not optimize out "impossible" contracts (i.e., contracts which cannot fail due to surrounding typed code), so compiling fully-typed benchmarks with contracts *maximizes* the number of contract checks performed.

6.1 Benchmark Programs

We adapted a number of benchmarks from Rastogi et al. [2015] and Takikawa et al. [2016], as well as using the TypeScript compiler itself as a benchmark. For each benchmark, we briefly describe their behavior, as well as their size in terms of lines of code, number of type annotations, and number of modules.

From [Rastogi et al. 2015], we used 5 benchmark programs (which were in turn adapted from the Octane benchmark suite). These benchmarks are unmodified from their original SafeTypeScript implementation. They are outlined below.

(1) **crypto** is a benchmark which uses an RSA encryption/decryption algorithm to encrypt a string, decrypt it, then verify that the original and decrypted string match. It measures the performance of integer arithmetic operation and array access. It uses mostly array and primitive contracts. Lines of code: 1657. Annotations: 635. Modules: 1.
(2) **navier-stokes** is a 2D Navier-Stokes fluid flow equation solver. It measures the effectiveness of numeric array access and floating point arithmetic. It uses mostly array and primitive contracts. Lines of code: 438. Annotations: 192. Modules: 1.
(3) **raytrace** is a JavaScript based ray tracer. The benchmark measures floating-point computations typical of a ray tracing algorithm. It has a number of types representing scenes,

cameras, objects, etc, which form the basis of our contracts, as well as arrays. Lines of code: 755. Annotations: 232. Modules: 1.

(4) **richards** is a standard operating system kernel simulation benchmark, used to measure the speed of accessing object properties, calling functions, and dealing with polymorphism. Its types encapsulate operating system structures such as processes, which are our contracts. Lines of code: 564. Annotations: 122. Modules: 1.

(5) **splay** is based on a V8 profiler's log processing module. It measures how fast the JavaScript engine is at allocating and reclaiming nodes, and also how the engine deals with frequent changes to a large tree object graph. Its types are simple, but recursive, tree nodes. Lines of code: 400. Annotations: 46. Modules: 1.

From Takikawa et al. [2016] we chose 6 benchmark programs and translated them from their Typed Racket implementation into TypeScript. We focused on reasonably portable benchmarks which showed significant performance problems in some configurations in Typed Racket. The benchmarks previously relied on a number of libraries, which we either ported to TypeScript or worked around. We ensured that everything that was timed in their implementation remained timed in our translation.

(1) **sieve** is a straightforward program that calculates primes using the sieve of Eratosthenes. The Typed Racket implementation used an infinite list, which we implemented as an pseudo-infinite list class which repeatedly calls a computeNextPrime function to populate its list of primes. That class and numbers are the principle types in the program. Lines of code: 66. Annotations: 22. Modules: 1.

(2) **morse** is a benchmark which generates morse code strings and computes the Levenshtein distance between generated words. Here, not much needed to be changed, and the adapted implementation is quite similar to the original. Primarily strings are used here, as well as arrays of strings. Lines of code: 134. Annotations: 35. Modules: 4.

(3) **suffixtree** calculates longest common substrings between two words using Ukkonen's suffix tree algorithm. The Typed Racket implementation was highly functional, and we tried to remain true to the original. Certain recursive loop structures were changed to more imperative loop structures due to JavaScript's lack of tail call optimization. While the purpose of this program is to operate over strings, it encapsulates these into a number of datatypes which label sections of the string, and forms a tree (hence the name) of labels for its computation. These classes form the bulk of the contracts, as well as arrays thereof. Lines of code: 1311. Annotations: 296. Modules: 5.

(4) **snake** implements the snake game. The program executes a pre-recorded set of moves. Contrary to its Typed Racket implementation, no external modules were required, and the benchmark is entirely self contained. Types included the game world and the snake itself, as well as game positions. Lines of code: 372. Annotations: 73. Modules: 7.

(5) **tetris**, much like snake, is an implementation of the famous Tetris game, executing a pre-recorded set of moves. As in snake, the benchmark does not rely on external libraries. Types include the game world, game pieces, as well as information for the (removed) graphical output such as pixel positions and colors. Lines of code: 585. Annotations: 185. Modules: 9.

(6) **gregor** is an unusual case. In Takikawa et al. [2016], the benchmark was described as a stress-test for a date and time library. Unfortunately, due to differences between Typed Racket and JavaScript's native numeric precision, we were required to write and include an additional BigNumber library, to perform the date calculations correctly to arbitrary precision. This changed the bulk of the benchmark's run-time from the date and time library itself to arbitrary-precision numeric arithmetic. The BigNumber and Date types form most of the

Table 1. Benchmark performance with intrinsic object contracts.

	Unchecked		Intrinsic contracts			Pessimal configuration		
Benchmark	**Mean**	**Std. Dev.**	**Mean**	**Std. Dev.**	**Slowdown**	**Mean**	**Std. Dev.**	**Slowdown**
crypto	307	0.688	334	5.57	8.87%	444	0.632	44.8%
navier-stokes	595	0.505	590	0.674	-0.84%	627	0.751	5.28%
raytrace	152	1.47	179	1.86	17.32%	(Fully typed)		
richards	232	1.25	241	1.74	4.08%	241	2.46	4.12%
splay	1469	5.31	1552	2.58	5.63%	1553	10.5	5.72%
gregor	10611	28.7	10616	25.9	0.04%	10681	13.6	0.67%
morse	370	1.66	372	5.12	0.44%	(Fully typed)		
sieve	2391	0.831	2475	2.34	3.49%	(Fully typed)		
snake	8133	24.8	8118	21.7	-0.18%	(Fully typed)		
suffixtree	1850	3.61	1866	6.96	0.89%	(Fully typed)		
tetris	2151	8.90	2329	9.87	8.28%	(Fully typed)		
tsc	51758	110	67925	280	31.2%	(Fully typed)		
Average					6.61%			10.18%

types in this benchmark, aside of course from numbers. Lines of code: 1283. Annotations: 433. Modules: 9.

Finally, the TypeScript compiler itself, tsc, is used as a benchmark. A simple benchmark harness was written which replaces TypeScript's system access library with a shim which provides an imaginary filesystem containing the SafeTypeScript benchmark programs. With this shim system library, the benchmarks are compiled internally. Types used by tsc include abstract syntax tree nodes, types encapsulating the types used in compiled programs, and many strings. tsc is particularly notable because many objects are constructed in various different ways and pass through various different codepaths, creating significant polymorphism. This is, naturally, the largest benchmark. Lines of code: 80,737. Annotations: 19,581. Modules: 35.

6.2 Results

The benchmarks were run on an AMD Opteron(tm) 6380 at 2.5GHz with 500GB RAM, running Ubuntu 16.04.2 LTS. Each benchmark was run 10 times, and each run included a number of executions of the underlying benchmark function depending on the particular benchmark. Each benchmark has a warm-up phase; the results are measured after JIT optimization has been performed. Performance is measured in milliseconds execution time, and thus higher is worse. The results are shown in Table 1: The column "unchecked" shows the performance of code with no contracts, while the column "intrinsic contracts" shows the performance of fully-typed code with contracts. While contract checking does, naturally, incur some slowdown, in all but two of the samples the slowdown was under 10%. In raytrace, it was 17.32%, and in tsc, the worst performer, 31.2%. In some benchmarks, the change in performance is within statistical noise; in two of these cases, our results happen to show a small speedup, but not of significance. The average slowdown across all benchmarks was 6.61%.

In Table 2 we show statistics on the number of checks performed at compile-time, as well as the number of basic block instances compiled and the peak memory usage. Because Higgs handles polymorphism by compiling different versions of basic blocks, the number of blocks and peak memory usage together represent this increase in polymorphism; note however that since `contract_can_fail` forms a block boundary, adding contracts will increase the number of

Table 2. Benchmark behavioral statistics.

Benchmark	Configuration	# checks	# captures	# blocks	mem (KB)
crypto	unchecked			16037	579880
crypto	full	671	95	18418	595308
crypto	pessimal	689	96	18538	597624
navier-stokes	unchecked			9667	490192
navier-stokes	full	484	0	10454	520520
navier-stokes	pessimal	467	0	10334	520024
raytrace	unchecked			12826	543136
raytrace	full	1364	282	15763	578548
richards	unchecked			10143	478296
richards	full	515	95	11945	495128
richards	pessimal	505	92	11963	495136
splay	unchecked			8681	759208
splay	full	1208	222	9280	768336
splay	pessimal	1208	222	9280	768784
gregor	unchecked			19287	897476
gregor	full	919	188	25292	964840
gregor	pessimal	908	189	25305	967456
morse	unchecked			14795	591412
morse	full	10	0	14946	590928
sieve	unchecked			11082	521064
sieve	full	34	9	11299	518508
snake	unchecked			14356	1028684
snake	full	192	34	17746	1071028
suffixtree	unchecked			16257	673464
suffixtree	full	458	94	18680	705092
tetris	unchecked			14263	878932
tetris	full	239	58	19054	930472
tsc	unchecked			181183	4844672
tsc	full	77471	13585	281077	8381876

compiled blocks even with no increase in polymorphism. All run-time checks in all benchmarks are subsumed by compile-time checks; that is, `contract_check` was never called at run-time in any benchmark. As such, the number of run-time checks (0) is excluded. The "# checks" column shows the number of contract checks which were compiled; this doesn't directly correspond to the number of annotations, as versioning can cause a check to be compiled multiple times, but it is related. This shows how many static checks were performed to avoid run-time checks. The "# captures" column shows the number times the outer check of `contract_can_fail` had sufficient static information to evaluate to `false`, and thus required caching and checking shapes. While `contract_check` is avoided in these cases, the behavior of the code is nonetheless changed. This cannot easily be distilled into a number of new operations which occurred at run-time, as many of these checks would turn out to be redundant anyway. In the common case, the slowdown comes principally from the run-time cost of adding contracts, and to a lesser degree from increased polymorphism.

In benchmarks with many subtypes, polymorphism increases substantially. tsc is an extreme case: the added block versions increase the memory usage almost twofold. It is likely that a compromise will need to be found for these polymorphic cases by which the improvements in performance can be balanced with a less severe increase in memory use. In the case of raytrace, several of the

contracts in that benchmark correspond to subtypes of each other, and neither our compiler nor HiggsCheck recognize this relationship, so nearly-identical shapes are created with related but non-identical contracts.

As was observed in Takikawa et al. [2016], the worst-case slowdown of gradually-typed languages comes from the interaction of typed and untyped code. Thus, to understand the likely cost of our contract system, we must mix typed code with untyped code. In the case of Typed Racket, which was used as an example in that work, and SafeTypeScript, checks and contracts are avoided entirely in fully-typed code. In our system, contracts are generated regardless. However, if some objects are generated with contracts and some aren't, this will make inline caches polymorphic, which affects performance. In some cases, the further checks induced by this polymorphism could be slower than the checks induced by contracts, so a minor slowdown is expected.

In Takikawa et al. [2016], it was shown that by adding or removing types from modules, a program can be viewed as a lattice of possible partially-typed programs. Each position in their lattice represents a configuration in which some modules were typed, and others untyped. At the top of the lattice is the fully-typed version, equivalent to the column "intrinsic contracts" in Table 1, and at the bottom is the fully-untyped version, equivalent to "unchecked". In Typed Racket, types may be added or removed at the granularity of modules, so this lattice was of a fairly small size. In TypeScript (and thus, SafeTypeScript), types may be added or removed at the granularity of single annotations. The richards benchmark, for instance, has 122 type annotations (including implicit type annotations, treated the same as explicit ones), and thus 2^{122} possible configurations, too many to be tested exhaustively.

To explore this state space, we used the simulated annealing algorithm to find pessimal configurations. For each benchmark, ten random configurations were chosen, and then for each of those configurations, the time was measured in a single run, as well as the time of ten random changes to the configuration, i.e. ten random steps of fixed length through the lattice. This process was then repeated 100 times with a decreasing number of annotations changed. The size of this change, i.e., number of annotations changed, started at half the total number of annotations in the benchmark, and decreased by 10% in each step. The worst case seen across all 10,000 such executions is reported in Table 1 as "pessimal configuration". This is probably not the true worst case, but represents a likely bad case of performance. In many benchmarks, no case was found that had worse performance than fully typed, and in several, the difference was within statistical noise. This process took roughly two weeks for these benchmarks, and necessitated the otherwise excessive 500GB of RAM on the test machine.

The crypto benchmark showed the worst slowdown in the pessimal configuration, 44.8%. Upon investigation, this slowdown occurs due to a critical array being initialized and filled before a contract is applied, then the contract applied afterwards. As all values in the array then had to have contracts checked and applied, several critical code paths that were monomorphic and involved no shape changes became polymorphic. The average slowdown across all pessimal cases was 10.18%, which we contend is quite reasonable for most uses of gradual types.

6.3 Threats to Validity

With the exception of tsc, the benchmarks used to measure HiggsCheck's performance are small and fairly simple. Our performance is also contingent on check redundancy: If a program moves many values between objects without actually using them, the slowdown could be worse, as types would be checked that a contract-free program would ignore. It is certainly possible to engineer programs with vastly worse performance.

We examine performance in terms of time in much more detail than space. While the memory consumption is in most cases not severe, extreme cases are demonstrated in tsc.

Our optimizations are principally an extension of inline caching. Situations in which inline caching becomes megamorphic, at which point VMs including Higgs abandon caching, will display substantial slowdowns. It is possible for a nearly megamorphic cache to become megamorphic by adding contracts, so it is possible to construct programs which show far more severe slowdown than our examples.

Our technique for finding a pessimal configuration is purely mechanical, and likely doesn't find the true worse case. However, our system also doesn't display the same performance characteristics as existing implementations of gradual typing: In other systems, pessimal configurations come from values frequently going between static and dynamic code, as this incurs the most checks and wrapping. Our system applies contracts universally, so going back and forth like this cannot incur more checks, only more polymorphism. Worst cases arise from same or related types being added in many different contexts, thus creating many distinct blame labels.

7 LIMITATIONS AND FUTURE WORK

The intrinsic object contracts presented herein cover many cases of object-oriented code, but are a compromise between generality and optimality. The most significant limitations are on functions, where these contracts test only the return value and not the argument-return relationship. For many cases this is sufficient; but for functional code it is not.

The relationship between contracts and objects—or, more generally, protected values—also depends on the implementation of objects in JavaScript and other dynamic languages: In shapes, there is a shared reference to which contracts can be attached. Without that, as would be the case particularly in functional languages with a simpler memory model or without mutation, it is unclear whether the value-contains-contract model is implementable.

Since these contracts are limited to type-related properties representable in the VM by design, they are less general than true pre- and post-condition contracts. However, they are sufficient for gradual typing, which is their design intent. We believe that they can be extended to more powerful, general contracts using intelligent inlining.

We apply intrinsic object contracts universally, and thus pay the cost of checks in almost all code paths. Many other implementations avoid this, and we could as well, for performance improvements. There is a caveat, however: Allowing some objects to have contracts and some not creates more shapes at run-time, and thus more polymorphism. Finding the right balance will require further study.

More robust support for generic types and polymorphic functions will require polymorphic contracts. We believe similar non-wrapping contracts are applicable here, but cooperation will likely be needed between typed callers and contractually-restricted functions. The design and implementation of such monotonic polymorphic contracts is future work.

While sufficient for simple situations, the blame tracking implemented herein is quite simple. It is likely that more rich blame information can be supported, but rectifying it with the necessity of reducing allocations and keeping shape trees reasonable is a challenge for future work.

Finally, we've explored the performance characteristics of this technique in terms of time, but demonstrated that the memory overhead should be examined as well. Much of this arises from basic block versioning, and so other speculative systems would have less overhead, but finding the right balance is future work.

8 CONCLUSIONS

"Is sound gradual typing dead?"

The performance cost of run-time contracts has been a hindrance to gradual typing. However, applying the techniques used by JIT compilation to the problem of contracts allows most of the

cost to be averted, as it is redundant or compatible with existing speculative optimizations. We've demonstrated that making the VM aware of contractual obligations allows for a rich type system to be checkable at run-time with reasonable cost, 7% in the typical case and no worse than 45% in the worst case, which is a vast reduction from the current state of the art, which can slow down execution 10s or 100s of times. Through HiggsCheck and a reimplementation of SafeTypeScript, we've shown that these contracts are practical for real, sound gradual typing. While further optimizations are likely still possible, we posit that the performance in HiggsCheck is acceptable for most cases of gradual typing. Soundness in gradual typing is, with some help from the VM, still very much alive.

ACKNOWLEDGEMENTS

The authors would like to acknowledge Rudi Chen, who brought HiggsCheck from a barely-working prototype to its current state. This work is partially funded by an NSERC grant.

REFERENCES

Gavin Bierman, Martín Abadi, and Mads Torgersen. 2014. Understanding typescript. In *European Conference on Object-Oriented Programming*. Springer, 257–281.

Bard Bloom, John Field, Nathaniel Nystrom, Johan Östlund, Gregor Richards, Rok Strniša, Jan Vitek, and Tobias Wrigstad. 2009. Thorn: Robust Concurrent Scripting on the JVM. In *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications (OOPSLA '09)*. ACM, New York, NY, USA, 789–790. https://doi.org/10.1145/1639950.1640016

Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski, and Armin Rigo. 2009. Tracing the meta-level: PyPy's tracing JIT compiler. In *Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*. ACM, 18–25.

C. Chambers, D. Ungar, and E. Lee. 1989. An Efficient Implementation of SELF a Dynamically-typed Object-oriented Language Based on Prototypes. In *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications (OOPSLA '89)*. ACM, New York, NY, USA, 49–70. https://doi.org/10.1145/74877.74884

Maxime Chevalier-Boisvert and Marc Feeley. 2015. Simple and Effective Type Check Removal through Lazy Basic Block Versioning. In *29th European Conference on Object-Oriented Programming (ECOOP 2015) (Leibniz International Proceedings in Informatics (LIPIcs))*, John Tang Boyland (Ed.), Vol. 37. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 101–123. https://doi.org/10.4230/LIPIcs.ECOOP.2015.101

Maxime Chevalier-Boisvert and Marc Feeley. 2016. Interprocedural Type Specialization of JavaScript Programs Without Type Analysis. In *30th European Conference on Object-Oriented Programming (ECOOP 2016) (Leibniz International Proceedings in Informatics (LIPIcs))*, Shriram Krishnamurthi and Benjamin S. Lerner (Eds.), Vol. 56. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 7:1–7:24. https://doi.org/10.4230/LIPIcs.ECOOP.2016.7

Robert Bruce Findler and Matthias Felleisen. 2001. Contract Soundness for Object-oriented Languages. In *Proceedings of the 16th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '01)*. ACM, New York, NY, USA, 1–15. https://doi.org/10.1145/504282.504283

Robert Bruce Findler and Matthias Felleisen. 2002. Contracts for Higher-order Functions. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming (ICFP '02)*. ACM, New York, NY, USA, 48–59. https://doi.org/10.1145/581478.581484

Andreas Gal, Brendan Eich, Mike Shaver, David Anderson, David Mandelin, Mohammad R. Haghighat, Blake Kaplan, Graydon Hoare, Boris Zbarsky, Jason Orendorff, Jesse Ruderman, Edwin W. Smith, Rick Reitmaier, Michael Bebenita, Mason Chang, and Michael Franz. 2009. Trace-based Just-in-time Type Specialization for Dynamic Languages. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '09)*. ACM, New York, NY, USA, 465–478. https://doi.org/10.1145/1542476.1542528

Arjun Guha, Jacob Matthews, Robert Bruce Findler, and Shriram Krishnamurthi. 2007. Relationally-parametric polymorphic contracts. In *Proceedings of the 2007 symposium on Dynamic languages*. ACM, 29–40.

Aseem Rastogi, Nikhil Swamy, Cédric Fournet, Gavin Bierman, and Panagiotis Vekris. 2015. Safe & Efficient Gradual Typing for TypeScript. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '15)*. ACM, New York, NY, USA, 167–180. https://doi.org/10.1145/2676726.2676971

Gregor Richards, Francesco Zappa Nardelli, and Jan Vitek. 2015. Concrete types for TypeScript. In *LIPIcs-Leibniz International Proceedings in Informatics*, Vol. 37. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

Jeremy G Siek, Michael M Vitousek, Matteo Cimini, and John Tang Boyland. 2015a. Refined criteria for gradual typing. In *LIPIcs-Leibniz International Proceedings in Informatics*, Vol. 32. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

Jeremy G Siek, Michael M Vitousek, Matteo Cimini, Sam Tobin-Hochstadt, and Ronald Garcia. 2015b. Monotonic references for efficient gradual typing. In *European Symposium on Programming Languages and Systems*. Springer, 432–456.

Jeremy G. Siek and Philip Wadler. 2010. Threesomes, with and Without Blame. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '10)*. ACM, New York, NY, USA, 365–376. https://doi.org/10.1145/1706299.1706342

Asumu Takikawa, Daniel Feltey, Ben Greenman, Max S. New, Jan Vitek, and Matthias Felleisen. 2016. Is Sound Gradual Typing Dead?. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '16)*. ACM, New York, NY, USA, 456–468. https://doi.org/10.1145/2837614.2837630

Sam Tobin-Hochstadt and Matthias Felleisen. 2008. The Design and Implementation of Typed Scheme. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '08)*. ACM, New York, NY, USA, 395–406. https://doi.org/10.1145/1328438.1328486

Michael M. Vitousek, Andrew M. Kent, Jeremy G. Siek, and Jim Baker. 2014. Design and Evaluation of Gradual Typing for Python. In *Proceedings of the 10th ACM Symposium on Dynamic Languages (DLS '14)*. ACM, New York, NY, USA, 45–56. https://doi.org/10.1145/2661088.2661101

Philip Wadler and Robert Bruce Findler. 2009. Well-typed programs canâĂŹt be blamed. In *European Symposium on Programming*. Springer, 1–16.

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

Tobias Wrigstad, Francesco Zappa Nardelli, Sylvain Lebresne, Johan Östlund, and Jan Vitek. 2010. Integrating Typed and Untyped Code in a Scripting Language. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '10)*. ACM, New York, NY, USA, 377–388. https://doi.org/10.1145/1706299.1706343

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