Refinement Types for TypeScript

Panagiotis Vekris Benjamin Cosman Ranjit Jhala
University of California, San Diego, USA
{pvekris, blcosman, jhala}@cs.ucsd.edu

Abstract

We present Refined TypeScript (RSC), a lightweight refinement type system for TypeScript, that enables static verification of higher-order, imperative programs. We develop a formal system for RSC that delineates the interaction between refinement types and mutability, and enables flow-sensitive reasoning by translating input programs to an equivalent intermediate SSA form. By establishing type safety for the intermediate form, we prove safety for the input programs. Next, we extend the core to account for imperative and dynamic features of TypeScript, including overloading, type reflection, ad hoc type hierarchies and object initialization. Finally, we evaluate RSC on a set of real-world benchmarks, including parts of the Octane benchmarks, D3, Transducers, and the TypeScript compiler. We show how RSC successfully establishes a number of value dependent properties, such as the safety of array accesses and downcasts, while incurring a modest overhead in type annotations and code restructuring.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features – Classes and objects, Constraints, Polymorphism; F.3.3 [*Logics and Meanings of Programs*]: Studies of Program Constructs – Object-oriented constructs, Type structure

General Terms Languages, Verification

Keywords Refinement Types, TypeScript, Type Systems, Immutability

1. Introduction

Modern *scripting* languages – like JavaScript, Python, and Ruby – have popularized the use of higher-order constructs that were once solely in the *functional* realm. This trend towards abstraction and reuse poses two related problems for static analysis: *modularity* and *extensibility*. First, how should analysis precisely track the flow of values across higher-order functions and containers or *modularly* account for external code like closures or library calls? Second, how can analyses be easily *extended* to new, domain specific properties, ideally by developers, while they are designing and implementing the code? (As opposed to by experts who can at best develop custom analyses run *ex post facto* and are of little use *during* development.)

Refinement types hold the promise of a precise, modular and extensible analysis for programs with higher-order functions and containers. Here, *basic* types are decorated with *refinement* predicates that constrain the values inhabiting the type [29, 40]. The extensibility and modularity offered by refinement types have enabled their use in a variety of applications in *typed*, *functional* languages, like ML [28, 40], Haskell [37], and $F^\sharp$ [33]. Unfortunately, attempts to apply refinement typing to scripts have proven to be impractical due to the interaction of the machinery that accounts for imperative updates and higher-order functions [5] (§6).

In this paper, we introduce Refined TypeScript (RSC): a novel, *lightweight* refinement type system for TypeScript, a typed superset of JavaScript. Our design of RSC addresses three intertwined problems by carefully integrating and extending existing ideas from the literature. First, RSC accounts for *mutation* by using ideas from IGJ [42] to track which fields may be mutated, and to allow refinements to depend on immutable fields, and by using SSA-form to recover path and flow-sensitivity. Second, RSC accounts for *dynamic typing* by using a recently proposed technique called Two-Phase Typing [39], where dynamic behaviors are specified via union and intersection types, and verified by reduction to refinement typing. Third, the above are carefully designed to permit refinement *inference* via the Liquid Types [28] framework to render refinement typing practical on real-world programs. Concretely, we make the following contributions:

- We develop a core calculus that permits locally flow-sensitive reasoning via SSA translation and formalizes the interaction of mutability and refinements via declarative refinement type checking that we prove sound (§3).

PLDI'16, June 13–17, 2016, Santa Barbara, CA, USA
© 2016 ACM. 978-1-4503-4261-2/16/06...$15.00
http://dx.doi.org/10.1145/2908080.2908110

```
function reduce(a, f, x) {
  var res = x;
  for (var i = 0; i < a.length; i++)
    res = f(res, a[i], i);
  return res;
}

function minIndex(a) {
  if (a.length ≤ 0) return -1;
  function step(min, cur, i) {
    return cur < a[min] ? i : min;
  }
  return reduce(a, step, 0);
}
```

Figure 1: Computing the Min-Valued Index with `reduce`

- We extend the core language to TypeScript by describing how we account for its various *dynamic* and *imperative* features; in particular we show how RSC accounts for type reflection via intersection types, encodes interface hierarchies via refinements and handles object initialization (§4).
- We implement `rsc`, a refinement type checker for TypeScript, and evaluate it on a suite of real-world programs from the Octane benchmarks, Transducers, D3 and the TypeScript compiler [1]. We show that RSC's refinement typing is *modular* enough to analyze higher-order functions, collections and external code, and *extensible* enough to verify a variety of properties from classic array-bounds checking to program specific invariants needed to ensure safe reflection: critical invariants that are well beyond the scope of existing techniques for imperative scripting languages (§5).

2. Overview

We begin with a high-level overview of refinement types in RSC, their applications (§2.1), and how RSC handles imperative, higher-order constructs (§2.2).

Types and Refinements A basic refinement type is a basic type, *e.g.* `number`, refined with a logical formula from an SMT decidable logic [24]. For example, the types

```
type nat     = {v:number | 0 ≤ v}
type pos     = {v:number | 0 < v}
type natN<n> = {v:nat    | v = n}
type idx<a>  = {v:nat    | v < len(a)}
```

describe (the set of values corresponding to) *non-negative* numbers, *positive* numbers, numbers *equal to* some value `n`, and *valid indexes* for an array `a`, respectively. Here, `len` is an *uninterpreted function* that describes the size of the array `a`. We write `t` to abbreviate trivially refined types, *i.e.* `{v:t | true}`; *e.g.* `number` abbreviates `{v:number | true}`.

[1] Our implementation and benchmarks can be found at `https://github.com/UCSD-PL/refscript`.

Summaries Function Types $(x_1 : \mathsf{T}_1, \ldots, x_n : \mathsf{T}_n) \Rightarrow \mathsf{T}$, where arguments are named x_i and have types T_i and the output is a T, are used to specify the behavior of functions. In essence, the *input* types T_i specify the function's preconditions, and the *output* type T describes the postcondition. Each input type and the output type can *refer to* the arguments x_i, yielding precise function contracts. For example, $(x{:}\mathsf{nat}) \Rightarrow \{\nu{:}\mathsf{nat} \mid x < \nu\}$ is a function type that describes functions that *require* a non-negative input, and *ensure* that the output exceeds the input.

Higher-Order Summaries This approach generalizes directly to precise descriptions for *higher-order* functions. For example, `reduce` from Figure 1 can be specified as $T_{\texttt{reduce}}$:

```
<A,B>(a:A[], f:(B, A, idx<a>)⇒B, x:B)⇒B        (1)
```

This type is a precise *summary* for the higher-order behavior of `reduce`: it describes the relationship between the input array `a`, the step ("callback") function `f`, and the initial value of the accumulator, and stipulates that the output satisfies the same *properties* `B` as the input `x`. Furthermore, it critically specifies that the callback `f` is only invoked on valid indices for the array `a` being reduced.

2.1 Applications

Next, we show how refinement types let programmers *specify* and statically *verify* a variety of properties — array safety, reflection (value-based overloading), and downcasts — potential sources of runtime problems that cannot be prevented via existing techniques.

2.1.1 Array Bounds

Specification We specify safety by defining suitable refinement types for array creation and access. For example, we view read `a[i]`, write `a[i] = e` and length access `a.length` as calls `get(a,i)`, `set(a,i,e)` and `length(a)` where

```
get    : <T>(a:T[],i:idx<a>) ⇒ T
set    : <T>(a:T[],i:idx<a>,e:T) ⇒ void
length : <T>(a:T[]) ⇒ natN<len(a)>
```

Verification Refinement typing ensures that the *actual* parameters supplied at each *call* to `get` and `set` are subtypes of the *expected* values specified in the signatures, and thus verifies that all accesses are safe. As an example, consider the function that returns the "head" element of an array:

```
function head<T>(arr:NEArray<T>){
  return arr[0];
}
```

The input type requires that `arr` be *non-empty*:

```
type NEArray<T> = {v:T[] | 0 < len(v)}
```

We convert `arr[0]` to `get(arr,0)` which is checked under environment $\Gamma_{\texttt{head}}$ defined as $\texttt{arr} : \{\nu{:}\texttt{T[]} \mid 0 < \texttt{len}(\nu)\}$ yielding the subtyping obligation

$$\Gamma_{\texttt{head}} \vdash \{\nu = 0\} \sqsubseteq \texttt{idx}\,\langle \texttt{arr} \rangle$$

which reduces to the logical *verification condition* (VC)

$$0 < \texttt{len}(\texttt{arr}) \Rightarrow (\nu = 0 \Rightarrow 0 \leq \nu < \texttt{len}(\texttt{arr}))$$

The VC is proved *valid* by an SMT solver [24], verifying subtyping, and hence, the array access' safety.

Path Sensitivity is obtained by adding branch conditions into the typing environment. Consider

```
function head0(a:number[]): number {
  if (0 < a.length) return head(a);
  return 0;
}
```

Recall that head should only be invoked with *non-empty* arrays. The call to head above occurs under $\Gamma_{\texttt{head0}}$ defined as: $\texttt{a} : \texttt{number[]},\ 0 < \texttt{len}(\texttt{a})$ *i.e.* which has the binder for the formal a, and the guard predicate established by the branch condition. Thus, the call to head yields the obligation

$$\Gamma_{\texttt{head0}} \vdash \{\nu = \texttt{a}\} \sqsubseteq \texttt{NEArray}\,\langle\texttt{number}\rangle$$

yielding the valid VC

$$0 < \texttt{len}(\texttt{a}) \Rightarrow (\nu = \texttt{a} \Rightarrow 0 < \texttt{len}(\nu))$$

Polymorphic, Higher-Order Functions Next, let us *assume* that reduce has the type $T_{\texttt{reduce}}$ described in (1), and see how to verify the array safety of minIndex (Figure 1). The challenge here is to precisely track which values can flow into min (used to index into a), which is tricky since those values are actually produced inside reduce.

Types make it easy to track such flows: we need only determine the *instantiation* of the polymorphic type variables of reduce at this call site inside minIndex. The type of the f parameter in the instantiated type corresponds to a signature for the closure step which will let us verify the closure's implementation. Here, rsc automatically instantiates (by building complex logical predicates from simple terms that have been predefined in a prelude)

$$\texttt{A} \mapsto \texttt{number} \qquad \texttt{B} \mapsto \texttt{idx}\,\langle\texttt{a}\rangle \tag{2}$$

Let us reassure ourselves that this instantiation is valid, by checking that step and 0 satisfy the instantiated type. If we substitute (2) into $T_{\texttt{reduce}}$ we obtain the following types for step and 0, *i.e.* reduce's second and third arguments:

```
step:(idx<a>,number,idx<a>)⇒idx<a>    0:idx<a>
```

The initial value 0 is indeed a valid idx<a> thanks to the a.length check at the start of the function. To check step, assume that its inputs have the above types:

```
min:idx<a>, curr:number, i:idx<a>
```

The body is safe as the index i is trivially a subtype of the required idx<a>, and the output is one of min or i and hence, of type idx<a> as required.

2.1.2 Overloading

Dynamic languages extensively use *value-based overloading* to simplify library interfaces. For example, a library may export

```
function $reduce(a, f, x) {
  if (arguments.length===3) return reduce(a,f,x);
  return reduce(a.slice(1),f,a[0]);
}
```

The function $reduce has *two* distinct types depending on its parameters' *values*, rendering it impossible to statically type without path-sensitivity. Such overloading is ubiquitous: in more than 25% of TypeScript libraries, more than 25% of the functions are value-overloaded [39].

Intersection Types Refinements let us statically *verify* value-based overloading via *Two-Phase Typing* [39]. First, we specify overloading as an intersection type. For example, $reduce gets the following signature, which is just the conjunction of the two overloaded behaviors:

```
<A>   (a:A[]+, f:(A, A, idx<a>)⇒A)⇒A       // 1
<A,B>(a:A[] , f:(B, A, idx<a>)⇒B, x:B)⇒B // 2
```

The type $\texttt{A[]}^+$ in the first conjunct indicates that the first argument needs to be a non-empty array, so that the call to slice and the access of a[0] both succeed.

Dead Code Assertions Second, we check each conjunct separately, replacing ill-typed terms in each context with assert(false). This requires the refinement type checker to prove that the corresponding expressions are *dead code*, as assert requires its argument to always be true:

```
assert : <A>(b:{v:bool | v = true}) ⇒ A
```

To check $reduce, we specialize it per overload context:

```
function $reduce1(a,f) {
  if (arguments.length===3) return assert(false);
  return reduce(a.slice(1), f, a[0]);
}

function $reduce2(a,f,x) {
  if (arguments.length===3) return reduce(a,f,x);
  return assert(false);
}
```

In each case, the "ill-typed" term (for the corresponding input context) is replaced with assert(false). Refinement typing easily verifies the asserts, as they respectively occur under the *inconsistent* environments

$$\Gamma_1 \doteq \texttt{arguments}{:}\{\texttt{len}(\nu) = 2\},\ \texttt{len}(\texttt{arguments}) = 3$$
$$\Gamma_2 \doteq \texttt{arguments}{:}\{\texttt{len}(\nu) = 3\},\ \texttt{len}(\texttt{arguments}) \neq 3$$

which bind arguments to an array-like object corresponding to the arguments passed to that function, and include the branch condition under which the call to assert occurs.

2.2 Analysis

Next, we outline how rsc uses refinement types to analyze programs with closures, polymorphism, assignments, classes and mutation.

2.2.1 Polymorphic Instantiation

`rsc` uses the framework of Liquid Typing [28] to *automatically synthesize* the instantiations of (2). In a nutshell, `rsc` (a) creates *templates* for unknown refinement type instantiations, (b) performs type checking over the templates to generate *subtyping constraints* over the templates that capture value-flow in the program, (c) solves the constraints via a *fixpoint* computation (abstract interpretation).

Step 1: Templates Recall that `reduce` has the polymorphic type $T_{\texttt{reduce}}$. At the call-site in `minIndex`, the type variables `A`, `B` are instantiated with the *known* base-type `number`. Thus, `rsc` creates fresh templates for the (instantiated) `A`, `B`:

$$\texttt{A} \mapsto \{\nu\texttt{:number} \mid \kappa_A\} \qquad \texttt{B} \mapsto \{\nu\texttt{:number} \mid \kappa_B\}$$

where the *refinement variables* κ_A and κ_B represent the *unknown refinements*. We substitute the above in the signature for `reduce` to obtain a *context-sensitive* template

$$(\texttt{a}\texttt{:}\kappa_A\texttt{[]},\ (\kappa_B, \kappa_A, \texttt{idx}\,\langle\texttt{a}\rangle) \Rightarrow \kappa_B,\ \kappa_B) \Rightarrow \kappa_B \qquad (3)$$

Step 2: Constraints Next, `rsc` generates *subtyping* constraints over the templates. Intuitively, the templates describe the *sets* of values that each static entity (*e.g.* variable) can evaluate to at runtime. The subtyping constraints capture the *value-flow* relationships *e.g.* at assignments, calls and returns, to ensure that the template solutions – and hence inferred refinements – soundly over-approximate the set of runtime values of each corresponding static entity.

We generate constraints by performing type checking over the templates. As `a`, `0`, and `step` are passed in as arguments, we check that they respectively have the types κ_A`[]`, κ_B and $(\kappa_B, \kappa_A, \texttt{idx}\,\langle\texttt{a}\rangle) \Rightarrow \kappa_B$. Checking `a` and `0` yields the subtyping constraints

$$\Gamma \vdash \texttt{number[]} \sqsubseteq \kappa_A\texttt{[]} \qquad \Gamma \vdash \{\nu = 0\} \sqsubseteq \kappa_B$$

where $\Gamma \doteq \texttt{a:number[]},\ 0 < \texttt{len(a)}$ from the *else-guard* that holds at the call to `reduce`. We check `step` by checking its body under the environment $\Gamma_{\texttt{step}}$ that binds the input parameters to their respective types

$$\Gamma_{\texttt{step}} \doteq \texttt{min:}\kappa_B,\ \texttt{cur:}\kappa_a,\ i\texttt{:idx}\,\langle\texttt{a}\rangle$$

As `min` is used to index into the array `a` we get

$$\Gamma_{\texttt{step}} \vdash \kappa_B \sqsubseteq \texttt{idx}\,\langle a\rangle$$

As `i` and `min` flow to the output type κ_B, we get

$$\Gamma_{\texttt{step}} \vdash \texttt{idx}\,\langle\texttt{a}\rangle \sqsubseteq \kappa_B \qquad \Gamma_{\texttt{step}} \vdash \kappa_B \sqsubseteq \kappa_B$$

Step 3: Fixpoint The above subtyping constraints over the κ variables are reduced via the standard rules for co- and contra-variant subtyping, into *Horn implications* over the κs. `rsc` solves the Horn implications via (predicate) abstract interpretation [28] to obtain the solution $\kappa_A \mapsto$ `true` and $\kappa_B \mapsto 0 \leq \nu < \texttt{len(a)}$ which is exactly the instantiation in (2) that satisfies the subtyping constraints, and proves `minIndex` is array-safe.

2.2.2 Assignments

Next, let us see how the signature for `reduce` in Figure 1 is verified by `rsc`. Unlike in the functional setting, where refinements have previously been studied, here, we must deal with imperative features like assignments and **`for`**-loops.

SSA Transformation We solve this problem in three steps. First, we convert the code into SSA form, to introduce new binders at each assignment. Second, we generate fresh templates that represent the unknown types (*i.e.* set of values) for each Φ-variable. Third, we generate and solve the subtyping constraints to infer the types for the Φ-variables, and hence, the "loop-invariants" needed for verification.

Let us see how this process lets us verify `reduce` from Figure 1. First, we convert the body to SSA form (§3.3):

```
function reduce(a, f, x) {
  var r0 = x, i0 = 0;
  while[i2 ≐ ϕ(i0,i1), r2 ≐ ϕ(r0,r1)](i2 < a.length) {
    r1 = f(r2, a[i2], i2);
    i1 = i2 + 1;
  }
  return r2;
}
```

where `i2` and `r2` are the Φ-variables for `i` and `r` respectively. Second, we generate templates for the Φ-variables:

$$\texttt{i2:}\{\nu\texttt{:number} \mid \kappa_{i2}\} \qquad \texttt{r2:}\{\nu\texttt{:B} \mid \kappa_{r2}\} \qquad (4)$$

We need not generate templates for the SSA variables `i0`, `r0`, `i1` and `r1` as their types are those of the expressions they are assigned. Third, we generate subtyping constraints as before; the Φ-assignment generates *additional* constraints

$$\Gamma_0 \vdash \{\nu = \texttt{i0}\} \sqsubseteq \kappa_{i2} \qquad \Gamma_1 \vdash \{\nu = \texttt{i1}\} \sqsubseteq \kappa_{i2}$$
$$\Gamma_0 \vdash \{\nu = \texttt{r0}\} \sqsubseteq \kappa_{r2} \qquad \Gamma_1 \vdash \{\nu = \texttt{r1}\} \sqsubseteq \kappa_{r2}$$

where Γ_0 is the environment at the "exit" of the basic block where `i0` and `r0` are defined:

$$\Gamma_0 \doteq \texttt{a:number[]},\ \texttt{x:B},\ \texttt{i0:natN}\,\langle 0\rangle,\ \texttt{r0:}\{\nu\texttt{:B} \mid \nu = \texttt{x}\}$$

Similarly, the environment Γ_1 includes bindings for variables `i1` and `r1`. In addition, code executing the loop body has passed the conditional check, so our *path-sensitive* environment is strengthened by the corresponding guard:

$$\Gamma_1 \doteq \Gamma_0,\ \texttt{i1:natN}\,\langle\texttt{i2} + 1\rangle,\ \texttt{r1:B},\ \texttt{i2} < \texttt{len(a)}$$

Finally, the above constraints are solved to

$$\kappa_{i2} \mapsto 0 \leq \nu < \texttt{len(a)} \qquad \kappa_{r2} \mapsto \texttt{true}$$

which verifies that the "callback" `f` is indeed called with values of type $\texttt{idx}\,\langle\texttt{a}\rangle$, as it is only called with $\texttt{i2} : \texttt{idx}\,\langle\texttt{a}\rangle$, obtained by plugging the solution into the template in (4).

```
type ArrayN<T,n> = {v:T[] | len(v) = n}
type grid<w,h>   = ArrayN<number,(w+2)*(h+2)>
type okW         = natLE<this.w>
type okH         = natLE<this.h>

class Field {
  immutable w: pos;
  immutable h: pos;
  dens       : grid<this.w, this.h>;

  constructor(w:pos, h:pos, d:grid<w,h>) {
    this.h = h; this.w = w; this.dens = d;
  }
  setDensity(x:okW, y:okH, d:number) {
    var rowS = this.w + 2;
    var i    = x+1 + (y+1) * rowS;
    this.dens[i] = d;
  }
  getDensity(x:okW, y:okH): number {
    var rowS = this.w + 2;
    var i    = x+1 + (y+1) * rowS;
    return this.dens[i];
  }
  reset(d:grid<this.w,this.h>) {
    this.dens = d;
  }
}
```

Figure 2: Two-Dimensional Arrays

2.2.3 Mutability

In the imperative, object-oriented setting (common in dynamic scripting languages), we must account for *class* and *object* invariants and their preservation in the presence of field mutation. For example, consider the code in Figure 2, modified from the Octane Navier-Stokes benchmark.

Class Invariants Class `Field` implements a *2-dimensional* vector, "unrolled" into a single array `dens`, whose size is the product of the width and height fields. We specify this invariant by requiring that width and height be strictly positive (*i.e.* `pos`) and that `dens` be a `grid` with dimensions specified by `this.w` and `this.h`. An advantage of SMT-based refinement typing is that modern SMT solvers support non-linear reasoning, which lets `rsc` specify and verify program specific invariants outside the scope of generic bound checkers.

Mutable and Immutable Fields The above invariants are only meaningful and sound if fields `w` and `h` cannot be modified after object creation. We specify this via the `immutable` qualifier, which is used by `rsc` to then (1) *prevent* updates to the field outside the `constructor`, and (2) *allow* refinements of fields (*e.g.* `dens`) to soundly refer to the values of those immutable fields.

Constructors We can create *instances* of `Field`, by using `new Field(...)` which invokes the `constructor` with the supplied parameters. `rsc` ensures that at the *end* of the constructor, the created object actually satisfies all specified class invariants *i.e.* field refinements. Of course, this only holds if the parameters passed to the constructor satisfy certain preconditions, specified via the input types. Consequently, `rsc` accepts the first call, but rejects the second:

```
var z = new Field(3,7,new Array(45)); // OK
var q = new Field(3,7,new Array(44)); // BAD
```

Methods `rsc` uses class invariants to verify `setDensity` and `getDensity`, that are checked *assuming* that the fields of `this` enjoy the class invariants, and method inputs satisfy their given types. The resulting VCs are valid and hence, check that the methods are array-safe. Of course, clients must supply appropriate arguments to the methods. Thus, `rsc` accepts the first call, but rejects the second as the `x` coordinate `5` exceeds the actual width (*i.e.* `z.w`), namely `3`:

```
z.setDensity(2, 5, -5)   // OK
z.getDensity(5, 2);      // BAD
```

Mutation The `dens` field is *not* `immutable` and hence, may be updated outside of the constructor. However, `rsc` requires that the class invariants still hold, and this is achieved by ensuring that the *new* value assigned to the field also satisfies the given refinement. Thus, the `reset` method requires inputs of a specific size, and updates `dens` accordingly. Hence:

```
var z = new Field(3,7,new Array(45));
z.reset(new Array(45));  // OK
z.reset(new Array(5));   // BAD
```

3. Formal System

Next, we formalize the ideas outlined in §2. We introduce our formal core FRSC (§3.1): an imperative, mutable, object-oriented subset of Refined TypeScript, that resembles the core of Safe TypeScript [27]. To ease refinement reasoning, we SSA-transform (§3.3) FRSC to a functional, yet still mutable, intermediate language IRSC (§3.2) that closely follows the design of CFJ [25] (the language used to formalize X10), which in turn is based on Featherweight Java [18]. We then formalize our static semantics in terms of IRSC (§3.4), prove them sound and connect them to those of FRSC (§3.5).

3.1 Source Language (FRSC)

The syntax of this language is given below. Meta-variable `e` ranges over expressions, which can be variables `x`, constants `c`, property accesses `e.f`, method calls `e.m(`$\overline{\texttt{e}}$`)`, object creations `new` C`(`$\overline{\texttt{e}}$`)`, and cast operations `<`T`>e`. Statements `s` include expressions, variable declarations, field updates, assignments, concatenations, conditionals and empty statements. Method declarations include a type signature, specifying input and output types, and a body `B`, *i.e.* a statement immediately followed by a returned expression. We distinguish between immutable and mutable class members, using $\circ$`f:`T and $\square$`f:`T, respectively. Finally, class and method definitions are associated with an invariant predicate p.

$$
\begin{array}{lcl}
\texttt{e} & ::= & \texttt{x} \mid \texttt{c} \mid \texttt{this} \mid \texttt{e.f} \mid \texttt{e.m}(\overline{\texttt{e}}) \mid \texttt{new}\ C(\overline{\texttt{e}}) \mid \texttt{<}T\texttt{>e} \\
\texttt{s} & ::= & \texttt{e} \mid \texttt{var x = e} \mid \texttt{e.f = e} \mid \texttt{x = e} \mid \texttt{s; s} \mid \\
 & & \texttt{if(e)\{s\}else\{s\}} \mid \texttt{skip} \\
\texttt{B} & ::= & \texttt{s; return e} \\
\widetilde{\texttt{M}} & ::= & \texttt{m}(\overline{\texttt{x}}\texttt{:}\ \overline{T})\ \{p\}\texttt{:}\,T\ \texttt{\{B\}} \\
\texttt{F} & ::= & \cdot \mid \circ\,\texttt{f}\texttt{:}\,T \mid \square\,\texttt{f}\texttt{:}\,T \mid \texttt{F}_1\texttt{;}\,\texttt{F}_2 \\
\widetilde{\texttt{C}} & ::= & \texttt{class}\ C\ \{p\}\ \texttt{extends}\ R\ \texttt{\{F,}\widetilde{\texttt{M}}\texttt{\}}
\end{array}
$$

The core system does not formalize (a) method overloading, which is orthogonal to the current contribution and has been investigated in previous work [39], or (b) method overriding, which means that method names are distinct from the ones defined in parent classes.

3.2 Intermediate Language (IRSC)

To maintain precision for stack-allocated variables, we transform FRSC programs into equivalent (in a sense that we will make precise in the sequel) programs in a functional language IRSC through SSA renaming. In IRSC, statements are replaced by let-bindings and new variables are introduced for each reassigned variable in FRSC code. Thus, IRSC has the following syntax:

$$
\begin{array}{lcl}
e & ::= & x \mid c \mid \mathbf{this} \mid e.f \mid e.m\,(\overline{e}) \mid \mathbf{new}\ C\,(\overline{e}) \mid \\
 & & e\ \mathbf{as}\ T \mid e.f \leftarrow e \mid u\,\langle e \rangle \mid \\
u & ::= & \langle\,\rangle \mid \mathbf{let}\ x = e\ \mathbf{in}\ \langle\,\rangle \mid \\
 & & \mathbf{letif}\,[\,\overline{x}, \overline{x}_1, \overline{x}_2\,]\,(e)\,?\,u_1 : u_2\ \mathbf{in}\ \langle\,\rangle \\
F & ::= & \cdot \mid \circ\, f\texttt{:}\,T \mid \square\, f\texttt{:}\,T \mid F_1\texttt{;}\ F_2 \\
\widetilde{M} & ::= & \cdot \mid \mathbf{def}\ m\,(\,\overline{x}\texttt{:}\,\overline{T}\,)\ \{p\}\texttt{:}\,T = e \mid \widetilde{M}_1\texttt{;}\ \widetilde{M}_2 \\
\widetilde{C} & ::= & \mathbf{class}\ C\ \{p\}\ \triangleleft\ R\ \{F\texttt{;}\ \widetilde{M}\}
\end{array}
$$

The majority of the expression forms e are unsurprising. An exception is the form of the *SSA context* u, which corresponds to the translation of a statement `s` and contains a *hole* $\langle\,\rangle$ that will hold the translation of the continuation of `s`. Form $u\,\langle e \rangle$ fills the hole of u with expression e.

3.3 Static Single Assignment (SSA) Transformation

Figure 3 describes the SSA transformation, that uses *translation environments* δ to map FRSC variables `x` to IRSC variables x. The translation of expressions `e` to e is routine; as expected, S-VAR maps a variable `x` to its bindings x in δ. The translating judgment for statements `s` has the form $\delta \Vdash \texttt{s} \hookrightarrow u; \delta'$. The output environment δ' is used for the translation of the expression that will fill the hole in u.

The most interesting case is the conditional statement (rule S-ITE). The condition expression and each branch are translated separately. To compute the variables that get updated in either branch (Φ-variables), we combine the produced translation states δ_1 and δ_2 as $\delta_1 \bowtie \delta_2$ defined as

$$\{(\texttt{x}, x_1, x_2) \mid \texttt{x} \mapsto x_1 \in \delta_1,\ \texttt{x} \mapsto x_2 \in \delta_2,\ x_1 \neq x_2\}$$

Fresh Φ-variables $\overline{x}'$ populate the output environment δ' and annotate the produced structure, along with the versions of the Φ-variables at the end of each branch ($\overline{x}_1$ and $\overline{x}_2$).

Assignment statements introduce a new SSA variable and bind it to the updated source-level variable (rule S-ASGN). Statement sequencing is emulated with nesting SSA contexts (rule S-SEQ); empty statements introduce a hole (rule S-SKIP); and, finally, method declarations fill in the hole introduced by the method statement with the translation of the return expression (rule S-MDECL).

Consistency To validate our transformation, we provide a consistency result that guarantees that stepping in the target language preserves the transformation relation, after the program in the source language has made an appropriate number of steps. We define a *runtime configuration* `Q` for FRSC (resp. Q for IRSC) for a *program* `P` (resp. P) as:

$$
\begin{array}{ll}
\texttt{P} \doteq \texttt{S;B} & P \doteq S; e \\
\texttt{Q} \doteq \texttt{K;B} & Q \doteq K; e \\
\texttt{K} \doteq \texttt{S;L;X;H} & K \doteq S; H
\end{array}
$$

Similar to Safe TypeScript [27], a *runtime state* `K` consists of *class signatures* `S`, a *call stack* `X`, a *local store* `L` of the current stack frame and a *heap* `H`. The runtime state K for IRSC only consists of signatures S and a heap H. SSA *consistency* is established via a weak forward simulation theorem that connects the dynamic semantics of the two languages, expressed through the reduction rules

$$\texttt{Q} \longrightarrow \texttt{Q}' \qquad\qquad Q \longrightarrow Q'$$

Rules for FRSC are adapted from Safe TypeScript and the rules for IRSC are straightforward, so we leave the details to the extended version [38]. Figure 4 presents some interesting cases: (a) To emulate TypeScript's type erasure, rule Q-CAST of FRSC trivially steps a cast operation to the enclosed expression. The corresponding rule R-CAST of IRSC, on the other hand, checks that the content of the cast location satisfies the cast type (Corollary 4 deems this check redundant). (b) In rule R-LIF of IRSC, expression e is produced assuming Φ-variables $\overline{x}$, so as soon as the branch has been determined, $\overline{x}$ are substituted for $\overline{x}_1$ or $\overline{x}_2$ (depending on the branch) in e. This formulation allows us to perform all SSA-related book-keeping in a single step, which is key to preserving the *invariant* that IRSC steps faster than FRSC.

We also extend our SSA transformation judgment to runtime configurations, leveraging the SSA environments that have been statically computed for each program entity. A *global SSA environment* Δ is used to map each AST node (`e`, `s`, *etc.*) to an SSA environment δ:

$$\Delta \ ::= \ \cdot \mid \texttt{e} \mapsto \delta \mid \texttt{s} \mapsto \delta \mid \ldots \mid \Delta_1; \Delta_2$$

We assume that the compile-time SSA translation yields this environment as a side-effect (*e.g.* $\delta \Vdash \texttt{e} \hookrightarrow e$ produces $\texttt{e} \mapsto \delta$) and the top-level program transformation judgment returns the net effect: $\texttt{P} \hookrightarrow P \rightsquigarrow \Delta$. Hence, the SSA transformation judgment for configurations becomes: $\texttt{K;B} \overset{\Delta}{\hookrightarrow} K; e$. We can now state our simulation theorem as:

$$\boxed{\delta \Vdash \mathtt{e} \hookrightarrow e \qquad \delta \Vdash \mathtt{s} \hookrightarrow u; \delta' \qquad \delta \Vdash \mathtt{B} \hookrightarrow e \qquad \widetilde{\mathtt{M}} \hookrightarrow \widetilde{M}}$$

$$\textsc{S-Var}\ \frac{}{\delta \Vdash \mathtt{x} \hookrightarrow \delta\,(\mathtt{x})} \qquad \textsc{S-This}\ \frac{}{\delta \Vdash \mathtt{this} \hookrightarrow \mathbf{this}}$$

$$\textsc{S-VarDecl}\ \frac{\delta \Vdash \mathtt{e} \hookrightarrow e \qquad \delta' = \delta[\mathtt{x} \mapsto x] \qquad x\ \mathsf{fresh}}{\delta \Vdash \mathtt{var\ x = e} \hookrightarrow \mathbf{let}\ x = e\ \mathbf{in}\ \langle\rangle; \delta'}$$

$$\textsc{S-Ite}\ \frac{\begin{array}{c}\delta \Vdash \mathtt{e} \hookrightarrow e \qquad \delta \Vdash \mathtt{s}_1 \hookrightarrow u_1; \delta_1 \qquad \delta \Vdash \mathtt{s}_2 \hookrightarrow u_2; \delta_2 \\ (\overline{\mathtt{x}}, \overline{x}_1, \overline{x}_2) = \delta_1 \bowtie \delta_2 \qquad \delta' = \delta[\overline{\mathtt{x}} \mapsto \overline{x}'] \qquad \overline{x}'\ \mathsf{fresh}\end{array}}{\delta \Vdash \mathtt{if(e)\{s_1\}else\{s_2\}} \hookrightarrow \mathbf{letif}\,[\,\overline{x}', \overline{x}_1, \overline{x}_2\,]\,(e)\,?\,u_1 : u_2\ \mathbf{in}\ \langle\rangle; \delta'}$$

$$\textsc{S-Asgn}\ \frac{\begin{array}{c}\delta \Vdash \mathtt{e} \hookrightarrow e \\ \delta' = \delta[\mathtt{x} \mapsto x'] \qquad x'\ \mathsf{fresh}\end{array}}{\delta \Vdash \mathtt{x = e} \hookrightarrow \mathbf{let}\ x' = e\ \mathbf{in}\ \langle\rangle; \delta'}$$

$$\textsc{S-DotAsgn}\ \frac{\delta \Vdash \mathtt{e} \hookrightarrow e \qquad \delta \Vdash \mathtt{e}' \hookrightarrow e'}{\delta \Vdash \mathtt{e.f = e}' \hookrightarrow \mathbf{let}\ \_ = e.f \leftarrow e'\ \mathbf{in}\ \langle\rangle; \delta} \qquad \textsc{S-Seq}\ \frac{\delta \Vdash \mathtt{s}_1 \hookrightarrow u_1; \delta_1 \qquad \delta_1 \Vdash \mathtt{s}_2 \hookrightarrow u_2; \delta_2}{\delta \Vdash \mathtt{s}_1\,;\,\mathtt{s}_2 \hookrightarrow u_1\,\langle u_2\rangle\,; \delta_2}$$

$$\textsc{S-Skip}\ \frac{}{\delta \Vdash \mathtt{skip} \hookrightarrow \langle\rangle\,; \delta}$$

$$\textsc{S-Body}\ \frac{\delta \Vdash \mathtt{s} \hookrightarrow u; \delta' \qquad \delta' \Vdash \mathtt{e} \hookrightarrow e}{\delta \Vdash \mathtt{s\,;\ return\ e} \hookrightarrow u\,\langle e\rangle} \qquad \textsc{S-MDecl}\ \frac{\mathsf{toString}\,(\mathtt{m}) = \mathsf{toString}\,(m) \qquad \{\overline{\mathtt{x}} \mapsto \overline{x}\} \Vdash \mathtt{B} \hookrightarrow e \qquad m, \overline{x}\ \mathsf{fresh}}{\mathtt{m(}\overline{\mathtt{x}}\mathtt{:}\ \overline{T}\mathtt{)}\ \{p\} : T\ \mathtt{\{B\}} \hookrightarrow \mathbf{def}\ m\,(\overline{x}{:}\overline{T})\ \{p\} : T = e}$$

Figure 3: Selected SSA Transformation Rules

$$\boxed{\mathtt{K; e} \longrightarrow \mathtt{K'; e'}} \qquad \boxed{K; e \longrightarrow K'; e'}$$

$$\mathtt{Q}\text{-}\textsc{Cast}\ \mathtt{K; <}T\mathtt{>e} \longrightarrow \mathtt{K; e} \qquad \textsc{R-Cast}\ \frac{\Gamma \vdash K.H\,(l) : S; S \leq T}{K; l\ \mathbf{as}\ T \longrightarrow K; l} \qquad \textsc{R-If}\ \frac{c = \mathsf{true} \Rightarrow i = 1 \qquad c = \mathsf{false} \Rightarrow i = 2}{K; \mathbf{letif}\,[\,\overline{x}, \overline{x}_1, \overline{x}_2\,]\,(c)\,?\,u_1 : u_2\,\mathbf{in}\,e \longrightarrow K; u_i\,\langle[\,\overline{x_i}/\overline{x}\,]\,e\rangle}$$

Figure 4: Selected Reduction Rules for FRSC and IRSC

Theorem 1 (Forward Simulation)**.** *If* $\mathtt{Q} \overset{\Delta}{\hookrightarrow} Q$*, then:*

(a) if Q *is terminal, then* $\exists\,\mathtt{Q}'$ *s.t.* $\mathtt{Q} \longrightarrow^* \mathtt{Q}'$ *and* $\mathtt{Q}' \overset{\Delta}{\hookrightarrow} Q$

(b) if $Q \longrightarrow Q'$*, then* $\exists\,\mathtt{Q}'$ *s.t.* $\mathtt{Q} \longrightarrow^* \mathtt{Q}'$ *and* $\mathtt{Q}' \overset{\Delta}{\hookrightarrow} Q'$

3.4 Static Semantics

We proceed by describing refinement checking for IRSC.

Types Type annotations on the source language are propagated unaltered through the translation phase. Our type language (shown below) resembles that of existing refinement type systems [19, 25, 28]. A *refinement type* T may be an existential type or have the form $\{\nu : N \mid p\}$, where N is a class name C or a primitive type B, and p is a logical predicate (over some decidable logic) which describes the properties that values of the type must satisfy. Type specifications (*e.g.* method types) are existential-free, while inferred types may be existentially quantified [20].

Logical Predicates Predicates p are logical formulas over terms t. These terms can be variables x, primitive constants c, the reserved value variable ν, the reserved variable this to denote the containing object, field accesses $t.f$, uninterpreted function applications $f\,(\overline{t})$ and applications of terms on built-in operators b, such as ==, <, +, etc.

$$\begin{array}{rcl} T, S, R, U & ::= & \exists x{:}T_1.\,T_2 \mid \{\nu : N \mid p\} \\ N & ::= & C \mid B \\ p & ::= & p_1 \wedge p_2 \mid \neg p \mid t \\ t & ::= & x \mid c \mid \nu \mid \mathsf{this} \mid t.f \mid f\,(\overline{t}) \mid b\,(\overline{t}) \end{array}$$

Structural Constraints Following CFJ, we reuse the notion of an Object Constraint System, to encode constraints related to the object-oriented nature of the program. Most of the rules carry over to our system. A key extension in our setting is we partition C has I (that encodes inclusion of an element I in a class C) into two cases: C hasMut I and C hasImm I, to account for elements that may be mutated. These elements can only be fields (*i.e.* there is no mutation on methods).

Environments and Well-formedness A type environment Γ contains *type bindings* $x : T$ and *guard predicates* p that encode path sensitivity. Γ is *well-formed* if all of its bindings are well-formed. A refinement type is well-formed in an environment Γ if all symbols (simple or qualified) in its logical predicate (i) are bound in Γ, and (ii) correspond to *immutable* fields of objects. We omit the rest of the well-formedness rules as they are standard in refinement type systems.

Besides well-formedness, our system's main judgment forms are those for subtyping and refinement typing [19].

Subtyping is defined by the judgment $\Gamma \vdash S \leq T$. The rules are standard among refinement type systems with ex-

$\boxed{\Gamma \vdash e : T \qquad \Gamma \vdash u \triangleright \Gamma'}$

$$\text{T-Var}\ \frac{\Gamma(x) = T}{\Gamma \vdash x : \mathsf{self}\,(T, x)} \qquad \text{T-Cst}\ \frac{}{\Gamma \vdash c : \mathsf{ty}\,(c)} \qquad \text{T-Ctx}\ \frac{\Gamma \vdash u \triangleright \overline{x} : \overline{S} \qquad \Gamma, \overline{x} : \overline{S} \vdash e : T}{\Gamma \vdash u\,\langle e \rangle : \exists \overline{x} : \overline{S}.\, T}$$

$$\text{T-Fld-I}\ \frac{\Gamma \vdash e : T \qquad z\ \mathsf{fresh} \qquad \Gamma, z : T \vdash z\ \mathsf{hasImm}\ f_i : T_i}{\Gamma \vdash e.f_i : \exists z : T.\, \mathsf{self}\,(T_i, z.f_i)}$$

$$\text{T-Fld-M}\ \frac{\Gamma \vdash e : T \qquad \Gamma, z : T \vdash z\ \mathsf{hasMut}\ g_i : T_i \qquad z\ \mathsf{fresh}}{\Gamma \vdash e.g_i : \exists z : T.\, T_i} \qquad \text{T-Inv}\ \frac{\Gamma \vdash e : T, \overline{e} : \overline{T} \qquad \Gamma, z : T \vdash z\ \mathsf{has}\ \left(\mathbf{def}\ m\left(\overline{z} : \overline{R}\right)\{p\} : S = e'\right) \qquad \Gamma, z : T, \overline{z} : \overline{T} \vdash \overline{T} \leq \overline{R},\ p \qquad z,\ \overline{z}\ \mathsf{fresh}}{\Gamma \vdash e.m\,(\overline{e}) : \exists z : T.\, \exists \overline{z} : \overline{T}.\, S}$$

$$\text{T-DotAsgn}\ \frac{\Gamma \vdash e_1 : T_1, e_2 : T_2 \qquad \Gamma, z_1 : \lfloor T_1 \rfloor \vdash z_1\ \mathsf{hasMut}\ f : S,\ T_2 \leq S \qquad z_1\ \mathsf{fresh}}{\Gamma \vdash e_1.f \leftarrow e_2 : T_2}$$

$$\text{T-New}\ \frac{\Gamma \vdash \overline{e} : \left(\overline{T}_{\mathtt{I}}, \overline{T}_{\mathtt{M}}\right) \qquad \vdash \mathsf{class}\,(C) \qquad \Gamma, z : C \vdash \mathsf{fields}\,(z) = \circ\, \overline{f} : \overline{R},\ \square\, \overline{g} : \overline{U} \qquad \Gamma, z : C, \overline{z}_{\mathtt{I}} : \mathsf{self}\left(\overline{T}_{\mathtt{I}}, z.\overline{f}\right) \vdash \overline{T}_{\mathtt{I}} \leq \overline{R},\ \overline{T}_{\mathtt{M}} \leq \overline{U},\ \mathsf{inv}\,(C, z) \qquad z, \overline{z}_{\mathtt{I}}\ \mathsf{fresh}}{\Gamma \vdash \mathbf{new}\ C\,(\overline{e}) : \exists \overline{z}_{\mathtt{I}} : \overline{T}_{\mathtt{I}}.\, \{\nu : C \mid \nu.\overline{f} = \overline{z}_{\mathtt{I}} \wedge \mathsf{inv}\,(C, \nu)\}}$$

$$\text{T-Cast}\ \frac{\Gamma \vdash e : S \qquad \Gamma \vdash T \qquad \Gamma \vdash S \lesssim T}{\Gamma \vdash e\ \mathbf{as}\ T : T} \qquad \text{T-CtxEmp}\ \Gamma \vdash \langle\,\rangle \triangleright \cdot$$

$$\text{T-LetIn}\ \frac{\Gamma \vdash e : T}{\Gamma \vdash \mathbf{let}\ x = e\ \mathbf{in}\ \langle\,\rangle \triangleright x : T} \qquad \text{T-LetIf}\ \frac{\Gamma \vdash e : S,\ S \leq \mathsf{bool} \qquad \Gamma, z : S, z \vdash u_1 \triangleright \Gamma_1 \qquad \Gamma, z : S, \neg z \vdash u_2 \triangleright \Gamma_2 \qquad \Gamma,\ \Gamma_1 \vdash \Gamma_1\,(\overline{x}_1) \leq \overline{T} \qquad \Gamma,\ \Gamma_2 \vdash \Gamma_2\,(\overline{x}_2) \leq \overline{T} \qquad \Gamma \vdash \overline{T} \qquad \overline{T}\ \mathsf{fresh}}{\Gamma \vdash \mathbf{letif}\,[\,\overline{x}, \overline{x}_1, \overline{x}_2\,]\,(e)\,?\,u_1 : u_2\ \mathbf{in}\ \langle\,\rangle \triangleright \overline{x} : \overline{T}}$$

Figure 5: Static Typing Rules for IRSC

istential types. For example, the rule for subtyping between two refinement types $\Gamma \vdash \{\nu : N \mid p\} \leq \{\nu : N \mid p'\}$ reduces to a *verification condition*: $\mathsf{Valid}\,([\![\,\Gamma\,]\!] \Rightarrow ([\![\,p\,]\!] \Rightarrow [\![\,p'\,]\!]))$, where $[\![\Gamma]\!]$ is the embedding of environment Γ into our logic accounting for both guard predicates and variable bindings:

$$[\![\Gamma]\!] \doteq \bigwedge \{p \mid p \in \Gamma\} \wedge \bigwedge \{[x/\nu]\,p, \mid x : \{\nu : N \mid p\} \in \Gamma\}$$

Here, we assume existential types are simplified to non-existential bindings when they enter the environment.

Details regarding structural and well-formedness constraints, and subtyping rules are included in the extended version [38].

Refinement Typing Rules Figure 5 contains rules of the two forms of our typing judgements: $\Gamma \vdash e : T$ and $\Gamma \vdash u \triangleright \Gamma'$. The first assigns a type T to an expression e under an environment Γ, and the second checks the body of an SSA context u under Γ and returns the environment Γ' of the variables introduced in u that are available when checking its hole (rule T-Ctx). Below, we discuss the novel rules:

[T-Fld-I] Immutable object parts can be assigned a more precise type, by leveraging the preservation of their *identity*. This notion, known as *self-strengthening* [20, 25], is defined with the aid of the *strengthening* operator $\oplus$:

$$\begin{aligned} \{\nu : N \mid p\} \oplus p' &\doteq \{\nu : N \mid p \wedge p'\} \\ (\exists x : S.\, T) \oplus p &\doteq \exists x : S.\, (T \oplus p) \\ \mathsf{self}\,(T, t) &\doteq T \oplus (\nu = t) \end{aligned}$$

[T-Fld-M] Here we avoid such strengthening, as the value of field g_i is mutable, so cannot appear in refinements.

[T-New] Similarly, only immutable fields are referenced in the refinement of the inferred type at object construction.

[T-Inv] Extracting the method signature using the has operator has already performed the necessary substitutions to account for the specific receiver object.

[T-Cast] Cast operations are checked *statically* obviating the need for a dynamic check. This rule uses the notion of *compatibility subtyping* ($\lesssim$), which is defined as:

Definition 1 (Compatibility Subtype)**.** $\Gamma \vdash S \lesssim T$ *iff* $\langle S \xrightarrow{\Gamma} \lfloor T \rfloor \rangle = R \neq \mathsf{fail}$ *with* $\Gamma \vdash R \leq T$.

Here, the operation $\lfloor T \rfloor$ extracts the base type of T, and $\langle T \xrightarrow{\Gamma} D \rangle$ succeeds when under environment Γ we can statically prove D's invariants, starting from the invariants contained in T. We use the predicate $\mathsf{inv}\,(D, \nu)$ (as in CFJ) to denote the conjunction of the class invariants of D and its supertypes (with the necessary substitutions of this by ν). We assume that part of these invariants is a predicate that states inclusion in the specific class ($\mathsf{instanceof}\,(\nu, D)$). Therefore, we can prove that T can safely be cast to D. For the output of this operation it holds that: $\lfloor \langle T \xrightarrow{\Gamma} D \rangle \rfloor = D$, which enables the use of traditional subtyping. Formally:

$$\langle \{\nu : \_ \mid p\} \xrightarrow{\Gamma} D \rangle \doteq \begin{cases} D \oplus p & \text{if } ([\![\Gamma]\!] \wedge [\![p]\!]) \Rightarrow \mathsf{inv}\,(D, \nu) \\ \mathsf{fail} & \text{otherwise} \end{cases}$$

$$\langle \exists x : S.\, T \xrightarrow{\Gamma} D \rangle \doteq \exists x : S.\, \langle T \xrightarrow{\Gamma, x : S} D \rangle$$

[T-DotAsgn] Only *mutable* fields may be reassigned.

[T-LetIf] To type conditional structures, we first infer a type for the condition and then check each of the branches u_1 and u_2, assuming that the condition is true or false, respectively, to achieve path sensitivity. Each branch assigns types to the Φ-variables which compose Γ_1 and Γ_2, and the propagated types for these variables are fresh types operating as upper bounds to their respective bindings in Γ_1 and Γ_2.

3.5 Type Safety

To state our safety results, we extend our type checking judgment to runtime locations l with the use of a *heap typing* Σ, mapping locations to types, and add a location typing rule:

$$\text{T-Loc}\ \frac{\Sigma\,(l) = T}{\Gamma; \Sigma \vdash l : T}$$

We establish type safety for IRSC in the form of a subject reduction (preservation) and a progress theorem that connect the static and dynamic semantics of IRSC. These theorems employ the notions of heap and signature well-formedness: $\Sigma \vdash H$ and $\vdash S$.

Theorem 2 (Subject Reduction)**.** *If* $\Gamma; \Sigma \vdash e : T$, $K; e \longrightarrow K'; e'$ *and* $\Sigma \vdash K.H$ *then* $\exists\, T', \Sigma' \supseteq \Sigma$ *s.t.* $\Gamma; \Sigma' \vdash e' : T'$, $\Gamma \vdash T' \lesssim T$, *and* $\Sigma' \vdash K'.H$.

Theorem 3 (Progress)**.** *If* $\Gamma; \Sigma \vdash e : T$, $\vdash S$ *and* $\Sigma \vdash H$ *then either* e *is a value, or* $\exists\, e', H', \Sigma' \supseteq \Sigma$ *s.t.* $\Sigma' \vdash H'$ *and* $S; H; e \longrightarrow S; H'; e'$.

We defer the proofs to the extended version [38]. As a corollary of the Progress Theorem we get that cast operators are guaranteed to succeed, hence they can safely be erased.

Corollary 4 (Safe Casts)**.** *Cast operations can safely be erased when compiling to executable code.*

With the use of our Simulation Theorem and extending our checking judgment for terms in IRSC to runtime configurations ($\vdash Q$), we can state a soundness result for FRSC:

Theorem 5. *(FRSC Type Safety) If* $\mathtt{Q} \overset{\Delta}{\hookrightarrow} Q$ *and* $\vdash Q$ *then either* $\mathtt{Q}$ *is a terminal form, or* $\exists \mathtt{Q}'$ *s.t.* $\mathtt{Q} \longrightarrow \mathtt{Q}'$, $\mathtt{Q}' \overset{\Delta}{\hookrightarrow} Q'$ *and* $\vdash Q'$.

4. Scaling to TypeScript

TypeScript (TS) extends JavaScript (JS) with modules, classes and a lightweight type system that enables IDE support for auto-completion and refactoring. TS deliberately eschews soundness [3] for backwards compatibility with existing JS code. In this section, we show how to use refinement types to *regain safety*, by presenting the highlights of Refined TypeScript (and our tool `rsc`), that scales the core calculus from §3 up to TS by extending the support for *types* (§4.1), *reflection* (§4.2), *interface hierarchies* (§4.3), and *imperative* programming (§4.4).

4.1 Types

First, we discuss how `rsc` handles core TS features like object literals, interfaces and primitive types.

Object Literal Types TS supports object literals, *i.e.* anonymous objects with field and method bindings. `rsc` types object members in the same way as class members: method signatures need to be explicitly provided, while field types and mutability modifiers are inferred based on use, *e.g.* in:

```
var point = { x: 1, y: 2 }; point.x = 2;
```

the field `x` is updated and hence, `rsc` infers that `x` is mutable.

Interfaces TS supports named object types in the form of interfaces, and treats them in the same way as their *structurally* equivalent class types. For example, the interface

```
interface PointI { number x, y; }
```

is equivalent to a class `PointC` defined as

```
class PointC { number x, y; }
```

In `rsc` these two types are *not* equivalent, as objects of type `PointI` do not necessarily have `PointC` as their constructor:

```
var pI = { x: 1, y: 2 }, pC = new PointC(1,2);
pI instanceof PointC;   // returns false
pC instanceof PointC;   // returns true
```

However, $\vdash$ `PointC` $\leq$ `PointI` *i.e.* instances of the *class* may be used to implement the *interface*.

Primitive Types We extend `rsc`'s support for primitive types to model the corresponding types in TS. TS has `undefined` and `null` types to represent the eponymous values, and treats these types as the "bottom" of the type hierarchy, effectively allowing those values to inhabit *every* type via subtyping. `rsc` also includes these two types, but *does not* treat them as "bottom" types. Instead `rsc` handles them as distinct primitive types inhabited solely by `undefined` and `null`, respectively, that can take part in unions. Consequently, the following code is accepted by TS but *rejected* by `rsc`:

```
var x = undefined; var y = x + 1;
```

Unsound Features in TS include (1) treating `undefined` and `null` as inhabitants of all types, (2) co-variant input subtyping, (3) allowing unchecked overloads, and (4) allowing a special "dynamic" `any` type to be ascribed to any term. `rsc` ensures soundness by (1) performing checks when non-null (non-undefined) types are required (*e.g.* during field accesses), (2) using the correct variance for functions and constructors, (3) checking overloads via two-phase typing (§2.1.2), and, (4) *eliminating* the `any` type.

Many uses of `any` (indeed, *all* uses, in our benchmarks §5) can be replaced with a combination of union or intersection types or downcasting, all of which are soundly checked via path-sensitive refinements. In future work, we wish to support the full language, namely allow *dynamically* checked uses of `any` by incorporating orthogonal dynamic techniques

from the contracts literature. We envisage a *dynamic cast* operation $\texttt{cast}_T :: (\texttt{x:any}) \Rightarrow \{\nu{:}T \mid \nu = \texttt{x}\}$. It is straightforward to implement $\texttt{cast}_T$ for first-order types T as a dynamic check that traverses the value, testing that its components satisfy the refinements [30]. Wrapper-based techniques from the contracts/gradual typing literature should then let us support higher-order types.

4.2 Reflection

JS programs make extensive use of reflection via "dynamic" type tests. `rsc` statically accounts for these by encoding type-tags in refinements. The following tests if `x` is a `number` before performing an arithmetic operation on it:

```
var r = 1;
if (typeof x === "number") r += x;
```

We account for this idiomatic use of **`typeof`** by *statically* tracking the "type" tag of values inside refinements using uninterpreted functions (akin to the size of arrays). Thus, values `v` of type `boolean`, `number`, `string`, *etc.* are refined with the predicate `ttag(v)= "boolean"`, `ttag(v)= "number"`, `ttag(v)= "string"`, *etc.*, respectively. Furthermore, **`typeof`** has type

```
typeof : <A>(z:A) ⇒ {v:string | v = ttag(z)}
```

so the output type of **`typeof`** `x` and the path-sensitive guard under which the assignment `r = x + 1` occurs, ensures that at the assignment `x` can be statically proven to be a `number`. The above technique coupled with two-phase typing (§2.1.2) allows `rsc` to statically verify reflective, value-overloaded functions that are ubiquitous in TS [39].

4.3 Interface Hierarchies

JS programs frequently build up object hierarchies that represent *unions* of different kinds of values, and then use value tests to determine which kind of value is being operated on. In TS this is encoded by building up a hierarchy of interfaces, and then performing *downcasts* based on *value* tests[2].

Implementing Hierarchies with Bit-vectors Figure 6 describes a slice of the hierarchy of types used by the TypeScript compiler (`tsc`) v1.0.1.0. `tsc` uses bit-vector valued flags to encode membership in a particular interface type, *i.e.* discriminate between the different entities. (*Older* versions of `tsc` used a class-based approach, where inclusion could be tested via `instanceof` tests.) For example, the enumeration `TypeFlags` above maps semantic entities to bit-vector values used as masks that determine inclusion in a subinterface of `Type`. Suppose `t` of type `Type`. The invariant here is that if `t.flags` masked with `0x00000800` is non-zero, then `t` can be safely treated as an `InterfaceType` object, or an `ObjectType` object, since the relevant flag emerges from the bit-wise disjunction of the `Interface` flag with some other flags.

[2] `rsc` handles other type tests, *e.g.* `instanceof`, via an extension of the technique used for `typeof` tests; we omit a discussion for space.

```
interface Type {
  immutable flags: TypeFlags;
  id: number;
  symbol?: Symbol;
  ...
}

interface ObjectType extends Type { ... }

interface InterfaceType extends ObjectType {
  baseTypes: ObjectType[];
  declaredProperties: Symbol[];
  ...
}

enum TypeFlags {
  Any       = 0x00000001, String    = 0x00000002,
  Number    = 0x00000004, Class     = 0x00000400,
  Interface = 0x00000800, Reference = 0x00001000,
  Object    = Class | Interface | Reference
  ...
}
```

Figure 6: Type hierarchies in the `tsc` compiler

Specifying Hierarchies with Refinements `rsc` allows developers to *create* and *use* `Type` objects with the above invariant by specifying a predicate `typeInv` [3]:

```
isMask<v,m,t> = mask(v,m) ⇒ impl(this,t)
typeInv<v> = isMask<v, 0x00000001, AnyType>
           ∧ isMask<v, 0x00000002, StringType>
           ∧ isMask<v, 0x00003C00, ObjectType>
```

and then refining `TypeFlags` with the predicate

```
type TypeFlags = {v:TypeFlags | typeInv<v>}
```

Intuitively, the refined type says that when `v` (that is the `flags` field) is a bit-vector with the first position set to `1` the corresponding object satisfies the `AnyType` interface, etc.

Verifying Downcasts `rsc` *verifies* the code that uses ad hoc hierarchies such as the above by proving the TS *downcast* operations (that allow objects to be used at particular instances) safe. For example, consider the following code that *tests* if `t` implements the `ObjectType` interface before performing a downcast from type `Type` to `ObjectType` that permits the access of the latter's fields:

```
function getPropertiesOfType(t: Type): Symbol[] {
  if (t.flags & TypeFlags.Object) {
    var o = <ObjectType>t;
    [...]
  }
}
```

`tsc` erases casts, thereby missing possible runtime errors. The same code *without* the if-test, or with a *wrong* test would pass the TypeScript type checker. `rsc`, on the other hand, checks casts *statically*. In particular, `<ObjectType>t` is treated as a call to a function with signature

[3] Modern SMT solvers easily handle formulas over bit-vectors, including operations that shift, mask bit-vectors, and compare them for equality.

```
(x:{A|impl(x,ObjectType)})⇒{v:ObjectType|v=x}
```

The if-test ensures that the *immutable* field `t.flags` masked with `0x00003C00` is non-zero, satisfying the third line in the type definition of `typeInv`, which in turn implies that `t` in fact implements the `ObjectType` interface.

4.4 Imperative Features

Immutability Guarantees Our system uses ideas from Immutability Generic Java [42] (IGJ) to provide statically checked immutability guarantees. In IGJ a type reference is of the form `C<M,`$\overline{\texttt{T}}$`>`, where *immutability* argument `M` works as proxy for the immutability modifiers of the contained fields (unless overridden). It can be one of: `Immutable` (or `IM`), when neither this reference nor any other reference can mutate the referenced object; `Mutable` (or `MU`), when this and potentially other references can mutate the object; and `ReadOnly` (or `RO`), when this reference cannot mutate the object, but some other reference may. Similar reasoning holds for method annotations. IGJ provides *deep immutability*, since a class's immutability parameter is (by default) reused for its fields; however, this is not a firm restriction imposed by refinement type checking.

Arrays TS's definitions file provides a detailed specification for the `Array` interface. We extend this definition to account for the mutating nature of certain array operations:

```
interface Array<M extends ReadOnly,T> {
  @Mutable   pop(): T;
  @Mutable   push(x:T): number;
  @Immutable get length(): {nat|v=len(this)}
  @ReadOnly  get length(): nat;
  [...]
}
```

Mutating operations (`push`, `pop`, field updates) are only allowed on mutable arrays, and the type of `a.length` encodes the exact length of an immutable array `a`, and just a natural number otherwise. For example, assume the following code:

```
for(var i = 0; i < a.length; i++) {
  var x = a[i];
  [...]
}
```

To prove the access `a[i]` safe we need to establish `0` $\leq$ `i` and `i < a.length`. To guarantee that the length of `a` is constant, `a` needs to be immutable, so `rsc` will flag an error unless `a: Array<IM,T>`.

Object Initialization Our formal core (§3) treats constructor bodies in a very limiting way: object construction is merely an assignment of the constructor arguments to the fields of the newly created object. In `rsc` we relax this restriction in two ways: (a) We allow class and field invariants to be violated *within* the body of the constructor, but checked for at the exit. (b) We permit the common idiom of certain fields being initialized *outside* the constructor, via an additional mutability variant that encodes reference *uniqueness*. In both cases, we still restrict constructor code so that it does not *leak* references of the constructed object (`this`) or *read* any of its fields, as they might still be in an uninitialized state.

(a) Internal Initialization: Constructors Type invariants do not hold while the object is being "cooked" within the constructor. To safely account for this idiom, `rsc` defers the checking of class invariants (*i.e.* the types of fields) by replacing: (a) occurrences of `this.`$\texttt{f}_i$`=`$\texttt{e}_i$, with $\texttt{\_f}_i = \texttt{e}_i$, where $\texttt{\_f}_i$ are *local* variables, and (b) all return points with a call `ctor_init(`$\overline{\texttt{\_f}_i}$`)`, where the signature for `ctor_init` is: $(\overline{\texttt{x}}{:}\overline{T}) \Rightarrow \texttt{void}$. Thus, `rsc` treats field initialization in a field- and path-sensitive way (through the usual SSA conversion), and establishes the class invariants via a single atomic step at the constructor's exit (return).

(b) External Initialization: Unique References Sometimes we want to allow immutable fields to be initialized outside the constructor. Consider the code (adapted from `tsc`):

```
function createType(flags:TypeFlags):Type<IM> {
  var r: Type<UQ> = new Type(checker, flags);
  r.id = typeCount++;
  return r;
}
```

Field `id` is expected to be *immutable*. However, its initialization happens after `Type`'s constructor has returned. Fixing the type of `r` to `Type<IM>` right after construction would disallow the assignment of the `id` field on the following line. So, instead, we introduce `Unique` (or `UQ`), a new mutability type that denotes that the current reference is the *only* reference to a specific object, and hence, allows mutations to its fields. When `createType` returns, we can finally fix the mutability parameter of `r` to `IM`. We could also return `Type<UQ>`, extending the *cooking* phase of the current object and allowing further initialization by the caller. `UQ` references obey stricter rules to avoid leaking of unique references:

- they cannot be reassigned,
- they generally cannot be referenced, unless this occurs at a context that guarantees that no aliases will be produced, *e.g.* the context of `e1` in `e1.f = e2`, or the context of a returned expression, and
- they cannot be cast to types of a different mutability (*e.g.* `<C<IM>>x`), as this would allow the same reference to be subsequently aliased.

§6 discusses more expressive initialization approaches.

5. Evaluation

To evaluate `rsc`, we have used it to analyze a suite of JS and TS programs, to answer two questions: (1) What kinds of properties can be statically verified for real-world code? (2) What kinds of annotations or overhead does verification impose? Next, we describe the properties, benchmarks and discuss the results.

Safety Properties We verify with `rsc` the following:

- ***Property Accesses*** `rsc` verifies that each field (`x.f`) or method lookup (`x.m(...)`) succeeds. Recall that **`undefined`** and **`null`** are not considered to inhabit the types to which the fields or methods belong.
- ***Array Bounds*** `rsc` verifies that each array read (`x[i]`) or write (`x[i] = e`) occurs within the bounds of the array (`x`).
- ***Overloads*** `rsc` verifies that functions with overloaded (*i.e.* intersection) types correctly implement the intersections in a path-sensitive manner as described in (§2.1.2).
- ***Downcasts*** `rsc` verifies that at each TS (down)cast of the form `<T>e`, the expression `e` is indeed an instance of `T`. This requires tracking program-specific invariants, *e.g.* bit-vector invariants that encode hierarchies (§4.3).

5.1 Benchmarks

We ported a number of existing JS or TS programs to `rsc`. We selected benchmarks that make heavy use of language constructs relevant to the safety properties described above. These include parts of the Octane test suite, developed by Google as a JavaScript performance benchmark [12] and already ported to TS by Rastogi *et al.* [27], the TS compiler [22], and the D3 [4] and Transducers [7] libraries:

- `navier-stokes`, which simulates two-dimensional fluid motion over time; `richards`, which simulates a process scheduler with several types of processes passing information packets; `splay`, which implements the *splay tree* data structure; and `raytrace`, which implements a ray-tracer that renders scenes involving multiple lights and objects; all from the Octane suite,
- `transducers`: a library that implements composable data transformations, a JavaScript port of Hickey's Clojure library, which is extremely dynamic in that some functions have 12 (value-based) overloads,
- `d3-arrays`: the array manipulating routines from the D3 [4] library, which makes heavy use of higher-order functions as well as value-based overloading,
- `tsc-checker`, which includes parts of the TS compiler (v1.0.1.0), abbreviated as `tsc`. We check 15 functions from `compiler/core.ts` and 14 functions from `compiler/checker.ts` (for which we needed to import 779 lines of type definitions from `compiler/types.ts`). These code segments were selected among tens of thousands of lines of code comprising the compiler codebase, because they exemplified interesting properties, like the bit-vector based type hierarchies explained in §4.3.

Results Figure 7 quantitatively summarizes the results of our evaluation. Overall, we had to add about 1 line of annotation per 5 lines of code (529 for 2522 LOC). The vast majority (334/529 or 63%) of the annotations are *trivial*, *i.e.* are TS-like types of the form `(x:nat)⇒ nat`; 20% (104/529) are trivial but have *mutability* information, and only 17% (91/529) mention refinements, *i.e.* are definitions like **`type`** `nat = {v:number|0≤v}` or dependent signatures like `(a:T[],n:idx<a>)⇒T`. These numbers show `rsc` has annotation overhead comparable to TS, as in 83% of the cases the annotations are either identical to TS annotations or to TS annotations with some mutability modifiers. Of course, in the remaining 17% of the cases, the signatures are more complex than the (non-refined) TS version.

Benchmark	LOC	T	M	R	Time (s)
`navier-stokes`	366	3	18	39	473
`splay`	206	18	2	0	6
`richards`	304	61	5	17	7
`raytrace`	576	68	14	2	15
`transducers`	588	138	13	11	12
`d3-arrays`	189	36	4	10	37
`tsc-checker`	293	10	48	12	62
TOTAL	2522	334	104	91	

Figure 7: LOC is the number of non-comment lines of source (computed via `cloc` v1.62). The number of RSC specifications given as JML style comments is partitioned into **T** trivial annotations *i.e.* TypeScript type signatures, **M** mutability annotations, and **R** refinement annotations, *i.e.* those which actually mention invariants. **Time** is the number of seconds taken to analyze each file.

Code Changes We had to modify the source in various small (but important) ways to facilitate verification. The total number of changes is summarized in Figure 8. The *trivial* changes include the addition of type annotations (accounted for above) and simple transformations to work around the current limitations of our front-end, *e.g.* converting `x++` to `x=x+1`. The *important* classes of changes are the following:

- ***Control-Flow:*** Some programs had to be restructured to work around `rsc`'s currently limited support for certain control flow structures (*e.g.* `break`). We also modified some loops to use explicit termination conditions.
- ***Classes and Constructors:*** As `rsc` does not yet support *default* constructor arguments, we changed relevant **`new`** calls in Octane to supply them explicitly, and refactored `navier-stokes` to use traditional OO style classes and constructors instead of JS records with function fields.
- ***Non-null Checks:*** In `splay` we added 5 explicit non-null checks for mutable objects as proving those required precise heap analysis that is outside `rsc`'s scope.
- ***Ghost Functions:*** `navier-stokes` has more than a hundred (static) array access sites, most of which compute indices via non-linear arithmetic (*i.e.* via computed indices of the form `arr[r*s + c]`); SMT support for non-linear integer arithmetic is brittle (and accounts for the anomalous time for `navier-stokes`). We factored axioms about non-linear arithmetic into *ghost functions* whose types were proven once via non-linear SMT queries, and

Benchmark	LOC	ImpDiff	AllDiff
navier-stokes	366	79	160
splay	206	58	64
richards	304	52	108
raytrace	576	93	145
transducers	588	170	418
d3-arrays	189	8	110
tsc-checker	293	9	47
TOTAL	2522	469	1052

Figure 8: LOC: number of non-comment lines of source (computed via `cloc` v1.62). The *number of lines changed* is counted as either **ImpDiff**: *important* changes, such as restructuring the original JS code to account for limited support for control flow constructs, replacing records with classes and constructors, and adding ghost functions; or **AllDiff**: the above plus *trivial* changes due to the addition of plain or refined type annotations (Figure 7), and simple edits to work around current limitations of our front-end.

which were then explicitly called at use sites to instantiate the axioms (thereby bypassing non-linear analysis). An example of such a function is:

```
/*@ mulThm :: (a:nat, b:{number | b ≥ 2})
           ⇒ {boolean | a + a ≤ a * b} */
```

which, when *instantiated* via a call `mulThm(x, y)` establishes the fact that (at the call-site), `x + x` $\leq$ `x * y`. The reported performance assumes the use of ghost functions. In cases where they were not used RSC would time out.

5.2 Transducers (A Case Study)

We now delve deeper into one of our benchmarks: the Transducers library. At its heart this library is about reducing collections, aka performing folds. A Transformer is anything that implements three functions: `init` to begin computation, `step` to consume one element from an input collection, and `result` to perform any post-processing. One could imagine rewriting `reduce` from Figure 1 by building a Transformer where `init` returns `x`, `step` invokes `f`, and `result` is the identity[4]. The Transformers provided by the library are composable - their constructors take, as a final argument, another Transformer, and then all calls to the outer Transformer's functions invoke the corresponding one of the inner Transformer. This gives rise to the concept of a Transducer, a function of type `(Transformer)⇒Transformer` and this library's namesake.

The main reason this library interests us is because some of its functions are massively overloaded. Consider, for example, the `reduce` function it defines. As discussed above, `reduce` needs a Transformer and a collection. There are two opportunities for overloading here. First of all, the main ways that a Transformer is more general than a simple step

[4] For simplicity of discussion we will henceforth ignore `init` and initialization in general, as well as some other details.

```
/*@ ((B, A) ⇒ B,           , A[]   ) ⇒ B
    (Transformer<A,B>      , A[]   ) ⇒ B
    ((B, string) ⇒ B)      , string) ⇒ B
    (Transformer<string, B>, string) ⇒ B
    ...
*/
function reduce(xf, col) {
  xf = (typeof xf == "function") ? wrap(xf) : xf;
  if (isString(col)) return stringReduce(xf,col);
  if (isArray(col))  return arrayReduce(xf,col);
  [...]
}
```

Figure 9: Sample adapted from Transducers benchmark

function is that it can be stateful and that it defines the `result` post-processing step. Most of the time the user does not need these features, in which case the Transformer is just a wrapper around a step function. Thus for convenience, the user is allowed to pass in either a full-fledged Transformer or a step function which will automatically get wrapped into one. Secondly, the collection being reduced can be a stunning array of options: an array, a string (*i.e.* a collection of characters, which are themselves just strings), an arbitrary object (*i.e.*, in JS, a collection of key-value pairs), an iterator (an object that defines a `next` function that iterates through the collection), or an iterable (an object that defines an `iterator` function that returns an iterator). Each of these collections needs to be dispatched to a type-specific reduce function that knows how to iterate over that kind of collection. In each overload, the type of the collection must match the type of the Transformer or step function. Thus our `reduce` begins as shown in Figure 9. Considering all five possible types of collections and the option between a step function or a Transformer, `reduce` has ten distinct overloads!

5.3 Unhandled Cases

This section outlines and explains some pitfalls of RSC.

Complex Constructor Patterns Due to our limited internal initialization scheme, certain common constructor patterns are not supported by RSC. For example, the code below:

```
class A<M extends RO> {
  f: nat;
  constructor() { this.setF(1); }
  setF(x: number) { this.f = x; }
}
```

Currently, RSC does not allow method invocations on the object under construction in the constructor, as it cannot track the (value of the) updates happening in the method `setF`. Note that this case is supported by IGJ. Section (§6) includes approaches that could lift this restriction.

Recovering Unique References RSC cannot recover the `Unique` state for objects after they have been converted to `Mutable` (or other state), as it lacks a fine-grained alias tracking mechanism. Assume, for example the function `distinct` below taken from the TS compiler v1.0.1.0:

```
1  function distinct<T>(a: T[]): T[] {
2    var res: T[] = [];
3    for (var i = 0, n = a.length; i < n; i++) {
4      var current = a[i];
5      for (var j = 0; j < res.length; j++) {
6        if (res[j] === current)
7          break;
8      }
9      if (j === res.length)
10       res.push(current);
11   }
12   return res;
13 }
```

Array `res` is defined at line 2 so it is initially typed as `Array<UQ,T>`. At lines 5–8 it is iterated over, so to prove the access at line 6 safe, we need to treat `res` as an immutable array. However, at line 10 an element is pushed on `res`, which requires `res` to be mutable. Our system cannot handle the interleaving of these two kinds of operations that (in addition) appear in a tight loop (lines 3–11). However, §6 includes approaches that could allow support for such cases.

Annotations per Function Overload A weakness of RSC, that stems from the use of Two-Phase Typing [39] in handling intersection types, is cases where type checking requires annotations under a specific signature overload. Consider for example the following code, which is a variation of the `reduce` function presented in §2:

```
1  /*@ <A>  (a:A[]+,f:(A,A,idx<a>)⇒A) ⇒ A
2      <A,B>(a:A[] ,f:(B,A,idx<a>)⇒B,x:B) ⇒ B
3   */
4  function reduce(a, f, x) {
5    var r, s;
6    if (arguments.length === 3) {
7      r = x; s = 0;
8    }
9    else {
10     r = a[0]; s = 1;
11   }
12   for (var i = s; i < a.length; i++)
13     r = f(r, a[i], i);
14   return r;
15 }
```

Checking the function body for the second overload (line 2) is problematic: without an annotation on `r`, its type at the end of the conditional will be `B + (A + undefined)` (`r` collects values from `x` and `a[0]`, at lines 7 and 10), instead of the intended `B`. This causes an error when `r` is passed to function `f` at line 13, expected to have type `B`, which cannot be overcome even with refinement checking, since this code is no longer guarded by the check on the length of `arguments` (line 7). A solution would be for the user to annotate the type of `r` as `B` at its definition at line 5, but only for the specific (second) overload. The assignment at line 10 will be invalid, but this is acceptable since that branch is provably (by the refinement checking phase [39]) dead. This option, however, is currently not available.

6. Related Work

RSC is related to several distinct lines of work.

Types for Dynamic Languages Original approaches incorporate *flow analysis* in the type system, using mechanisms to track aliasing and flow-sensitive updates [1, 35]. Typed Racket's *occurrence* typing narrows the type of unions based on control dominating type tests, and its *latent predicates* lift the results of tests across higher-order functions [36]. DRuby [10] uses intersection types to *represent* summaries for overloaded functions. TeJaS [21] combines occurrence typing with flow analysis to analyze JS [21]. Unlike RSC none of the above reason about relationships *between* values of multiple program variables, which is needed to account for value-overloading and richer program safety properties.

Program Logics At the other extreme, one can encode types as formulas in a logic, and use SMT solvers for all the analysis (subtyping). DMinor explores this idea in a first-order functional language with type tests [2]. The idea can be scaled to higher-order languages by embedding (*nesting*) the typing relation inside the logic [6]. DJS combines nested refinements with alias types [31], a restricted separation logic, to account for aliasing and flow-sensitive heap updates to obtain a static type system for a large portion of JS [5]. DJS proved to be extremely difficult to use. First, the programmer had to spend a lot of effort on manual heap related annotations; a task that became especially cumbersome in the presence of higher-order functions. Second, nested refinements precluded the possibility of refinement inference, further increasing the burden on the user. In contrast, mutability modifiers have proven to be lightweight [42] and two-phase typing lets `rsc` use liquid refinement inference [28], yielding a system that is more practical for real-world programs. *Extended Static Checking* [9] uses Floyd-Hoare style first-order contracts (pre-, post-conditions and loop invariants) to generate verification conditions discharged by an SMT solver. Refinement types can be viewed as a generalization of Floyd-Hoare logics that uses types to compositionally account for polymorphic higher-order functions and containers that are ubiquitous in modern languages like TS.

X10 [25] is a language that extends an object-oriented type system with *constraints* on the immutable state of classes. Compared to X10, in RSC: (a) we make mutability parametric [42], and extend the refinement system accordingly, (b) we crucially obtain flow-sensitivity via SSA transformation, and path-sensitivity by incorporating branch conditions, (c) we account for reflection by encoding tags in refinements and two-phase typing [39], and (d) our design ensures that we can use liquid type inference [28] to automatically synthesize refinements.

Analyzing TypeScript Feldthaus *et al.* present a hybrid analysis to find discrepancies between TS interfaces [41] and their JS implementations [8], and Rastogi *et al.* extend TS with an efficient gradual type system that mitigates the unsoundness of TS's type system [27].

Object and Reference Immutability `rsc` builds on existing methods for statically enforcing immutability. In particular,

we build on Immutability Generic Java which encodes object and reference immutability using Java generics [42]. Subsequent work extends these ideas to allow (1) richer *ownership* patterns for creating immutable cyclic structures [43], (2) *unique* references, and ways to recover immutability after violating uniqueness, without requiring alias analysis [13].

Reference immutability has recently been combined with rely-guarantee logics (originally used to reason about thread interference), to allow refinement type reasoning. Gordon *et al.* [14] treat references to shared objects like threads in rely-guarantee logics, and so multiple aliases to an object are allowed only if the guarantee condition of each alias implies the rely condition for all other aliases. Their approach allows refinement types over mutable data, but resolving their proof obligations depends on theorem-proving, which hinders automation. Militão *et al.* present Rely-Guarantee Protocols [23] that can model complex aliasing interactions, and, compared to Gordon's work, allow temporary inconsistencies, can recover from shared state via ownership tracking, and resort to more lightweight proving mechanisms.

The above extensions are orthogonal to `rsc`; in the future, it would be interesting to see if they offer practical ways for accounting for (im)mutability in TS programs.

Object Initialization A key challenge in ensuring immutability is accounting for the construction phase where fields are *initialized*. We limit our attention to *lightweight* approaches *i.e.* those that do not require tracking aliases, capabilities or separation logic [11, 31]. Haack and Poll [17] describe a flexible initialization schema that uses secret tokens, known only to *stack-local* regions, to initialize all members of cyclic structures. Once initialization is complete the tokens are converted to global ones. Their analysis is able to infer the points where new tokens need to be introduced and committed. The *Masked Types* [26] approach tracks, within the type system, the set of fields that remain to be initialized. X10's *hard-hat* flow-analysis based approach to initialization [44] and *Freedom Before Commitment* [32] are the most permissive of the lightweight methods, allowing, unlike `rsc`, method dispatches or field accesses in constructors.

7. Conclusions and Future Work

We have presented RSC which brings SMT-based modular and extensible analysis to dynamic, imperative, class-based languages by harmoniously integrating several techniques. First, we restrict refinements to immutable variables and fields (cf. X10 [34]). Second, we make mutability parametric (cf. IGJ [42]) and recover path- and flow-sensitivity via SSA. Third, we account for reflection and value overloading via two-phase typing [39]. Finally, our design ensures that we can use liquid type inference [28] to automatically synthesize refinements. Consequently, we have shown how `rsc` can verify a variety of properties with a modest annotation overhead similar to TS. Finally, our experience points to several avenues for future work, including: (1) more permissive but lightweight techniques for object initialization [44], (2) automatic inference of trivial types via flow analysis [16], (3) verification of security properties, *e.g.* access-control policies in JS browser extensions [15].

Acknowledgments

We would like to thank our anonymous reviewers and our shepherd Cormac Flanagan for their feedback, and Alexander Bakst for his helpful comments on earlier drafts of this paper. This work was supported by NSF Grants CNS-1223850, CNS-0964702 and gifts from Microsoft Research.

References

[1] C. Anderson, P. Giannini, and S. Drossopoulou. Towards Type Inference for JavaScript. In *Proceedings of ECOOP*, 2005.

[2] G. M. Bierman, A. D. Gordon, C. Hriţcu, and D. Langworthy. Semantic Subtyping with an SMT Solver. In *Proceedings of ICFP*, 2010.

[3] G. M. Bierman, M. Abadi, and M. Torgersen. Understanding TypeScript. In *Proceedings of ECOOP*, 2014.

[4] M. Bostock. `http://d3js.org/`.

[5] R. Chugh, D. Herman, and R. Jhala. Dependent Types for JavaScript. In *Proceedings of OOPSLA*, 2012.

[6] R. Chugh, P. M. Rondon, and R. Jhala. Nested Refinements: A Logic for Duck Typing. In *Proceedings of POPL*, 2012.

[7] Cognitect Labs. `https://github.com/cognitect-labs/transducers-js`.

[8] A. Feldthaus and A. Møller. Checking Correctness of TypeScript Interfaces for JavaScript Libraries. In *Proceedings of OOPLSA*, 2014.

[9] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended Static Checking for Java. In *Proceedings of PLDI*, 2002.

[10] M. Furr, J.-h. D. An, J. S. Foster, and M. Hicks. Static Type Inference for Ruby. In *Proceedings of the Symposium on Applied Computing*, 2009.

[11] P. Gardner, S. Maffeis, and G. D. Smith. Towards a program logic for JavaScript. In *Proceedings of POPL*, 2012.

[12] Google Developers. `https://developers.google.com/octane/`.

[13] C. S. Gordon, M. J. Parkinson, J. Parsons, A. Bromfield, and J. Duffy. Uniqueness and Reference Immutability for Safe Parallelism. In *Proceedings of OOPSLA*, 2012.

[14] C. S. Gordon, M. D. Ernst, and D. Grossman. Rely-guarantee References for Refinement Types over Aliased Mutable Data. In *Proceedings of PLDI*, 2013.

[15] A. Guha, M. Fredrikson, B. Livshits, and N. Swamy. Verified Security for Browser Extensions. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2011.

[16] S. Guo and B. Hackett. Fast and Precise Hybrid Type Inference for JavaScript. In *Proceeding of PLDI*, 2012.

[17] C. Haack and E. Poll. Type-Based Object Immutability with Flexible Initialization. In *Proceedings of ECOOP*, 2009.

[18] A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: A Minimal Core Calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.*, May 2001.

[19] K. Knowles and C. Flanagan. Hybrid Type Checking. *ACM Trans. Program. Lang. Syst.*, Feb. 2010.

[20] K. Knowles and C. Flanagan. Compositional Reasoning and Decidable Checking for Dependent Contract Types. In *Proceedings of PLPV*, 2008.

[21] B. S. Lerner, J. G. Politz, A. Guha, and S. Krishnamurthi. TeJaS: Retrofitting Type Systems for JavaScript. In *Proceedings of DLS*, 2013.

[22] Microsoft Corporation. TypeScript v1.4. `http://www.typescriptlang.org/`.

[23] F. Militão, J. Aldrich, and L. Caires. Rely-Guarantee Protocols. In *Proceedings of ECOOP*, 2014.

[24] G. Nelson. Techniques for Program Verification. Technical Report CSL81-10, Xerox Palo Alto Research Center, 1981.

[25] N. Nystrom, V. Saraswat, J. Palsberg, and C. Grothoff. Constrained Types for Object-oriented Languages. In *Proceedings of OOPSLA*, 2008.

[26] X. Qi and A. C. Myers. Masked Types for Sound Object Initialization. In *Proceedings of POPL*, 2009.

[27] A. Rastogi, N. Swamy, C. Fournet, G. Bierman, and P. Vekris. Safe & Efficient Gradual Typing for TypeScript. In *Proceedings of POPL*, 2015.

[28] P. M. Rondon, M. Kawaguci, and R. Jhala. Liquid Types. In *Proceedings of PLDI*, 2008.

[29] J. Rushby, S. Owre, and N. Shankar. Subtypes for Specifications: Predicate Subtyping in PVS. *IEEE TSE*, 1998.

[30] E. L. Seidel, N. Vazou, and R. Jhala. Type Targeted Testing. In *Proceedings of ESOP*, 2015.

[31] F. Smith, D. Walker, and G. Morrisett. Alias Types. In *Proceedings of ESOP*, 2000.

[32] A. J. Summers and P. Mueller. Freedom Before Commitment: A Lightweight Type System for Object Initialisation. In *Proceedings of OOPSLA*, 2011.

[33] N. Swamy, J. Chen, C. Fournet, P.-Y. Strub, K. Bhargavan, and J. Yang. Secure Distributed Programming with Value-dependent Types. In *Proceedings of ICFP*, 2011.

[34] O. Tardieu, N. Nystrom, I. Peshansky, and V. Saraswat. Constrained Kinds. In *Proceedings of OOPSLA*, 2012.

[35] P. Thiemann. Towards a Type System for Analyzing JavaScript Programs. In *Proceedings of ESOP*, 2005.

[36] S. Tobin-Hochstadt and M. Felleisen. Logical Types for Untyped Languages. In *Proceedings of ICFP*, 2010.

[37] N. Vazou, E. L. Seidel, R. Jhala, D. Vytiniotis, and S. Peyton-Jones. Refinement Types for Haskell. In *Proceedings of ICFP*, 2014.

[38] P. Vekris, B. Cosman, and R. Jhala. Refinement Types for TypeScript (Extended version). `http://arxiv.org/abs/1604.02480`.

[39] P. Vekris, B. Cosman, and R. Jhala. Trust, but Verify: Two-Phase Typing for Dynamic Languages. In *Proceedings of ECOOP*, 2015.

[40] H. Xi and F. Pfenning. Dependent Types in Practical Programming. In *Proceedings of POPL*, 1999.

[41] B. Yankov. `http://definitelytyped.org`.

[42] Y. Zibin, A. Potanin, M. Ali, S. Artzi, A. Kiezun, and M. D. Ernst. Object and Reference Immutability Using Java Generics. In *Proceedings of ESEC/FSE*, 2007.

[43] Y. Zibin, A. Potanin, P. Li, M. Ali, and M. D. Ernst. Ownership and Immutability in Generic Java. In *Proceedings of OOPSLA*, 2010.

[44] Y. Zibin, D. Cunningham, I. Peshansky, and V. Saraswat. Object Initialization in X10. In *Proceedings of ECOOP*, 2012.