Refinement Types for TypeScript

Panagiotis Vekris
University of California, San Diego
pvekris@cs.ucsd.edu

Benjamin Cosman
University of California, San Diego
blcosman@cs.ucsd.edu

Ranjit Jhala
University of California, San Diego
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 core of RSC that delineates the interaction between refinement types and mutability. Next, we extend the core to account for the imperative and dynamic features of TypeScript. Finally, we evaluate RSC on a set of real world benchmarks, including parts of the Octane benchmarks, D3, Transducers, and the TypeScript compiler.

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 [27, 37]. The extensibility and modularity offered by refinement types have enabled their use in a variety of applications in *typed*, *functional* languages, like ML [26, 37], Haskell [35], and $F^\sharp$ [31]. 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 [39] 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 that is essential for analyzing real world applications. Second, RSC accounts for *dynamic typing* by using a recently proposed technique called two-phase typing [36], 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 [26] framework to render refinement typing practical on real world programs. Concretely, we make the following contributions:

```
function reduce(a, f, x) {
  var res = x, i;
  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 develop a core calculus that formalizes the interaction of mutability and refinements via declarative refinement type checking that we prove sound (§ 3).
- 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 crucially permits locally flow-sensitive reasoning via SSA translation (§ 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. 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 [22]. 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}.

Summaries Function Types $(x_1 : T_1, \ldots, x_n : T_n) \Rightarrow T$, 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, $(x{:}\mathtt{nat}) \Rightarrow \{\nu{:}\mathtt{nat} \mid x < \nu\}$ 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 $T_{\mathtt{reduce}}$:

```
<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 down-casts — 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    : (a:T[],i:idx<a>) ⇒ T
set    : (a:T[],i:idx<a>,e:T) ⇒ void
length : (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(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 $\Gamma_{\mathtt{head}}$ defined as $\mathtt{arr} : \{\nu{:}\mathtt{T[]} \mid 0 < \mathtt{len}(\nu)\}$ yielding the subtyping obligation:

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

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

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

The VC is proved *valid* by an SMT solver [22], 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 $\Gamma_{\mathtt{head0}}$ defined as: $\mathtt{a} : \mathtt{number[]},\ 0 < \mathtt{len}(\mathtt{a})$ *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:

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

yielding the valid VC: $0 < \mathtt{len}(\mathtt{a}) \Rightarrow \nu = \mathtt{a} \Rightarrow 0 < \mathtt{len}(\nu)$.

Polymorphic, Higher Order Functions Next, let us *assume* that reduce has the type $T_{\mathtt{reduce}}$ 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:

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

Let us reassure ourselves that this instantiation is valid, by checking that step and 0 satisfy the instantiated type. If we substitute (2) into $T_{\mathtt{reduce}}$ 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 libraries, more than 25% of the functions are value-overloaded [36].

Intersection Types Refinements let us statically *verify* value-based overloading [36]. 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,B>(a:A[], f:(A, A, idx<a>)⇒A)⇒A       // 1
∧ <A,B>(a:A[], f:(B, A, idx<a>)⇒B, x:B)⇒B // 2
```

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 : (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 `assert`s, 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 [26] 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:

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

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 $\kappa_A[\,]$, κ_B and $(\kappa_B, \kappa_A, \texttt{idx}\langle\texttt{a}\rangle) \Rightarrow \kappa_B$. Checking `a` and `0` yields the subtyping constraints:

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

where $\Gamma \doteq \texttt{a}{:}\texttt{number}[\,],\ 0 < \texttt{len}(\texttt{a})$ 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:

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

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 [26] to obtain the solution $\kappa_A \mapsto \texttt{true}$ and $\kappa_B \mapsto 0 \leq \nu < \texttt{len}(\texttt{a})$ 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 (§ 3.1)

```
function reduce(a, f, x) {
  var r0 = x, i0 = 0;
  while [i2,r2 = ϕ((i0, r0), (i1, 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 blocks where `i0`,`r0` are defined:

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

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:

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

Finally, the above constraints are solved to:

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

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).

2.2.3 Mutation

In the imperative, object-oriented setting (common to dynamic scripting languages), we must account for *class* and *object* invariants and their preservation in the presence of field mutation. For

```
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

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 bounds 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` co-ordinate `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: an imperative, mutable, object-oriented subset of Refined TypeScript, that closely follows the design of CFJ [23], (the language used to formalize X10), which in turn is based on Featherweight Java [17]. To ease refinement reasoning, we translate FRSC to a functional, yet still mutable, intermediate language IRSC. We then formalize our static semantics in terms of IRSC.

3.1 Formal Language

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 $e.m\,(\overline{e})$, object construction **new** $C\,(\overline{e})$, and cast operations e **as** T. Statements s include variable declarations, field updates, assignments, conditionals, concatenations and empty statements. Method declarations include a type signature, specifying input and output types, and a body, *i.e.* a statement immediately followed by a returned expression. In class definitions, unlike CFJ, we distinguish between mutable and immutable members, using $\overline{\Diamond f\colon T}$ and $\overline{g\colon S}$, respectively. We do not formalize method overloading or overriding, so method names are distinct from the ones defined in parent classes. As in CFJ, each class and method definition is associated with an invariant p. Finally, programs are sequences of class declarations followed by a statement.

$$
\begin{array}{rcl}
e & ::= & x \mid c \mid \textbf{this} \mid e.f \mid e.m\,(\overline{e}) \mid \textbf{new}\ C\,(\overline{e}) \mid e\ \textbf{as}\ T \\
s & ::= & \textbf{var}\ x = e \mid e.f = e \mid x = e \mid \\
  &   & \textbf{if}\,(e)\ \ \textbf{then}\ s\ \textbf{else}\ s \mid s;\, s \mid \textbf{skip} \\
M & ::= & \textbf{def}\ m\,\left(\overline{x\colon T}\right)\{p\} : T {=} \{s;\, \textbf{return}\ e\} \\
L & ::= & \textbf{class}\ C\,\left(\overline{\Diamond f\colon T};\, \overline{g\colon S}\right)\{p\}\ \textbf{extends}\ R\ \left\{\overline{M}\right\} \\
P & ::= & \overline{L};\, s
\end{array}
$$

Intermediate Language (IRSC) This language retains the expressivity of FRSC, *but* has no variable assignments. Statements are replaced by let-bindings and new variables are introduced for each variable being reassigned in the respective FRSC code. The rest of the language features are only slightly adjusted, to obtain the following syntax:

$$
\begin{array}{rcl}
u, w & ::= & x \mid c \mid \textbf{this} \mid u.f \mid u.m\,(\overline{u}) \mid \textbf{new}\ C\,(\overline{u}) \mid \\
  &   & u\ \textbf{as}\ T \mid \textbf{let}\ x = u\ \textbf{in}\ u \mid u.f = u \mid \\
  &   & \textbf{if}\,(u)\ \ \textbf{then}\ u\ \textbf{else}\ u \\
\mathcal{M} & ::= & \textbf{def}\ m\,\left(\overline{x\colon T}\right)\{p\} : T = u \\
\mathcal{L} & ::= & \textbf{class}\ C\,\left(\overline{\Diamond f\colon T};\, \overline{g\colon S}\right)\{p\}\ \textbf{extends}\ R\ \{\overline{\mathcal{M}}\} \\
\mathcal{P} & ::= & \overline{\mathcal{L}};\, u
\end{array}
$$

SSA Transformation We translate FRSC to IRSC via a Static Single Assignment transformation ($\Vdash$), described in Figure 3. This process uses a *translation state* Δ, to map FRSC to IRSC variables. The translation of expressions e to u is routine: as expected, S-VAR maps the source level x to the current binding of x in Δ.

SSA Transformation $\boxed{\Delta \Vdash e \hookrightarrow u}$ $\boxed{\Delta \Vdash s \hookrightarrow \mathcal{E}[\,]\,/\,\Delta'}$ $\boxed{\cdot \Vdash M \hookrightarrow \mathcal{M}}$

$$[\text{S-VAR}]\quad \Delta \Vdash x \hookrightarrow \Delta(x)$$

$$[\text{S-VARDECL}]\quad \frac{\Delta \Vdash e \hookrightarrow u \qquad \Delta' = \Delta, x \mapsto x_0 \qquad (x_0 \text{ fresh})}{\Delta \Vdash \textbf{var}\ x = e \hookrightarrow \textbf{let}\ x_0 = u\ \textbf{in}\ [\,]\,/\,\Delta'}$$

$$[\text{S-DOTASGN}]\quad \frac{\Delta \Vdash e \hookrightarrow u \qquad \Delta \Vdash e' \hookrightarrow u'}{\Delta \Vdash e.f = e' \hookrightarrow \textbf{let}\ \_ = u.f = u'\ \textbf{in}\ [\,]\,/\,\Delta}$$

$$[\text{S-ITE}]\quad \frac{\begin{array}{c}\Delta \Vdash e \hookrightarrow u \qquad \Delta \Vdash s_1 \hookrightarrow \mathcal{E}_1[\,]\,/\,\Delta_1 \qquad \Delta \Vdash s_2 \hookrightarrow \mathcal{E}_2[\,]\,/\,\Delta_2\\ (\overline{\phi}, \overline{x}) = \mathsf{splitOnCommon}(\Delta_1, \Delta_2) \qquad \overline{\phi_1} = \Delta_1(\overline{\phi}) \qquad \overline{\phi_2} = \Delta_2(\overline{\phi}) \qquad \Delta' = \Delta, \overline{\phi} \mapsto \overline{\phi}' \qquad (\overline{\phi}' \text{ fresh})\end{array}}{\Delta \Vdash \textbf{if}\,(e)\ \textbf{then}\ s_1\ \textbf{else}\ s_2 \hookrightarrow \textbf{let}\ \overline{\phi}' = \left(\textbf{if}\,(u)\ \textbf{then}\ \mathcal{E}_1[\overline{\phi_1}]\ \textbf{else}\ \mathcal{E}_2[\overline{\phi_2}]\right)\ \textbf{in}\ [\,]\,/\,\Delta'}$$

$$[\text{S-ASGN}]\quad \frac{\Delta \Vdash e \hookrightarrow u \qquad x_i = \Delta(x) \qquad \Delta' = \Delta, x \mapsto x_{i+1} \qquad (x_{i+1} \text{ fresh})}{\Delta \Vdash x = e \hookrightarrow \textbf{let}\ x_{i+1} = u\ \textbf{in}\ [\,]\,/\,\Delta'}$$

$$[\text{S-SEQ}]\quad \frac{\Delta \Vdash s_1 \hookrightarrow \mathcal{E}_1[\,]\,/\,\Delta_1 \qquad \Delta_2 \Vdash s_2 \hookrightarrow \mathcal{E}_2[\,]\,/\,\Delta_2}{\Delta \Vdash s_1; s_2 \hookrightarrow \mathcal{E}_1[\mathcal{E}_2[\,]]\,/\,\Delta_2}$$

$$[\text{S-SKIP}]\quad \Delta \Vdash \textbf{skip} \hookrightarrow [\,]\,/\,\Delta$$

$$[\text{S-METH}]\quad \frac{\Delta_0 = \overline{x} \mapsto \overline{x_0} \qquad (\overline{x_0} \text{ fresh}) \qquad \Delta_0 \Vdash s \hookrightarrow \mathcal{E}[\,]\,/\,\Delta \qquad \Delta \Vdash e \hookrightarrow u}{\cdot \Vdash \textbf{def}\ m\,(\overline{x}\!:\overline{T})\ \{p\} : T = \{s;\ \textbf{return}\ e\} \hookrightarrow \textbf{def}\ m\,(\overline{x}\!:\overline{T})\ \{p\} : T = \mathcal{E}[u]}$$

Figure 3: Selected SSA Transformation Rules

The translating judgment of statements s has the form: $\Delta \Vdash s \hookrightarrow \mathcal{E}[\,]\,/\,\Delta'$. The *SSA context* $\mathcal{E}[\,]$ is an expression containing a *hole* $[\ ]$ in its body (*e.g.* $\textbf{let}\ x = u\ \textbf{in}\ [\,]$). The hole is to be filled in by the translation of the subsequent statements. Output environment Δ' reflects the potential introduction of new SSA variables. Rule S-VARDECL introduces such fresh variable x_0 and assigns it to the binding for the newly declared variable x. Field assignments do not affect Δ (rule S-DOTASGN).

The most interesting case is conditionals (rule S-ITE). Each branch is translated separately producing an SSA context and a translation state. Meta-function $\mathsf{splitOnCommon}$ is used to partition variables into those whose version differs at the end of the two branches ($\overline{\phi}$) and those for which it remains the same ($\overline{x}$). The body of each branch returns a tuple containing the latest versions of the $\overline{\phi}$ variables for that branch. Fresh variables $\overline{\phi}'$ are introduced as the join of the updated variables of each branch, updating the returned SSA environment Δ' appropriately.

Assignment statements increase the versioning of the SSA variable corresponding 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 body with the translation of the return expression (rule S-METH).

3.2 Static Semantics

Types Our type language (shown below) resembles that of existing refinement type systems [18, 23, 26]. 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 [19].

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 access $t.f$ and uninterpreted function applications $f\,(\overline{t})$.

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

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; we defer them to the supplemental material. The key extension in our setting is we partition $C\ \mathsf{has}\ I$ (that encodes inclusion of an element I in a class C) into two cases: $C\ \mathsf{hasMut}\ I$ and $C\ \mathsf{hasImm}\ I$, 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 are (i) 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 (details can be found in the supplemental material).

Besides well-formedness, our system's main judgment forms are those for subtyping and refinement typing [18].

Subtyping is defined by the judgment $\Gamma \vdash S \leq T$. The rules are standard among refinement type systems with existential 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 that accounts 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 have been simplified to non-existential bindings when they entered the environment. The full set of rules is included in the supplemental material.

Refinement Typing Rules Most rules of our typing judgement $\Gamma \vdash u : T$, are in Figure 4. We discuss the novel ones.

[T-FIELD-I] Similarly to CFJ, we perform *self* strengthening [19], defined with the aid of the $\Cap$ operator:

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

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

Typing rules $\boxed{\Gamma \vdash u : T}$

$$[\text{T-VAR}]\ \frac{\Gamma(x) = T}{\Gamma \vdash x : \mathsf{self}(T, x)} \qquad [\text{T-CST}]\ \Gamma \vdash c : \mathsf{ty}(c) \qquad [\text{T-FIELD-I}]\ \frac{\Gamma \vdash u : T \qquad \Gamma, z{:}T \vdash z\ \mathsf{hasImm}\ f_i{:}T_i \qquad (z\ \text{fresh})}{\Gamma \vdash u.f_i : \exists z{:}T.\ \mathsf{self}(T_i, z.f_i)}$$

$$[\text{T-FIELD-M}]\ \frac{\begin{array}{c}\Gamma \vdash u : T \\ \Gamma, z{:}T \vdash z\ \mathsf{hasMut}\ g_i : T_i \\ (z\ \text{fresh})\end{array}}{\Gamma \vdash u.g_i : \exists z{:}T.\, T_i} \qquad [\text{T-INV}]\ \frac{\begin{array}{c}\Gamma \vdash u : T, \overline{u} : \overline{T} \\ \Gamma, z{:}T, \overline{z}{:}\overline{T} \vdash z\ \mathsf{has}\ \left(\mathbf{def}\ m\left(\overline{z}{:}\overline{R}\right)\{p\} : S = u'\right),\ \overline{T} \leq \overline{R},\ p \\ (z,\ \overline{z}\ \text{fresh})\end{array}}{\Gamma \vdash u.m(\overline{u}) : \exists z{:}T.\, \exists \overline{z}{:}\overline{T}.\, S}$$

$$[\text{T-NEW}]\ \frac{\begin{array}{c}\Gamma \vdash \overline{u} : \left(\overline{T}_I, \overline{T}_M\right) \qquad \vdash \mathsf{class}(C) \qquad \Gamma, z{:}C \vdash \mathsf{fields}(z) = \Diamond\overline{f}{:}\overline{R},\ \overline{g}{:}\overline{U} \\ \Gamma, z{:}C, \overline{z}_I : \mathsf{self}\left(\overline{T}_I, z.\overline{f}\right) \vdash \overline{T}_I \leq \overline{R},\ \overline{T}_M \leq \overline{U},\ \mathit{inv}(C, z) \qquad (z, \overline{z}\ \text{fresh})\end{array}}{\Gamma \vdash \mathbf{new}\ C(\overline{u}) : \exists \overline{z}_I{:}\overline{T}_I.\{\nu{:}C \mid \nu.\overline{f} = \overline{z}_I \wedge \mathit{inv}(C, \nu)\}} \qquad [\text{T-CAST}]\ \frac{\begin{array}{c}\Gamma \vdash u : S \qquad \Gamma \vdash T \\ \Gamma \vdash S \lesssim T\end{array}}{\Gamma \vdash u\ \mathbf{as}\ T : T}$$

$$[\text{T-LET}]\ \frac{\Gamma \vdash u_1 : T_1 \qquad \Gamma, x{:}T_1 \vdash u_2 : T_2}{\Gamma \vdash \mathbf{let}\ x = u_1\ \mathbf{in}\ u_2 : \exists x{:}T_1.\, T_2} \qquad [\text{T-ASGN}]\ \frac{\begin{array}{c}\Gamma \vdash u_1 : T_1, u_2 : T_2 \\ \Gamma, z_1 : \lfloor T_1 \rfloor \vdash z_1\ \mathsf{hasMut}\ f{:}S,\ T_2 \leq S \qquad (z_1\ \text{fresh})\end{array}}{\Gamma \vdash u_1.f = u_2 : T_2}$$

$$[\text{T-IF}]\ \frac{\begin{array}{c}\Gamma \vdash u : T_0,\ T_0 \leq \mathsf{bool} \qquad \Gamma, u \vdash u_1 : T_1, T_1 \leq T \\ \Gamma, \neg u \vdash u_2 : T_2, T_2 \leq T \qquad \Gamma \vdash T \qquad (T\ \text{fresh})\end{array}}{\Gamma \vdash \mathbf{if}\,(u)\ \mathbf{then}\ u_1\ \mathbf{else}\ u_2 : T} \qquad [\text{T-LOC}]\ \frac{\Sigma[l] = T}{\Gamma; \Sigma \vdash l : T}$$

Figure 4: Typing Rules

[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*, which is defined as:

Definition 1 (Compatibility Subtype). *A type* S *is a compatibility subtype of a type* T *under an environment* Γ *(we write* $\Gamma \vdash S \lesssim T$*), iff* $\langle S \xrightarrow{\Gamma} \lfloor T \rfloor \rangle = R \neq \mathsf{fail}$ *with* $\Gamma \vdash R \leq T$.

Here, $\lfloor T \rfloor$ 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. We use the predicate $\mathit{inv}(D, \nu)$ (as in CFJ), to denote the conjunction of the class invariants of C 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 casted to D. Formally:

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

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

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

[T-IF] We type conditionals with an upper bound of the types inferred for each of the two banches. Path sensitivity is encoded by including p and $\neg p$ in the environment used to check the then- and else-branch, respectively.

3.3 Type Soundness

The dynamic behavior of IRSC is described by a small step operational semantics of the form $H, u \longmapsto H', u'$, where *heaps* H map runtime *locations* l to objects $\mathbf{new}\ C(\overline{v})$. *Values* ν are either primitive constants c or runtime locations l. We exclude variables from the set of values, as they are eliminated by substitution when evaluating a top-level expression. The details of the operational semantics are standard and, hence, deferred to the supplementary material. Figure 4 includes rule T-LOC that checks location l under an environment Γ and a *store typing* Σ, that maps locations to types.

We establish type soundness results for IRSC in the form of a subject reduction (preservation) and a progress theorem that connect the static and dynamic semantics of IRSC.

Theorem 1 (Subject Reduction). *If (a)* $\Gamma; \Sigma \vdash u : T$*, (b)* $\Gamma; \Sigma \vdash H$*, and (c)* $H, u \longmapsto H', u'$*, then for some* T' *and* $\Sigma' \supseteq \Sigma$*: (i)* $\Gamma; \Sigma' \vdash u' : T'$*, (ii)* $\Gamma \vdash T' \lesssim T$*, and (iii)* $\Gamma; \Sigma' \vdash H'$.

Theorem 2 (Progress). *If* $\Gamma; \Sigma \vdash u : T$ *and* $\Gamma; \Sigma \vdash H$*, then either* u *is a value, or there exist* u'*,* H' *and* $\Sigma' \supseteq \Sigma$ *s.t.* $\Gamma; \Sigma' \vdash H'$ *and* $H, u \longmapsto H', u'$.

We defer the proofs to the supplementary material. As a corollary of the progress theorem we get that cast operators are guaranteed to succeed, hence they can safely be removed.

Corollary 3 (Safe Casts). *Cast operations can safely be erased when compiling to executable code.*

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* place them at the bottom of the type hierarchy. Instead rsc treats them as distinct primitive types inhabited solely by undefined and null , respectively. Consequently, the following code is accepted by TS but *rejected* by rsc:

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

Unsound Features TS has several unsound features deliberately chosen for backwards compatibility. These 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) segregating undefined and null , (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 $\mathtt{cast}_T :: (x{:}\mathtt{any}) \Rightarrow \{\nu{:}T \mid \nu = x\}$. It is straightforward to implement $\mathtt{cast}_T$ for first-order types T as a dynamic check that traverses the value, testing that its components satisfy the refinements [28]. 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 an array). 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 (z:A) $\Rightarrow$ {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.

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[1].

Implementing Hierarchies with bit-vectors The following describes a slice of the hierarchy of types used by the TypeScript compiler (tsc) v1.0.1.0:

```
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 .. }
```

tsc uses bit-vector valued flags to encode membership within 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 sub-interface 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 value, or an ObjectType value, since the relevant flag emerges from the bit-wise disjunction of the Interface flag with some other flags.

Specifying Hierarchies with Refinements rsc allows developers to *create* and *use* Type objects with the above invariant by specifying it a predicate typeInv [2]:

```
isMask<v,m,t> = mask(v,m) ⇒ impl(this,t)
typeInv<v> = isMask<v, 0x00000001, Any>
           ∧ isMask<v, 0x00000002, String>
           ∧ 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 Any interface, etc.

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

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

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:

$$(x:\{A \mid \mathbf{impl}(x, \mathtt{ObjectType})\}) \Rightarrow \{v:\mathtt{ObjectType} \mid v = x\}$$

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

Arrays TS's definitions file provides a detailed specification for the Array interface. Borrowing notation from Immutability Generic Java [39] [3], we extend this definition to account for the mutating nature of certain array operations:

```
interface Array<K 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 and pop) are only allowed on mutable arrays, and a.length returns the exact length of an immutable array a, and just a natural number otherwise.

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 $\mathtt{this.f_i} = e_i$, with $\_f_i = e_i$, where $\_f_i$ are *local* variables, and (b) all return points with a call $\mathtt{ctor\_init}\left(\overline{\_f_i}\right)$, where the signature for ctor_init is: $(\overline{f}{:}\,\overline{T}) \Rightarrow \mathtt{void}$. 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).

[3] In IGJ a type reference is of the form $\mathtt{C<M,\overline{T}>}$, where *immutability* argument M works as proxy for the immutability modifiers of the contained fields (unless overridden) and is 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.

(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<UM> = 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 UniqueMutable (or UM), 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. UM references obey stricter rules to avoid leaking of unique references. When createType returns, we can finally fix the mutability parameter of r to IM. We could also return Type<UM>, extending the *cooking* phase of the current object and allowing further initialization by the caller. We discuss more expressive approaches to initialization in § 6.

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 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 field or methods belong,
- ***Array Bounds*** rsc verifies that each array read (x[i]) or write (x[i] = e) occurs within the bounds of 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).

Benchmarks We took a number of existing JS or TS programs and ported them to rsc. We selected benchmarks that make heavy use of language constructs connected to the safety properties described above. These include parts of the Octane test suite, developed by Google as a JavaScript performance benchmark [12], the TS compiler [21], and the D3 [4] and Transducers libraries [7]:

- 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 raytracer 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,

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 5: **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.

- `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`).

Results Figure 5 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`. The numbers show `rsc` has annotation overhead comparable with TS.

Code Changes We had to modify the source in various small (but important) ways in order to facilitate verification.

- ***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.
- ***Constructors:*** As `rsc` does not yet support *default* constructor arguments, we modified relevant **`new`** calls in Octane to supply those explicitly. We also refactored `navier-stokes` to use traditional OO style constructors instead of JS records with function-valued 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 which were then explicitly called at use sites to instantiate the axioms (thereby bypassing non-linear analysis.)

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, 33]. 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 [34]. DRuby [10] uses intersection types to *represent* summaries for overloaded functions. TeJaS [20] combines occurrence typing with flow analysis to analyze JS [20]. 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 the typing relation inside the logic [6]. DJS combines nested refinements with alias types [29], 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 [39] and two-phase typing lets `rsc` use liquid refinement inference [26], 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.

Analyzing TypeScript Feldthaus *et al.* present a hybrid analysis to find discrepancies between TS interfaces [38] 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 [25].

Object Immutability `rsc` builds on existing methods for statically enforcing immutability. In particular, we build on Immutability Generic Java (IGJ) which encodes object and reference immutability using Java generics [39]. Subsequent work extends these ideas to allow (1) richer *ownership* patterns for creating immutable cyclic structures [40], (2) *unique* references, and ways to recover immutability after violating uniqueness, without requiring an alias analysis [13]. The above extensions are orthogonal to `rsc`; in the future, it would be interesting to see if they offer practical ways for accounting for immutability 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, 29]. Haack and Poll [16] 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* approach tracks, within the type system, the set of fields that remain to be initialized [24]. X10's *hardhat* flow-analysis based approach to initialization [41] and *Freedom Before Commitment* [30] are perhaps 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. To ensure soundness, as in X10's class-constraints, RSC's refinements are restricted to immutable variables and fields [32]. However, this alone is far too restrictive for TS. First, we make mutability parametric [39], and extend the refinement system accordingly. Second, we crucially obtain flow-sensitivity via SSA transformation, and path-sensitivity by incorporating branch conditions. Third, we account for reflection by encoding tags in refinements and two-phase typing [36]. Fourth, our design ensures that we can use liquid type inference [26] 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 [41], (2) automatic inference of trivial types via flow analysis [15], (3) verification of security properties, *e.g.* access-control policies in JS browser extensions [14].

References

[1] C. Anderson, P. Giannini, and S. Drossopoulou. Towards Type Inference for Javascript. In *Proceedings of the 19th European Conference on Object-Oriented Programming*, 2005.

[2] G. M. Bierman, A. D. Gordon, C. Hriţcu, and D. Langworthy. Semantic Subtyping with an SMT Solver. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, 2010.

[3] G. M. Bierman, M. Abadi, and M. Torgersen. Understanding typescript. In *ECOOP 2014 - Object-Oriented Programming - 28th European Conference, Uppsala, Sweden, July 28 - August 1, 2014. Proceedings*, pages 257–281, 2014.

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

[5] R. Chugh, D. Herman, and R. Jhala. Dependent types for javascript. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, pages 587–606, New York, NY, USA, 2012. ACM.

[6] R. Chugh, P. M. Rondon, and R. Jhala. Nested Refinements: A Logic for Duck Typing. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 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 the ACM International Conference on Object Oriented Programming Systems Language and Applications*, 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 the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI '02, pages 234–245, New York, NY, USA, 2002. ACM. ISBN 1-58113-463-0.

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

[11] P. Gardner, S. Maffeis, and G. D. Smith. Towards a program logic for javascript. In *POPL*, pages 31–44, 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 *OOPSLA*, 2012.

[14] A. Guha, M. Fredrikson, B. Livshits, and N. Swamy. Verified security for browser extensions. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, SP '11, pages 115–130, Washington, DC, USA, 2011. IEEE Computer Society.

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

[16] C. Haack and E. Poll. Type-Based Object Immutability with Flexible Initialization. In *ECOOP*, Berlin, Heidelberg, 2009.

[17] A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: A Minimal Core Calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450, May 2001. ISSN 0164-0925.

[18] K. Knowles and C. Flanagan. Hybrid Type Checking. *ACM Trans. Program. Lang. Syst.*, 32(2), Feb. 2010.

[19] K. Knowles and C. Flanagan. Compositional reasoning and decidable checking for dependent contract types. In *Proceedings of the 3rd Workshop on Programming Languages Meets Program Verification*, PLPV '09, pages 27–38, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-330-3.

[20] B. S. Lerner, J. G. Politz, A. Guha, and S. Krishnamurthi. TeJaS: Retrofitting Type Systems for JavaScript. In *Proceedings of the 9th Symposium on Dynamic Languages*, 2013.

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

[22] G. Nelson. Techniques for program verification. Technical Report CSL81-10, Xerox Palo Alto Research Center, 1981.

[23] N. Nystrom, V. Saraswat, J. Palsberg, and C. Grothoff. Constrained Types for Object-oriented Languages. In *Proceedings of the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications*, OOPSLA '08, pages 457–474, New York, NY, USA, 2008. ACM.

[24] X. Qi and A. C. Myers. Masked types for sound object initialization. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '09, pages 53–65, New York, NY, USA, 2009. ACM.

[25] A. Rastogi, N. Swamy, C. Fournet, G. Bierman, and P. Vekris. Safe & efficient gradual typing for typescript. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 167–180, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3300-9.

[26] P. M. Rondon, M. Kawaguci, and R. Jhala. Liquid Types. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2008.

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

[28] E. L. Seidel, N. Vazou, and R. Jhala. Type targeted testing. In *Proceedings of the 24th European Symposium on Programming on Programming Languages and Systems - Volume 9032*, pages 812–836, New York, NY, USA, 2015. Springer-Verlag New York, Inc. ISBN 978-3-662-46668-1.

[29] F. Smith, D. Walker, and G. Morrisett. Alias Types. In *In European Symposium on Programming*, pages 366–381. Springer-Verlag, 1999.

[30] A. J. Summers and P. Mueller. Freedom before commitment: A lightweight type system for object initialisation. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, pages 1013–1032, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0940-0.

[31] N. Swamy, J. Chen, C. Fournet, P.-Y. Strub, K. Bhargavan, and J. Yang. Secure distributed programming with value-dependent types. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*, ICFP '11, pages 266–278, New York, NY, USA, 2011. ACM.

[32] O. Tardieu, N. Nystrom, I. Peshansky, and V. Saraswat. Constrained kinds. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, pages 811–830, New York, NY, USA, 2012. ACM.

[33] P. Thiemann. Towards a Type System for Analyzing Javascript Programs. In *Proceedings of the 14th European Conference on Programming Languages and Systems*, 2005.

[34] S. Tobin-Hochstadt and M. Felleisen. Logical Types for Untyped Languages. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, 2010.

[35] N. Vazou, E. L. Seidel, R. Jhala, D. Vytiniotis, and S. Peyton-Jones. Refinement Types for Haskell. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, 2014.

[36] P. Vekris, B. Cosman, and R. Jhala. Trust, but verify: Two-phase typing for dynamic languages. In *29th European Conference on Object-Oriented Programming, ECOOP 2015, July 5-10, 2015, Prague, Czech Republic*, pages 52–75, 2015.

[37] H. Xi and F. Pfenning. Dependent Types in Practical Programming. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1999.

[38] B. Yankov. http://definitelytyped.org.

[39] Y. Zibin, A. Potanin, M. Ali, S. Artzi, A. Kiezun, and M. D. Ernst. Object and Reference Immutability Using Java Generics. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, 2007.

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

[41] Y. Zibin, D. Cunningham, I. Peshansky, and V. Saraswat. Object initialization in x10. In *Proceedings of the 26th European Conference on Object-Oriented Programming*, ECOOP'12, pages 207–231, Berlin, Heidelberg, 2012. Springer-Verlag.