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 Types [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 IGI [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 framework to render refinement typing practical on real world programs. Concretely, we make the following contributions:

• 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:

\[
\begin{align*}
\text{type } \text{nat} & \quad = \{ v : \text{number} \mid 0 \leq v \} \\
\text{type } \text{pos} & \quad = \{ v : \text{number} \mid 0 < v \} \\
\text{type } \text{natNn} & \quad = \{ v : \text{nat} \mid v = n \} \\
\text{type } \text{idxNa} & \quad = \{ v : \text{nat} \mid v < 1\text{en}(a) \}
\end{align*}
\]

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, \( 1\text{en} \) is an uninterpreted function.
that describes the size of the array \( a \). We write \( t \) to abbreviate truly refined types, i.e. \( \{ v : t \mid \text{true} \} \); e.g. \texttt{number} abbreviates \( \{ v : \text{number} \mid \text{true} \} \).

**Specifications** 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, \( \{ x : \text{nat} \} \Rightarrow \{ y : \text{nat} \mid x < y \} \) 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, \texttt{reduce} from Figure 1 can be specified as \texttt{reduce}:

\[
\langle A, B \rangle (a : \alpha C), f : (B, A, \text{id}<a>) \Rightarrow 0, x : B) \Rightarrow B
\] (1)

This type is a precise summary for the higher-order behavior of \texttt{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 \( a[i] \), write \( a[i] = e \) and length access \( a.length \) as calls \texttt{get}(\( a, i \)), \texttt{set}(\( a, i, e \)) and \texttt{length}(\( a \)) where:

\[
\begin{align*}
\text{get} : & \{ a : T[i] \mid i : \text{idx}<a> \} \Rightarrow T \\
\text{set} : & \{ a : T[i] \mid i : \text{idx}<a>, e : T \} \Rightarrow \text{void} \\
\text{length} : & \{ a : T[] \} \Rightarrow \text{nat}<\text{len}(a)>
\end{align*}
\]

**Verification** Refinement typing ensures that the actual parameters supplied at each \texttt{call} to \texttt{get} and \texttt{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:

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

The input type requires that \( \text{arr} \) be non-empty:

\[
\text{type NEArray<T> = } \{ v : T[i] \mid 0 < \text{len}(v) \}
\]

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

\[
\Gamma_{\text{head}} \vdash \{ v = 0 \} \subseteq \text{idx}(\text{arr})
\]

which reduces to the logical verification condition (VC):

\[
0 < \text{len}(\text{arr}) \Rightarrow v = 0 \Rightarrow 0 \leq v < \text{len}(\text{arr})
\]

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:

\[
\begin{align*}
\text{function head0}(\text{a : number}[]): \text{number} \{ \\
\quad \text{if} (0 < \text{a.length}) \text{return head}(\text{a}); \\
\quad \text{return 0} ;
\}
\end{align*}
\]

Recall that \texttt{head} should only be invoked with non-empty arrays. The call to \texttt{head} above occurs under \( \Gamma_{\text{head0}} \) defined as:

\[
\text{head0} : \{ a : \text{number}[] \mid 0 < \text{len}(a) \} \Rightarrow \text{number}
\]

yielding the valid VC: \( 0 < \text{len}(a) \Rightarrow v = a \Rightarrow 0 \leq v < \text{len}(v) \).

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

Types make it easy to track such flows: we need only determine the instantiation of the polymorphic type variables of \texttt{reduce} at this call site inside \texttt{minIndex}. The type of the \( f \) parameter in the instantiated type corresponds to a signature for the closure \texttt{step} which will let us verify the closure’s implementation. Here, \texttt{rsc} automatically instantiates:

\[
\begin{align*}
A & \mapsto \text{number} \\
B & \mapsto \text{idx}(a)
\end{align*}
\]

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

\[
\text{step} : (\text{idx}<a>, \text{number}, \text{idx}<a>) \Rightarrow \text{idx}<a> \quad \emptyset : \text{idx}<a> -
\]

The initial value \( \emptyset \) is indeed a valid \text{idx}<\( a \) thanks to the \texttt{a.length} check at the start of the function. To check \texttt{step}, assume that its inputs have the above types:

\[
\begin{align*}
\text{min} : & \text{idx}<a>, \text{curr : number}, i : \text{idx}<a>
\end{align*}
\]

The body is safe as the index \( i \) is trivially a subtype of the required \text{idx}<\( a \), and the output is one of \text{min} or \( i \) and hence, of type \text{idx}<\( a \) as required.

#### 2.1.2 Overloading

Dynamic languages extensively use \textit{value-based overloading} to simplify library interfaces. For example, a library may export:

\[
\text{function $\text{reduce}(a, f, x)$}
\]

\[
\quad \text{if} (\text{arguments.length} === 3) \text{return reduce}(a, f, x); \\
\quad \text{return reduce}(\text{a.slice}(1), f, a[0]);
\]

The function \texttt{$\text{reduce}$} has two distinct types depending on its parameters’ \textit{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 \textit{value-based} overloading [36]. First, we specify overloading as an intersection type. For example, \texttt{$\text{reduce}$} gets the following signature, which is just the conjunction of the two overloaded behaviors:

\[
\begin{align*}
\land & \langle A, B \rangle (a : A[]), f : (A, A, \text{idx}<a>) \Rightarrow \text{A} \quad \text{// 1} \\
\land & \langle A, B \rangle (a : A[]), f : (B, A, \text{idx}<a>) \Rightarrow \text{B}, x : B \Rightarrow \text{B} \quad \text{// 2}
\end{align*}
\]

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

\[
\text{assert} : (b : (v : \text{bool} \mid v = \text{true})) \Rightarrow \text{A}
\]

To check \texttt{$\text{reduce}$}, we specialize it per overload context:
Recall that the constraints via a subtyping constraints for unknown refinement type instantiations, (b) performs environments: verifies the known synthesize text) is replaced with 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:

\[ \Gamma_1 \vdash \text{arguments} : [\text{len}(\nu) = 2], \text{len}(\text{arguments}) = 3 \]
\[ \Gamma_2 \vdash \text{arguments} : [\text{len}(\nu) = 3], \text{len}(\text{arguments}) \neq 3 \]

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 \( \Gamma_{\text{reduce}} \). 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:

\[ A \mapsto \{ v : \text{number} \mid \kappa_A \} \quad B \mapsto \{ v : \text{number} \mid \kappa_B \} \]

where the refinement variables \( \kappa_A \) and \( \kappa_B \) represent the unknown refinements. We substitute the above in the signature for reduce to obtain a context-sensitive template:

\[ (a : \kappa_A, (B, \kappa_A, \text{id}x(a)) \Rightarrow \kappa_B, \kappa_B) \Rightarrow \kappa_B \]  

(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 calls, 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 \( \kappa_A[I] \), \( \kappa_B \) and \( (\kappa_B, \kappa_A, \text{id}x(a)) \Rightarrow \kappa_B \). Checking a and 0 yields the subtyping constraints:

\[ \Gamma \vdash \text{number}[] \subseteq \kappa_A[] \quad \Gamma \vdash \{ v = 0 \} \subseteq \kappa_B \]

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

\[ \Gamma_{\text{step}} \vdash \text{min} : \kappa_B, \text{cur} : \kappa_A, i : \text{id}x(a) \]

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

\[ \Gamma_{\text{step}} \vdash \kappa_B \subseteq \text{id}x(a) \]

As i and min flow to the output type \( \kappa_B \), we get:

\[ \Gamma_{\text{step}} \vdash \text{id}x(a) \subseteq \kappa_B \quad \Gamma_{\text{step}} \vdash \kappa_B \subseteq \kappa_B \]

Step 3: Fixpoint The above subtyping constraints over the \( \kappa \) variables are reduced via the standard rules for co- and contra-variant subtyping, into Horn implications over the \( \kappa_s \). rsc solves the Horn implications via (predicate) abstract interpretation [26] to obtain the solution \( \kappa_A \mapsto \text{true} \) and \( \kappa_B \mapsto 0 \leq v < \text{len}(a) \) which is exactly the instantiation in (2) that satisfies the subtyping constraints, and proves \text{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 \( \phi \) variable. Third, we generate and solve the subtyping constraints to infer the types for the \( \phi \)-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)

\[ \text{function reduce}(a, f, x) \{ \]
\[ \text{var } r = x, i = 0; \]
\[ \text{while }\{ i2, r2 = \phi((i0, r0), (i1, r1)) \}
\[ i2 < a.length; \]
\[ r1 = f(r2, a[i2], i2); i1 = i2 + 1; \]
\[ \text{return } r2; \}
\]

where i2 and r2 are the \( \phi \) variables for \( i \) and \( r \) respectively. Second, we generate templates for the \( \phi \)-variables:

\[ i2 : \{ v : \text{number} \mid \kappa_{i2} \} \quad r2 : \{ v : \text{B} \mid \kappa_{r2} \} \]  

(4)

We need not generate templates for the SSA variables i0, r0, i1 and r1 as they are those of the expressions they are assigned. Third, we generate subtyping constraints as before; the \( \phi \) assignment generates additional constraints:

\[ \Gamma_0 \vdash \{ v = i0 \} \subseteq \kappa_{i2} \quad \Gamma_1 \vdash \{ v = i1 \} \subseteq \kappa_{i2} \]
\[ \Gamma_0 \vdash \{ v = r0 \} \subseteq \kappa_{r2} \quad \Gamma_1 \vdash \{ v = r1 \} \subseteq \kappa_{r2} \]

where \( \Gamma_0 \) is the environment at the “exit” of the basic blocks where \( i0, r0 \) are defined:

\[ \Gamma_0 \equiv a : \text{number}[], x : \text{B}, i0 : \text{natN}(0), r0 : \{ v : \text{B} \mid v = x \} \]

Similarly, the environment \( \Gamma_1 \) 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:

\[ \Gamma_1 \equiv \Gamma_0, i1 : \text{natN}(i1 + 1), r1 : \text{B}, i2 < \text{len}(a) \]

Finally, the above constraints are solved to:

\[ \kappa_{i2} \mapsto 0 \leq v < \text{len}(a) \quad \kappa_{r2} \mapsto \text{true} \]

which verifies that the “callback” \( f \) is indeed called with values of type \( \text{id}x(a) \), as it is only called with \( i2 : \text{id}x(a) \), 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
The class 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 prevent updates to the field outside the `constructor`, and to 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:

```java
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 of `5` exceeds the actual width (i.e., `z.w`), namely 3:

```java
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:

```java
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 XI0), 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(T)`, object construction `new C(T)`, 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 `♭f:T` and `♭♭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.

```latex
\begin{align*}
\text{e} & \quad::= \quad x \mid c \mid \text{this} \mid \text{e.f} \mid \text{e.m(T)} \mid \text{new C(T)} \mid \text{e as T} \\
\text{s} & \quad::= \quad \text{var}\ \text{x} = \text{e} \mid \text{e.f} = \text{e} \mid \text{e} = \text{e} \\
\text{if (e)} \quad\text{then} \quad\text{s} \quad\text{else} \quad\text{s} \\
\text{M} & \quad::= \quad \text{def}\ \text{m} \{\text{x}.\text{T}\} \{\text{p}\} : \text{T} = \text{s}; \text{return}\ \text{e}\} \\
\text{L} & \quad::= \quad \text{class}\ \text{C} \{\text{♭f:T}; \text{♭♭S}\} \{\text{p}\} \text{ extends}\ \text{R}\ \{\text{M}\} \\
\text{P} & \quad::= \quad \text{♭♭C}; \text{s}
\end{align*}
```

**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:

```latex
\begin{align*}
\text{u.w} & \quad::= \quad x \mid c \mid \text{this} \mid \text{u.f} \mid \text{u.m(T)} \mid \text{new C(T)} \mid \\
\text{u as T} \mid \text{let}\ x = \text{u}\ \text{in}\ \text{u} \mid \text{u.f} = \text{u} \\
\text{if (u)} \quad\text{then}\ \text{u} \quad\text{else}\ \text{u} \\
\text{M} & \quad::= \quad \text{def}\ \text{m} \{\text{x}.\text{T}\} \{\text{p}\} : \text{T} = \text{u} \\
\text{L} & \quad::= \quad \text{class}\ \text{C} \{\text{♭♭f:T}; \text{♭♭S}\} \{\text{p}\} \text{ extends}\ \text{R}\ \{\text{M}\} \\
\text{P} & \quad::= \quad \text{♭♭C}; \text{u}
\end{align*}
```

**SSA Transformation** We translate FRSC to IRSC via a Static Single Assignment transformation (♭), described in Figure 3. This process uses a translation state $\Delta$, 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 $\Delta$. 

---

*Figure 2: Two-Dimensional Arrays*
The translating judgment of statements $s$ has the form: $\Delta \vdash s \rightarrow E[\cdot] / \Delta'$. The SSA context $E[\cdot]$ is an expression containing a hole $[\cdot]$ in its body (e.g. $\text{let } x = u \text{ in } [\cdot]$). The hole is to be filled in by the translation of the subsequent statements. Output environment $\Delta'$ reflects the potential introduction of new SSA variables. Rule $\text{S-VarDecl}$ introduces such fresh variable $x_0$ and assigns it to the binding for the newly declared variable $x$. Field assignments do not affect $\Delta$ (rule $\text{S-DotAsgn}$).

The most interesting case is conditionals (rule $\text{S-If}$). Each branch is translated separately producing an SSA context and a refinement type. These terms can be variables $x$, primitive constants $c$, the reserved value variable $\nu$, the reserved variable this to denote the containing object, field access $t.f$ and uninterpreted function applications $f(T)$.

### 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 $\mathcal{C}$ has $I$ (that encodes inclusion of an element $I$ in a class $C$) into two cases: $C \text{ hasMut } I$ and $C \text{ hasImm } I$, 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 $\Gamma$ contains type bindings $x : T$ and guard predicates $p$ that encode path sensitivity. $\Gamma$ is well-formed if all of its bindings are well-formed. A refinement type is well-formed in an environment $\Gamma$ if all symbols (simple or qualified) in its logical predicate are (i) bound in $\Gamma$, 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
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 $\Gamma \vdash \{v : N \mid p\} \leq \{v : N \mid p'\}$ reduces to a verification condition: Valid($\Gamma$) $\Rightarrow$ $\{p\} \Rightarrow \{p\}$, where $\Gamma$ is the embedding of environment $\Gamma$ into our logic that accounts for both guard predicates and variable bindings:

$$\Gamma \equiv \bigwedge (p \mid p \in \Gamma) \land \bigwedge (x : [v : N \mid p] \in \Gamma)$$

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 $\Gamma \vdash : T$, 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 $\ominus$ operator:

$$\{v : N \mid p\} \ominus p' \equiv \{v : N \mid p \land p'\}$$

$$\ominus x : S, T \vdash p \equiv \ominus x : S, (T \ominus p)$$

$$\text{self } (T, t) \equiv T \ominus (v = t)$$

**[T-Field-M]** Here we avoid such strengthening, as the value of field $g_i$ is mutable, so cannot appear in refinements.
Type IRSC is described by a small step operational semantics of the form H, u \rightarrow H', u', where heaps H map runtime locations l to objects new C [\forall]. Values \nu are either primitive constants 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 \Gamma and a store typing \Sigma, 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 \mapsto H', u', then for some T' and \Sigma' \supseteq \Sigma: (i) \Gamma; \Sigma' \vdash u' : T', (ii) \Gamma; \Sigma' \leq 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 \mapsto 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:

```javascript
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:

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

is equivalent to a class `PointC` defined as:

```javascript
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:

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

However, `pC` is not an instance of `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`:

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

**Unsound Features** TS has several unsound features deliberately chosen for backwards compatibility. These include (1) treating `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 `newclassing`, 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 `castT : (x: any) ⇒ (v:T | v = x)`. It is straightforward to implement `castT` 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 frequently use 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:

```javascript
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 `2: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.

### 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.

**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:

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

```javascript
interface InterfaceType extends Type { ... }
```

```javascript
enum TypeFlags { Any = 0x00000001, String = 0x00000002,
 Number = 0x00000004, Class = 0x00000040,
 Interface = 0x00000080, 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 `0x00000080` 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`:

```javascript
isMask<v, m, t> = mask(v, m) ⇒ impl(this, t) typeInv<v><><> = isMask<v, 0x00000001, Any>∧ isMask<v, 0x00000002, String>∧ isMask<v, 0x0000300, ObjectType>
```

and then refining `TypeFlags` with the predicate

```javascript
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:

```javascript
function getPropertiesOfType(t: Type): Symbol[] {
  if (t.flags & TypeFlags.Object) {
    var o = new 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>x is treated as a call to a function with signature:

```javascript
(x: (impl(x, ObjectType)) ⇒ (v: ObjectType|v=x)
```

The if-test ensures that the immutable field t.flags masked with 0x000003C00 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] 1, we extend this definition to account for the mutating nature of certain array operations:

```javascript
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 this.f1 = e1, with f1 = e1, where f1 are local variables, and (b) all return points with a call ctor_init (T), where the signature for ctor_init is: (T,T) ⇒ 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):

```javascript
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 UM.

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[1]) or write (x[1] = 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.
Table: LOC vs Time

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>LOC</th>
<th>T</th>
<th>M</th>
<th>R</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>navier-stokes</td>
<td>366</td>
<td>3</td>
<td>18</td>
<td>39</td>
<td>473</td>
</tr>
<tr>
<td>splay</td>
<td>206</td>
<td>18</td>
<td>2</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>Richards</td>
<td>304</td>
<td>61</td>
<td>5</td>
<td>17</td>
<td>7</td>
</tr>
<tr>
<td>raytrace</td>
<td>576</td>
<td>68</td>
<td>14</td>
<td>2</td>
<td>15</td>
</tr>
<tr>
<td>transducers</td>
<td>588</td>
<td>138</td>
<td>13</td>
<td>11</td>
<td>12</td>
</tr>
<tr>
<td>d3-arrays</td>
<td>189</td>
<td>3</td>
<td>4</td>
<td>10</td>
<td>37</td>
</tr>
<tr>
<td>tsc-checker</td>
<td>293</td>
<td>10</td>
<td>48</td>
<td>12</td>
<td>62</td>
</tr>
<tr>
<td><strong>TOTAL</strong></td>
<td>2522</td>
<td>344</td>
<td>104</td>
<td>91</td>
<td></td>
</tr>
</tbody>
</table>

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|P ≤ v} or dependent signatures like (a:T[]; n:idx<ao)⇒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[rs + 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]. DIS 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]. DIS 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: Feldhaus 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 may 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


