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 [29, 39]. The extensibility and modularity offered by refinement types have enabled their use in a variety of applications in typed, functional languages, like ML [28, 39], Haskell [37], and F♯ [33]. Unfortunately, attempts to apply refinement typing to scripts have proven to be impractical due to the interaction of the machinery that accounts for imperative updates and higher-order functions [5] (§6).

In this paper, we introduce Refined TypeScript (RSC): a novel, lightweight refinement type system for TypeScript, a typed superset of JavaScript. Our design of RSC addresses three intertwined problems by carefully integrating and extending existing ideas from the literature. First, RSC accounts for mutation by using ideas from IGJ [41] 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 [38], where dynamic behaviors are specified via union and intersection types, and verified by reduction to refinement typing. Third, the above are carefully designed to permit refinement inference via the Liquid Types [28] framework to render refinement typing practical on real world programs. Concretely, we make the following contributions:

• We develop a core calculus that 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 extendable enough to verify a variety of properties from classic array-bounds checking to program specific invariants needed to ensure safe reflection: critical invariants that are well beyond the scope of existing techniques for imperative scripting languages (§5).

2. Overview

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

Types and Refinements

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

\[
\text{type } \text{nat} = \{ v : \text{number} \mid 0 \leq v \} \\
\text{type } \text{pos} = \{ v : \text{number} \mid 0 < v \}
\]

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 [38], where dynamic behaviors are specified via union and intersection types, and verified by reduction to refinement typing. Third, the above are carefully designed to permit refinement inference via the Liquid Types [28] framework to render refinement typing practical on real world programs. Concretely, we make the following contributions:

• We develop a core calculus that 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 extendable enough to verify a variety of properties from classic array-bounds checking to program specific invariants needed to ensure safe reflection: critical invariants that are well beyond the scope of existing techniques for imperative scripting languages (§5).

2. Overview

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

Types and Refinements

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

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 \((x_1 : T_1, ..., x_n : T_n) \Rightarrow T\), 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, \((x : \text{nat}) \Rightarrow \{\nu : \text{nat} | x < \nu\}\) 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 \(T_{\text{reduce}}:\)

\[ <A,B> : \forall a:A[|f:(B,A,idx<\text{arr}>) \Rightarrow B, x:B) \Rightarrow B \quad (1) \]

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:

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

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:

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

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

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

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

\[
\Gamma_{\text{head}} \vdash \{\nu = 0\} \subseteq \text{idx}\langle\text{arr}\rangle
\]

which reduces to the logical verification condition (VC):

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

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

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

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

Recall that head should only be invoked with non-empty arrays. The call to head above occurs under \(\Gamma_{\text{head0}}\) defined as: \(a\.number[| 0 < \text{len}(a)\) 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:

\[
\Gamma_{\text{head0}} \vdash \{\nu = a\} \subseteq \text{NEArray(number)}
\]

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

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

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

\[ A \mapsto \text{number} \quad B \mapsto \text{idx}(a) \quad (2) \]

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

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

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

\[
\text{min} : \text{idx}<a>, \text{curr} : \text{number}, \text{i} : \text{idx}<a>
\]

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 value-based overloading to simplify library interfaces. For example, a library may export:

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

The function \$_\text{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 [38].

**Intersection Types** Refinements let us statically verify value-based overloading via an approach called Two-Phased Typing [38]. First, we specify overloading as an intersection type. For example, \$_\text{reduce} gets the following signature, which is just the conjunction of the two overloading behaviors:

\[
(a, B) : A : [+, f : (A, A, \text{idx}<a>) \Rightarrow A) \Rightarrow A \quad // \ 1
(a, B) : A : [+, f : (B, A, \text{idx}<a>) \Rightarrow B, x : B) \Rightarrow B \quad // \ 2
\]

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

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

```plaintext
assert : b : (v : bool | v = true)) \Rightarrow A
```

To check \$_\text{reduce}, we specialize it per overload context:

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

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

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

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

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

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

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

\[
A \mapsto \{\nu : \text{number} | \kappa_A\} \quad B \mapsto \{\nu : \text{number} | \kappa_B\}
\]

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:

\[
(a : \kappa_A[\], \ (\kappa_B, \kappa_A, \text{idx}(a)) \Rightarrow \kappa_B, \kappa_B) \Rightarrow \kappa_B \quad (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 \( a, \text{Ø}, \text{step} \) are passed in as arguments, we check that they respectively have the types \( \kappa_A[\], \kappa_B \) and \( (\kappa_B, \kappa_A, \text{idx}(a)) \Rightarrow \kappa_B \). Checking \( a, \text{Ø} \) yields the subtyping constraints:

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

$$\Gamma_{\text{step}} \vdash \min \cdot \kappa_B, \text{cur} \cdot \kappa_a, i \cdot \text{idx}(a)$$

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

$$\Gamma_{\text{step}} \vdash \kappa_B \sqsubseteq \text{idx}(a)$$

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

$$\Gamma_{\text{step}} \vdash \text{idx}(a) \sqsubseteq \kappa_B \quad \Gamma_{\text{step}} \vdash \kappa_B \sqsubseteq \kappa_B$$

\textbf{Step 3: Fixpoint} The above subtyping constraints over the $\kappa$ variables are reduced via the standard rules for co- and contra-variant subtyping, into \textit{Horn implications} over the $\kappa$. $rsc$ solves the Horn implications via (predicate) abstract interpretation \cite{AC98} to obtain the solution $\kappa_A \mapsto \text{true}$ and $\kappa_B \mapsto 0 \leq \nu < \text{len}(a)$ which is exactly the instantiation in (2) that satisfies the subtyping constraints, and proves $\min\text{Index}$ is array-safe.

\subsection{2.2.2 Assignments}

Next, let us see how the signature for \texttt{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.

\textbf{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 for the SSA variables $\phi$ and hence, the “loop-invariants” needed for verification.

Let us see how this process lets us verify \texttt{reduce} from Figure 1. First, we convert the body to SSA form (§3.1)

\begin{verbatim}
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;
}
\end{verbatim}

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

$$i2 : \nu :: \text{number} | \kappa_{i2} \quad r2 : \nu :: \text{B} | \kappa_{r2} \quad (4)$$

We need not generate templates for the SSA variables $x0$, $r0$, $i1$ and $r1$ as their types are those of the expressions they are assigned. Third, we generate subtyping constraints as before; the $\phi$ assignment generates additional constraints:

$$\Gamma_0 \vdash \{ \nu :: i0 \} \sqsubseteq \kappa_{i2} \quad \Gamma_1 \vdash \{ \nu :: i1 \} \sqsubseteq \kappa_{i2}$$

Finally, the above constraints are solved to:

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

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

\subsection{2.2.3 Mutation}

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

\textbf{Class Invariants} Class $\text{Field}$ implements a 2-dimensional vector, “unrolled” into a single array $\text{dens}$, 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. \textit{pos}) and that $\text{dens}$ be a $\text{grid}$ with dimensions specified by

\begin{verbatim}

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;
    }
}
\end{verbatim}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure2.png}
\caption{Two-Dimensional Arrays}
\end{figure}
The above invariants are additionally enforced by the mutable and immutable fields.

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

Constructors We can create \textit{instances} of Field, by using \texttt{new Field(...)} which invokes the \texttt{constructor} with the supplied parameters. \texttt{rsc} ensures that at the \textit{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, \texttt{rsc} accepts the first call, but rejects the second:

\[
\begin{align*}
\texttt{var } z &= \texttt{new Field}(3,7, \texttt{new Array}(45)); // OK \\
\texttt{var } q &= \texttt{new Field}(3,7, \texttt{new Array}(44)); // BAD
\end{align*}
\]

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

\[
\begin{align*}
\texttt{z.setDensity}(2,5,-5); // OK \\
\texttt{z.getDensity}(5,2); // BAD
\end{align*}
\]

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

\[
\begin{align*}
\texttt{var } z &= \texttt{new Field}(3,7, \texttt{new Array}(45)); \\
\texttt{z.reset}(\texttt{new Array}(45)); // OK \\
\texttt{z.reset}(\texttt{new Array}(5)); // BAD
\end{align*}
\]

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 CFI [25], (the language used to formalize X10), which in turn is based on Featherweight Java [18]. 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

3.1.1 Source Language (FRSC)

The syntax of this language is given below. Meta-variable \( e \) ranges over expressions, which can be variables \( x \), constants \( c \), property accesses \( e.f \), method calls \( e.m(\bar{e}) \), object construction \( \texttt{new } C(\bar{e}) \), and cast operations \( \langle T \rangle.e \). 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, \textit{i.e.} a statement immediately followed by a returned expression. Class definitions distinguish between immutable and mutable members, using \( \circ f : T \) and \( \sqcap f : T \), respectively. As in CFJ, each class and method definition is associated with an invariant \( p \).

\[
\begin{align*}
\texttt{e} &::= x | c | \texttt{this}.e.f | e.m(\bar{e}) | \texttt{new } C(\bar{e}) | \langle T \rangle.e \\
\texttt{s} &::= \texttt{var } x = e | e.f = e | x = e | \texttt{if}(e)(s)\texttt{else}(s) | s ; s | \texttt{skip} \\
\texttt{B} &::= s ; \texttt{return } e \\
\texttt{M} &::= m(\bar{x} : T ) \{ p \} : T \{ B \} \\
\texttt{F} &::= \circ f : T | \sqcap f : T | F_{1} ; F_{2} \quad \texttt{and } \texttt{extends } R \{ F , \bar{M} \}
\end{align*}
\]

The core system does not formalize: (a) method overloading, which is orthogonal to the current contribution and has been investigated in previous work [38], or (b) method overriding, which means that method names are distinct from the ones defined in parent classes.

3.1.2 Intermediate Language (IRSC)

FRSC, while syntactically similar to TS, is not entirely suitable for refinement type checking in its current form, due to features like assignment. To overcome this challenge we translate FRSC to a functional language IRSC through a Static Single Assignment (SSA) transformation, which produces programs that are equivalent (in a sense that we will make precise in the sequel). In IRSC, statements are replaced by let-bindings and new variables are introduced for each variable being reassigned in the respective FRSC code. Thus, IRSC has the following syntax:

\[
\begin{align*}
\texttt{e} &::= x | c | \texttt{this}.e.f | e.m(\bar{e}) | \texttt{new } C(\bar{e}) | e \texttt{ as } T | e.f \leftarrow e | U \langle e \rangle \\
\texttt{U} &::= \{ \} | \texttt{let } x = e \texttt{ in } \{ \} | \texttt{letif } [ \bar{x} , \bar{x}_1 , \bar{x}_2 ](e) \uparrow U_{1} : U_{2} \texttt{ in } \{ \} \\
\texttt{F} &::= \circ f : T | \sqcap f : T | F_{1} ; F_{2} \quad \texttt{and } \texttt{extends } R \{ F , \bar{M} \}
\end{align*}
\]

The majority of the expression forms \( e \) are unsurprising. An exception is the form of the SSA context \( U \), which corresponds to the translation of a statement \( s \) and contains a \texttt{hole } \{ \} that will hold the translation of the continuation of \( s \).

\textbf{SSA Transformation} Figure 3 describes the SSA transformation, that uses a \textit{translation environment }\( \delta \), to map FRSC
variables $x$ to IRSC variables $x$. The translation of expressions $e$ to $e'$ is routine: as expected, S-Var maps the source level $x$ to the current binding of $x$ in $\delta$. The translating judgment of statements $s$ has the form: $\delta \vdash s \rightarrow U; \delta'$. The output environment $\delta'$ is used for the translation of the expression that will fill the hole in $U$.

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

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

Fresh $\Phi$-variables $\pi'$ populate the output SSA environment $\delta'$. Along with the versions of the $\Phi$-variables for each branch ($\pi_1$ and $\pi_2$), they are used to annotate the produced structure.

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

### 3.1.3 Consistency

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

$$
\begin{align*}
\text{P} & \equiv S; B & \text{P} & \equiv S; e \\
\text{P} & \equiv S; B & \text{R} & \equiv K; B & \text{R} & \equiv K; e \\
\text{K} & \equiv S; L; x; H & \text{K} & \equiv S; L; x; H
\end{align*}
$$

Runtime state $K$ consists of the call stack $X$, the local store of the current stack frame $L$ and the heap $H$. The runtime state for IRSC, $R$ only consists of the signatures $S$ and a heap $H$.

We establish the consistency of the SSA transformation by means of a weak forward simulation theorem that connects the dynamic semantics of the two languages. To that end, we define small-step operational semantics for both languages, of the form $R \rightarrow R'$ and $R \rightarrow R'$. Figure 4 presents the dynamic behavior of the two languages. Rules for FRSC have been adapted from Rastogi et al. [27]. Note how in rule R-CAST the cast operation reduces to a call to the built-in check function, where $\lfloor T \rfloor$ encodes type $T$. Rules for IRSC are mostly routine, with the exception of rule R-LETIF: expression $e$ has been produced assuming $\Phi$-variables $\pi$. After the branch has been determined we pick the actual $\Phi$-variables ($\pi_1$ or $\pi_2$) and replace them in $e$. This formulation allows us to perform all the SSA-related bookkeeping in a single reduction step, which is key to preserving our consistency invariant that IRSC steps faster than FRSC.

We also extend our SSA transformation judgment to runtime configurations, leveraging the SSA environments that have been statically computed for each program entity, which now form a global SSA environment $\Delta$, mapping each AST node (e, s, etc.) to an SSA environment $\delta$:

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

We assume that the compile-time SSA translation yields this environment as a side-effect (e.g. $\delta \vdash e \mapsto e$ produces...
Operational Semantics for FRSC

\[
\begin{array}{ll}
\text{R-VAL} & K; v \rightarrow K; \text{skip} \\
\text{R-EVALCTX} & S; L; \cdots; H; e \rightarrow S; L'; \cdots; H'; e' \\
\text{R-VAR} & K; x \rightarrow K; K.L(x) \\
\text{R-NEW} & S; L; X; H; E[e] \rightarrow S; L'; X; H'; E[e'] \\
\text{R-DOTREF} & K.H(1) = \{\text{proto}: l'; f: \overline{F}\} \\
& \text{LETIN } \Gamma \vdash K.H(1); S; S \leq T \\
& K; l; f \leftrightarrow v \rightarrow K; H'[f \mapsto v] \\
\end{array}
\]

Operational Semantics for IRSC

\[
\begin{array}{ll}
\text{RC-ECTX} & K; e \rightarrow K'; e' \\
\text{R-FIELD} & K.H(l) = \{\text{proto}: l'; f: \overline{F}\} \\
& f := v \in \overline{F} \\
\text{R-CALL} & \text{resolveMethod}(H, l, m) = \{\text{proto}: \overline{S}\}; \text{return } e \\
& \Gamma \vdash K; e; \text{letif } [\overline{x_1}, \overline{x_2}]; e \rightarrow K; [\overline{v}/\overline{x_1}, \overline{v}/\overline{x_2}] e \\
\end{array}
\]

\[
\begin{array}{ll}
\text{R-LETIF} & c = \text{true } \Rightarrow i = 1 \\
& c = \text{false } \Rightarrow i = 2 \\
\end{array}
\]

Figure 4: Reduction Rules for FRSC (adapted from Safe TypeScript [27]) and IRSC

e \rightarrow \delta \) and the top-level program transformation judgment returns the net effect: \( P \mapsto P \mapsto \Delta \). Hence, the SSA transformation judgment for configurations becomes: \( K; E \Delta \rightarrow K; e \). We can now state our consistency theorem as:

**Theorem 1** (SSA Consistency). For configurations \( E \) and \( R \) and global store typing \( \Delta \), if \( R \Delta \rightarrow R \), then either both \( R \) and \( R \) are terminal, or if for some \( R' \), \( R \rightarrow R' \), then there exists \( R' \) such that \( R \rightarrow R' \) and \( R' \Delta \rightarrow R' \).

### 3.2 Static Semantics

Having drawn a connection between source and target language we can now describe refinement checking procedure in terms of IRSC.

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

**Logical Predicates** Predicates \( p \) are logical formulas over terms \( t \). These terms can be variables \( x \), primitive constants \( c \), the reserved value variable \( v \), the reserved variable this to denote the containing object, field accesses \( t.f \), uninter-
Typing Rules

<table>
<thead>
<tr>
<th>Rule</th>
<th>Context</th>
<th>Premise</th>
</tr>
</thead>
<tbody>
<tr>
<td>T-Var</td>
<td>( \Gamma (x) = T )</td>
<td>( \Gamma \vdash x : \text{self}(T, x) )</td>
</tr>
<tr>
<td>T-Cst</td>
<td>( \Gamma \vdash c : ty(c) )</td>
<td></td>
</tr>
<tr>
<td>T-Field-I</td>
<td>( \Gamma \vdash e : T ) ( \Gamma, z : T \vdash z \text{ hasImm } f_i : T_i )</td>
<td>( \Gamma \vdash e.f_i : \exists z : \text{self}(T, z, f_i) )</td>
</tr>
<tr>
<td>T-Ctx</td>
<td>( \Gamma \vdash U \vdash \pi : S )</td>
<td>( \Gamma, \pi : S \vdash e : T )</td>
</tr>
<tr>
<td>T-New</td>
<td>( \Gamma \vdash e : T ) ( \Gamma, z : T \vdash z \text{ hasMut } g_i : T_i )</td>
<td>( \Gamma \vdash e.g_i : \exists z : T, T_i )</td>
</tr>
<tr>
<td>T-Inv</td>
<td>( \Gamma \vdash e : T ) ( \Gamma, z : T \vdash z \text{ has } (\text{def m } (\varpi : R) { p : S = e }) )</td>
<td>( \Gamma \vdash e.m(\varpi) : \exists z : T, \exists \varpi : T, S )</td>
</tr>
<tr>
<td>T-Asgn</td>
<td>( \Gamma \vdash e_1 : T_1, e_2 : T_2 )</td>
<td>( \Gamma, z_1 : [T_1] \vdash z_1 \text{ hasMut } f : S, T_2 \leq S )</td>
</tr>
<tr>
<td>T-Field-M</td>
<td>( \Gamma \vdash e : T ) ( \Gamma, z : T \vdash z \text{ hasMut } g_i : T_i )</td>
<td>( \Gamma \vdash e.g_i : \exists z : T, T_i )</td>
</tr>
<tr>
<td>T-CtxEmp</td>
<td>( \Gamma \vdash )</td>
<td>( \Gamma \vdash e \text{ as } T : T )</td>
</tr>
<tr>
<td>T-LetIn</td>
<td>( \Gamma \vdash e : T )</td>
<td>( \Gamma \vdash )</td>
</tr>
<tr>
<td>T-LetIf</td>
<td>( \Gamma \vdash e : S ) ( \Gamma, z : S \vdash U_1 \vdash \Gamma_1 )</td>
<td>( \Gamma \vdash )</td>
</tr>
<tr>
<td></td>
<td>( \Gamma, \Gamma_2 \vdash \Gamma_2(\varpi_1) \leq T )</td>
<td>( \Gamma \vdash )</td>
</tr>
<tr>
<td></td>
<td>( \Gamma, z : S, \neg z \vdash U_2 \vdash \Gamma_2 )</td>
<td>( \Gamma \vdash U \vdash (T \text{ fresh}) )</td>
</tr>
</tbody>
</table>

Figure 5: Static Typing Rules for IRSC

interpreted function applications \( f(\bar{t}) \) and applications of terms on built-in operators \( b \), such as \( \preceq, <, + \), etc.

\[
T, S, R \ ::= \exists x : T_1, T_2 \{ \nu : N \mid p \} \\
N \ ::= C \mid B \\
p \ ::= p_1 \land p_2 \mid \neg p \mid t \\
t \ ::= x \mid c \mid \nu \mid \text{this} \mid t.f \mid f(\bar{t}) \mid b(\bar{t})
\]

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 \( C \) has \( f \) (that encodes inclusion of an element \( I \) in a class \( C \)) into two cases: \( C \) hasMut \( I \) and \( C \) hasImm \( I \), to account for elements that may be mutated. These elements can only be fields \( i.e \) there is no mutation on methods.

Environments And Well-formedness A type environment \( \Gamma \) 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 [19].

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: \( \text{Valid}(\Gamma) \Rightarrow [p] \Rightarrow [p'] \), where \( [\Gamma] \) is the embedding of environment \( \Gamma \) into our logic that accounts for both guard predicates and variable bindings:

\[
[\Gamma] = \bigwedge \{ p \mid p \in \Gamma \} \land \bigwedge \{ \{x/\nu \mid p, x : \{\nu : N \mid p \}} \in \Gamma \}
\]

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 Figure 5 contains most rules of the two forms of our typing judgements: \( \Gamma \vdash e : T \) and \( \Gamma \vdash U \vdash \Gamma' \). The first form assigns a type \( T \) to an expression \( e \) under a typing environment \( \Gamma \). The second form checks the body of an SSA context \( U \) under \( \Gamma \) and returns an environment \( \Gamma' \) of the variables introduced in \( U \) that are going to be available in its hole. Below, we discuss the novel rules:

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

\[
\{\nu : N \mid p \} \oplus p' = \{\nu : N \mid p \land p' \} \\
(\exists x : S, T) \oplus p = \exists x : S, (T \land p) \\
\text{self}(T, t) = T \land (\nu = t)
\]

[T-Field-M] Here we avoid such strengthening, as the value of field \( g_i \) is mutable, so cannot appear in refinements.
[T-NEW] Similarly, only immutables 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 $\Gamma$ (we write $\Gamma \vdash S \triangleleft T$), iff $(S \triangleright \{T\}) = R \neq \text{fail}$ with $\Gamma \vdash R \leq T$.

Here, $\{T\}$ extracts the base type of $T$, and $\{T \triangleright X\}$ succeeds when under environment $\Gamma$ we can statically prove $D$'s invariants. We use the predicate $\text{inv}(D, \nu)$ (as in CFJ), to denote the conjunction of the class invariants of $C$ and its supertypes (with the necessary substitutions of this by $\nu$). We assume that part of these invariants is a predicate that states inclusion in the specific class (instanceof $(\nu, D)$). Therefore, we can prove that $T$ can safely be cast to $D$. Formally:

$$\langle \nu: C \mid p \rangle \triangleright D \equiv \begin{cases} D \land p \quad \text{if $[\Gamma] \Rightarrow [p] \Rightarrow \text{inv}(D, \nu)$} \\ \text{fail} \quad \text{otherwise} \end{cases}$$

$$\langle \exists x: S. T \triangleright X \rangle \triangleright D \equiv \exists x: S. \langle T \triangleright \frac{\Gamma, x: S \vdash D}{D} \rangle$$

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

[T-LET] To type conditional structures, we first infer a type for the condition and then check each of the branches $U_1$ and $U_2$, assuming that the condition is true or false, respectively, to achieve path sensitivity. Each branch assigns types to the $\Phi$-variables which compose $\Gamma_1$ and $\Gamma_2$, and the propagated types for these variables are fresh types operating as upper bounds to their respective bindings in $\Gamma_1$ and $\Gamma_2$.

### 3.3 Type Soundness

We reuse the operational semantics for IRSC defined earlier, and extend our type checking judgment to runtime locations $l$ with the use of a heap typing $\Sigma$, mapping locations to types:

$$\Sigma(l) = T \quad \Gamma; \Sigma \vdash l : T$$

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 2 (Subject Reduction).** If (a) $\Gamma; \Sigma \vdash e : T$, (b) $\Gamma; K \vdash_{\Sigma} H$, and (c) $\nu; H; e \rightarrow H'; e'$, then for some $T'$ and $\Sigma' \supseteq \Sigma$: (i) $\Gamma; \Sigma' \vdash e' : T'$, (ii) $\Gamma; T' \leq T$, and (iii) $\Gamma; K \vdash_{\Sigma'} H'$.

**Theorem 3 (Progress).** If $\Gamma; \Sigma \vdash e : T$ and $\Gamma; K \vdash_{\Sigma} H$, then either $e$ is a value, or there exist $e'$, $H'$ and $\Sigma' \supseteq \Sigma$ s.t. $\Gamma; K \vdash_{\Sigma'} H'$ and $H; e \rightarrow H'; e'$.

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

With the use of our Consistency Theorem (Theorem 1) and extending our checking judgment for terms in IRSC to runtime configurations ($\vdash R$), we can state a soundness result for FRSC:

**Theorem 5. (FRSC Type Safety)** If $R \xrightarrow{\Delta} R$ and $\vdash R$ then either $R$ is a terminal form, or there exists $R$ s.t. $R \rightarrow R$.

### 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:

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

is equivalent to a class `PointC` defined as:

```typescript
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 PointC; // returns false
pC instanceof PointC; // returns true
```

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

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

```javascript
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) performing checks when non-null (non-undefined) types are required (e.g. during field accesses), (2) using the correct variance for functions and constructors, (3) checking overloads via two-phase typing §2.1.2, and, (4) eliminating the any type.

Many uses of any (indeed, all uses, in our benchmarks §5) can be replaced with a combination of union or intersection types or downcasting, all of which are soundly checked via path-sensitive refinements. In future work, we wish to support the full language, namely allow dynamically checked uses of any by incorporating orthogonal dynamic techniques from the contracts literature. We envisage a dynamic cast operation cast \(_T\) :: \(x:v\) \(\Rightarrow\) \(\{v:T | v = x\}\). It is straightforward to implement cast \(_T\) for first-order types \(T\) as a dynamic check that traverses the value, testing that its components satisfy the refinements [30]. Wrapper-based techniques from the contracts/gradual typing literature should then let us support higher-order types.

### 4.2 Reflection

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

```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 \((z:A) \Rightarrow \{v:string | v = ttag(z)\}\) 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 [4].

**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 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 = 0x00000000, Interface= 0x00000000, Reference= 0x00000100, 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 \(0x00000000\) 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 type flag with some other flags.

**Specifying Hierarchies with Refinements** rsc allows developers to create and use Type objects with the above invariant by specifying a predicate typeInv [5]:

```javascript
isMask<v,m,t> = mask(v,m) \Rightarrow impl(this.t)
typeInv<v> = isMask<v, 0x00000001, Any>
\& isMask<v, 0x00000002, String>
\& isMask<v, 0x00003C00, 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.

**Verifying Downcasts** rsc verifies the code that uses ad-hoc hierarchies such as the above by proving the TS downcast

```
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.
```
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 = <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>` is treated as a call to a function with signature:

```javascript
(x: A) => (y: ObjectType) => {v: ObjectType | v = x}
```

The if-test ensures that the `immutable` field `t.flags` masked with `@0x00000000` is non-zero, satisfying the third line in the type definition of `typeInvariants`, which, in turn implies that `t` in fact implements the `ObjectType` interface.

### 4.4 Imperative Features

**Immutability Guarantees** Our system uses ideas from Immutability Generic Java [41] (IGJ) to provide statically checked immutability guarantees. In IGJ a type reference is of the form `C<M,T>`, where `immutability` argument `M` works as proxy for the immutability modifiers of the contained fields (unless overridden). It can be one of: Immutable (or IM), when neither this reference nor any other reference can mutate the referenced object; Mutatable (or MU), when this and potentially other references can mutate the object; and ReadOnly (or RO), when this reference cannot mutate the object, but some other reference may. Similar reasoning holds for method annotations. IGJ provides deep immutability, since a class’s immutability parameter is (by default) reused for its fields; however, this is not a firm restriction imposed by refinement type checking.

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

```typescript
interface Array<K extends Readonly, T> {
  @Mutable pop(): T;
  @Mutable push(x: T): number;
  @Immutable get length(): {nat|v=1len(this)}
  @ReadOnly get length(): nat;
  [..]
}
```

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

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

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

**Object initialization** Our formal core (§3) treats constructor bodies in a very limiting way: object construction is merely an assignment of the constructor arguments to the fields of the newly created object. In `rsc` we relax this restriction in two ways: (a) We allow class and field invariants to be violated `within` the body of the constructor, but checked for at the exit. (b) We permit the common idiom of certain fields being initialized `outside` the constructor, via an additional mutability variant that encodes reference `uniqueness`. In both cases, we still restrict constructor code so that it does not `leak` references of the constructed object (this) or `read` any of its fields, as they might still be in an uninitializable 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.\_\_\_\_f\_\_\_i \leftarrow e_i, with \_\_\_\_f\_\_\_i = e_i, where \_\_\_\_f\_\_\_i are local variables, and (b) all return points with a call `ctor_init` \_\_\_\_\_f\_\_\_i = (\_\_\_\_f\_\_\_i), where the signature for `ctor_init` is: `(f: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`):

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

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

- they cannot be re-assigned,
- they cannot be generally referenced, unless this occurs at a context that guarantees that no aliases will be produced, e.g. the context of `e1` in `e1.f = e2`, or the context of a returned expression, and
they cannot be cast to types of a different mutability (e.g. \textless C\textless 1M\textgreater \textgreater \times), as this would allow the same reference to be subsequently aliased.

More expressive initialization approaches are discussed in §6.

5. Evaluation

To evaluate \texttt{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 \texttt{rsc} the following:

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

5.1 Benchmarks

We took a number of existing JS or TS programs and ported them to \texttt{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 [27], the TS compiler [22], and the D3 [4] and Transducers libraries [7]:

\begin{itemize}
\item \texttt{navier-stokes} which simulates two-dimensional fluid motion over time; \texttt{richards}, which simulates a process scheduler with several types of processes passing information packets; \texttt{splay}, which implements the \texttt{splay tree} data structure; and \texttt{raytrace}, which implements a raytracer that renders scenes involving multiple lights and objects; all from the Octane suite,
\item \texttt{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,
\item \texttt{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,
\end{itemize}

<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>36</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>TOTAL</td>
<td>2522</td>
<td>334</td>
<td>104</td>
<td>91</td>
<td></td>
</tr>
</tbody>
</table>

Figure 6: LOC is the number of non-comment lines of source (computed via \texttt{cloc v1.62}). The number of RSC specifications given as JML style comments is partitioned into \(T\) trivial annotations \textit{i.e.} TypeScript type signatures, \(M\) mutability annotations, and \(R\) refinement annotations, \textit{i.e.} those which actually mention invariants. \textbf{Time} is the number of seconds taken to analyze each file.

\begin{itemize}
\item \texttt{tsc-checker} which includes parts of the TS compiler (v1.0.1.0), abbreviated as tsc. We check 15 functions from \texttt{compiler/core.ts} and 14 functions from \texttt{compiler/checker.ts} (for which we needed to import 779 lines of type definitions from \texttt{compiler/types.ts}). These code segments were selected among tens of thousands of lines of code comprising the compiler codebase, as they exemplified interesting properties, like the bit-vector based type hierarchies explained in §4.3.
\end{itemize}

\textbf{Results} Figure 6 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 \textit{trivial}, \textit{i.e.} are TS-like types of the form \((x:\texttt{nat}) \Rightarrow \texttt{nat}\); 20% (104/529) are trivial but have \textit{mutability} information, and only 17% (91/529) mention refinements, \textit{i.e.} are definitions like \texttt{type \texttt{nat} = \{v:number|0\leq v\} or dependent signatures like \langle a:T []],n:idx<a|\rangle\Rightarrow T}. These numbers show \texttt{rsc} has annotation overhead comparable with TS, as in 83% cases the annotations are either identical to TS annotations or to TS annotations with some mutability modifiers. Of course, in the remaining 17% cases, the signatures are more complex than the (non-refined) TS version.

\textbf{Code Changes} We had to modify the source in various small (but important) ways in order to facilitate verification. The total number of changes is summarized in Figure 7. The \textit{trivial} changes include the addition of type annotations (accounted for above), and simple transforms to work around current limitations of our front end, \textit{e.g.} converting \texttt{x++} to \(x = x + 1\). The \textit{important} classes of changes are the following:

\begin{itemize}
\item \textbf{Control-Flow:} Some programs had to be restructured to work around \texttt{rsc}'s currently limited support for certain control flow structures (\textit{e.g.} \texttt{break}). We also modified some loops to use explicit termination conditions.
\end{itemize}
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>LOC</th>
<th>ImpDiff</th>
<th>AllDiff</th>
</tr>
</thead>
<tbody>
<tr>
<td>navier-stokes</td>
<td>366</td>
<td>79</td>
<td>160</td>
</tr>
<tr>
<td>splay</td>
<td>206</td>
<td>58</td>
<td>64</td>
</tr>
<tr>
<td>richards</td>
<td>304</td>
<td>52</td>
<td>108</td>
</tr>
<tr>
<td>raytrace</td>
<td>576</td>
<td>93</td>
<td>145</td>
</tr>
<tr>
<td>transducers</td>
<td>588</td>
<td>170</td>
<td>418</td>
</tr>
<tr>
<td>d3-arrays</td>
<td>189</td>
<td>8</td>
<td>110</td>
</tr>
<tr>
<td>tsc-checker</td>
<td>293</td>
<td>TODO</td>
<td>TODO</td>
</tr>
<tr>
<td><strong>TOTAL</strong></td>
<td><strong>2522</strong></td>
<td>TODO</td>
<td>TODO</td>
</tr>
</tbody>
</table>

Figure 7: LOC is the number of non-comment lines of source (computed via cloc v1.62). The number of lines at which code was changed, which is counted as either: ImpDiff: the important changes that require restructuring the original JavaScript code to account for limited support for control flow constructs, to replace records with classes and constructors, and to add ghost functions, or, AllDiff: the above plus trivial changes due to the addition of plain or refined type annotations (Figure 6), and simple edits to work around current limitations of our front end.

- **Classes and 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 classes and 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). An example of such a function is:

```java
/*@ mulThm1 :: (a:nat, b:(number | b ≥ 2)) ⇒ (boolean | a + a ≤ a * b) */
```

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

5.2 Transducers (A Case Study)

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

The main reason this library interests us is because some of its functions are massively overloaded. Consider, for example, the `reduce` function it defines. As discussed above, `reduce` needs a Transformer and a collection. There are two opportunities for overloading here. First of all, the main ways that a Transformer is more general than a simple step function is that it can be stateful and that it defines the result post-processing step. Most of the time the user does not need these features, in which case their Transformer is just a wrapper around a step function. Thus for convenience, the user is allowed to pass in either a full-fledged Transformer or a step function which will automatically get wrapped into one. Secondly, the collection being reduced can be a stunning array of options: an Array, a string (i.e. a collection of characters, which are themselves just strings), an arbitrary object (i.e., in JS, a collection of key-value pairs), an iterator (an object that defines a next function that iterates through the collection), or an iterable (an object that defines an iterator function that returns an iterator). Each of these collections needs to be dispatched to a type-specific reduce function that knows how to iterate over that kind of collection. In each overload, the type of the collection must match the type of the Transformer or step function. Thus our `reduce` begins as shown in Figure 8:

If you count all 5 types of collection and the 2 options for step function vs Transformer, this function has 10 distinct

3 For simplicity of discussion we will henceforth ignore `init` and initialization in general, as well as some other details.
overloads! Another similar function offers 5 choices of input collection and 3 choices of output collection for a total of 15 distinct overloads.

5.3 Unhandled Cases
This section outlines some cases that RSC fails to handle and explains the reasons behind them.

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

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

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

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

```javascript
function distinct<T>(a: T[]): T[] {
  var result: T[] = [];
  for (var i = 0, n = a.length; i < n; i++) {
    var current = a[i];
    for (var j = 0; j < result.length; j++) {
      if (result[j] === current) {
        break;
      }
    }
    if (j === result.length) {
      result.push(current);
    }
  }
  return result;
}
```

Checking the function body for the second overload (line 2) is problematic: without a user type annotation on `res`, the inferred type after joining the environments of each conditional branch will be `res: B + (A + undefined)` (as `res` is collecting values from `x` and `a[0]`, at lines 7 and 10, respectively), instead of the intended `res: B`. This causes an error when `res` is passed to function `f` at line 14, expected to have type `B`, which cannot be overcome even with refinement checking, since this code is no longer executed under the check on the length of the `arguments` variable (line 6). A solution to this issue would be for the user to annotate the type of `res` as `B` at its definition at line 5, but only for the specific (second) overload. The assignment at line 10 will be invalid, but this is acceptable since that branch is provably (by the refinement checking phase [38]) dead. This option, however, is currently not available.

6. Related Work
RSC is related to several distinct lines of work.

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

Program Logics At the other extreme, one can encode types as formulas in a logic, and use SMT solvers for all the analysis (subtyping). DMinor explores this idea in a first-order functional language with type tests [2]. The idea can be scaled to higher-order languages by embedding the typing relation inside the logic [6]. DJS combines nested refinements with alias types [31], a restricted separation logic, to consider for example the following code, which is a variation of the `reduce` function presented in §2:

```javascript
/*@ <A> (a:A[], f:(A,A,idx<A>)⇒A) ⇒ A */
/*@ <A,B>(a:A[], f:(B,A,idx<A>)⇒B,x:B) ⇒ B */
function reduce(a, f, x) {
  var res, s;
  if (arguments.length === 3) {
    res = x;
    s = 0;
  } else {
    res = a[0];
    s = 1;
  }
  for (var i = s; i < a.length; i++)
    res = f(res, a[i], i);
  return res;
}
```

The `results` array is defined at line 2 so it is initially typed as `Array<UQ,T>`. At lines 5–9 it is iterated over, so in order to prove the access at line 6 safe, we need to treat `results` as an immutable array. However, later on at line 11 the code pushes an element onto `results`, an operation that requires a mutable receiver. Our system cannot handle the interleaving of these two kinds of operations that (in addition) appear in a tight loop (lines 3–13). The alias tracking section in the related work (§6) includes approaches that could allow support for such cases.

Annotations per Function Overload A weakness of RSC, that stems from the use of Two-Phased Typing [38] in handling intersection types, is cases where type checking requires annotations under a specific signature overload. Consider for example the following code, which is a variation of the `reduce` function presented in §2:
account for aliasing and flow-sensitive heap updates to obtain a static type system for a large portion of JS [5]. DJJS 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 [41] and two-phase typing lets $rsc$ use liquid refinement inference [28], yielding a system that is more practical for real world programs. Extended Static Checking [9] uses Floyd-Hoare style first-order contracts (pre-, post-conditions and loop invariants) to generate verification conditions discharged by an SMT solver. Refinement types can be viewed as a generalization of Floyd-Hoare logics that uses types to compositionally account for polymorphic higher-order functions and containers that are ubiquitous in modern languages like TS.

X10 [25] is a language that extends an object-oriented type system with constraints on the immutable state of classes. Compared to X10, in RSC: (a) we make mutability parametric [41], and extend the refinement system accordingly, (b) we crucially obtain flow-sensitivity via SSA transformation, and path-sensitivity by incorporating branch conditions, (c) we account for reflection by encoding tags in refinements and two-phase typing [38], and (d) our design ensures that we can use liquid type inference [28] to automatically synthesize refinements.

Analyzing TypeScript Feldthaus et al. present a hybrid analysis to find discrepancies between TS interfaces [40] and their JS implementations [8], and Rastogi et al. extend TS with an efficient gradual type system that mitigates the unsoundness of TS’s type system [27].

Object and Reference Immutability $rsc$ builds on existing methods for statically enforcing immutability. In particular, we build on Immutability Generic Java (IGJ) which encodes object and reference immutability using Java generics [41]. Subsequent work extends these ideas to allow (1) richer ownership patterns for creating immutable cyclic structures [42], (2) unique references, and ways to recover immutability after violating uniqueness, without requiring an alias analysis [13].

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


