

Formalizing, Mechanizing, and Verifying Class-based Refinement Types

Anonymous Authors 

Abstract

Refinement types have been extensively used in class-based languages to specify and verify fine-grained logical specifications. Despite the advances in practical aspects such as applicability and usability, two fundamental issues persist. First, the soundness of existing class-based refinement type systems is inadequately explored, casting doubts on their reliability. Second, the expressiveness of existing systems is limited, restricting the depiction of semantic properties related to object-oriented constructs. This work tackles these issues through a systematic framework. We formalize a declarative class-based refinement type calculus that is expressive and concise. We rigorously develop the soundness meta-theory of this calculus, followed by its mechanization in Coq. Finally, we propose an SMT-based verification approach, ensuring the calculus's verifiability.

2012 ACM Subject Classification Theory of computation → Type structures; Software and its engineering → Formal software verification

Keywords and phrases Refinement Types, Program Verification, Object-oriented Programming

Digital Object Identifier 10.4230/LIPIcs.CVIT.2016.23

1 Introduction

Refinement types have been widely used in class-based languages [44, 61, 10, 52, 24, 30, 33] to enhance the capabilities of traditional type systems, allowing for more precise safety guarantees. These types extend basic data types (e.g., integer type, boolean type, and class types) with logical constraints that specify detailed conditions on the data. For example, $\{\nu : C \mid \nu.f > 0\}$ characterizes instances of type C with the property that their f field exceeds zero. The logical constraint (e.g., $\nu.f > 0$) is often called the *refinement* of the type.

Despite the advancements in various practical aspects (briefly surveyed in Section 8), a comprehensive examination of the fundamental aspects of class-based refinement types remains elusive. The primary reason stems from the intricate logical interpretation associated with refinements, which determines their meanings. In existing class-based refinement type systems, the logical interpretations are often defined via the Satisfiability Modulo Theories (SMT) relation [9, 32], since they are typically analyzed algorithmically via SMT solvers. Although this interpretation is closer to the actual algorithmic interpretation, it brings two crucial problems. Firstly, the **soundness** of the type system is difficult to define and argue formally, since it depends on the SMT relation, which is both complex and intricate to define properly. This complexity has led to a paucity of mechanized soundness proofs in previous systems, putting their reliability in doubt. Secondly, the **expressiveness** of the refinement types is limited by the need to adhere to decidable theory combinations (e.g., QF-EUFLIA [9]), which impedes the representation of semantic properties relevant to user-defined classes and methods.

To address those fundamental issues, this work makes three consecutive contributions.

1. Formalization To formalize a foundational calculus, this paper introduces Refinement Featherweight Java (RFJ), an FJ-like [28] calculus with expressive refinements capable of stating arbitrary properties about user-defined elements. Our basic methodology is to construct a declarative, SMT-independent logical interpretation within the language, which greatly increases the expressiveness and benefits the meta-theoretical development.



© Jane Open Access and Joan R. Public;
licensed under Creative Commons License CC-BY 4.0

42nd Conference on Very Important Topics (CVIT 2016).

Editors: John Q. Open and Joan R. Access; Article No. 23; pp. 23:1–23:30

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Apart from that, RFJ is equipped with several features important for refinement-type-based verification, yet mostly absent from previous systems, such as interfaces for decomposing proof obligations among subclasses (c.f., Section 2.2), general selfication—a typing mechanism [60, 47] (detailed in Section 2.1) that seamlessly integrates accurate term information into refinements, and flexible method overriding through co/contra-variance [11]. Besides those critical features, RFJ closely mirrors FJ, avoiding the usage of non-standard judgments (e.g., object constraint systems [44, 61]) and non-standard constructs (e.g., ANF [10], existential types [61]). Thus, we believe RFJ can be an ideal base to explore further extensions.

2. Mechanization The soundness properties of RFJ are rigorously established and mechanized in Coq. Although leveraging an in-language logical interpretation reduces the proof difficulty, the proof is still challenging and requires non-standard techniques. For example, we make a novel use of big-step semantics to obtain a convenient induction principle for proving the preservation lemma under arbitrary type substitution. We introduce a novel approach to establish the logical soundness property (one major soundness property of RFJ), as the standard logical relation technique [56] is ineffective for first-order languages like RFJ [51].

3. Verification The expressive refinements provided by RFJ fall out of the scope of existing SMT theories, casting ambiguity on the system’s algorithmic verifiability. We address this concern by proposing an algorithmic verification approach based on a fragment (termed LFJ) of order-sorted first-order logic (OS-FOL) [53]. In particular, we define a type-directed translation from RFJ to the LFJ. We define an intended model of LFJ and map the RFJ refinement subtyping problem to the LFJ validity problem under this model. We devise an axiomatization of the intended model covering the semantics of RFJ programs. The axiomatization can be used by SMT solvers to perform algorithmic verification. Thus, the expressive refinements of RFJ are not only meta-theoretical constructs: they are amenable to algorithmic analysis within SMT solvers. Additionally, we develop a refinement type checker that leverages Z3 SMT solver [18] for checking the validity of LFJ formulas. The type checker is evaluated against a small yet representative benchmark derived from a Java textbook [22] and prior systems [59].

In the remainder of this paper, we detail our contributions. Section 2 provides an overview. Sections 3, 4, and 5 each describe one of the three contributions. Section 6 discusses the mechanization and implementation. Sections 7, 8, and 9 review related work and conclude.

The accompanying code of this paper, including the meta-theory mechanization and type checker implementation, is available at <https://zenodo.org/records/10990919>.

2 Overview

This section serves as an overview of the whole paper. We start with an example program to demonstrate the expressiveness and features of RFJ. Then, we discuss the actual verification through LFJ. At last, we turn back to the meta-theory of RFJ and the challenges of developing the meta-theory. The sequence of discussion—starting with verification before addressing meta-theoretical concerns—is intentionally chosen to contrast with the presentation order in subsequent sections, aiming to enhance comprehension by familiarizing readers with the system through its verification aspects first.

2.1 RFJ by Example

In this section, we illustrate RFJ using a program extended from a textbook example [22] (we add some methods to make it more interesting). The program models various pizzas and

```

1  class Pizza{
2      {v:int|v>0} price(){return 1;}
3      Pizza remA(){return new Pizza();}
4      Pizza sell (this.price()>5){return this;}}
5  class Crust extends Pizza{
6      {v:int|v>0} price(){return 1;}
7      Pizza remA(){return new Crust();}
8      Pizza sell (){return this;}}
9  class Cheese extends Pizza{
10     p:Pizza
11     {v:int|v>0} price(){return let pp = this.p.price() in pp + 1;}
12     Pizza remA(){return new Cheese(this.p.remA());}}
13  class Anchovy extends Pizza{
14     p:Pizza
15     {v:int|v>0 && v>=this.p.price()} price(){return let pp = this.p.price() in pp;}
16     Pizza remA(){return this.p.remA();}}
17  class MagicAnchovy extends Anchovy{
18     {v:int|v>0 && v>this.p.price()} price(){return let pp = this.p.price() in pp + 1;}}
19  class Main{
20     int assertSingleCheesePizza(x: {v:Pizza|v = new Cheese(new Crust())}){
21         return 0; }
22     int testRemA(){
23         return let p1 = new Anchovy(new Cheese(new Crust())) in
24             this.assertSingleCheesePizza(p1.remA());}}

```

■ **Figure 1** An example RFJ program. In refinements, v stands for ν .

three operations on them: computing the price of a pizza (the `price` method), removing all anchovies from a pizza (the `remA` method), and selling a pizza (the `sell` method).

Simple Verification Our initial focus is a basic property: the price of any pizza must be positive. To enforce this property, we refine the return types of `price` methods with a refinement $\nu > 0$, where ν denotes the value being refined. RFJ's refinement subtyping mechanism guarantees that the price methods indeed return positive values. Pick `Pizza.price` for an example, RFJ enforces the following subtyping constraint for the return type:

$$this : \{\nu : Pizza | true\} \vdash \{\nu : int | \nu = 1\} <: \{\nu : int | \nu > 0\} \quad (1)$$

In refinement type systems like RFJ, such subtyping constraints have logical interpretations. In particular, Constraint (1) requires all ν satisfying $\nu = 1$ must also satisfy $\nu > 0$, which holds under RFJ logical interpretation (formally defined in Section 3.3). Note that the subtyping constraint is checked within a specific type environment $this : \{\nu : Pizza | true\}$, which contains the types of all visible variables. Those variables may be referred to by the refinement types, as demonstrated in the following example.

Method Override RFJ supports overriding methods in subclasses. For example, `Cheese.price` overrides `Pizza.price` to provide a different price computation. To preserve the logical property, the return type must still be validated, yielding the following constraint:

$$this : \{\nu : Cheese | true\}, pp : \{\nu : int | \nu > 0\} \vdash \{\nu : int | \nu = pp + 1\} <: \{\nu : int | \nu > 0\} \quad (2)$$

The `pp` item in the environment is introduced by the `let` binding. Since `pp` is bound to `this.p.price()`, RFJ sets its type as the type of `this.p.price()`, which is $\{\nu : int | \nu > 0\}$.

Refinement for `this` and Override with Co/contra-variance In RFJ, every method has an implicit `this` parameter with the same type as the enclosing class of this method

(e.g., in `Cheese.price()`, `this` has `Cheese` type). We have seen `this` appearing in previous subtyping constraints, but with a trivial refinement `true`. `this` can also be given a non-trivial refinement to ensure that methods are invoked on objects satisfying specific criteria. For instance, `Pizza.sell` includes a refinement at the first place of the parameter list (marked `cyan`), specifying that only the pizza whose `price` is greater than 5 can be sold.

Meanwhile, suppose that a `Crust` can also be sold regardless of its price. This can be achieved by **overriding** the method `sell` in `Crust`, as the example shows. In the overriding method, the refinement of `this` is `true` and thus omitted, making it a supertype of the previous refinement `this.price > 5`, obeying **contra-variance** of parameter types¹.

Now, consider a property for `Anchovy.price()`: the price is not only positive, but also not less than that of `this.p`. This extra property is marked `olive` in the program. The property makes the new return type a subtype of the old, obeying return type **co-variance**.

General Selfication Checking `Anchovy.price()`'s return type yields this constraint:

$$this : Anchovy, pp : \{int | \nu > 0\} \vdash \{int | \nu = pp\} <: \{int | \nu > 0 \& \nu \geq this.p.price()\} \quad (3)$$

Here, we omit the refinement binder ν and the refinement when it is trivial (i.e., `true`). However, this constraint can not be proved currently, essentially due to the loss of the `this.p.price()` term information in the type of `pp`. Luckily, RFJ's general selfication² mechanism addresses this by ensuring the persistence of such information. In a nutshell, it works by equating the term being typed to the refinement of its type, giving `this.p.price() : \{ \nu : int | \nu > 0 \& \nu = this.p.price() \}`, which is also the type of `pp`. The strengthened type of `pp` lets the constraint be proved. With the same technique, we can prove the validity of `MagicAnchovy.price`, which further overrides `Anchovy.price`.

Referring to Methods Next, we turn to the `remA` methods for removing all anchovies from a pizza. Consider the method `testRemA`, where we assess the correctness of the `remA` implementations. For the assertion in Line 24, RFJ enforces the subtyping constraint below:

$$p1 : \{An | \nu = An(Ch(Cr()))\} \vdash \{Pi | \nu = p1.remA()\} <: \{Pi | \nu = Ch(Cr())\} \quad (4)$$

We omit `this` from the type environment, which does not affect the meaning of this subtyping constraint. Meanwhile, we abbreviate class names to their initial two letters (e.g., `Ch` represents `Cheese`), and omit the `new` keyword (e.g., `An(Ch(Cr()))` represents `new An(new Ch(new Cr()))`), in order to save space. Proving Constraint (4) demands intricate reasoning about the program's semantics, particularly the semantics of the `remA` methods. This contrasts with the previous example, where no specific knowledge about the `price` methods is required. In our meta-theoretical calculus, since the logical interpretation is built upon the program semantics, Constraint (4) does not pose a significant challenge. Nevertheless, facilitating its efficient handling within SMT solvers requires a theory about RFJ program semantics, which is discussed in detail in Section 2.2.

Proving with Interfaces At last, we consider a more interesting property concerning `price` and `remA`: the price of a pizza should not increase after removing all anchovies. One way to express this is to append the following method to `Pizza`: `\{bool | this.price() >= this.remA().price()\}` `remA_noinc_price()\{return true;\}`, yielding the subtyping constraint below:

$$this : Pizza \vdash \{bool | \nu = true\} <: \{bool | this.price() \geq this.remA().price()\} \quad (5)$$

¹ Strictly speaking, for the type of `this`, we use co-variance for the base type and contra-variance for the refinement, c.f. Section 3.1.

² We name it *general* to distinguish from the cases like [31], where selfication is only used for variables.

152 This property holds in our meta-theoretical calculus. However, it breaks the proof modularity
 153 and is not verifiable in the algorithmic verification, even with the theory extended with RFJ
 154 program semantics. The mitigation of this challenge is facilitated by another key feature of
 155 RFJ: interfaces. We discuss that in detail in the following section.

156 2.2 Algorithmic Verification

157 In conventional refinement type systems, subtyping constraints are typically dispatched to
 158 SMT solvers, which facilitates automated reasoning and significantly reduces implementation
 159 efforts. We utilize this methodology by providing a logical encoding of RFJ to a dedicated
 160 OS-FOL named LFJ. A detailed exposition of LFJ is provided in Section 5. Here, we offer a
 161 concise overview of it, drawing upon the examples discussed in Section 2.1.

162 **EUFLIA** After being encoded to LFJ, the subtyping constraints (1), (2), and (3) fall into
 163 the theory of Equality, Uninterpreted Functions, and Linear Integer Arithmetic (EUFLIA), a
 164 domain widely supported by contemporary SMT solvers [9, 5, 18]. LFJ incorporates EUFLIA
 165 for verifying those constraints.

Theory $\mathcal{T}_{\mathcal{J}}$ We have illustrated in Section 2.1 that the verification of Constraint (4) requires knowledge about program semantics. The knowledge is encoded in theory $\mathcal{T}_{\mathcal{J}}$, an order-sorted first-order theory about RFJ semantics. We illustrate $\mathcal{T}_{\mathcal{J}}$ utilizing Constraint (4), which is translated to the LFJ formula below:

$$\forall p1 : An, \nu : Pi. p1 = An_{cr}(Ch_{cr}(Cr_{cr}())) \wedge \nu = An_{remA}(p1) \Rightarrow \nu = Ch_{cr}(Cr_{cr}())$$

166 Here, An_{cr} represents the constructor for **Anchovy**, while An_{remA} represents the *conditional*
 167 *function* composed of the possible *denotations* of **Anchovy.remA**. We give a detailed explanation
 168 of *conditional function* and its necessity in Section 5.2. For now, just note that this special
 169 model causes a restriction to our theory: only *final* methods (methods not overridden in
 170 subclasses) can be precisely characterized in $\mathcal{T}_{\mathcal{J}}$. Luckily, An_{remA} is indeed a *final* method.

For final methods such as An_{remA} , $\mathcal{T}_{\mathcal{J}}$ contains axioms characterizing their semantics:

$$\forall this : An. An_{remA}(this) = Pi_{remA}(An_p(this)) \quad (ax\ 1)$$

The RHS of the equation is a translation of the method body of **Anchovy.remA**, with An_p representing the accessor of the p field. By (ax 1) and the equality axioms, we can deduce that $\nu = Pi_{remA}(An_p(An_{cr}(Ch_{cr}(Cr_{cr}()))))$, which can be handled by an axiom for An_p :

$$\forall p : Pi. An_p(An_{cr}(p)) = p \quad (ax\ 2)$$

171 letting us deduce $\nu = Pi_{remA}(Ch_{cr}(Cr_{cr}()))$. However, here we get stuck again: **Pizza.remA**
 172 is not *final* and can not be included in $\mathcal{T}_{\mathcal{J}}$, how can we reason about its semantics? The key
 173 observation lies in the *override axiom*:

$$\forall p : Ch. Pi_{remA}(p) = Ch_{remA}(p) \quad (ax\ 3)$$

174 *ax 3* asserts the equivalence between Pi_{remA} and Ch_{remA} given the parameter is a *Cheese*.
 175 In other words, Pi_{remA} directs to Ch_{remA} when the receiving object belongs to Ch . With
 176 *ax 3* in hand, we can deduce $\nu = Ch_{remA}(Ch_{cr}(Cr_{cr}()))$, which finally lets us deduce
 177 $\nu = Ch_{cr}(Cr_{cr}())$ by the semantics of Ch_{remA} and Cr_{remA} (they are all final methods).

178 **Verifying with Interfaces** Even with $\mathcal{T}_{\mathcal{J}}$, Constraint (5) still can not be verified. In
 179 particular, it would be translated to $\forall this : Pi. Pi_{price}(this) \geq Pi_{price}(Pi_{remA}(this))$, whose

```

25 interface Pizza{
26     {int|v>0} price()
27     Pizza remA()
28     Pizza sell (this.price()>5)
29     {v:bool|this.price()>=this.remA().price()} remA_noinc_price() }
30 class Anchovy implements Pizza{
31     p:Pizza
32     {v:int|v>0 && v>=this.p.price()} price(){return let pp = this.p.price() in pp;}
33     Pizza remA(){return this.p.remA();}
34     {v:bool|this.price()>=this.remA().price()} remA_noinc_price(){
35         return let p1 = this.p.remA_noinc_price() in
36         let p2 = this.price() in true;}}

```

■ **Figure 2** Proving `remA_noinc_price` for `Anchovy`.

verification requires a precise characterization concerning Pi_{price} and Pi_{remA} , going beyond the ability of OS-FOL (we give a detailed explanation about this issue in Section 5.3). In such cases, we must utilize another feature of RFJ: interfaces. By making `Pizza` an interface, we eliminate the necessity of proving the property directly for `Pizza`. Instead, we decompose the proof obligation into subclasses implementing `Pizza`. To illustrate the idea, we pick the proof for `Anchovy` for an example, which is shown in Figure 2.

The first thing to note is that, the method body of `Anchovy.remA_noinc_price` is not trivial (e.g., `return true`). This means that the SMT solver can not prove this property automatically. Rather, we need to manually introduce two logical assumptions (`p1` and `p2`) to the proof context. `p1` is the “induction hypothesis” of this proof, asserting the `remA_noinc_price` property holds for `this.p` ($this.p.price() \geq this.p.remA().price()$). `p2` is the refinement of `this.price()` (note that the third equation comes from general selfication): $\nu > 0 \ \&\& \ \nu \geq this.p.price() \ \&\& \ \nu = this.price()$. Since `Anchovy.price` is not *final*, we do not have the axiom about its semantics. However, the refinement of `Anchovy.price` works like a “lemma” asserting its property, providing sufficient information for this proof. The third important assumption is about the semantics of `Anchovy.remA`: $this.remA() = this.p.remA()$. This is included in $\mathcal{T}_{\mathcal{J}}$ since `Anchovy.remA` is *final*. After those are all set, we can deduce $this.price() \geq this.p.price() \geq this.p.remA().price()$ by `p1` and `p2`, then $this.price() \geq this.remA().price()$ since $this.remA() = this.p.remA()$ (`p3`).

We need not prove the same property for `MagicAnchovy`, since it inherits the property from `Anchovy`. Using this approach, we could prove other properties about `remA`, such as no `Anchovy` exists after `remA`, and the idempotence of `remA`, all included in our test suite.

2.3 Meta-theoretical Arguments

As we have seen in the previous sections: the design of RFJ aims at expressiveness and ease of use. At the same time, this very desirable combination leads to a tricky meta-theory. One major contribution of this paper is to establish the meta-theory rigorously (i.e., in Coq). The detailed development of the meta-theory is given in Section 4. In this section, we briefly overview the soundness theorems, and several challenges in proving them.

2.3.1 Soundness Theorems

RFJ satisfies the following soundness properties:

Type soundness requires the type system to be able to preclude evaluation from being stuck. Formally, if a closed expression is well-typed ($\emptyset \vdash e : t$), then it would never be stuck in a state where it can not be evaluated and is not a value yet. Since type soundness is

typically guaranteed by the base type system (in our case, the FJ type system) already, the core of proving that in refinement type systems is to ensure the additional refinement type mechanisms (e.g., general selfcication) do not break the promise of the base type system.

Logical soundness requires the type system to infer only *true* refinements. Formally, if a typing judgment $(\Gamma \vdash e : \{w|p\})$ is made by the type system, the refinement formula p must be *true* whenever the conditions stipulated by Γ are fulfilled. Logical soundness serves as a complement to type soundness, going beyond the guarantee that the evaluations of well-typed programs would never get stuck by also ensuring that the evaluations of such programs adhere to the logical constraints specified by type refinements.

2.3.2 Challenges

Logical Interpretation Refinements are logical formulas and should be interpreted logically. For example, the subtyping relation between two refinement types $(\Gamma \vdash \{w|p\} <: \{w|q\})$ is defined as the truth of the implication $p \Rightarrow q$ under the assumption set Γ . While our approach to algorithmic verification leverages a translation of RFJ into first-order logic (c.f., Section 2.2), employing this logic directly for defining the logical interpretation can prove both intricate and unwieldy. Rather, we choose to define it *inside the language*, allowing the algorithmic verification approach to serve as an external algorithm that scrutinizes the intrinsic logical interpretation. Nevertheless, articulating a precise logical interpretation is still challenging due to complex typing mechanisms like interfaces and nominal subtyping.

Type Substitution Different from previous class-based refinement type calculus, RFJ uses type substitution instead of ANF [29] or existential types [31]. This increase its generality and usability (detailed in Section 7). However, this also increases its meta-theoretical complexity. One important property of type substitution is the invariant under substitution. Formally:

$$\Gamma \vdash [x \mapsto e]t <: [x \mapsto e']t \text{ and } [x \mapsto e']t <: [x \mapsto e]t \quad \text{if } e \rightsquigarrow e'$$

That is, it does not matter if we substitute the term before or after evaluation to a type: the two are coherent. Since our subtyping is defined via the logical interpretation, which is defined via the operational semantics, proving this property requires complicated lemmas asserting the invariant of operational semantics.

General Selfcication As we show previously, general selfcication can increase the precision of the verification by recording term information in refinements. However, together with this benefit comes non-standard meta-theories. General selfcication affects term substitution and type preservation lemmas intricately and several non-standard properties (e.g., exactness, $\Gamma \vdash e : t \text{ then } \Gamma \vdash e : \text{self}(t, e)$) of it have to be proved for proving those lemmas.

First Order Functions The logical relation technique [56] is frequently employed in prior research [8, 27] to establish the logical soundness theorem. However, this technique can not be applied to RFJ, since RFJ is a first-order language without explicit function abstraction but with recursive method definition. Thus, we do not have a strong enough induction principle about methods when performing induction on typing. This challenge is not unique to us and has been encountered in previous studies [58, 51]. However, the workaround adopted by these studies, which essentially inlines methods at call-sites, is incompatible with RFJ, since that requires particular type structures supporting the strong normalisation of derivation reduction, a property that RFJ lacks.

3 Declarative Calculus: RFJ

This section outlines the syntax, semantics, and typing rules of RFJ, built upon the classical calculus FJ extended with primitive data types (integers and booleans) and let bindings. The FJ parts follow closely the classical textbook presentation [48]. To delineate the extensions unique to RFJ, we highlight the extended features in *gray background*.

3.1 Syntax and Lookup Functions

Syntax	Sub-nominal	Base-subtyping	Subtyping
$\mathcal{C} ::=$ <i>class definitions:</i> <i>class C extends D implements $\bar{I}\{\bar{t} \ \bar{f}; \ K \ \bar{M}\}$</i> $\mathcal{I} ::=$ <i>interface $I\{\bar{Q}\}$ interface Defs.</i> $K ::= C(\bar{t} \ \bar{f}) \{super(\bar{f}); \ this.\bar{f} = \bar{f}; \}$ $Q ::= t \ m(p, t \ x)$ <i>method Defs.</i> $M ::= Q\{return \ e; \}$ <i>method Defs.</i> $e, p, q ::=$ <i>terms:</i> x <i>variable</i> $e.f$ <i>field access</i> $e.m(e)$ <i>method invocation</i> $new \ C(\bar{e})$ <i>instance creation</i> n <i>integer</i> b <i>boolean</i> $\neg e$ <i>unary operation</i> $e \oplus e$ <i>binary operation</i> $let \ x = e \ in \ e$ <i>let binding</i> $\oplus ::= = \mid \vee \mid \wedge$ <i>binary operators</i> $v ::= n \mid b \mid new \ C(\bar{v})$ <i>values</i> $N ::= C \mid \mathbf{I}$ <i>nominal types</i> $w, u ::= \top \mid int \mid bool \mid N$ <i>base types</i> $s, t, r ::= \{\nu : w \mid p\}$ <i>refinement types</i>	$N_1 <:_n N_2$ $N <:_n N$ $\frac{N_1 <:_n N_2 \quad N_2 <:_n N_3}{N_1 <:_n N_3}$ $\frac{CT(C) = \text{class } C \text{ extends } D \dots\{\dots\}}{C <:_n D}$ $CT(C) = \text{class } C \dots \text{implements } \bar{I}\{\dots\}$ $C <:_n I_i$ $w <:_b \top$ $int <:_b int$ $bool <:_b bool$ $\frac{N_1 <:_n N_2}{N_1 <:_b N_2}$ $\frac{w <:_b u \quad \Gamma, \nu : w \models p \Rightarrow q}{\Gamma \vdash \{\nu : w \mid p\} <: \{\nu : u \mid q\}}$ (R-SUBTYPING)		

Figure 3 Syntax and subtyping.

The syntax of RFJ is depicted on the left side of Figure 3. The metavariables C , D , and E range over class names; f and g range over field names; m ranges over method names; x ranges over parameter names; ν ranges over refinement binder names. We also use n to range over integers, and b to range over booleans (i.e., *true* and *false*). In a nutshell, RFJ extends FJ by refinement types, interfaces, and the \top type, each highlighted in dark gray to distinguish the enhancements. We have discussed refinement types and interfaces extensively. For the \top type, it is introduced mainly to characterize the equality between any two values, not just values of the same type. Compared to strictly monomorphic equality which demands

type uniformity for comparands, \top -typed equality is closer to the actual Java equality [26] and the equality used in order-sorted logics [53].

Besides the extension, RFJ simplifies FJ in two aspects, widely adopted in prior studies [51, 10, 39, 25]. First, casts are not included since they complicate the calculus and are orthogonal with refinement types, the focus of this work. Second, a single parameter is used instead of an arbitrary number of parameters. However, this does not impact the expressiveness of RFJ, because empty parameters can be modeled by a single parameter that is not referred to in the method body, while multiple parameters can be modeled by declaring a class containing those parameters and using it as a single parameter.

At last, we introduce several remarkable notations. Firstly, note that we use e , p , and q to range over RFJ terms. The latter symbols (p and q) are used to range over RFJ terms that have bool type (also called *formulas*). Secondly, We use two shorthands for refinement types: we omit the binders declaration in $\{\nu : w|p\}$ when the binder is just ν (a reserved name), and we short $\{w|p\}$ as w when p is *true*.

Subtyping The right half of Figure 3 explicates RFJ's subtyping relations, featuring sub-nominal ($<:_n$), subtyping amongst base types ($<:_b$), and refinement subtyping ($<:_r$). The sub-nominal relation is a straightforward extension of FJ's subclassing to account for interface types. The base subtyping relation is also standard. Refinement subtyping combines base subtyping and logical implication (defined in Section 3.3). It is parameterized by type environments with the usual construction, which is used for logical implication. Note that when checking logical implication, we use the type of the sub-base-type (w) instead of the super-base-type (u) for ν , thus making refinement subtyping transitive. In the following of this paper, we short refinement subtyping as subtyping when there is no ambiguity.

Auxiliary Definitions The lookup functions, override relation, and *interface implemented* relation are shown in Figure 4. The lookup functions should be pretty self-explanatory, only to note that although we use \rightarrow in *mtype* and *mtypei*, it is not *arrow type* constructor, but an intuitive type signature representation, as in original FJ [11]. We explain the override and *interface implemented* relations subsequently.

In RFJ, the criterion for valid method override differs from FJ's strict type matching, utilizing co/contra-variance instead. This is encapsulated by the override relation ($override(m, C, D, p \rightarrow x : t \rightarrow r)$), which ensures the class C properly overrides the method m of the class D , with a method of signature $p \rightarrow x : t \rightarrow r$, encoding three constraints:

1. For **this**, we have co-variance in the base type (the base type is C , which is a subtype of D) and contra-variance in the refinement (as the $\emptyset \vdash \{C|q\} <: \{C|p\}$ states). Using co-variance for the base type is widely known as a seminal work [11] on method overriding has pointed out: the parameters that determine the selection must be co-variantly overridden (i.e., have a lesser type). However, since method selection relies solely on the base type (note that *mbody* considers the class but disregards refinement), we must require the refinement to be more general (contra-variant) to ensure compatibility.
2. The contra-variance on the parameter type and co-variance on the return type (ignore the subtyping context for now) follow the function subtyping principle [48].
3. Since the parameter type refinement may refer to **this**, while the return type refinement may refer to **this** and the parameter, their co/contra-variance must be assessed under a type environment with those variables, as the definition shows. Here, note that we opt for 'narrower' subtype contexts: we assess contra-variance ($t' <: t$) within the context of $\{C|q\}$ rather than $\{C|p\}$, and likewise for co-variance ($r <: r'$), within $\{C|q\}$ and t' . This decision renders the overriding rule more permissive: subtyping in a narrower context is easier to satisfy, as the narrowing property of subtyping shows (c.f., Section 4.3.1).

Field lookup	$fields(C) = \bar{t} \bar{f}$	Override	$override(m, C, D, p \rightarrow x : t \rightarrow r)$
	$fields(Object) = \bullet$		$mtype(m, D) = q \rightarrow x : t' \rightarrow r'$
	$CT(C) = class\ C\ exds\ D\ imp\ \bar{I}\{\bar{t}\ \bar{f};\ K\ \bar{M}\}$		$\emptyset \vdash \{C q\} <: \{C p\}$
	$fields(D) = \bar{s}\ \bar{g}$		$\emptyset, this : \{C q\} \vdash t' <: t$
	$fields(C) = \bar{s}\ \bar{g}, \bar{t}\ \bar{f}$		$\emptyset, this : \{C q\}, x : t' \vdash r <: r'$
C-method-type	$mtype(m, C) = p \rightarrow x : t \rightarrow r$	I-method-type	$mtypei(m, I) = p \rightarrow x : t \rightarrow r$
	$CT(C) = class\ C\ exds\ D\ imp\ \bar{I}\{\bar{t}\ \bar{f};\ K\ \bar{M}\}$		$IT(I) = interface\ I\{\bar{Q}\}$
	$r\ m\ (p, t\ x)\ \{return\ e;\} \in \bar{M}$		$r\ m\ (p, t\ x) \in \bar{Q}$
	$mtype(m, C) = p \rightarrow x : t \rightarrow r$		$mtypei(m, I) = p \rightarrow x : t \rightarrow r$
	$CT(C) = class\ C\ exds\ D\ imp\ \bar{I}\{\bar{t}\ \bar{f};\ K\ \bar{M}\}$	Implement	$implement(m, C, I, p \rightarrow x : t \rightarrow r)$
	$m\ is\ not\ defined\ in\ \bar{M}$		$mtype(m, C) = q \rightarrow x : t' \rightarrow r'$
	$mtype(m, C) = mtype(m, D)$		$\emptyset \vdash \{C p\} <: \{C q\}$
C-method-body	$mbody(m, C) = (x, e)$		$\emptyset, this : \{C p\} \vdash t <: t'$
	$CT(C) = class\ C\ exds\ D\ imp\ \bar{I}\{\bar{t}\ \bar{f};\ K\ \bar{M}\}$		$\emptyset, this : \{C p\}, x : t \vdash r' <: r$
	$r\ m\ (p, t\ x)\ \{return\ e;\} \in \bar{M}$		$implement(m, C, I, p \rightarrow x : t \rightarrow r)$
	$mbody(m, C) = (x, e)$	Interface implemented	$C \triangleright I$
	$CT(C) = class\ C\ exds\ D\ imp\ \bar{I}\{\bar{t}\ \bar{f};\ K\ \bar{M}\}$		$IT(I) = interface\ I\{r\ m(p, t\ x)\}$
	$m\ is\ not\ defined\ in\ \bar{M}$		$implement(m, C, I, p \rightarrow x : t \rightarrow r)$
	$mbody(m, C) = mbody(m, D)$		$C \triangleright I$

■ **Figure 4** Auxiliary definitions.

At last, note that we simplify the presentation by assuming identical parameter names (x); otherwise, they should be renamed to a fresh variable for checking return type co-variance.

For valid interface implementations, $C \triangleright I$ confirms that a class properly implements all methods declared in the interface. The implementation relation ($implement(m, C, I, p \rightarrow x : t \rightarrow r)$) is a dual of the override relation, ensuring that the method m of interface I with type signature $p \rightarrow x : t \rightarrow r$, is overridden in the class C .

3.2 Operational Semantics

Now, we present the operational semantics of RFJ. We first present the small-step semantics, defined in Figure 5. The semantics aligns with that of FJ³, diverging only to accommodate the integration of new constructs—specifically, primitive operations and let bindings. The standard semantics of the boolean operations—including negation, conjunction, and disjunc-

³ To be specific, we align with the semantics in the textbook presentation [48], which, diverges from the nondeterministic beta-reduction semantics in the original paper [28].

<div style="display: flex; justify-content: space-between; align-items: center;"> <div>Evaluation</div> <div style="border: 1px solid black; padding: 2px;">$e \rightsquigarrow e'$</div> </div> $\frac{fields(C) = \bar{t} \ \bar{f}}{(new \ C(\bar{v})).f_i \rightsquigarrow v_i}$ $\frac{mbody(m, C) = (x, e_0)}{(new \ C(\bar{v})).m(v) \rightsquigarrow [this \mapsto (new \ C(\bar{v})); x \mapsto v]e_0}$ $\frac{e_0 \rightsquigarrow e'_0}{e_0.f \rightsquigarrow e'_0.f}$ $\frac{e_0 \rightsquigarrow e'_0}{e_0.m(e) \rightsquigarrow e'_0.m(e)}$ $\frac{e \rightsquigarrow e'}{v_0.m(e) \rightsquigarrow v_0.m(e')}$ $\frac{e_i \rightsquigarrow e'_i}{new \ C(\bar{v}, e_i, \bar{e}) \rightsquigarrow new \ C(\bar{v}, e'_i, \bar{e})}$	<div style="display: flex; justify-content: space-between; align-items: center;"> <div></div> <div style="border: 1px solid black; padding: 2px;">$\oplus \ ok \ v_0 \ v_1$</div> </div> $\frac{e \rightsquigarrow e'}{\neg e \rightsquigarrow \neg e'}$ $\frac{}{\neg b \rightsquigarrow \neg_p(b)}$ $\frac{e_0 \rightsquigarrow e'_0}{e_0 \oplus e \rightsquigarrow e'_0 \oplus e}$ $\frac{e \rightsquigarrow e'}{v_0 \oplus e \rightsquigarrow v_0 \oplus e'}$ $\frac{}{\oplus \ ok \ v_0 \ v_1}$ $\frac{v_0 \oplus v_1 \rightsquigarrow \oplus_p(v_0, v_1)}{e_0 \rightsquigarrow e'_0}$ $\frac{}{let \ x = e_0 \ in \ e \rightsquigarrow let \ x = e'_0 \ in \ e}$ $\frac{}{let \ x = v_0 \ in \ e \rightsquigarrow [x \mapsto v_0]e}$ <div style="display: flex; justify-content: space-between; align-items: center;"> <div>Valid binary operation</div> <div style="border: 1px solid black; padding: 2px;">$\oplus \ ok \ v_0 \ v_1$</div> </div> $\frac{}{\wedge \ ok \ b_0 \ b_1}$ $\frac{}{\vee \ ok \ b_0 \ b_1}$ $\frac{}{= \ ok \ v_0 \ v_1}$
---	---

■ **Figure 5** Small-step semantics of RFJ.

tion, are preserved. The only thing worth noting is the equality operation, which is defined for every pair of values. RFJ equality is defined as the syntactic equality (i.e., we view values as finite term trees [21]: two values are equal *iff* their corresponding trees are identical).

Multi-step and Big-step Semantics We define the multi-step semantics ($e \rightsquigarrow^* e'$) as the transitive closure of the small-step semantics, used for type soundness and logical truth later.

Despite being directly derivable from small-step semantics, multi-step semantics do not provide a convenient induction principle, which makes the related proof intricate. To mitigate this, we introduce big-step semantics and prove its coincidence with the multi-step semantics terminating with a value (i.e., $e \Downarrow v$ *iff* $e \rightsquigarrow^* v$). The big-step semantics mirrors the small-step semantics, and we omit its formal definition from this paper. We defer its comprehensive exposition to the accompanying Coq development.

3.3 Logical Interpretation

Figure 6 defines the logical notations, which are used in the refinement subtyping relation and logical soundness theorem. The definitions make use of closing substitutions, i.e., partial mappings from variables to values. The application of a closing substitution θ to a term e is defined as the function $\theta(e)$, which simply substitutes each variable-value pair sequentially. We also lift $\theta(\cdot)$ to refinement types: $\theta(\{\nu : w | p\}) = \{\nu : w | \theta(p)\}$.

Logical Truth and Entailment The core of our logical interpretation is the logical truth relation, which means that the logical formula evaluates to *true* under the given interpretation (i.e., RFJ operational semantics). Note that this relation is defined only for closed formulas (i.e., *sentences*), and a closing substitution is applied whenever this relation is checked.

$\theta ::= \theta, x : v \mid \emptyset$ Closing Substitution $\emptyset(e) = e$ $(\theta, x : v)(e) = [x \mapsto v]\theta(e)$	Environment Denotation $\boxed{\theta \in \llbracket \Gamma \rrbracket}$ $\overline{\emptyset \in \llbracket \emptyset \rrbracket}$ $\frac{v \in \llbracket \theta(t) \rrbracket \quad \theta \in \llbracket \Gamma \rrbracket}{\theta, x : v \in \llbracket \Gamma, x : t \rrbracket}$
Logical Truth $\boxed{\models p}$ $\frac{p \rightsquigarrow^* \text{true}}{\models p}$	Type Denotation $\boxed{v \in \llbracket t \rrbracket}$ $\frac{\models [\nu \mapsto n]p}{n \in \llbracket \{\nu : \text{int} \mid p\} \rrbracket}$ (DENINT)
Logical Entailment $\boxed{\Gamma \models p}$ $\frac{\forall \theta \in \llbracket \Gamma \rrbracket. \models \theta(p)}{\Gamma \models p}$	$\frac{\models [\nu \mapsto b]p}{b \in \llbracket \{\nu : \text{bool} \mid p\} \rrbracket}$ (DENBOOL)
Logical Implication $\boxed{\Gamma \models p \Rightarrow q}$ $\frac{\forall \theta \in \llbracket \Gamma \rrbracket. \text{if } \models \theta(p) \text{ then } \models \theta(q)}{\Gamma \models p \Rightarrow q}$	$\frac{\models [\nu \mapsto \text{new } C(\bar{v})]p \quad \text{fields}(C) = \bar{t} \ \bar{f} \quad \bar{v} \in \llbracket [\text{this} \mapsto \text{new } C(\bar{v})]t \rrbracket}{\text{new } C(\bar{v}) \in \llbracket \{\nu : C \mid p\} \rrbracket}$ (DENCCLASS) $\frac{w <:_b u \quad v \in \llbracket \{\nu : w \mid p\} \rrbracket}{v \in \llbracket \{\nu : u \mid p\} \rrbracket}$ (UPCAST)

■ **Figure 6** Logical interpretation of RFJ.

349 With the logical truth relation in hand, we can define the logical entailment relation
 350 $(\Gamma \models p)$, which signifies the truth of a formula p under the type and logical constraints
 351 encoded within Γ . It requires that for every closing substitution that satisfies Γ (formally
 352 defined later), the closing substitution must also satisfy the formula p (i.e., make it a truth).
 353 Similarly, we define the logical implication relation $(\Gamma \models p \Rightarrow q)$, by requiring all closing
 354 substitution that satisfies Γ and p also satisfies q .

The logical implication relation is used for defining the subtyping relation (c.f., Section 3.1).
 To illustrate, we revisit the subtype constraint (4) presented in Section 2.1, which imposes
 the following constraint by the definition of subtyping and logical implication:

$$\forall \theta \in \llbracket \Gamma \rrbracket. \text{if } \models \theta(\nu = p1.\text{remA}()) \text{ then } \models \theta(\nu = \text{new } Ch(\text{new } Cr()))$$

355 where $\Gamma = p1 : \{An \mid \nu = \text{new } An(\text{new } Ch(\text{new } Cr()))\}, \nu : Pi$. There are infinite closing
 356 substitutions satisfying Γ . In particular, $p1$ can only be $\text{new } An(\text{new } Ch(\text{new } Cr()))$, but ν
 357 can be any $Pizza$, since any $Pizza$ v satisfies $v \in \llbracket Pi \rrbracket$. However, there is only one closing
 358 substitution that also satisfies the *if* condition $(\models \theta(\nu = p1.\text{remA}()))$, i.e., the one whose
 359 ν is $\text{new } Ch(\text{new } Cr())$. This closing substitution also satisfies the *then* condition. Thus,
 360 Constraint (4) holds under the logical interpretation.

361 **Type and Environment Denotation** Now, we formally define what is meant by “a
 362 substitution satisfies a type environment.” This relation is defined by the environment

denotation relation $\theta \in \llbracket \Gamma \rrbracket$, which is a natural lift of the type denotation relation ($v \in \llbracket t \rrbracket$), determining if a value is denoted by a type. Type denotation is defined by casing on the structure of the value, with an additional *upcast* rule for upcasting the base type. Basically, type denotation relation ($v \in \llbracket \{u|p\} \rrbracket$) encapsulates two facets: the value v belongs to the base type u , and it satisfies the refinement p . DENCCLASS additionally requires the denotation for the fields of the class, to justify the nominal nature of class types.

3.4 Typing

In this section, we define the typing relations in RFJ, as shown in Figure 7. We first define the term typing, depending on the type well-formedness relation, which in turn depends on the FJ term typing. After the term typing is defined, we define the method typing ($M \text{ ok in } C$), class typing ($C \text{ ok}$) and interface typing ($\mathcal{I} \text{ ok}$).

Well-formedness For a refinement type $\{\nu : w|p\}$ to be deemed well-formed under environment Γ , denoted as $\Gamma \vdash_w \{\nu : w|p\}$, the refinement p must have *bool* type under the type environment. In the definition, \vdash_F is the FJ term typing relation, which is used to check if the refinement does have *bool* type. Note that we can not use the RFJ term typing here, since it depends on the type well-formedness relation. We do not define the FJ term typing separately. It is a standard textbook relation [48] and can be obtained by removing the *gray* parts of RFJ typing. Since the FJ term typing is only defined for base types and base type environments, we must use an erase function ($[\cdot]$) to convert refinement type environments to base type environments. The erase function is naturally lifted from the erase function of refinement types (i.e., $[\{\nu : w|p\}] = w$).

Based on the type well-formedness, we define the well-formedness of type environment:

$$(1) \vdash_w \emptyset \quad (2) \vdash_w \Gamma, \Gamma \vdash_w t, x \notin \Gamma \implies \vdash_w \Gamma, x : t$$

which simply asserts that all types are well-formed and all variables are unique.

Term Typing RFJ term typing is an extension of FJ term typing, replacing base types with refinement types and using refinement subtyping for subtyping. Notably, RFJ term typing utilizes an explicit subsumption rule (T-SUB), which deviates from the implicit algorithmic subtyping commonly attributed to FJ. This deviation is not borne from necessity but is rather a methodological choice, aimed at simplifying the meta-theoretical development.

The types of primitive operations (used in T-UNOP and T-BINOP) follow their semantics:

$$\begin{aligned} \neg_t &\doteq x : \text{bool} \rightarrow \{\text{bool} | v = \neg x\} \\ \wedge_t &\doteq x : \text{bool} \rightarrow y : \text{bool} \rightarrow \{\text{bool} | v = x \wedge y\} \\ \vee_t &\doteq x : \text{bool} \rightarrow y : \text{bool} \rightarrow \{\text{bool} | v = x \vee y\} \\ =_t &\doteq x : \top \rightarrow y : \top \rightarrow \{\text{bool} | v = x = y\} \end{aligned}$$

RFJ typing utilizes several mechanisms absent in FJ typing, i.e., well-formedness checking, type substitution, and general selfication. We briefly discuss those non-standard mechanisms.

1. Well-formedness checking. Three rules (T-VAR, T-LET, and T-SUB) include type well-formedness checking in their premises, guaranteeing the inference of only well-formed types, which is required to establish various lemmas (e.g., the structural properties).
2. Type substitution. Refinement types can refer to visible variables. For example, the type of a field \mathbf{f} can be $\{\nu : \text{int} | \nu = \text{this.h}\}$, specifying it equal to the h field of the object. For those refinements to refer to proper variables, we must substitute these references with actual terms during typing. Continuing the example, suppose we are typing $a.f$, the type should be updated to $\{\nu : \text{int} | \nu = a.h\}$, by substituting this to a , as T-FIELD rule shows.

Type well-formedness	$\boxed{\Gamma \vdash_w t}$	$\frac{\Gamma \vdash e_0 : s_0 \quad \Gamma, x : s_0 \vdash e : t \quad \Gamma \vdash_w t}{\Gamma \vdash \text{let } x = e_0 \text{ in } e : \text{self}(t, \text{let } x = e_0 \text{ in } e)}$ (T-LET)
RFJ Typing	$\boxed{\Gamma \vdash e : t}$	$\frac{\neg_t \doteq x : t_0 \rightarrow r \quad \Gamma \vdash s_0 <: t_0}{\Gamma \vdash \neg e_0 : [x \mapsto e_0] r}$ (T-UNOP)
	$\frac{x : t \in \Gamma \quad \boxed{\Gamma \vdash_w t}}{\Gamma \vdash x : \text{self}(t, x)}$ (T-VAR)	$\frac{\oplus_t \doteq x : t_0 \rightarrow y : t \rightarrow r \quad \Gamma \vdash e_0 : s_0 \quad \Gamma \vdash s_0 <: t_0}{\Gamma \vdash e : s \quad \Gamma \vdash s <: [x \mapsto e_0] t}$ (T-BINOP)
	$\frac{}{\Gamma \vdash n : \{ \text{int} \mid \nu = n \}}$ (T-INT)	$\frac{\Gamma \vdash e_0 \oplus e : [x \mapsto e_0; y \mapsto e] r}{\Gamma \vdash e : s \quad \Gamma \vdash s <: t \quad \boxed{\Gamma \vdash_w t}}$ (T-SUB)
	$\frac{}{\Gamma \vdash b : \{ \text{bool} \mid \nu = b \}}$ (T-BOOL)	method typing $\boxed{M \text{ ok in } C}$ $CT(C) = \text{class } C \text{ exds } D \text{ impls } \bar{I}\{\dots\}$ $\text{this} : \{C p\}, x : t \vdash e_0 : r$ $\text{override}(m, C, D, p \rightarrow x : t \rightarrow r)$ $\frac{\emptyset \vdash_w \{C p\} \quad \text{this} : \{C p\} \vdash_w t}{\text{this} : \{C p\}, x : t \vdash_w r}$ $r \text{ m}(p, t \ x) \{ \text{return } e_0; \} \text{ ok in } C$
	$\frac{\Gamma \vdash e_0 : \{ C_0 p \} \quad \text{fields}(C_0) = \bar{t} \bar{f}}{\Gamma \vdash e_0.f_i : \text{self}([\text{this} \mapsto e_0] t_i, e_0.f_i)}$ (T-FIELD)	class typing $\boxed{C \text{ ok}}$ $K = C(\bar{s} \bar{g}, \bar{t} \bar{f}) \{ \text{super}(\bar{g}); \text{this}.f = \bar{f}; \}$ $\text{fields}(D) = \bar{s} \bar{g} \quad \bar{M} \text{ ok in } C$ $\frac{\emptyset, \text{this} : C \vdash_w \bar{t} \quad C \triangleright \bar{I}}{\text{class } C \text{ exds } D \text{ impls } \bar{I} \{ \bar{t} \bar{f}; K \bar{M} \} \text{ ok}}$
	$\frac{\text{mtype}(m, C_0) = q \rightarrow x : t \rightarrow r \quad \Gamma \vdash \{ C_0 p \} \quad \Gamma \vdash \{ C_0 p \} <: \{ C_0 q \}}{\Gamma \vdash e : s \quad \Gamma \vdash s <: [\text{this} \mapsto e_0] t}$ (T-INVOK)	interface method typing $\boxed{Q \text{ ok in } I}$ $\frac{\emptyset \vdash_w \{I p\} \quad \text{this} : \{I p\} \vdash_w t}{\text{this} : \{I p\}, x : t \vdash_w r}$ $r \text{ m}(p, t \ x) \text{ ok in } I$
	$\frac{\text{mtype}_i(m, I_0) = q \rightarrow x : t \rightarrow r \quad \Gamma \vdash e_0 : \{ I_0 p \} \quad \Gamma \vdash \{ I_0 p \} <: \{ I_0 q \}}{\Gamma \vdash e : s \quad \Gamma \vdash s <: [\text{this} \mapsto e_0] t}$ (T-INVOKI)	interface ok $\boxed{I \text{ ok}}$ $\frac{\bar{Q} \text{ ok in } I}{\text{interface } I \{ \bar{Q} \} \text{ ok}}$
	$\frac{\text{fields}(C) = \bar{t} \bar{f}}{\Gamma \vdash \bar{e} : \bar{s} \quad \Gamma \vdash \bar{s} <: [\text{this} \mapsto \text{new } C(\bar{e})] t}$ (T-NEW)	
	$\frac{}{\Gamma \vdash \text{new } C(\bar{e}) : \text{self}(C, \text{new } C(\bar{e}))}$	

■ **Figure 7** Typing relations of RFJ.

3. General selfication. Each rule except the subsumption rule and the rules for primitives (T-INT, T-BOOL, T-UNOP and T-BINOP) is companioned with a selfication operation (*self*), ensuring the terms are always recorded in their types. Selfication is not required for subsumption, as it is performed in prior derivations, and primitive rules inherently equate terms in their types (e.g., T-INT assigns $\{\text{int} \mid \nu = 2\}$ to 2).

Method, Class Typing and Interface Typing The method, class, and interface typings

are relations to identifying valid methods, classes, and interfaces. RFJ's approach to these typings closely mirrors that of FJ, with the addition of well-formedness checks for method and field types. Additionally, the class typing judgment is extended with a checking $C \triangleright \bar{I}$ that ensures the interfaces are properly implemented.

Termination At last, we address one tricky issue in typing: termination. As a Turing-complete language, the well-typedness of RFJ terms does not ensure the termination of its evaluation. However, non-terminating evaluations can lead to unsound refinements. For instance, $\emptyset \vdash \text{new } C().m() : \{\text{bool} \mid 0 = 1\}$ is derivable, where $C.m$ is defined as `bool m(){return this.m();}`. Consequently, our logical soundness theorem is strictly applicable to terms that are both well-typed and **terminating** (defined below). In practice, a termination checker should be equipped to ensure the termination where logical soundness is concerned.

$$\frac{\forall \theta \in \llbracket \Gamma \rrbracket. \theta(e) \rightsquigarrow^* v}{\Gamma \downarrow e} \text{ terminating}$$

Main Theorems The following theorems link typing to semantics and logical entailment.

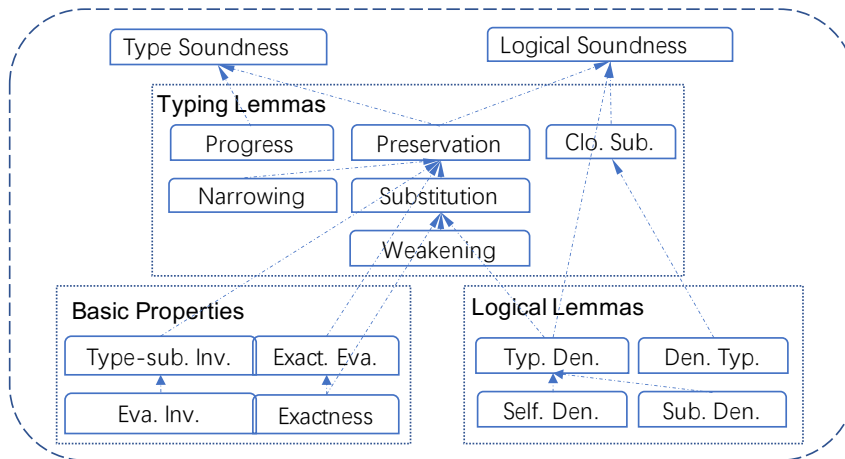
► **Theorem 1** (Type Soundness). *if $\emptyset \vdash e : t$ and $e \rightsquigarrow^* e'$, then e' is a value or $\exists e''. e' \rightsquigarrow e''$.*

► **Theorem 2** (Logical Soundness). *if $\Gamma \vdash e : \{\nu : w \mid p\}$, $\vdash_w \Gamma$, and $\Gamma \downarrow e$, then $\Gamma \models [\nu \mapsto e]p$.*

The major steps to establish those theorems are given in the next section.

4 Meta-theoretical Results

We argue the proposed system possesses type soundness and logical soundness. The proof of type soundness follows the “Type Soundness = Preservation + Progress” approach [63]. The approach to logical soundness is different from that of previous refinement type systems, as their approach does not apply to RFJ (c.f., Section 2.3). Our proof approach can be summarized as “Logical Soundness = Preservation + Typing Denotation + Closing Substitution.” We give an overview of the critical lemmas and theorems used in the proof and the dependency relation in Figure 8. In the remainder of this section, we provide a brief overview of the proof. For a detailed exposition, please refer to the Coq development.



■ **Figure 8** Proof Overview. Arrows signify the dependencies among lemmas and theorems.

4.1 Basic Properties

► **Lemma 3** (Type-substitution Invariant). *if $e \rightsquigarrow e'$, then $\Gamma \vdash [x \mapsto e]t <: \Gamma \vdash [x \mapsto e']t$ and $\Gamma \vdash [x \mapsto e']t <: \Gamma \vdash [x \mapsto e]t$.*

This lemma states the coherence of types under substitution with pre-or-post-evaluation terms. This lemma is important to prove the preservation lemma. Since subtyping relies eventually on evaluation, the primary challenge of proving this lemma hinges on demonstrating that evaluation remains unaffected by the substitution with pre-or-post-evaluation terms, as the next lemma shows.

► **Lemma 4** (Evaluation Invariant). *if $e \rightsquigarrow e'$, then $[x \mapsto e]p \rightsquigarrow^* v \Leftrightarrow [x \mapsto e']p \rightsquigarrow^* v$.*

Since the multi-step evaluation (\rightsquigarrow^*) does not give a very useful induction principle, we first prove this lemma using the big-step semantics, then link the lemma back to multi-step semantics via the correspondence between big-step and multi-step semantics (i.e., $e \Downarrow v \Leftrightarrow e \rightsquigarrow^* v$).

► **Lemma 5** (Exactness Evaluation). *if $e \rightsquigarrow e'$ and $\Gamma \vdash e' : t$ then $\Gamma \vdash e' : \text{self}(t, e)$.*

This lemma ensures the term after evaluation (e') can have the type selfified with the term before evaluation (e), which is often needed to prove the preservation of typing throughout evaluation steps. This lemma requires the exactness lemma, as shown below.

► **Lemma 6** (Exactness). *if $\Gamma \vdash e : t$ then $\Gamma \vdash e : \text{self}(t, e)$.*

This lemma states what we mean by “term information is always recorded”: for any well-typed term e , we can always construct a typing where the term is selfified (recorded in the type). Apart from being used for Lemma 5, this lemma is important for the substitution lemma (Lemma 12).

4.2 Logical Lemmas

► **Lemma 7** (Typing Denotation). *if $\Gamma \vdash v : t$, then $\forall \theta \in \llbracket \Gamma \rrbracket. v \in \llbracket \theta(t) \rrbracket$.*

This lemma states that typing implies denotation. It can be proved by induction on typing and using Lemma 8 and 9 for selfication and subtyping. This lemma is important for the substitution lemma (Lemma 12). It is also a milestone for logical soundness, as we discuss in Section 4.5.

► **Lemma 8** (Selfication Denotation). *if $v \in \llbracket t \rrbracket$ then $v \in \llbracket \text{self}(t, v) \rrbracket$.*

This lemma states that selfication preserves denotation, which is used to prove Lemma 7.

► **Lemma 9** (Subtyping Denotation). *if $\Gamma \vdash s <: t$, then $\forall \theta \in \llbracket \Gamma \rrbracket. v \in \llbracket \theta(s) \rrbracket \Rightarrow v \in \llbracket \theta(t) \rrbracket$.*

This lemma argues that every value denoted by a type is included in the denotation of its supertype. This lemma is used in proving the subtyping cases of Lemma 7.

► **Lemma 10** (Denotation Typing). *if $v \in \llbracket t \rrbracket$ and $\emptyset \vdash_w t$, then $\emptyset \vdash v : t$.*

This lemma states that denotation implies typing, which is crucial for Lemma 16. The basic proof idea is to first construct a “ground type” for v : $\emptyset \vdash v : \{\nu : w \mid \nu = v\}$, where w is the *inherent* base type of the value (*int* for n , *bool* for b , and C for *new* $C(\dots)$), and then link the “ground type” to t by $\emptyset \vdash \{\nu : w \mid \nu = v\} <: t$, which holds due to $v \in \llbracket t \rrbracket$.

4.3 Typing Lemmas

4.3.1 Structural Lemmas for Typing

As usual, we establish structural properties (weakening, narrowing and substitution) for RFJ typing. Since typing relies on subtyping which in turn, relies on logical implication, we need those structural properties for subtyping and logical implication, too.

► **Lemma 11** (Narrowing). *for any variable x not in Γ and Γ' :*

1. *if $\Gamma, x : r, \Gamma' \vdash p \Rightarrow q$ and $\Gamma \vdash r' <: r$, then $\Gamma, x : r', \Gamma' \vdash p \Rightarrow q$.*
2. *if $\Gamma, x : r, \Gamma' \vdash s <: t$ and $\Gamma \vdash r' <: r$, then $\Gamma, x : r', \Gamma' \vdash s <: t$.*
3. *if $\Gamma, x : r, \Gamma' \vdash e : t$ and $\Gamma \vdash r' <: r$, then $\Gamma, x : r', \Gamma' \vdash e : t$.*

The first narrowing lemma can be proved by observing that a denotation θ' of $\Gamma, x : r', \Gamma'$ is always a denotation of $\Gamma, x : r, \Gamma'$. Using the first lemma, the remaining two are easy.

► **Lemma 12** (Substitution). *for any distinct variables x and y not in Γ and Γ' :*

1. *if $\Gamma, x : r_x, y : r_y, \Gamma' \models p \Rightarrow q$ and $\Gamma \vdash v_x : r_x, \Gamma \vdash v_y : [x \mapsto v_x]r_y$, then $\Gamma, [x \mapsto v_x; y \mapsto v_y]\Gamma' \models [x \mapsto v_x; y \mapsto v_y]p \Rightarrow [x \mapsto v_x; y \mapsto v_y]q$.*
2. *if $\Gamma, x : r, y : r_y, \Gamma' \vdash s <: t$ and $\Gamma \vdash v_x : r_x, \Gamma \vdash v_y : [x \mapsto v_x]r_y$, then $\Gamma, [x \mapsto v_x; y \mapsto v_y]\Gamma' \vdash [x \mapsto v_x; y \mapsto v_y]s <: [x \mapsto v_x; y \mapsto v_y]t$.*
3. *if $\Gamma, x : r, y : r_y, \Gamma' \vdash e : t$ and $\Gamma \vdash v_x : r_x, \Gamma \vdash v_y : [x \mapsto v_x]r_y$, then $\Gamma, [x \mapsto v_x; y \mapsto v_y]\Gamma' \vdash [x \mapsto v_x; y \mapsto v_y]e : [x \mapsto v_x; y \mapsto v_y]t$.*

Since RFJ has double substitution operations in method invocation (we must substitute for **this** and the parameter), we need double substitution lemmas. The first substitution lemma follows from the observation that a denotation of $\Gamma, x : r_x, y : r_y, \Gamma'$ can be constructed from a denotation of $\Gamma, [x \mapsto v_x; y \mapsto v_y]\Gamma'$ by adding $x : v_x$ and $y : v_y$. The core step of this construction is to prove that v_x is indeed a denotation of r_x and v_y is indeed a denotation of $[x \mapsto v_x]r_y$, utilizing Lemma 7. Using the first lemma, the second lemma is easy. The third lemma can be proved by induction on typing. The T-VAR case requires the exactness lemma (Lemma 6) and weakening lemma (shown below). The other cases are easy.

► **Lemma 13** (Weakening). *for any variable x not in Γ and Γ' :*

1. *if $\Gamma, \Gamma' \models p \Rightarrow q$, then $\Gamma, x : r, \Gamma' \models p \Rightarrow q$.*
2. *if $\Gamma, \Gamma' \vdash s <: t$, then $\Gamma, x : r, \Gamma' \vdash s <: t$.*
3. *if $\Gamma, \Gamma' \vdash e : t$, then $\Gamma, x : r, \Gamma' \vdash e : t$.*

The first weakening lemma can be proved by observing that we can always construct a denotation θ' of Γ, Γ' from a denotation θ of $\Gamma, x : t, \Gamma'$, by removing the x entry from θ . Since x is fresh, removing it from θ does not impact the validity of this implication. With the first weakening lemma in hand, the remaining two are straightforward.

4.3.2 Progress & Preservation

► **Lemma 14** (Progress). *if $\emptyset \vdash e : t$ then e is a value or $\exists e'. e \rightsquigarrow e'$.*

The proof is done by induction on typing, following the standard approach of FJ.

► **Lemma 15** (Preservation). *if $\emptyset \vdash e : t$ and $e \rightsquigarrow e'$, then $\emptyset \vdash e' : t$.*

The proof is done by induction on the typing judgment and using the structural lemmas for substitutions and environment narrowings. To argue the preservation in the presence of general selfication and type substitution, Lemma 5 and Lemma 3 must also be utilized.

507 4.3.3 Closing Substitution

508 ► **Lemma 16** (Closing Substitution). *if $\Gamma \vdash e : t$, then $\forall \theta \in \llbracket \Gamma \rrbracket. \emptyset \vdash \theta(e) : \theta(t)$.*

509 The closing substitution lemma bears a similarity with the substitution lemma (Lemma 12).
 510 They both concern the invariance of typing under substitution. The closing substitution
 511 lemma can be proved by induction on typing. Most of the cases are standard, except for the
 512 variable case, which requires proving $\emptyset \vdash \theta(x) : \theta(t)$ under $\Gamma \vdash x : t$. Since θ is a denotation
 513 of Γ , we know that x must be in θ and $\theta(x) \in \llbracket \theta(t) \rrbracket$. Thus, Lemma 10 can be applied to
 514 construct the expected typing judgment.

515 4.4 Type Soundness

516 To improve the readability, we reproduce Type Soundness (Theorem 1) below:

517 ► **Corollary 17** (Type Soundness). *if $\emptyset \vdash e : t$ and $e \rightsquigarrow^* e'$, then e' is a value or $\exists e''. e' \rightsquigarrow e''$.*

518 Type soundness is an easy corollary of progress and preservation, following the standard
 519 argument [63].

520 4.5 Logical Soundness

521 To improve readability, we reproduce the Logical Soundness (Theorem 2) below:

522 ► **Corollary 18** (Logical Soundness). *if $\Gamma \vdash e : \{\nu : w|p\}$, $\vdash_w \Gamma$, and $\Gamma \downarrow e$, then $\Gamma \models [\nu \mapsto e]p$.*

523 The key to proving logical soundness is to observe that it can be reduced to closed logical
 524 soundness (shown below) if we can derive a corresponding closed typing judgment given any
 525 typing judgment. This is facilitated by the closing substitution lemma (Lemma 16).

526 ► **Theorem 19** (Closed Logical Soundness). *if $\emptyset \vdash e : \{\nu : w|p\}$ and $\downarrow e$, then $\models [\nu \mapsto e]p$.*

527 Closed logical soundness is a natural consequence of preservation and typing denotation.
 528 Supposing e evaluates to v , the proof skeleton is that:

- 529 1. due to the preservation lemma, $\emptyset \vdash v : \{\nu : w|p\}$.
- 530 2. due to the typing denotation lemma, $v \in \llbracket \{\nu : w|p\} \rrbracket$, thus $\models [\nu \mapsto v]p$ by the definition of
 531 denotation.
- 532 3. at last, we can apply the evaluation invariant lemma to get $\models [\nu \mapsto e]p$.

533 5 Logical Encoding: LFJ

534 Following the standard procedure as outlined in, e.g., [6], we convert RFJ to an algorithmic
 535 bidirectional type system. The only judgment whose algorithmic property was unexplored
 536 was the class-based refinement subtyping. In this section, we present an encoding of RFJ
 537 to an order-sorted first-order logic [53], named LFJ, which gives a convenient axiomatic
 538 approach to determine RFJ refinement subtyping by invoking logical decision procedures.

539 5.1 Language

540 Figure 9 presents the syntax of LFJ. The constant symbols (c) are for RFJ values. We
 541 assume each RFJ value has a corresponding LFJ constant symbol. The function symbols (g)
 542 are for methods (N_m), field selectors (C_f), class constructors (C_{cr}), and primitive operations
 543 in RFJ. We associate methods with nominal names and field selectors with class names, for

544 attributing more precise semantics (detailed later). Note that interfaces have no fields. The
 545 terms in LFJ do not contain quantification: they are viewed as implicitly quantified and a
 546 universal quantification would be added to the outermost to close them. Sorts in LFJ consist
 547 of \top , Int , $Bool$, and N . The sorts have an apparent correspondence with RFJ base types.
 548 We denote $|w|$ as the translation of a base type w to its sort, and $|t|$ as the translation from
 549 a refinement type t to its sort. The subsort relation \sqsubseteq is straightforwardly translated from
 550 the base-subtyping relation. The signatures of functions are also translated from their RFJ
 551 type definitions, e.g., the signature of C_m is $C \rightarrow |t| \rightarrow |r|$ if $mtype(m, C) = p \rightarrow x : t \rightarrow r$.

Syntax		Term Translation
$e, p ::=$	<i>terms:</i>	$ x = x$
x	<i>variable</i>	$ v = c_v$
c	<i>constant</i>	$ \neg e_0 = \neg(e_0)$
$g(\bar{e})$	<i>apply</i>	$ e_0 \oplus e_1 = \oplus(e_0 , e_1)$
$let\ x = e\ in\ e$	<i>let binding</i>	$ new\ C(\bar{e}) = C_{cr}(\bar{e})$
$g ::= N_m \mid C_f \mid C_{cr} \mid \neg \mid \oplus$	<i>functions</i>	$ e_0.m(e_1) = \delta(e_0)_m(e_0 , e_1)$
		$ e_0.f = \delta(e_0)_f(e_0)$
		$ let\ x = e_0\ in\ e = let\ x = e_0 \ in\ e $
Sorts		
$s ::= \top \mid Int \mid Bool \mid N$	<i>sorts</i>	
$ w = match\ w\ with$	<i>base translation</i>	
$ \top \Rightarrow \top \mid int \Rightarrow Int \mid bool \Rightarrow Bool \mid N \Rightarrow N$		
$ t = t $	<i>type translation</i>	
$\sqsubseteq \doteq \mid <:_b \mid$	<i>subsort</i>	
		Environment Translation
		$ \emptyset = true$
		$ \Gamma, x : \{\nu : u \mid p\} = \Gamma \wedge \nu \mapsto x p $

■ **Figure 9** LFJ syntax and translation.

552 **Translation** The translation from RFJ terms and type environments to LFJ terms is mostly
 553 straightforward. The only thing to note is the association of type information during the
 554 translation of method invocations and field accesses, marked **brown** in Figure 9. This is
 555 facilitated by the *typeof* function: $\delta(e)$ is the static type of expression e . δ can be constructed
 556 during type checking. The association of type information is important for two purposes (we
 557 take method invocations as an example, but the argument also applies to field accesses):

- 558 ■ *Disambiguation.* Suppose the method m is defined by two classes C and D , which share no
 559 common superclass except **Object**. If methods are not associated with nominal types, the
 560 LFJ function representation of m would necessitate an assumed domain of *Object* for its
 561 first parameter, rendering the model for the function inherently partial, because not all
 562 *Object* has an m implementation. Incorporating type information ensures model totality
 563 for the first parameter by guaranteeing the existence of at least one implementation of
 564 m ; such existence is verified by static type checking. This totality guarantee plays an
 565 important role in the intended model (c.f., Section 5.2).
- 566 ■ *Axiomatization.* The aim of LFJ is to provide an axiomatization of its intended model
 567 (c.f., Section 5.3). By associating type information, the axiomatization can be crafted
 568 with greater specificity and accuracy.

5.2 Intended Model

Domain:	Functions:
$G_I = G_\top = \emptyset$	$\neg, \wedge, \vee = \text{normal}$
$G_C = \{C(\overline{d_s}) \mid \overline{d_s} \in \overline{D_{ t }}\}, fs(C) = \bar{t} \bar{f}$	$C_{cr}(\bar{d}) = C(\bar{d})$
$G_{Int} = \mathcal{Z}$	$C_{f_i}(C'(\bar{d})) = d_i, C' \sqsubseteq C \text{ and } fs(C) = \bar{t} \bar{f}$
$G_{Bool} = \{T, F\}$	$N_m(this, x) = \begin{cases} \llbracket mb(m, C) \rrbracket(this, x) & \text{if } this = C(\bar{d}) \\ \dots \text{ proceeds for all } C \sqsubseteq N \end{cases}$
$D_s = \{d \mid d \in G_{s'} \wedge s' \sqsubseteq s\}$	

■ **Figure 10** The intended model of LFJ. *fs* is short for *fields*.

In this section, we delineate the construction of an intended model \mathcal{A} for LFJ, given in Figure 10. This model bears similarities with several denotational semantics of class-based languages [54, 12], especially in the usage of *conditional functions* as models of method invocations, whereas we work with order-sorted logic, different from those semantics.

Domains Each sort s is associated with a dynamic domain G_s and a static domain D_s . The dynamic domain of a sort is a *set* containing all values inherently belonging to the sort. The dynamic domains of \top and I (i.e., interfaces) are both \emptyset . G_{Int} and G_{Bool} are standard. G_C is the finite term trees [21] generated in a sort-correct manner (i.e., each field is drawn from the static domain of the corresponding sort). The static domain (or simply, domain) for a sort s aggregates the dynamic domains of its subsorts, as in standard OS-FOL [53].

Functions The model adopts conventional interpretations for equality and boolean operators. The intended functions for constructors and fields are the constructing and destructing functions for term trees. The intended function of N_m is just a *conditional function* composed of the denotations of the implementation functions conditioned by the first parameter (i.e., the receiving object). We do not detail the denotations in this paper: because we require termination for well-typed RFJ programs, those denotations are total on their domains and can be constructed using standard fixed-point techniques as shown in, e.g., [43].

Algorithmic Subtyping With the intended model \mathcal{A} in hand, we now define the algorithmic subtyping relation:

$$\frac{w <_b u \quad \mathcal{A} \models_L \forall \bar{x}. |\Gamma| \wedge |p| \Rightarrow |q|}{\Gamma \vdash \{\nu : w|p\} <_L \{\nu : u|q\}} A\text{-Subtyping}$$

where \models_L is the normal semantics of OS-FOL [53]. We assume all variables in Γ are distinct and are not ν . We use a universal quantification $\forall \bar{x}$ to close the formula, where \bar{x} is the variables used in Γ , p and q .

We explain the algorithmic subtyping and the intended model by revisiting Constraint (4), which is translated to the formula below:

$$\forall p1 : An, \nu : Pi. p1 = An_{cr}(Ch_{cr}(Cr_{cr}())) \wedge \nu = An_{remA}(p1) \Rightarrow \nu = Ch_{cr}(Cr_{cr}()) \quad (\phi)$$

As defined by the algorithmic subtyping rule, the validity $\mathcal{A} \models \phi$ determines the subtyping constraint (4). Due to the interpretation of the constructor functions, we know that $p1$ must be $An(Ch(Cr))$. To proceed, we need the interpretation of An_{remA} :

$$An_{remA}(this) = \begin{cases} \llbracket An_{remA} \rrbracket(this) & \text{if } this = An(\dots) \\ \llbracket An_{remA} \rrbracket(this) & \text{if } this = Ma(\dots) \end{cases}$$

The interpretation of An_{remA} illustrates the conditional function nature of An_{remA} . This function redirects to the implementation functions depending on the *dynamic type* of *this* (although in this case, the directed functions are the same). Because we know $p1$ is $An(Ch(Cr))$, we choose the first branch, which maps $An(Ch(Cr))$ to $Ch(Cr)$, thus the validity is verified.

Soundness We establish the soundness of the algorithmic subtyping with respect to the refinement subtyping, which is a corollary of the semantic equivalence and translation-substitution distributivity. Semantic equivalence states the true sentences in RFJ logical interpretation is also true in \mathcal{A} , and vice versa. Translation-substitution distributivity states it does not matter whether we apply a closing substitution prior or post to the translation.

► **Proposition 20** (Semantic Equivalence). $\mathcal{A} \models_L |p| \Leftrightarrow \models p$

► **Proposition 21** (Translation-substitution Distributivity). $\mathcal{A} \models_L |\theta|(|p|) \Leftrightarrow \mathcal{A} \models_L |\theta(p)|$

► **Corollary 22.** *if $\Gamma \vdash s <_L t$, then $\Gamma \vdash s < t$.*

Proof. We give a brief proof sketch of Corollary 22 here. Suppose s is $\{\nu : w|p\}$ and t is $\{\nu : u|q\}$. To prove $\Gamma \vdash \{\nu : w|p\} < \{\nu : u|q\}$, we need to prove $\forall \theta \in \llbracket \Gamma, \nu : w \rrbracket. \text{if } \models \theta(p) \text{ then } \models \theta(q)$. By $\Gamma \vdash s <_L t$, we have $\mathcal{A} \models_L \forall \bar{x}. |\Gamma| \wedge |p| \Rightarrow |q|$, which gives us $\forall \sigma. \mathcal{A} \models_L \sigma(|\Gamma| \wedge |p|) \Rightarrow \mathcal{A} \models_L \sigma(|q|)$ (by the semantics of OS-FOL). Pick σ as $|\theta|$, due to Propositions 20 and 21, we have $\mathcal{A} \models_L |\theta|(|\Gamma| \wedge |p|)$, which let us deduce $\mathcal{A} \models_L |\theta|(|q|)$. Using Propositions 20 and 21 again, but in the reverse direction, we have $\models \theta(q)$. ◀

Mechanizing the Model While *semantic equivalence* and *translation-substitution distributivity* might appear self-evident, our attempt at their mechanization unveils profound complexities, due to its model-theoretic nature. To mechanize the model, one must mechanize the function interpretations, which imposes two technical challenges. (1) The class table lookup functions, which should be used to define C_f and N_m , are not actual Coq functions. They are defined by induction on the inheritance hierarchy. The well-foundedness of the induction is guarded by the assumption that the inheritance hierarchy is finite and acyclic, instead of the structural decreasing requirement of Coq [4]. Thus, those “functions” are actually “relations” in Coq and can not be referred to in a function environment (e.g., to define C_f or N_m). (2) The denotations of FJ methods (and thus, RFJ methods) have never been mechanized before. There exist significant efforts in mechanizing the method denotations in Coq, which is interesting and important but exceeds the scope of our paper.

5.3 Theory

To utilize the capability of deductive reasoning for checking subtyping algorithmically, we axiomatize the intended model \mathcal{A} by a theory $\mathcal{T}_{\mathcal{J}}$. $\mathcal{T}_{\mathcal{J}}$ consists of the usual theory of Equality, Uninterpreted Functions, and Linear Integer Arithmetic (EUFLIA) [1]. Besides, it is equipped with axioms for N_m , C_f , and C_{cr} . We specify them in this section.

- (1) $inject : fs(C) = \bar{f} \bar{t} \rightarrow \forall \bar{x} : |\bar{t}|, \bar{y} : |\bar{t}|. C_{cr}(\bar{x}) = C_{cr}(\bar{y}) \Rightarrow \bar{x} = \bar{y}$
- (2) $discriminate : fs(C) = \bar{f} \bar{t}, fs(D) = \bar{f}' \bar{t}' \rightarrow \forall \bar{x} : |\bar{t}|, \bar{y} : |\bar{t}'|. C_{cr}(\bar{x}) \neq D_{cr}(\bar{y})$
- (3) $access : fs(C) = \bar{f} \bar{t}, fs(C') = \bar{f}' \bar{t}', C' \sqsubseteq C \rightarrow \forall \bar{x} : |\bar{t}'|. C_{f_i}(C'_{cr}(\bar{x})) = x_i$
- (4) $invoke : mb(m, C) = (x, e), mt(m, C)_x = t_x, C.m \text{ final} \rightarrow \forall o : C, x : |t_x|. C_m(o, x) = |e|$
- (5) $override : N' \sqsubseteq N, mt(m, N')_x = t_x \rightarrow \forall o : N', x : |t_x|. N_m(o, x) = N'_m(o, x)$

The above listing gives five axiom schemata. \rightarrow means “instantiate”: if the condition on the left is satisfied, one can instantiate an axiom following the schema on the right. fs ,

636 mb , and mt_x are shorts for *fields*, *mbody*, and the parameter part of *mtype* (or *mtypei* for
637 interfaces), respectively.

638 We have seen all but *inject* and *discriminate* in Section 2.2. *inject* asserts the equality
639 between fields based on equality between objects, whereas *discriminate* asserts the inequality
640 between objects created with different constructors. The two axiom schemata are important
641 to reason about constructor behaviors.

642 The correctness of those axioms under \mathcal{A} should be obvious. We only give a discussion
643 on *invoke* here. *invoke* is the schema that is responsible for characterizing the method
644 invocation semantics of RFJ. Remember that we define the intended function of the N_m as
645 the conditional function composed of the implementation functions. Thus, a characterization
646 should account for all the possible implementation functions. Formally, the characterization
647 \mathcal{R} should be a logical consequence of all possible implementation functions: $\forall C_i \sqsubseteq N.v =$
648 $|C_i.m(this, x)| \Rightarrow \mathcal{R}(v)$. Because finding a precise characterization is not easy (despite that
649 imprecise ones always exist, e.g., *true*), we only give characterizations for the final methods,
650 i.e., the case where there is only one implementation method. For those methods, the method
651 body can be directly used.

652 **Encoding to Many-sorted Logic** The above axioms are defined in OS-FOL and should be
653 used in order-sorted deductive reasoning. Unfortunately, we are not aware of any SMT solver
654 that implements that. Thus, we translate the axioms into many-sorted logic following the
655 strategy suggested by Leino [35]. The translation of primitive data types is straightforward.
656 For objects, a unified sort *Object* is designated. Furthermore, we introduce a sort *Nominal*
657 to encompass all nominal entities—classes and interfaces in the targeting RFJ program. We
658 also declare the sub-nominal relation between those entities. The association of nominal
659 information with objects is facilitated through the *Tag* function, which relates objects with
660 their nominal identifiers. The sort requirements become sub-nominal checkings on tags, e.g.,
661 instead of $\forall x : C.p(x)$, we use $\forall x : \text{Object.sub-nominal}(\text{Tag}(x), C) \Rightarrow p(x)$.

662 The direct encoding of the sort \top into many-sorted logic remains outside our current scope,
663 primarily influencing the polymorphic nature of equality. Nevertheless, given the uniform
664 *Object* sort for all object values, object equality is still \top -typed essentially, circumventing
665 potential limitations posed by the absence of a direct \top sort.

666 6 Mechanization and Implementation

667 6.1 Coq Mechanization

668 We mechanize the meta-theory of RFJ in Coq. There are two major technical challenges
669 around the mechanization. (1) *Binders*. Handling binders is cumbersome and complex [3],
670 especially considering the number of binder structures present in RFJ (e.g., methods, let-
671 bindings, and refinement types). To address this issue, we adopt the locally nameless
672 representation [13]. Although the locally nameless representation has been widely used in
673 mechanizing functional languages [8, 27, 13], to the best of our knowledge, ours is the first
674 mechanization of a class-based language that utilizes this technique. (2) *Nested Inductive*
675 *Types*. The presence of nested inductive types within our definitions poses a significant
676 challenge; that is, the default induction principles generated by Coq fell short when proving
677 the most critical properties. To mitigate this issue, we specify the custom induction principles
678 for a range of inductive definitions (e.g., terms, typing judgments, and big-step semantics),
679 following the classical methodology [15].

680 **Mechanization Outline** We briefly overview the structure of the mechanization, which
681 contains about 15K lines of Coq code:

1. Definitions (3K): language definitions as presented in Section 3.
2. Lemmas (11K):
 - a. Basic Lemmas (5K): miscellaneous lemmas concerning basic operations, semantics, and class/interface definitions (some of which are listed in Section 4.1).
 - b. Logical Lemmas (2K): lemmas concerning the logical interpretation (c.f., Section 4.2).
 - c. Typing Lemmas (4K): basic, structural, and crucial lemmas of typing (c.f., Section 4.3).
3. Theorems (1K): type and logical soundness theorems (c.f., Sections 4.4 and 4.5).

6.2 Python Implementation

We implement a refinement type checker for RFJ. The implementation is written in roughly 2,000 lines of Python code, with Z3 [18] as the SMT backend. In addition to all features of RFJ, the type checker also supports a form of *if-then-else* following the standard practice [29], to increase the scope of the evaluation. The concrete syntax supported in the implementation is a subset of Python with static types. We opt for Python just to reuse its parser and editor supports. RFJ can be implemented for any other class-based language.

Test Suite To test the type checker, we handcraft a test suite, including all the major examples that do not use type-test/downcast or imperative features from a Java textbook [22], as well as some interesting examples inspired by previous work [59]. Each example is paired with some non-trivial properties. In total, there are 14 examples with about 1,500 LOC, covering all important features of RFJ. We list several representative examples here.

Name	Features	LOC	Properties
pizza	classes, overrides	135	remA_noinc_price, remA_idempotent
pizza visitor	visitors, upcasts	110	noObj_after_rem, noObj_after_effective_sub
tree	visitor interfaces	152	height_ge_root
geometry	factory methods	184	origin_in_shape
list	data structures	125	contains_weakening, inserts_preserve_sortedness
λ calculus	data structures	71	size_positive, substitution_nodc_size
stlc	meta-theories	307	map_extend_included, typing_weakening

Table 1 Several representative examples.

Type-checking each example took under 5 seconds, on an Apple M1 machine.

7 Discussion

In this section, we discuss specific designs of RFJ in greater detail.

Type Substitution vs ANF and Existential types In the realm of refinement type systems, the conventional strategy often involves leveraging ANF [29, 34] or existential types [44, 31, 8] to maintain the logic of refinements within a decidable framework, such as EUFLIA [9]. Our approach, however, consciously eschews these mechanisms and sticks to simple type substitution for three compelling reasons. (1) *From the theoretical perspective.* We want to argue the soundness of our system within a broader, more generalized framework: all RFJ programs expressed in ANF are inherently valid within our system, while the converse does not hold. Thus, our results perfectly apply to the condition where ANF is required (e.g., a particular implementation may perform ANF transformation before type checking). (2) *From the algorithmic perspective.* Recent advances [38, 41] have shown a complete algorithm for formula validity under a user-specified theory exists, which is exactly what we need to perform algorithmic subtyping checking. The fact that all our examples are checked with few time efforts also evidences that a reasonably efficient algorithm exists even if the logic falls outside the familiar decidable fragment. (3) *From the pragmatical perspective.* Eliminating ANF and existential types significantly lowers the barrier between the programmer's intent

and the underlying type system, simplifying the debugging process. To further lower the barrier, our typing rules are carefully formulated without using any implicit environment extension (e.g., the *Field* and *Invoke* rules in [44]). The only cases that would extend the typing and subtyping environment are *Let* and method typing, thereby maintaining a clear correspondence between the code and its type-level representation.

Axiomatization vs Reflection As pointed out by prior work [60], there are two kinds of methodologies to support user-defined functions in refinement type systems: axiomatization and reflection. Axiomatization articulates the semantics of user-defined functions through logical axioms, an approach we adopt and have elaborated on in Section 5.3. In contrast, reflection directly incorporates the function definition into the return type’s refinement (e.g., the return type of `Anchovy.remA` can be declared as $\{\nu : \text{Pizza} \mid \nu = \text{this.p.remA}()\}$ to reflect its definition). In our system, programmers can utilize reflection by manually specifying the method return type (reflection annotation could also be provided to automate this process). Those reflections are always valid thanks to general reflection, which ensures that terms are always recorded in refinements. Notably, reflection offers an alternative to the *final* constraint of the invoke axiom schema (c.f. Section 5.3): one can reflect the definition of an overriding method and the overridden method simultaneously, as long as the return types of those methods obey the co-variance principle.

The major difference between reflection and axiomatization resides in the instantiation strategy of method definitions. With reflection, instantiations of the reflected functions are performed within the type system, either by the programmer or an algorithm (e.g., PLE in [60]). With axiomatization, instantiation is delegated to the SMT solver, although special mechanisms such as *trigger/fuel* [2, 37] are needed to keep the process in control. Currently, no special algorithm or mechanism for reflection or axiomatization is employed in RFJ. However, we identify the comparison of these two methodologies in RFJ, especially in the context of a reflection instantiation algorithm and more advanced type system features (e.g., occurrence typing and union/intersection types) as important future work.

8 Related Work

This work intersects three research topics: class-based refinement type systems, mechanization of refinement types and class-based languages, and SMT-based reasoning in program verifiers.

Class-based Refinement Type Systems In the literature, many works have focused on class-based refinement type systems. For example, Nystrom et al. [44] formalize core X10 as a refinement type system. However, they focus only on the functional aspects. Vekris et al. [61] introduce a refinement type calculus that not only conducts immutability analysis but also integrates union and intersection types, with the caveat that only immutable fields are subject to refinement. Campos et al. [10] combine refinement types with class-based linear types, further increasing the support for imperative features. Kuncak et al. [52] present qualified type, a form of refinement type, and offer an in-depth discussion on qualifier inference. Gamboa et al. [24] address the practical challenges of incorporating refinement types into existing class-based systems by proposing a design approach to usability.

All the aforementioned work limits their refinements to well-established decidable SMT theories (e.g., EUFLIA). In contrast, the system of Kazerounian et al. [30] supports expressive refinements by translating the source program to Rosette [57], an intermediate verification language. The system of Lanzinger et al. [33] translates Java programs to a Java interactive theorem prover [1] for discharging subtyping constraints. However, the reliance on external verification tools complicates the analysis of their meta-theoretical properties.

This work addresses the expressiveness and soundness problems by providing an expressive and mechanized calculus grounded in Featherweight Java. We anticipate that extensions such as generics and imperative features could be seamlessly integrated into our framework, prospects we reserve for future exploration.

Mechanization of Refinement Types and Class-based Languages Several pieces of recent work have been dedicated to the mechanization of refinement types. Lehmann et al. [34] formalize a refinement type system in Coq. Their logical interpretation is axiomatized via a few basic requirements. This interpretation, however, leaves the semantics of logical formulas nebulous. Meanwhile, their proof focuses solely on the closed logical soundness, rather than general logical soundness. Wang et al. [62] mechanize in Coq a calculus that uses refinement types for complexity analysis, defining logical interpretations through denotational semantics that link refinements to Coq definitions. This method restricts the scope of terms that can be utilized as refinements due to the limitation of denotational semantics. Borkowski et al. [8] mechanizes a polymorphic refinement type system in Coq. They use an axiomatized logical interpretation for type soundness, and an operational-semantics-based logical interpretation for logical soundness. Hamza et al. [27] formalize a polymorphic refinement type system in Coq. They also employ an operational-semantics-based logical interpretation (named reducibility in the original paper). Our work draws inspiration from the two works on using operational-semantics-based logical interpretations, yet our proof diverges notably, especially given the inapplicability of logical relation techniques in our context. Moreover, our framework includes several special mechanisms such as general selfcication and nominal subtyping, extending beyond the capabilities of the systems devised by those authors. Chen's work [14] in Agda takes a unique route by integrating Agda to define a denotational semantics for refinements. However, the algorithmic properties are complicated, due to the reliance on Agda's logic. Ghalayini et al. [24] opt for a categorical-theoretical perspective for logical interpretation in their mechanized refinement type system in Lean [17], contrasting with the semantic logical interpretation in our work.

Apart from the abovementioned differences, our research sets itself apart by focusing on a class-based calculus. This foundation renders our model particularly adept at mirroring object-oriented programming paradigms, a facet not directly addressed by the aforementioned mechanizations. There are also several mechanizations of class-based languages [39, 19, 16]. However, neither of them supports refinement types.

SMT-based Deductive Reasoning in Program Verifiers Since its inception, SMT solvers have played a pivotal role in the automated verification of **functional** properties. Simplify [20] and ESC/Java [23] are among the earliest examples. Subsequently, a wave of advanced program verifiers like Dafny [36], Leon/Stainless [7, 27], F* [55] and Liquid Haskell [59, 60] have garnered attention in both academia and industry. Among those systems, Dafny and Leon/Stainless all support some object-oriented constructs. However, they lack RFJ's support for nominal subtyping and method inheritance. Recent scholarly work has delved into the foundational aspects of SMT-based deductive reasoning, focusing especially on the completeness problem [41, 38, 42]. However, the arguments of those papers are all set upon many-sorted logic, diverging from the order-sorted logic in our study.

On the other hand, SMT solvers have also been extensively used in verifying **heap** properties. The modeling and verification of those properties (typically in separation logic [45, 46]) are, in general, beyond the ability of vanilla SMT theories [40]. Despite these challenges, research has successfully identified certain significant fragments yielding effective decision procedures falling into the SMT realm [40, 50, 49]. Currently, RFJ is a purely functional calculus. However, we believe that it is promising to incorporate those

advancements to support the reasoning of heaps, considering imperative features are ambitious in class-based object-oriented languages,

9 Conclusion and Future Work

This paper introduces Refinement Featherweight Java (RFJ), advancing class-based refinement types with expressive refinements for comprehensive logical constraints. We mechanize RFJ in Coq, proving its soundness rigorously. We bridge the declarative calculus and algorithmic verification via a specified fragment in OS-FOL, making RFJ’s refinements accessible for SMT reasoning. The deliberate choice of FJ and OS-FOL for our fundamental framework facilitates important future extensions, such as polymorphic and imperative features, and a thorough exploration of algorithmic properties.

References

- 1 Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H. Schmitt, and Mattias Ulbrich, editors. *Deductive Software Verification - The KeY Book - From Theory to Practice*, volume 10001 of *Lecture Notes in Computer Science*. Springer, 2016. doi:10.1007/978-3-319-49812-6.
- 2 Nada Amin, K. Rustan M. Leino, and Tiark Rompf. Computing with an SMT solver. In Martina Seidl and Nikolai Tillmann, editors, *Tests and Proofs - 8th International Conference, TAP@STAF 2014, York, UK, July 24-25, 2014. Proceedings*, volume 8570 of *Lecture Notes in Computer Science*, pages 20–35. Springer, 2014. doi:10.1007/978-3-319-09099-3_2.
- 3 Brian E. Aydemir, Aaron Bohannon, Matthew Fairbairn, J. Nathan Foster, Benjamin C. Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich, and Steve Zdancewic. Mechanized metatheory for the masses: The poplmark challenge. In Joe Hurd and Thomas F. Melham, editors, *Theorem Proving in Higher Order Logics, 18th International Conference, TPHOLs 2005, Oxford, UK, August 22-25, 2005, Proceedings*, volume 3603 of *Lecture Notes in Computer Science*, pages 50–65. Springer, 2005. doi:10.1007/11541868_4.
- 4 Bruno Barras, Samuel Boutin, Cristina Cornes, Judicaël Courant, Jean-Christophe Filliatre, Eduardo Gimenez, Hugo Herbelin, Gerard Huet, Cesar Munoz, Chetan Murthy, et al. *The Coq proof assistant reference manual: Version 6.1*. PhD thesis, Inria, 1997.
- 5 Clark W. Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, volume 6806 of *Lecture Notes in Computer Science*, pages 171–177. Springer, 2011. doi:10.1007/978-3-642-22110-1_14.
- 6 Gavin M. Bierman, Andrew D. Gordon, Catalin Hritcu, and David E. Langworthy. Semantic subtyping with an SMT solver. *J. Funct. Program.*, 22(1):31–105, 2012. doi:10.1017/S0956796812000032.
- 7 Régis Blanc, Viktor Kuncak, Etienne Kneuss, and Philippe Suter. An overview of the leon verification system: verification by translation to recursive functions. In *Proceedings of the 4th Workshop on Scala, SCALA@ECOOP 2013, Montpellier, France, July 2, 2013*, pages 1:1–1:10. ACM, 2013. doi:10.1145/2489837.2489838.
- 8 Michael Borkowski, Niki Vazou, and Ranjit Jhala. Mechanizing refinement types. *Proc. ACM Program. Lang.*, 8(POPL):2099–2128, 2024. doi:10.1145/3632912.
- 9 Aaron R. Bradley and Zohar Manna. *The calculus of computation - decision procedures with applications to verification*. Springer, 2007. doi:10.1007/978-3-540-74113-8.
- 10 Joana Campos and Vasco T. Vasconcelos. Dependent types for class-based mutable objects. In Todd D. Millstein, editor, *32nd European Conference on Object-Oriented Programming, ECOOP 2018, July 16-21, 2018, Amsterdam, The Netherlands*, volume 109 of *LIPICs*, pages

- 13:1–13:28. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018. URL: <https://doi.org/10.4230/LIPICs.EC00P.2018.13>, doi:10.4230/LIPICs.EC00P.2018.13.
- 11 Giuseppe Castagna. Covariance and contravariance: Conflict without a cause. *ACM Trans. Program. Lang. Syst.*, 17(3):431–447, 1995. doi:10.1145/203095.203096.
- 12 Giuseppe Castagna, Giorgio Ghelli, and Giuseppe Longo. A semantics for lambda&-early: A calculus with overloading and early binding. In Marc Bezem and Jan Friso Groote, editors, *Typed Lambda Calculi and Applications, TLCA '93, Utrecht, The Netherlands, March 16-18, 1993, Proceedings*, volume 664 of *Lecture Notes in Computer Science*, pages 107–123. Springer, 1993. URL: <https://doi.org/10.1007/BFb0037101>, doi:10.1007/BFb0037101.
- 13 Arthur Charguéraud. The locally nameless representation. *J. Autom. Reason.*, 49(3):363–408, 2012. URL: <https://doi.org/10.1007/s10817-011-9225-2>, doi:10.1007/S10817-011-9225-2.
- 14 Zilin Chen. A hoare logic style refinement types formalisation. In *TyDe '22: 7th ACM SIGPLAN International Workshop on Type-Driven Development, Ljubljana, Slovenia, 11 September 2022*, pages 1–14. ACM, 2022. doi:10.1145/3546196.3550162.
- 15 Adam Chlipala. *Certified Programming with Dependent Types - A Pragmatic Introduction to the Coq Proof Assistant*. MIT Press, 2013. URL: <http://mitpress.mit.edu/books/certified-programming-dependent-types>.
- 16 Samuel da Silva Feitosa, Rodrigo Geraldo Ribeiro, and André Rauber Du Bois. Towards an extrinsic formalization of featherweight java in agda. *CLEI Electron. J.*, 24(3), 2021. URL: <https://doi.org/10.19153/cleiej.24.3.3>, doi:10.19153/CLEIEJ.24.3.3.
- 17 Leonardo de Moura and Sebastian Ullrich. The lean 4 theorem prover and programming language. In André Platzer and Geoff Sutcliffe, editors, *Automated Deduction - CADE 28 - 28th International Conference on Automated Deduction, Virtual Event, July 12-15, 2021, Proceedings*, volume 12699 of *Lecture Notes in Computer Science*, pages 625–635. Springer, 2021. doi:10.1007/978-3-030-79876-5_37.
- 18 Leonardo Mendonça de Moura and Nikolaj S. Bjørner. Z3: an efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008. doi:10.1007/978-3-540-78800-3_24.
- 19 Benjamin Delaware, William R. Cook, and Don S. Batory. Product lines of theorems. In Cristina Videira Lopes and Kathleen Fisher, editors, *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011, part of SPLASH 2011, Portland, OR, USA, October 22 - 27, 2011*, pages 595–608. ACM, 2011. doi:10.1145/2048066.2048113.
- 20 David Detlefs, Greg Nelson, and James B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005. doi:10.1145/1066100.1066102.
- 21 Khalil Djelloul, Thi-Bich-Hanh Dao, and Thom W. Frühwirth. Theory of finite or infinite trees revisited. *Theory Pract. Log. Program.*, 8(4):431–489, 2008. doi:10.1017/S1471068407003171.
- 22 Matthias Felleisen and Daniel P. Friedman. *A little Java, a few patterns*. MIT Press, 1996.
- 23 Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for java. In Jens Knoop and Laurie J. Hendren, editors, *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Berlin, Germany, June 17-19, 2002*, pages 234–245. ACM, 2002. doi:10.1145/512529.512558.
- 24 Catarina Gamboa, Paulo Canelas, Christopher Steven Timperley, and Alcides Fonseca. Usability-oriented design of liquid types for java. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*, pages 1520–1532. IEEE, 2023. doi:10.1109/ICSE48619.2023.00132.

- 912 25 Simon J. Gay, Nils Gesbert, António Ravara, and Vasco Thudichum Vasconcelos. Modular
913 session types for objects. *Log. Methods Comput. Sci.*, 11(4), 2015. doi:10.2168/LMCS-11(4:
914 12)2015.
- 915 26 James Gosling, William N. Joy, and Guy L. Steele Jr. *The Java Language Specification*.
916 Addison-Wesley, 1996.
- 917 27 Jad Hamza, Nicolas Voirol, and Viktor Kuncak. System FR: formalized foundations for
918 the stainless verifier. *Proc. ACM Program. Lang.*, 3(OOPSLA):166:1–166:30, 2019. doi:
919 10.1145/3360592.
- 920 28 Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight java: a minimal
921 core calculus for java and GJ. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450, 2001.
922 doi:10.1145/503502.503505.
- 923 29 Ranjit Jhala and Niki Vazou. Refinement types: A tutorial. *Found. Trends Program. Lang.*,
924 6(3-4):159–317, 2021. doi:10.1561/25000000032.
- 925 30 Milod Kazerounian, Niki Vazou, Austin Bourgerie, Jeffrey S. Foster, and Emina Torlak.
926 Refinement types for ruby. In Isil Dillig and Jens Palsberg, editors, *Verification, Model*
927 *Checking, and Abstract Interpretation - 19th International Conference, VMCAI 2018, Los*
928 *Angeles, CA, USA, January 7-9, 2018, Proceedings*, volume 10747 of *Lecture Notes in Computer*
929 *Science*, pages 269–290. Springer, 2018. doi:10.1007/978-3-319-73721-8_13.
- 930 31 Kenneth L. Knowles and Cormac Flanagan. Compositional reasoning and decidable checking for
931 dependent contract types. In Thorsten Altenkirch and Todd D. Millstein, editors, *Proceedings*
932 *of the 3rd ACM Workshop Programming Languages meets Program Verification, PLPV 2009,*
933 *Savannah, GA, USA, January 20, 2009*, pages 27–38. ACM, 2009. doi:10.1145/1481848.
934 1481853.
- 935 32 Daniel Kroening and Ofer Strichman. *Decision Procedures - An Algorithmic Point of View*.
936 Texts in Theoretical Computer Science. An EATCS Series. Springer, 2008. doi:10.1007/
937 978-3-540-74105-3.
- 938 33 Florian Lanzinger, Alexander Weigl, Mattias Ulbrich, and Werner Dietl. Scalability and
939 precision by combining expressive type systems and deductive verification. *Proc. ACM*
940 *Program. Lang.*, 5(OOPSLA):1–29, 2021. doi:10.1145/3485520.
- 941 34 Nico Lehmann and Éric Tanter. Formalizing simple refinement types in coq. In *2nd Interna-*
942 *tional Workshop on Coq for Programming Languages (CoqPL’16), St. Petersburg, FL, USA,*
943 2016.
- 944 35 K Rustan M Leino. This is boogie 2. *manuscript KRML*, 178(131):9, 2008.
- 945 36 K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In
946 Edmund M. Clarke and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence,*
947 *and Reasoning - 16th International Conference, LPAR-16, Dakar, Senegal, April 25-May 1,*
948 *2010, Revised Selected Papers*, volume 6355 of *Lecture Notes in Computer Science*, pages
949 348–370. Springer, 2010. doi:10.1007/978-3-642-17511-4_20.
- 950 37 K. Rustan M. Leino and Clément Pit-Claudel. Trigger selection strategies to stabilize program
951 verifiers. In Swarat Chaudhuri and Azadeh Farzan, editors, *Computer Aided Verification - 28th*
952 *International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings,*
953 *Part I*, volume 9779 of *Lecture Notes in Computer Science*, pages 361–381. Springer, 2016.
954 doi:10.1007/978-3-319-41528-4_20.
- 955 38 Christof Löding, P. Madhusudan, and Lucas Peña. Foundations for natural proofs and quantifier
956 instantiation. *Proc. ACM Program. Lang.*, 2(POPL):10:1–10:30, 2018. doi:10.1145/3158098.
- 957 39 Julian Mackay, Hannes Mehnert, Alex Potanin, Lindsay Groves, and Nicholas Cameron.
958 Encoding featherweight java with assignment and immutability using the coq proof assistant.
959 In Wei-Ngan Chin and Aquinas Hobor, editors, *Proceedings of the 14th Workshop on Formal*
960 *Techniques for Java-like Programs, FTfJP 2012, Beijing, China, June 12, 2012*, pages 11–19.
961 ACM, 2012. doi:10.1145/2318202.2318206.
- 962 40 P. Madhusudan, Gennaro Parlato, and Xiaokang Qiu. Decidable logics combining heap
963 structures and data. In Thomas Ball and Mooly Sagiv, editors, *Proceedings of the 38th ACM*

- 964 *SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin,*
 965 *TX, USA, January 26-28, 2011*, pages 611–622. ACM, 2011. doi:10.1145/1926385.1926455.
- 966 41 Adithya Murali, Lucas Peña, Ranjit Jhala, and P. Madhusudan. Complete first-order reasoning
 967 for properties of functional programs. *Proc. ACM Program. Lang.*, 7(OOPSLA2):1063–1092,
 968 2023. doi:10.1145/3622835.
- 969 42 Adithya Murali, Lucas Peña, Christof Löding, and P. Madhusudan. A first-order logic with
 970 frames. *ACM Trans. Program. Lang. Syst.*, 45(2):7:1–7:44, 2023. doi:10.1145/3583057.
- 971 43 Hanne Riis Nielson and Flemming Nielson. *Semantics with applications*, volume 104. Springer,
 972 1992.
- 973 44 Nathaniel Nystrom, Vijay A. Saraswat, Jens Palsberg, and Christian Grothoff. Constrained
 974 types for object-oriented languages. In Gail E. Harris, editor, *Proceedings of the 23rd Annual*
 975 *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and*
 976 *Applications, OOPSLA 2008, October 19-23, 2008, Nashville, TN, USA*, pages 457–474. ACM,
 977 2008. doi:10.1145/1449764.1449800.
- 978 45 Peter W. O’Hearn. Resources, concurrency, and local reasoning. *Theor. Comput. Sci.*, 375(1-
 979 3):271–307, 2007. URL: <https://doi.org/10.1016/j.tcs.2006.12.035>, doi:10.1016/J.TCS.
 980 2006.12.035.
- 981 46 Peter W. O’Hearn, John C. Reynolds, and Hongseok Yang. Local reasoning about programs that
 982 alter data structures. In Laurent Fribourg, editor, *Computer Science Logic, 15th International*
 983 *Workshop, CSL 2001. 10th Annual Conference of the EACSL, Paris, France, September 10-13,*
 984 *2001, Proceedings*, volume 2142 of *Lecture Notes in Computer Science*, pages 1–19. Springer,
 985 2001. doi:10.1007/3-540-44802-0_1.
- 986 47 Xinming Ou, Gang Tan, Yitzhak Mandelbaum, and David Walker. Dynamic typing with
 987 dependent types. In Jean-Jacques Lévy, Ernst W. Mayr, and John C. Mitchell, editors,
 988 *Exploring New Frontiers of Theoretical Informatics, IFIP 18th World Computer Congress,*
 989 *TC1 3rd International Conference on Theoretical Computer Science (TCS2004), 22-27 August*
 990 *2004, Toulouse, France*, volume 155 of *IFIP*, pages 437–450. Kluwer/Springer, 2004. doi:
 991 10.1007/1-4020-8141-3_34.
- 992 48 Benjamin C. Pierce. *Types and programming languages*. MIT Press, 2002.
- 993 49 Ruzica Piskac, Thomas Wies, and Damien Zufferey. Automating separation logic with trees
 994 and data. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification - 26th*
 995 *International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014,*
 996 *Vienna, Austria, July 18-22, 2014. Proceedings*, volume 8559 of *Lecture Notes in Computer*
 997 *Science*, pages 711–728. Springer, 2014. doi:10.1007/978-3-319-08867-9_47.
- 998 50 Andrew Reynolds, Radu Iosif, Cristina Serban, and Tim King. A decision procedure for
 999 separation logic in SMT. In Cyrille Artho, Axel Legay, and Doron Peled, editors, *Automated*
 1000 *Technology for Verification and Analysis - 14th International Symposium, ATVA 2016, Chiba,*
 1001 *Japan, October 17-20, 2016, Proceedings*, volume 9938 of *Lecture Notes in Computer Science*,
 1002 pages 244–261, 2016. doi:10.1007/978-3-319-46520-3_16.
- 1003 51 Reuben N. S. Rowe and Steffen van Bakel. Semantic types and approximation for featherweight
 1004 java. *Theor. Comput. Sci.*, 517:34–74, 2014. URL: <https://doi.org/10.1016/j.tcs.2013.08.017>,
 1005 doi:10.1016/J.TCS.2013.08.017.
- 1006 52 Georg Stefan Schmid and Viktor Kuncak. Smt-based checking of predicate-qualified types
 1007 for scala. In Aggelos Biboudis, Manohar Jonnalagedda, Sandro Stucki, and Vlad Ureche,
 1008 editors, *Proceedings of the 7th ACM SIGPLAN Symposium on Scala, SCALA@SPLASH*
 1009 *2016, Amsterdam, Netherlands, October 30 - November 4, 2016*, pages 31–40. ACM, 2016.
 1010 doi:10.1145/2998392.2998398.
- 1011 53 Peter H. Schmitt and Mattias Ulbrich. Axiomatization of typed first-order logic. In Nikolaj S.
 1012 Bjørner and Frank S. de Boer, editors, *FM 2015: Formal Methods - 20th International*
 1013 *Symposium, Oslo, Norway, June 24-26, 2015, Proceedings*, volume 9109 of *Lecture Notes in*
 1014 *Computer Science*, pages 470–486. Springer, 2015. doi:10.1007/978-3-319-19249-9_29.

- 1015 **54** Thomas Studer. Constructive foundations for featherweight java. In Reinhard Kahle, Peter
 1016 Schroeder-Heister, and Robert F. Stärk, editors, *Proof Theory in Computer Science, In-*
 1017 *ternational Seminar, PTCS 2001, Dagstuhl Castle, Germany, October 7-12, 2001, Proceed-*
 1018 *ings*, volume 2183 of *Lecture Notes in Computer Science*, pages 202–238. Springer, 2001.
 1019 doi:10.1007/3-540-45504-3_13.
- 1020 **55** Nikhil Swamy, Catalin Hritcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud,
 1021 Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss,
 1022 Jean Karim Zinzindohoue, and Santiago Zanella Béguelin. Dependent types and multi-
 1023 monadic effects in F. In Rastislav Bodík and Rupak Majumdar, editors, *Proceedings of the*
 1024 *43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages,*
 1025 *POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 256–270. ACM, 2016.
 1026 doi:10.1145/2837614.2837655.
- 1027 **56** William W. Tait. Intensional interpretations of functionals of finite type I. *J. Symb. Log.*,
 1028 32(2):198–212, 1967. doi:10.2307/2271658.
- 1029 **57** Emina Torlak and Rastislav Bodík. Growing solver-aided languages with rosette. In Antony L.
 1030 Hosking, Patrick Th. Eugster, and Robert Hirschfeld, editors, *ACM Symposium on New Ideas*
 1031 *in Programming and Reflections on Software, Onward! 2013, part of SPLASH '13, Indianapolis,*
 1032 *IN, USA, October 26-31, 2013*, pages 135–152. ACM, 2013. doi:10.1145/2509578.2509586.
- 1033 **58** Steffen van Bakel and Maribel Fernández. Normalization, approximation, and semantics
 1034 for combinator systems. *Theor. Comput. Sci.*, 290(1):975–1019, 2003. doi:10.1016/
 1035 S0304-3975(02)00548-0.
- 1036 **59** Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon L. Peyton Jones.
 1037 Refinement types for haskell. In Johan Jeuring and Manuel M. T. Chakravarty, editors,
 1038 *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming,*
 1039 *Gothenburg, Sweden, September 1-3, 2014*, pages 269–282. ACM, 2014. doi:10.1145/2628136.
 1040 2628161.
- 1041 **60** Niki Vazou, Anish Tondwalkar, Vikraman Choudhury, Ryan G. Scott, Ryan R. Newton, Philip
 1042 Wadler, and Ranjit Jhala. Refinement reflection: complete verification with SMT. *Proc. ACM*
 1043 *Program. Lang.*, 2(POPL):53:1–53:31, 2018. doi:10.1145/3158141.
- 1044 **61** Panagiotis Vekris, Benjamin Cosman, and Ranjit Jhala. Refinement types for typescript.
 1045 In Chandra Krintz and Emery D. Berger, editors, *Proceedings of the 37th ACM SIGPLAN*
 1046 *Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara,*
 1047 *CA, USA, June 13-17, 2016*, pages 310–325. ACM, 2016. doi:10.1145/2908080.2908110.
- 1048 **62** Peng Wang, Di Wang, and Adam Chlipala. Timl: a functional language for practical complexity
 1049 analysis with invariants. *Proc. ACM Program. Lang.*, 1(OOPSLA):79:1–79:26, 2017. doi:
 1050 10.1145/3133903.
- 1051 **63** Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Inf.*
 1052 *Comput.*, 115(1):38–94, 1994. URL: <https://doi.org/10.1006/inco.1994.1093>, doi:10.
 1053 1006/INCO.1994.1093.