Formalizing, Mechanizing, and Verifying Class-based Refinement Types

Ke Sun
Key Lab of HCST (PKU), MOE; School of Computer Science, Peking University, China

Di Wang[1]
Key Lab of HCST (PKU), MOE; School of Computer Science, Peking University, China

Sheng Chen
The Center for Advanced Computer Studies, UL Lafayette, USA

Meng Wang
University of Bristol, UK

Dan Hao
Key Lab of HCST (PKU), MOE; School of Computer Science, Peking University, China

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 (named RFJ), that is expressive and concise. We rigorously develop the soundness meta-theory of this calculus, followed by its mechanization in Coq. Finally, to ensure the calculus's verifiability, we propose an algorithmic verification approach based on a fragment of first-order logic (named LFJ), and implement this approach as a type checker.

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.ECOOP.2024.24

Supplementary Material *Software (ECOOP 2024 Artifact Evaluation approved artifact)*: `https://doi.org/10.4230/DARTS.10.2.29`

Funding This work is sponsored by National Natural Science Foundation of China Grant No. 62232001, NSF Grant 1750886, and EPSRC Grant EP/T008911/1.

Acknowledgements We thank the anonymous reviewers for their helpful comments.

1 Introduction

Refinement types have been widely used in class-based languages [47, 67, 10, 56, 26, 33, 36] 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, $\{\nu : C | \nu.f > 0\}$ 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.

[1] Corresponding author

© Ke Sun, Di Wang, Sheng Chen, Meng Wang, and Dan Hao;
licensed under Creative Commons License CC-BY 4.0

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, 35], 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 [30] 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. 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 selfification—a typing mechanism [66, 51] (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 [47, 67]) and non-standard constructs (e.g., ANF [10], existential types [67]). 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 [62] is ineffective for first-order languages like RFJ [55].

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 of order-sorted first-order logic (OS-FOL) [57]. We name this fragment as LFJ, and 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

```
class Pizza{
    {v:int|v>0} price(){return 1;}
    Pizza remA(){return new Pizza();}
    Pizza sell (this.price()>5){return this;}}
class Crust extends Pizza{
    {v:int|v>0} price(){return 1;}
    Pizza remA(){return new Crust();}
    Pizza sell (){return this;}}
class Cheese extends Pizza{
    p:Pizza
    {v:int|v>0} price(){return let pp = this.p.price() in pp + 1;}
    Pizza remA(){return new Cheese(this.p.remA());}}
class Anchovy extends Pizza{
    p:Pizza
    {v:int|v>0 && v>=this.p.price()} price(){return let pp = this.p.price() in pp;}
    Pizza remA(){return this.p.remA();}}
class MagicAnchovy extends Anchovy{
    {v:int|v>0 && v>this.p.price()} price(){return let pp = this.p.price() in pp + 1;}}
class Main{
    int assertSingleCheesePizza(x: {v:Pizza|v = new Cheese(new Crust())}{
        return 0; }
    int testRemA(){
        return let p1 = new Anchovy(new Cheese(new Crust())) in
        this.assertSingleCheesePizza(p1.remA());}}
```

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

checker that leverages Z3 [19] for checking the validity of LFJ formulas. The type checker is evaluated against a small yet representative benchmark derived from a Java textbook [23] and prior systems [65].

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 as the supplementary material of this paper.

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. Finally, 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 [23] (we add some methods to make it more interesting). The program models various pizzas and 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 of `this` (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 types[2].

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 selfification 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 selfification[3]

[2] 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.

[3] We name it *general* to distinguish from the cases like [34], where selfification is only used for variables.

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:

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

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 Finally, we consider a more interesting property concerning `price` and `remA`: the price of a pizza should not increase after removing all anchovies. We can express this by appending the following method to `Pizza`: `{bool|this.price()>=this.remA().price()} remA_noinc_price(){return true;}`, yielding the subtyping constraint below:

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

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

2.2 Algorithmic Verification

In conventional refinement type systems, subtyping constraints are typically discharged by SMT solvers, which facilitate automated reasoning and significantly reduce implementation efforts. We adapt this methodology by providing a logical encoding of RFJ into a dedicated order-sorted first-order logic, named LFJ. A detailed exposition of LFJ is provided in Section 5. Here, we offer a concise overview of it, drawing upon the examples discussed in Section 2.1.

EUFLIA After being encoded into LFJ, the subtyping constraints (1), (2), and (3) fall into the theory of Equality, Uninterpreted Functions, and Linear Integer Arithmetic (EUFLIA), a domain widely supported by contemporary SMT solvers [9, 4, 19]. LFJ incorporates EUFLIA for verifying those constraints.

Reasoning about Program Semantics We have illustrated in Section 2.1 that the verification of Constraint (4) requires knowledge about program semantics. We encode the knowledge with several axioms, which are discussed formally in Section 5.3. Currently, We illustrate them utilizing Constraint (4), which is translated to the LFJ formula shown below:

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

Here, the An_{cr}, Ch_{cr}, Cr_{cr} functions represent the constructors for the classes `An`, `Ch`, and `Cr`, respectively. An_{remA} represents the *conditional function* composed of the possible *implementations* of `Anchovy.remA`. The characterization of `Anchovy.remA` is shown below:

$$An_{remA}(this) = \begin{cases} Pi_{remA}(An_p(this)) & \text{if } this = An_{cr}(...) \\ Pi_{remA}(An_p(this)) & \text{if } this = Ma_{cr}(...) \end{cases}$$

This characterization redirects the function application to the implementation, depending on the *class* of *this* (although the implementations are the same). Since the two classes both use `Anchovy.remA`, all the implementations are the logic translation of the method body of `Anchovy.remA`. Here, An_p denotes the access function of the field `p` of the class `An`, while Pi_{remA} is the conditional function for `Pizza.remA`.

Because we know $p1$ is $An_{cr}(Ch_{cr}(Cr_{cr}()))$, we choose the first branch and deduce $\nu = Pi_{remA}(An_p(An_{cr}(Ch_{cr}(Cr_{cr}()))))$, which can be then handled by an axiom for An_p:

$$\forall p : Pi.\, An_p(An_{cr}(p)) = p$$

With this axiom, we can deduce $\nu = Pi_{remA}(Ch_{cr}(Cr_{cr}()))$. This time, we need to utilize the semantics of Pi_{remA}, and choose the branch for $this = Ch_{cr}(...)$, i.e., $Pi_{remA}(this) = Ch_{cr}(Pi_{remA}(Ch_p(this)))$, which lets us deduce $\nu = Ch_{cr}(Pi_{remA}(Ch_p((Ch_{cr}(Cr_{cr}())))))$. Following the routine we just outlined, we can deduce $\nu = Ch_{cr}(Cr_{cr}())$ eventually.

Verifying with Interfaces Even with the knowledge of program semantics, Constraint (5) still can not be verified. In particular, it would be translated to $\forall this : Pi.\, Pi_{price}(this) \geq Pi_{price}(Pi_{remA}(this))$, whose verification requires induction. In several previous refinement type systems for functional languages [66], induction is supported but is based on pattern matching and recursive functions. In object-oriented languages, pattern-matching constructs are typically not included. Thus, in this paper, we propose to utilize *interface*, a feature that is included in most object-oriented languages, to perform induction. To use interface-based induction in our case, we reimplement `Pizza` as an interface, which decomposes the proof obligation into subclasses implementing `Pizza`. To illustrate, we pick the proof of Constraint (5) for `Anchovy` as an example, 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 an SMT solver would not prove this property automatically. Before we explain the method body, we first give a brief informal proof to help understanding. The assumptions to facilitate the proof are listed below. Once the assumptions are within the proof context, the formula we want ($this.price() \geq this.remA().price()$) can be easily deduced within EUFLIA.

$$this.price() \geq this.p.price() \qquad (p_1) : a\ property\ of\ Anchovy.price$$
$$this.p.price() \geq this.p.remA().price() \qquad (p_2) : the\ induction\ hypothesis\ of\ this.p$$
$$this.remA() = this.p.remA() \qquad (p_3) : a\ property\ of\ Anchovy.remA$$

```
interface Pizza{
    {int|v>0} price()
    Pizza remA()
    Pizza sell (this.price()>5)
    {v:bool|this.price()>=this.remA().price()} remA_noinc_price() }
class Anchovy implements Pizza{
    p:Pizza
    {v:int|v>0 && v>=this.p.price()} price(){return let pp = this.p.price() in pp;}
    Pizza remA(){return this.p.remA();}
    {v:bool|this.price()>=this.remA().price()} remA_noinc_price(){
        return let p1 = this.price() in
        let p2 = this.p.remA_noinc_price() in true;}}
```

Figure 2 Proving `remA_noinc_price` for `Anchovy`.

The first assumption (property) is about `Anchovy.price`, and we have proved it in Section 2.1, so we can just refer to it as a lemma. In principle, even if we have not proved it, the solver can still automatically prove the property with the given program semantics and use it to prove the formula we want. However, leaving such a property to the solver often increases the searching time for proving the formula. Thus, we prove it as a separate property and introduce it (`p1` in the body of `remA_noinc_price()`) to the proof context so that the solver can use it directly. The second property is the induction hypothesis of `this.p` ($\mathit{this.p.price()} \geq \mathit{this.p.remA().price()}$), and we can utilize it by referring to the method `remA_noinc_price` of `this.p`. The third property asserts that for every `this:Anchovy`, it holds that `this.remA()=this.p.remA()`. This is obvious since no matter whether `this` is an `Anchovy` or a `MagicAnchovy`, calling `remA` on it would resolve to the same method implementation. Like p_1, this property is also automatically derivable, so we *can* leave it to the SMT solver, or explicitly prove it like we do for p_1. In particular, p_3 represents a special case where we can add an axiom to the solver, to spare the efforts of automatic search and manual proof. As a result, it is not included in the method body. We will discuss this further in Section 5.3.

We need not prove the same property for `MagicAnchovy`, since it inherits the property from `Anchovy`. Similarly, we can prove other properties such as the fact that no `Anchovy` exists after `remA`, as well as the idempotence of `remA`; all have been 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

Type Soundness The type system should be able to preclude evaluation from being stuck. Formally, if a closed expression is well-typed ($\varnothing \vdash e : t$), 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 selfification) do not break the promise of the base type system.

Logical Soundness The type system should 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 and General Selfification Different from previous class-based refinement type calculus, RFJ uses type substitution instead of ANF [32] or existential types [34]. This increases the generality and usability (detailed in Section 7). However, this also increases its meta-theoretical complexity, and several nonstandard lemmas about type substitution and typing have to be proved. Similarly, although general selfification increases the precision of the verification by recording term information in refinements, it affects substitution and preservation lemmas intricately and several non-standard properties (e.g., exactness, $\Gamma \vdash e : t$ *then* $\Gamma \vdash e : self(t, e)$) of it have to be proved for proving those lemmas.

First Order Functions The logical relation technique [62] is frequently employed in prior research [8, 29] 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 [64, 55]. 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 [52]. To delineate the extensions unique to RFJ, we highlight the extended features in *gray background*.

3.1 Syntax and Lookup Functions

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;

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

Figure 3 Syntax and subtyping.

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 [28] and the equality used in order-sorted logics [57].

Besides the extension, RFJ simplifies FJ in two aspects, widely adopted in prior studies [55, 10, 42, 27]. 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.

Finally, 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 ($<:$). The sub-nominal relation is a straightforward extension of FJ's subclassing to account for interface types. The base subtyping relation is also standard. Refinment 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, to make refinement subtyping transitive. In the remaining of this paper, we short refinement subtyping as subtyping when there is no ambiguity.

Field lookup $\boxed{fields(C) = \overline{t}\ \overline{f}}$

$$fields(Object) = \bullet$$

$$\frac{CT(C) = class\ C\ exds\ D\ imps\ \overline{I}\{\overline{t}\ \overline{f};\ K\ \overline{M}\} \qquad fields(D) = \overline{s}\ \overline{g}}{fields(C) = \overline{s}\ \overline{g}, \overline{t}\ \overline{f}}$$

C-method-type $\boxed{mtype(m, C) = p \to x : t \to r}$

$$\frac{CT(C) = class\ C\ exds\ D\ imps\ \overline{I}\{\overline{t}\ \overline{f};\ K\ \overline{M}\} \qquad r\ m\ (p,\ t\ x)\ \{return\ e;\} \in \overline{M}}{mtype(m, C) = p \to x : t \to r}$$

$$\frac{CT(C) = class\ C\ exds\ D\ imps\ \overline{I}\{\overline{t}\ \overline{f};\ K\ \overline{M}\} \qquad m\ is\ not\ defined\ in\ \overline{M}}{mtype(m, C) = mtype(m, D)}$$

C-method-body $\boxed{mbody(m, C) = (x, e)}$

$$\frac{CT(C) = class\ C\ exds\ D\ imps\ \overline{I}\{\overline{t}\ \overline{f};\ K\ \overline{M}\} \qquad r\ m\ (p,\ t\ x)\ \{return\ e;\} \in \overline{M}}{mbody(m, C) = (x, e)}$$

$$\frac{CT(C) = class\ C\ exds\ D\ imps\ \overline{I}\{\overline{t}\ \overline{f};\ K\ \overline{M}\} \qquad m\ is\ not\ defined\ in\ \overline{M}}{mbody(m, C) = mbody(m, D)}$$

Override $\boxed{override(m, C, D, p \to x : t \to r)}$

$$\frac{\begin{array}{c} mtype(m, D) = q \to x : t' \to r' \\ \implies (\varnothing \vdash \{C|q\} <: \{C|p\} \\ \varnothing, this : \{C|q\} \vdash t' <: t \\ \varnothing, this : \{C|q\}, x : t' \vdash r <: r') \end{array}}{override(m, C, D, p \to x : t \to r)}$$

I-method-type $\boxed{mtypei(m, I) = p \to x : t \to r}$

$$\frac{IT(I) = interface\ I\{\overline{Q}\} \qquad r\ m\ (p,\ t\ x)\ \in \overline{Q}}{mtypei(m, I) = p \to x : t \to r}$$

Implement $\boxed{implement(m, C, I, p \to x : t \to r)}$

$$\frac{\begin{array}{c} mtype(m, C) = q \to x : t' \to r' \\ \varnothing \vdash \{C|p\} <: \{C|q\} \\ \varnothing, this : \{C|p\} \vdash t <: t' \\ \varnothing, this : \{C|p\}, x : t \vdash r' <: r \end{array}}{implement(m, C, I, p \to x : t \to r)}$$

Interface implemented $\boxed{C \rhd I}$

$$\frac{IT(I) = interface\ I\{\overline{r\ m(p,\ t\ x)}\} \qquad implement(m, C, I, \overline{p \to x : t \to r})}{C \rhd I}$$

Figure 4 Auxiliary definitions.

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 $\to$ 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 ($override(m, C, D, p \to x : t \to r)$), which ensures the class C properly overrides the method m of the class D (if m does exists in D), 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 $\varnothing \vdash \{C|q\} <: \{C|p\}$ 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 [52].
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).

Finally, 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 ($implement(m, C, I, p \to x : t \to r)$) is a dual of the override relation, ensuring that the method m of interface I with type signature $p \to x : t \to r$, is overridden in the class C. Note that the formalization is different from override in that, *override* only requires the subtyping constraints to hold *if the method m exist*, while *implement* requires the method m does exist, and satisfies the subtyping constraints.

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 FJ[4], 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 disjunction, 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 [22]: two values are equal *iff* their corresponding trees are identical).

[4] To be specific, we align with the semantics in the textbook presentation [52], which diverges from the nondeterministic beta-reduction semantics in the original paper [30].

Evaluation $\boxed{e \rightsquigarrow e'}$

$$\frac{fields(C) = \overline{t}\ \overline{f}}{(new\ C(\overline{v})).f_i \rightsquigarrow v_i}$$

$$\frac{mbody(m, C) = (x, e_0)}{(new\ C(\overline{v})).m(v) \rightsquigarrow [this \mapsto (new\ C(\overline{v})); x \mapsto v]e_0}$$

$$\frac{e_0 \rightsquigarrow e_0'}{e_0.f \rightsquigarrow e_0'.f}$$

$$\frac{e_0 \rightsquigarrow e_0'}{e_0.m(e) \rightsquigarrow e_0'.m(e)}$$

$$\frac{e \rightsquigarrow e'}{v_0.m(e) \rightsquigarrow v_0.m(e')}$$

$$\frac{e_i \rightsquigarrow e_i'}{new\ C(\overline{v}, e_i, \overline{e}) \rightsquigarrow new\ C(\overline{v}, e_i', \overline{e})}$$

$$\frac{e \rightsquigarrow e'}{\neg e \rightsquigarrow \neg e'}$$

$$\frac{}{\neg b \rightsquigarrow \neg_p(b)}$$

$$\frac{e_0 \rightsquigarrow e_0'}{e_0 \oplus e \rightsquigarrow e_0' \oplus e}$$

$$\frac{e \rightsquigarrow e'}{v_0 \oplus e \rightsquigarrow v_0 \oplus e'}$$

$$\frac{\oplus\ ok\ v_0\ v_1}{v_0 \oplus v_1 \rightsquigarrow \oplus_p(v_0,\ v_1)}$$

$$\frac{e_0 \rightsquigarrow e_0'}{let\ x = e_0\ in\ e \rightsquigarrow let\ x = e_0'\ in\ e}$$

$$\frac{}{let\ x = v_0\ in\ e \rightsquigarrow [x \mapsto v_0]e}$$

Valid binary operation $\boxed{\oplus\ ok\ v_0\ v_1}$

$$\frac{}{\wedge\ ok\ b_0\ b_1}$$

$$\frac{}{\vee\ ok\ b_0\ b_1}$$

$$\frac{}{=\ ok\ v_0\ v_1}$$

Figure 5 Small-step semantics of RFJ.

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.

With the logical truth relation in hand, we can define the logical entailment relation ($\Gamma \vDash p$), which signifies the truth of a formula p under the type and logical constraints encoded within Γ. It requires that for every closing substitution that satisfies Γ (formally

$\theta ::= \theta, x : v \mid \varnothing$

Closing Substitution

$$\varnothing(e) = e$$

$$(\theta, x : v)(e) = [x \mapsto v]\theta(e)$$

Logical Truth $\boxed{\vDash p}$

$$\frac{p \rightsquigarrow^* true}{\vDash p}$$

Logical Entailment $\boxed{\Gamma \vDash p}$

$$\frac{\forall \theta \in [\![\Gamma]\!]. \vDash \theta(p)}{\Gamma \vDash p}$$

Logical Implication $\boxed{\Gamma \vDash p \Rightarrow q}$

$$\frac{\forall \theta \in [\![\Gamma]\!].if \vDash \theta(p)\ then \vDash \theta(q)}{\Gamma \vDash p \Rightarrow q}$$

Environment Denotation $\boxed{\theta \in [\![\Gamma]\!]}$

$$\frac{}{\varnothing \in [\![\varnothing]\!]}$$

$$\frac{v \in [\![\theta(t)]\!] \qquad \theta \in [\![\Gamma]\!]}{\theta, x : v \in [\![\Gamma, x : t]\!]}$$

Type Denotation $\boxed{v \in [\![t]\!]}$

$$\frac{\vDash [\nu \mapsto n]p}{n \in [\![\{\nu : int|p\}]\!]} \quad (\textsc{DenInt})$$

$$\frac{\vDash [\nu \mapsto b]p}{b \in [\![\{\nu : bool|p\}]\!]} \quad (\textsc{DenBool})$$

$$\frac{\begin{array}{c}\vDash [\nu \mapsto new\ C(\overline{v})]p \\ fields(C) = \overline{t}\ \overline{f} \qquad \overline{v} \in \overline{[\![[this \mapsto new\ C(\overline{v})]t]\!]}\end{array}}{new\ C(\overline{v}) \in [\![\{\nu : C|p\}]\!]} \quad (\textsc{DenClass})$$

$$\frac{w <:_b u \qquad v \in [\![\{\nu : w|p\}]\!]}{v \in [\![\{\nu : u|p\}]\!]} \quad (\textsc{Upcast})$$

Figure 6 Logical interpretation of RFJ.

defined later), the closing substitution must also satisfy the formula p (i.e., make it a truth). Similarly, we define the logical implication relation ($\Gamma \vDash p \Rightarrow q$), by requiring all closing 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:

$$\forall \theta \in [\![\Gamma]\!].\ if \vDash \theta(\nu = p1.remA())\ then \vDash \theta(\nu = new\ Ch(new\ Cr()))$$

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

Type and Environment Denotation Now, we formally define what is meant by "a substitution satisfies a type environment." This relation is defined by the environment denotation relation $\theta \in [\![\Gamma]\!]$, which is a natural lift of the type denotation relation ($v \in [\![t]\!]$), 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 ($v \in [\![\{u|p\}]\!]$) encapsulates two facets: the value v belongs to the base type u, and it satisfies the refinement p. DenClass 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 ($M\ ok\ in\ C$), class typing ($\mathcal{C}\ ok$), and interface typing ($\mathcal{I}\ ok$).

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 [52] 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 ($\lfloor \cdot \rfloor$) to convert refinement type environments to base type environments. The erase function is naturally lifted from the erase function of refinement types (i.e., $\lfloor \{\nu : w|p\} \rfloor = w$).

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

$$(1)\ \vdash_w \varnothing \qquad (2)\ \vdash_w \Gamma\ ,\ \Gamma \vdash_w t\ ,\ x \notin \Gamma \implies \vdash_w \Gamma, x : t$$

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:

$$\neg_t \doteq x : bool \to \{bool|v = \neg x\}$$

$$\wedge_t \doteq x : bool \to y : bool \to \{bool|v = x \wedge y\}$$

$$\vee_t \doteq x : bool \to y : bool \to \{bool|v = x \vee y\}$$

$$=_t \doteq x : \top \to y : \top \to \{bool|v = x = y\}$$

RFJ typing utilizes several mechanisms absent in FJ typing, i.e., well-formedness checking, type substitution, and general selfification. 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 `f` can be $\{\nu : int|\nu = this.h\}$, 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 $\{\nu : int|\nu = a.h\}$, by substituting *this* to a, as T-Field rule shows.

Type well-formedness $\boxed{\Gamma \vdash_w t}$

$$\frac{\lfloor\Gamma\rfloor, \nu : w \vdash_F p : \mathit{bool}}{\Gamma \vdash_w \{\nu : w | p\}}$$

RFJ Typing $\boxed{\Gamma \vdash e : t}$

$$\mathit{self}(\{\nu : w|p\}, e) = \{\nu : w | p \wedge \nu = e\}$$

$$\frac{x : t \in \Gamma \qquad \Gamma \vdash_w t}{\Gamma \vdash x : \mathit{self}(t, x)} \text{ (T-Var)}$$

$$\frac{}{\Gamma \vdash n : \{\mathit{int} | \nu = n\}} \text{ (T-Int)}$$

$$\frac{}{\Gamma \vdash b : \{\mathit{bool} | \nu = b\}} \text{ (T-Bool)}$$

$$\frac{\Gamma \vdash e_0 : \{C_0 | p\} \qquad \mathit{fields}(C_0) = \overline{t}\ \overline{f}}{\Gamma \vdash e_0.f_i : \mathit{self}([\mathit{this} \mapsto e_0]\ t_i, e_0.f_i)} \text{ (T-Field)}$$

$$\frac{\begin{array}{c}\mathit{mtype}(m, C_0) = q \to x : t \to r \\ \Gamma \vdash e_0 : \{C_0 | p\} \qquad \Gamma \vdash \{C_0|p\} <: \{C_0|q\} \\ \Gamma \vdash e : s \qquad \Gamma \vdash s <: [\mathit{this} \mapsto e_0]\ t\end{array}}{\Gamma \vdash e_0.m(e) : \mathit{self}([\mathit{this} \mapsto e_0; x \mapsto e]\ r, e_0.m(e))} \text{ (T-Invok)}$$

$$\frac{\begin{array}{c}\mathit{mtypei}(m, I_0) = q \to x : t \to r \\ \Gamma \vdash e_0 : \{I_0 | p\} \qquad \Gamma \vdash \{I_0|p\} <: \{I_0|q\} \\ \Gamma \vdash e : s \qquad \Gamma \vdash s <: [\mathit{this} \mapsto e_0] t\end{array}}{\Gamma \vdash e_0.m(e) : \mathit{self}([\mathit{this} \mapsto e_0; x \mapsto e] r, e_0.m(e))} \text{ (T-InvokI)}$$

$$\frac{\begin{array}{c}\mathit{fields}(C) = \overline{t}\ \overline{f} \\ \Gamma \vdash \overline{e} : \overline{s} \quad \Gamma \vdash \overline{s} <: [\mathit{this} \mapsto \mathit{new}\ C(\overline{e})]\ t\end{array}}{\Gamma \vdash \mathit{new}\ C(\overline{e}) : \mathit{self}(C, \mathit{new}\ C(\overline{e}))} \text{ (T-New)}$$

$$\frac{\Gamma \vdash e_0 : s_0 \qquad \Gamma, x : s_0 \vdash e : t \qquad \Gamma \vdash_w t}{\Gamma \vdash \mathit{let}\ x = e_0\ \mathit{in}\ e : \mathit{self}(t, \mathit{let}\ x = e_0\ \mathit{in}\ e)} \text{ (T-Let)}$$

$$\frac{\begin{array}{c}\Gamma \vdash e_0 : s_0 \\ \neg_t \doteq x : t_0 \to r \qquad \Gamma \vdash s_0 <: t_0\end{array}}{\Gamma \vdash \neg e_0 : [x \mapsto e_0]\ r} \text{ (T-Unop)}$$

$$\frac{\begin{array}{c}\oplus_t \doteq x : t_0 \to y : t \to r \\ \Gamma \vdash e_0 : s_0 \qquad \Gamma \vdash s_0 <: t_0 \\ \Gamma \vdash e : s \qquad \Gamma \vdash s <: [x \mapsto e_0]\ t\end{array}}{\Gamma \vdash e_0 \oplus e : [x \mapsto e_0; y \mapsto e]\ r} \text{ (T-Binop)}$$

$$\frac{\Gamma \vdash e : s \qquad \Gamma \vdash s <: t \qquad \Gamma \vdash_w t}{\Gamma \vdash e : t} \text{ (T-Sub)}$$

method typing $\boxed{M\ \mathit{ok\ in}\ C}$

$$\frac{\begin{array}{c}\mathit{CT}(C)\ =\ \mathit{class}\ C\ \mathit{exds}\ D\ \mathit{imps}\ \overline{I}\{...\} \\ \mathit{this} : \{C|p\}, x : t \vdash e_0 : r \\ \mathit{override}(m, C, D, p \to x : t \to r) \\ \varnothing \vdash_w \{C|p\} \qquad \mathit{this} : \{C|p\} \vdash_w t \\ \mathit{this} : \{C|p\}, x : t \vdash_w r\end{array}}{r\ m(p,\ t\ x)\{\mathit{return}\ e_0;\}\ \mathit{ok\ in}\ C}$$

class typing $\boxed{\mathcal{C}\ \mathit{ok}}$

$$\frac{\begin{array}{c}K = C(\overline{s}\ \overline{g}, \overline{t}\ \overline{f})\{\mathit{super}(\overline{g});\ \mathit{this}.\overline{f} = \overline{f};\} \\ \mathit{fields}(D) = \overline{s}\ \overline{g} \qquad \overline{M}\ \mathit{ok\ in}\ C \\ \varnothing, \mathit{this} : C \vdash_w \overline{t} \qquad C \triangleright \overline{I}\end{array}}{\mathit{class}\ C\ \mathit{exds}\ D\ \mathit{imps}\ \overline{I}\{\overline{t}\ \overline{f}; K\ \overline{M}\}\ \mathit{ok}}$$

interface method typing $\boxed{Q\ \mathit{ok\ in}\ I}$

$$\frac{\begin{array}{c}\varnothing \vdash_w \{I|p\} \qquad \mathit{this} : \{I|p\} \vdash_w t \\ \mathit{this} : \{I|p\}, x : t \vdash_w r\end{array}}{r\ m(p,\ t\ x)\ \mathit{ok\ in}\ I}$$

interface ok $\boxed{\mathcal{I}\ \mathit{ok}}$

$$\frac{\overline{Q}\ \mathit{ok\ in}\ I}{\mathit{interface}\ I\{\overline{Q}\}\ \mathit{ok}}$$

Figure 7 Typing relations of RFJ.

3. General selfification. Each rule except the subsumption rule and the rules for primitives (T-Int, T-Bool, T-Unop and T-Binop) is companioned with a selfification operation (self), ensuring the terms are always recorded in their types. selfification 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 $\{\mathit{int}|\nu = 2\}$ 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 $C \triangleright \overline{I}$ that ensures the interfaces are properly implemented.

Termination Finally, 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, $\varnothing \vdash new\ C().m() : \{bool|0 = 1\}$ 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.

$$\frac{\forall \theta \in [\![\Gamma]\!].\theta(e) \rightsquigarrow^* v}{\Gamma \downarrow e}\ terminating$$

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

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

▶ **Theorem 2** (Logical Soundness). *If* $\Gamma \vdash e : \{\nu : w|p\}$, $\vdash_w \Gamma$, *and* $\Gamma \downarrow e$ *, then* $\Gamma \vDash [\nu \mapsto e]p$.

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 [70]. 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.

4.1 Basic Properties

▶ **Lemma 3** (Evaluation Invariant). *If* $e \rightsquigarrow e'$*, then* $[x \mapsto e]p \rightsquigarrow^* v \Leftrightarrow [x \mapsto e']p \rightsquigarrow^* v$.

This lemma asserts that evaluation remains unaffected by the substitution with pre-or-post-evaluation terms, as the next lemma shows. 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 4** (Type-substitution Invariant). *If* $e \rightsquigarrow e'$*, then* $\Gamma \vdash [x \mapsto e]t <: [x \mapsto e']t$ *and* $\Gamma \vdash [x \mapsto e']t <: [x \mapsto e]t$.

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 Lemma 3.

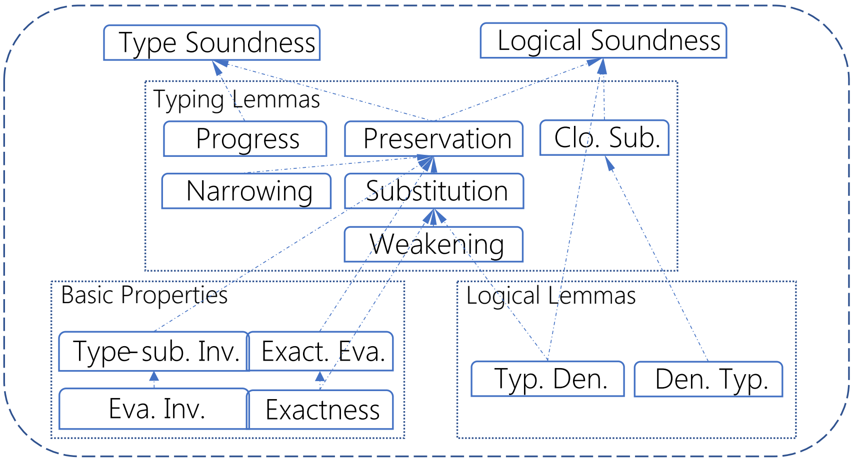

Figure 8 Proof Overview. Arrows signify the dependencies among lemmas and theorems.

▶ **Lemma 5** (Exactness). *If* $\Gamma \vdash e : t$ *then* $\Gamma \vdash e : self(t, e)$.

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 6, this lemma is important for the substitution lemma (Lemma 10).

▶ **Lemma 6** (Exactness Evaluation). *If* $e \rightsquigarrow e'$ *and* $\Gamma \vdash e' : t$ *then* $\Gamma \vdash e' : self(t, e)$.

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 above.

4.2 Logical Lemmas

▶ **Lemma 7** (Typing Denotation). *If* $\Gamma \vdash v : t$, *then* $\forall \theta \in [\![\Gamma]\!] . v \in [\![\theta(t)]\!]$.

This lemma states that typing implies denotation. It can be proved by induction on the typing judgment. This lemma is important for the substitution lemma (Lemma 10) and is a milestone for logical soundness, as we discuss in Section 4.5.

▶ **Lemma 8** (Denotation Typing). *If* $v \in [\![t]\!]$ *and* $\varnothing \vdash_w t$, *then* $\varnothing \vdash v : t$.

This lemma states that denotation implies typing, which is crucial for Lemma 14. The basic proof idea is to first construct a "ground type" for v: $\varnothing \vdash v : \{\nu : w | \nu = v\}$, where w is the *inherent* base type of the value (*int* for n, *bool* for b, and C for *new* $C(...)$), and then link the "ground type" to t by $\varnothing \vdash \{\nu : w | \nu = v\} <: t$, which holds due to $v \in [\![t]\!]$.

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 9** (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 10** (Substitution). *for any distinct variables x and y not in Γ and Γ':*
1. If $\Gamma, x : r_x, y : r_y, \Gamma' \vDash p \Rightarrow q$ *and* $\Gamma \vdash v_x : r_x$, $\Gamma \vdash v_y : [x \mapsto v_x]r_y$*, then* $\Gamma, [x \mapsto v_x; y \mapsto v_y]\Gamma' \vDash [x \mapsto v_x; y \mapsto v_y]p \Rightarrow [x \mapsto v_x; y \mapsto v_y]q$.
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 5) and weakening lemma (shown below). The other cases are easy.

▶ **Lemma 11** (Weakening). *for any variable x not in Γ, Γ', p, q, s and t:*
1. *If* $\Gamma, \Gamma' \vDash p \Rightarrow q$*, then* $\Gamma, x : r, \Gamma' \vDash p \Rightarrow q$.
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 12** (Progress). *If* $\varnothing \vdash e : t$ *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 13** (Preservation). *If* $\varnothing \vdash e : t$ *and* $e \rightsquigarrow e'$*, then* $\varnothing \vdash e' : t$.

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 selfification and type substitution, Lemma 6 and Lemma 4 must also be utilized.

4.3.3 Closing Substitution

▶ **Lemma 14** (Closing Substitution). *If* $\Gamma \vdash e : t$*, then* $\forall \theta \in [\![\Gamma]\!]. \varnothing \vdash \theta(e) : \theta(t)$.

The closing substitution lemma bears a similarity with the substitution lemma (Lemma 10). They both concern the invariance of typing under substitution. The closing substitution lemma can be proved by induction on typing. Most of the cases are standard, except for the variable case, which requires proving $\varnothing \vdash \theta(x) : \theta(t)$ under $\Gamma \vdash x : t$. Since θ is a denotation of Γ, we know that x must be in θ and $\theta(x) \in [\![\theta(t)]\!]$. Thus, Lemma 8 can be applied to construct the expected typing judgment.

4.4 Type Soundness

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

▶ **Corollary 15** (Type Soundness). *If* $\varnothing \vdash e : t$ *and* $e \rightsquigarrow^* e'$, *then* e' *is a value or* $\exists e''. e' \rightsquigarrow e''$.

Type soundness is an easy corollary of progress and preservation [70].

4.5 Logical Soundness

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

▶ **Corollary 16** (Logical Soundness). *If* $\Gamma \vdash e : \{\nu : w|p\}$, $\vdash_w \Gamma$, *and* $\Gamma \downarrow e$ *, then* $\Gamma \vDash [\nu \mapsto e]p$.

The key to proving logical soundness is to observe that it can be reduced to closed logical soundness (shown below) if we can derive a corresponding closed typing judgment given any typing judgment. This is facilitated by the closing substitution lemma (Lemma 14).

▶ **Theorem 17** (Closed Logical Soundness). *If* $\varnothing \vdash e : \{\nu : w|p\}$ *and* $\downarrow e$, *then* $\vDash [\nu \mapsto e]p$.

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

1. Due to the preservation lemma, $\varnothing \vdash v : \{\nu : w|p\}$.
2. Due to the typing denotation lemma, $v \in [\![\{\nu : w|p\}]\!]$, thus $\vDash [\nu \mapsto v]p$.
3. Lastly, we can apply the evaluation invariant lemma to get $\vDash [\nu \mapsto e]p$.

5 Logical Encoding: LFJ

Following the standard procedure as outlined in, e.g., [6], we convert RFJ to an algorithmic bidirectional type system. The only judgment whose algorithmic property was unexplored was the class-based refinement subtyping. In this section, we present an encoding of RFJ to an order-sorted first-order logic [57], named LFJ, which gives a convenient axiomatic approach to determine RFJ refinement subtyping by invoking logical decision procedures.

5.1 Language

Figure 9 presents the syntax of LFJ. The constant symbols (c) are for RFJ values. We assume each RFJ value has a corresponding LFJ constant symbol. The function symbols (g) are for methods (N_m), field selectors (C_f), class constructors (C_{cr}), and primitive operations in RFJ. We associate methods with nominal names and field selectors with class names, for attributing more precise semantics (detailed later). Note that interfaces have no fields. The terms in LFJ do not contain quantification: they are viewed as implicitly quantified and a universal quantification would be added to the outermost to close them. Sorts in LFJ consist of $\top$, Int, $Bool$, and N. The sorts have an apparent correspondence with RFJ base types. We denote $|w|$ as the translation of a base type w to its sort, and $|t|$ as the translation from a refinement type t to its sort. The subsort relation $\sqsubseteq$ is straightforwardly translated from the base-subtyping relation. The signatures of functions are also translated from their RFJ 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

$e, p ::=$		*terms*:
	x	*variable*
	c	*constant*
	$g(\overline{e})$	*apply*
	$let\ x = e\ in\ e$	*let binding*
$g ::=$	$N_m \mid C_f \mid C_{cr} \mid \neg \mid \oplus$	*functions*

Sorts

$s ::=$	$\top \mid Int \mid Bool \mid N$	*sorts*
$\lvert w \rvert =$	$match\ w\ with$ $\lvert \top \Rightarrow \top \lvert int \Rightarrow Int \lvert bool \Rightarrow Bool \lvert N \Rightarrow N$	*base translation*
$\lvert t \rvert =$	$\lvert \lfloor t \rfloor \rvert$	*type translation*
$\sqsubseteq \doteq$	$\mid <:_b \mid$	*subsort*

Term Translation

$$
\begin{aligned}
|x| &= x \\
|v| &= c_v \\
|\neg e_0| &= \neg(|e_0|) \\
|e_0 \oplus e_1| &= \oplus(|e_0|, |e_1|) \\
|new\ C(\overline{e})| &= C_{cr}(\overline{|e|}) \\
|e_0.m(e_1)| &= \delta(e_0)_m(|e_0|, |e_1|) \\
|e_0.f| &= \delta(e_0)_f(|e_0|) \\
|let\ x = e_0\ in\ e| &= let\ x = |e_0|\ in\ |e|
\end{aligned}
$$

Environment Translation

$$
\begin{aligned}
|\varnothing| &= true \\
|\Gamma, x : \{\nu : u | p\}| &= |\Gamma| \wedge |[\nu \mapsto x]p|
\end{aligned}
$$

Figure 9 LFJ syntax and translation.

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

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

5.2 Intended Model

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 [58, 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.

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

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

G_C is the finite term trees [22] 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 [57].

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, [46].

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

$$\frac{w <:_b u \qquad \mathcal{A} \vDash_L \forall \overline{x}. |\Gamma| \wedge |p| \Rightarrow |q|}{\Gamma \vdash \{\nu : w|p\} <:_L \{\nu : u|q\}}\textit{A-Subtyping}$$

where $\vDash_L$ is the normal semantics of OS-FOL [57]. We assume all variables in Γ are distinct and are not ν. We use a universal quantification $\forall \overline{x}$ to close the formula, where $\overline{x}$ is the variables used in Γ, p and q.

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 are also true in $\mathcal{A}$, and vice versa. Translation-substitution distributivity states it does not matter whether we apply a closing substitution prior to or after the translation.

▶ **Proposition 18** (Semantic Equivalence). $\mathcal{A} \vDash_L |p| \Leftrightarrow \vDash p$

▶ **Proposition 19** (Translation-substitution Distributivity). $\mathcal{A} \vDash_L |\theta|(|p|) \Leftrightarrow \mathcal{A} \vDash_L |\theta(p)|$

▶ **Corollary 20.** *If* $\Gamma \vdash s <:_L t$*, then* $\Gamma \vdash s <: t$*.*

Proof. We give a brief proof sketch of Corollary 20 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 $\forall \theta \in [\![\Gamma, \nu : w]\!].\ if \vDash \theta(p)\ then \vDash \theta(q)$. By $\Gamma \vdash s <:_L t$, we have $\mathcal{A} \vDash_L \forall \overline{x}. |\Gamma| \wedge |p| \Rightarrow |q|$, which gives us $\forall \sigma. \mathcal{A} \vDash_L \sigma(|\Gamma| \wedge |p|) \Rightarrow \mathcal{A} \vDash_L \sigma(|q|)$ (by the semantics of OS-FOL). Pick σ as $|\theta|$, due to Propositions 18 and 19, we have $\mathcal{A} \vDash_L |\theta|(|\Gamma| \wedge |p|)$, which let us deduce $\mathcal{A} \vDash_L |\theta|(|q|)$. Using Propositions 18 and 19 again, but in the reverse direction, we have $\vDash \theta(q)$. ◀

5.3 Theory

To utilize the capability of deductive reasoning for checking subtyping algorithmically, we axiomatize the intended model $\mathcal{A}$ by a theory $\mathcal{T}_{\mathcal{J}}$. $\mathcal{T}_{\mathcal{J}}$ includes 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 and explain them in this section.

(1) $\textit{generate} : \twoheadrightarrow\ \forall x : N.\ \bigvee_{C \sqsubseteq N} \exists \overline{y} : |fs(C)_t|.x = C(\overline{y})$

(2) $\textit{inject} : \twoheadrightarrow\ \forall \overline{x} : |fs(C)_t|, \overline{y} : |\overline{t}|.\ C_{cr}(\overline{x}) = C_{cr}(\overline{y}) \Rightarrow \overline{x} = \overline{y}$

(3) $\textit{discriminate} : C \neq D \twoheadrightarrow\ \forall \overline{x} : |fs(C)_t|, \overline{y} : |fs(D)_t|.\ C_{cr}(\overline{x}) \neq D_{cr}(\overline{y})$

(4) $\textit{access} : fs(C) = \overline{f}\ \overline{t},\ C' \sqsubseteq C\ \twoheadrightarrow\ \forall \overline{x} : |fs(C')_t|.\ C_{f_i}(C'_{cr}(\overline{x})) = x_i$

(5) $\textit{invoke} : mt(m, N)_x = t_x,\ C \sqsubseteq N,\ mb(m, C) = (x, e) \twoheadrightarrow$

$\quad \forall o : N, x : |t_x|, \overline{d} : |fs(C)_t|.\ o = C(\overline{d}) \Rightarrow N_m(o, x) = |e|$

The above listing gives five axiom schemata. The symbol $\twoheadrightarrow$ means "instantiate": if the condition on the left is satisfied, one can instantiate an axiom following the schema on the right. The symbols *fs*, *mb*, fs_t, and mt_x short for *fields*, *mbody*, the type part of *fields*, and the parameter part of *mtype* (or *mtypei* for interfaces), respectively.

The axiom schemata are straightforward given the intended model $\mathcal{A}$. However, they may not be as efficient as we want. To address this, we add two derivable properties as axioms, to speed up deductive reasoning. The first covers cases where the branches of a method direct to the same implementation. We have seen such a case in our example: *Anchovy.remA* has two branches that direct to the same implementation. We call these methods like *Anchovy.remA* *final methods*. Final methods have the same implementation on all branches, and there is no need to actually do the branching. We axiomatize their semantics using the axiom schema (6) shown below. An instantiation of (6) gives the property p_3 we discussed in Section 2.2. The second covers cases where a method is called on a subclass of the declared type. For example, suppose we have $\nu = Pi_{remA}(x)$ and $x : An$, and we want to prove $\nu = An_{remA}(x)$. With basic axioms (1) through (5) above, we have to first deduce the fact that x can only be $An(...)$ or $Ma(...)$, then analyze the semantics of Pi_{remA} and An_{remA} for those two cases, and finally deduce that the equality holds for both cases. However, this is mostly redundant: Pi_{remA} and An_{remA} are the same function if the first argument is known to be an An. We axiomatize this fact using schema (7) shown below. For certain cases involving comparing method-call results, axiom schema (7) can speed up reasoning significantly.

(6) $\textit{final} : mt(m, C)_x = t_x,\ C.m\ \textit{final},\ mb(m, C) = (x, e),\ \twoheadrightarrow\ \forall o : C, x : |t_x|.\ C_m(o, x) = |e|$

(7) $\textit{override} : N' \sqsubseteq N,\ mt(m, N')_x = t_x\ \twoheadrightarrow\ \forall o : N', x : |t_x|.\ N_m(o, x) = N'_m(o, x)$

Encoding into Many-sorted Logic The aforementioned axioms are defined in OS-FOL and should be used in order-sorted deductive reasoning. Unfortunately, we are not aware of any SMT solver that supports order-sortedness. Thus, we translate the axioms into many-sorted logic following the strategy suggested by Leino [38]. The translation of primitive data types is straightforward. For objects, a unified sort *Object* is designated. We then introduce a sort *Nominal* to encompass all nominal entities, i.e., classes and interfaces in the targeting RFJ program. We also declare the sub-nominal relation between those entities. The association of nominal information with objects is facilitated through the *Tag* function, which relates objects with their nominal identifiers. The sort requirements become sub-nominal checkings on tags, e.g., instead of $\forall x : C.\, p(x)$, we use $\forall x : \textit{Object}.\ \textit{sub-nominal}(Tag(x), C) \Rightarrow p(x)$.

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

6 Mechanization and Implementation

6.1 Coq Mechanization

We mechanize the meta-theory of RFJ in Coq. There are two major technical challenges around the mechanization. (1) *Binders.* Handling binders is cumbersome and complex [3], especially considering the number of binder structures present in RFJ (e.g., methods, let-bindings, and refinement types). To address this issue, we adopt the locally nameless representation [13]. Although the locally nameless representation has been widely used in mechanizing functional languages [8, 29, 13], to the best of our knowledge, ours is the first mechanization of a class-based language that utilizes this technique. (2) *Nested Inductive Types.* The presence of nested inductive types within our definitions poses a significant challenge; that is, the default induction principles generated by Coq fell short when proving the most critical properties. To mitigate this issue, we specify the custom induction principles for a range of inductive definitions (e.g., terms, typing judgments, and big-step semantics), following the classical methodology [15].

We briefly overview the structure of the mechanization, which 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 [19] 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 [32], 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.

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 [23], as well as some interesting examples inspired by previous work [65]. 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 in Table 1.

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.

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_nodec_size
stlc	meta-theories	307	map_extend_included, typing_weakening

Table 1 Several representative examples.

Type Substitution vs ANF and Existential types In the realm of refinement type systems, the conventional strategy often involves leveraging ANF [32, 37] or existential types [47, 34, 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 [41, 44] 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 costing only a little time 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 [47]). 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 [66], 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 $\{\nu : Pizza | \nu = this.p.remA()\}$ 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 [66]). With axiomatization, instantiation is delegated to the SMT solver, although special mechanisms such as *trigger/fuel* [2, 40] 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 Class is an important and time-honored abstraction in object-oriented programming [16, 59, 25, 31], with numerous pieces of literature devoted [69, 60, 5, 55] to its extensions. In particular, many works have focused on class-based refinement type systems. For example, Nystrom et al. [47] formalize core X10 as a refinement type system. However, they focus only on the functional aspects. Vekris et al. [67] 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. [56] present qualified type, a form of refinement type, and offer an in-depth discussion on qualifier inference. Gamboa et al. [26] 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), and thus have significant issues concerning soundness and expressiveness, as we have explained before. Meanwhile, although there are systems [33, 63] exploring the support for more expressive refinements, their approach is mainly pragmatic (i.e., they both rely on external verification tools to support the expressive refinements), which complicates the analysis of their meta-theoretical properties further.

This work addresses the expressiveness and soundness issues in a fundamental way, 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. [37] 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. [68] 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. [29] formalize a polymorphic refinement type system in Coq. They also employ an operational-semantics-based logical interpretation (named reduciblity 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 selfification 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. [26] opt for a categorical-theoretical perspective for logical interpretation in their mechanized refinement type system in Lean [18], 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 [42, 20, 17]. 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 [21] and ESC/Java [24] are among the earliest examples. Subsequently, a wave of advanced program verifiers like Dafny [39], Leon/Stainless [7, 29], F* [61] and Liquid Haskell [65, 66] 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 [44, 41, 45]. 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 [48, 49]) are, in general, beyond the ability of vanilla SMT theories [43]. Despite these challenges, research has successfully identified certain significant fragments yielding effective decision procedures falling into the SMT realm [43, 54, 53]. 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 [50].

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 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.

5 Lorenzo Bettini, Viviana Bono, Mariangiola Dezani-Ciancaglini, Paola Giannini, and Betti Venneri. Java & lambda: a featherweight story. *Log. Methods Comput. Sci.*, 14(3), 2018. doi:10.23638/LMCS-14(3:17)2018.

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.ECOOP.2018.13, doi:10.4230/LIPICS.ECOOP.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, International Conference on 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 William R. Cook. On understanding data abstraction, revisited. In Shail Arora and Gary T. Leavens, editors, *Proceedings of the 24th Annual ACM SIGPLAN Conference on Object-*

Oriented Programming, Systems, Languages, and Applications, OOPSLA 2009, October 25-29, 2009, Orlando, Florida, USA, pages 557–572. ACM, 2009. doi:10.1145/1640089.1640133.

17 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.

18 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.

19 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.

20 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.

21 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.

22 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.

23 Matthias Felleisen and Daniel P. Friedman. *A little Java, a few patterns*. MIT Press, 1996.

24 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.

25 Maurizio Gabbrielli, Simone Martini, and Saverio Giallorenzo. *Programming languages: principles and paradigms, Second Edition*. Undergraduate Topics in Computer Science. Springer, 2023. doi:10.1007/978-3-031-34144-1.

26 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.

27 Simon J. Gay, Nils Gesbert, António Ravara, and Vasco Thudichum Vasconcelos. Modular session types for objects. *Log. Methods Comput. Sci.*, 11(4), 2015. doi:10.2168/LMCS-11(4:12)2015.

28 James Gosling, William N. Joy, and Guy L. Steele Jr. *The Java Language Specification*. Addison-Wesley, 1996.

29 Jad Hamza, Nicolas Voirol, and Viktor Kuncak. System FR: formalized foundations for the stainless verifier. *Proc. ACM Program. Lang.*, 3(OOPSLA):166:1–166:30, 2019. doi:10.1145/3360592.

30 Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight java: a minimal core calculus for java and GJ. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450, 2001. doi:10.1145/503502.503505.

31 Bart Jacobs. Objects and classes, co-algebraically. In Burkhard Freitag, Cliff B. Jones, Christian Lengauer, and Hans-Jörg Schek, editors, *Object Orientation with Parallelism and*

Persistence (the book grow out of a Dagstuhl Seminar in April 1995), pages 83–103. Kluwer Academic Publishers, 1995.

32 Ranjit Jhala and Niki Vazou. Refinement types: A tutorial. *Found. Trends Program. Lang.*, 6(3-4):159–317, 2021. doi:10.1561/2500000032.

33 Milod Kazerounian, Niki Vazou, Austin Bourgerie, Jeffrey S. Foster, and Emina Torlak. Refinement types for ruby. In Isil Dillig and Jens Palsberg, editors, *Verification, Model Checking, and Abstract Interpretation - 19th International Conference, VMCAI 2018, Los Angeles, CA, USA, January 7-9, 2018, Proceedings*, volume 10747 of *Lecture Notes in Computer Science*, pages 269–290. Springer, 2018. doi:10.1007/978-3-319-73721-8_13.

34 Kenneth L. Knowles and Cormac Flanagan. Compositional reasoning and decidable checking for dependent contract types. In Thorsten Altenkirch and Todd D. Millstein, editors, *Proceedings of the 3rd ACM Workshop Programming Languages meets Program Verification, PLPV 2009, Savannah, GA, USA, January 20, 2009*, pages 27–38. ACM, 2009. doi:10.1145/1481848.1481853.

35 Daniel Kroening and Ofer Strichman. *Decision procedures - an algorithmic point of view*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2008. doi:10.1007/978-3-540-74105-3.

36 Florian Lanzinger, Alexander Weigl, Mattias Ulbrich, and Werner Dietl. Scalability and precision by combining expressive type systems and deductive verification. *Proc. ACM Program. Lang.*, 5(OOPSLA):1–29, 2021. doi:10.1145/3485520.

37 Nico Lehmann and Éric Tanter. Formalizing simple refinement types in coq. In *2nd International Workshop on Coq for Programming Languages (CoqPL'16), St. Petersburg, FL, USA*, 2016.

38 K Rustan M Leino. This is boogie 2. *manuscript KRML*, 178(131):9, 2008.

39 K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In Edmund M. Clarke and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16, Dakar, Senegal, April 25-May 1, 2010, Revised Selected Papers*, volume 6355 of *Lecture Notes in Computer Science*, pages 348–370. Springer, 2010. doi:10.1007/978-3-642-17511-4_20.

40 K. Rustan M. Leino and Clément Pit-Claudel. Trigger selection strategies to stabilize program verifiers. In Swarat Chaudhuri and Azadeh Farzan, editors, *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part I*, volume 9779 of *Lecture Notes in Computer Science*, pages 361–381. Springer, 2016. doi:10.1007/978-3-319-41528-4_20.

41 Christof Löding, P. Madhusudan, and Lucas Peña. Foundations for natural proofs and quantifier instantiation. *Proc. ACM Program. Lang.*, 2(POPL):10:1–10:30, 2018. doi:10.1145/3158098.

42 Julian Mackay, Hannes Mehnert, Alex Potanin, Lindsay Groves, and Nicholas Cameron. Encoding featherweight java with assignment and immutability using the coq proof assistant. In Wei-Ngan Chin and Aquinas Hobor, editors, *Proceedings of the 14th Workshop on Formal Techniques for Java-like Programs, FTfJP 2012, Beijing, China, June 12, 2012*, pages 11–19. ACM, 2012. doi:10.1145/2318202.2318206.

43 P. Madhusudan, Gennaro Parlato, and Xiaokang Qiu. Decidable logics combining heap structures and data. In Thomas Ball and Mooly Sagiv, editors, *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, pages 611–622. ACM, 2011. doi:10.1145/1926385.1926455.

44 Adithya Murali, Lucas Peña, Ranjit Jhala, and P. Madhusudan. Complete first-order reasoning for properties of functional programs. *Proc. ACM Program. Lang.*, 7(OOPSLA2):1063–1092, 2023. doi:10.1145/3622835.

45 Adithya Murali, Lucas Peña, Christof Löding, and P. Madhusudan. A first-order logic with frames. *ACM Trans. Program. Lang. Syst.*, 45(2):7:1–7:44, 2023. doi:10.1145/3583057.

46 Hanne Riis Nielson and Flemming Nielson. *Semantics with applications*, volume 104. Springer, 1992.

47 Nathaniel Nystrom, Vijay A. Saraswat, Jens Palsberg, and Christian Grothoff. Constrained types for object-oriented languages. In Gail E. Harris, editor, *Proceedings of the 23rd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2008, October 19-23, 2008, Nashville, TN, USA*, pages 457–474. ACM, 2008. doi:10.1145/1449764.1449800.

48 Peter W. O'Hearn. Resources, concurrency, and local reasoning. *Theor. Comput. Sci.*, 375(1-3):271–307, 2007. URL: https://doi.org/10.1016/j.tcs.2006.12.035, doi:10.1016/J.TCS.2006.12.035.

49 Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In Laurent Fribourg, editor, *Computer Science Logic, 15th International Workshop, CSL 2001. 10th Annual Conference of the EACSL, Paris, France, September 10-13, 2001, Proceedings*, volume 2142 of *Lecture Notes in Computer Science*, pages 1–19. Springer, 2001. doi:10.1007/3-540-44802-0_1.

50 Johan Östlund and Tobias Wrigstad. Welterweight java. In Jan Vitek, editor, *Objects, Models, Components, Patterns, 48th International Conference, TOOLS 2010, Málaga, Spain, June 28 - July 2, 2010. Proceedings*, volume 6141 of *Lecture Notes in Computer Science*, pages 97–116. Springer, 2010. doi:10.1007/978-3-642-13953-6_6.

51 Xinming Ou, Gang Tan, Yitzhak Mandelbaum, and David Walker. Dynamic typing with dependent types. In Jean-Jacques Lévy, Ernst W. Mayr, and John C. Mitchell, editors, *Exploring New Frontiers of Theoretical Informatics, IFIP 18th World Computer Congress, TC1 3rd International Conference on Theoretical Computer Science (TCS2004), 22-27 August 2004, Toulouse, France*, volume 155 of *IFIP*, pages 437–450. Kluwer/Springer, 2004. doi:10.1007/1-4020-8141-3_34.

52 Benjamin C. Pierce. *Types and programming languages.* MIT Press, 2002.

53 Ruzica Piskac, Thomas Wies, and Damien Zufferey. Automating separation logic with trees and data. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, volume 8559 of *Lecture Notes in Computer Science*, pages 711–728. Springer, 2014. doi:10.1007/978-3-319-08867-9_47.

54 Andrew Reynolds, Radu Iosif, Cristina Serban, and Tim King. A decision procedure for separation logic in SMT. In Cyrille Artho, Axel Legay, and Doron Peled, editors, *Automated Technology for Verification and Analysis - 14th International Symposium, ATVA 2016, Chiba, Japan, October 17-20, 2016, Proceedings*, volume 9938 of *Lecture Notes in Computer Science*, pages 244–261, 2016. doi:10.1007/978-3-319-46520-3_16.

55 Reuben N. S. Rowe and Steffen van Bakel. Semantic types and approximation for featherweight java. *Theor. Comput. Sci.*, 517:34–74, 2014. URL: https://doi.org/10.1016/j.tcs.2013.08.017, doi:10.1016/J.TCS.2013.08.017.

56 Georg Stefan Schmid and Viktor Kuncak. Smt-based checking of predicate-qualified types for scala. In Aggelos Biboudis, Manohar Jonnalagedda, Sandro Stucki, and Vlad Ureche, editors, *Proceedings of the 7th ACM SIGPLAN Symposium on Scala, SCALA@SPLASH 2016, Amsterdam, Netherlands, October 30 - November 4, 2016*, pages 31–40. ACM, 2016. doi:10.1145/2998392.2998398.

57 Peter H. Schmitt and Mattias Ulbrich. Axiomatization of typed first-order logic. In Nikolaj S. Bjørner and Frank S. de Boer, editors, *FM 2015: Formal Methods - 20th International Symposium, Oslo, Norway, June 24-26, 2015, Proceedings*, volume 9109 of *Lecture Notes in Computer Science*, pages 470–486. Springer, 2015. doi:10.1007/978-3-319-19249-9_29.

58 Thomas Studer. Constructive foundations for featherweight java. In Reinhard Kahle, Peter Schroeder-Heister, and Robert F. Stärk, editors, *Proof Theory in Computer Science, International Seminar, PTCS 2001, Dagstuhl Castle, Germany, October 7-12, 2001, Proceedings*, volume 2183 of *Lecture Notes in Computer Science*, pages 202–238. Springer, 2001. doi:10.1007/3-540-45504-3_13.

59 Ke Sun, Sheng Chen, Meng Wang, and Dan Hao. What types are needed for typing dynamic objects? A python-based empirical study. In Chung-Kil Hur, editor, *Programming Languages and Systems - 21st Asian Symposium, APLAS 2023, Taipei, Taiwan, November 26-29, 2023, Proceedings*, volume 14405 of *Lecture Notes in Computer Science*, pages 24–45. Springer, 2023. doi:10.1007/978-981-99-8311-7_2.

60 Ke Sun, Yifan Zhao, Dan Hao, and Lu Zhang. Static type recommendation for python. In *37th IEEE/ACM International Conference on Automated Software Engineering, ASE 2022, Rochester, MI, USA, October 10-14, 2022*, pages 98:1–98:13. ACM, 2022. doi:10.1145/3551349.3561150.

61 Nikhil Swamy, Catalin Hritcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean Karim Zinzindohoue, and Santiago Zanella Béguelin. Dependent types and multi-monadic effects in F. In Rastislav Bodík and Rupak Majumdar, editors, *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 256–270. ACM, 2016. doi:10.1145/2837614.2837655.

62 William W. Tait. Intensional interpretations of functionals of finite type I. *J. Symb. Log.*, 32(2):198–212, 1967. doi:10.2307/2271658.

63 Emina Torlak and Rastislav Bodík. Growing solver-aided languages with rosette. In Antony L. Hosking, Patrick Th. Eugster, and Robert Hirschfeld, editors, *ACM Symposium on New Ideas in Programming and Reflections on Software, Onward! 2013, part of SPLASH '13, Indianapolis, IN, USA, October 26-31, 2013*, pages 135–152. ACM, 2013. doi:10.1145/2509578.2509586.

64 Steffen van Bakel and Maribel Fernández. Normalization, approximation, and semantics for combinator systems. *Theor. Comput. Sci.*, 290(1):975–1019, 2003. doi:10.1016/S0304-3975(02)00548-0.

65 Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon L. Peyton Jones. Refinement types for haskell. In Johan Jeuring and Manuel M. T. Chakravarty, editors, *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming, Gothenburg, Sweden, September 1-3, 2014*, pages 269–282. ACM, 2014. doi:10.1145/2628136.2628161.

66 Niki Vazou, Anish Tondwalkar, Vikraman Choudhury, Ryan G. Scott, Ryan R. Newton, Philip Wadler, and Ranjit Jhala. Refinement reflection: complete verification with SMT. *Proc. ACM Program. Lang.*, 2(POPL):53:1–53:31, 2018. doi:10.1145/3158141.

67 Panagiotis Vekris, Benjamin Cosman, and Ranjit Jhala. Refinement types for typescript. In Chandra Krintz and Emery D. Berger, editors, *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, pages 310–325. ACM, 2016. doi:10.1145/2908080.2908110.

68 Peng Wang, Di Wang, and Adam Chlipala. Timl: a functional language for practical complexity analysis with invariants. *Proc. ACM Program. Lang.*, 1(OOPSLA):79:1–79:26, 2017. doi:10.1145/3133903.

69 Stefan Wehr, Ralf Lämmel, and Peter Thiemann. Javagi : Generalized interfaces for java. In Erik Ernst, editor, *ECOOP 2007 - Object-Oriented Programming, 21st European Conference, Berlin, Germany, July 30 - August 3, 2007, Proceedings*, volume 4609 of *Lecture Notes in Computer Science*, pages 347–372. Springer, 2007. doi:10.1007/978-3-540-73589-2_17.

70 Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Inf. Comput.*, 115(1):38–94, 1994. URL: https://doi.org/10.1006/inco.1994.1093, doi:10.1006/INCO.1994.1093.