UNIVERSIDADE DE LISBOA

FACULDADE DE CIÊNCIAS

DEPARTAMENTO DE INFORMÁTICA

LiquidJava: Extending Java with Refinements

Catarina Ventura Gamboa

Mestrado em Engenharia Informática
Especialização em Engenharia de Software

Dissertação orientada por:
Prof. Doutor Alcides Miguel Cachulo Aguiar Fonseca
Prof. Doutor Christopher Steven Timperley

2021

Agradecimentos

O final desta dissertação representa o culminar de uma experiência repleta de emoções, com altos e baixos, momentos inspiradores e de resiliência, que me ajudaram a crescer como pessoa e investigadora. Começo por agradecer ao professor Alcides por aceitar ser meu orientador e me incentivar a escrever artigos, fazer apresentações e participar em conferências, dando ao mesmo tempo espaço para que o desenvolvimento da tese fosse fluído e dependesse de mim em primeiro lugar. Gostaria também de agradecer ao Chris por aceitar ser meu coorientador e por todo o seu apoio durante a tese, pelas ideias e opiniões, e pelas reuniões que representavam sempre um ponto alto da semana (Thank you Chris!).

Durante este ano de pandemia, as reuniões semanais foram sem dúvida uma parte fundamental da minha saúde mental. Por isso agradeço a todos os *Cool Kids* que participaram nestas reuniões e me deram feedback para melhorar o trabalho. Agradeço principalmente ao Paulo pelo seu incentivo sempre positivo e ao Guilherme pelas perguntas incisivas que me ajudaram a crescer e estar preparada para o pior.

Neste período de final de escrita de tese, que coincidiu com a abertura da faculdade, quero agradecer a todos os membros do LASIGE que me acolheram de braços abertos e me fizeram sentir em casa todos os dias. A todos os que me têm ajudado a focar e a ser produtiva nos intervalos entre os almoços na cantina velha, os lanches de pão quente no c8 e jantares em direito, o meu muito obrigada por tudo. Dado que este caminho pela área de informática começou na licenciatura, chegando a este ponto não posso deixar de agradecer àqueles que ficaram comigo durante manhãs, tardes e noites a terminar projetos e mil e uma entregas. Agradeço à Rita que me aturou desde o seu primeiro dia na FCUL e me ajudou a manter sempre uma dose saudável de realidade no dia-a-dia, ao Vasco por tolerar as minhas manias e fazer um esforço para não se passar de cada vez que eu fazia esquemas fora das linhas, e ao Bernardo pelos seus comentários sarcásticos sempre de mão dada com as melhores promoções.

No final, como não podia deixar de ser, agradeço à minha família por todo o apoio e carinho que me deram nos momentos fáceis e difíceis, sem vocês nunca teria chegado tão longe. Aos meus pais por sempre procurarem dar-me a melhor educação possível sem fazerem qualquer pressão para que eu tivesse as melhores notas ou me comparasse com os outros. À minha irmã, Madalena, obrigada por me aturares sempre, nunca teres medo

de dar a tua opinião e me ajudares a ser uma melhor pessoa. Aos meus avós, Nelita, Augusto, São e Luís, obrigada por todos os ensinamentos e pelos almoços e jantares que sabem sempre a restaurantes de cinco estrelas. Agradeço ainda aos meus tios, Marta e Litó por me mostrarem que cada um deve seguir o caminho que sente ser o seu sem se preocupar com o que o resto do mundo acha, e aos meus primos, Mariana, Martim e Joana, que espero que por esta altura já saibam o que é um mestrado.

Finalmente, o final do mestrado representa apenas o início da minha experiência de investigação pelo que deixo um agradecimento a todos os que me incentivaram a continuar nesta área e a procurar novas formas de tornar o mundo melhor.

A todos os que consideram que "porque sim"não é resposta

Resumo

A área de desenvolvimento de software tem apresentado um grande crescimento nos últimos anos, havendo mais produtos a serem criados a cada dia. Com este crescimento, uma das maiores preocupações das organizações de desenvolvimento software é a garantia de qualidade do software produzido [46, 30]. As organizações estão sobre constante pressão para entregar produtos de excelente qualidade em prazos reduzidos e a baixos custos. Se o software for entregue ao cliente com erros e não realizar o que é suposto, a reputação da empresa pode ser posta em causa e a confiança dos clientes nos produtos produzidos diminuirá, levando-os a procurar outras empresas concorrentes. Assim, dados os elevados custos de encontrar erros e vulnerabilidades mais tarde no processo de desenvolvimento, os programadores e as organizações tentam detetar e corrigir os problemas o mais cedo possível quando são mais baratos e fáceis de tratar [66]. Para que estes erros sejam detetados cedo, é necessário aplicar métodos de verificação de software que permitam localizar erros e que deem aos utilizadores uma maior confiança na qualidade do software criado.

Um dos métodos de verificação mais usados na atualidade advém dos sistemas de tipos implementados em várias linguagens de programação modernas. Com os sistemas de tipos integrados na linguagem, apenas valores que pertencem ao tipo esperado são aceites no programa, sendo que esta verificação é feita em tempo de compilação do programa. Uma vez que os sistemas de tipos estão integrados na linguagem, até podem passar despercebidos como forma de verificação, uma vez que, para um programador, são parte natural da implementação.

Contudo, estes sistemas de tipos por vezes não são suficientes para garantir o correto comportamento do programa. Com esta motivação, os tipos refinados foram propostos como um passo incremental nos sistemas de tipos tradicionais, permitindo que os tipos básicos de uma linguagem sejam mais específicos com o uso de predicados lógicos. Assim, os tipos refinados permitem restringir os valores aceites nos diferentes tipos básicos mantendo a verificação integrada na linguagem de programação. Um tipo refinado pode ser representado por $\{x : B|p\}$, onde x é uma variável com o tipo básico B refinado pelo predicado p. Um exemplo de um tipo refinado que identifica os números inteiros positivos pode ser representado por $\{x : int|x > 0\}$.

O mecanismo de tipos refinados, apesar de parecer útil, ainda não se tornou popular na comunidade de desenvolvimento de software. Podem existem várias razões para tal,

sendo que uma das possibilidades é a baixa popularidade, na industria, das primeiras linguagens que tiveram implementações de tipos refinados (ex.: ML [24], Haskell [61]). Outra possível explicação vem do significativo aumento do esforço para o programador para perceber e escrever as especificações nos refinamentos.

Este trabalho propõe a utilização de tipos refinados integrados em Java, uma das linguagens de programação mais populares no mundo, com foco na usabilidade e com o objetivo de promover o uso global de tipos refinados como técnica de verificação de software.

Para integrar os refinamentos em Java, propomos o uso de anotações (como `@Refinement` ↪) onde se insere a linguagem dos refinamentos como uma *string*. A linguagem de refinamentos inclui recursos para modelar a especificação de variáveis, métodos, atributos de classe e estados de classes.

Para que os utilizadores não precisem de aprender uma nova linguagem para os refinamentos, esta deve ser o mais parecida possível de Java. Assim, criámos um inquérito para averiguar a sintaxe preferida de programadores de Java para a sintaxe dos refinamentos. Este inquérito continha duas ou três opções de sintaxes para adição de refinamentos em variáveis, métodos e atributos de classe, opções estas que os participantes avaliaram em termos de preferência. No total o inquérito obteve 50 respostas de programadores familiares com Java, e os resultados foram utilizados na construção da gramática para a linguagem dos refinamentos.

Os refinamentos para modelar classes são introduzidos com uma nova especificação que permite modelar o estado dos objetos da classe. Para modelar classes, é possível criar múltiplos conjuntos de estados e propriedades que podem ser usados na especificação dos métodos da classe dentro do refinamento `@StateRefinement(from="predicate", to="` ↪ `predicate")`. Usando esta anotação, os métodos da classe especificam qual o estado em que os objetos devem estar para invocar o método (expressado no argumento `from` ↪) e qual o estado dos objetos quando o método terminar (expressado no argumento `to`). Desta forma, a especificação é capaz de impor protocolos em classes e modelar com sucesso máquinas de estado em classes Java como, por exemplo, na `java.net.Socket`.

A verificação dos refinamentos em Java é feita através de regras de verificação de tipos, que criámos para esta extensão, e da tradução das relações de subtipos para condições de verificação (VCs). Estas condições são enviadas para um SMT Solver que as verifica automaticamente e indica se todos os refinamentos são respeitados ou se algum deles não é possível de provar. Neste último caso, uma mensagem de erro relacionada com o tipo refinado é mostrada ao utilizador.

As regras de verificação foram implementadas num protótipo designado por LiquidJava, que representa a extensão de Java para tipos líquidos (o subconjunto decidível dos tipos refinados). Este protótipo foi também integrado numa extensão para o editor de código *Visual Studio Code*, de modo a melhorar a usabilidade do sistema LiquidJava. Com

a utilização desta extensão, os utilizadores obtêm os erros e relatórios dos mesmos em tempo real enquanto desenvolvem os programas, com as linhas erradas são sublinhadas a vermelho e acompanhadas de mensagens de erro.

Para avaliar a extensão LiquidJava, desenvolvemos um estudo com utilizadores focado na usabilidade da extensão como uma ferramenta de verificação de software. Assim, conduzimos um estudo com 30 participantes familiares com Java e pedimos-lhes que realizassem tarefas relacionadas com a interpretação de refinamentos, o uso de LiquidJava para detetar e tratar erros, e a adição de refinamentos em código Java. O estudo mostrou que os refinamentos em variáveis e métodos são muito intuitivos dado que 86% dos participantes conseguiu utilizar os refinamentos corretamente sem nunca ter uma introdução ao tópico. Embora os refinamentos em classes se tenham mostrado mais difíceis de entender sem uma introdução inicial (apenas 46% dos participantes os usou corretamente), após um vídeo de 4 minutos e acesso a um website com exemplos, 100% dos participantes conseguiu anotar o protocolo com refinamentos na classe corretamente. Quanto à deteção e correção de erros usando Java e LiquidJava, o exercício com melhores resultados usando LiquidJava aumentou a localização do erro por 93% e a correção por 47% (dado que os restantes 53% se aproximaram da resolução, mas não obtiveram uma versão completamente correta) quando comparado com as repostas em Java. Este exercício usava o protocolo da classe `java.net.Socket`, o que mostra que o LiquidJava pode ser mais útil quando aplicado a classes e protocolos menos conhecidos, reduzindo o tempo passado na localização dos erros. Finalmente, o estudo mostrou que os participantes acharam fácil a adição de refinamentos em programas Java e todos declararam estar abertos a usar LiquidJava nos seus projetos.

No futuro, é esperado que o LiquidJava evolua para ter uma verificação completa da linguagem Java, com uma melhor usabilidade dentro de editores de código e mensagens de erro mais explícitas. Assim, prevê-se que o LiquidJava possa ser utilizado em produtos de software crítico e em projetos de âmbito geral de modo a melhorar a qualidade do código produzido.

Palavras-chave: Tipos Refinados, Verificação de Software, Linguagens de Programação, Java

Abstract

Software development is an area with continued growth over the years, with more applications being produced every day. As software demand grows, so does the demand for software reliability. Bugs and vulnerabilities that are found earlier during the development lifecycle are easier and cheaper to fix, whereas bugs found in production are difficult and expensive to address and may have dire consequences. Thus, it is important to use software verification techniques to improve software quality.

Type systems are one of the most popular verification techniques since they help prevent bugs early in the development lifecycle. Refinement types are more powerful than traditional type systems since they extend a type system with predicates over the existing types, detecting more classes of bugs. However, despite the perceived utility of refinement types, they have not yet been adopted by mainstream developers.

This work aims to promote wide usage of refinement types by adding them to Java, one of the most popular programming languages in the world, with a focus on usability. Thus, we defined usability requirements and followed them in the design of the additional type system with refinements. To promote accessibility, we conducted a series of developer surveys to design the syntax of refinements for variables, methods and classes. We propose an approach of using refinements in classes to model type state using state sets and ghost properties. Finally, we created an implementation of LiquidJava and integrated it into an IDE so developers can use the verification information while they are developing the code. To evaluate the prototype's usability, we conducted a research study with 30 Java developers, concluding that users intend to use LiquidJava and that it helped find more bugs and debug faster.

Keywords: Refinement Types, Software Verification, Programming Languages, Java

Contents

List of Figures xviii

Acronyms xx

1 Introduction 1

1.1 Motivation . . . 1

1.2 Goals . . . 2

1.3 Document Structure . . . 3

2 Background and Related Work 5

2.1 Domain-specific Verification . . . 5

2.2 Java Verification . . . 6

2.3 Refinement Types for Verification . . . 7

2.4 Type State . . . 8

3 Language Design Process 11

3.1 Requirements . . . 11

3.2 Design Solution . . . 12

3.2.1 Refinements Design . . . 12

3.2.2 System Design . . . 13

4 RJ – The Language of Refinements 15

4.1 Refinable Targets . . . 15

4.2 RJ Grammar . . . 16

4.3 Survey on Refinements Syntax . . . 18

4.3.1 Background of Participants . . . 18

4.3.2 Anonymous Variable . . . 20

4.3.3 Variables . . . 21

4.3.4 Method declarations . . . 22

4.3.5 Predicate Aliases . . . 24

4.3.6 Ghost/Uninterpreted Functions . . . 27

4.4 Class Refinements . . . 29

4.4.1 Ghost 29
4.4.2 StateSet 30

5 LiquidJava Type System **33**
5.1 Approach 33
5.2 Verification Conditions 35
5.3 Notation 35
5.4 Refinements Verification 36
5.4.1 Variables 37
5.4.2 Class Instance Fields 39
5.4.3 Branching Conditions 40
5.4.4 Method Invocations 40
5.4.5 External Libraries 42
5.4.6 Method Definitions 43
5.4.7 Discussion 49

6 An Implementation of LiquidJava **51**
6.1 LiquidJava System 51
6.1.1 Architecture 51
6.1.2 SMT-based Verification 52
6.1.3 Error Messages 52
6.2 Integration with IDE 55

7 Evaluation **59**
7.1 Study Configuration 59
7.2 Participants 61
7.3 Interpreting Refinements without prior explanation 61
7.4 Using LiquidJava to Detect Bugs 62
7.4.1 Exercises and Answers 63
7.4.2 Discussion 66
7.5 Adding LiquidJava Annotations 66
7.6 Final Overview 68
7.7 Study Conclusions 69

8 Future Work **73**

9 Conclusion **77**

A Liquid Type Checking Rules **79**

B Figures and Listings from Evaluation **81**

Bibliography **99**

List of Figures

3.1 Design of LiquidJava Verifier. 14

4.1 Grammar for the Refinements Language. 17
4.2 Background information on the participants of the syntax survey. 19
4.3 Connection between the participants background information on refinement types and functional languages and JML. 20
4.4 Syntax options to represent anonymous variables. 21
4.5 Participants answers on their syntax preference for the anonymous variable. 22
4.6 Syntax options for the refinements in variables. 23
4.7 Answers on the syntax for refinements in variables. 23
4.8 Syntax options for the refinement of Methods. 24
4.9 Answers on the syntax for the refinements in Methods. 25
4.10 Syntax options for the declaration and usage of Aliases. 26
4.11 Preference answers on the alias syntax. 26
4.12 Syntax options for the placement of the declaration of ghost functions. . . 28
4.13 Preference answers to the placement of ghost functions. 29
4.14 Finite State Machine that represents the protocol of the FileReader class. . 31

5.1 Visual representation of the subtyping relationships verified for a simple program. 34
5.2 Notation. 36
5.3 Split of the storage of variable's refinements into local and global context. 38
5.4 Code and verification conditions on usage of variables assignments in different code places. 39
5.5 Verification of `inRange` method return and invocation. 41
5.6 Verification of the body of the `fib` method. 42
5.7 ArrayDeque verification with ghost property size. 45
5.8 Finite State Machine that represents the protocol. 47
5.9 Socket class annotation and client verification. 47

6.1 Architecture of the LiquidJava System. 51
6.2 Client code of ArrayDeque with error and respectively error message. . . 54
6.3 IDE Plugin reporting an error. 55

B.1 Background information of the 30 participants selected to participate in the study. 82
B.2 Answers on interpreting LiquidJava refinements. 82
B.3 Answers correctness and duration of the Fibonacci exercise, in both versions. 83
B.4 Answers and time spent on the Sum exercise, in both versions. 84
B.5 ArrayDeque Exercise. 87
B.6 Answers and time spent on the ArrayDeque exercise, in both versions. . . 88
B.7 Answers and time spent on the Socket exercise, in both versions. 88
B.8 Results of the annotations with LiquidJava. 88
B.9 Protocol of the TrafficLight that must be followed to the annotation. . . . 89
B.10 Participants' answers to the ease of writing the specification with LiquidJava. 89
B.11 Main subjects that the participants enjoyed to use during the study. 89
B.12 Main subjects that the participants disliked during the study. 90
B.13 All the participants stated that they would use LiquidJava in their projects. 90

Chapter 1

Introduction

This thesis proposes the integration of refinement types in the Java language. To this end, the current chapter introduces the motivation and relevance of this work (Section 1.1), defines its goals (Section 1.2) and, finally, describes the structure of the remainder of the document (Section 1.3).

1.1 Motivation

Nowadays, one of the major concerns in software development is guaranteeing software quality [46, 30]. Organizations are under constant pressure to deliver products with excellent quality in tight schedules and reduced costs. If the software is delivered to the client with incorrect behaviours, the reputation of the company can be at stake, and the confidence of buyers in the products will decrease. Moreover, given the increased costs of finding errors and vulnerabilities later in the development lifecycle, developers and organizations aim to detect and treat the issues as soon as possible when they are easier and cheaper to address[66]. Thereby, it is necessary to create methods of software verification that allow developers and organizations to have greater confidence in the quality of the created software.

One of the most popular methods for establishing guarantees for the correct behaviour of a program is the usage of type systems [29]. Type systems implemented in multiple modern languages, such as Java or Haskell, allow developers to specify the expected type of different operations, verifying at compile-time if these types are respected. Therefore, unwanted errors can be eliminated before execution time. However, a program with all the correct types can still have multiple errors; some examples include out-of-bounds accesses or division by zero errors.

Refinement types have been proposed as a static verification mechanism that is embedded within the programming language and prevents some classes of errors that cannot be caught using a regular type system. The main idea of refinement types is to extend a language with predicates over the basic types, restricting the allowed values in variables

or methods.

```
@Refinement("y > 0 && y < 50")
int y;
y = 10; // okay
y = 100; // okay in plain Java, but it is a refinement type error
```

Listing 1.1: Simple refinement usage.

Listing 1.1 presents an elementary error detected when a variable is assigned a value outside of its range.

Despite the perceived utility, refinement types have not yet been adopted by mainstream developers. One of the possible explanations is that the earliest languages with refinement type support (ML [24], Haskell [61]) are not widely popular in the industry. Other possible reason is that there is a non-insignificant overhead in thinking about and writing the refinement specification.

Java is one of the most popular programming languages in the world, achieving podium places in multiple rankings throughout the years. Some examples of these rankings include the TIOBE Index [11] that counts the hits of 25 search engines for the language, and PYLP [51] that determines the popularity of a programming language by the number of tutorial searches on Google.

1.2 Goals

The main goal of this work is to promote the wide usage of refinement types by integrating these types within Java. This goal can be achieved by creating an additional type system with refinements on top of the existing Java type system, extending the language with predicates over the basic types.

The expected contributions of this study include:

- a language designed with the input of developers (presented on Chapter 4);
- a formal definition for the additional type system (Chapter 5);
- an implementation of the type checking algorithm, named LiquidJava, and the integration of the prototype as an editor plugin (Chapter 6);
- an evaluation of LiquidJava using a reserach study performed by 30 developers (Chapter 7).

Despite the importance of choosing the right language to promote the wide usage of refinement types, other efforts must be made to ensure the success of this work. With this concern, we defined the following requirements for LiquidJava:

- Liquid type checking should work on top of an existing Java type checker.
- Refinements must be optional – A valid Java program with or without refinements should type check.
- Refinements need to be expressive – The language of the refinements should be as close to Java as possible, so developers do not have to learn a different language for the specification.
- Liquid type checking should be decidable – without introducing unnecessary overhead to the compilation process.

Each of the presented requirements will be detailed and analysed in section 3.1. The work on this thesis focuses on the conception, implementation and evaluation of LiquidJava as a software verification technique that can be used to increase the confidence of Java software quality.

1.3 Document Structure

The remainder of this document is organized into eight chapters. Firstly, we introduce the background and related work (Chapter 2), identifying studies and tools that focus on similar approaches of software verification. Afterwards, the design process of LiquidJava is described (Chapter 3), detailing the decisions that lead to the creation of the refinements language and the LiquidJava type system. Both these topics are then detailed with the presentation of the refinements syntax and language constructs(Chapter 4), and the type system rules and verification examples (Chapter 5). An implementation of the presented concepts is thereupon revealed (Chapter 6) with the details of the prototype and its integration into a development editor (IDE). Finally, an evaluation of LiquidJava is performed (Chapter 7) with a research study directed to Java developers. To conclude, the last chapters of the document present the future directions of current work (Chapter 8) and the conclusions of this dissertation (Chapter 9).

Chapter 2

Background and Related Work

This chapter presents methodologies and tools for software verification. It starts with traditional verification approaches in specific and critical domains (Section 2.1), and the applications of these methodologies into verification tools for Java programs (Section 2.2). After the overview of verification methods, the focus changes to the multiple flavours and implementations of refinement types in the scope of program verification (Section 2.3). Finally, the chapter reviews work related to modelling object behaviour with type state, presenting frameworks that apply these concepts into the Java language (Section 2.4).

2.1 Domain-specific Verification

Several methodologies for software verification have been introduced to specific domains. Critical software, where the consequences of software errors can easily lead to catastrophic results, is a popular context for applying software verification. Some examples of these programs run on medical devices, banking networks and space missions.

Model Checking [18] is one of the most popular software verification techniques used in critical software. This automated technique is able to verify finite-state systems, including sequential, concurrent and distributed systems. There are several implementations of this technique, with Spin [27] being one of the most popular tools, but all suffer from the state explosion problem [14], which limits the complexity of systems that can be tested.

Another popular technique is Design-by-Contract (DbC). This methodology was proposed by B. Meyer within the Eiffel programming language [42] and relies on the fulfilling of a contract between the clients of a class and the class itself. This contract is specified in methods, as pre- and post-conditions, where the former must be respected by the clients when invoking the class methods, and the implementation of the method must guarantee the post-conditions. If either side does not respect the contracts, an error exists. Dafny [37] is an example of a language and verifier that was created with the purpose of designing verified software by means of DbC.

Dependent and refinement types are also used as verification techniques that introduce

domain-specific information to a program via the type system. Fully dependently typed languages allow the user to define custom types that depend on values. A typical example is a type of list with a given length, expressed as `Vec A n`, where A is the type of elements and n is the length of the list that can have an arbitrary value. Examples of fully dependently typed languages include Agda [45], a functional programming language, Coq [12], a proof-assistant, and Idris [10], a general-purpose language designed to encourage and boost type-driven development. While in these languages developers can create types with values attached, in languages with refinement types, basic types can be constrained with a logic formula, creating a subtype of the already existing basic type. Languages extended with refinements types are introduced latter in this chapter (Section 2.3).

Certified Compilation is also a small domain with an interest for verification. To execute any program written in a high-level language, the source code must be translated to a low-level language through a compiler. If the compiler itself has errors, the translation process can add incorrect behaviours to the executed code. To avoid this problem, compilers can also be formally verified, leading to a certified compilation.

CompCert [38] is one of the first verified compilers that target a complete compilation chain, starting with Clight [9] (subset of C programming language), going through eight intermediate languages and finishing in PowerPC assembly code. The implementation and proof of the compiler was done using the Coq[12] proof assistant. Other example is the compiler for CakeML [58], a functional language based on a subset of StandardML. This verified compiler was developed inside the HOL4[3] theorem prover and is used within the pipeline of constructing verified applications in CakeML.

2.2 Java Verification

As a popular language, Java has been a target for the most popular methodologies of program verification.

JavaPathFinder (JPF) [40] is a model checking tool created to analyse Java bytecode that was developed by NASA Ames Research Center for mission critical programs. JPF is designed to be used in distributed systems where it is able to check runtime errors, assertion violations and deadlocks.

Within the methodology of Design-by-Contract, Java has several implementations. According to a study by Aghaei, in 2018 [2], two of the best approaches in terms of functionality are Bean Validation and OpenJML.

Bean Validation, renamed to Jakarta Bean Validation, is a Java API for JavaBean validation in JavaEE and JavaSE [25], and consequently applied essentially in enterprise applications. Bean Validation uses annotations to express constraints on methods, fields, parameters, among others. The API makes available several annotations, such as `@Email` for strings, `@FutureOrPresent` for dates or `@NotNull` for objects, and besides the built-in

annotations, it is possible for the developer to create custom annotations more suited for the working domain.

OpenJML [15] is an implementation of Java Modeling Language (JML) [36], a notation for writing specifications within the Java language. JML is a very expressive language and can encode specifications of methods pre- and post- conditions, as well as class invariants. The three major drawbacks of OpenJML are incompleteness, an undecidable logic when the specification includes quantified assertions, and the fault messages/warnings that can be false positives [15].

According to the collaborative Awesome Java ranking on formal verification [39], the most popular tool for Java verification is the Checker Framework[47]. This tool uses a pluggable types approach, where type checkers can be created and plugged into the code as specific annotations in a Java program. The framework allows the creation of custom type checkers and includes multiple ones by defaults, such as the `Nullness Checker` that detects misusages of variables annotated with `@NonNull`.

2.3 Refinement Types for Verification

As introduced in Section 1.1, refinement types enrich a language type system with predicates, leading to more expressive types than the ones in Java and similar languages [29]. Since refinements are integrated into the programming language, the barrier between implementation and proof is thinner, which makes the program verification easier for developers. Moreover, the type checking of the refinements performs the automatical verification of the program, proving if all the refinements are respected or if any of them is violated.

Freeman and Pfenning [24], in 1991, introduced the concept of refinement types inside ML, a strongly-typed functional programming language, allowing the detection of more errors in compile time. Liquid Types [53] (*Logically Qualified Data Types*), proposed in 2008, represent a subset of refinement types that only use predicates over a decidable logic to ensure the inference and type checking decidability. LiquidHaskell [60] presents an implementation of liquid types in Haskell where each type is decorated with a predicate from a decidable refinement logic. In this work, Vazou et al. introduce type aliases to improve the brevity and readability of the predicates, the usage of refinements in function contracts capturing pre- and post-conditions, and the introduction of measure functions to specify properties of data types.

Since the initial proposal of refinement types, there has been an effort to bring their usefulness to more mainstream, imperative programming languages. CSolve [52] uses refinement type checking to verify heap layout and pointer usage within C programs using macros to express the liquid refinements. Kazerounian [32] introduced refinements for Ruby, an object-oriented and dynamic scripting language. The proposed system, RTR,

adds the refinements on top of a ruby type system and verifies the properties through a translation to non-object-oriented Rosette, a solver-aided host language that interacts with Z3 for logical verification. DJS [13] is a typed extension of Javascript that targets features of dynamic languages, as extensible objects and prototype inheritance, as well as imperative features such as flow-sensitive strong updates on mutable variables.

In 2016, Schmid et. al [55] and Vekris et .al [62] proposed the addition of refinements in Scala and Typescript, respectively, with constructs to model and verify object-oriented aspects of the language. In both studies, refinements are introduced as class invariants on immutable class fields (that cannot be modified outside the class constructor), which allows the usage of the immutable fields on the refinements of the mutable ones.

In the Java language, Stein et al. [56] applied a specific set of refinement types to stream-based processing but used a fixed type hierarchy, limiting the expressiveness of the refinements since they do not use logical predicates to qualify the types.

2.4 Type State

In object-oriented software systems, objects usually store data that can be accessed and modified by methods. Often, interfaces and classes define or implement methods that are implicitly expected to be used in a specific sequence, defining a protocol for the class. However, mainstream object-oriented languages offer only informal documentation to capture these protocols, not verifying if the protocol is followed, leading to unexpected runtime errors. This problem has been previously explored with different optics, and this section presents alternatives to make protocols explicit and statically verifiable.

Strom et al. [57] introduced the concept of typestate in object-oriented languages as a refinement of the concept of type that determines the subset of operations allowed in a specific object context. With typestates, it is possible to define the legal and illegal operations of each state, leading to the identification of valid and invalid execution sequences. An object typestate can be described using a Finite State Machine graph with states and state transitions.

Following the concept of typestate, the paradigm of typestate-oriented programming [4] was presented as an extension of object-oriented programming. In this paradigm, introduced with the Plaid programming language, objects are modelled according to their internal states, and methods may imply a transition to a new object state. Therefore, each method specifies if its execution produces a change in the object state, making it explicit in the specification. The definition of states and their transitions allows for the accurate description of protocols and can prevent errors in the invocation pipeline.

In Java, the typestate concept was introduced by Bierhoff and Aldrich [7], following the work on Fugue [17], and relates typestate with subtyping and inheritance. In this approach, methods are annotated with pre- and post-conditions for the object state and all the

methods parameters. Each method describes the possible source and destination states, allowing the definition of multiple cases if different sources result in different states. Subclasses can model their states using the states defined in the superclass and split them into new ones, allowing for a more precise specification. One of the drawbacks of this approach is the impossibility of using dependent types in the specification (e.g. the specification of the return value refer to a parameter). Another one is that the type checking occurs dynamically, forcing the program's execution to verify the state properties.

Mungo [34] is also a tool that allows for the definition of object protocols in Java, using the notion of type state and statically verifying if the defined protocol is followed. In Mungo, the user creates a new *.protocol* file with the specification of the possible states and the allowed transitions, and associates the specification to a class with the `@Typestate` ↪ annotation with the protocol's name file. In this approach, the transitions may depend on the return value of the methods if they represent Java enumerates. In June 2021, Java Typestate Checker [44] was introduced as a new extended implementation of Mungo and a plugin for the Checker Framework. It extends Mungo with modelling function arguments and return values, verification of protocol completion, control over shared resources, and detection of null-pointer exceptions. However, both approaches define the protocol in external files instead of integrating the information in the Java class. Using the external file implies the replication of methods' signatures in both the class and the specification, making the changes harder to apply. It also implies the increasing number of files in the project, which may lead to difficult navigation in large projects, making it hard to match the class with the specification file.

The latest long-term support Java version, Java 17 (released in September 2021), finalised the introduction of sealed classes [1] that can be used to implement state machines. Sealed classes restrict the classes that can extend or implement a class. This feature can improve the implementation of state machines in Java since it is possible to limit the states available for a class and, for each state, create a specific class that implements the allowed transitions (as exemplified in a technical blog post[2]). However, this approach involves the creation of a new class for each state and the methods to be split into multiple classes, which can make the superclass information disperse. Moreover, the state transitions are defined by the interfaces the class implements, so each class method cannot have a specific transition or depend on the values of the method's parameters.

[1] https://openjdk.java.net/jeps/409
[2] https://benjiweber.co.uk/blog/2020/10/03/sealed-java-state-machines/

Chapter 3

Language Design Process

This chapter focuses on the design process for the integration of refinement types within the Java programming language. The design requirements are addressed in Section 3.1 and the design decisions that were incorporated in the solution are presented in Section 3.2.

3.1 Requirements

This section details the four requirements designed to promote the wide usage of LiquidJava, introduced previously in Section 1.2.

R.1 Liquid type checking should work on top of an existing Java type checker The purpose of the liquid type checker is to verify the restrictions introduced by refinement types to a Java program. Hence, the traditional type checking of Java programs should still be performed before the additional refinement verification. Thus, a Java program regardless of having refinements or not, should always be compiled by a regular Java compiler.

Regarding the liquid type checker, this requirement allows a better maintenance of the code, since it only checks the use of refinements and not any other Java features, thus there is no unnecessary repeated code between the Java type checker and the liquid type checker.

Another advantage is that the liquid type checking should keep working with new versions of Java. Implementing the liquid type checking on top of a Java type checker prevents an early deprecation of LiquidJava with new Java releases.

R.2 Refinements must be optional A Java program without refinements must be successfully validated by the liquid type checker, allowing specifications to be added after the implementation, including to pre-existing codebases. This grants the developer the possibility of incrementally build the specification of the program, start-

ing with small verifications (i.e. inside a function) and leading to a complete specification of the program.

R.3 **Refinements need to be expressive** Refinements must be written with a syntax that is as similar to Java as possible to enable developers to annotate the specifications without learning a completely new language.

R.4 **Liquid type checking should be decidable** To provide good usability, the type-checker must be fast and consistent in order to be used during the iterative development process. To this end, refinements are limited to Liquid Types [53], since they are decidable and typically quickly verifiable, using SMT Solvers. Thus, the predicates are restricted to a decidable logics using quantifier-free linear integer arithmetic with uninterpreted functions, and accepting only SMT-decidable operations.

3.2 Design Solution

The solution developed can be split into two main topics: the design of the refinements language and how it is incorporated into Java programs; and the design of the verification system to validate these programs. These topics are detailed in Section 3.2.1 and Section 3.2.2 and, together, they answer the requirements presented in the previous section; specifically, the first focuses on answering the requirements **R.2** to **R.4** and the latter answers **R.1**.

3.2.1 Refinements Design

In previous refined typed languages [29, 13], variable and method declarations have been the main target of annotations. However, in Java, classes are a core concept, and a desirable target for refinements. Furthermore, the language in which refinements are written needs to be flexible and understandable. Taking this into account, we made two major decisions regarding the refinements design:

- Refinements are encoded as Java Annotations in the source code, since annotations are optional, support all necessary targets and are commonly used in Java since their introduction in JDK 1.5 (Section 3.2.1.1);

- Refinements are expressed within strings, which are arguments to annotation, using a refinements language, RJ , that follows a syntax based on the feedback of Java developers (Section 3.2.1.2).

These decisions culminate in the example present in Listing 3.1 and are detailed in the remainder of this section.

```java
@Refinement("y > 0 && y < 50")
int y;
```

Listing 3.1: An example of a variable annotated with a refinement in LiquidJava.

3.2.1.1 Refinements as Annotations

Refinements are encoded as annotations on target Java features, so that adding the refinements (regardless of them being correct or not) does not prevent the program from being compiled with a regular Java compiler. This solution fulfills **R.2** since these annotations are always optional as they are not verified by the Java type checker.

Additionally, annotations have a target code element to which they can be applied[1], and these targets include local variables, parameters, methods, fields and classes which allows us to add refinements in all the desired code elements.

Using annotations to express restrictions on variables has become more popular over the years, with `@NonNull` and `@NonEmpty` being present in many Android and enterprise applications. Further, two of the verification tools presented in Section 2.2 use annotations to encode specifications in the source code. In both cases, the tool makes available multiple annotations for the developer to incorporate in the source code. However, the creation of custom annotations requires some expertise on the usage of the tool.

The popularity of annotations gives us some confidence that their usage within LiquidJava will not constitute a barrier to the adoption of the system. Thus, a new `@Refinement` annotation was created to express refinements.

Other annotations will be presented in Chapter 4 as syntax sugar for some features in the refinements language.

3.2.1.2 Refinements Language

Refinements are written within strings inside the annotations and use a personalized language that aims to be easy to understand by java developers. This language, named as RJ , is detailed in Chapter 4, and aims to be as similar to Java as possible, addressing **R.3**, and allowing a fast adaptation of Java developers to the writing of specifications. To improve the usability chances of refinements, we conducted an online survey to assess the best syntax for RJ while keeping the language verifiable by SMT Solvers, fulfilling **R.4**.

3.2.2 System Design

To verify a Java program annotated with refinements, a LiquidJava verifier proves that all refinements are respected throughout the program or shows that there is a violation of the specification. The verifier design is summarized in Figure 3.1.

[1] https://docs.oracle.com/javase/tutorial/java/annotations/basics.html

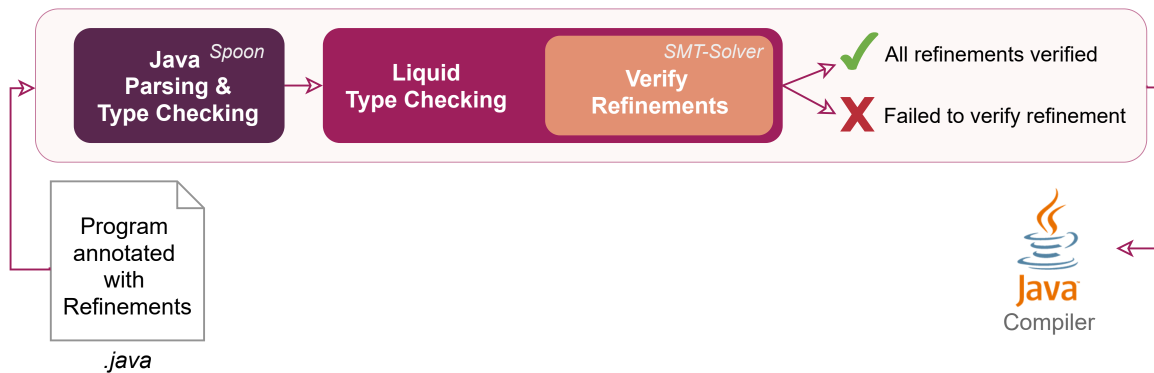


Figure 3.1: Design of LiquidJava Verifier.

The verifier system receives as input a Java program annotated with refinements and performs a static verification before the execution of the program.

The first step of the system involves parsing the input to an abstract syntax tree (AST) used for the verification. Since the parsing always happens before the liquid type checking, we can fulfil **R.1** by choosing a parser that incorporates Java type checking. We used the Spoon[49] framework, which uses the Eclipse Compiler Kit, adopting Java features as they are released.

Taking an AST representation of the program as input, the liquid type checking traverse the AST and all the expressions are checked against their expected type through subtyping relationships. These relationships are then discharged to an SMT Solver, which proves their satisfiability. The verification process will be explained in detail in Chapter 5.

Chapter 4

RJ – The Language of Refinements

To express the refinements in LiquidJava, we designed a new language, named RJ (Refinements in Java), that is used inside the refinement annotations. This chapter introduces this language, its features and the syntax used to describe each feature. To this end, we present the code elements to which we can add refinements (Section 4.1), the grammar for RJ language (Section 4.2), a survey on the syntax preferences of Java developers for LiquidJava features(Section 4.3), and, finally, the language features added to model classes (Section 4.4).

4.1 Refinable Targets

Refinements constrain the allowed values in certain code elements, modelling their behaviour. In LiquidJava, we can introduce refinements in the following code elements:

- **Variable declarations** - These refinements ensure that any variable assignment must respect the refined type of the variable.
- **Method definitions** - Refinements can be applied both to the parameters and the return value of any method. Everywhere a method is called, the arguments will have to fulfill the refinement of the parameters, and the return value will have the expected refinement of the method declaration.
- **Class fields definitions** - Fields can have refinements that work as invariants throughout the all class, including inside each class method.
- **Class definitions** - Refinements can be used in classes to model the state of objects. Classes are considered the fundamental programming elements of the Java language [33]. Although classes themselves do not have a specific value that can be refined, they can have methods that produce changes to the internal state of the objects. In this view, we can refine the object state when a method is called and when the method has ended using the annotation `@StateRefinement(from="predicate" ↪ , to = "predicate")`.

Previous works on refinement types focused on refinements in variables and methods. When refinements were introduced into object-oriented languages, class fields could also be modelled with refinements. However, the modelling of object type state as state refinements is novel in LiquidJava.

4.2 RJ Grammar

Since refinement types aim to extend the type system by adding predicates, the core syntax of the RJ relies on logical predicates. The predicates allowed in the language belong to the SMT-decidable logic and are drawn from the quantifier-free linear arithmetic and uninterpreted functions logic. Predicates can use boolean and integer literals and variables, linear arithmetics (e.g. addition, subtraction), boolean operators (e.g. greater than), logical operators (e.g. and, or), the ternary expression `if-then-else`, and invocation of uninterpreted functions. The refinements language also includes anonymous variables and the declaration of predicate alias and ghost functions, three language constructs that will be detailed in Section 4.3. The refinements language is mainly used inside the `@Refinement` ↪ annotation. However, we introduced new annotations as syntax sugar to facilitate the usage of additional language constructs, such as the declaration of predicate alias and ghost functions, and to model the class state. These annotations also use the refinements language to encode the declarations and the refinement predicates.

Figure 4.1 depicts the formal grammar of the refinements language. The decisions for the refinements syntax are detailed in Section 4.3 and Section 4.4. In the first section, we describe the implementation based on results of an online survey for Java developers regarding their preference on possible options for the refinements syntax on traditional refinements constructs. In the latter section, we focus on the syntax for the features that model classes.

$$
\begin{array}{lrcl}
\textit{start} & S & ::= & Pred \mid AliasDecl \mid GhostDecl; \\
\textit{predicate} & Pred & ::= & Exp \\
 & & \mid & Pred\ LOP\ Pred \\
 & & \mid & \mathbf{!}\ Pred \\
 & & \mid & Pred\ \mathbf{?}\ Pred\ \mathbf{:}\ Pred; \\
\textit{expression} & Exp & ::= & Exp\ BOP\ Exp \\
 & & \mid & Oper; \\
\textit{operand} & Oper & ::= & LitExp \\
 & & \mid & Oper\ AOP\ Oper \\
 & & \mid & UOP\ Oper; \\
\textit{literal expression} & LitExp & ::= & c \\
 & & \mid & x \\
 & & \mid & _ \\
 & & \mid & x\ \mathbf{(}\ Arg\ \mathbf{)}; \\
\textit{argument} & Arg & ::= & Pred \\
 & & \mid & Pred\mathbf{,}\ Arg; \\
 & & & \\
\textit{ghost decl.} & GhostDecl & ::= & \mathbf{ghost}\ GhostDecl' \\
 & & \mid & GhostDecl' \\
 & GhostDecl' & ::= & T\ x\mathbf{(}\ ArgDecl\ \mathbf{)} \\
 & & \mid & T\ x; \\
\textit{alias decl.} & AliasDecl & ::= & \mathbf{type}\ AliasDecl' \\
 & & \mid & AliasDecl'; \\
 & AliasDecl' & ::= & x\ \mathbf{(}\ ArgDecl\ \mathbf{)}\ \mathbf{\{}\ Pred\ \mathbf{\}}; \\
\textit{argument decl.} & ArgDecl & ::= & T\ x; \\
\textit{logical operator} & LOP & ::= & \&\& \mid || \,; \\
\textit{boolean operator} & BOP & ::= & > \mid >= \mid < \mid <= \mid == \mid != ; \\
\textit{unary operator} & UOP & ::= & ! \mid + \mid - ; \\
\textit{arithmetic operator} & AOP & ::= & + \mid - ; \\
 & & & \\
\textit{Types} & T & ::= & \mathbf{int} \mid \mathbf{boolean}; \\
\end{array}
$$

Figure 4.1: Grammar for the Refinements Language.

4.3 Survey on Refinements Syntax

To assess the best syntax for the language of refinements, we created and shared an online survey with possible syntaxes for LiquidJava features based on features of other implementations of refinement types. These features include type refinements of variables and methods and the use of predicate aliases and anonymous variables. For each of the language features, we proposed two or three different syntaxes that the participants evaluated with three preference levels: *Not Acceptable*, *Acceptable*, and *Preferable*.

The survey started by asking the background of participants and briefly explaining the concept of refinement types, with the annotation of a variable, before questioning the preferred syntax options for the LiquidJava features.

We sent the survey by email to current and former students and researchers of two universities (Faculdade de Ciências da Universidade de Lisboa and Carnegie Mellon University) as well as Java developers in enterprises, and also shared the survey link during the POPL2021[1] conference. As a result, we obtained a total of 50 valid answers from participants familiar with Java.

This section presents the background of participants (Section 7.2) and the study questions with the corresponding answers (Sections 4.3.2 and 4.3.6).

4.3.1 Background of Participants

At the beginning of the survey, all participants answered background-related questions pertaining to their familiarity with Java, functional programming languages, refinement types and JML. For each of the technologies, the participants evaluated their knowledge with one of the four possibilities: *Not Familiar*, *Vaguely Familiar*, *Familiar* and *Very Familiar*. The answers to these questions are presented in Figure 4.2.

Since the survey aims to assess the best syntax for the usage of refinements in Java, we were interested only in participants with some familiarity with the language; otherwise, the answers would have not been demonstrative of the population that uses Java and might use LiquidJava. To this end, we only accepted answers from participants at least *Vaguely Familiar* with Java. From the selected answers (shown in Figure 4.2a), 52% of the participants considered themselves *Very Familiar* with Java, 34% *Familiar* and the remaining 14% *Vaguely Familiar*.

We inquired the familiarity with functional programming languages to explore if it has a connection to their preferred syntaxes since some syntax proposals were inspired in previous implementations of refinement types developed in functional languages. Their background in this topic is spread through the four familiarity levels as shown in Figure 4.2b, with 62% of the participants choosing the middle options of *Familiar* and *Vaguely Familiar*.

[1] https://popl21.sigplan.org/

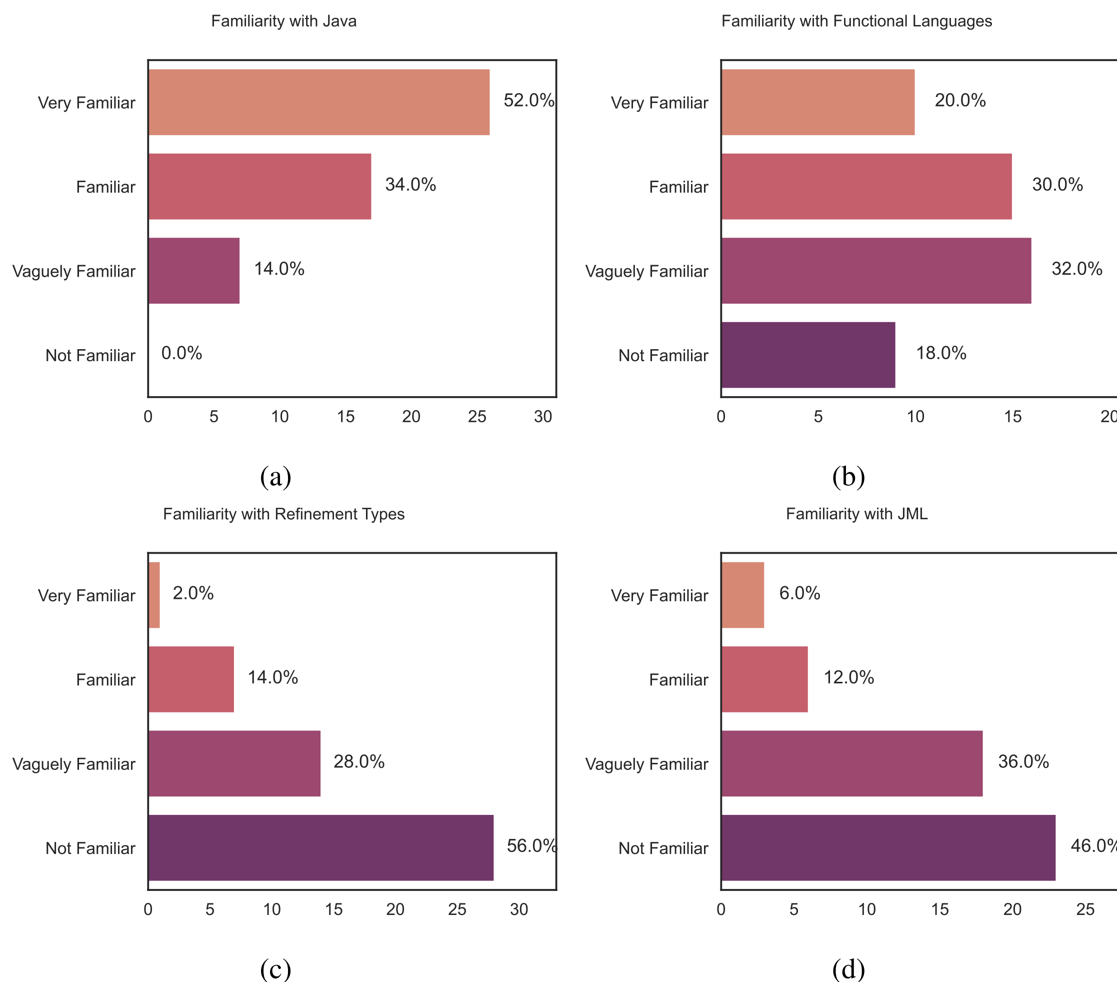
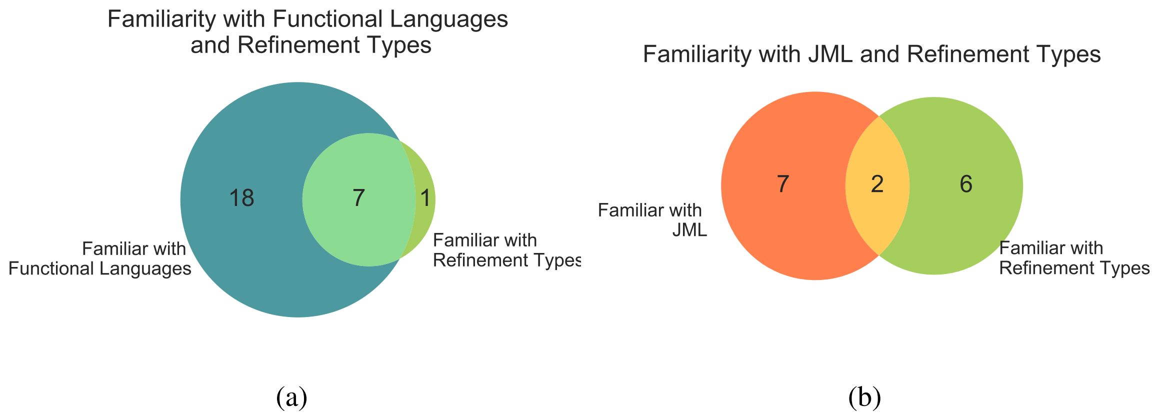


Figure 4.2: Background information on the participants of the syntax survey.

We also gathered their familiarity with refinement types (Figure 4.2c) and with JML (Figure 4.2d). Familiarity with JML was relevant since it is a popular specification language to model the Java language. The distribution over the four familiarity levels was similar for both verification techniques, since the higher percentage of participants considered themselves *Not Familiar* with the topics (56% regarding refinement types, and 46% regarding JML) and a very small amount considered themselves *Very Familiar* (only one participant on refinement types and three on JML). 42 out of the 50 participants are only *Vaguely Familiar* or *Not Familiar* at all with the topic of refinement types, and, 41 participants chose the same options for JML, as shown in Figure 4.2c and Figure 4.2d.

After filtering the answers to only contain participants familiar with Java, we analyzed other connections between the background information of participants. From now on, we refer to the participants as familiar with a topic if they answered that they were *Familiar* or *Very Familiar* with it.

Figure 4.3a represents, in a Venn diagram, the participants familiar with functional languages and refinement types. Since the most popular implementations of refinement



(a) (b)

Figure 4.3: Connection between the participants background information on refinement types and functional languages and JML.

types were developed in functional languages, we expected that the participants familiar with refinement types were also familiar with functional languages. This expectation was confirmed considering that only 1 participant familiar with refinement types is *Vaguely Familiar* with functional languages, and the remaining are all *Familiar* or *Very Familiar* with the topic.

We also compared the familiarity of participants with JML and refinement types since both techniques are used for software verification. However, only 2 participants are familiar with both topics, as presented in Figure 4.3b. Additionally, all the participants that are *Very Familiar* with JML are also *Very Familiar* with Java, and all of them are familiar with the Java language.

The following sections present the syntax options for LiquidJava features and the answers given by the participants.

4.3.2 Anonymous Variable

The anonymous variable can be used as a placeholder for the refined variable inside the refinement. This feature can minimise the effort developers, since the placeholder is shorter than most variable names and makes the same refinement reusable without changing the variable name. For this feature, we presented three possible syntaxes shown in Figure 4.4, all of them with syntaxes that cannot be used to name a variable in Java in order to prevent name clashing. The first syntax was inspired in previous implementations of refinement types where the `v` was commonly used to name the value being refined. For example `{v ↪ : int | v > 0}` could be a refinement used to state that the value of a variable must be positive, independently of the variable name. The second syntax option used the `?` symbol as the anonymous variable, giving another meaning to an existing symbol in Java (the question mark is already used to represent wildcards[2]). Finally, the last proposed

[2] https://docs.oracle.com/javase/tutorial/java/generics/wildcards.html

```
1 @Refinement("\\v > 0")
2 int biggerThanZero = 10;
```
(a)

```
@Refinement("? > 0")
int biggerThanZero = 10;
```
(b)

```
@Refinement("_ > 0")
int biggerThanZero = 10;
```
(c)

Figure 4.4: Syntax options to represent anonymous variables.

syntax used the _ (underscore) symbol, commonly used in documents as a placeholder to the add personalized information; in this case, it works as a placeholder for the refined variable. The underscore symbol has also been used in other programming languages, such as Scala[3] to ignore the name of the variable and use the symbol as a placeholder in functions, similarly to using lambda expressions.

We asked participants to evaluate the three syntaxes in terms of preference and got the answers presented in Figure 4.5a. From these answers, we discarded the first syntax since 60% of the participants evaluated it as *Not Acceptable*, the highest rate when compared to the other two options. To chose the preferable syntax between the second and third options, we took a more detailed approach since the third option has more *Preferable* answers, but also more *Not Acceptable* answers, whereas the second option has a higher rate of *Acceptable* answers. To differentiate the two options, we decided to analyse the effects of removing the answers that selected the first option as *Preferable* and got the plot in Figure 4.5b. With this exercise, it is possible to see that the number of *Not Acceptable* answers is the same in both remaining options; and the third option remains the one with the highest *Preferable* rate with 50% of the participants choosing this as the *Preferable* option. Therefore, we concluded that the third option, with the underscore, was the best choice for the anonymous variable.

4.3.3 Variables

The type of variables can be refined to ensure that all the assignments given to the variable respect the written boundaries. For the syntax of the refinements in variables, we proposed two options, presented in Figure 4.6. The first option contains only the predicate, and the variable is referred to by its name (Figure 4.6a).

The second option is closer to a lambda expression, where the refined variable is introduced at the beginning just before the refinement predicate (Figure 4.6b). The latter option is inspired by the syntax used in previous implementations of refinement types [29]. The refinements presented in these examples represent a grading system from 0 to 20, and the following conditions are expressed:

- **negativeGrade** is an integer smaller than 10;
- **excellentGrade** is an integer equal to 19 or to 20;

[3]https://www.baeldung.com/scala/underscore

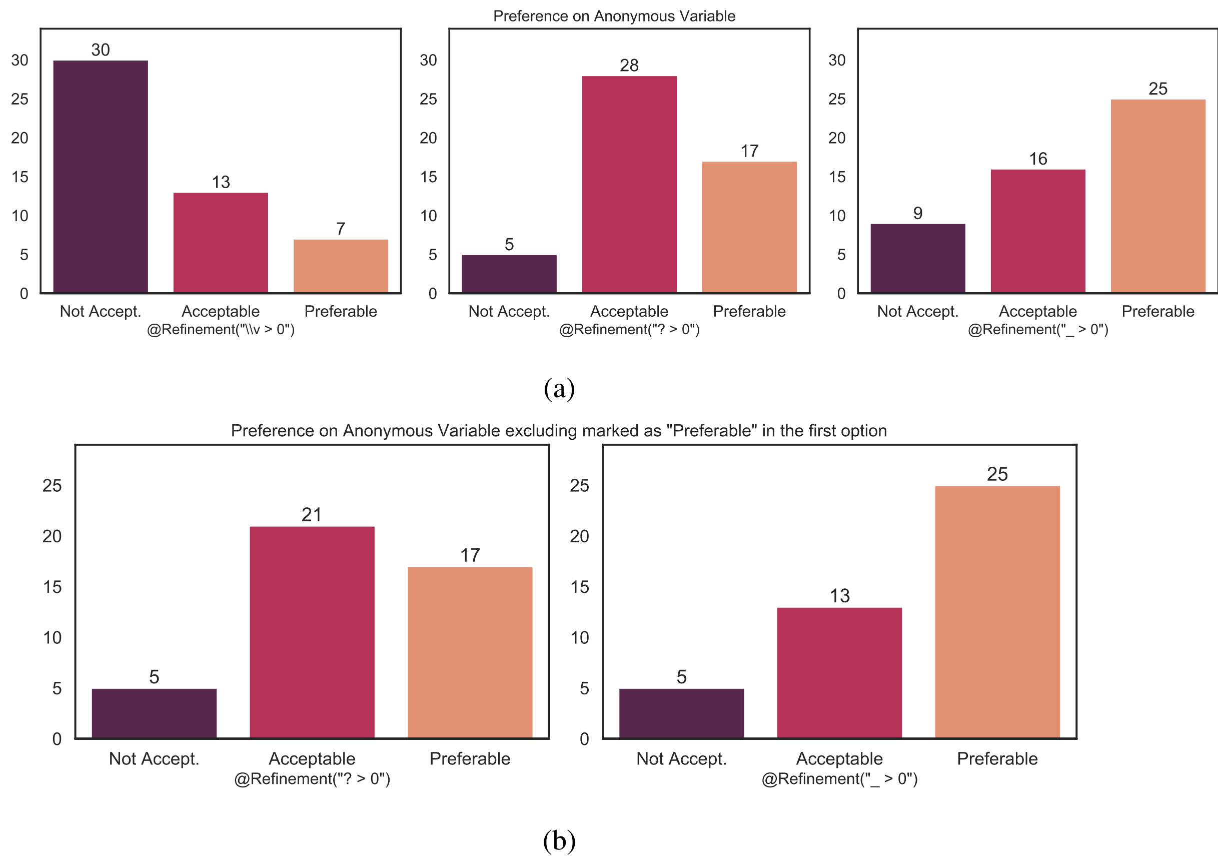


Figure 4.5: Participants answers on their syntax preference for the anonymous variable.

- **goodGrade** is an integer with a value between negativeGrade and excellentGrade.

The preferences of participants are shown in Figure 4.7 where it is evident that the first option was preferred, with 66% of *Preferable* answers and only 6% of *Not Acceptable* answers, leading to our choice of the first option as the syntax of the refinements in variables.

Additionally, since the second syntax was closer to previous implementations of refinement types, we checked if the participants familiar with them had a particular preference for this syntax. As expected, 71% of the participants familiar with refinement types chose the second syntax as *Preferable* and two of them found the first syntax *Not Acceptable*, which might show that the participants prefer to keep the syntaxes they already use. However, since the participants familiar with refinement types are only 18% of the sample, we kept the first option as the syntax for the refinements in variables.

4.3.4 Method declarations

In method declarations, we can refine the parameters and the return value. For the refinements syntax, we proposed two options shown in Figure 4.8. The first option attaches each of the refinements to the type of variable that it is refining. Therefore, the parame-

```
1 @Refinement("negativeGrade < 10")
2 int negativeGrade = 8;
3 @Refinement("excellentGrade == 19 || excellentGrade == 20")
4 int excellentGrade = 19;
5 @Refinement("goodGrade > negativeGrade && goodGrade < excellentGrade")
6 int goodGrade = 17;
```

(a)

```
@Refinement("{ x | x < 10}")
int negativeGrade = 8;
@Refinement("{ y | y == 19 || y == 20}")
int excellentGrade = 19;
@Refinement("{ x | x > negativeGrade && x < excellentGrade}")
int goodGrade = 17;
```

(b)

Figure 4.6: Syntax options for the refinements in variables.

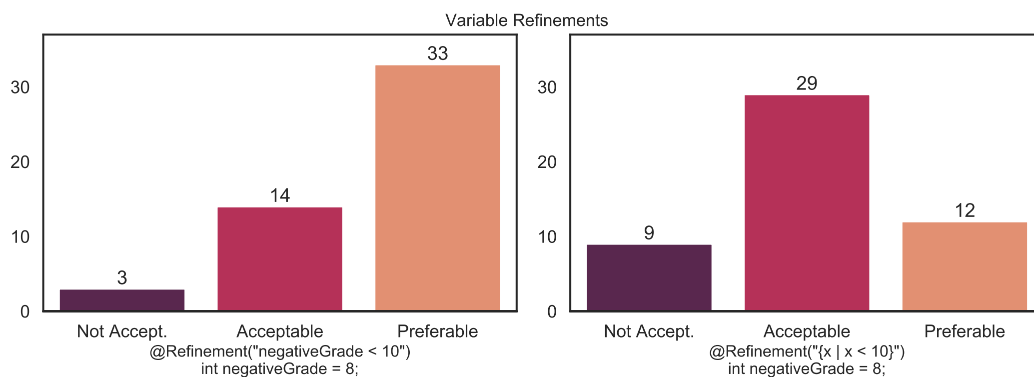


Figure 4.7: Answers on the syntax for refinements in variables.

```
1 @Refinement("_ >= 0 && _ <= 100")
2 static int percentageFromGrade(@Refinement("grade >= 0") int grade,
3                                 @Refinement("scale >= 0") int scale) {...}
```

(a)

```
@Refinement("{grade >= 0} -> {scale > 0} -> {_ >= 0 && _ <= 100}")
static int percentageFromGrade( int grade, int scale){...}
```

(b)

Figure 4.8: Syntax options for the refinement of Methods.

ters have the refinements just before their basic type, and the return refinement is above the method, before the return type. The latter option has a syntax inspired by the type signatures used in functional languages, such as Haskell.Therefore, the parameters and return refinements are written in the same line, split by the -> symbol, with a specific order starting with the parameters and finishing with the return type of the method.

The example chosen to represent both syntaxes includes a refinement for each of the parameters and the return value. These refinements express the following conditions:

- **grade**, the first parameter, is an int greater than or equal to 0;
- **scale**, the second parameter, is a positive int;
- **the return value** must be an int between 0 and 100.

Figure 4.9 shows the participants preferences on both options for the refinements in methods. In both plots, the *Preferable* option had more answers than the two other ratings; however, the first option has 10% more *Preferable* answers and 16% less *Not Acceptable* answers. Therefore, the first option was the syntax chosen for the refinements in methods.

Since the second syntax had a similar flavour to syntaxes used in functional languages, we further analysed the answers of participants familiar with functional languages. However, we did not find a relation between their background and their preference for the second syntax, since only 36% of the participants familiar with functional languages chose this option as *Preferable*, as opposed to the 56% that chose the first syntax. Furthermore, 24% of these participants pointed the second option as *Not Acceptable*, comparing to the 12% that pointed the same option for the first syntax. Therefore we cannot see a leaning of the participants familiar with functional languages to the syntax option more similar to the syntax used in functional languages.

4.3.5 Predicate Aliases

To simplify the reuse of predicates inside refinements, it is possible to create aliases for them and enable their invocation inside refinements. Like macros (used in other programming languages such as C or C++), aliases are created at the beginning of a file and can

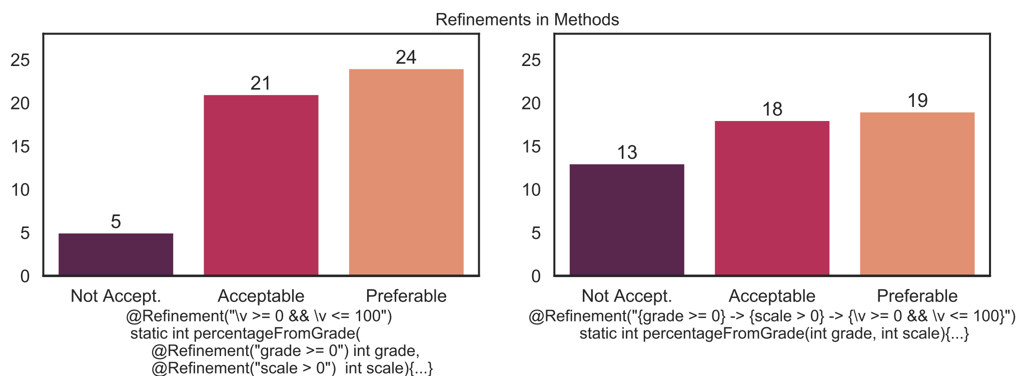


Figure 4.9: Answers on the syntax for the refinements in Methods.

take parameters to encode the predicate. Aliases can then be invoked with variables inside refinements, improving the readability of the expressions and making the predicates easily reusable.

For the survey, we proposed three possible syntaxes for the addition of aliases in LiquidJava. The syntaxes, presented in Figure 4.10, used the alias PtGrade, that describes an int between 0 and 20, representing the grade range used in the Portuguese higher education system. The two first syntaxes declare the alias above the class declaration and show an invocation of the alias inside the refinement. The first syntax has a similar style to the refinements presented in the Æon language [22], while the second syntax is similar to the creation of a method in Java. The last syntax intends to shorten the invocation of the alias by creating a new annotation for each new alias. This latter option involves creating a file for the new annotation and its refinement, but simplifies the alias invocation.

The preferences of participants on the proposed options are in Figure 4.11, from where the third option (with the separate files) was discarded, since 56% of the participants considered it *Not Acceptable*. Between the first two options, the second had a clear preference since 44% considered it *Preferable* when compared with the other two, and only 2% classified it as *Not Acceptable*. Therefore, the chosen syntax for predicate aliases in LiquidJava was the second option presented.

In the chosen syntax, the keyword `type` helps to identify the alias declaration inside the `@Refinement` annotation and differentiate the beginning of the alias declaration from the beginning of a predicate. However, this distinction could become more evident if the declarations of new constructs were introduced in other annotations, as suggested by one participant in the comments of the study. To this end, new annotations were introduced as syntax sugar allowing the developers to use more specific annotations instead of the `@Refinement`. Therefore, we created the `@RefinementAlias` annotation for the alias declaration, as exemplified in Listing 4.1, that can be placed above the class declaration and makes the usage of the `type` keyword optional.

```
1 @RefinementAlias("PtGrade(int v) { 0 <= v && v <= 20 }")
```

```
1  @Refinement("PtGrade refines Integer where (_ >= 0 && _ <= 20")
2  class MyClass{
3     ...
4     @Refinement("positiveGrade == PtGrade && positiveGrade >= 10 )")
5     int positiveGrade = 12;
6  }
```

(a)

```
@Refinement("type PtGrade(int x) { x >= 0 && x <= 20}")
class MyClass{
   ...
   @Refinement("PtGrade(positiveGrade) && positiveGrade >= 10")
   int positiveGrade = 12;
}
```

(b)

```
//File PtGrade.java
@Refinement("{int x | x >= 0 && x <= 20}")
@Retention(RetentionPolicy.CLASS)
@Target({ElementType.METHOD, ElementType.FIELD,
                  ElementType.LOCAL_VARIABLE,
                  ElementType.PARAMETER, ElementType.TYPE})
public @interface PtGrade{ }

//File MyClass.java
class MyClass{
   ...
   @PtGrade @Refinement("positiveGrade >= 10")
   int positiveGrade = 12;
}
```

(c)

Figure 4.10: Syntax options for the declaration and usage of Aliases.

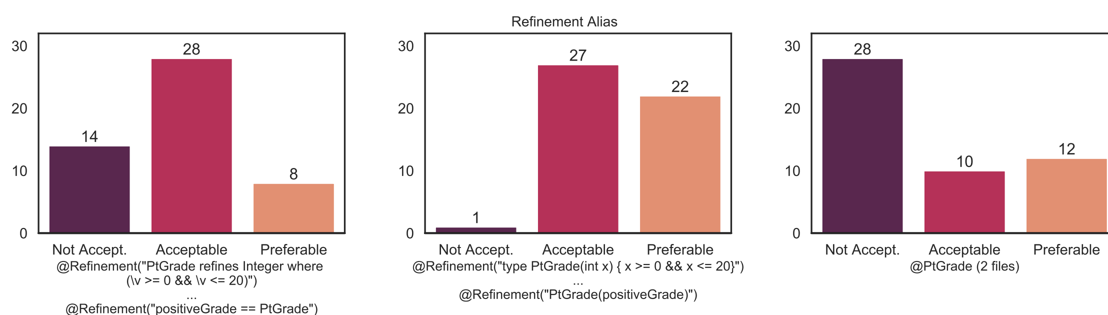


Figure 4.11: Preference answers on the alias syntax.

Listing 4.1: Examples of Refinements in Variables.

4.3.6 Ghost/Uninterpreted Functions

In order to introduce more operations inside refinements, we included the declaration and usage of uninterpreted functions inside the refinements language. Uninterpreted functions belong to the SMT-decidable logic [6] and do not have an implementation; therefore, the only information the SMT solver has about their behaviour is encoded by the axiom $\forall x, y. x = y => f(x) = f(y)$. However, despite the limited information of uninterpreted functions, they are useful to encode properties in refinements.

In LiquidJava, uninterpreted functions can be described as *ghost functions* to distinguish them from regular functions called inside Java code, since *ghosts* can only be used inside the specification. Traditionally, the term *ghost code* is used for code intended only for the program specification [21], not interfering with its execution. Since ghost functions only have the signature before they are used inside predicates, we proposed three possible placements for the declaration of these functions inside the `@Refinement` annotation, all with a syntax similar to the creation of a function in Java with the addition of the keyword `ghost` at the beginning.

Considering that all annotations must have a target code element, we proposed the placement of the ghost declaration above three different code elements: classes, methods and class attributes. The example used to illustrate the three possibilities introduces the `len` ghost function that receives a List as parameter and returns an int value. The ghost function is then used inside the refinement of the `createList()` and `append()` methods.

Participants evaluated their preference in the three placement options, and the results, presented in Figure 4.12, showed that they preferred the ghost function declaration above any method, with 44% of the participants declaring it as *Preferable* and only one participant selecting it as *Not Acceptable*.

For the declaration of the ghost functions, we introduced the `@Predicate` annotation as syntax sugar, similarly to what happened with the alias (Section 4.3.5).

```
1  //Declaration above the class
2  @Refinement("ghost int len(List xs")
3  class MyList{
4     static final int MAX_VALUE = 50;
5
6     @Refinement("len(_) == 0")
7     public List createList() {...}
8
9     @Refinement("len(_) == len(xs) + 1")
10    public List append(List xs, int k) {...}
11 }
```

(a)

```
1  //Declaration above any method of the class
2  class MyList{
3     static final int MAX_VALUE = 50;
4
5     @Refinement("ghost int len(List xs")
6     @Refinement("len(_) == 0")
7     public List createList() {...}
8
9     @Refinement("len(_) == len(xs) + 1")
10    public List append(List xs, int k) {...}
11 }
```

(b)

```
1  //Declaration above any class attribute
2  class MyList{
3     @Refinement("ghost int len(List xs")
4     static final int MAX_VALUE = 50;
5
6     @Refinement("len(_) == 0")
7     public List createList() {...}
8
9     @Refinement("len(_) == len(xs) + 1")
10    public List append(List xs, int k) {...}
11 }
```

(c)

Figure 4.12: Syntax options for the placement of the declaration of ghost functions.

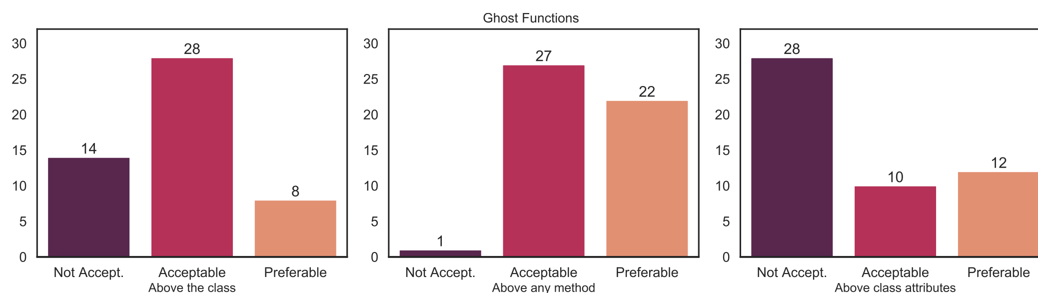


Figure 4.13: Preference answers to the placement of ghost functions.

4.4 Class Refinements

Most of the previous implementations of refinement types do not offer features to model classes since they were not developed in object-oriented programing languages, and the works that introduce refinements to languages with this paradigm only have limited constructs to model classes via their fields. Therefore, we introduce class refinements to model the objects internal state in each class method.

We created the annotation `@StateRefinement(from="predicate", to="predicate ↪ ")` to model the source state and the destination states of the class methods. The *from* and *to* predicates follow the refinements language and model the object state. Thus, the *from* predicate defines the state of the accepted objects, while the *to* predicate defines the new state of the object at the end of the invocation. If a method has different destination states depending on the source state, we can add multiple `@StateRefinement` annotation to the method, each with one of the possible conjunctions of source and destination states.

To model the class state, one can create ghost functions that represent class properties (such as the size of a list) or define a set of states that the class objects can have. The states and the ghost properties are invoked inside the predicates as functions that take the current object as argument, using the `this` keyword. Moreover, if it is necessary to refer to the previous state of the object, it is possible to use the `old` keyword with the current object, resulting in the expression `old(this)`.

This section presents the syntax used for the creation of the ghost properties (Section 4.4.1) and the state set (Section 4.4.2), exemplifying their usage.

4.4.1 Ghost

Ghost functions can be used within the context of the class to model the state properties throughout the methods. To simplify the creation of the ghost functions in this context, we introduced the `@Ghost` annotation that allows a shorter syntax for the declaration of the ghost function, removing the mandatory `ghost` keyword, and assuming that the created function only receives one parameter with the class type.

```
1  @Ghost("int size")
2  public class MyArrayList<E>{
3    @StateRefinement(to="size(this) == 0")
4    public MyArrayList(){...}
5
6    @StateRefinement(to="size(this) == (size(old(this)) + 1)")
7    public boolean add(E elem){...}
8
9    @StateRefinement(from = "index >= 0 && index <= size(this)",
10                    to = "size(this) == (size(old(this)) + 1)")
11   public boolean add(int index, E elem){...}
12
13   @StateRefinement(from = "size(this) > 0 && index >= 0 && index < size(this)",
14                    to = "size(this) == size(old(this))")
15   public void get(int index){...}
16
17   @StateRefinement(to = "size(this) == 0")
18   public void clear(){...}
19 }
```

Listing 4.2: Annotation of a list with the state transitions related to the size property.

An example of the ghost declaration is in the first line of Listing 4.2, where we model a list with the `size` property and define its value in the state transitions. In this example, we can also see the initialization of the `size` property in the constructor (that can only have a `to` state) and its modification in the methods `add` and `clear`. Moreover, the method `add` (with two parameters), and the method `get` use the current state of the `size` property to check if the index parameter is guaranteed to be within the allowed range.

4.4.2 StateSet

Java classes usually define protocols that client programs must follow. However, these protocols are primarily defined through informal documentation using natural language (inside Javadoc). Therefore, protocols are not enforced during the code development leading to runtime exceptions. In LiquidJava, we propose adding class states to model a class protocol defined by finite state machines. To this end, we enumerate the possible class states using the `@StateSet` annotation and implement the protocol in the state refinements of the methods by invoking the states with the current (**`this`**) object.

Listing 4.3 presents an example of a *FileReader* class, similar to the *InputStreamReader*[4] class from *java.io* library. In this example, we define the two possible class states as *open* and *close*, and we specify the protocol described in Figure 4.14 in the class methods. The constructor is always the first method to be invoked, representing the entrance arrow in the state machine and getting the object in a *open* state. The method `read`

[4]https://docs.oracle.com/javase/7/docs/api/java/io/InputStreamReader.html

```
1  @StateSet({"open", "closed"})
2  public class FileReader{
3     @StateRefinement(to="open(this)")
4     public FileReader(String filename){}
5
6     @StateRefinement(from="open(this)")
7     public void read(){}
8
9     @StateRefinement(from="open(this)",
10                             to="closed(this)")
11    public void close(){}
12 }
```

Listing 4.3: Annotation of the FileReader class with the state transitions allowed.

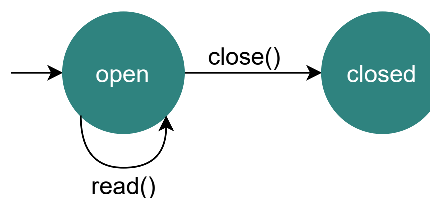


Figure 4.14: Finite State Machine that represents the protocol of the FileReader class.

can only be called if the object is in the *open* state and does not change the object state, which means that we can omit the `to` predicate, and the object will remain in the *open* state. The last method in the class, `close()`, requires the object to be in the *open* state and transforms the internal state of the object to a *close* state.

The states declared in the same `@StateSet` are disjoint, so the object can only be in one of the states from each set at any moment. In the previous example, an object cannot be *open* and *close* at the same time. However, we can define multiple state sets to encode overlapping states between sets. Listing 4.4 augments the previous specification of the `FileReader` with a new state set representing the reading process, with three possible states: *nothingRead*, *startedReading* and *finishedReading*.

```
1  @StateSet({"open", "closed"})
2  @StateSet({"nothingRead", "startedReading", "finishedReading"})
3  public class FileReader{
4        @StateRefinement(to="open(this) && nothingRead(this)")
5        public FileReader(String filename){}
6
7        @StateRefinement(from="open(this)", to="startedReading(this)")
8        public void read(){}
9
10      @StateRefinement(from="finishedReading(this)")
11       @StateRefinement(from="startedReading(this)")
12       public void getText(){}
13
14       @StateRefinement(from="open(this)", to="closed(this) && finishedReading(this)")
15       public void close(){}
16 }
```

Listing 4.4: Annotation of the FileReader class with the state transitions allowed.

In this example, the object created stays with the state *open* and *nothingRead*, from which the client can call the `read` or `close` method, since both only require the object to be *open*. If the client invokes the `read` method, the object state will stay in the *open* and *startedReading* state, but if the client invokes the `close` method, the object will be with the states *closed* and *finishedReading*. From any of these states, the client can call the `getText` method, since it accepts the object in either the *startedReading* and *finishedReading*.

Now that we have two different state sets, the objects must always be in one state of each set. Nevertheless, if a method does not change the state from one of the sets, it can be omitted from the refinement to simplify the writing of the refinements. The method `read` represents this behaviour since the `from` predicate only refers to the *open* state (from the first state set), accepting any state from the second set, and in the `to` predicate we only refer the state change for the second set (to *startedReading*) while omitting that from the first set, the object remains with the *open* state.

The previous example shows the usage of two state sets, each with its own disjoint states, but we can include any desired number of state sets to model a class. We can also combine state sets and ghost properties in the same class to model the expected behaviour.

Chapter 5

LiquidJava Type System

This chapter describes the type system of LiquidJava and its verification process. Thus, we start by presenting an overall approach of the verification (Section 5.1), followed by the introduction of verification conditions (Section 5.2) and the formal notation used to write the type system rules (Section 5.3). Finally, we describe how the verification is performed for variables, methods and classes using the liquid type checking algorithm (Section 5.4).

5.1 Approach

In liquid type checking, each expression is checked against its expected refined type through a subtyping relationship that an SMT Solver proves.

Figure 5.1 represents the type checking performed on a simple annotated code composed by a method and its invocation using a local variable. In this simple example, there are four subtyping verifications. Starting in the `inRange` method, the return value must be a subtype of the return type refinement, which means that the return expression (`lowerBound + 1`) should have a value between the value of the parameters. However, this relationship can only be proved if we consider the refinements of the parameters since we can only be sure that the return expression is lower or equal to the `upperBound` because the latter has a value strictly greater than `lowerBound`. The verification condition used to prove this relationship is presented in Equation (5.1), where we use the parameters refinements and the return expression to prove that it follows the written specification.

$$\begin{aligned} \forall lowerBound : \text{int} \,.\, true \Rightarrow \\ \forall upperBound : \text{int} \,.\, upperBound > lowerBound \Rightarrow \\ \forall return : \text{int} \,.\, return = lowerBound + 1 \Rightarrow \\ return \geq lowerBound \wedge return \leq upperBound \end{aligned} \tag{5.1}$$

$$\forall value_1 : \text{int} \,.\, value_1 = 55 \Rightarrow value_1 > 50 \tag{5.2}$$

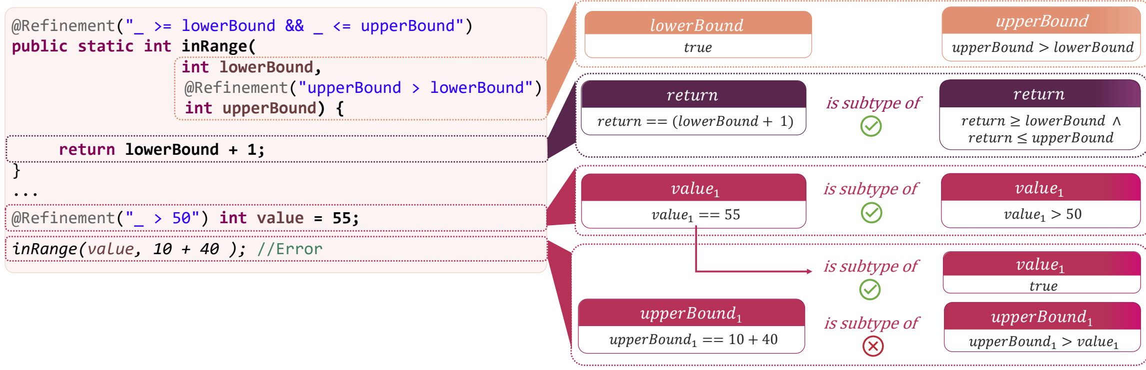


Figure 5.1: Visual representation of the subtyping relationships verified for a simple program.

$$\forall value_1 : \text{int} \, . \, value_1 = 55 \; \Rightarrow \; true \tag{5.3}$$

$$\begin{aligned} \forall value_1 : \text{int} \, . \, value_1 = 55 \Rightarrow \\ \forall upperBound_1 : \text{int} \, . \, upperBound_1 = 10 + 40 \Rightarrow \\ upperBound_1 > value_1 \end{aligned} \tag{5.4}$$

The next verification condition is generated from the local variable declaration with value assignment, where we ensure that the value stored in the variable (55 in the example) is a subtype of the declared variable refinement (value must be greater than 50. This relationship is verified through the expression in Equation (5.2) and directly proved by the SMT Solver.

The last two verification conditions are drawn from the invocation of the `inRange` ↪ method to ensure that the given arguments respect the refinements introduced in the method definition. Since the refinement of the first parameter is omitted, we use the default refinement of **`true`**, which does not apply any restriction to the type. Therefore, any value could be used for the first argument, and the subtyping relationship will always hold as represented in Equation (5.3). Finally, the last verification ensures that the expression used for the second argument is greater than the one used for the first argument, as presented in Equation (5.4). However, this relationship does not hold since the sum of 10 and 40 is not greater than the current value of `value` (55), which means that the specification is not followed, and an error message will be displayed to the developer.

This simple example represents broadly the approach taken for the verification of the refinements inside the Java language. The remainder of the current chapter details this approach by describing verification conditions (presented inside the equations), and how type checking handles refinements inside Java code.

5.2 Verification Conditions

Verification conditions (VC's) represent the subtyping relationships that must be verified inside liquid type checking [29]. They work as an intermediate representation between the program specification and the SMT Solver encoding. VC constraints are composed of simple quantifier-free predicates and of implications, as represented in Equation (5.5).

$$\forall x : \mathrm{B}.p \Rightarrow c \tag{5.5}$$

The presented implication states that for a variable `x` of type `B`, if the predicate `p` holds so must the constraint `c`. These implications include the refinements of variables followed by the final predicate that we mean to prove. Equation (5.6) is the last verification condition presented in the previous section and contains two implications for the two variables used in the example, and the final predicate is the condition we want to prove.

$$\begin{aligned} \forall value_1 : \mathrm{int} \,.\, value_1 = 55 \Rightarrow \\ \forall upperBound_1 : \mathrm{int} \,.\, upperBound_1 = 10 + 40 \Rightarrow \\ upperBound_1 > value_1 \end{aligned} \tag{5.6}$$

During the liquid type checking, the verification conditions are gathered and translated to the SMT Solver. For the translation, we introduce all the variables presented in the implications and join all implication predicates (as antecedents) with the negation of the last predicate (as consequent). Thus, the formula sent is valid if the SMT Solver finds it unsatisfiable. For the previous example, the logical formula sent to the SMT Solver is represented in Equation (5.7).

$$value_1 = 55 \wedge upperBound_1 = (10 + 40) \wedge \neg(upperBound_1 > value_1) \tag{5.7}$$

Since the formula is satisfiable ($upperBound_1$ is less or equal to $value_1$), and not unsatisfiable as expected, it is impossible to prove the VC and an error message will be displayed to the developer.

5.3 Notation

To verify LiquidJava programs, we apply the liquid type checking algorithm. Type checking rules formally define the type checking process. The following section (Section 5.4) details the verification of LiquidJava features, introducing the type checking rules along with the decisions and verification examples. Thus, the present section introduces the notation used in the referred rules.

Figure 5.2 introduces the notation used in liquid type checking rules. We start by defining the meta-variables used inside the rules, for types, variables, fresh-variables, expressions and state expressions. For the context, we split it in two: a global (Γ) and a local

Meta-variables
$B, T, U, \ldots$: types
$\alpha, \beta, \ldots$: fresh variables
$x, y, \ldots$: variables
$a, b, \ldots$: state expressions
$e, p, \ldots$: expressions

Typing context
Γ : global context
Δ : local context

Types
B : base Java type
$\{x : B|e\}$: refined type
$f : \{\overline{x_i : T_i|e_i}\} \xrightarrow{a,b} \{v : T|e_v\}$: function type

Type Operations
$U <: T$: subtyping relationship, reads as U is a subtype of T
$e[x := y]$: substitution, reads as e with x substituted with y
$obj[a \to b]$: change the object state from a to b, reads as obj changes fromstate a to state b

Figure 5.2: Notation.

(Δ); to store the instance values of the variables in a distinct place from the declaration refinements. Both contexts can store variables and methods. Variables are stored with the basic Java type (*B*) and the refinement (*e*). As for methods, we store their parameters (represented in the notation by the overline as a limited sequence of variables), the return type and the from-to state transitions allowed in the method (represented by $\xrightarrow{a,b}$). The rules may also use type operations, specifically subtyping relationships, substitutions and change of object state.

All terms are assumed to be in Administrative Normal Form (ANF), as standard in Liquid Types [29]. ANF requires terms to be represented either as constants or variables to ensure the decidability of the type checking algorithm.

5.4 Refinements Verification

This section describes the type checking rules, including the verification steps taken to prove if all refinements are respected or if any of them is violated. Thus, we present the verification of declaration, assignment and access of variables (Section 5.4.1) and class fields (Section 5.4.2), the use of branch conditions (Section 5.4.3) that help in the verification of return values of methods, and the verification of invocations of class methods (Section 5.4.4) and methods from external libraries (Section 5.4.5). Finally, we present

the declaration of class methods (Section 5.4.6) that model object state with ghost properties and state sets, and that can override methods that belong to superclasses. The type checking rules are introduced along with the verification features with the aid of examples, and the complete set of rules can be found in Appendix A.

5.4.1 Variables

Variables appear in declarations, assignments and as expressions.

5.4.1.1 Variable Declaration

```
1 @Refinement("a > 0")
2 int a;
```

Listing 5.1: Variable Declaration Example.

Variables are the simpler elements of code to which we can add refinements. The refinements will ensure that the variable only is assigned values that are a subtype of the annotated refinement type.

$$\frac{\Gamma, \{x : T, e\}; \Delta \vdash S\ valid}{\Gamma; \Delta \vdash \text{@Refinement}(e)\ T\ x;\ S\ valid} \quad \textbf{(var-decl)}$$

Equation (**var-decl**) represents the typing rule for the variable declaration where we introduce its basic type and refinement in the global context. For example, the declaration from Listing 5.1 would add to the context $\{a :\ int \mid a > 0\}$.

5.4.1.2 Variable Assignment and Variable Access

In imperative programming languages, like Java, variables are mutable and can be assigned multiple values throughout the program. Pierce [50] presented a way of saving these values as references using operations of allocation, dereferencing and assignment to store and retrieve the concrete values. In this approach, a reference is stored in a cell of type `Ref T`, any reference stored in this cell needs to be a subtype of `T`, and a dereference of the cell produces a value of type `T`.

Chung [13] names the previous approach as weak updates because the dereference of the cell always produces a value with the weaker supertype `T` instead of the stronger type of the stored value. Using weak updates would produce a type error in Listing 5.2, where that example is perfectly correct in the Java semantics. Thus, we follow the strong updates approach.

To save the declared refinement of the variables and the refinement of each assigned expression, we split the information into the global and local contexts. Thus, the refinement introduced in the declaration is stored in the global context (Γ), and the refinements

```
1 @Refinement("a > 0")
2 int a = 10; //(a == 10) <: (a > 0) ~> Correct
3 @Refinement("b > 5")
4 int b = a; //(b == a && a > 0) <: (b > 5) ~> Type Error
```

Listing 5.2: Variable assignments under a weak updates type checking.

Figure 5.3: Split of the storage of variable's refinements into local and global context.

of assignments are stored in the local context (Δ), as exemplified in Figure 5.3 for the previous example. This way, for the liquid type checking of assignments, we verify if the assignment is a subtype of the refinement of the variable stored in the global context, and if it is, the new refinement is added to the local context, as formalized in Equation (**var-assign**).

$$\frac{x : T \in \Gamma \quad \Gamma; \Delta \vdash e : K \quad \Gamma; \Delta \vdash K <: T \quad \Gamma; \Delta, x : K \vdash S \; valid}{\Gamma; \Delta \vdash x = e; \; S \; valid} \qquad \textbf{(var-assign)}$$

$$\frac{x : T \in \Delta}{\Gamma; \Delta \vdash x : T} \qquad \textbf{(var)}$$

To ensure that the stronger types of the variables are used when they are referred to in the code, we get their refinement from the local context, as presented in Equation (**var**). With this approach, the example of Listing 5.2 becomes valid since the verification conditions for the assignment of `b` change to the ones presented in Equation (5.8).

$$\begin{aligned} \forall b : \text{int} \;.\; b = a_1 \Rightarrow \\ \forall a_1 : \text{int} \;.\; a_1 == 10 \Rightarrow \\ b > 5 \end{aligned} \tag{5.8}$$

To distinguish between the usage of the declared refinement from the remaining (stronger) refinements, we append a number suffix to the name of the variable stored in the local context. This also allows us to distinguish different values that the variable might have during its lifetime. The verification of the last assignment in Section 5.4.1.2 represents this situation, where it is helpful to distinguish different assignments of `a` to prove that the assignment to `c` respects the declaration refinement.

However, suppose a variable has a global refinement that refers to other variables. In that case, the verification conditions will use the global refinements of the latter variables

```
1 @Refinement("a > 0")
2 int a = 10;
3 @Refinement("b > 5")
4 int b = a;
5 a = 50;
6 @Refinement("_ > 55")
7 int c = a + b;
```

$$
\begin{aligned}
\forall c : \text{int} \;.\; c == a_2 + b_1 \Rightarrow \\
\forall a_2 : \text{int} \;.\; a_2 == 50 \Rightarrow \\
\forall b_1 : \text{int} \;.\; b_1 == a_1 \Rightarrow \\
\forall a_1 : \text{int} \;.\; a_1 == 10 \Rightarrow \\
c > 55
\end{aligned} \tag{5.9}
$$

Figure 5.4: Code and verification conditions on usage of variables assignments in different code places.

since those refinements are always true in any code location. This way, there is no need to track the effects of an assignment in other variables besides the assignee. For example, in Listing 5.3, variable `d` has a global refinement that depends on `a`. To ensure that `d` has a value always smaller than `a`, it needs to have a value that respects the global refinement of `a`. Thus the first assignment produces a type error, but the second one is correct.

To make the first assignment also correct, the type-checking algorithm would need to keep track of the dependencies between variables and verify if the refinements of the dependent variables still hold with new assignments. Therefore, using the previous example, each time `a` changes its value, there would be a need to verify if the new value respects the global refinement of `a` and also check if the value assigned to `d` still respects the refinement. Listing 5.4 illustrates this option with an example where the value assigned to `a` provokes the invalidity of the refinement of `b`.

Given the added number of verifications, we decided not to follow this option. Thus, each time a variable is assigned a new value, it is only necessary to check if the value respects the variable refinement without checking if other variables that depend on the changed one still respect the refinements.

```
1 @Refinement("a > 0")
2 int a = 10;
3 @Refinement("d < a")
4 int d = 7; // (d == 7) && (a > 0) <: (d > a) ~> Type Error
5 d = −1; // (d == −1) && (a > 0) <: (d > a) ~> Correct
```

Listing 5.3: Use of global refinements of variables within other refinements.

```
1 @Refinement("a > 0")
2 int a = 10;
3 @Refinement("d < a")
4 int d = 7; // Correct
5 a = 5; // Correct for a,
       ↪ refinement of d
       ↪ becomes incorrect
```

Listing 5.4: Other tracking option (not implemented).

5.4.2 Class Instance Fields

It is possible to introduce refinements to model the class fields and ensure that the refinements hold in any method where the fields are assigned new values. Listing 5.5 shows the

class `Color` that defines a color by its RGB value and stores each color component in one field, making the total of three fields all with the same value limitations on the range of 0 and 255. To avoid repeating these refinements, we created the `RGB` alias and used it as a refinement to each field using the variable name or the anonymous variable.

```
1 @RefinementAlias("RGB(int v) { v >= 0 && v <= 255}")
2 public class Color {
3     @Refinement("RGB(r)") int r;
4     @Refinement("RGB(_)") int g;
5     @Refinement("RGB(_)") int b;
6     ...
7 }
```

Listing 5.5: Introduce refinements in class fields.

Wherever an assignment to one of the fields occurs, the refinement will be verified using the same principles of variable verification.

5.4.3 Branching Conditions

Inside branches of if statements, there are additional assumptions available, based on whether the branching condition was true or not.

Assuming that the expressions inside if conditions are possible to translate to ANF, we can convert them into predicates and use them to verify the code within the branch scope.

$$\frac{\Gamma \vdash e : bool \quad \Gamma, \{\alpha : int|e\} \vdash S_1\ valid \quad \Gamma \vdash S_2\ valid}{\Gamma; \Delta \vdash (\text{ if } (e)\{S_1\}\ S_2\ )\ valid} \quad \textbf{(if-then)}$$

$$\frac{\Gamma \vdash e : bool \quad \Gamma, \{\alpha : int|e\} \vdash S_1\ valid \quad \Gamma, \{\beta : int|\neg e\} \vdash S_2\ valid \quad \Gamma \vdash S_3\ valid}{\Gamma; \Delta \vdash (\text{ if } (e)\{S_1\} \text{ else } \{S_2\}\ S_3\ )\ valid} \quad \textbf{(if-then-else)}$$

Equation (**if-then**) and Equation (**if-then-else**) represent, formally, the introduction of a fresh variable with the refinement from the if condition into the context both for the *then* and *else* branches. Adding the branch condition as a refinement in the context, allows us to use the branch information to verify statements inside the desired scope.

5.4.4 Method Invocations

Method types describe the type of the parameters and the expected return type. The return type is used to verify if the method returns the expected value. Parameter types are used to validate the correct invocation of the method.

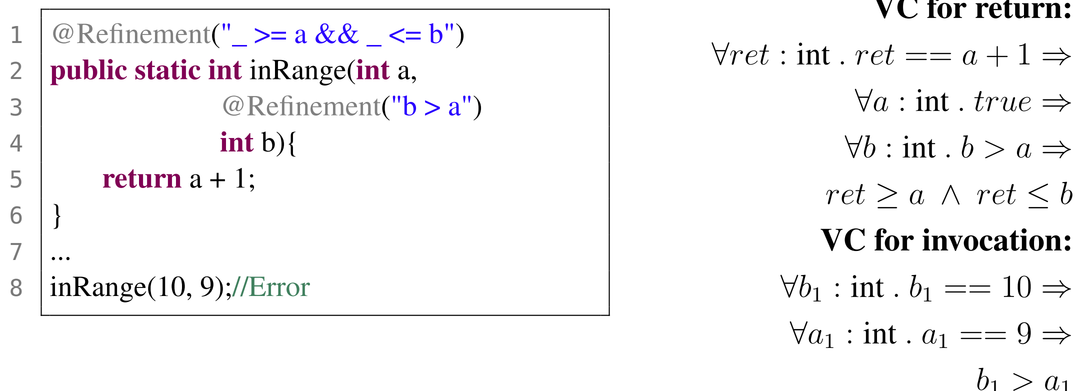


Figure 5.5: Verification of `inRange` method return and invocation.

Parameter types are declared immediately before the parameters basic types. The return refinement is declared before the method signature, above the return type, where the anonymous variable is an alias for the return value.

Figure 5.5 presents a method and the verification conditions for both the return and invocation. First, we verify if the return value is within the range of both parameters, which can only be valid if we consider that `b` is strictly greater than `a`, as represented by the refinement. The verification condition for the return is also represented in the figure, where we can see the substitution of the return value for the ret variable and the sequence of refinements used to prove that the return value follows the desired refinement. Secondly, we verify he invocation of the `inRange` method with the given parameters where we find an error, since the second parameter does not follow the refinement, being lesser than the first parameter.

To address more complex methods, we also added support for verifying recursion inside methods. To implement the verification of a recursive call, we only need to verify if the arguments respect the declared refinements. Additionally, if the recursive call is inside the return expression, we can use the return refinement of the method as the refinement of the invocation.

Figure 5.6 presents an example that shows the verification on a method with recursive calls and if statements. In the `fib` method, the only parameter has a refinement that should be respected inside the recursive calls. As detailed in the first verification condition, we can only conclude that the invocation of `fib(n-2)` respects the parameter refinement because, in that scope, we know that the value of n is greater than 1, so $n - 2$ always will be greater or equal to zero. As for verifying the returned expression type against the expected return type, we use the return refinement of the method and the branch condition to verify that the refinement is respected.

At the moment, we can only verify loop instructions through recursion, all other loop constructs (e.g: for, while, for-each, etc.) are not verified and represent a part of future

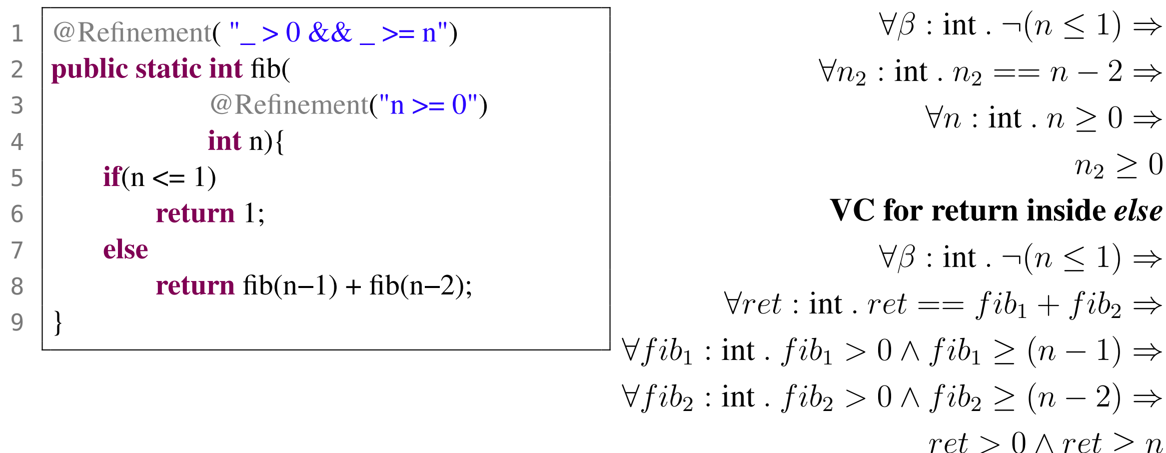


Figure 5.6: Verification of the body of the `fib` method.

work.

5.4.5 External Libraries

Until now, we have only seen how we can introduce refinements in code that we are developing. However, most projects use existing libraries without changing them. Therefore, external libraries need to be annotated with refinement types so that code with external invocations can be verified. To this end, users can create interface files that contain the annotated signatures of the methods from the library they want to use. Section 5.4.5 represents a part of the annotated file for the `java.lang.Math` library.

```
1  @ExternalRefinementsFor("java.lang.Math")
2  public interface MathRefinements {
3      @Refinement("(arg0 > 0)?( _ == arg0):(_ == −arg0)")
4      public int abs(int arg0);
5
6      @Refinement("(a > b)? (_ == a):(_ == b)")
7      public int max(int a, int b);
8
9      @Refinement("(a > b)? (_ == b):(_ == a)")
10     public int min(int a, int b);
11
12     @Refinement(" _ > 0.0 && _ < 1.0")
13     public long random(long a, long b);
14 }
```

To annotate an external library, we start by creating an interface and including the name of the external library inside the `@ExternalRefinementsFor` annotation. Then, for each method of the class, we can introduce refinements for the parameters and the re-

turn value. Afterwards, external libraries invocations are verified as any other method invocation.

Since refinements for libraries are the same independently of the client code, we added the refinements of some Java libraries and made those files available in a Github repository[1] so that developers can find the desired libraries annotated, and the community can add more library annotations.

5.4.6 Method Definitions

Class methods can have refinements in the parameters and return value, as we saw before, but they can also have `@StateRefinement` annotations that model the behaviour of the object. Within the `@StateRefinement` annotations it is possible to introduce the expected type state of the object when the method is invoked (in the `from` argument) and the type state of the object at the end of its execution (`to` argument).

$$\frac{\Gamma, \{f : \{\overline{x_j : T_j | e_j}\} \xrightarrow{a,b*} \{v : T|e\}\}, \{expret : T|e\} \vdash S,\ S_1\ valid}{\begin{array}{ll}\Gamma; \Delta \vdash & \text{@StateRefinement(from=}e_{f1},\ \text{to=}e_{t1}) \\ & \ldots \\ & \text{@StateRefinement(from=}e_{fn},\ \text{to=}e_{tn}) \\ & \text{@Refinement(}e\text{) public } T\ f\ (\text{@Refinement(}\overline{e_j}\text{)}\ \overline{T_j\ x_j})\ \{S_1\}\ S_2\ valid\end{array}}$$

(met-decl)

To store the method declaration in the context, we follow the rule presented in Equation (**met-decl**), where we store for the method f all the parameters with their types and refinements (which is represented by the vector of $x_j : T_j|e_j$), the return type and its refinement, and also all the possible state transitions represented with the `@StateRefinement`'s annotations. These transitions are represented in the rule by the symbol $\xrightarrow{a,b}$, where a represents the union of all possible `from` states ($a = \bigvee_{i=1}^{n} e_{fi};$) and b represents the `to` states available according to the `from` state used ($\bigvee_{i=1}^{n} e_{fi} \rightarrow e_{ti}$).

As for the invocation of the methods with a target object, we need to check the parameters used in the invocation and the state of the target object at the time of invocation. Finally, when the invocation is finished, we must change the object's current state inside the context. The rule in Equation (**met-inv**) can formally describe this type checking.

$$\frac{\begin{array}{ccc}\Gamma \vdash y_i : T_i & \Gamma; \Delta \vdash obj : U & \Gamma \vdash U <: \{\beta : U|a\} \\ f : \{\overline{x_i : T_i}\} \xrightarrow{a,b} \{v : T \mid e\} \in \Gamma & & \Gamma; \Delta, obj[a \rightarrow b] \vdash S\ valid\end{array}}{\Gamma; \Delta \vdash obj.f\ (\overline{y_i}) : \{T|e[x_i := y_i]\}\ S\ valid}$$

(met-inv)

The same rule applies to methods that do not have any `@StateRefinement` annotation and the class is not modelled using States or Ghost properties, since it is identical to have the object type states as true (`@StateRefinement(from="true", to="true")`).

[1] `https://github.com/CatarinaGamboa/liquid-java-external-libs`

The type state used to model the class can depend on ghost properties and on states from state sets. When a ghost property or state set is created for a class, an instance of the class always has to have a value for the property and has to be in one of the allowed states, as will be detailed below with examples.

5.4.6.1 Ghost Properties

When a ghost property is created, it is converted into an uninterpreted function that can only have one value at a time for a given object. To ensure that at every moment the ghost has a defined value, we have a default starting value (e.g., 0 when the property has type int) and when the property is not referred in the method transition states, we assume that its value has not changed (`ghost_prop(this)== ghost_prop(old(this))`). The `old` keyword can be used inside refinements to refer to the previous version of an object, specifically the object version at the time of the method invocation (similarly to its meaning in other languages such as JML [36] and Dafny [37]).

Figure 5.7 presents the annotation of the class `java.util.ArrayDeque` using the `size` ↪ ghost property. The method representing the constructor (the first of the sequence) has no `@StateRefinement` which means that `size` will start with the default value of 0. Additionally, the `size()` method does not change the internal state of the object, since it represent a getter method, thus it does not have any `@StateRefinement` annotation and we can assume that the `size` property has the same value as before. Finally, the two methods that change the value of `size` are `add` and `remove`. While `add` only has a `to` state, which means we can invoke the method in any state of the object, `remove` also includes a `from` state restraining the method invocation to when the `size` is greater than zero.

Below the annotation of the ArrayDeque library, we have a small client code that, along with Equation (5.10), exemplifies the liquid type checking of a class with a ghost variable. To check the invocation of the `remove()` method and ensure that the `size` of `ad` is positive, we need to get the previous versions of the object since together they represent the total value of the `size` property. Thus, to prove that $size(ad_2) > 0$, we need to use the state of the object from the moment it was created until the current state. Inside the verification conditions, it is possible to see that the `old(this)` invocations were translated to the previous state of the object; for example, in the second line, $old(ad_1)$ was translated to ad_0, the object state at the start of the `add` method.

5.4.6.2 StateSet

Each `@StateSet` created for a class is represented by an uninterpreted function with as many possible values as states added inside the annotation. We ensure that the object is always in one of the states from each set, and, again, we assume that when there is no state transition, the object state remains in the same state.

```
@ExternalRefinementsFor("java.util.ArrayDeque")
@Ghost("int size")
public interface ArrayDequeRefinements<E> {

    public void ArrayDeque();

    @StateRefinement(to = "size(this) == (size(old(this)) + 1)")
    public boolean add(E elem);

    @StateRefinement(from = "size(this)> 0",
                     to = "size(this) == (size(old(this)) − 1)")
    public void remove();

    @Refinement("_ == size(this)")
    public int size();
}
...
ArrayDeque<Integer> ad = new ArrayDeque<Integer>();
ad.add(10);
ad.size();
ad.remove();
```

$$\begin{aligned} \forall ad_0 : \text{ArrayDeque} \,.\, size(ad_0) == 0 \Rightarrow \\ \forall ad_1 : \text{ArrayDeque} \,.\, size(ad_1) == size(ad_0) + 1 \Rightarrow \\ \forall ad_2 : \text{ArrayDeque} \,.\, size(ad_2) == size(ad_1) \Rightarrow \\ size(ad_2) > 0 \end{aligned} \tag{5.10}$$

Figure 5.7: ArrayDeque verification with ghost property size.

To exemplify the type checking when a class is modelled through a state set, we introduced annotations to the Socket class[2] from the external library `java.net`, as presented in Listing 5.6. The Socket class contains methods that are underlying connected by a protocol order specified only through informal documentation. This can lead to an incorrect usage of the library: as an example, the connect method should not be called multiple times sequentially, or an exception will occur[3]. A state machine can represent the protocol that a client of the Socket class must use, as shown in Figure 5.8, and it can be modelled in the code using the `@StateSet` and `@StateRefinement` annotations. In this example, the first method represents the constructor, thus containing only a `to` state with the `unconnected` state, however, if the constructor had no annotation we would assume that the starting state of the object would be the first of the state set (`unconnected` in this case), so, for this example, it would be optional to include the constructor annotation.

[2]https://docs.oracle.com/javase/7/docs/api/java/net/Socket.html

[3]`https://www.tabnine.com/code/java/methods/java.net.Socket/bind?snippet=59225ab84002b00004dbfef2`

The `bind` and `connect` methods have a simple state transition, no different from what we analysed in previous examples. The `sendUrgentData` method only contains a `from` state, requiring the object to be in a `connected` state when the method is invoked and does not change the object state, thus the method does not need a `to` state. As for the `close` method, it can be invoked in any state of the object, except the `closed` state, so adding a `from` type state with all the available states as options (`from = "unconnected(this)|| ↪ bound(this)|| connected(this)"`) is the same of having the negation of the `closed` state.

The verification of the state transitions using a state set is similar to the verification when using ghost properties. When a method contains a `from` state we compare the current object state with the expected one, and if it is valid, we change the object to the `to` state and store it in the context. The client code in Listing 5.6 represents an error when invoking `close` when the object is already in the `closed` state, as emphasized by Equation (5.11).

5.4.6.3 Combining State Sets and Ghost properties

It is also possible to model a class with both state sets and ghost properties, have refinements on parameters and return of methods, and also have field annotations.

Listing 5.7 represents a simplified version of an Auction class where we use refinements to model the expected values of the method's parameters, as well as the steps of the auction and the properties that should be enforced. Specifically, this class models the following properties:

- There are three sequential states for the auction: open, close and prizeSent;
- The clients that can bid have an id number (positive), and the same client cannot bid twice sequentially (raising their own bid);
- The bid values need to be positive and always greater than the last placed bid;
- After the auction is closed, the prize must only be sent if a client placed a bid.

Below the annotation of the Auction class, we can find a client code exemplifying three refinement type errors due to incorrect use of the protocol (each error presented assumes that the previous errors were removed).

5.4.6.4 Class Inheritance

In Java, we can define hierarchies by describing the classes that extend the behaviour of others. Since subclasses have the behaviour of the parents and extend it further, they will also automatically inherit the refinements provided to the class methods, which means that the user does not need to rewrite the same annotations. However, if the subclass aims to override a method and its refinements, the new refinements must follow the Liskov

```
@ExternalRefinementsFor("java.net.Socket")
@StateSet({"unconnected", "bound", "connected", "closed"})
public interface SocketRefinements {
    @StateRefinement(to="unconnected(this)")
    public void Socket();
    @StateRefinement(from="unconnected(this)",
                     to="bound(this)")
    public void bind(SocketAddress add);
    @StateRefinement(from="bound(this)",
                     to="connected(this)")
    public void connect(SocketAddress add);
    @StateRefinement(from="connected(this)")
    public void sendUrgentData(int n);
    @StateRefinement(from="!closed(this)",
                     to="closed(this)")
    public void close();
}
...
//Client code (addr1, addr2 and port defined before)
Socket socket = new Socket();
socket.bind(addr1, port);
socket.connect(addr2);
socket.sendUrgentData(90);
socket.close();
socket.close();//Error
```

Listing 5.6: Annotation of Socket class using a state set.

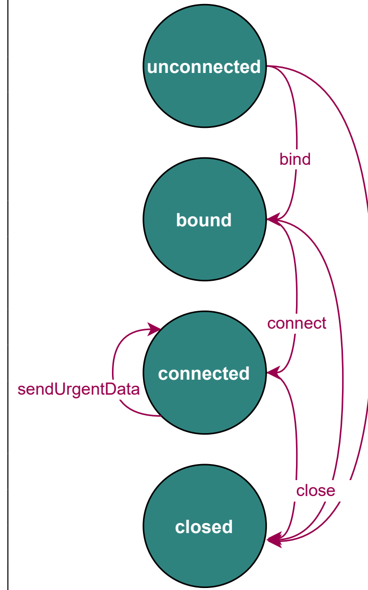


Figure 5.8: Finite State Machine that represents the protocol.

$$\begin{aligned}
\forall socket_0 : \text{Socket} \,.\, stateset_1(socket_0) == unconnected \;\; &[\text{line } 20] \Rightarrow \\
\forall socket_1 : \text{Socket} \,.\, stateset_1(socket_1) == bound \;\; &[\text{line } 21] \Rightarrow \\
\forall socket_2 : \text{Socket} \,.\, stateset_1(socket_2) == connected \;\; &[\text{line } 22] \Rightarrow \\
\forall socket_3 : \text{Socket} \,.\, stateset_1(socket_3) == stateset_1(socket_2) \;\; &[\text{line } 23] \Rightarrow \\
\forall socket_4 : \text{Socket} \,.\, stateset_1(socket_4) == closed \;\; &[\text{line } 24] \Rightarrow \\
!(state - set_1(socket_4) == closed)&
\end{aligned} \tag{5.11}$$

Figure 5.9: Socket class annotation and client verification.

```
1  @StateSet({"open", "closed", "prizeSent"})
2  @Ghost("int currentValue")
3  @Ghost("int clientNumber")
4  public class Auction{
5      @StateRefinement(to="open(this) && currentValue(this) == start && clientNumber(this)
           ↪ == −1")
6      public Auction(@Refinement("_ >= 0")int start){...}
7
8      @StateRefinement(from="currentValue(this) < value && clientNumber(this) != client",
9                       to = "currentValue(this) == value && clientNumber(this) == client")
10     public void bid(@Refinement("_ > 0") int client, @Refinement("_ > 0") int value){...}
11
12     @StateRefinement(to = "closed(this)")
13     public void endAuction(){...}
14
15     @StateRefinement(from="closed(this) && clientNumber(this) != −1 && currentValue(this
           ↪ ) > 0", to = "prizeSent(this)")
16     public void sendPrize(){...}
17
18
19     public static void main(String[] args){
20         Auction auction = new Auction(300);
21         auction.bid(1, 350);
22         auction.bid(2, 500);
23         auction.bid(1, 400);//Error, due to the value bid
24         auction.bid(2, 600);//Error, due to same client
25         auction.sendPrize();//Error, sendPrize before end of auction
26         auction.endAuction();
27     }
28 }
```

Listing 5.7: Annotation of class with ghost properties and state sets simultaneously.

substitution principle [41]. This principle ensures that any operation applied to a supertype must be possible to apply to the subtype and have similar results. To ensure this behaviour, the pre-conditions of the subclass should be broader than the super-classes pre-conditions, and the post-conditions should be narrower.

Listing 5.8 presents a simple example of the relationship in super and subclass refinements for the same method. In the example, `Animal` is the superclass of `Mammal` and has the method `untilMaximumLifespan` that returns the remaining time to achieve the maximum lifespan. Following the Liskov principle, the age parameter has a broader range in the subclass when compared to the superclass ($(age >= 0) :> (age > 0)$) to ensure that any client that invokes the method with a valid value for the supertype can also invoke the method from the subclass. For the return refinements, we ensure that the return value of the subclass is within the range of the superclass ($(_ < 250) <: (_ < 11000)$), so that the client receives a value inside the expected limit. The numbers that bound the return values were inspired by the maximum lifespan in the animal realm in general and in mammals[4].

```
1 public class Animal{
2     @Refinement("_ < 11000")
3     public untilMaximumLifespan(@Refinement("_ > 0") int age){...}
4 }
5
6 public class Mammal extends Animal{
7     @Refinement("_ < 250")
8     public untilMaximumLifespan(@Refinement("_ >= 0") int age){...}
9 }
```

Listing 5.8: Refinements in Super and Subclass.

5.4.7 Discussion

This section presented the current type system of LiquidJava and the formal definition of the main features of the language. Concretely, for variables, we presented the formal verification for their declaration, assignments and accesses, and presented a similar verification for class fields. For methods, we defined the verification of their invocations and return values, the use of recursion, and how we can use branch conditions inside type checking. We also showed how to introduce refinements for external libraries and use them inside the code to be verified. Finally, we introduced the declaration of class methods that can include state transitions using ghost properties and state sets to model the object state.

We also presented the verification on methods declarations that override the refinements from superclasses; however, this feature remains to be formalized, despite being implemented in our prototype.

[4] https://www.nationalgeographic.com/animals/article/animals-oldest-sponges-whales-fish

Other Java features that we would like to verify but are not currently supported include loop constructs, expansion on object aliasing, and methods with objects as parameters. Loop constructs such as *while* and *for* are used in many programs, which makes them essential to be included in the verification system. Therefore, it is vital to introduce loop invariants in the code modelling, which represents one of the main challenges since Java annotations cannot have loops as targets (they can only be attached to variables, methods, fields, parameters and classes). Aliasing, on another end, is typical in languages that allow mutable objects since it happens when multiple variables point and can change the same memory address. Hence, supporting the tracking of aliasing on objects allows the verification of more complex data structures and real-world programs. Finally, class methods usually receive other objects as parameters, but, at the moment, it is not possible to use their state in other refinements or make a state change on them during a method. To add this feature, it is necessary to change the type checking algorithm for methods and, for example, include the changes of parameters in the state refinements of the method.

Chapter 6

An Implementation of LiquidJava

Having a formalization of LiquidJava in place, this chapter describes our implementation of the type checker detailing the system architecture (Section 6.1.1), the verification with the SMT Solver (Section 6.1.2), and the error messages (Section 6.1.3). Finally, the chapter describes the type checker integration in an IDE (Section 6.2), with the goal of improving the usability of LiquidJava.

6.1 LiquidJava System

This section adds the implementation details to the system design (introduced in Section 3.2.2) by describing the architecture of the system, the verification integration with the SMT Solver and the error messages.

6.1.1 Architecture

The compilation pipeline of LiquidJava is summarized in Figure 6.1 (extending the scheme presented in Figure 3.1).

The system is built on top of Spoon (introduced in Section 3.2.2), a library for Java source code analysis and transformation which offers a metamodel of the abstract syntax

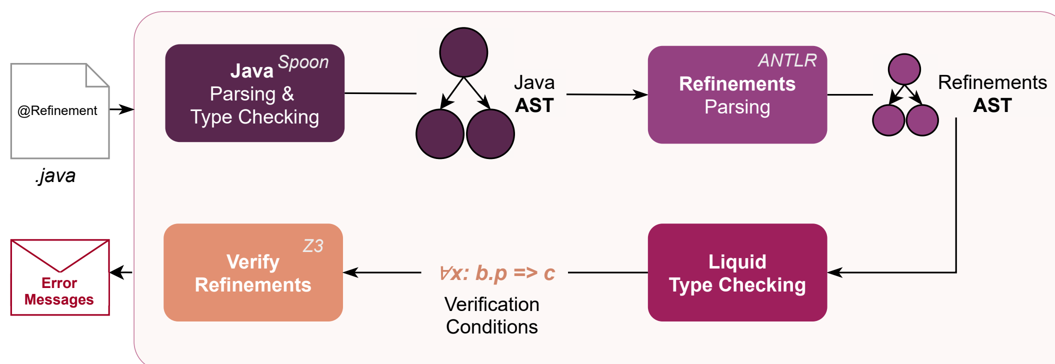


Figure 6.1: Architecture of the LiquidJava System.

tree (AST) of the input program allowing new compiler passes to access and transform the AST. Our implementation traverses the AST to perform the liquid type checking.

Refinement annotations are parsed using the ANTLR framework [48], configured with the refinements grammar (presented in Section 4.2). Refinements are parsed and stored as metadata of the annotated AST elements.

Because the Spoon AST is already enriched with direct links to declarations of variables, fields and methods, the global and local contexts are not explicitly represented in our implementation.

When traversing elements, it is often required to verify subtyping relationships. The subtyping assertion is first converted into a VC, which is discharged to an SMT Solver. In this implementation, we use Z3 [16], a SMT Solver created by Microsoft Research designed for an easy incorporation in software development, verification and analysis. However, our implementation is prepared to, in the future, use another SMT Solver or even let the developer decide which SMT Solver is better suited for their needs.

6.1.2 SMT-based Verification

In our implementation, verification conditions do not usually refer to all the variables in the context.

Our implementation filters only the variables in the VC that appear in the condition we are trying to prove, or that have a refinement. For instance, the example in the Socket class (Figure 5.8) can be flattened to use only one variable (Equation (6.1)).

$$\begin{aligned} \forall socket_4 : \text{Socket} \;.\; stateset_1(socket_4) == closed \Rightarrow \\ !(stateset_1(socket_4) == closed) \end{aligned} \tag{6.1}$$

VCs are then translated to the SMTLib language and verified using the SMT solver. For the translation to SMTLib, we use the Java API of Z3 through the Z3-TurnKey Distribution[1], since it eases the setup of Z3 into Java projects. After sending the verification conditions to the SMT Solver, the system waits for the result of the computation. If the result is *UNSAT* the verification continues, but if the result is *SAT*, the verification process ends (as explained in Section 5.2) and an error is displayed to the developer.

6.1.3 Error Messages

Whenever a refinement is not respected, a refinement type error message is sent to the user. The error message includes a brief description of the error and a detailed explanation of the verification performed. For the description, we only display an overall error message and the condition that was not possible to verify. As for the complete information, the explanation includes the location of the element that did not pass the verification,

[1] https://github.com/tudo-aqua/z3-turnkey

the expected type and the verification condition sent to the SMT Solver. However, the verification conditions might be hard to track without knowing where each local variable was created. Thus, we also introduced a translation table with the code that originated the local variable and its location.

An example of an error message displayed to the developer for the client code of the `ArrayDeque` class and specification is shown in Figure 6.2. In the error message is possible to see the two distinct parts divided by a horizontal bar (line 3). Above the bar is the summary of the error, and below is the detailed explanation. Within the detailed explanation, we present the code of the erroneous line followed by the expected type that was not possible to verify. Afterwards, we present the *State found* with the verification conditions used to verify the required condition, where we can see the distinct names for different local versions of `p`. Finally, to connect the instance names with the location where they were created, we present the translation table (from line 21 to 24).

However, the presented information might still not be the most useful for the developers, since some might not have the patience to understand the verification conditions or the translation table. Thus, one possible improvement could be the addition of a counter-example for the verification condition. The messages for other more general errors, such as syntax errors, could also be improved to provide small hints that would help the developer correct the code.

```
1 ArrayDeque<Integer> p = new ArrayDeque<>();
2 p.add(2);
3 p.size();
4 p.remove();
5 p.remove();
```

```
1 Refinement Type Error: Failed to check state transitions.
2 Expected possible states:(size(this) > 0)
3 ________________________________________________
4 Failed to check state transitions when calling remove() in:
5 p.remove()
6
7 Expected possible states:(size(this) > 0)
8
9 State found:
10 ––––––––––––––––––––––––––––––––––––––––
11 #p_27:ArrayDeque, (size(#p_27) == (size(#p_25) − 1)) =>
12 #p_25:ArrayDeque, (size(#p_25) == size(#p_24)) =>
13 #p_24:ArrayDeque, (size(#p_24) == (size(#p_22) + 1)) =>
14 #p_22:ArrayDeque, (size(#p_22) == 0)
15 ––––––––––––––––––––––––––––––––––––––––
16
17 Instance translation table:
18 ––––––––––––––––––––––––––––––––––––––––
19 | Variable Name | Created in | File
20 ––––––––––––––––––––––––––––––––––––––––
21 | #p_22 | ArrayDeque<Integer> p = new ArrayDeque<>() | Test2.java:1, 1
22 | #p_25 | p.size() | Test2.java:2, 1
23 | #p_24 | p.add(2) | Test2.java:3, 1
24 | #p_27 | p.remove() | Test2.java:4, 1
25 ––––––––––––––––––––––––––––––––––––––––
26 Location: (../Test2.java:5)
```

Figure 6.2: Client code of ArrayDeque with error and respectively error message.

6.2 Integration with IDE

Nowadays, most developers prefer to use integrated development environments (IDEs) for software development since they integrate services that enhance productivity, such as content completion, documentation popups, and real-time type checking [31]. IDEs can provide instant feedback to developers while they are implementing the code, and help them correct errors before executing the programs.

To enhance the usability of LiquidJava, we created an IDE plugin to allow the developers to use LiquidJava inside the development environment and use the verification information to modify the code while they are developing the program.

We used the Language Server Protocol (LSP) [64] to develop the plugin, since it decouples the implementation of the language server from the development of the editor's interface. By using LSP we only need to create one implementation of the LiquidJava language server, and afterwards it can be paired with a client for any editor that implements LSP (e.g., Visual Studio Code [65], Eclipse [23], Emacs [63]).

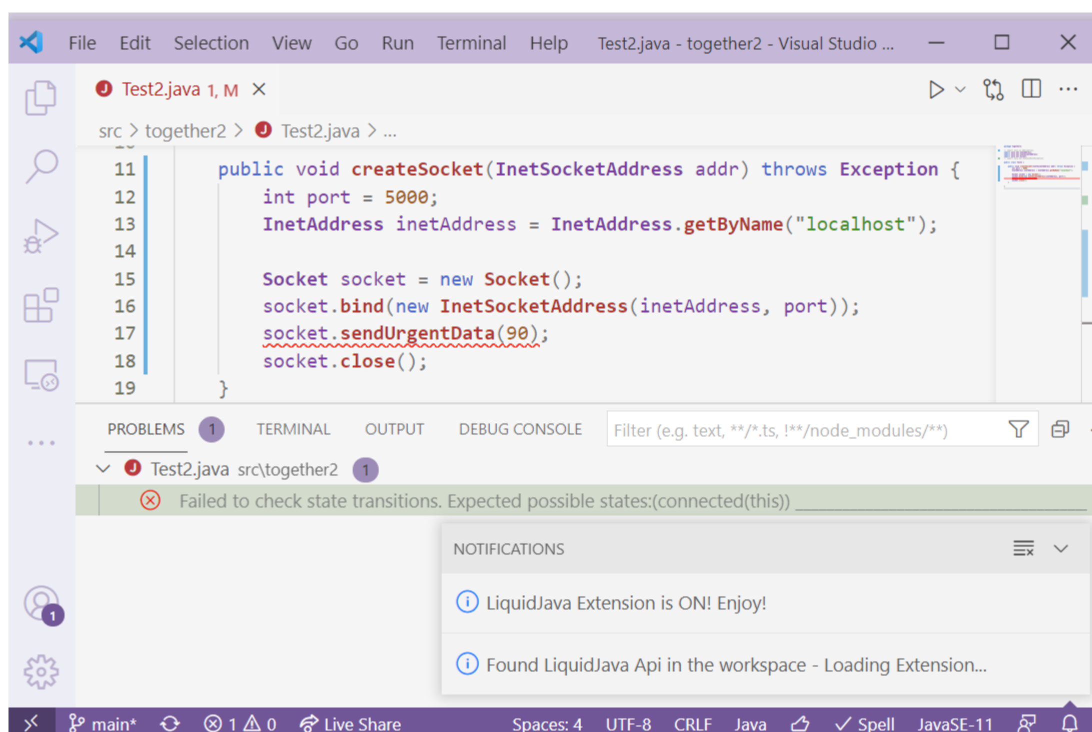


Figure 6.3: IDE Plugin reporting an error.

We chose to create the client plugin for Visual Studio Code, given that it is one most used IDEs for Java development [1, 19] and has more documentation on the interaction with LSP, since both are from Microsoft. However, as mentioned before, this does not prevent us from creating plugins for other editors in the future.

The main features of the plugin can be seen in Figure 6.3 and include:

- **Error Reporting** – The plugin informs the user, in real-time, of the refinement type

errors found, underlining the exact location of the code elements whose specification could not be proved;

- **Error Information** – A detailed version of the error can be found if the user hovers the code, getting all the verification conditions used to try to prove the specification and their location. To get an overview of all errors and warnings of the project, the developer can used the problems tab, where a simplified version of the error is presented.

The LiquidJava website[2] [26] contains the plugin available for download and examples for the usage of refinements in variables, methods and classes.

Language Server Protocol

The Language Server Protocol [64] aims to standardize the communication between language servers that provide language-specific information and the development tools. This way, language developers only need to create one instance of the language server, and the tools for development can integrate multiple languages with minimal effort.

The interaction between the language server and the client tool is accomplished using JSON-RPC since they run separate processes. Therefore, whenever the tool detects a user action (e.g., saves a file, opens a file), it notifies the language server, giving it a chance to publish information to the tool (e.g., location of definitions, autocomplete proposals).

Within the LiquidJava plugin, the connection to the language server is only started if the open project contains the *liquid-java-api.jar*; otherwise, a notification is given to the developer with the information that the plugin will not connect since it did not find the API. However, if the API is present in the project, the client tool will start the language server and connect to it. The LiquidJava language server receives events of change of Java files and verifies if they have refinement type errors. Thus, the server receives the changed documents, performs the type checking and, if it finds an error, creates a *Diagnostic* object (recognized by the protocol). The sent *Diagnostic* contains the diagnostic type (*Error* or *Warning*), its message and location (filename, line number and columns of start and finish). The list of *Diagnostics* is then published into the editor, which underlines the error or warning and provides the error message to the developer. In Visual Studio Code, the error message and overview is shown in the *Problems* tab, and the explanation of the verification is only available upon hovering over the underlined code.

Since the LiquidJava plugin only provides an extra verification for Java code, it must work in complement with an existing Java plugin. Thus, the plugin was created to complete the plugin Language Support for Java(TM) by Red Hat[3], one of the more popular

[2] https://catarinagamboa.github.io/liquidjava.html
[3] `https://marketplace.visualstudio.com/items?itemName=redhat.java`

plugins for Java development in VSCode, which already performs the common editor features such as type checking, syntax highlighting, code completion and others.

Current Limitations

The LiquidJava language plugin was developed as a prototype with the main intention of providing the refinement type errors to the developers. However, it is possible to integrate other information into the plugin, such as an autocomplete that includes defined class states or properties, or hover information that contains the refinement of the local variable at the desired location. Moreover, since this plugin was only a prototype, the verification is not as efficient as it could because each time a document changes, the liquid type checking is performed for all the project files and not only the changed part.

Chapter 7

Evaluation

To evaluate the usability of our approach, we conducted a user study with volunteer Java developers, to answer the following research questions:

RQ1 Are refinements easy to understand?

RQ2 Is it easier and faster to find implementation errors using LiquidJava than with plain Java?

RQ3 How hard is it to annotate a program with refinements?

RQ4 Are developers open to using LiquidJava in their projects?

With the intent to answer these questions, we designed an experiment to gather information on the usability and understanding of LiquidJava by asking the participants to complete a series of tasks related to the subject.

In this chapter, we present the experimental protocol (Section 7.1), a description of the participants and their background (Section 7.2), a detailed analysis of the tasks and their answers (Section 7.3 to Section 7.6) and, finally, we discuss the results of the study (Section 7.7).

The figures and listings referred during this chapter are in Appendix B due to space constraints.

7.1 Study Configuration

The intended audience for this study was developers who are familiar with Java, but not necessarily familiar with refinement types or Liquid Java. Section 7.2 will describe the participants and the process used to select them.

We conducted synchronous study sessions through the Zoom video platform, and gave the participants a survey with the study guidelines and answer placeholders, and a GitHub

repository with the study files [1]. During the study, the participants were required to install the LiquidJava plugin extension, and to this end, they needed to have the Visual Studio Code application installed with the jdk11 and the Language Support for Java(TM) by Red Hat extension enabled. The participants were also asked to share their screen with the VSCode editor and the document with the answers.

The study was divided into six parts, as follows:

- **Task 1: Find the error in plain Java** – Participants had to find and fix semantic errors in two Java programs, where the implementation did not correspond to the informal documentation presented in the associated Javadoc.
- **Task 2:Interpreting Refinements without prior explanation** – Participants had to interpret the refinements present in different sections of the code (variables, methods and classes) and use them correctly and incorrectly. The examples used and answers are detailed in Section 7.3. This task aim to answer **RQ1** by counting the correct uses of the refinements.
- **Overview of LiquidJava** – We introduced participants to LiquidJava using a 4-minute video and a webpage [26] explaining the examples of the previous task. Both resources were then available to be used within the remainder of the study. In the rest of the study, participants used LiquidJava through an IDE extension created for Visual Studio Code;
- **Task 3: Find the error with LiquidJava** – Similarly to Task 1, participants had to find and fix the incorrect behaviour of the programs. However, for this task, they were aided by the LiquidJava plugin. This task, paired with Task 1, intend to answer **RQ2**. The exercises were the same in both tasks, but they were split into two sets so that each participant could have different exercises in each task. Hence, half the participants had one set of exercises for Task 1 and a different one for Task 3, and the remaining half had the reverse set order. Therefore, the plain Java results serve as a baseline for the LiquidJava results.
- **Task 4: Annotate Java programs with LiquidJava** – Participants were asked to add LiquidJava annotations to three Java programs that targeted the LiquidJava features of refinements on variables, fields, methods and classes. This part addresses **RQ3**, and includes a final question about the difficulty of adding the annotations.
- **Final Comments** – Participants had the opportunity to express their thoughts on the overall experience of using LiquidJava by sharing what they most liked and disliked about the framework and whether they would use LiquidJava in their projects. This

[1] `https://github.com/CatarinaGamboa/liquidjava-examples`

overview aimed to answer the last research question, **RQ4**, and provide feedback to improve the project.

The examples used in the different tasks are detailed in Sections 7.3 to 7.6, along with the obtained answers and result discussion.

7.2 Participants

The study had 30 participants, the exact amount planned at the beginning of the study, and their background information is summarized in Figure B.1. The registration was disseminated through social media channels such as Twitter, Facebook and Instagram and also through personalized emails to Java developers.

The main condition to participate in the study was familiarity with Java, and more than 90% of the participants considered themselves *Familiar* or *Very Familiar* with Java. The remaining participants, who considered themselves only *Vaguely Familiar* with Java, were accepted in the study because they had already created test cases and used frameworks to test Java programs (such as JUnit). From all the participants, 80% were *Vaguely Familiar* or *Not Familiar* with refinement types which shows that despite their utility, refinement types are not widely known and used. The last background question asked if the participant had previously contacted with LiquidJava, and only 3 participants answered affirmatively, since they attended talks about the project but had never used it.

7.3 Interpreting Refinements without prior explanation

Since 90% of the participants had no previous contact with LiquidJava, and more than 80% were not familiar with refinement types, we wanted to understand if, without a prior explanation, the added specifications were intuitive to use. Thus, the study included a task with refinements examples that the participants needed to interpret and use in a correct and incorrect form. Specifically, we presented three code snippets with LiquidJava refinements with an increasing difficulty level (as showed in Listing B.1), and asked the participants to implement a correct and incorrect usage for each of the represented features.

In Task 1, the participants had to assign a correct and incorrect value to the variable x, which allowed a range of values representing the limits of the Earth surface temperature according to NASA. The second task asked of the participants to implement a correct and incorrect invocation of function1, where the second parameter depends on the first. Finally, the last task presented a class protocol with three possible states and methods that modelled the object state, and the participants were asked to create a *MyObj* object and implement a correct and incorrect sequence of at least three invocations. The *MyObj* class could represent a Vending Machine object with the three states *sX, sY* and *sZ* as *Show*

Items, Item Selected and *Paid*, respectively. The anonymization of the states and the class name were intentional to make the participants try to understand the refinements instead of calling the methods according to their mental idea of how a vending machine works.

Figure B.2 shows the evaluation of the answers given by the participants. Each answer was classified as *Correct* if both the correct and incorrect usage of the specification were correct, *Incorrect* if, at least, one of the usages was incorrect, or *Unanswered* if the answer was left blank. To the first question related to the variable value assignment, 86.7% of the participants answered correctly. The remaining 4 participants understood the error when the examples were explained and claimed that the error was a pure distraction and misread the logical operators. The invocation of the annotated method had only one incorrect answer (3.3%). In the last question, in which the class protocol was described using the `@StateRefinements`, 46.7% of the answers were correct, and the remaining amount was split into incorrect and blank answers, showing that this example is less intuitive and harder to understand without a prior explanation, but still not impossible to understand.

Overall, refinement annotations in variables and methods are intuitive and easy to understand. However, the annotations of classes and its methods with protocols are less intuitive and, in half of the cases, the participants would need a previous explanation to understand how this annotations could be used.

7.4 Using LiquidJava to Detect Bugs

This task aims to validate if using Java with Liquid Types makes it easier and faster to find implementation errors in LiquidJava, compared with plain Java. To this end, we chose four exercises with implementation errors that the participants had to find and fix, firstly only looking to the plain Java code (*Task 1: Find the error in plain Java*) and then with the help of LiquidJava and its plugin for VSCode (*Task 3 - Find the error in LiquidJava*). Between both tasks, the participants had a small introduction to LiquidJava with a short video, accessed the LiquidJava website and installed the LiquidJava plugin on their computer.

Each exercise had a plain Java version and a LiquidJava version with the same implementation errors to allow us to compare the number of participants that found and fixed the bug, as well as the time taken, in each version. Half the participants started with two exercises in *Task 1* and the other half used the same exercises with LiquidJava for *Task 3*. Therefore, one participant never used the same exercise in both tasks, avoiding tainting the second task with previous knowledge of the solution, and allowing us to obtain a plain Java baselines for every exercise. With this split, note that the maximum number of answers to each version is now 15 since only half the participants viewed each exercise version.

In both *Task 1* and *Task 3*, we gathered the time spent in each exercise and the given

answers. The answers were then evaluated into one of four possible categories: *Correct*, *Incorrect*, *Unanswered*, and *Compiler Correct*. The last category, *Compiler Correct*, represents the answers that, despite not having any error detected by the LiquidJava compiler, are not utterly correct according to the exercise.

The exercises and the results obtained are presented in Section 7.4.1 and the overall discussion is in Section 7.4.2.

7.4.1 Exercises and Answers

The four exercises were split into two groups of similar difficulty, and each group is used in a different task for each participant. The first exercises of each group represent problems with recursive methods and return values inside conditional branches. These exercises are incorrect versions of `fibonacci` (Section 7.4.1.1), and `sum` (Section 7.4.1.2). The second exercise of each group uses an external Java library whose methods contain an underlying protocol that should be followed but is only described using documentation. The client code of these exercises use the `java.util.ArrayDeque` class (Section 7.4.1.3) and `java.net.Socket` class (Section 7.4.1.4). The modelling of each of these external classes has already been presented in this document in Section 5.4.6.1 and Section 5.4.6.2, respectively. The remainder of this section presents each of the exercises, the data collected and a small discussion of the results.

7.4.1.1 Fibonacci Exercise

This exercise presents a recursive implementation of the method `fibonacci` that computes the nth Fibonacci number. However, the code contains an implementations error in the base case of the recursion, where the starting values of `F(0)= 1` and `F(1)= 1` are not respected. Listing B.2 shows the method with the LiquidJava refinement annotations according to the informal documentation. The plain Java version has the same Java code, but without the annotations of lines 1,2, 10 and 11. The refinements were introduced to the parameter and the method's return, according to the Javadoc. The aliases were introduced to give more meaningful names to the predicates and show another feature that can be used in LiquidJava.

The answers and the time spent on the exercise in the plain Java and LiquidJava versions are plotted in Figure B.3. We can see that all the participants could find and fix the exercise in LiquidJava, whereas in plain Java, only 73,3% found the error and 66,7% were able to fix it. The answers accepted as *Correct* changed the return of the base case to `return 1`, and all the different answers were determined as incorrect since they did not fix the presented error. Additionally, in plain Java, 2 participants in total decided to skip this question without answering the possible location of the bug or propose a fix.

The time spent on the Java version was, on average, smaller than the time spent in LiquidJava. The average time spent in the plain Java exercise was of 2 minutes and

52 seconds, while the time spent on LiquidJava was one of 3 minutes and 22 seconds, reaching a difference of only 30 seconds. This result suggests that, since Fibonacci is a popular exercise, participants are already used to its plain implementation, and when the new refinements were added, they spent more time on understanding the different sections of the code.

7.4.1.2 Sum Exercise

The `sum` exercise, presented in Listing B.3, contains a recursive method that should implement the sum of all numbers between 0 and the given parameter. However, it contains a bug in the base case since the method returns 0 if the gotten parameter is 1. This exercise was inspired by the recursion example presented in *Refinement Types: A Tutorial* [29].

Listing B.3 represents the LiquidJava version of the exercise, the plain version is, again, similar but without the annotations for the refinements (lines 1 and 6). The informal documentation does not specify any conditions to the parameter, leading to the omission of its refinement. However, it specifies the return conditions that can be simplified into the return refinement presented, using the same expression as in the refinement types tutorial.

The answers and the time spent in this exercise are plotted in Figure B.4. It is possible to see that in the plain Java version, only one participant was unable to find and fix the bug, while the 14 others were able to find the error, and 13 correctly fixed it. In the LiquidJava version, every participant was able to locate the error, but only 7 participants were able to fix it correctly, and the same number answered with *Compiler Correct* options.

The answers accepted as *Correct* adjusted the base case to one of the following versions: `if(n <= 0)` and `if(n < 1)`. The answers considered *Compiler Correct* are the ones that silenced the compiler error but are not correct according to the informal specification. Among these answers, 6 out of the 7 participants changed the base case's return value, changing it to `return 1`, while the other participant answered `return Math.abs(`
`↪ n)`.

The 46.7% of *Compiler Correct* answers suggests that the participants were more focused on silencing the compiler error than on correcting the program according to the informal specification. Additionally, we may relate the 40% of `return 1` answers with the Task 1 of these participants, which included the Fibonacci exercise where the correct fix was changing the return line to `return 1`. This line of thought might indicate that the participants were biased by the previous sections of the study and opted for the same answer as they used in the beginning.

The time plot shown in Figure B.4 contains two similar distributions, with a shorter average time in the LiquidJava version. In minutes, the participants spent 4 minutes and 30 seconds on average, finding and fixing the error in plain Java, whereas in LiquidJava, they spent an average of 3 minutes and 32 seconds, being faster by almost 1 minute.

7.4.1.3 ArrayDeque Exercise

This exercise presented a client code of the `ArrayDeque` class from the standard `java.` ↪ `util` library. The class contains popular methods to add, remove and get elements from an ArrayDeque object and these operations may depend on the number of elements present on the deque, and if these dependencies are not respected, then exceptions arise. Listing B.4 presents a client code that includes sequential invocations of methods on the `ArrayDeque`. However, the invocation `p.getLast()` on line 9 produces an exception since it is called when the object is empty.

The code presented in the LiquidJava version, includes the client of Listing B.4 and a separate file with the modelling of ArrayDeque with the ghost variable size as shown in Listing B.5 (with more methods than presented in Section 5.4.6.1).

Figure B.6 presents the results of the answers and the time spent in each version of the exercise. In this case, all the participants were able to correctly find and fix the error in both plain Java and in LiquidJava. There were multiple options to fix the code to prevent the raising of the exception; the accepted answers included removing `p.getLast()` on line 9, verifying if the queue is empty with `if(!p.isEmpty)` right before line 9, changing the line to `p.peekLast()`, among others.

The average time that participants took in this exercise in LiquidJava (2 minutes and 56 seconds) was 45 seconds less than using plain Java (3 minutes and 41 seconds).

7.4.1.4 Socket Exercise

The last exercise uses the `Socket` class from the external library `java.net`. The class was modelled as detailed before (Section 5.4.6.2) and a client method `createSocket(...)` was created with incorrect usage of the methods order (Listing B.6). The error lies in the invocation of `socket.sendUrgentData(90)`, that was made before the socket being connected to a server. The same client code was shown to participants in both Tasks 1 and 3. However, the code in Task 3 contained a file with the `Socket` class annotations of states and the allowed state transitions between the methods.

Figure B.7 shows the evaluation of the answers and the time spent in the Socket exercise. It is possible to see that only one participant was able to pinpoint the location of the error while using Java, and no participant was able to fix the error correctly. Additionally, we can also see that the percentage of blank answers was higher in this exercise than in all others, since 40% to 46% of the participants decided to move forward without answering.

However, in LiquidJava, all the participants were able to find the error, and 46,7% were able to fix the error correctly, while 53,3% silenced the error. The correct answer to fix the client code, given by 7 participants, relied on the addition of the line `socket.` ↪ `connect(addr);` between lines 8 and 9, where `addr` is the server address passed as an argument. The remaining eight answers also introduced the invocation to the connect method in the correct location but used the local address already used in the bind, which

would lead to a socket bound and connected to the same address and therefore produce an exception. Once again, the participants focused on silencing the error but did not try to understand the meaning of the provided answer.

Regarding the time spent on this exercise, in the Java version the participants spent an average of 5 minutes and 35 seconds, compared to the 4 minutes and 42 seconds spent in the LiquidJava version, showing that the participants were faster by 52 seconds in LiquidJava with a much higher rate of correct answers.

7.4.2 Discussion

With the results gathered from Task 1 and Task 3, we can conclude that LiquidJava constantly helped the participants find the bugs present in the code since the percentage of participants who found the bugs was always higher, or equal, in the LiquidJava version when compared to the plain Java version. LiquidJava also helped the participants to fix the bugs according to the error information provided. However, the participants focused on silencing the errors, which lead to some answers that were only considered *Compiler Correct* because the bug was not totally fixed.

The task of finding and fixing the bugs was faster in LiquidJava in all but one exercise. The latter refers to the Fibonacci exercise, which may have had a shorter time because of its popularity since most developers are familiar with its plain Java version. From all the exercises, the one that benefited the most from the LiquidJava version was the Socket exercise, where we moved from having only one participant finding the error in plain Java to 100% in LiquidJava, and from no participant fixing the bug to 46,7% fixing it and 53,3% silencing the error. This means that LiquidJava is more useful when used in more complex programs, with classes with protocols less known by developers.

Overall, LiquidJava helped the participants find and fix the bugs, and in some cases, it helped them do it faster.

7.5 Adding LiquidJava Annotations

In Task 4, participants were asked to add LiquidJava annotations to the implemented code according to the informal documentation written in the program as comments. In this step, participants could use the website and the video as help in writing the refinements.

Participants had to annotate programs with increasing order of difficulty. The first program relied only on the annotation of a variable with its bounds. The second program expected the annotation of a method by specifying the parameters and return refinements. Finally, the third program required the annotation of a class protocol and the class fields. We presented an example of a correct usage of the refinement and another example for its incorrect usage for each program to help the developers test their refinements.

The participants shared their proposals for the annotation of each exercise, and we evaluated them with the four categories used in the previous section, of *Correct*, *Incorrect*, *Unanswered* and *Compiler Correct*. The results of the annotations are in Figure B.8 and will be analysed along with each exercise below.

The first and more straightforward program is presented in Listing B.7 and contained the simple task of restricting the value of the variable with an upper and lower bound, which could be accomplished with the annotation: `@Refinement("currentMonth >= 1 ↪ && currentMonth <= 12 ")`. Already in the first exercise, all the participants used the website as a resource to look for the right syntax to use, and 100% of them annotated the variable correctly.

Listing B.8 shows the method presented in the second exercise where participants should add a refinement to the second parameter, changing the signature of the method to `public static int inRange(int a, @Refinement("b > a")int b)`, and refine the return type of the method, adding the refinement `@Refinement("_ >= a && _ <= b")` above the method.

Twenty-four of all participants were able to add the expected annotations leading to 80% of `Correct` answers. However, 20% only added the annotations to the parameter, silencing the example error but not completing the exercise in its totality, and leading to 6 *Compiler Correct* answers.

The last exercise asked the participants to annotate the class TrafficLight (Listing B.9), which uses RGB values (between 0 and 255) to define the light colour and follows the protocol defined by Figure B.9. The evaluation of this annotation was split into the addition of the refinements to model the class protocol and the specification on the class private fields.

100% of the participants were able to correctly model the class by declaring the starting states and the state transitions allowed in each method. This constitutes a significant increase in the understanding of the class protocols when compared to the first time the participants tried to understand the protocol in Task 1, where half of the participants could not interpret the class protocol.

However, only 43,3% of the participants annotated the class fields, and the remaining participants did not add any refinement, leaving an incorrect assignment in the class implementation. The participants who did not annotate the class fields probably misinterpreted the exercise, not realising the need for these annotations.

After finishing introducing the annotations, the participants had to evaluate the ease of the addition of the annotations from *0 - Very Difficult* to *5 - Very Easy* and the results of the answers are presented in Figure B.10. All the participants considered that adding the annotations was easy, and 60% even considered that it was very easy, concluding that the refinements are simple to add to implemented code.

7.6 Final Overview

At the end of the tasks we asked the participants about the overall experience of using LiquidJava using three questions:

- What did you enjoy the most while using LiquidJava?
- What did you dislike the most while using LiquidJava?
- Would you use LiquidJava in your projects?

The first two questions were open-ended, and the different answers focused on diverse aspects, as expected. To process the answers and capture their essence, we used a qualitative coding approach [54]. We started with the set of answers to each question, and we used inductive coding to create the codes. With these codes, we reviewed all passages and identified the main topics used within the answers, leading to a cohesive and systematic view of the results.

The topics that the participants enjoyed about LiquidJava are presented in Figure B.11. The participants mostly enjoyed the error reporting on the incorrect code, the use of state refinements to model objects and the intuitive and non-intrusive syntax.

1. **Error Reporting**:"[...] the fact it reported un-compliances with the specifications"; "[with] refinement types we have an assurance that the program is logically consistent at the time of coding."; "Helps to define the program's logic and avoid future errors";
2. **State Refinements**: "the state refinement it is very useful"; "Specification of states, because enforcing correct state transitions in the implementations is tedious and error-prone".
3. **Syntax**: "The simple syntax based on annotations", "Very intuitive syntax", "[the refinements addition is] non-intrusive";
4. **Plugin**: "Excellent integration with VSCode", "Very well assisted by the custom vs code extension";
5. **Understandability**: "Very intuitive", "Easy to understand in terms of semantics";
6. **Useful**: "Highly useful", "It helped me a lot to known which values/methods are correct when I'm coding";
7. **Resources**: "Using the video and the examples it was easy to understand how the refinements work", "The examples and the video were very illucidative";

8. **Flexibility**: "I liked the flexibility of scenarios in which I could use LiquidJava", "+1 to the diverse set of refinements that can be written in different parts of the code".

All 30 participants answered the first question; however, only 26 answered the second question regarding what they disliked the most about LiquidJava. The topics pointed by the participants are represented in Figure B.12 and include eight answers that only stated that there was nothing they disliked about LiquidJava. The remaining points they disliked about LiquidJava involve the syntax of certain elements inside the refinements and some plugin features. We present below a sample of the answers given for each topic:

1. **Syntax**: "[in the state refinement] repetition of this", "the usage of _ for the return value";
2. **Plugin features**: "Improve the usability of the plugin -remove double quotes; - use auto-complete inside the refinements", "not being able to correct multiple files at the same time";
3. **Not Intuitive**: "Hard to understand without access to documentation, mainly DFA protocols";
4. **Error Messages**: "Some error messages are not straight to the point";
5. **Verbose**: "Makes Java more verbose";
6. **Other**: "The state transition is a bit harder to get but paired with the given documentation is fine.","The installation process was not very user friendly".

The last question of the study asked the participants if they would use LiquidJava in their projects, to which all the participants answered affirmatively, as can be seen in Figure B.13. However, in the final suggestions, one participant declared that he would only use it in critical parts of the project, and other two participants referred that they are not currently using Java in any project but would like to have similar verifications in other programming languages.

7.7 Study Conclusions

The research study focused on the usability of LiquidJava to verify Java programs. Thus, it included tasks related to the understandability of LiquidJava, the usage of LiquidJava to find and fix implementations bugs and the addition of LiquidJava annotations in Java code in order to add a new layer of verification. This study had four main research questions, and the six parts of the study aimed to answer them.

The first research question (**RQ1**) inquired if refinements are easy to understand, and Task 2 (Section 7.1) and 4 (Section 7.1) helped answer this question. From the interpretation of refinements before an overview, we can assess that refinements on variables and methods are intuitive and easy to understand at first sight, whereas the refinements to model class protocols are more challenging to understand without a prior explanation (Section 7.3). However, after a short video (4-minutes) and having access to a webpage with examples, the participants were able to use and add LiquidJava annotations in the code correctly. Moreover, the class protocol, which was less intuitive at the start, had 100% correct answers when the participants were asked to model the object state in Task 4 (Section 7.5). Therefore, we can conclude that refinements in variables and methods are easy to understand without a prior explanation, and even though the features to model classes are not very intuitive at first, they are easy to understand with few resources.

RQ2 aimed to identify if it is easier and faster to find implementation errors with LiquidJava when compared with plain Java. The two main tasks that answer this question are Task 1 (Section 7.1) and 3 (Section 7.1), where participants tried to find and fix implementation errors in incorrect Java and LiquidJava code. The results from these tasks show that LiquidJava helps developers find implementation bugs, since all participants had a higher or equal rate of correct answers while using LiquidJava. As for fixing the bugs, LiquidJava helped in all but one case, since developers focused on silencing compiler errors disregarding the reasoning behind the changes applied. As for the time taken in each exercise, participants, in general, finished the LiquidJava exercises faster. The one outlier to this, was the fibonacci exercise, probably because it is a traditional algorithm that developers are used to see in its plain form and not with the annotations.

Overall, the exercise where LiquidJava helped developers the most was the `Socket` client, where no participant was able to fix the error using plain Java, but with LiquidJava 46% could fix it correctly, and the remaining ones found the error and silenced it. Therefore, to answer **RQ2**, LiquidJava helps identify the error location and to fix the errors, but it seems more useful when applied to lesser-known classes and protocols than to mainstream classes or simple code.

The third research question (**RQ3**) asked if it is hard to annotate a program with refinements. The fourth study task answers this question by asking developers to annotate code with refinements and then inquiring their opinion on the ease of the task (Section 7.5). For the first part, the results to the task of adding refinements show that all participants were able to add refinements to variables and to model class protocols, 80% were able to add refinements to methods correctly (the remaining silenced the errors) and 43% were able to introduce refinements in class fields (the remaining participants left the answer blank). Therefore, participants were able to understand the concept of refinement and correctly introduce them into the code. Moreover, the opinion of developers shows that they considered the task Easy or even Very Easy. Thus, we can conclude that refinements are easy

to add to the code to model the desired behaviour of programs.

Finally, the last research question (**RQ4**) asked if developers are open to using LiquidJava in their projects. The fact that we got all the desired participants to the study, already helps answering this question, since it shows that Java developers are open to participate in studies to discover new approaches to improve their code quality. However, the last question of the study is the one that gives us confidence that participants are open to using LiquidJava since we asked them if they would use LiquidJava in their projects, and all participants answered affirmatively. Therefore, we are confident that participants find LiquidJava accessible for its gains and are ready to use new useful verification tools.

Chapter 8

Future Work

This chapter presents the remaining work on the thesis subject and future directions for continuing the work.

Support for loops and aliasing

The type system for LiquidJava, presented in Chapter 5, only covers a subset of the Java language. In order to support verification of real-world Java programs, we will need to expand support for *while* and *for* loops, possibly using loop invariants (à lá JML[36]). Previous language extensions with refinement types, for example for C[52] or TypeScript [62], implement loop invariants to verify code with loops. However, one of the main challenges in this work is that Java does not support annotations on loops, only on classes, fields, methods, parameters and local variables.

In order to support complex data structures (and eventually concurrent programs), it is important to keep track of aliasing, especially in the verification of object state. A possible solution for this is to implement permissions and ownership as uninterpreted functions, similarly to Plaid [4] and ConSORT [59]. This approach would keep track of aliasing and display errors when the permission assumptions are violated, similarly to how the Rust [1] borrow checker works.

A complete formalization of LiquidJava type checking

During this work, we presented the type system rules for all LiquidJava features except the override methods from subclasses. Therefore, we intend to introduce the remaining type rules but also formalize all the language. For the latter, we aim to formalize the addition of refinements into a smaller subset of the Java language, the Featherweight Java [28], to expedite the proofs of type preservation and progress.

[1] https://www.rust-lang.org/

Improved Error Messages

Writing useful and clear compiler error messages is a challenge, shared by all compilers and programming languages. Advanced type systems, with dependent and liquid types in particular, are even more challenging because of the many context dependencies in the types.

During the research study, participants found the error location very helpful, but none of them spent more than 1 minute trying to understand the error message. This shows that, despite our effort to provide helpful information for the cause of the error (Section 6.1.3), the error messages are still not straightforward or user-friendly. Thus, we aim to improve the error messages provided by LiquidJava and explore how they can be more helpful for the users.

In LiquidJava, there are two types of errors: those that are in the refinements language (inside the annotations) and those that are in the Java code that arise from using variables and methods with the wrong type.

For the first ones, it is possible to improve the error messages by giving hints (e.g., "inside refinements use == instead of =", "alias RBG not found, maybe you meant RGB") and better textual explanations (e.g., "state *yellow* is not defined in *TrafficLight* class" instead of "function yellow not found").

As for the type errors raised by the verification process, one possible improvement is the addition of a counter-example that shows why the verification could not be performed. However, the quantity of information for one error would increase, which would probably not be beneficial for the developer. Thus, in symbioses with the editor plugin, splitting the error information into collapsed sections that the user could expand as desired could be interesting.

Improve IDE integration

As introduced in Section 6.2, the IDE integration only implemented the presentation of errors reported by the LiquidJava verifier. However, LSP allows the implementation of other interesting features for the plugin, such as autocomplete using context information, code highlighting of the elements used to create a verification condition, provide local refinements on hovering a code element, among others. Furthermore, it is possible to create personalized views on the editor that provoke custom events that are sent to the language server. With this feature, it is possible, for example, to create a side menu for LiquidJava where the errors are presented with a tree view where the error sections have a main description and can be expanded to show the detailed information, similarly to view panels implemented for other dependently type languages such as Lean [35] or Coq [5]. Within a custom view, it would also be interesting to allow the users to see a graph representation of the object state modelling, translating the method transitions to a

representation of a state machine.

Besides the features that could be introduced to the plugin, it is essential to minimise the time and RAM consumption during the verification. The current implementation of the verification is unfeasible when the project size increases since every time a change occurs in the document, the verification of the complete project is performed. Thus, it is important to enhance the performance of the plugin by only verifying the changed code instead of the overall project.

In the future, we would also like to create the client-side of other tools used for Java development, such as Eclipse or IntelliJ, since both can communicate via LSP and are popular amongst Java users.

Refinements Inference

Despite the research study results stating that the participants found it easy or very easy to introduce the refinements in the code, it is still an additional task for developers. Thus, we should try to minimize the effort of the developers in the annotation process as much as possible by, for example, infering the refinements. To this end, we could implement the Liquid Type inference algorithm presented in previous studies [53] and translate the informal documentation into refinements with the help of existing frameworks such as Toradocu [8], Daikon[20] or EvoSpex [43].

Chapter 9

Conclusion

Type systems are one of the most popular software verification techniques to establish guarantees regarding the behaviour of code. This popularity comes from the thin barrier between verification and code development due to types being integrated into the programming language, and type errors typically being shown at compile time, before the program execution. However, a correctly typed program can still have multiple errors that could be caught with stronger type systems, like refinement types. Refinement types have been introduced in functional programming languages (e.g., Haskell and ML) but they are yet to become popular in the mainstream developer community, despite their perceived utility.

In this work, we introduced refinement types to the Java programming language, with the goal to promote the wide usage of refinement types as a software verification technique. To integrate refinements into Java, we proposed the addition of `@Refinement` annotations with a new language for the refinements. The refinements language includes features to model variables, methods, class fields and class states.

To reduce the cognitive load of the syntax for the refinements language, we proposed two to three syntax options similar to Java for each feature we intended to support and created an online survey to assess the preferable syntax among Java developers. The syntax for refinements in variables, methods and class fields was decided using the answers on the syntax preference of 50 java developers that participated in the survey.

We proposed a novel specification to model class state with refinements that allow developers to specify multiple state sets and combine states with ghost properties. This specification can enforce protocols in classes and successfully model state machines in Java classes, like in the `java.net.Socket` class.

To verify the refinements in the Java code, we created the type checking rules for this extension and translated the subtyping relationships into verification conditions which are automatically verified using an SMT-Solver. If any of the verification conditions is not provable, subtyping fails, and a refinement type error is displayed to the user.

We implemented these rules in LiquidJava, an extension of Java with Liquid Types,

and integrated it into a plugin for Visual Studio code editor to improve the usability of the system. Therefore, developers get the error reporting in real-time with the errors underlined and accompanied with error messages while developing their programs.

To evaluate LiquidJava, we developed a research study focused on the usability of LiquidJava as a software verification tool. Hence, we conducted the study with 30 participants familiar with Java and asked them to perform tasks related to the interpretability of refinements, the use of LiquidJava to find and fix errors, and the addition of refinements into Java code. The study showed that refinements in variables and methods are very intuitive since more than 86% of the participants could use the refinements of these features without a prior introduction to refinement types. Although refinements in classes were more difficult to understand without a prior explanation (only 46% of participants used them correctly), they became easy to use after a 4-minute video and access to a website with examples, since 100% of participants annotated the protocol refinements correctly. Furthermore, the exercise with best results in detecting errors in LiquidJava when compared to Java used the protocol of the class `java.net.Socket`, showing that LiquidJava might be more useful when applied to lesser-known classes and protocols, reducing the time spent on error localization. Finally, the study showed that participants found it easy to annotate Java programs with refinements and all of them declared that they would use LiquidJava in their projects.

It is expected that, from now on, LiquidJava can evolve towards a feature-complete verification of the Java language, with enhanced usability inside editors and more insightful error messages. Thus, leading to the use of LiquidJava in critical software products and general scope projects to improve code quality.

Appendix A

Liquid Type Checking Rules

$$\frac{\Gamma, \{x : B, e\}; \Delta \vdash S\ valid}{\Gamma; \Delta \vdash \text{@Refinement}(e)\ B\ x;\ S\ valid}\ \textbf{(var-decl)}$$

$$\frac{x : T \in \Gamma \quad \Gamma; \Delta \vdash e : K \quad \Gamma; \Delta \vdash K <: T \quad \Gamma; \Delta, x : K \vdash S\ valid}{\Gamma; \Delta \vdash x = e;\ S\ valid}\ \textbf{(var-assign)}$$

$$\frac{x : T \in \Delta}{\Gamma; \Delta \vdash x : T}\ \textbf{(var)}$$

$$\frac{\Gamma \vdash e : bool \quad \Gamma, \{\alpha : int|e\} \vdash S_1\ valid \quad \Gamma \vdash S_2\ valid}{\Gamma; \Delta \vdash (\text{ if } (e)\{S_1\}\ S_2\ )\ valid}\ \textbf{(if-then)}$$

$$\frac{\Gamma \vdash e : bool \quad \Gamma, \{\alpha : int|e\} \vdash S_1\ valid \quad \Gamma, \{\beta : int|\neg e\} \vdash S_2\ valid \quad \Gamma \vdash S_3\ valid}{\Gamma; \Delta \vdash (\text{ if } (e)\{S_1\} \text{ else } \{S_2\}\ S_3\ )\ valid}\ \textbf{(ite)}$$

$$\frac{\Gamma, \{f : \{\overline{x_j : T_j|e_j}\} \xrightarrow{a,b*} \{v : T|e\}\}, \{expret : T|e\} \vdash S,\ S_1\ valid}{\begin{array}{ll}\Gamma; \Delta \vdash & \text{@StateRefinement(from=}e_{f1},\ \text{to=}e_{t1}) \\ & \ldots \\ & \text{@StateRefinement(from=}e_{fn},\ \text{to=}e_{tn}) \\ & \text{@Refinement}(e) \text{ public } T\ f\ (\text{@Refinement}(\overline{e_j})\ \overline{T_j\ x_j})\ \{S_1\}\ S_2\ valid\end{array}}\ \textbf{(met-decl)}$$

$$*(a = \textstyle\bigvee_{i=1}^{n} e_{fi}; b = \bigvee_{i=1}^{n} e_{fi} \rightarrow e_{ti})$$

$$\frac{\Gamma \vdash e :\ T \quad expret : U \vdash \Gamma \quad \Gamma \vdash T <: U \quad \Gamma \vdash S\ valid;}{\Gamma; \vdash return\ e;\ S\ valid}\ \textbf{(return)}$$

$$\frac{\begin{array}{ccc} \Gamma \vdash y_i : T_i & \Gamma; \Delta \vdash obj : U & \Gamma \vdash U <: \{\beta : U | a\} \\ f : \{\overline{x_i : T_i}\} \xrightarrow{a,b} \{v : T \mid e\} \in \Gamma & & \Gamma; \Delta, obj[a \to b] \vdash S\ valid \end{array}}{\Gamma; \Delta \vdash obj.f\ (\overline{y_i}) : \{T | e[x_i := y_i]\}\ S\ valid} \textbf{(met-inv)}$$

Appendix B

Figures and Listings from Evaluation

```
//1 – Variable Refinement
@Refinement("−25 <= x && x <= 45")
int r;

//2 – Function/Method Refinement
@Refinement("_ >= 0")
public static double function1(@Refinement("a >= 0") double a,
                               @Refinement("b >= a") double b){
    return (a + b)/2;
}

//3 – Class Protocol Refinement
@StateSet({"sX", "sY", "sZ"})
public class MyObj {

    @StateRefinement(to="sY(this)")
    public MyObj() {}

    @StateRefinement(from="sY(this)", to="sX(this)")
    public void select(int number) {}

    @StateRefinement(from="sX(this)", to="sZ(this)")
    public void pay(int account) {}

    @StateRefinement(from="sY(this)", to="sX(this)")
    @StateRefinement(from="sZ(this)", to="sX(this)")
    public void show() {}
}
```

Listing B.1: Variable refinement in LiquidJava and verification of its assignments.

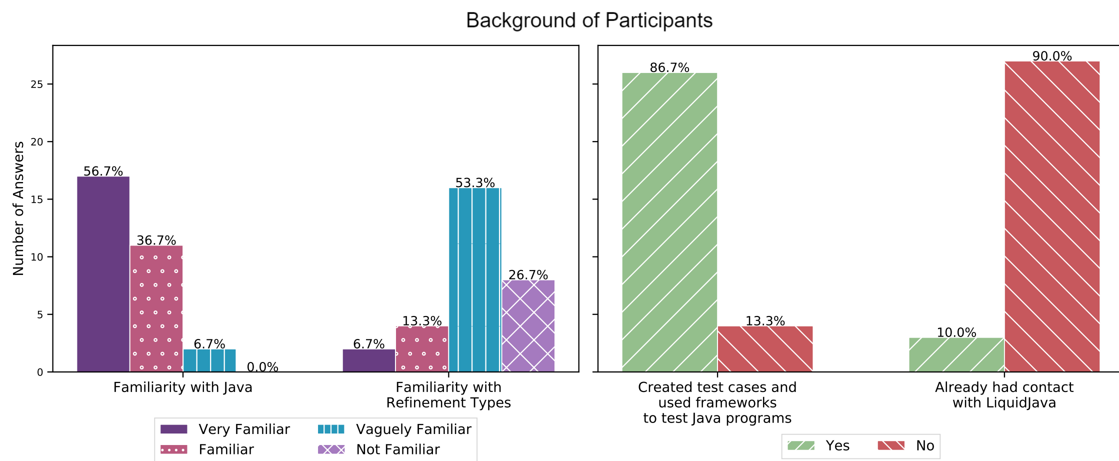


Figure B.1: Background information of the 30 participants selected to participate in the study.

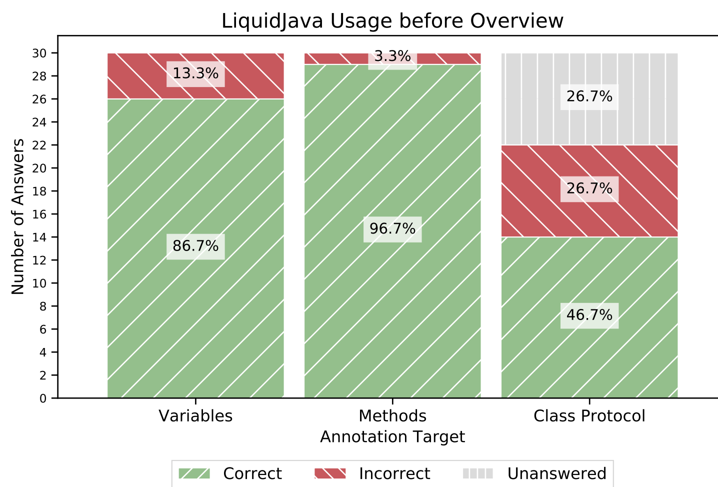
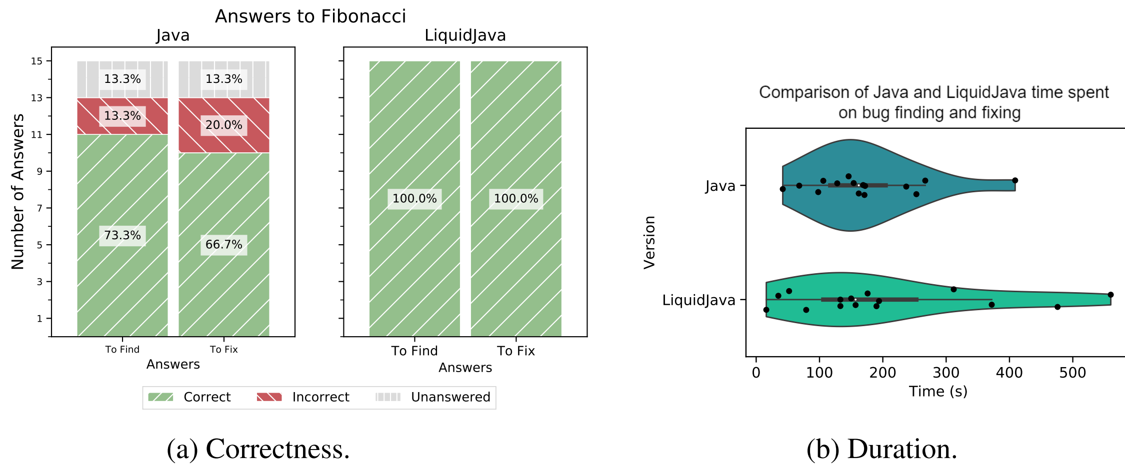


Figure B.2: Answers on interpreting LiquidJava refinements.



(a) Correctness. (b) Duration.

Figure B.3: Answers correctness and duration of the Fibonacci exercise, in both versions.

```
1  @RefinementAlias("Nat(int x) {x >= 0}")
2  @RefinementAlias("GreaterEqualThan(int x, int y) {x >= y}")
3  public class Test1 {
4      /**
5      * Computes the fibonacci of index n
6      * @param n The index of the required fibonacci number (greater or equal to 0)
7      * @return The fibonacci nth number. The fibonacci sequence follows the formula
8              Fn = Fn−1 + Fn−2 and has the starting values of F0 = 1 and F1 = 1
9      */
10     @Refinement( "_ >= 0 && GreaterEqualThan(_, n)")
11     public static int fibonacci(@Refinement("Nat(n)") int n){
12         if(n <= 1)
13             return 0;
14         else
15             return fibonacci(n−1) + fibonacci(n−2);
16     }
17 }
```

Listing B.2: Fibonacci in LiquidJava.

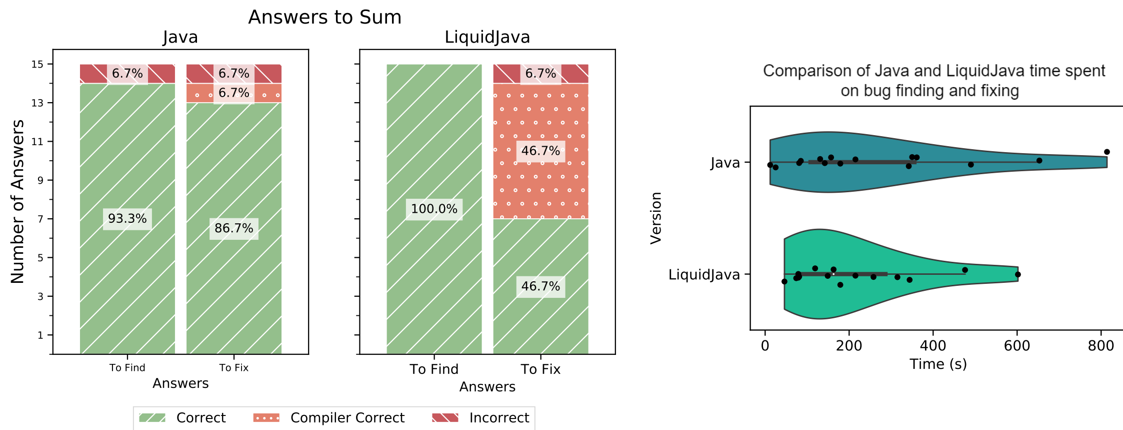


Figure B.4: Answers and time spent on the Sum exercise, in both versions.

```
1  @RefinementAlias("Nat(int x) {x >= 0}")
2  public class Test1 {
3      /** The sum of all numbers between 0 and n
4      * @param n
5      * @return a positive value that represents the sum of all numbers between 0 and n, or 0 if
           ↪ n is negative */
6      @Refinement("Nat(_) && _ >= n")
7      public static int sum(int n) {
8          if(n <= 1)
9              return 0;
10         else {
11             int t1 = sum(n−1);
12             return n + t1;
13         }
14     }
15 }
```

Listing B.3: Sum exercise in LiquidJava.

```
1  //Exercise 2
2  public class Test2 {
3    public void createSocket(InetSocketAddress addr) throws IOException{
4          int port = 5000;
5          InetAddress inetAddress = InetAddress.getByName("localhost");
6
7          Socket socket = new Socket();
8          socket.bind(new InetSocketAddress(inetAddress, port));
9          socket.sendUrgentData(90);
10         socket.close();
11     }
12 }
```

Listing B.6: Client code of Socket class.

```java
public class Variable {
  public static void main(String[] args) {
    /* A month needs to have a value between 1 and 12*/
    int currentMonth;

    currentMonth = 13; //Error
    currentMonth = 5; //Correct
  }
}
```

Listing B.7: Variable to be annotated with LiquidJava and two assignments to test the refinement.

```java
public class Method {
  /**
  * Returns a value within the range
  * @param a The minimum border
  * @param b The maximum border, greater than a
  * @return A value in the interval [a, b] (including the border values)
  */
  public static int inRange(int a, int b){
      return a + 1;
  }

  public static void main(String[] args) {
      inRange(10, 11); //Correct
      inRange(10, 9); //Error
  }
}
```

Listing B.8: Method to be annotated with LiquidJava and two invocations to test the refinements.

```
1  public class TrafficLight {
2
3    private int r;
4    private int g;
5    private int b;
6
7    public TrafficLight() {
8      r = 76; g = 187; b = 23;
9    }
10
11   public void transitionToGreen() {
12     r = 76; g = 187; b = 23;
13   }
14
15   public void transitionToAmber() {
16     r = 255; g = 120; b = 0;
17   }
18
19   public void transitionToRed() {
20     r = 230; g = 0; b = −1;
21   }
22 }
23 //Correct Test − different file
24 TrafficLight tl = new TrafficLight();
25 tl.transitionToAmber();
26 tl.transitionToRed();
27 tl.transitionToGreen();
28 tl.transitionToAmber();
29
30 //Incorrect Test − different file
31 TrafficLight tl = new TrafficLight();
32 tl.transitionToAmber();
33 tl.transitionToRed();
34 tl.transitionToAmber();
35 tl.transitionToGreen();
```

Listing B.9: Java class to introduce class refinements that follow a protocol.

```
1  public class Test2 {
2      public static void main(String[] args) throws IOException{
3          ArrayDeque<Integer> p = new ArrayDeque<>();
4          p.add(2);
5          p.remove();
6          p.offerFirst(6);
7          p.getLast();
8          p.remove();
9          p.getLast();
10         p.add(78);
11         p.add(8);
12         p.getFirst();
13 }
14 }
```

Listing B.4: Client code that uses the ArrayDeque class.

```
1  @ExternalRefinementsFor("java.util.ArrayDeque")
2  @Ghost("int size")
3  public interface ArrayDequeRefinements<E> {
4
5      public void ArrayDeque();
6
7      @StateRefinement(to="size(this) == (size(old(this)) + 1)")
8      public boolean add(E elem);
9
10     @StateRefinement(to="size(this) == (size(old(this)) + 1)")
11     public boolean offerFirst(E elem);
12
13     @StateRefinement(from="size(this) > 0", to = "size(this) == (size(old(this)))")
14     public E getFirst();
15
16     @StateRefinement(from="size(this) > 0", to = "size(this) == (size(old(this)))")
17     public E getLast();
18
19     @StateRefinement(from="size(this)> 0", to="size(this) == (size(old(this)) − 1)")
20     public void remove();
21
22     @Refinement("_ == size(this)")
23     public int size();
24
25 }
```

Listing B.5: Refinements to model size of ArrayDeque.

Figure B.5: ArrayDeque Exercise.

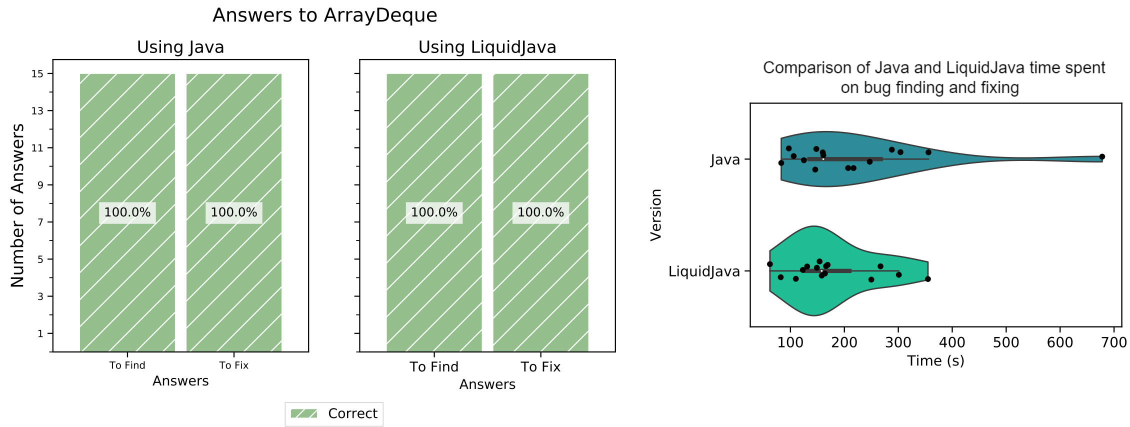


Figure B.6: Answers and time spent on the ArrayDeque exercise, in both versions.

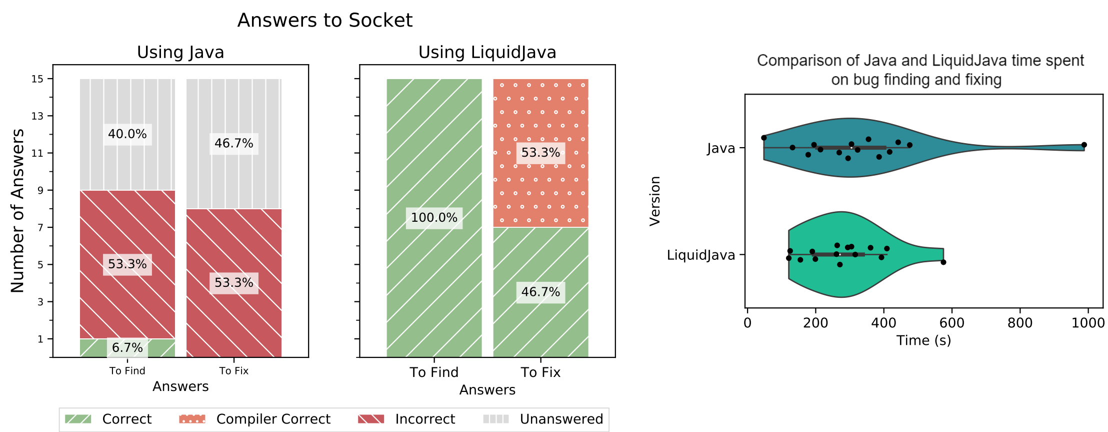


Figure B.7: Answers and time spent on the Socket exercise, in both versions.

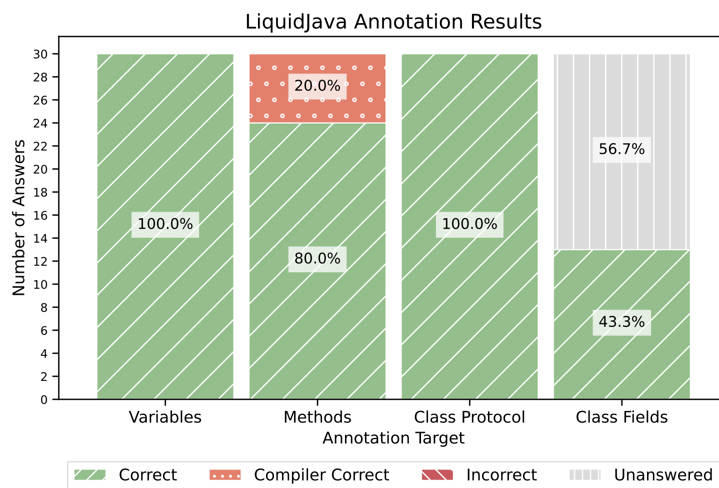


Figure B.8: Results of the annotations with LiquidJava.

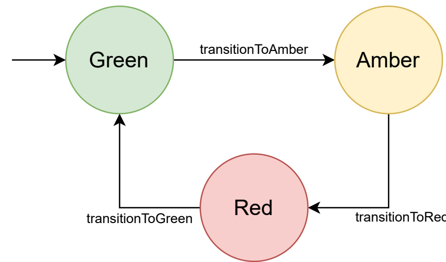


Figure B.9: Protocol of the TrafficLight that must be followed to the annotation.

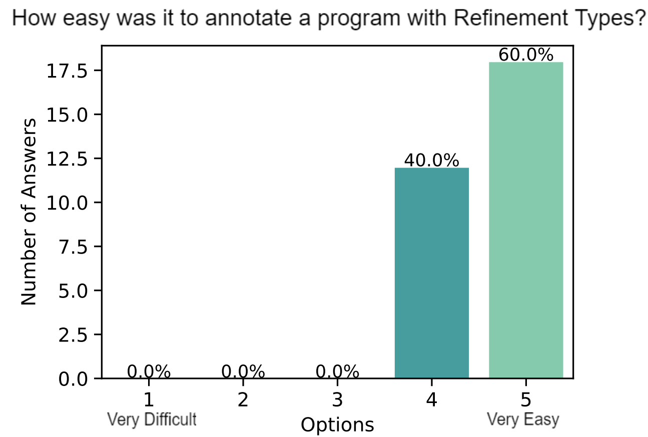


Figure B.10: Participants' answers to the ease of writing the specification with LiquidJava.

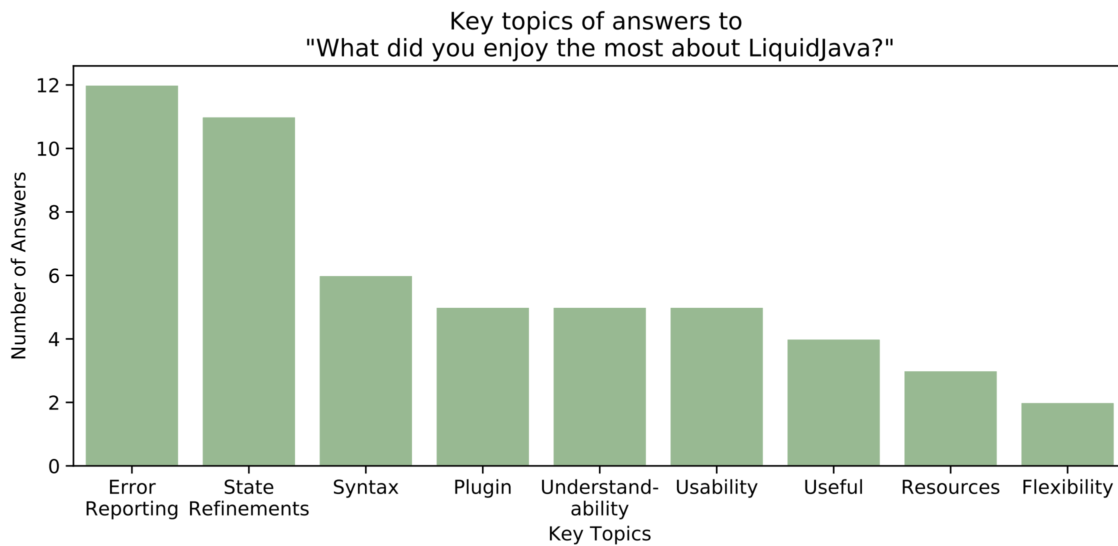


Figure B.11: Main subjects that the participants enjoyed to use during the study.

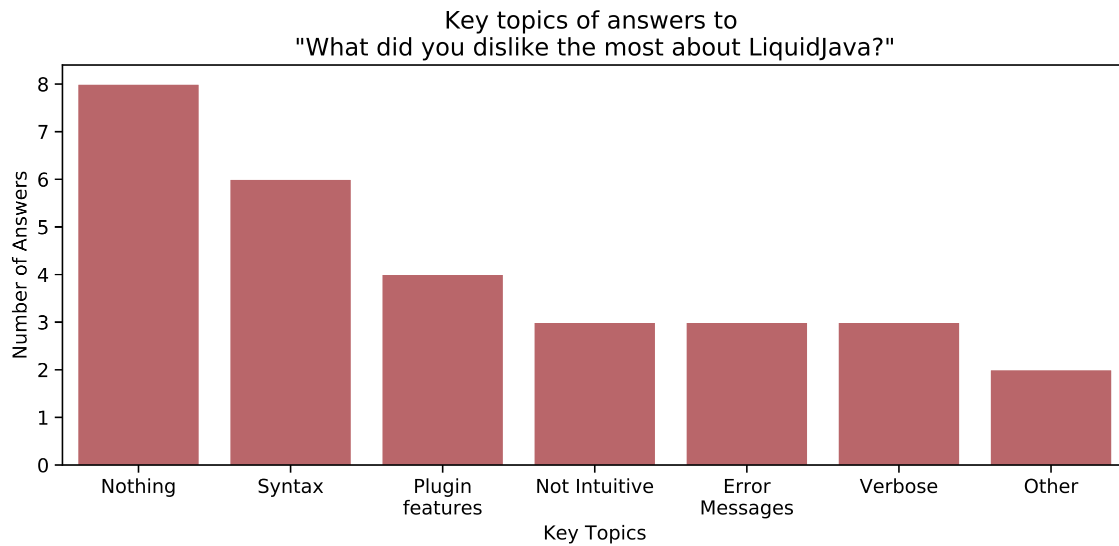


Figure B.12: Main subjects that the participants disliked during the study.

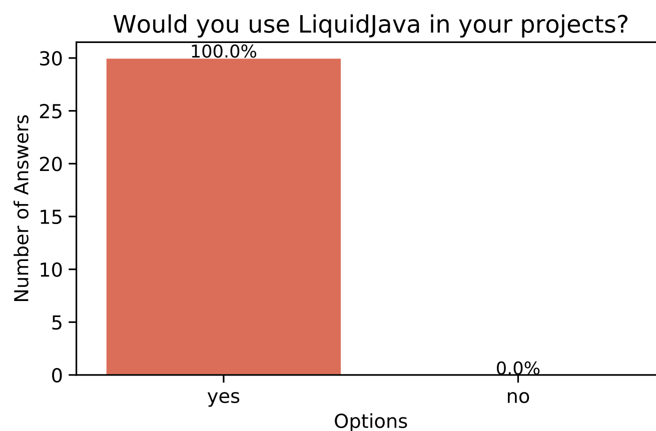


Figure B.13: All the participants stated that they would use LiquidJava in their projects.

Bibliography

[1] JRebel Report 2021. 2021 java technology report. https://www.jrebel.com/blog/2021-java-technology-report.

[2] Majid Aghaei. An experimental evaluation of java design-by-contract extensions. http://jultika.oulu.fi/files/nbnfioulu-201812063243.pdf.

[3] Kübra AKSOY, Sofiène TAHAR, and Yusuf ZEREN. Introduction to hol4 theorem prover. http://eds.yildiz.edu.tr/AjaxTool/GetArticleByPublishedArticleId?PublishedArticleId=3936.

[4] Jonathan Aldrich, Joshua Sunshine, Darpan Saini, and Zachary Sparks. Typestate-oriented programming. In Shail Arora and Gary T. Leavens, editors, *Companion to the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2009, October 25-29, 2009, Orlando, Florida, USA*, pages 1015–1022. ACM, 2009.

[5] Emilio Jesús Gallego Arias, Benoît Pin, and Pierre Jouvelot. jscoq: Towards hybrid theorem proving interfaces. In Serge Autexier and Pedro Quaresma, editors, *Proceedings of the 12th Workshop on User Interfaces for Theorem Provers, UITP 2016, Coimbra, Portugal, 2nd July 2016*, volume 239 of *EPTCS*, pages 15–27, 2016.

[6] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The SMT-LIB Standard: Version 2.6. Technical report, Department of Computer Science, The University of Iowa, 2017. Available at www.SMT-LIB.org.

[7] Kevin Bierhoff and Jonathan Aldrich. Lightweight object specification with typestates. *SIGSOFT Softw. Eng. Notes*, 30(5):217–226, September 2005.

[8] Arianna Blasi, Alberto Goffi, Konstantin Kuznetsov, Alessandra Gorla, Michael D. Ernst, Mauro Pezzè, and Sergio Delgado Castellanos. Translating code comments to procedure specifications. In Frank Tip and Eric Bodden, editors, *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018*, pages 242–253. ACM, 2018.

[9] Sandrine Blazy and Xavier Leroy. Mechanized semantics for the clight subset of the C language. *J. Autom. Reason.*, 43(3):263–288, 2009.

[10] Edwin Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming*, 23(05):552–593, 2013.

[11] TIOBE Software BV. Tiobe index for november 2020. `https://www.tiobe.com/tiobe-index/`.

[12] Adam Chlipala. An introduction to programming and proving with dependent types in coq. *J. Formaliz. Reason.*, 3(2):1–93, 2010.

[13] Ravi Chugh, David Herman, and Ranjit Jhala. Dependent types for javascript. In Gary T. Leavens and Matthew B. Dwyer, editors, *Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2012, part of SPLASH 2012, Tucson, AZ, USA, October 21-25, 2012*, pages 587–606. ACM, 2012.

[14] Edmund M. Clarke, William Klieber, Milos Novácek, and Paolo Zuliani. Model checking and the state explosion problem. In Bertrand Meyer and Martin Nordio, editors, *Tools for Practical Software Verification, LASER, International Summer School 2011, Elba Island, Italy, Revised Tutorial Lectures*, volume 7682 of *Lecture Notes in Computer Science*, pages 1–30. Springer, 2011.

[15] David R. Cok. Openjml: Software verification for java 7 using jml, openjdk, and eclipse. In Catherine Dubois, Dimitra Giannakopoulou, and Dominique Méry, editors, *Proceedings 1st Workshop on Formal Integrated Development Environment, F-IDE 2014, Grenoble, France, April 6, 2014*, volume 149 of *EPTCS*, pages 79–92, 2014.

[16] Leonardo Mendonça de Moura and Nikolaj 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.

[17] Robert DeLine and Manuel Fähndrich. Typestates for objects. In Martin Odersky, editor, *ECOOP 2004 - Object-Oriented Programming, 18th European Conference, Oslo, Norway, June 14-18, 2004, Proceedings*, volume 3086 of *Lecture Notes in Computer Science*, pages 465–490. Springer, 2004.

[18] Clarke E., Emerson E., and Sifakis J. Turing lecture: Model checking: Algorithmic verification and debugging. `https://dl.acm.org/doi/pdf/10.1145/1592761.1592781`.

[19] JVM ecosystem REPORT 2021. Snyk report 2021. `https://snyk.io/jvm-ecosystem-report-2021/`.

[20] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.*, 69(1-3):35–45, 2007.

[21] Jean-Christophe Filliâtre, Léon Gondelman, and Andrei Paskevich. The spirit of ghost code. *Formal Methods Syst. Des.*, 48(3):152–174, 2016.

[22] Alcides Fonseca, Paulo Santos, and Sara Silva. The usability argument for refinement typed genetic programming. In Thomas Bäck, Mike Preuss, André H. Deutz, Hao Wang, Carola Doerr, Michael T. M. Emmerich, and Heike Trautmann, editors, *Parallel Problem Solving from Nature - PPSN XVI - 16th International Conference, PPSN 2020, Leiden, The Netherlands, September 5-9, 2020, Proceedings, Part II*, volume 12270 of *Lecture Notes in Computer Science*, pages 18–32. Springer, 2020.

[23] Eclipse Foundation. Eclipse ide 2021-06: The eclipse foundation. `https://www.eclipse.org/eclipseide/`.

[24] Timothy S. Freeman and Frank Pfenning. Refinement types for ML. In David S. Wise, editor, *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation (PLDI), Toronto, Ontario, Canada, June 26-28, 1991*, pages 268–277. ACM, 1991.

[25] Morling G. Bean validation specification. `http://beanvalidation.org/`.

[26] Catarina Gamboa. Liquidjava - project website, 2021.

[27] Gerard J. Holzmann. *The SPIN Model Checker - primer and reference manual*. Addison-Wesley, 2004.

[28] 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.

[29] Ranjit Jhala and Niki Vazou. Refinement types: A tutorial. *CoRR*, abs/2010.07763, 2020.

[30] Briski K., Chitale P, Hamilton V., Pratt A., Starr B., Veroulis J., and Villard B. Minimizing code defects to improve software quality and lower development costs. ftp://ftp.software.ibm.com/software/rational/info/do-more/RAW14109USEN.pdf.

[31] Lennart C. L. Kats, Richard G. Vogelij, Karl Trygve Kalleberg, and Eelco Visser. Software development environments on the web: a research agenda. In Gary T. Leavens and Jonathan Edwards, editors, *ACM Symposium on New Ideas in Programming and Reflections on Software, Onward! 2012, part of SPLASH '12, Tucson, AZ, USA, October 21-26, 2012*, pages 99–116. ACM, 2012.

[32] 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.

[33] David Holmes Ken Arnold, James Gosling. *THE Java™ Programming Language, Fourth Edition*. Addison Wesley Professional, 2005.

[34] Dimitrios Kouzapas, Ornela Dardha, Roly Perera, and Simon J. Gay. Typechecking protocols with mungo and stmungo: A session type toolchain for java. *Sci. Comput. Program.*, 155:52–75, 2018.

[35] Lean for vs code, 2021.

[36] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. JML: A notation for detailed design. In Haim Kilov, Bernhard Rumpe, and Ian Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, volume 523 of *The Kluwer International Series in Engineering and Computer Science*, pages 175–188. Springer, 1999.

[37] 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.

[38] Xavier Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, 2009.

[39] Awesome Java LibHunt. Java formal verification. https://java.libhunt.com/categories/412-formal-verification?order=activity.

[40] Gary Lindstrom, Peter C. Mehlitz, and Willem Visser. Model checking real time java using java pathfinder. In Doron A. Peled and Yih-Kuen Tsay, editors, *Automated Technology for Verification and Analysis, Third International Symposium, ATVA 2005, Taipei, Taiwan, October 4-7, 2005, Proceedings*, volume 3707 of *Lecture Notes in Computer Science*, pages 444–456. Springer, 2005.

[41] Barbara Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst.*, 16(6):1811–1841, 1994.

[42] Bertrand Meyer. Applying "design by contract". *Computer*, 25(10):40–51, 1992.

[43] Facundo Molina, Pablo Ponzio, Nazareno Aguirre, and Marcelo F. Frias. Evospex: An evolutionary algorithm for learning postconditions. In *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*, pages 1223–1235. IEEE, 2021.

[44] João Mota, Marco Giunti, and António Ravara. Java typestate checker. In Ferruccio Damiani and Ornela Dardha, editors, *Coordination Models and Languages - 23rd IFIP WG 6.1 International Conference, COORDINATION 2021, Held as Part of the 16th International Federated Conference on Distributed Computing Techniques, DisCoTec 2021, Valletta, Malta, June 14-18, 2021, Proceedings*, volume 12717 of *Lecture Notes in Computer Science*, pages 121–133. Springer, 2021.

[45] Ulf Norell. Dependently typed programming in agda. In Pieter W. M. Koopman, Rinus Plasmeijer, and S. Doaitse Swierstra, editors, *Advanced Functional Programming, 6th International School, AFP 2008, Heijen, The Netherlands, May 2008, Revised Lectures*, volume 5832 of *Lecture Notes in Computer Science*, pages 230–266. Springer, 2008.

[46] Upadhyay P. The role of verification and validation in system developmentlife cycle. `http://www.iosrjournals.org/iosr-jce/papers/Vol5-issue1/D0511720.pdf`.

[47] Matthew M. Papi, Mahmood Ali, Telmo Luis Correa Jr., Jeff H. Perkins, and Michael D. Ernst. Practical pluggable types for java. In Barbara G. Ryder and Andreas Zeller, editors, *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2008, Seattle, WA, USA, July 20-24, 2008*, pages 201–212. ACM, 2008.

[48] Terence John Parr and Russell W. Quong. ANTLR: A predicated- *LL(k)* parser generator. *Softw. Pract. Exp.*, 25(7):789–810, 1995.

[49] Renaud Pawlak, Martin Monperrus, Nicolas Petitprez, Carlos Noguera, and Lionel Seinturier. SPOON: A library for implementing analyses and transformations of java source code. *Softw. Pract. Exp.*, 46(9):1155–1179, 2016.

[50] Benjamin C. Pierce. *Types and programming languages*. MIT Press, 2002.

[51] Pypl popularity of programming language. `http://pypl.github.io/PYPL.html`.

[52] Patrick Maxim Rondon, Alexander Bakst, Ming Kawaguchi, and Ranjit Jhala. Csolve: Verifying C with liquid types. In P. Madhusudan and Sanjit A. Seshia, editors, *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*, volume 7358 of *Lecture Notes in Computer Science*, pages 744–750. Springer, 2012.

[53] Patrick Maxim Rondon, Ming Kawaguchi, and Ranjit Jhala. Liquid types. In Rajiv Gupta and Saman P. Amarasinghe, editors, *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*, pages 159–169. ACM, 2008.

[54] Johnny Saldaña. *The Coding Manual for Qualitative Researchers*. SAGE Publications, 2009.

[55] 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.

[56] Benno Stein, Lazaro Clapp, Manu Sridharan, and Bor-Yuh Evan Chang. Safe stream-based programming with refinement types. *CoRR*, abs/1808.02998, 2018.

[57] Robert E. Strom and Shaula Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Software Eng.*, 12(1):157–171, 1986.

[58] Yong Kiam Tan, Magnus O. Myreen, Ramana Kumar, Anthony C. J. Fox, Scott Owens, and Michael Norrish. The verified cakeml compiler backend. *J. Funct. Program.*, 29:e2, 2019.

[59] John Toman, Ren Siqi, Kohei Suenaga, Atsushi Igarashi, and Naoki Kobayashi. Consort: Context- and flow-sensitive ownership refinement types for imperative programs. In Peter Müller, editor, *Programming Languages and Systems - 29th European Symposium on Programming, ESOP 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings*, volume 12075 of *Lecture Notes in Computer Science*, pages 684–714. Springer, 2020.

[60] Niki Vazou, Eric L. Seidel, and Ranjit Jhala. Liquidhaskell: experience with refinement types in the real world. In Wouter Swierstra, editor, *Proceedings of the 2014*

ACM SIGPLAN symposium on Haskell, Gothenburg, Sweden, September 4-5, 2014, pages 39–51. ACM, 2014.

[61] 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.

[62] 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.

[63] Emacs website. Gnu emacs. `https://www.gnu.org/software/emacs/`.

[64] Official LSP website. Language server protocol. `https://microsoft.github.io/language-server-protocol/`.

[65] Visual Studio Code website. Visual studio code. `https://code.visualstudio.com/`, 2016.

[66] Hyrum Wright, Titus Delafayette Winters, and Tom Manshreck. *Software Engineering at Google*. 2020.