Featherweight Liquid Java : Typestate and Liquid Types in Java

CATARINA GAMBOA, fc49535, Faculdade de Ciências da Universidade de Lisboa, Portugal

Stateful Object-Oriented software defines APIs that have an implicit required protocol usage. For instance, some methods may only make sense after the object transitions to another state. In fact, there are many software components that work as a state machine. Writing software that does not conform to these sometime implicit expectations can cause bugs.

Within techniques to find bugs in software, the ones that provide immediate feedback to developers when their code is not conforming to the specification is useful to increase productivity, and provides some assurance that those type of bugs are not present. Moreover, specifying the state machine in the code makes these implicit assumptions explicit and serves as documentation.

Therefore, this paper presents Featherweight LiquidJava, a lightweight version of Java that includes typestates and liquid types to support immediate feedback to developers of incorrect invocations of protocols that depend on the object state. Furthermore, by encoding typestates within the logic of liquid types, it is possible to express richer properties, relating states to variable values. We present motivation and introduction to LiquidJava as well as an initial step into the formalization of the language.

CCS Concepts: • **Software and its engineering → State based definitions**; **Object oriented languages**; **Classes and objects**; **Constraints**.

Additional Key Words and Phrases: Typestate, Liquid Types, Refinement Types

ACM Reference Format:
Catarina Gamboa. 2022. Featherweight Liquid Java : Typestate and Liquid Types in Java. In . ACM, New York, NY, USA, 16 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Writing software with state is a common practice in software engineering, especially in object-oriented programming (OOP). In fact, state machines are a common form of software documentation, describing which methods that can be called in each state. However, in popular OOP programming languages, this information is only in documentation, and misuse of the state machine is only detected at runtime, from side-effects caused by the misuse.

Motivation

As a very simple illustrative example, Java's `Thread.start()` will throw an `IllegalThreadStateException` if called a second time. Similarly, `Thread.join()` will also raise the same exception if the Thread has not been started before the invocation. These exceptions, such as `IllegalStateException`, are used very frequently in Java code (179k results from a Github code search[1]) to raise errors when a method is called at a wrong time in the program.

[1]https://github.com/search?l=Java&q=IllegalStateException&type=Issues

Software Analysis, 2022, Lisboa, Portugal
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-XXXX-X/18/06...$15.00
https://doi.org/XXXXXXX.XXXXXXX

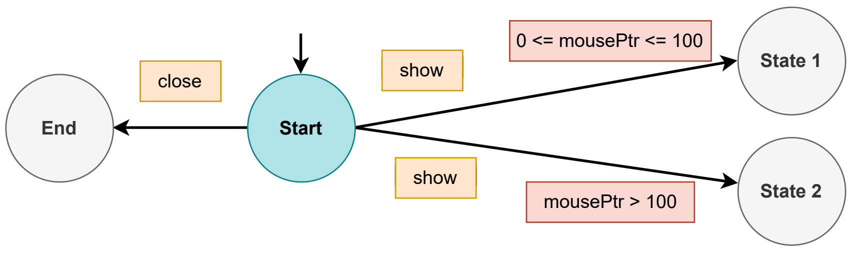


Fig. 1. Example of Context-dependent Symbolic Automata.

David Khourshid identified implicit state machines as the reason for the Apple Facetime bug that would allow a user to hear the other side of the call, even before the call was accepted. "Implicit state machines are dangerous because, since the application logic implied by the state machine is scattered around the code base, it only exists in the developers' heads, and they're not very well-specified. Nobody can quickly reference what states the application can be in, nor how the states can transition due to events, nor what happens when events occur in certain states, because that crucial information is just not readily available without studying and untangling the code in which it exists." [14]

One of the proposed solutions to ensure the fulfillment of the implicit protocol is making the state machine explicit in the documentation, or in the source code. Integration in the source code, together with a verification tool, supports automatic detection of violations and assurance that the protocol is not broken.

Design Patterns for Typestates

Fluent APIs [9] can encode state machines, where each state has an interface, which is returned from each method call. However, this approach is only practical when there is a single implicit state machine. Statecharts [26] provide an alternative formalism that supports a hierarchical organization of states, with substates refining more abstract states, with the goal of promoting reuse of states. States have also been promoted to first-class in the Plaid programming language [30], making it impossible to invoke a method not defined in the current state.

However, in the previous approaches, the verification of protocol violations was based solely on the state of objects, and the value of fields or variables was not taken into account to verify whether a given transition was correct or not. In a forward step from traditional automata, symbolic automata aims to encode this additional information that allows more expressive state transitions.

Context-dependent Symbolic Automata

Symbolic Automata [6] is an extension of classical automata, but with transitions being labeled with formulas over a single variable instead of a single, discrete label. As an example, to transition from state A to state B, the formula `size(this)> 0` must hold. Symbolic Automata have an higher expressive power, derived from the formula language supported (e.g., based on the theory supported by an SMT solver).

We aim to support a more expressive class of automata – Context-dependent Symbolic Automata – where the formula can refer to more than a single variable, taken from the program context, which might mutate between and during transitions. A simple example of such automata is shown in Figure 1, where a program that is in the state *Start* can go to three different states, *State1*, *State1* and *End* depending on the method invoked and the value of the variable in context *mousePtr*. Therefore, if the method *show* is invoked, the program goes to *State 1* if the *mousePtr* context variable has a value between zero and 100, or goes to *State 2* if the variable has a value greater than 100.

In language specification typestates are useful to encode the state transitions without the predicates. However, they do not have primitives to include predicates in the transitions and therefore need to be combined with other techniques to allow the more expressive specification.

Liquid Types

To support the verification of these more complex automata, we propose to combine Typestate with Logically Qualified Data Types, also known as Liquid Types [24]. These types extend traditional types with a refinement predicate taken from a decidable Boolean Logic theory. The same logic is followed by Symbolic Automata restraining the logic to SMT Algebra. Therefore, with Liquid Types it is possible to refine the types of variables, parameters and return values. An example of a method annotated with Liquid Types is $\{ret > x\}$ int *increase* $(\{x > 0\} \;\; x)$ that only accepts an integer greater than zero as parameter and ensures the result is larger than the input.

Liquid Types represent the decidable portion of refinement types, a concept introduced in the ML language by Freeman and Pfenning in 1991 [11]. Since then, there have been multiple implementations of refinement types in functional languages like Haskell (LiquidHaskell [31]) or Rust (Flux [18]), imperative languages like C (Csolve [23], RefinedC [27]), scripting languages such as Javascript (DJS [5]) and Typescript [32], and object-oriented languages like Scala [28].

Liquid Type Checkers have been used in a wide range of problems with different complexities. From detecting simple division by zero errors and out-of-bounds array access bugs [33], to complex security issues [2] and protocol violations [4]. However, this is the first work that aims to join Liquid Types with Typestate to support the verification of more complex state machines.

Contributions

This paper presents the initial steps into the formalization of LiquidJava, a subset of Java that supports both Liquid Types and Typestate. It introduces support for mutable variables and typestate on top of traditional Liquid Type systems.

We show the type system for LiquidJava with detailed descriptions about each of the rules along with some examples that show the rules in action. The type system is created on top of Featherweight Java [13] to reduce the complexity of the target language.

Following the type system, we implemented the type system in a stand-alone Java type checker, which has been integrated as a plugin for Visual Studio Code.

The remainder of the paper introduces LiquidJava Section 2 and the initial formalization of the language Section 3. Then, we compare our approach to related work(Section 4), and, finally, discuss remaining work and draw conclusions (??).

2 INTRODUCTION TO LIQUIDJAVA

This subsection describes in high level the LiquidJava language and the different phases of the verification performed. First, we outline the underlying architecture of the system (section 2), then we zoom into the verification conditions sent to the SMT solver (section 2).

Architecture

The pipeline for verification of LiquidJava programs is represented in fig. 2. The process starts with an input Java file with refinements and type states annotated. Then the Java code from this file is parsed and type checked, producing a Java AST where we append the parsed refinements metadata. After all the typing information is in the AST we apply a liquid type checking algorithm where verification conditions are singled out and sent to an SMT-Solver to be verified. In this

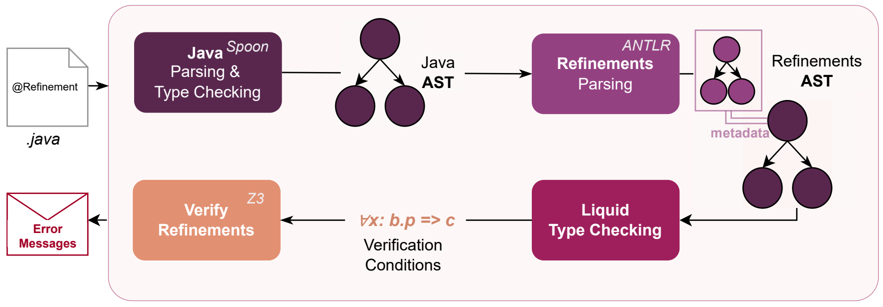


Fig. 2. Architecture of the LiquidJava System.

verification either the verification terminates successfully resuming the type checking algorithm until all conditions are verified, or the verification is unsuccessful and an error message is sent to the user.

In detail, th input of the system is a Java program with the liquid types encoded as string inside annotations [2]. These annotations are not verified by the Java type checker and are always optional in the code. Therefore, by adding the refinements inside annotations, the Java program can still be compiled by a regular Java checker without any modifications.

The LiquidJava program starts by being parsed through Spoon [22], a library for Java source code analysis and manipulation. Spoon provides the abstract syntax tree (AST) of the Java program, including all the annotations, allowing accesses and modifications of the tree. Before starting the liquid type checking algorithm, all the refinements written as string are also parsed and stored as metadata within the Java AST. The syntax of these expressions is detailed in the next subsection (fig. 5).

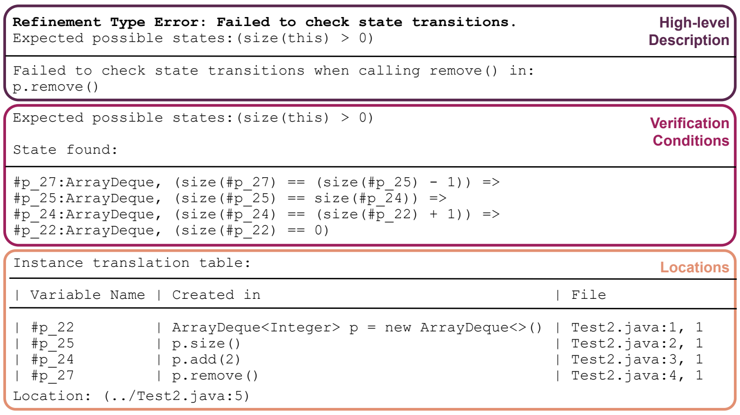
Therefore, before starting the verification there is an intermediate step where we traverse the AST to find the annotations with refinements and parse them using the ANTLR framework [21]. This framework allows us to introduce the custom refinements grammar, composed of predicates in the quantifier free logic with uninterpreted functions and linear integer arithmetic (QF-UFLIA), and produces an AST that we can use to reason about the predicates.

Now that the AST contains the nodes of the Java program and the predicates metadata, the liquid type checker starts traversing the AST and applying the typing rules presented in the Section 3.3. During the rules application, when there is a verification of a subtyping relationship or an $SmtValid$ invocation, they are discarded to an SMT Solver. The current implementation uses the Z3 Solver [7] given its easy incorporation in software development. The next two section delve into the SMT-based verification and the error messages produced by the system that are output from the system if the verification fails.

SMT-based verification

The SMT Solver receives verification conditions (VC's) that represent the subtyping relationships to be verified. These work as a intermediate representation between the program specification and the SMT Solver encoding. During the execution of the liquid type checking algorithm these conditions are gathered using program refinements and are translated into SMTLib. This translation is implemented using the Java API of Z3 through the Z3-TurnKey Distribution [3]. To simplify the queries sent to Z3, each verification condition is flatten to its minimal form (i.e., removing

[2]https://docs.oracle.com/javase/tutorial/java/annotations/
[3]https://github.com/tudo-aqua/z3-turnkey



```
Refinement Type Error: Failed to check state transitions.
Expected possible states:(size(this) > 0)

Failed to check state transitions when calling remove() in:
p.remove()

Expected possible states:(size(this) > 0)

State found:

#p_27:ArrayDeque, (size(#p_27) == (size(#p_25) - 1)) =>
#p_25:ArrayDeque, (size(#p_25) == size(#p_24)) =>
#p_24:ArrayDeque, (size(#p_24) == (size(#p_22) + 1)) =>
#p_22:ArrayDeque, (size(#p_22) == 0)

Instance translation table:

| Variable Name | Created in                            | File

| #p_22         | ArrayDeque<Integer> p = new ArrayDeque<>() | Test2.java:1, 1
| #p_25         | p.size()                              | Test2.java:2, 1
| #p_24         | p.add(2)                              | Test2.java:3, 1
| #p_27         | p.remove()                            | Test2.java:4, 1
Location: (../Test2.java:5)
```

Fig. 3. Example of an error message produced by LiquidJava.

refinements not needed for the verification of a predicate). Given the decidability of the verification conditions, the result of the SMT computation can be either UNSAT, meaning that the verification was proved and the algorithm can continue, or SAT, terminating the verification and creating an error message to output from the system.

Error Messages. The error message output from the system aims to inform the user of the reason for the verification failure. Therefore the error message is split into three main parts: high level error description, the failed verification conditions sent to the SMT solver, and, finally, a translation table to translate each variable of the local context to the place in code where it was created. An example of an error produced by an example presented above is shown in fig. 3.

3 FORMALIZATION OF LIQUIDJAVA

3.1 Notation

The following sections present the typing rules of LiquidJava. We start by summarizing the notation used in the next sections.

Ψ	*set of quantifier-free formulas over free x*
A	*algebra*
$SMT_{T'}$	*SMT algebra*
q	*state*
Q	*state set*
FS	*set of final states*
$q \xrightarrow{p} q'$	*transition*
δ	*transition set*
$a(C)$	*object state automaton for class C*
H	*state pointer list*
H^0_Γ	*initial state pointer list*
$[E]^t_{\Gamma;\Delta/H}$	*statement interpretation*
$prec(m)$	*method preconditions*

3.2 Syntax

This section sets the syntax used for the formalization of LiquidJava. The syntax is based on Featherweight Java [13] with extensions for refinement types and type states.

Starting with the types of the language (T), they are defined as refinement types in the form $\{p\}\ T'$ therefore including a basic type and a predicate. The basic types can be primitive (int, boolean) or defined as classes (C, D). If the refinement predicate is the boolean expression true, then $\{\text{true}\}\ T'$ is shorten to T'.

C, D	$\in$	*Class Names*	
T'	$::=$	int | boolean | C	*basic type*
T	$::=$	$\{p\}\ T'$	*refined type*

```
// Variable Declaration
----------------------------
@Refinement("x > 0") int x;


// Method Declaration
------------------------------
@Refinement(
   "0 <= return && return <= 100")
int percentage(
   @Refinement("max > 0") int max,
   @Refinement("0 <= val && val <= max")
       ↪  int val)
   {...}
```

a) Concrete Syntax

```
----------------------
{x > 0} int x;


---------------------
{0 <= return && return <= 100}
int percentage(
   {max > 0} int max,
   {0 <= val && val <= max} int val)

   {...}
```

b) Formalization Syntax

Fig. 4. Examples showing Syntax for Types

These types are used, for example, when declaring variables and methods' parameters and return types. Figure 4 includes two examples that demonstrate the syntax for writing the types divided into a concrete and a formalization version. The concrete syntax version is the one designed in previous work by the authors[12] and that is implemented in a prototype. On the other hand, the formalization syntax aims to be more simple while still capturing the language features. By putting both of them side by side we aim to show the translation between both of the versions.

The syntax constructs are defined in Figure 5. The predicates used in the refinements are logical expressions (defined in e, p), the statements and assignments consist of variable definitions, if-then-else expressions, return of variables and assignments (E definition). For assignments, variables can be assigned with expressions, new objects and the result of method invocations.

The constructs to introduce typestate can be split into two: the definition of the states available in the class to which we call *state sets*, and the addition of state transitions in the methods.

A stateset $\{\overline{s}\}$ in the class definition declares the possible states of the object. To describe type state transitions, the annotation $@(s \gg s \mid p)$ in method signatures can be used and therefore

$$
\begin{array}{rcll}
x, y, z, \mathit{this} & \in & \mathit{Identifier\ Names} & \\
m & \in & \mathit{Method\ Names} & \\
s & \in & \mathit{State\ Names} & \\
c & ::= & b \in \{\mathrm{true}, \mathrm{false}\} \mid n \in \mathbb{N} \mid C\{\} \mid \mathrm{null} & \mathit{constants} \\
op & ::= & + \mid - \mid < \mid \leq \mid > \mid \geq \mid == \mid != \mid \&\& \mid || & \mathit{binary\ operation} \\
e, p & ::= & x \mid c \mid e\ op\ e \mid !e \mid x(\overline{x}) & \mathit{expression} \\
E & ::= & T\ x \mid \mathrm{if}\ e\ \mathrm{then}\ B\ \mathrm{else}\ B \mid \mathrm{return}\ x & \mathit{statement} \\
 & \mid & x = e \mid x = \mathrm{new}\ C() \mid x = x.m(\overline{x}) & \mathit{assignment} \\
B & ::= & \cdot \mid E; B & \mathit{body} \\
S & ::= & @(s \gg s \mid p) & \mathit{state\ change} \\
M & ::= & \overline{S}\ T\ m\ (\overline{T\ x})\ \{B\} & \mathit{method\ definition} \\
SS & ::= & \mathrm{stateset}\ \{\overline{s}\} & \mathit{class\ state\ set} \\
CL & ::= & \overline{SS}\ \mathrm{class}\ C\ \mathrm{extends}\ D\ \{\overline{M}\} & \mathit{class\ definition} \\
PG & ::= & \overline{CL}\ \mathrm{class\ Main}\ \{\ \mathrm{public\ static\ void\ main}()\ \{B\}\ \} & \mathit{program}
\end{array}
$$

Fig. 5. LiquidJava Syntax

determine how the method invocation affects the object's state. The refinement predicates can appear both in the types T as explained before or also in the state transition as the condition p that must be met to allow the transition.

There can be multiple transition annotations for a single method. A method call is possible when the object satisfies the input state and condition of one of the transition annotations.

Figure 6 includes an examples for the concrete and formal syntax conversion. In this case the formal syntax simplifies the predicates, splitting the state and the refinements and removing the states arguments, only allowing state changes in the current object (*this*). The example portrayed implements in LiquidJava the automata presented in the introduction of the paper (Figure 1).

3.3 Typing Rules

This section describes the typing rules that belong to the liquid type checking algorithm.

To store the environment for the typing rules, we use two contexts that keep track of the object types. Γ is a global context where the user-provided type signatures are stored. Δ is a local context with more precise types that are deduced from reassignments to a variable.

$$
\begin{array}{ccll}
\Gamma & ::= & \emptyset \mid \Gamma, x : T & \mathit{global\ context} \\
\Delta & ::= & \emptyset \mid \Delta, x_i : T & \mathit{local\ context}
\end{array}
$$

We start by defining Entailment rules fig. 7. These three rules depend on both contexts and check a predicate using a SMT solver. The first rule $SMT - Emp$ validates the predicate using the SMT call *SmtValid* when the contexts are empty. Then, $SMT - \Gamma$ and $SMT - \Delta$ correspond to checking a predicate depending on the information in each context.

The rules for Subtyping are essential in systems with refinement types and for LiquidJava they are shown in Figure 8. $s - Id$ shows the identity rule for a given type. In the transitive rule, $s - Tr$, a given type T'' is subtype of another T if there is a subtyping relation with an intermediate T'. A class is a subtype of another if it defined in the class definition. The rule $s - Ref$ shows how the subtyping works when refinements are present. Therefore, it is only possible to prove that qT_1' is a subtype of pT_2' if the SMT verifies that $q \implies p$ and the two basic types follow also the subtyping relationship.

```
@StateSet({"start", "state1",
          "state2", "end"})
class MyObject{
 @StateRefinement(
   from = "start(this) && 0 <= mousePtr
          && mousePtr <= 100",
     to = "state1(this)")
 @StateRefinement(
   from = "start(this)
          && mousePtr > 100",
      to = "state2(this)")
 int show (int mousePtr){...}

 @StateRefinement(from = "start(this)",
                    to = "end(this)")
 int close (){...}
}
```

a) Concrete Syntax

```
stateset{start, state1, state2, end}

class MyObject extends Object{

   @(start >> state1 |
    0 <= mousePtr && mousePtr <= 100)

   @(start >> state2 | mousePtr > 100)

   int show (int mousePtr){...}



   @(start >> end)

   int close (){...}
}
```

b) Formalization Syntax

Fig. 6. Examples showing Syntax for State Sets and State Changes in methods

Entailment $\boxed{\Gamma;\ \Delta \vdash_{SMT} p}$

$$\textbf{SMT-Emp}\ \frac{SmtValid(p)}{\emptyset;\ \emptyset \vdash_{SMT} p} \qquad \textbf{SMT-}\Gamma\ \frac{\Gamma;\ \Delta \vdash_{SMT} \forall x : T.p \Rightarrow p}{\Gamma, x : \{p\}\ T;\ \Delta \vdash_{SMT} p}$$

$$\textbf{SMT-}\Delta\ \frac{\Gamma;\ \Delta \vdash_{SMT} \forall x : T.p \Rightarrow p}{\Gamma;\ \Delta, x_i : \{p\}\ T \vdash_{SMT} p}$$

Fig. 7. Entailment by SMT solver.

Expression Typing. Expression typing is presented in Figure 9. For constants the rules are immediate. The rule for subtyping $t - Sub$ is also regular. To proof that a variable has a given type, it should be the same as the local context contains for the variable with the maximum instance, as given in $t - Var$. This is important to reason about Strong Updates, as will be clarified in the set of rules for Body Typing.

Finally, $t - App$ gives the type for a given invocation of defined functions. An example for the use of this rule is the invocation of the *old* predicate over an object.

Body Typing. The typing rules for the body of methods are in Figure 10. These rules define an input and output environment, since changes to both contexts can happen. Therefore, the expression $\Gamma; \Delta \vdash B \dashv \Gamma; \Delta$ informs that the Body of a method will be typechecked against an initial global and local contexts and will produce another pair of global and local contexts.

Subtyping $\boxed{\Gamma;\, \Delta \vdash T <: T}$

$$\textbf{s-Id}\ \frac{}{\Gamma;\, \Delta \vdash T <: T} \qquad \textbf{s-Tr}\ \frac{\Gamma;\, \Delta \vdash T'' <: T' \quad \Gamma;\, \Delta \vdash T' <: T}{\Gamma;\, \Delta \vdash T'' <: T}$$

$$\textbf{s-Decl}\ \frac{decl(C) = SS \text{ class } C \text{ extends } D\ \{\overline{M}\}}{\Gamma;\, \Delta \vdash C <: D} \qquad \textbf{s-Ref}\ \frac{\Gamma;\, \Delta \vdash_{SMT} q \Rightarrow p \quad \Gamma;\, \Delta \vdash T'_1 <: T'_2}{\Gamma;\, \Delta \vdash \{q\}\ T'_1 <: \{p\}\ T'_2}$$

Fig. 8. Subtyping Rules.

Expression Typing $\boxed{\Gamma;\, \Delta \vdash e : T}$

$$\textbf{t-Bool}\ \frac{}{\Gamma;\, \Delta \vdash b : \text{boolean}} \qquad \textbf{t-Int}\ \frac{}{\Gamma;\, \Delta \vdash n : \text{int}}$$

$$\textbf{t-Sub}\ \frac{\Gamma;\, \Delta \vdash x : T' \quad \Gamma;\, \Delta \vdash T' <: T}{\Gamma;\, \Delta \vdash x : T}$$

$$\textbf{t-Var}\ \frac{\Delta(x_{max(i)}) = T}{\Gamma;\, \Delta \vdash x : T} \qquad \textbf{t-App}\ \frac{defined\, T\ x(\overline{z_i : T_i}) \quad \Gamma;\, \Delta \vdash y_i : T_i}{\Gamma;\, \Delta \vdash x(\overline{y}) : T}$$

Fig. 9. Typing rules for expressions.

$T - Emp$ defines that with an empty operation there are no changes in the environment.

The sequence rule, $T - Seq$ shows the transitive relation between the contexts with two consecutive statements are performed.

$T - Decl$ shows that if p is a boolean, then the declaration of a variable with type pT' will produce a change on the global context where a new variable is introduced with the given type.

The following four rules ($T - Var$, $T - Expr$, $T - New$, $T - Inv$) represent different types of assignments. In all of them, despite the complexity, the assignment is checked against the assignee expected type (from the global context), and in all, a new instance of the assignee is created in the local context with the new type instance. This represents the reasoning behind strong updates. Therefore, we can analyze an example before we detail each of the typing rules.

Strong Updates. In imperative programming languages, like Java, variables are mutable and can be assigned multiple values throughout the program. In LiquidJava, a variable is declared with a given basic type (int, boolean, or an object type) and a given predicate. However, throughout the program the assignments to the variable have a stronger type that the declared one, and by saving and using the stronger types we can make more expressive verifications.

Listing 1 contains an example of strong updates. If we did not support strong updates, the verification would fail to check the assignment of b against the expected refinement type, as expressed in the comments in the code.

In this example, the information stored in the global context at the end of the typechecking of this code snippet is:

$$\Gamma, a : \{a > 0\}int, b : \{b > 40\}int \tag{1}$$

Body Typing $\boxed{\Gamma;\ \Delta \vdash B \dashv \Gamma;\ \Delta}$

$$\textbf{T-Emp}\ \frac{}{\Gamma;\ \Delta \vdash \cdot \dashv \Gamma;\ \Delta}$$

$$\textbf{T-Seq}\ \frac{\Gamma;\ \Delta \vdash E \dashv \Gamma';\ \Delta' \quad \Gamma';\ \Delta' \vdash B \dashv \Gamma'';\ \Delta''}{\Gamma;\ \Delta \vdash E; B \dashv \Gamma'';\ \Delta''}$$

$$\textbf{T-Decl}\ \frac{\Gamma;\ \Delta \vdash p : \text{boolean}}{\Gamma;\ \Delta \vdash \{p\}\ T'\ x \dashv \Gamma, x : \{p\}\ T';\ \Delta}$$

$$\textbf{T-Var}\ \frac{\Gamma \vdash x : T \quad \Gamma;\ \Delta \vdash y : T \quad \mathit{fresh}\ x_i}{\Gamma;\ \Delta \vdash x = y \dashv \Gamma;\ \Delta, x_i : T}$$

$$\textbf{T-Expr}\ \frac{\Gamma \vdash x : T \quad \Gamma;\ \Delta \vdash e : T \quad e \notin \mathit{Identifier\ Names} \quad \mathit{fresh}\ x_i}{\Gamma;\ \Delta \vdash x = e \dashv \Gamma;\ \Delta, x_i : T}$$

$$\textbf{T-New}\ \frac{\Gamma \vdash x : T \quad \mathit{fresh}\ x_i \quad \Gamma;\ \Delta \vdash \{\_ = \text{new}\ C()\}\ C <: T}{\Gamma;\ \Delta \vdash x = \text{new}\ C() \dashv \Gamma;\ \Delta, x_i : \{\_ = \text{new}\ C()\}\ C}$$

$$\textbf{T-Inv}\ \frac{\begin{array}{c} \mathit{decl}(m) = \overline{S}\ T\ m\ (\overline{T\ x})\ \{B\} \quad \Gamma \vdash z : T \quad \Gamma;\ \Delta \vdash x'_i : T_i \quad \Gamma;\ \Delta \vdash y : C \\ \Gamma, \overline{x : T}, \text{ret} : T;\ \Delta \vdash B \dashv \Gamma';\ \Delta' \\ \Gamma;\ \Delta \vdash_{SMT} \mathit{prec}(m)[\mathit{this} := y][\overline{x} := \overline{x'}] \quad \mathit{fresh}\ z_i \quad \mathit{fresh}\ y_i \end{array}}{\Gamma;\ \Delta \vdash z = y.m(\overline{x'}) \dashv \Gamma;\ \Delta, z_i : T, y_i : \mathit{postc}(m)[\mathit{this} := y][\overline{x} := \overline{x'}]}$$

$$\textbf{T-Ret}\ \frac{\Gamma;\ \Delta \vdash x : T \quad \text{ret} : T \in \Gamma}{\Gamma;\ \Delta \vdash \text{return}\ x \dashv \Gamma;\ \Delta}$$

$$\textbf{T-If}\ \frac{\begin{array}{c} \Gamma;\ \Delta \vdash x : \text{boolean} \\ \Gamma, \_ : \{x\}\ \text{int};\ \Delta \vdash B \dashv \Gamma';\ \Delta' \\ \Gamma, \_ : \{!x\}\ \text{int};\ \Delta \vdash B' \dashv \Gamma'';\ \Delta'' \end{array}}{\Gamma;\ \Delta \vdash \text{if}\ x\ \text{then}\ B\ \text{else}\ B' \dashv \Gamma;\ \mathit{merge}(x, \Gamma, \Delta', \Delta'')}$$

Fig. 10. Typing rules for method bodies.

While the information in the local context is:

$$\Delta, a_0 : \{a0 == 50\}\mathit{int}, b_1 : \{b_1 == a_0\}\mathit{int} \tag{2}$$

Therefore, in assignment rules, the assignment is checked against Γ and if correctly typed, a new instance is introduced in Δ.

```
{a > 0} int a;
a = 50; // Verification: a == 50 <: a > 0 ? Correct
{b > 40} int b;
b = a; /*Verification: (Strong Updates) (b == a && a == 50) <: (b > 40) True
                       (Weak Updates) (b == a && a > 0) <: (b > 40) False*/
```

Listing 1. Example for Strong Updates

$T - Var$ is the simplest assignment since it is only a variable by itself. Therefore if the type of the assignment y is a subtype of the global type of the assignee, a new $fresh$ variable x_i will be created, where i represents an increasing counter, and is appended to the local context.

$T - Expr$ is similar to $T - Var$ with the exception that the expression is not an identifier.

$T - New$ represents when a variable is assigned with a new object. Each class has a default initial state represented by $\_ = newC()$. Therefore if the default state is subtype of the assignee type, a new $fresh$ variable will be stored in Δ with the default refined type.

Invocation. $T - Inv$ is the most complex rule. To help understand it, we give two definitions (definition 3.2 and definition 3.2).

Definition 3.1 (Invocation Precondition for method $\overline{S}\ T\ m\ (\overline{T\ x})\ \{B\}$). Let $\overline{S}$ be a sequence of annotations: $@(s_1^{in} \gg s_1^{out} \mid p_1)\ @(s_2^{in} \gg s_2^{out} \mid p_2)\ \ldots\ @(s_n^{in} \gg s_n^{out} \mid p_n)$. Then, $prec(m)$ is a precondition for invocation of m:

$$prec(m) = \left(\bigvee_i s_i^{in}(this) \wedge p_i\right) \wedge \left(\bigwedge_{i,j\neq i} (s_i^{in}(this) \wedge p_i \wedge s_j^{in}(this) \wedge p_j) \Rightarrow \bot\right)$$

A state transition $@(s^{in} \gg s^{out} \mid p)$ can be triggered when $s^{in}(this) \wedge p$. Hence, the first clause of $prec(m)$ ensures that at least one of the transitions is satisfied. The second clause disallows multiple transitions to be valid at the same time.

Definition 3.2 (Invocation Postcondition for method $\overline{S}\ T\ m\ (\overline{T\ x})\ \{B\}$). Let $\overline{S}$ be a sequence of annotations: $@(s_1^{in} \gg s_1^{out} \mid p_1)\ @(s_2^{in} \gg s_2^{out} \mid p_2)\ \ldots\ @(s_n^{in} \gg s_n^{out} \mid p_n)$. Then, $prec(m)$ is a precondition for invocation of m:

$$postc(m) = \bigwedge_i (s_i^{in}(this) \wedge p_i \Rightarrow s^{out}(this))$$

A state transition $@(s^{in} \gg s^{out} \mid p)$ can be triggered when $s^{in}(this) \wedge p$. Hence, the first clause of $prec(m)$ ensures that at least one of the transitions is satisfied. The second clause disallows multiple transitions to be valid at the same time.

Now, going back to $T - Inv$ rule, we start by retrieving the declaration of the method and the type of the assignee. Then, each of the arguments of the invocation needs to have the type declared in the declaration, and the return type of the method has to be the same as the type of the assignee. With the additional information from the declaration types, the body B of the method is type checked. At last, the preconditions of the method are verified with the substitution of the declared variables by the target variable and the parameters, and two fresh variables are created. If all these conditions are correctly checked, a new instance of the assignee z will be put in context with the return type. Moreover, the postcondition of the method with the substitutions is appended to Δ changing the type of the target variable y.

Local Context Merging $\boxed{x \in merge(cond, \Delta, \Delta)}$

$$\text{M-Common}\ \frac{x_i : T \in \Delta_l \quad x_i : T \in \Delta_r}{x_i : T \in merge(cond, \Delta_l, \Delta_r)}$$

$$\text{M-Left}\ \frac{x_i : \{p\}\ T' \in \Delta_l \quad x_i : T \notin \Delta_r}{x_i : \{cond \Rightarrow p\}\ T' \in merge(cond, \Delta_l, \Delta_r)}$$

$$\text{M-Right}\ \frac{x_i : T \notin \Delta_l \quad x_i : \{p\}\ T' \in \Delta_r}{x_i : \{!cond \Rightarrow p\}\ T' \in merge(cond, \Delta_l, \Delta_r)}$$

$$\text{M-NewMax}\ \frac{x_{\max(\Delta_l)} : \{p_l\}\ T \in \Delta_l \quad x_{\max(\Delta_r)} : \{p_r\}\ T \in \Delta_r \quad fresh\ x_i}{x_i : \{\text{if } cond \text{ then } p_l \text{ else } p_r\}\ T \in merge(cond, \Delta_l, \Delta_r)}$$

Fig. 11. Rules for Merging Local Context.

Return. Given the previous rule, where the body of a method is evaluated with the special variable ret, the return rule only needs to verify if the return variable has the correct type.

If Statement. The last body for method's body, $T - If$, uses the variable x as condition for deciding which branch should be pursued. Therefore, x need to have a boolean type and a path variable (denoted by the symbol _) will have the a boolean value related to it as refinement in context. Then, the true branch includes the path variable with x as refinement, and the false branch gets a path variable with $!x$ as a refinement. With the path condition in context, it is possible to typecheck the statements in the body of each branch. After this verification, the local context is resultant from the merging of the two contexts retrieved at the end of the branches verification. The details for the Local Context merging can be found in Figure 11.

The remaining Typing Rules for methods, classes and programs are in Figure 12.

4 RELATED WORK

Fluent Interfaces have been commonly used in practice given that they display a set of methods that can be called in sequence while providing meaningful and easier to read code [19]. These interfaces basically represent an internal Domain Specific Language (DSL) that specify the behavior of a class [10], and as so, they can implement easily implement protocols and state machines [25]. There are multiple examples of Java interfaces that follow this design pattern, some examples include `StringBuilder` [4], Date Time API [5], Apache Camel [6] and, Java Object Oriented Querying (jOOQ) API [7]. These interfaces represent implicit protocols, however, they often do not verify if the several methods are being invoked in the right order.

Approaches for statically verifying typestate make the states and allowed state changes explicit, and verify if the programs abide by the written protocol. Strom et al. [29] introduced the concept of typestate in object oriented languages and presented how one can encode state machines graphs

[4]https://docs.oracle.com/javase/7/docs/api/java/lang/StringBuilder.html
[5]https://www.oracle.com/technical-resources/articles/java/jf14-date-time.html
[6]https://camel.apache.org/manual/java-dsl.html
[7]https://www.jooq.org

Method Typing $\boxed{\overline{S}\ T\ m\ (\overline{T\ x})\ \{B\}\ \text{OK in}\ C}$

$$\textbf{m-OK}\ \frac{\begin{array}{c} decl(C) = SS\ \text{class}\ C\ \text{extends}\ D\ \{\overline{M}\} \quad override(\overline{S}\ T\ m\ (\overline{T\ x})\ \{B\}, D) \\ \overline{x : T}, this : C, \text{ret} : T;\ \overline{x_0 : T} \vdash B \dashv \Gamma;\ \Delta \end{array}}{\overline{S}\ T\ m\ (\overline{T\ x})\ \{B\}\ \text{OK in}\ C}$$

Class Typing $\boxed{SS\ \text{class}\ C\ \text{extends}\ D\ \{\overline{M}\}\ \text{OK}}$

$$\textbf{c-OK}\ \frac{M_i\ \text{OK in}\ C}{SS\ \text{class}\ C\ \text{extends}\ D\ \{\overline{M}\}\ \text{OK}}$$

Program Typing $\boxed{\overline{CL}\ \text{class Main \{ public static void main()}\ \{B\}\ \text{\} OK}}$

$$\textbf{p-OK}\ \frac{CL_i\ \text{OK} \quad \emptyset;\ \emptyset \vdash B \dashv \Gamma;\ \Delta}{\overline{CL}\ \text{class Main \{ public static void main()}\ \{B\}\ \text{\} OK}}$$

Fig. 12. Typing of a Liquid Java program.

in the language. Following this concept, the paradigm of typestate-oriented programming was introduced in the Plaid language [1]. In Java, the typestate concept was introduced by Bierhoff et al.[3], following the work on Fugue [8], adding typestates as method pre- and post-conditions and relating typestate with subtyping and inheritance. However in this approach it is not possible to use dependent types or include refinement types.

Mungo [16] is also a tool for definition of object protocols in Java that uses the notion of typestate and statically verifies if the protocols are followed. In 2021, this tool was extended, as Java Typestate Checker [20], to include the modelling of function parameters and return values, verification of protocol completion, control over shared resources and null-pointer detection. Both tools rely on extra files to encode the protocol instead of using specifications inside Java code. The latest long-term support Java version, Java 17 (released in September 2021) also introduces sealed classes [8] that can be used to define state machines [9] by creating an interface with the allowed states and defining each class following the interface and their allowed transitions.

However, these previous mechanisms for typestate do not reach the expressiveness of symbolic automata since they lack the use of predicate formulas in state transitions along with the labels.

A closer approach that allies methods pre- and post-conditions with typestate is described by Kim et al. [15]. Here, the specifications are written in the Java Modelling Language (JML) [17] which allows the exposure of Java programs' behavior and the typestate information is incorporated into existing constructs. Although, this approach was not incorporated into static checkers, thus, not allowing verification at compile time.

5 CONCLUSIONS AND FUTURE WORK

This paper presents the first steps into the formalization of LiquidJava as a language that joins refinement types with type state within Java. However, there are still many steps within the language formalization. Specially, it is essential to find an appropriate set of operational semantics

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

for the language that lets us prove interesting properties about the language. At the moment, using as operational semantics a language with explicit type state and using the smt solver dynamically to prove the relevant properties makes the typechecking algorithm match the operational semantics. Therefore, this approach makes all the proofs trivial and not interesting to pursue.

One possibility to overcome this issue is to include more non-trivial properties in the LiquidJava language, such as aliasing, that provide an additional challenge and will impact the proofs needed making them more interesting from a theoretical point of view.

The current state of the formalization serves, therefore, as an intermediate step into the full verification of the language which we aim to obtain in future research steps.

ACKNOWLEDGEMENTS

The current state of the formalization work counted with the collaboration of Elizaveta Vasilenko, Niki Vazou, Alcides Fonseca and Jonathan Aldrich.

Moreover, the initial development of LiquidJava counted with collaborations of Paulo Santos and Christopher Timperley.

This work was supported by *Fundação para a Ciência e Tecnologia* (FCT) in the LASIGE Research Unit under the ref. UIDB/00408/2020 and UIDP/00408/2020 and through the CMU–Portugal Dual Degree PhD Program, by the CMU–Portugal project CAMELOT (LISBOA-01-0247-FEDER-045915), and the RAP project under the reference (EXPL/CCI-COM/1306/2021).

REFERENCES

[1] Jonathan Aldrich, Joshua Sunshine, Darpan Saini, and Zachary Sparks. 2009. Typestate-oriented programming. In *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*, Shail Arora and Gary T. Leavens (Eds.). ACM, 1015–1022. https://doi.org/10.1145/1639950.1640073

[2] Jesper Bengtson, Karthikeyan Bhargavan, Cédric Fournet, Andrew D. Gordon, and Sergio Maffeis. 2008. Refinement Types for Secure Implementations. In *Proceedings of the 21st IEEE Computer Security Foundations Symposium, CSF 2008*. IEEE Computer Society, 17–32. https://doi.org/10.1109/CSF.2008.27

[3] Kevin Bierhoff, Nels E. Beckman, and Jonathan Aldrich. 2009. Practical API Protocol Checking with Access Permissions. In *ECOOP 2009 - Object-Oriented Programming, 23rd European Conference, Genoa, Italy, July 6-10, 2009. Proceedings (Lecture Notes in Computer Science, Vol. 5653)*, Sophia Drossopoulou (Ed.). Springer, 195–219. https://doi.org/10.1007/978-3-642-03013-0_10

[4] Nuno Burnay, Antónia Lopes, and Vasco T. Vasconcelos. 2020. Statically Checking REST API Consumers. In *Software Engineering and Formal Methods - 18th International Conference*, Frank S. de Boer and Antonio Cerone (Eds.), Vol. 12310. 265–283. https://doi.org/10.1007/978-3-030-58768-0_15

[5] Ravi Chugh, David Herman, and Ranjit Jhala. 2012. Dependent types for JavaScript. In *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*, Gary T. Leavens and Matthew B. Dwyer (Eds.). ACM, 587–606. https://doi.org/10.1145/2384616.2384659

[6] Loris D'Antoni and Margus Veanes. 2017. The Power of Symbolic Automata and Transducers. 47–67. https://doi.org/10.1007/978-3-319-63387-9_3

[7] Leonardo Mendonça de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *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 (Lecture Notes in Computer Science, Vol. 4963)*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer, 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

[8] Robert DeLine and Manuel Fähndrich. 2004. Typestates for Objects. In *ECOOP 2004 - Object-Oriented Programming, 18th European Conference, Oslo, Norway, June 14-18, 2004, Proceedings (Lecture Notes in Computer Science, Vol. 3086)*, Martin Odersky (Ed.). Springer, 465–490. https://doi.org/10.1007/978-3-540-24851-4_21

[9] Martin Fowler. 2005. Fluent Interface. *Martin Fowler Bliki* (2005). http://www.martinfowler.com/bliki

[10] Martin Fowler. 2010. *Domain Specific Languages* (1st ed.). Addison-Wesley Professional.

[11] Timothy S. Freeman and Frank Pfenning. 1991. Refinement Types for ML. In *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation (PLDI), Toronto, Ontario, Canada, June 26-28, 1991*,

David S. Wise (Ed.). ACM, 268–277. https://doi.org/10.1145/113445.113468

[12] Catarina Gamboa, Paulo Alexandre Santos, Christopher Steven Timperley, and Alcides Fonseca. 2021. User-driven Design and Evaluation of Liquid Types in Java. *CoRR* abs/2110.05444 (2021). arXiv:2110.05444 https://arxiv.org/abs/2110.05444

[13] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. 2001. Featherweight Java: A Minimal Core Calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.* 23, 3 (may 2001), 396–450. https://doi.org/10.1145/503502.503505

[14] David Khourshid. 2019. The FaceTime Bug and the Dangers of Implicit State Machines. *Medium* (January 2019). https://medium.com/@DavidKPiano/the-facetime-bug-and-the-dangers-of-implicit-state-machines-a5f0f61bdaa2

[15] Taekgoo Kim, Kevin Bierhoff, Jonathan Aldrich, and Sungwon Kang. 2009. Typestate Protocol Specification in JML. In *Proceedings of the 8th International Workshop on Specification and Verification of Component-Based Systems* (Amsterdam, The Netherlands) *(SAVCBS '09)*. Association for Computing Machinery, New York, NY, USA, 11–18. https://doi.org/10.1145/1596486.1596490

[16] Dimitrios Kouzapas, Ornela Dardha, Roly Perera, and Simon J. Gay. 2018. Typechecking protocols with Mungo and StMungo: A session type toolchain for Java. *Sci. Comput. Program.* 155 (2018), 52–75. https://doi.org/10.1016/j.scico.2017.10.006

[17] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. 1999. JML: A Notation for Detailed Design. In *Behavioral Specifications of Businesses and Systems*, Haim Kilov, Bernhard Rumpe, and Ian Simmonds (Eds.). The Kluwer International Series in Engineering and Computer Science, Vol. 523. Springer, 175–188. https://doi.org/10.1007/978-1-4615-5229-1_12

[18] Nicolás Lehmann, Adam T. Geller, Gilles Barthe, Niki Vazou, and Ranjit Jhala. 2022. Flux: Liquid Types for Rust. *ArXiv* abs/2207.04034 (2022).

[19] Robert Liguori and Patricia Liguori. 2017. *Java Pocket Guide* (4th ed.). O'Reilly Media, Inc.

[20] João Mota, Marco Giunti, and António Ravara. 2021. Java Typestate Checker. In *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 (Lecture Notes in Computer Science, Vol. 12717)*, Ferruccio Damiani and Ornela Dardha (Eds.). Springer, 121–133. https://doi.org/10.1007/978-3-030-78142-2_8

[21] Terence John Parr and Russell W. Quong. 1995. ANTLR: A Predicated- *LL(k)* Parser Generator. *Softw. Pract. Exp.* 25, 7 (1995), 789–810. https://doi.org/10.1002/spe.4380250705

[22] Renaud Pawlak, Martin Monperrus, Nicolas Petitprez, Carlos Noguera, and Lionel Seinturier. 2016. SPOON: A library for implementing analyses and transformations of Java source code. *Softw. Pract. Exp.* 46, 9 (2016), 1155–1179. https://doi.org/10.1002/spe.2346

[23] Patrick Maxim Rondon, Alexander Bakst, Ming Kawaguchi, and Ranjit Jhala. 2012. CSolve: Verifying C with Liquid Types. In *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings (Lecture Notes in Computer Science, Vol. 7358)*, P. Madhusudan and Sanjit A. Seshia (Eds.). Springer, 744–750. https://doi.org/10.1007/978-3-642-31424-7_59

[24] Patrick Maxim Rondon, Ming Kawaguchi, and Ranjit Jhala. 2008. Liquid types. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*, Rajiv Gupta and Saman P. Amarasinghe (Eds.). ACM, 159–169. https://doi.org/10.1145/1375581.1375602

[25] Ori Roth. 2021. Fluent API: Practice and theory. https://blog.sigplan.org/2021/03/02/fluent-api-practice-and-theory/

[26] Miro Samek. 2002. *Practical statecharts in C/C++: Quantum programming for embedded systems.* CRC Press.

[27] Michael Sammler, Rodolphe Lepigre, Robbert Krebbers, Kayvan Memarian, Derek Dreyer, and Deepak Garg. 2021. RefinedC: Automating the Foundational Verification of C Code with Refined Ownership Types. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 158–174. https://doi.org/10.1145/3453483.3454036

[28] Georg Stefan Schmid and Viktor Kuncak. 2016. SMT-based checking of predicate-qualified types for Scala. In *Proceedings of the 7th ACM SIGPLAN Symposium on Scala, SCALA@SPLASH 2016, Amsterdam, Netherlands, October 30 - November 4, 2016*, Aggelos Biboudis, Manohar Jonnalagedda, Sandro Stucki, and Vlad Ureche (Eds.). ACM, 31–40. https://doi.org/10.1145/2998392.2998398

[29] Robert E. Strom and Shaula Yemini. 1986. Typestate: A Programming Language Concept for Enhancing Software Reliability. *IEEE Trans. Software Eng.* 12, 1 (1986), 157–171. https://doi.org/10.1109/TSE.1986.6312929

[30] Joshua Sunshine, Karl Naden, Sven Stork, Jonathan Aldrich, and Éric Tanter. 2011. First-class state change in plaid. In *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011, part of SPLASH 2011, Portland, OR, USA, October 22 - 27, 2011*, Cristina Videira Lopes and Kathleen Fisher (Eds.). ACM, 713–732. https://doi.org/10.1145/2048066.2048122

[31] Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon L. Peyton Jones. 2014. Refinement types for Haskell. In *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming, Gothenburg, Sweden, September 1-3, 2014*, Johan Jeuring and Manuel M. T. Chakravarty (Eds.). ACM, 269–282. https://doi.org/10.

1145/2628136.2628161

[32] Panagiotis Vekris, Benjamin Cosman, and Ranjit Jhala. 2016. Refinement types for TypeScript. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, Chandra Krintz and Emery D. Berger (Eds.). ACM, 310–325. https://doi.org/10.1145/2908080.2908110

[33] Hongwei Xi and Frank Pfenning. 1998. Eliminating Array Bound Checking Through Dependent Types. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation (PLDI)*. ACM, 249–257. https://doi.org/10.1145/277650.277732