Karlsruhe Institute of Technology

More Efficient Property Types with SMT-Decidable Assertions

Bachelor's thesis of

Johann Zuber

At the KIT Department of Informatics

KASTEL – Institute of Information Security and Dependability

First examiner:	Prof. Bernhard Beckert
First advisor:	Florian Lanzinger, M.Sc.

11. November 2024 – 11. March 2025

Karlsruher Institut für Technologie
Fakultät für Informatik
Postfach 6980
76128 Karlsruhe

Abstract

Property type systems extend static type systems by allowing users to refine types with logical predicates. *Kukicha* is a novel two-step method to implement property types in object-oriented programming languages by combining efficient type rules with the power of deductive verification. However, the Kukicha type rules so far are only capable of verifying a small subset of property types. Support for *dependent types* in particular is very limited and requires the use of deductive verification, taking away from the simpler type rules' convenience. We introduce a new component to the Kukicha method that can automatically verify a subset of property types that are out of scope for the type rules but also don't require the overhead of deductive verification. We formalise our extension within the theoretical framework of Kukicha and prove that it is sound. We further implement our extension in the *Property Checker*, the Kukicha implementation for Java, by leveraging *SMT solving* tools. Finally, we examine the impact of our work using a small case study.

Zusammenfassung

Property-Typsysteme erweitern statische Typsysteme um die Möglichkeit, Typen mit logischen Prädikaten einzuschränken. *Kukicha* ist eine neuartige zweistufige Methode zur Implementierung von Property-Typen in objektorientierten Programmiersprachen, die effiziente Typregeln mit der Stärke von deduktiver Verifikation kombiniert. Allerdings können die Typregeln bisher nur eine kleine Teilmenge von Property-Typen verifizieren. Insbesondere die Unterstützung von *dependent types* ist sehr begrenzt und erfordert in den meisten Fällen den Einsatz von deduktiver Verifikation, was die Vorzüge der simpleren Typregeln schmälert. Wir führen eine neue Komponente in die Kukicha-Methode ein, die automatisch eine Teilmenge von Property-Typen verifizieren kann, für die die Typregeln allein nicht ausreichen, aber die auch nicht den vollen zusätzlichen Aufwand von deduktiver Verifikation erfordern. Wir formalisieren unsere Erweiterung innerhalb des theoretischen Rahmens von Kukicha und beweisen ihre Korrektheit. Außerdem implementieren wir unsere Erweiterung im *Property Checker*, der Kukicha-Implementierung für Java, indem wir Werkzeuge aus dem Bereich des *SMT-Solving* nutzen. Schließlich untersuchen wir die Auswirkungen unserer Arbeit anhand einer kleinen Fallstudie.

Contents

1. Introduction

Static type systems are one of the most prolific approaches to guaranteeing certain safety and correctness properties about the run-time behaviour of computer programs. Many modern programming languages include a type system that can at least ensure that the value of an expression has a specific structure. However, static type systems are usually limited to assertions about the coarse-grained structure of data rather than concrete values and relationships. For example, most static type systems allow us to specify a structure consisting of two numbers to describe a point on the unit circle (x, y), but do not let us model the required restrictions $x, y \in [-1, 1]$ and $|x + y| = 1$.

There are some exceptions to this. Programming languages like *Lean 4* are built around type systems that are expressive enough to model examples like the one above [1]. However, due to their academic background and a high focus on interactive theorem proving, these languages have no relevance in conventional software development compared to well-established industry languages like *Java* [2].

Property types aim to improve the expressivity of conventional static type systems by extending them with a mechanism to *refine* types, i.e. constrain them further via logical predicates [3]. Properties are associated with nominal *property qualifiers* that developers can use to annotate type uses in their programs. A common example referenced throughout this work is the `Interval` qualifier which can be attached to an numeric type to constrain its value space to an arbitrary range.

The idea behind property types, i.e. the attachment of arbitrary predicates to types, can be actualised on a lower level using formal verification, e.g. *KeY* for Java [4]. Users of such tools must first provide a formal specification for the functions in their program they want to verify. Then, an external verifier program symbolically evaluates the code in those functions, and, with the use of some calculus, determines whether their behaviour matches the specification. This approach can be much more powerful than a type system, but also requires significantly more effort on the user's end.

F. Lanzinger *et al.* [3] introduce a method for implementing property types for object-oriented languages dubbed “Kukicha”. This method combines classical type systems with deductive verifiers in order to find a middle ground between the ease of use of static type systems and the power of formal verification. Using this method, a program with property type annotations is verified by a multi-step process:

1. The program is type-checked as usual to make sure it is well-typed in terms of the base type system.

2. A set of custom type rules that approximate the semantics of property types is applied to the program, producing type judgments.
3. The type judgments are translated to assumptions (well-typed judgments) and assertions (ill-typed judgments) for a deductive verifier and added as annotations to the program.
4. The resulting, annotated version of the program is plugged into a deductive verifier with the goal to verify the correctness of the use of property types in full.

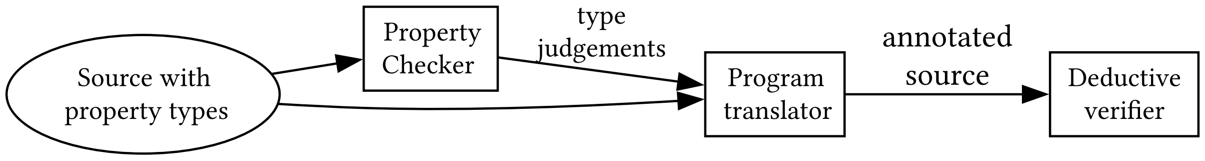

Figure 1: The Kukicha verification pipeline

The work of F. Lanzinger *et al.* [3] is the basis for the work of this thesis. We aim to extend the Kukicha method by introducing a novel step in the process outlined above. Like before, a type system is used to prove a subset of the assertions that arise from the original property types. However, unlike before, when this leaves a remainder of unproven assertions, they are not directly passed on to a deductive verifier whose success is generally undecidable. Instead, the assertions are put through a layer that attempts to derive a sufficient condition for the assertion to then prove it in a decidable system. In practice, this will be achieved using *satisfiability modulo theories* (SMT).

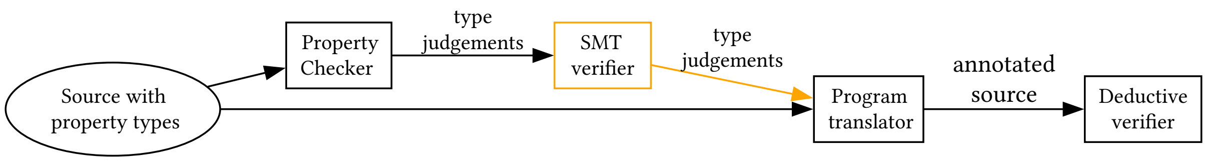

Figure 2: The Kukicha verification pipeline, extended with a step to decrease the number of type errors before deductive verification

Over the course of this work, we aim to answer the following questions:

(RQ1) Can the Kukicha method be extended with a mechanism for validity-based deduction of property types?

(RQ2) Does the SMT-based implementation of a property type deduction mechanism in the Property Checker lead to a significant reduction of type errors?

Our contributions are a theoretical extension of Kukicha (Chapter 2) and a software extension of the Property Checker application [5], which is described in Chapter 3.

2. Extending Field-Assignment-Language

Field-Assignment-Language (FAL) is a theoretical object-oriented programming language that acts as the basis for the theory behind the Kukicha method. Its syntax and semantics are limited to the aspects that are the most relevant for the formalisation of Kukicha [3]. This chapter focuses on FAL and Kukicha's theory; the transfer to a practical implementation happens in Chapter 3.

F. Lanzinger *et al.* [3] define the concept of *property type hierarchies*, which are user-specified semilattices of property types. These hierarchies are used in a subtyping rule to make the type system more powerful.

While simple to implement, this approach has two shortcomings:

1. Hierarchies must specified manually in practice, despite there being a natural definition for what it means for two property types to be in a subtype relationship.
2. Hierarchies do not fully support dependent types. Dependent types are only comparable to the top type $\top$ and themselves.

We will alleviate the first shortcoming by *inferring* property hierarchies based on a black-box function $valid$ that decides the universal validity of logical formulae with respect to the underlying language's semantics.

The second point is a limitation of using a simple semilattice, since determining the relationship of dependent types A and B intuitively requires information about the values of their *dependencies*. We will construct a mechanism to achieve this in a second step, and unify the result with inferred hierarchies.

As a prerequisite to these additions, we begin this section by summarising the work of F. Lanzinger *et al.* [3] and introducing the fundamentals of the Kukicha method that are required to understand the rest of this work.

2.1. Fundamentals

The Kukicha method consists of two main components: type checking and deductive verification. The type checking process involves three type systems, built on top of each other [3].

Property types A property type is nominal *type refinement*, i.e. a named constraint on an existing *base type*. The names of these refinements are called *property qualifiers*. Subtyping relationships between property types can be defined using *property type hierarchies*. The property type system ensures that values in the program fulfil the properties expected of them by assignments or method parameters. The packing and uniqueness type systems are the basis for property types.

Packing types The packing type system keeps track of which fields satisfy their declared property type. This allows for temporary violations of property types, for example where modifying the state of an object temporarily leads to inconsistencies between mutually-dependent fields. In the packing type system, each field is either *committed* or *uncommitted*, depending on the packing type of the object and which class the field is declared in. To violate the declared type of a field, its owner must first be *unpacked up to* the field's declaring class.

Uniqueness types The uniqueness type system assigns one of three types to every reference expression: Unique, $\mathit{MaybeAliased}$, or $\mathit{ReadOnly}$, with their relationship being $\mathit{Unique} \preceq \mathit{MaybeAliased} \preceq \mathit{ReadOnly}$. The fields of $\mathit{ReadOnly}$ references are always uncommitted but cannot be reassigned. $\mathit{MaybeAliased}$ references only allow the modification of uncommitted fields. Unique references can only have $\mathit{ReadOnly}$ aliases and can therefore be *unpacked* to be mutated without affecting the safety of code holding aliases.

For this work, the primary focus is the property type system. The lower-level uniqueness and packing type systems also play a role in certain places, but for the most part we operate on the more abstract level of properties and the associated refinements. That said, what we develop in this work is an *extension*: we make no direct changes to any of the type systems. This has the advantage that the soundness proofs from F. Lanzinger *et al.* [3] remain valid.

What follows are some relevant definitions, copied almost verbatim from F. Lanzinger *et al.* [3].

Definition 1 (Property types). *A* property qualifier *q is a tuple* $\left(\beta_q, \mathit{Args}_q, \mathit{Wf}_q, \mathit{Refin}_q\right)$ *where*

1. *β_q is a type from the base type system.*
2. *Args_q is a list of base-typed identifiers $a_q^i : \beta_q^i$.*
3. *Wf_q is a formula whose only free variables are Args_q. In addition, all field accesses must be on the respective variable's abstract state.*
4. *Refin_q is a formula whose only free variables are Args_q and $\mathit{subject} : \beta_q$. In addition, all field accesses must be on the respective variable's abstract state.*

Wf_q and Refin_q are formulae in typed quantifier-free first-order logic over types $\mathfrak{B}$, universe $\mathfrak{U}$, and evaluation function eval.
A property type $\rho = q(a_1, ..., a_n)$ consists of a qualifier q and expressions a_i such that $\mathit{decl}_B(a_i) = \beta_q^i$. We write Wf_ρ for $\mathit{Wf}_q\left[a_q^i / a_i\right]$ and Refin_ρ for $\mathit{Refin}_q\left[a_q^i / a_i\right]$.

$\mathfrak{B}$ are the base types in the underlying programming language (FAL), $\mathfrak{U}$ is their combined set of values. decl_B is the function that maps an expression to its base type. We omit the formal definition for *abstract state* here; this restriction exists for the purpose of consistency between the packing and property type systems, the details of which has little relevance for the rest of this work.

Definition 2 (Property type viewpoint adaptation). *Given a property type $\rho = q(r_1, ..., r_n)$, expressions $a_0, ..., a_n$ and a method m with formal parameters $x_1, ..., x_n$, the* adapted type *$a_j \overset{m}{\blacktriangleright} \rho$ is defined as*

$$a_j \overset{m}{\blacktriangleright} \rho := q(r_1[this/a_0; x_1/a_1; ...; x_n/a_n], ..., r_n[this/a_0; x_1/a_1; ...; x_n/a_n])$$

Given a single expression e, the adapted type $e \blacktriangleright \rho$ is defined as

$$e \blacktriangleright \rho := q(r_1[this/e], ..., r_n[this/e])$$

The viewpoint adaptation operators are useful to adapt types from method and field declarations to their concrete uses by replacing *this* with the actual receiver or replacing formal parameter references with the actual arguments of a method call.

Definition 3 (Property type hierarchy). *A* property type hierarchy *is a semilattice of property types such that*

1. *For the top element $\top$, we have $Wf_\top = Refin_\top = true$ and $\beta_\top = any$.*
2. *There is no type ρ in the semilattice for which Wf_ρ is an invalid closed formula.*
3. *Any dependent type, i.e., any type ρ for which Wf_ρ is not a closed formula or $Refin_\rho$ contains free variables other than $subject$, is only comparable to $\top$ and itself.*
4. *For any two independent types such that $\rho_0 \preceq \rho_1$, the following formula is valid:*

$$\beta_{\rho_0} \preceq \beta_{\rho_1} \land \forall\ subject : \beta_{\rho_0}.Refin_{\rho_0} \rightarrow Refin_{\rho_1}$$

F. Lanzinger *et al.* [3] build a set of type rules for property types around this notion of hierarchies. We won't copy them here for the sake of brevity, however they lead to another core concept of the Kukicha method: the difference between syntactic well-typedness and semantic well-typedness.

An expression e is *syntactically well-typed* if its expected type ρ can be deduced using the property expression type rules and a property type hierarchy. In that case, we write $\Gamma \vdash e : \rho$, where Γ is the type context. On the other hand, e is *semantically well-typed* if it actually conforms to its property type in every possible program state that conforms to the type context Γ. I.e., the well-formedness condition Wf_ρ and property refinement $Refin_\rho$ are fulfilled in any actual program state (denoted σ, l) for which Γ is a valid type context. In this case, we write $\Gamma \vDash e : \rho$. The formal definition of program states and what it means for a state to conform to a type context can be found in F. Lanzinger *et al.* [3].

Crucially, syntactic well-typedness implies semantic well-typedness, but not the other way around. That is, a property type hierarchy may be insufficient to deduce that all the property types in a program are correct. An obvious example for where this can be the case is places where dependent types are used, because there is never a relationship between two different dependent types. For this reason, the property type system produces an error context that is amended after every statement, for each ill-typed expression that occurs in it. This error awareness is part of the the syntactic *statement typing* rules that have both a pre- and a post-context: $\Gamma, \mathcal{E} \vdash s \dashv \Gamma', \mathcal{E}'$. $\mathcal{E}$ is a set containing all the ill-typed expressions $\Gamma \nvdash e : \rho$ encountered up to statement s, $\mathcal{E}' \setminus \mathcal{E}$ is the set of errors added by s. If a statement does not add any new errors (i.e. $\mathcal{E} = \mathcal{E}'$), it is syntactically well-typed. Again, syntactic well-typedness for statements implies the analog concept of semantic well-typedness: $\Gamma \vDash s \dashv \Gamma'$.

After the syntactic statement type rules have been applied and we have a set of type errors for every statement, we translate the program to a form where all syntactic type errors $\Gamma \nvdash e : \rho$ result in the addition of `assert` $e : \rho$ statements before the statement that needs this condition to be semantically well-typed. E.g. for an assignment to be semantically well-typed, the type of the right-hand value must match the type of the assigned-to field or variable, which can be expressed as an assertion before the assignment.

The translated version of the program is *semantically equivalent* to the original program, i.e. the original program is semantically well-typed iff the translation is semantically well-typed. We can then use a deductive verifier on the translated, annotated program to dismiss the syntactic type errors and thus ensure the semantic well-typedness of the original program.

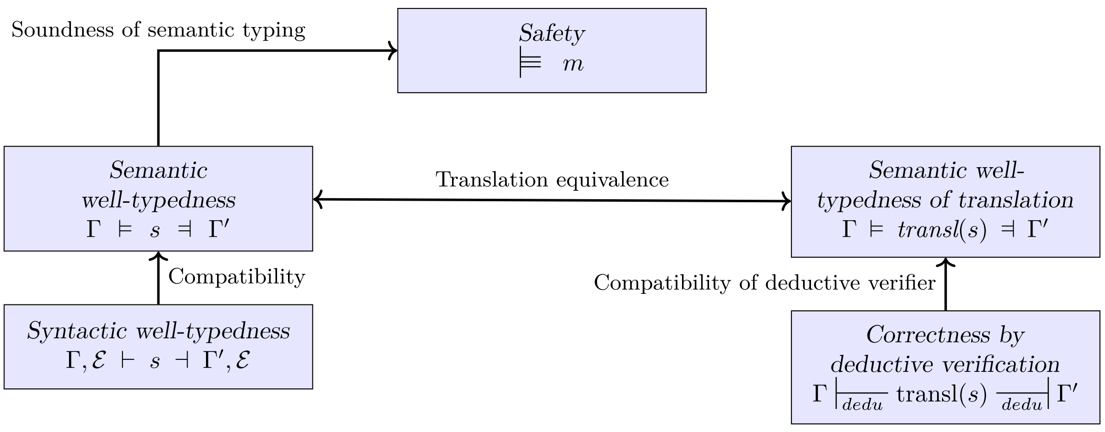
Soundness of semantic typing
Safety $\models m$
Semantic well-typedness $\Gamma \vDash s \dashv \Gamma'$
Translation equivalence
Semantic well-typedness of translation $\Gamma \vDash transl(s) \dashv \Gamma'$
Compatibility
Compatibility of deductive verifier
Syntactic well-typedness $\Gamma, \mathcal{E} \vdash s \dashv \Gamma', \mathcal{E}$
Correctness by deductive verification $\Gamma \vdash_{dedu} \text{transl}(s) \dashv_{dedu} \Gamma'$

Figure 3: Relationship between the type checker and the deductive verification in Kukicha [3].

2.2. Validity modulo language semantics

The Kukicha method for implementing property types consists of two components: a property type system and a deductive verifier [3]. The mechanisms these components use for verification are type rules and a calculus for the specific programming language, respectively.

The contribution of this work is a new component in the Kukicha method that aims to improve the type systems accuracy by reducing the number of false negatives, i.e. type judgements that are syntactically ill-typed but semantically well-typed. The vehicle of choice for this extension is the *validity* problem, where we ask if a boolean formula F is valid in all possible interpretations of its contained functions and variables, i.e. $\vDash F$.

The formulae we are interested in are the property conditions and well-formedness conditions of property types. E.g., to prove that a variable x has the property type $P(2, y)$, we claim that it is sufficient to show the following:

1. $\beta_x \preceq \beta_P$, i.e. the base type of x is a subtype of the base type of qualifier P,

2. $Wf_{P(2,y)}$ is universally valid, i.e. the property type is well-formed for the arguments 2 and y, and
3. $Refin_{P(2,y)}[subject/x]$ is universally valid, i.e. the refinement of the property type holds for the subject x.

The basic idea behind our work is this: if we had access to a function that *decides* the validity problem and therefore lets us answer point 2 and 3, we would get a new mechanism to check (dependent) property types that is different from both subtyping rules and full deductive verification.

An important premise for this argument is that the validity problem is *decidable*. This, in turn, depends on the type of logic that is allowed in property refinements and well-formedness conditions. The definition of property types restricts these to formulae in quantifier-free, typed first-order logic. While full first-order logic is generally undecidable [6], quantifier-free first-order logic *is* decidable[1] and therefore usable for our extension.

Definition 4 (valid function). *The predicate valid takes a typed, quantifier-free formula in first-order logic and determines whether it is universally valid, i.e.*

$$valid(F) :\Leftrightarrow F \text{ is universally valid.}$$

With the following lemma, we formalise the connection between universal validity and semantic well-typedness mentioned earlier.

Lemma 1 (Validity implies semantic well-typedness). *If a property refinement $Refin_\rho[subject/e]$ and well-formedness condition $Wf_\rho[subject/e]$ are universally valid, they are valid in any program state σ. By extension, $e : \rho$ must then be semantically well-typed in any type context Γ.*

$$\begin{aligned} &valid\big(\big(Refin_\rho \land Wf_\rho\big)[subject\ /e]\big) \\ &\Rightarrow \sigma \vDash \big(Refin_\rho \land Wf_\rho\big)[subject\ /e] \\ &\Rightarrow \Gamma \vDash e : \rho \end{aligned}$$

Proof. Let F be a formula with $valid(F)$, let σ, l be a program state and let Γ be a type context. If F is *universally* valid, it is also valid for the specific interpretation derived from σ. In particular, if F is valid for all possible interpretations of the evaluation function *eval*, it is especially valid for $eval = eval_\sigma$, where $eval_\sigma$ is the evaluation function in state σ [3]. Therefore, $\sigma \vDash \big(Refin_\rho \land Wf_\rho\big)[subject\ /e]$. Since this is true for an arbitrary σ, it is also true for all states σ, l that conform to Γ [3], hence $\Gamma \vDash e : \rho$. ∎

FAL has virtually no expression syntax; a FAL expression is either a field access or a local variable [3]. For more complex languages, it is useful to amend the definition of *valid* slightly by restricting validity to a logic modelling the language's expression semantics. *valid* would thus determine validity *modulo language semantics* rather than validity across *all* interpretations. This can make the function more effective as a tool for type checking.

[1] Formulae in quantifier-free first-order logic with equality can be reduced to equality formulae [7, Ch. IX], which are decidable [7, Ch. III].

For instance, when our programming language allows arithmetic expressions and comparisons like $x + 1 - 1 = x$, the $valid$ function should interpret these builtin operations “as expected”, e.g. using a theory of integers. If $+$ and $-$ in this example were not bound by a theory and instead left uninterpreted, the formula would not be universally valid even though in the semantics of our language it would be. It is crucial here that any theory used does not conflict with the real semantics. Again, take integer arithmetic as an example: the formula $x + 1 > x$ is universally valid modulo the standard theory of integers, but *not* universally valid in two’s complement semantics, where the addition of 1 to an integer could lead to an overflow.

In practice (Chapter 3), we implement a $valid$ function using *satisfiability modulo theories* (SMT). SMT is a versatile tool that is used for many similar applications [8], [9], [10]. Since satisfiability is the dual concept to validity, $valid$ can easily be defined in terms of sat (the function deciding satisfiability): $valid(F) = \neg(sat(\neg F))$. Our approximation of Java’s expression semantics using standard theories is discussed in detail in Section 3.4, with notes on decidability specifically in Section 3.4.5.

2.3. Inferring property type hierarchies

The basis for the property type system in Kukicha is the property type hierarchy. In this section, we integrate our new validity-based type-checking mechanism (Section 2.2) into the concept of hierarchies.

Definition 5 (Inferred property type hierarchy). *We call the semilattice* $\underset{valid}{\preceq}$ *the* inferred property type hierarchy. *It is defined as follows.*

For two independent property types ρ_0, ρ_1*:*

$$\rho_0 \underset{valid}{\preceq} \rho_1 :\Leftrightarrow Wf_{\rho_0} \wedge Wf_{\rho_1} \wedge \beta_{\rho_0} \preceq \beta_{\rho_1} \wedge valid\Big(Refin_{\rho_0} \to Refin_{\rho_1}\Big)$$

“Independent” means that the only free variable in both $Refin_{\rho_0}$ *and* $Refin_{\rho_1}$ *is* subject.

Furthermore, for any property type ρ*, including any dependent type, we specify* $\rho \underset{valid}{\preceq} \rho$ *and* $\rho \underset{valid}{\preceq} \top$*, where* $\top$ *is the top type with* $\beta_\top = any$ *and* $Wf_\top = Refin_\top = true$*.*

Corollary 1. $\underset{valid}{\preceq}$ *constitutes a valid property type hierarchy as defined by F. Lanzinger et al. [3].*

We now show that a hierarchy inferred from $valid$ is as powerful as any hierarchy that could be specified manually.

Theorem 1 (Validity subsumes property type hierarchies). $\underset{valid}{\preceq}$ *is the most general property type hierarchy. That is, given any property type hierarchy* $\preceq$ *and property types* ρ_0, ρ_1*:*

$$\rho_0 \preceq \rho_1 \Rightarrow \rho_0 \underset{valid}{\preceq} \rho_1$$

Proof. If one of ρ_0 or ρ_1 is a dependent type, the implication holds by definition of property type hierarchies.

If both types are independent:

$$\begin{aligned}
&\rho_0 \preceq \rho_1 \\
\Rightarrow\ & Wf_{\rho_0} \wedge Wf_{\rho_1} \wedge \beta_{\rho_0} \preceq \beta_{\rho_1} \wedge \forall\ subject : \beta_{\rho_0}\ .\ Refin_{\rho_0} \Rightarrow Refin_{\rho_1} \\
\Rightarrow\ & Wf_{\rho_0} \wedge Wf_{\rho_1} \wedge \beta_{\rho_0} \preceq \beta_{\rho_1} \wedge valid\left(Refin_{\rho_0} \rightarrow Refin_{\rho_1}\right) \\
\Rightarrow\ & \rho_0 \underset{valid}{\preceq} \rho_1
\end{aligned}$$

∎

What this means is that, given a function *valid*, manually specifying a property type hierarchy is not necessary anymore. Note that this does not remove the concept of type hierarchies, it only simplifies their specification. The property type rules [3] do not change, but we can always specify $\underset{valid}{\preceq}$ as the hierarchy to use to make them as precise as possible.

2.4. Comparing dependent types

We will now extend the notion of property subtypes to support dependent types.

Example 1: dependent types

To illustrate the problem with dependent types, consider the following pseudocode example. Our hypothetical language here, in contrast to FAL, supports integer arithmetic expressions.

```
@gte(5) int x = 6
@gte(4) int y = x;
@gte(y) int z = x + y;
```

`gte` (“greater than or equal to”) is a property qualifier formally defined as the tuple $(\beta, \text{Args}, Wf, Refin)$, where

- $\beta = \text{int}$
- $\text{Args} = (a : \text{int})$
- $Wf = \text{true}$
- $Refin = subject \geq a$

The most general property hierarchy on property types with only the qualifier `gte` is intuitively $\text{gte}(a) \preceq \text{gte}(b) \Leftrightarrow a \geq b$.

Specifying this as the hierarchy to use for type checking, we can easily verify that the second assignment is syntactically well-typed, since $\text{gte}(5) \preceq \text{gte}(4)$. However, we cannot do the same for the third assignment, because we cannot compare the declared type $\text{gte}(y)$ to anything, *even if* we knew a specific type for `x + y`.

Notwithstanding, we can construct a sufficient condition for $\Gamma \vDash x + y : \mathrm{gte}(y)$:

$$\begin{aligned} & x \geq 5 \wedge y \geq 4 \rightarrow (x+y) \geq y \\ \equiv\ & \mathit{Refin}_{\mathrm{gte}(5)}[\mathit{subject}/x] \wedge \mathit{Refin}_{\mathrm{gte}(4)}[\mathit{subject}/y] \rightarrow \mathit{Refin}_{\mathrm{gte}(y)}[\mathit{subject}/(x+y)] \end{aligned}$$

We can determine the validity of this formula using a *valid* function that respects the integer semantics of this language. Furthermore, the known types of x and y act as constraints for the type goal by being on the left side of the implication. This is the basic idea behind the dependent type handling introduced in this section.

Definition 6 (Local refinement context). *Given a type context* Γ, *we define the* local refinement context

$$C_\Gamma := \left\{ r(e, \rho) \mid e \in \bigcup_{x \in\, \mathit{locals}(\Gamma)} \mathit{reach}(\emptyset, x),\ \Gamma \vdash e : \rho \right\}$$

where $\mathit{locals}(\Gamma)$ *is the set of parameters and local variables in the context of* Γ, *and reach and r are defined as*

$$\mathit{reach}(V, e) := \{e\} \cup \bigcup_{\substack{f \in\, \mathit{fields}(\beta_e), \\ (\beta_e, f) \notin V}} \mathit{reach}(V \cup \{(\beta_e, f)\}, e.f)$$

$$r(e, \rho) := \begin{cases} (\mathit{Wf}_{e' \blacktriangleright \rho} \wedge \mathit{Refin}_{e' \blacktriangleright \rho})[\mathit{subject}/e] & \text{if } e = e'.f, \\ (\mathit{Wf}_\rho \wedge \mathit{Refin}_\rho)[\mathit{subject}/e] & \text{otherwise} \end{cases}$$

In essence, $\mathit{reach}(V, e)$ is an edge traversal of a graph of base types connected by field relationships. It returns a set of expressions consisting of the expression e and all fields that are (transitively) reachable from e. The parameter V is used to keep track of what fields have already been processed, so that the set is always finite even in the case of a cyclic data structure.

$r(e, \rho)$ constructs the logical formula that must be fulfilled for $e : \rho$ to be semantically well-typed. The notation β_e in the above definition refers to the base type of expression e. These are made available by the underlying base type system of the language.

Example 2: local refinement context

Consider the following example of two classes and the property qualifiers `P`, `Q`, and `R`:

```
class A {
  @Q(this.h) Int g;
  Int h;
  B i;
}

class B {
  @P(this.g) A f;
  void foo(B this, @Q(y) Int x, @R Int y) {
    // Context here
  }
}
```

For context Γ inside method `foo`, the local refinement context is

$$\begin{aligned} C_\Gamma = \Big\{ & true, \\ & p_{\texttt{this.f}}(\texttt{this.g}), \\ & q_{\texttt{this.f.g}}(\texttt{this.f.h}), \\ & p_{\texttt{this.f.i.f}}(\texttt{this.f.i.g}), \\ & q_{\texttt{this.f.i.f.g}}(\texttt{this.f.i.f.h}), \\ & q_{\texttt{x}}(\texttt{y}), \\ & r_{\texttt{y}} \Big\} \end{aligned}$$

The notation $p_a(b)$ is used for brevity and stands for $\left(Wf_{P(b)} \wedge Refin_{P(b)}\right)[subject/a]$ (analogous for `Q` and `R`).

Theorem 2 (Consistency of the local refinement context). *For any type context Γ, all formulae in C_Γ are valid in any program state σ, l that conforms to Γ, i.e.*

$$\forall f \in C_\Gamma : \Gamma \vDash f$$

Furthermore, for any expression e and property type ρ, the following holds:

$$valid\left(\bigwedge_{f \in C_\Gamma} f \rightarrow r(e, \rho)\right) \Rightarrow \Gamma \vDash e : \rho$$

Proof. Let $f = r(e, \rho) \in C_\Gamma$. Per definition of C_Γ, $\Gamma \vdash e : \rho$, i.e. $e : \rho$ is syntactically well-typed. This implies that $e : \rho$ is also semantically well-typed, i.e. $\Gamma \vDash e : \rho$ [3]. If that is the case, the well-formedness condition Wf_ρ and property condition $Refin_\rho$ must hold for subject e. This is exactly what $r(e, \rho)$ describes, so $\Gamma \vDash r(e, \rho)$.

Now, let $valid\left(\bigwedge_{f \in C_\Gamma} f \rightarrow r(e, \rho)\right)$ for some e, ρ. This means that the formula $\bigwedge_{f \in C_\Gamma} f \rightarrow r(e, \rho)$ is universally valid modulo FAL semantics. Thus, we have two cases:

1. $valid\left(\neg \bigwedge_{f \in C_\Gamma} f\right) \Rightarrow \exists f \in C_\Gamma : \neg\, valid\ (f) \Rightarrow \exists f \in C_\Gamma : \Gamma \nvDash f.$
2. $valid(r(e, \rho)) \Rightarrow \Gamma \vDash r(e, \rho) \Rightarrow \Gamma \vDash e : \rho.$

The first case is contradictory to what we've shown in the first step. Thus, we must be in the second case. ∎

Theorem 2 shows that a type judgement is semantically valid if it can be proven valid in the local refinement context. In the next step, we use this property to show that we can remove syntactic type errors from the error context (a process we call *mending*) without changing the outcome of deductive verification.

Definition 7 (Translation with mending). *F. Lanzinger et al. [3] define a function $transl$ that annotates statements with* assumptions *and* assertions *derived from the type context. We now define an alternative version of this function, $transl_m$, which does the same in principle but potentially results in more assumptions than $transl$.*

Given an atomic statement s, we define $transl_m(s) := a_0; s; a_1$.

The statement a_0 is a sequence of `assert`*/*`assume` *statements defined as follows.*
Consider the judgement $\Gamma, \mathcal{E} \vdash s \dashv \Gamma', \mathcal{E}'$, and let

$$\text{Errors} = \left\{ (\Gamma \nvdash e : \rho) \in (\mathcal{E}' \setminus \mathcal{E}) \mid \neg\ valid\left(\bigwedge_{f \in C_\Gamma} f \Rightarrow r(e, \rho) \right) \right\}$$

Then, for every premise of the form $\Gamma \vdash^* e : \rho$,

1. *if $(\Gamma \nvdash e : \rho) \in$ Errors, add* `assert` *$e : \rho$ to a_0*
2. *if $(\Gamma \nvdash e : \rho) \notin$ Errors, add* `assume` *$e : \rho$ to a_0*

The statement a_1 is defined exactly as in F. Lanzinger et al. [3], i.e. it is empty if s is not a method call and a sequence of `assume` *statements corresponding to the posttypes of the invoked method otherwise.*

We define the translation of methods m analogously to F. Lanzinger et al. [3].

In F. Lanzinger *et al.* [3], the correctness of the translation function plays a key role in the soundness of the entire Kukicha method. Its purpose is to translate the program to an annotated form that can be interpreted by a deductive verifier. In order for the verifier's result to have any bearing on the correctness of the original program, the translated program must be semantically equivalent to the original program. This is referred to as *translation equivalence*. In the following, we show translation equivalence for $transl_m$.

Theorem 3 (Translation equivalence). *For any well-typed context Γ and atomic statement s, the following propositions are equivalent:*

(1) $\Gamma \vDash s \Dashv \Gamma'$
(2) $\Gamma \vDash^* transl_m(s) \ {}^*\!\!\Dashv \Gamma'$

That is, $\text{transl}_m(s)$ *is semantically well-typed if and only if s is semantically well-typed.*

The syntax $\Gamma \vDash^* s \ {}^*\!\!\Dashv \Gamma'$ is taken from F. Lanzinger et al. [3]: it describes a type judgement where all declared property types are replaced by the top type, to ensure that the property type information is fully and independently encoded in the assumptions and assertions inserted in the translation.

Proof. Let s be an atomic statement, and let Γ be a well-typed context.

For the direction (1) $\Rightarrow$ (2), assume that $\Gamma \nvDash^* transl_m(s) \ {}^*\!\!\not\Dashv \Gamma'$. Since all declared property types in this judgment are $\top$, it cannot be the case that _ $\nvDash^* s \ {}^*\!\!\not\Dashv$ _. That is, s itself must be semantically well-typed. Thus, it must be the case that $\Gamma \nvDash^* \alpha \ {}^*\!\!\not\Dashv$ _ for some inserted assertion or assumption α.

If $\alpha = \texttt{assert}\ e : \rho$, we know that it was added because $\Gamma \vDash e : \rho$ is a necessary condition for $\Gamma \vDash s \dashv \Gamma'$. Thus, it not holding would make s unsafe.

If $\alpha = \texttt{assume}\ e : \rho$, we know that it was added because either $\Gamma \vdash e : \rho$ or $\mathit{valid}\Big(\bigwedge_{f \in C_\Gamma} f \to r(e, \rho)\Big)$. In the first case, $\Gamma \vDash \alpha \dashv \_$ follows from the compatibility of syntactic and semantic well-typedness. In the second case, per Theorem 2: $\mathit{valid}\Big(\bigwedge_{f \in C_\Gamma} f \to r(e, \rho)\Big) \Rightarrow \Gamma \vDash e : \rho$, and thus $\Gamma \vDash \alpha \dashv \_$.

Both cases lead to a contradiction. Thus, the premise $\Gamma \nvDash^{*} \mathit{transl}_m(s)\ ^{*}\!\not\dashv \Gamma'$ must be false

The other direction is analogous to the proof from F. Lanzinger *et al.* [3]. The only difference between $\mathit{transl}_m(s)$ and $\mathit{transl}(s)$ is that there are possibly assumptions $\texttt{assume}\ e : \rho$ that correspond to type judgments $\Gamma \nvdash e : \rho$. These judgments must be semantically well-typed for s to be semantically well-typed as well. But since $\texttt{assume}\ e : \rho$ is semantically well-typed as a consequence of $\Gamma \vDash^{*} \mathit{transl}_m(s)\ ^{*}\!\dashv \Gamma'$, we know that this is the case.

The proof of equivalence for non-atomic statements is analogous to the proof for *transl* from F. Lanzinger *et al.* [3]. ∎

With our equivalence-preserving modification of the translation function, we have successfully integrated a new mechanism to check dependent types into the Kukicha verification pipeline. Crucially, we have shown that this mechanism is compatible with property hierarchies on one end (via inferred hierarchies) and deductive verification on the other end (via mending translation). At its core, the choice of the *valid* function determines the power of the mechanism: the more formulae whose validity is decided by *valid*, the more potential for turning previous assertions into assumptions in the translation process.

Corollary 2 (Context across multiple property type systems). *The union of local refinement contexts across multiple property type systems is still a set of universally valid formulae. That is to say: when we want to prove the validity of a refinement for one property type system, we can use the existing knowledge from* all *available property type systems as context.*

With this, we can confidently give a positive answer to to *RQ1*:

(RQ1) Can the Kukicha method be extended with a mechanism for validity-based deduction of property types?

We have shown that our theoretical extension is sound and fits neatly between the existing property type system and deductive verification, both in terms of verification power and coherence in the cooperative verification pipeline.

3. Integrating SMT into the Property Checker

In the previous chapter, we extended the theoretical foundations of the Kukicha method with a mechanism to reduce the number of type errors. To put this into practice, we consequently extend the Property Checker [5], which is an implementation of the Kukicha method for the Java programming language built on the Checker Framework [3], [11].

The practical work was not limited to merely transferring the theoretical additions to code. Adjustments first had to be made in the existing implementation to support the analysis of dependent types, which had been missing from the implementation thus far. This was not an oversight but rather a simplification enabled by the limitations of dependent types in the property type rules [3].

Section 3.1 explains how we parse and work with dependent types on a structural level. Then, in Section 3.2, we describe how we use this structure to keep track of the validity of dependent types. Section 3.3 is concerned with the implementation of the local refinement context and how the context consistency theorem is applied to remove type errors. Section 3.4 details how we map property types to SMT. Finally, in Section 3.5, we briefly go over some changes we had to make to the Checker Framework in the process of our implementation.

3.1. Property refinement introspection

To specify refinements associated with property qualifiers in practice, a textual notation is required. The Property Checker application uses a custom language consisting of qualifier and subtyping relationship definitions. The boolean formulae for property and well-formedness conditions are specified in parameterised, JML-compatible Java expression syntax. For example, an `Interval` property qualifier for integers could be defined like this:

```
annotation Interval(int min, int max) int
    :<==> "§subject§ >= §min§ && §subject§ <= §max§"
    for "§min§ <= §max§";
```

The first line defines the typed parameter names $Args_{\text{Interval}} = (min : int, max : int)$ and the subject base type $\beta_{\text{Interval}} = int$. The second line defines the refinement $Refin_{\text{Interval}} = subject \geq min \land subject \leq max$, and the third line defines the well-formedness condition $Wf_{\text{Interval}} = min \leq max$.

These definitions are only treated as template strings until a property type's actual arguments and subject are substituted into them. For instance, for the following variable in Java:

```
@Interval(min="1", max="4") int x;
```

We get the actual property condition `"x >= 1 && x <= 4"` and well-formedness condition `"1 <= 4"`. No further analysis of these expressions is performed; they are passed directly to a JML-based deductive verifier, which then also performs the task of syntax validation.

For the extension developed in the context of this work, we need to be able to perform our own analysis of refinement expressions. Most notably, we need to translate refinements to a format that can be used as input for an SMT solver.

To analyse Java expressions like the ones above, we make use of the `JavaExpression` model implemented in the `dataflow.expression` package of the Checker Framework. This model implements a subset of actual Java expressions that we consider sufficient for our purposes. Its supported features include, among others: literals, variables, field accesses, and method calls [11]. The Checker Framework further provides mechanisms to parse strings into this representation via its `JavaExpressionParseUtil` utility class. This parsing is context-aware, meaning that parsed expressions are guaranteed to be type safe in terms of the base type system. For example, attempting to parse the expression `x > 5` in the context of a source code position where `x` is a variable of type `String` leads to a parsing error.

Example 3: property type representations

To illustrate the different representations of property formulae that are used throughout the type checking process, consider the property type `@Interval(min="1", max="x") int`.

1. At first, we have the actual arguments `1` and `x`, as well as the underlying property (`§subject§ >= §min§ && §subject§ <= §max§`) and well-formedness condition (`§min§ <= §max§`) represented as plain strings.
2. When the type is added to the type context, we parse property and well-formedness conditions into a combined `JavaExpression`:
   1. We combine property and well-formedness condition into a conjunction:
      `"(§min§ <= §max§) && (§subject§ >= §min§ && §subject§ <= §max§)"`
   2. We substitute the parameter placeholders `§min§` and `§max§` for their actual arguments, and `§subject§` for a synthetic variable `#1`:
      `"(1 <= x) && (#1 >= 1 && #1 <= x)"`
      The `#1` syntax is a Checker Framework extension of the Java expression syntax normally used to refer to method parameters.
   3. We parse the expression in the appropriate context (at the corresponding position in the method body if declared locally; at the field declaration level if declared on a field) and obtain a type-annotated abstract syntax tree for the expression.
3. To turn the type into a proof goal, we substitute the subject variable `#1` for the actual expression in the AST. E.g., the proof goal for the assignment
   `@Interval(min="1", max="x") int y = a + b;`

is:

```
(1 <= x) && ((a + b) >= 1 && (a + b) <= x)
```

The resulting concrete expression can then be analysed, translated to an intermediate SMT representation, and finally translated into SMT-LIB (see Section 3.4).

3.2. Dependent type analysis

Since property types can depend on program values (dependent types), care must be taken to invalidate types whose dependencies have changed as a result of a statement in the program. For example, a type like `@Interval(min="x", max="y") int` cannot be assumed to hold anymore when `x` or `y` are reassigned.

There are two ways values can change: assignment statements and impure method calls. The value of local variables and parameters can only change through assignments, the value of fields can change through both assignments and method calls. Whenever one of these operations is encountered in the source code, we must find and invalidate all types that depend on the changed field(s) or local variable.

In practice, we can only approximate a solution to this search. That is, certain dependent types may be invalidated by assignments or method calls even though their dependencies did not change. This is primarily due to two reasons.

First, the aliasing information we have from the uniqueness type system is limited. For example, we know that a `MaybeAliased` expression cannot be an alias of a `Unique` expression, but we cannot answer definitively whether a `MaybeAliased` expression is an alias of another `MaybeAliased` expression. This leads to the first approximation in our analysis: we consider a type dependent on a field access if it contains any field access whose receiver *could be* an alias of the dependency's receiver.

Second, method bodies are generally opaque. When a refinement contains a method call, e.g. `§subject§.foo() > 4`, we don't know what fields this method reads, so we simply assume it reads *every* field. This leads to the assumption that the return value of any method call could change if any field is reassigned anywhere. Similarly, we assume that all impure method calls reassign all fields they are allowed to reassign (also indirectly, through deeper method calls) given the restrictions of the uniqueness and packing type systems.

Put together, we heuristically consider an expression E dependent on local variable or field access expression D if one of the following is true:

1. E cannot be analysed due to parsing errors, e.g. when it uses features not supported by the Checker Framework `dataflow.expression` package.
2. E contains a verbatim reference to D.

3. D is a field access $e.f$, where e is an expression and f is a field, and, for any field access $e'.f$ in E, e and e' *could* be aliases according to the uniqueness type system. That is, **none of the following is true:**
   - e and e' are both `Unique`
   - e is `MaybeAliased` and e' is `Unique`
   - e is `Unique` and e' is `MaybeAliased`
4. D is a field access and E contains a method call.

Otherwise, E is not dependent on D[2].

Thus, to invalidate all types that depend on a given field or local variable, we go through all types stored in the type context and remove those whose refinement expression (conjunction of property and well-formedness conditions) depends on the field or variable according to the above heuristic.

Example 4: Dependent type invalidation

To illustrate the aliasing rules, take this simple example of a method with three parameters that have the same base type but different uniqueness types, and three local variables whose property types all depend on the same field of one of the parameters.

```java
public void foo(@Unique Bar a, @MaybeAliased Bar b, @ReadOnly Bar c) {
  @Prop("a.f") int x = 4;
  @Prop("b.f") int y = 5;
  @Prop("c.f") int z = 6;
  a.modifyFields();
}
```

Let's assume that the receiver of the method `modifyFields` in `Bar` is declared as `@Unique`, i.e. the method is allowed to unpack `a`. This means that, after the method call, all fields of `a`, the fields of those fields etc. could all have been assigned a new value, as far as the uniqueness and packing type systems allow it. E.g. if `a.f` is `Unique`, the field `a.f.g` could have changed too, whereas if `a.f` is `MaybeAliased`, `a.f.g` could only have been changed if `g` is not committed.

However, the implementation of `modifyFields` is opaque to our analysis, so we don't *know* what fields have actually changed. In this case, let's assume that `Bar` only has one `ReadOnly` field `f`, so `a.f` is the only field that can be reassigned by `modifyFields`. Again, we don't know that it *has* changed, but we assume it has in order to be conservative. As a result, the types of the local variables are changed by the method call as follows:

- `x` becomes the top type – it references `a.f` directly
- `y` does not change – `b` cannot be an alias of `a`, therefore `b.f` refers to a value that has not changed

[2]The proof sketch for this heuristic is to show that if E depends on D, at least one of the points 1-4 *must* be true. For there to be a dependency, E must contain an expression that references the same location in the heap as D. This can either be verbatim or through an alias of the receiver in the case of a field access. This is covered by points 2 and 3. The same can happen in any of the methods called, which is covered by point 4. Point 1 is there as a fallback to cover cases we can't analyse.

- `z` becomes the top type – `c` could be an alias of `a`, so reassigning `a.f` could have changed `c.f`.

We do this kind of invalidation for two types of statements: assignments (including indirect assignments like `+=`) and method/constructor calls. When a field or variable is assigned, we invalidate all types that depend on that assigned variable or field. When a method or constructor is called, we invalidate all types that depend on any field that could have been reassigned and is reachable from the receiver or one of the arguments. Whether a field belonging to a passed argument can be assigned by the method or not depends on the uniqueness type system. For example, when an argument `a` is passed to a method as `@ReadOnly`, we know that no field `a.*.b` can be assigned, so we don't invalidate the types that depend on it.

Example 5: Possibly modified fields

Consider the following example with method `baz`:

```java
class Foo {
  final double y;
  @Unique final Bar b1;
  @ReadOnly Bar b2;
}

class Bar {
  int x;
  @Unqiue Foo f;
}

static void baz(@ReadOnly Bar b, @Unique Foo f) {
  // ...
}
```

Assume this method is called as follows, where `b` and `f` are local variables from the context: `baz(b, f)`. The computed list of fields that could have been reassigned as a result of this method call is this:

- `f.b2`
- `f.b1.x`
- `f.b1.f`
- `f.b1.f.b2`
- `f.b1.f.b1.x`
- `f.b1.f.b1.f`
- …

Technically, the list is infinite due to the recursive nature of the structure. However, in practice, we can cut it off once fields start repeating. The reason is that if a type could depend on `f.b1.f.b1.x`, it could also depend on `f.b1.x` (through aliasing). Therefore, any types that depend on "deeper" recursive fields are already invalidated when going through the "higher" fields.

3.3. Mending conditions and contexts

Type checking is performed at many different places in the source code, for example for assignments (variable type), method calls (parameter types) and return values (return type). When the type can be verified through a relationship in the property type hierarchy, these checks are successful, otherwise the checker reports an error. Type errors cause the generation of assertions which must then be verified by a deductive verifier.

For all source code positions where this process of subtype relationship checking and assertion generation takes place, we compute the local refinement context and a sufficient condition for verifying the required type. We call the latter a *mending condition* if the subtype relationship check failed, because proving this condition later allows us to remove the reported type error.

Local refinement contexts and mending conditions are computed for the following kinds of type errors:

- ill-typed assignments
- ill-typed method results (return values)
- ill-typed method receivers
- ill-typed method or constructor call arguments
- ill-typed method or constructor postconditions

Local refinement contexts are first computed for each individual property type hierarchy. After all checkers have run, the produced contexts are merged to obtain contexts containing type information across all involved hierarchies. Then, the validity of each mending condition is tested given the corresponding combined context. When a mending condition is determined to be universally valid, we remove the type error it stems from.

The practical implementation of how the local refinement context is computed differs from its theoretical definition (Section 2.4) in two major ways:

1. In theory, all reachable fields (excluding cycles) are included. In practice, only the types of fields on `this` are tracked, so everything else is excluded from the context.
2. In theory, method calls are not expressions and are therefore not relevant to the validity of refinements, which can only contain expressions. In Java, however, method calls *are* expressions. Thus, all currently known return types of previous method calls are included in the context.

The restriction on included fields in particular significantly reduces the size of the local refinement context. As currently implemented, it is thus the union of the following sets:

- the current refinements of all local parameters and variables that are in scope
- the current refinement of `this`
- the current refinements of all fields on `this`

“Current” refinement means refinement in the context of the current statement. For example, when the type of a local variable x depends on another local variable y, the type of x is its declared type before and the top type after y is assigned a new value (see Section 3.2).

Example 6: Linked List refinement context

Consider the following singly-linked, immutable list implementation:

```
class List {

  final Object element;

  @Nullable
  @Length(len="size - 1")
  final List next;

  final int size;

  @Pure
  @Length(len="rest.size + 1")
  List(Object element, @NonNull List rest) {
    this.element = element;
    this.next = rest;
    this.size = rest.size + 1;
    Packing.pack(this, List.class);
  }

  @Pure
  @Length(len="1")
  List(Object element) {
    this.element = element;
    this.next = null;
    this.size = 1;
    Packing.pack(this, List.class);
  }

  @Pure
  Object nth(@ReadOnly List this,
             @Interval(min="0", max="size - 1") int i) {
    // ...
  }

  @Pure
  @NonNull
  @Length(len="this.size + other.size")
  List prependAll(@ReadOnly List this,
                  @ReadOnly List other) {
    // ...
  }
}
```

The property qualifiers are defined as follows:

```
annotation Interval(int min, int max) int
    :<==> "§subject§ >= §min§ && §subject§ <= §max§"
    for "§min§ <= §max§"

annotation Length(int length) List
    :<==> "(§subject§ == null && §length§ == 0) || (§subject§ != null &&
§subject§.size == §length§)"
    for "§length§ >= 0"
```

Notably, `null` is considered the empty list.

Now, consider the following piece of code:

```
@Interval(min="1",max="2") int i = 2;
@Length(len="1") List a = new List("foo");
@Length(len="2") List b = new List("bar", new List("baz"));
Object el = a.prependAll(b).nth(i);
```

Our focus is on line 4, specifically the assertion that `i` is in the interval between `0` and `a.prependAll(b).size`, as specified by the parameter declaration in `nth`. We obtain the following combined refinement context:

- `((a.size + b.size) >= 0)`
  `&& (((a.prependAll(b) == null) && ((a.size + b.size) == 0))`
  `|| ((a.prependAll(b) != null) && (a.prependAll(b).size == (a.size + b.size))))`
- `(1 >= 0) && (((a == null) && (1 == 0)) || ((a != null) && (a.size == 1)))`
- `(2 >= 0) && (((b == null) && (2 == 0)) || ((b != null) && (b.size == 2)))`
- `(1 <= 2) && ((i >= 1) && (i <= 2))`

And the following mending condition:
`(0 <= a.prependAll(b).size - 1) && ((i >= 0) && (i <= a.prependAll(b).size - 1))`

3.4. Translation of refinements to SMT

So far, we have detailed how how we parse, analyse, and collect property refinements. This section describes how we translate from our model of Java expressions to a model of SMT expressions, and finally to a format understood by SMT solvers.

To interface with SMT solvers, we use the JavaSMT library [12]. This library provides a uniform and idiomatic Java API for solvers implementing SMT-LIB [13] standard.

Example 7: From Java Expression to SMT

When a Java expression like

```
this.sizeSum(a, b) == (a.size + b.size)
```

is used as a refinement, it is first converted to an SMT expression model:

```
BinaryOperation[
  operator=EQUALS
  left=FunctionCall[
    method={sizeSum}
    arguments=(
      Literal[value={this}]
      Variable[expr={a}]
      Variable[expr={b}]
    )
  ]
  right=BinaryOperation[
    operator=PLUS
    left=Variable[expr={a.size}]
    right=Variable[expr={b.size}]
  ]
]
```

This AST is then used to construct formulae with JavaSMT, which in turn emits SMT-LIB code:

```
(declare-fun v_4 () Int)
(declare-fun v_3 () Int)
(declare-fun f_1 (Int Int Int) Int)
(declare-fun v_2 () Int)
(declare-fun v_1 () Int)
(assert (= (f_1 1 v_1 v_2) (+ v_3 v_4)))
```

3.4.1. Theories for Java types

All values that are referenced from refinements must be represented in SMT in some capacity for solvers to work with them. We map each Java type to a theory that is understood by SMT solvers. Currently, we divide Java types into integers types, floating point types, booleans, and “unknown”. The latter includes all declared types (classes). The following subsections detail how we assign a theory to each of these categories.

Primitive integer types

Java has 5 primitive integer types: `byte` (8 bit signed), `short` (16 bit signed), `int` (32 bit signed), `long` (64 bit signed), and `char` (16 bit unsigned) [14]. There are essentially two common theories that are used to model these types in SMT:

- Integers (SMT-LIB: `Ints` [13]): supported by most SMT solvers, but does not offer overflow behaviour natively. Also does not allow for modelling of bitwise operations such as `&` (AND) and `|` (OR).

- Bitvectors (SMT-LIB: `FixedSizeBitVectors` [13]): theory matching the actual Java integer semantics. Supports bitwise operation in addition to integer operations with overflow, but is less commonly implemented, and much slower in practice.

For our implementation, we have opted to use integers to model Java's primitive integers. This means that arithmetic operations are supported, and bitwise operations are not, for now. However, nothing stands in the way of supporting both in the future; inspiration may be taken from OpenJML's approach of allowing the user to choose which of the two theories to use [8].

We first specify functions on $\mathbb{Z}$ that correspond to Java's integer operations [14] *without* overflow. The theory of integers gives us access to the following core arithmetic functions in SMT [13]:

Function	Corresponding operator in Java	Description
add $(+)$	+	Addition on $\mathbb{Z}$
sub $(-)$	-	Subtraction on $\mathbb{Z}$
div	n/a	Division on $\mathbb{Z}$ (Euclidean semantics)
mul $(\cdot)$	*	Multiplication on $\mathbb{Z}$
mod	n/a	Modulo on $\mathbb{Z}$ (Euclidean semantics)

In Java, the division operator `/` rounds towards zero and the operator `%` implements modulo in accordance with that notion of division [14], whereas `div` and `mod` from SMT-LIB [13] use Boute's Euclidean semantics [15]. Thus, the native SMT-LIB functions do not map cleanly to Java.

To illustrate the difference, take these two examples:

- SMT: $\mathrm{mod}(-10, 3) = 2$
  Java: `-10 % 3 == -1`
- SMT: $\mathrm{div}(-10, 3) = -4$
  Java: `-10 / 3 == -3`

We therefore need additional functions *jdiv* and *jmod* that correspond to the Java semantics on whole numbers [4, Ch. 5]. A possible definition for these functions is given by the `precOfInt` taclet[3] in the KeY project [4]:

$$\mathrm{jdiv}(a, n) := \begin{cases} \mathrm{div}(a, n) & \text{if } a \geq 0 \\ -\mathrm{div}(-a, n) & \text{otherwise} \end{cases}$$

$$\mathrm{jmod}(a, n) := a - \mathrm{jdiv}(a, n) \cdot n$$

Now that we can express Java's integer operations in SMT on $\mathbb{Z}$, we still need to model the overflow semantics that constrain integer values to a fixed number of bits. To do so, we apply

[3]See https://github.com/KeYProject/key/blob/627d7455d82ad379acda01f59040a3324182bf77/key.core/src/main/resources/de/uka/ilkd/key/smt/newsmt2/Int.DefinedSymbolsHandler.preamble.xml (accessed 2025-03-10)

the following transformation in SMT to every occurrence of a signed integer, where i is the SMT integer formula and b is the number of bits in the underlying Java type [4, Ch. 5]:

$$\text{overflow}(i, b) := \text{mod}\big((i + 2^{b-1}), 2^b\big) - 2^{b-1}$$

Effectively, this means that all arithmetic operations on Java integer types are implemented as a combination of the corresponding operation on whole numbers (the theory of integers) and the `overflow` function applied to the result. For example, to translate the Java expression `a + b`, where a and b are `int`s, we construct the formula $\text{overflow}(\text{add}(a, b), 32)$.

Unsigned overflow (for `char`) works the same, except that we do not use the 2^{b-1} offset in the *overflow* function. I.e., we simply use $\text{mod}(i, 2^{16})$.

Floating point types

Java has two IEEE-754 floating point types: `float` (single precision) and `double` (double precision) [14]. Like for integers, there is a standard theory for floating point numbers [16] (in SMT-LIB: `FloatingPoint` [13]). We map Java `floats` directly to single precision numbers, and `doubles` to double precision numbers in this theory. We therefore support all of Java's operators on floating point values.

Booleans

Booleans are the basis for all refinement expressions, since all property refinements must be boolean predicates. We map Java `booleans` to this universally supported theory (in SMT-LIB: `Core` [13]) directly, and therefore support all of Java's boolean operators.

It is worth noting here that SMT-LIB makes no distinction between "formula" (boolean value) and "term" (domain value). Instead, the values *true* and *false* are *part* of the underlying domain. Boolean variables and functions are specified like variables or functions of any other type [13]. Therefore, there is no need to separate formulae into a logic and a domain layer. This simplification is possible because SMT-LIB uses a *many-sorted* logic, which is elaborated on in Section 3.4.5.

Other types

Types other than the ones listed above are treated as "UNKNOWN". Unknown types primarily appear in function arguments, receivers, and return values. For example, the return value of `x.foo()` is generally dependent on `x`, and cannot be assumed to be equal to `y.foo()`. This poses a problem for the translation to SMT, since we do not have a general theory for objects.

To work around this problem, we conceptually assign a unique integer to each object in the JVM heap, and then represent object types that appear in code as integers. In other words, we treat methods that return a reference type as if they returned integers, reference type variables as integer variables, and so on. Furthermore, we assign unique integer *literals* to string literals, `this`, and `null`. The implications are that `this` and string literals are never equal to `null`, and `this` is never equal to a string literal.

This approach allows us to represent objects in SMT, albeit with a fair amount of imprecision. For example, our model does not include aliasing rules like $a = b \Rightarrow a.f = b.f$. To support rules like this, we would need to use a more complex mapping of objects like in the heap-array model [4, Ch. 2]. We did not deem this to be worth the additional cost.

In the future, additional type mappings besides the general object representation may be introduced where a suitable theory is available. Notably, SMT-LIB defines a theory for strings and one for arrays, which could potentially be used to model Java `String`s and array types, respectively [13].

3.4.2. Field accesses and local variables

Field accesses are expressions of the form $e.f$, where e is another expression (such as a method call or a field access), and f is a field of the class of the expression e. Local variables are identifiers that do not refer to a field in the current context. Each unique field access and local variable is mapped to a unique variable in SMT. The SMT type of the variable is chosen according to the categorisation given in Section 3.4.1.

The uniqueness of a field access or local variable is only dependent on syntax, not semantics. This means that the expressions `x` and `this.y` are mapped to different variables, even if `x` is an alias of `this.y`.

Variable names are based on a number incremented at every encountered field access or local variable. For instance, given the expression `foo.f == bar.baz.f`, the field access `foo.f` will be mapped to a variable named `v_0`, and `bar.baz.f` will be mapped to a variable named `v_1`. This naming scheme guarantees that all variables use valid SMT-LIB names [13]. On the flip side, the original Java Expression cannot be derived from the SMT-LIB variable name alone.

3.4.3. Method calls

A method call is an expression of the form $e.m(a_1, ..., a_n)$, where e is the receiver expression, m is the name of a method in the type of the receiver, and $a_1, ..., a_n$ are the argument expressions.

Methods are treated as black boxes by our analysis. Thus, in order to model them in SMT, we use the theory of uninterpreted functions (UF)[4], which allows the use of function symbols in SMT formulae whose definition is unknown. A formula containing an uninterpreted function symbol is thus only universally valid if the formula is valid for all possible definitions of the function symbol.

Similar to field accesses, each unique method is assigned a name based on an incrementing number. The uniqueness of a method is determined by the underlying Java element, i.e. two methods are the same if they have the same name and are called on the same type.

In order to convert a Java method call expression to SMT, we first translate the receiver and the arguments to SMT, map the underlying method to the name of an uninterpreted function, and construct a function call whose arguments concatenation of receiver and original arguments. For example, the expression `map.get(5)` may be translated to `f_0(v_0, 5)`, where `f_0` is the unique name for the method `get` in the `java.util.Map` interface, and `v_0` is the variable name associated with local variable `map` (see Section 3.4.2).

[4]Also known as the empty theory.

As mentioned earlier, the methods called in refinement expressions must be deterministic and free of side effects. This is also a necessary condition for the soundness of the mapping to SMT because uninterpreted functions are functions in the mathematical sense, i.e. free of side effects by nature.

3.4.4. Built-in constants and functions

In addition to the handling of local variables, fields and methods described in Section 3.4.2 and Section 3.4.3 respectively, we provide a more direct, built-in mapping between Java expressions and SMT for a number of constants and functions. This list can easily be extended in the future, but currently it entails the following:

- `Math.PI`, `Math.E`, `Math.PI` are mapped to their literal values
- `Double.NaN`, `Double.PLUS_INFINITY`, `Double.MINUS_INFINITY` are mapped to the corresponding floating point construct. The same is done for the equivalent constants in `Float`.
- `Math.abs(double)` and `Math.abs(float)`
- `Math.IEEEremainder(double,double)` is mapped to the IEEE *remainder* operation [17].
- `Math.round(double)` and `Math.round(float)` are mapped to the IEEE *round* operation, with rounding mode "round to nearest, ties to even" [17].
- `Math.sqrt(double)` is mapped to the IEEE *squareRoot* operation [17].
- the constants `MAX_VALUE` and `MIN_VALUE` in `Character`, `Byte`, `Short`, `Integer`, and `Long`

3.4.5. Decidability

In SMT-LIB, theories are combined to *many-sorted logics* with equality as well as optional extensions or restrictions [13]. The universe of a many-sorted logic is partitioned into multiple sets, called *sorts*, which can be subject to different theories [18]. This is crucial for determining the validity of boolean Java expressions, since we cannot define a single standard interpretation for all operators across all types. For example, the + in `x + y` should be interpreted differently depending on whether `x` and `y` are floating-point or integer variables.

Due to the definition of property types, all our logics are restricted to quantifier-free (`QF`) formulae. Furthermore, all logics include the `UF` extension, allowing for free (uninterpreted) function symbols and sorts [13].

The following table lists all logics that may be used to determine the validity of property types, based on theories enumerated in the previous sections:

SMT-LIB Logic	Description	Decidable?
`QF_UFLIA`	linear integer arithmetic	Yes [19]
`QF_UFNIA`	non-linear integer arithmetic	No [20]
`QF_UFFP`[5]	IEEE-754 floating point arithmetic	Yes[6]
`QF_UF`	core theory (booleans)	Yes[7]

[5]As of version 2.7, the theory identifier `FP` is not fully integrated into the hierarchy of SMT-LIB logics yet, so "`QF_UFFP`" is not found in the standard [13].

[6]Standard models for the floating point theory are finite [16], making it decidable [7, Ch. I].

[7]See footnote 1 on page 15.

These logics are dynamically combined to obtain a logic that covers all the functionality needed to determine the validity of each constructed formula. For example, the theory of floating-point arithmetic is not included by default, but it is included automatically if floating-point arithmetic is used in a refinement expression.

Because non-linear integer arithmetic is undecidable, any combined logic that includes this theory is also undecidable. Thus, it appears that our system can no longer decide the validity of mending conditions when they involve the product of two integer variables. This seems problematic at first, given that we earlier built the theory of our method on the assumption that the validity problem is decidable. However, since we apply overflow coercion to every integer in any given formula, the *effective* domain of formulae involving non-linear integer arithmetic is finite. Furthermore, the entire problem can be avoided using timeouts in practice. If the solver is unable to prove that a formula is valid after a certain amount of time, we abort the attempt and proceed as if we had found it to *not* be universally valid. In that case, the corresponding assertion is not dismissed and left to the deductive verifier as usual. As a result, our system remains functional even if the underlying logic is technically undecidable.

3.4.6. SMT Solver

In the previous subsections, we have described how we map Java expressions to a model that uses logical theories to represent Java semantics. What is still missing is the final step of the process: actually determining whether the mending conditions are universally valid under the constraints of the context.

As already mentioned at the beginning of Section 3.4, we use the JavaSMT library [12] to interface with SMT solvers. This library provides high level APIs to interact with various solvers implementing the SMT-LIB standard, including the ability to verify whether a formula is satisfiable or not [13]. The library's API is solver-agnostic but its consumers can choose the solver implementation from a list of available options, including the prolific Z3 [12], [21].

For the evaluation and any other tests performed in the context of this work, we used the SMTInterpol solver [22]. The solver is written in Java and comes bundled with JavaSMT. Its features are sufficient to run all current Property Checker test cases, including the case study. However, it lacks support for floating-point and non-linear integer arithmetic.

For now, the solver implementation choice is fixed in the code and not configurable from the outside. Changing it requires little effort, however. First, if the solver requires any additional dependencies, they must be included, e.g. `libz3.so` for Z3. Then, when creating an instance of the `SolverContext` interface, only the solver parameter needs to be changed, e.g. specifying `Solvers.Z3` instead of `Solvers.SMTINTERPOL`. In the future, this option should be configurable through a command line parameter or similar.

3.5. Changes in the Checker Framework

In the process of integrating SMT-based type verification into the Property Checker, a few adjustments were made to the underlying Checker Framework. These adjustments were necessary in places where the implementation of the Checker Framework was incompatible with the requirements for our work and the only way to rectify it was by changing framework-internal code. This further led to improvements in the property type system implementation.

3.5.1. Dependent types helper

The Checker Framework provides an abstract helper class called `DependentTypesHelper` that performs automatic viewpoint adaptation of property type argument expressions. This helper class is used in many different places within the framework, and many of the default implementations for other components rely on a correct implementation being available.

However, the design choices in this helper class are unfortunately too narrow to be applied to our implementation of property types. To recall, the actual arguments for property types can be arbitrary Java expressions in practice. We enable this by allowing users to pass them as strings, e.g. `@Interval(min="1", max="x + 3")`. Thus, viewpoint-adaptation should be applied to all arguments by default.

However, this desired behaviour is not supported by the Checker Framework. It expects dependent type qualifiers, i.e. annotations parameterised by Java expressions, to explicitly mark all parameters that can be Java expressions using a `@JavaExpression` annotation, and that all of these parameters are of type `String[]`. These design constraints restrict our ability to implement viewpoint-adaptation without creating a conflict with our design of property types, so we changed `DependentTypesHelper` in accordance with our notion of dependent types. In our custom Checker Framework version [5], `DependentTypesHelper` now considers all annotation parameters of type `String` Java expressions to be viewpoint-adapted.

3.5.2. Nominal parameter references

The Checker Framework Java expression parser uses a custom extension of the Java expression syntax to represent method or constructor parameters. While it is possible to refer to parameters by name when parsing at a specific path in the Java AST (which allows the Checker Framework to resolve local variable and parameter references), parameters are referred to by index at the top level. For example, if we want to specify that the return value of the method `int foo(int x)` is an integer between `0` and `x`, we must write `@Interval(min="0", max="#1")` instead of `@Interval(min="0", max="x")`.

In principle, this is not a fundamental problem for our implementation of property types. But because we already make conflicting use of this parameter syntax to represent the special `§subject§` parameter (see Section 3.1) and generally prefer nominal parameter references for clarity, we opted to adjust the Java expression parser to support referencing parameters by name at the method declaration level [5].

3.5.3. Pseudo method calls for object creation

One major limitation of the Checker Framework `JavaExpression` abstraction is that it does not support object creation expressions, i.e. `new ...` expressions. This does not break anything in our implementation but means that we cannot integrate handling of the constructor return types with the handling of method return types, despite them being functionally equivalent for our purposes.

We thus decided to represent object creation expressions as static method calls. Specifically, an expression like `new Foo(bar, 2)` is mapped to `Foo."<init>"(bar, 2)` and "returns" a value of type `Foo`. This allows us to at least support constructor calls as subjects of property types. The current workaround is limited to constructor calls at the root of the AST; subjects that *contain* constructor calls as nested expressions are generally unsupported[8].

The change required to enable this representation of constructor calls as pseudo-methods in the Checker Framework was the removal of an artificial restriction by which two constructor call expressions could never be equal [5].

3.5.4. Impact on the property type system implementation

One side effect of the aforementioned changes to the Checker Framework is improved accuracy of our property type checker. Before the dependent types helper adjustment and the support for nominal parameter references, the checker would often deal with incorrect types. For example, the framework-assigned type of a method call would be the type *declared at the return type* of the method, unadapted. Hence, when the method `Interval(min="x", max="y") int foo(int x, int y)` was called as `foo(1, x + 3)`, the type of this expression would be incorrectly determined as `@Interval(min="x", max="y") int` instead of `@Interval(min="1", max="x + 3") int`. This imprecision caused a class of type errors that stopped occurring after correct viewpoint-adaptation was implemented (see Section 4.1).

The correction further allowed us to improve the implementation of *flow-sensitive type refinements* [11], the Checker Framework's facilities for keeping track of type contexts and updating them in accordance with operations in the source code. In particular, we implemented invalidation of dependent types after method calls and assignments (Section 3.2). The previous code implementing invalidation of dependent types was flawed in three ways:

1. it was only implemented for fields, meaning that reassigning a local variable would not invalidate the types depending on it,
2. it failed to consider that method calls on the subject in the property condition could lead to indirect dependencies on fields, and
3. it invalidated every type whose arguments were not all literals, regardless of whether their refinement actually depended on the field in question, leading to a high level of imprecision.

With our new analysis, all of these issues were resolved.

[8]Note that when we say "unsupported", this does not mean that the feature cannot be used at all, only that it will be skipped by the SMT-based component of the Kukicha verification pipeline.

4. Evaluation

To evaluate the contribution of this work, we look at a case study first introduced by F. Lanzinger *et al.* [23] and later refined by F. Lanzinger *et al.* [3]. The case study is a Java application implementing the business logic for a fictional web shop. It consists of abstractions for customers, products, and lists of orders. Orders are a combination of a customer and a product. Multiple property type systems are used to guarantee the absence of certain run-time exceptions, that lists of orders are sorted by price, and that an order can only be created if the customer is old enough to buy the (age-restricted) product [3].

We first run the case study through our modified Property Checker [5] exactly as F. Lanzinger *et al.* [3] to see how our work changes the number of type errors emitted by the checker (Section 4.1) as well as the number of assertions produced by the program translator (Section 4.2), compared to the results of F. Lanzinger *et al.* [3]. Finally, we evaluate whether and to which extent our work allows us to simplify the case study's code without increasing the effort required for verification (Section 4.3).

4.1. Reduction of type checker errors

Originally, **35 type errors** were reported by the Property Checker throughout the case study [3]. Due to the consequences of our changes to the Checker Framework detailed in Section 3.5.4, this number was **reduced to 23**. In other words, 12 of the type errors reported were caused by incorrect viewpoint adaptation in the previous implementation. In the case study, these errors can be categorised in three groups.

First, return type errors. The following example used to contain a type error in `customer18` because the type of `new Customer(name, 18)` was determined to be of type `@AgedOver(age="age")`, which was not compatible with the declared return type of `@AgedOver(age="18")`. Now, with correct viewpoint adaptation, the type of `new Customer(name, 18)` is `@AgedOver(age="18")`.

```
public class Customer {

  public @AgedOver(age="age") Customer(
    String name,
    @Interval(min="14", max="150") int age
  ) {
    // ...
  }

  public static @AgedOver(age="18") Customer customer18(String name) {
      return new Customer(name, 18);
  }
}
```

This also means that replacing calls to helper methods like `customer18` with direct constructor calls won't result in a type error anymore. Thus, these methods have effectively become obsolete.

Second, argument type errors. The example below used to contain a type error in `order18` because the expected type for the constructor argument `customer` was `@AgedOver(age="witness")`, which was incompatible with the type `@AgedOver(age="18")`. With correct viewpoint adaptation, the expected type is `@AgedOver(age="18")` (analogously for the argument `product`).

Third, postcondition errors. In the same example, the implicit posttype of the constructor argument `customer` is `@AgedOver(age="witness")` since no explicit posttype is declared. Similarly, the posttype of the method argument `customer` of the method `order18` is `@AgedOver(age="18")`. Because of the lack of viewpoint adaptation, the type of `customer` was determined to be `@AgedOver(age="witness")` after the call to the `Order` constructor, which is incompatible with `@AgedOver(age="18")`.

```
public class Order {

  public Order(
    int witness,
    @AgedOver(age="witness") Customer customer,
    @AllowedFor(age="witness") Product product
  ) {
    // ...
  }

  public static Order order18(
    @AgedOver(age="18") Customer customer,
    @AllowedFor(age="18") Product product
  ) {
    return new Order(18, customer, product);
  }
}
```

Looking at these examples, it becomes apparent that these improvements only happened in very trivial helper method contexts involving literal expressions. Correct viewpoint adaptation alone does not make the property type rules more powerful than before; the type checker is still unable to verify dependent types.

4.2. Reduction of assertions in translation

From the remaining 23 type errors in the case study, 16 mending conditions, i.e. formulae whose validity is a sufficient condition for removing the corresponding type error, were generated. Of those 16, 5 mending conditions could be proven via SMT, leading to a reduction in the number of leftover assertions by 5.

Two similar type errors are found in the `SortedList` class, in two methods that both call `this.first.getHead()`. The field `first` in this class is a linked list `Node` and can generally be `null`, to indicate an empty list. This leads to the error: the nullness type system cannot guarantee that `this.first` is non-null, and thus it cannot guarantee that calling a method on this field is safe.

However, in both cases, `this` is declared as `@NonEmpty`. The property is defined as follows: `§subject§ != null && §subject§.first != null`. This means that, in the below example, the local refinement context at the return statement includes the expression `this != null && this.first != null`, which trivially implies the mending condition `this.first != null`.

```
public @MaybeAliased Order getHead(@Unique @NonEmpty @Inv SortedList this) {
  // :: error: nullness.method.invocation.invalid
  return this.first.getHead();
}
```

While this is trivial to solve for any SMT solver, it illustrates the power of combining type information from different property type hierarchies (in this case, nullness and emptiness), to strengthen the constraints in the context.

Another type error that is successfully mended is found at the `remove` method of the `SortedList` class, which removes the last element (tail) from the list. This is the method definition:

```
@EnsuresPossiblyEmpty("this")
// :: error: empty.contracts.postcondition.not.satisfied
public Order remove(@Unique @NonEmpty @Inv SortedList this) {
  Order result = this.first.getHead();
  Packing.unpack(this, SortedList.class);
  this.first = this.first.stealTail();
  Packing.pack(this, SortedList.class);
  return result;
}
```

The type error is connected to the postcondition on `this` (denoted by `@EnsuresPossiblyEmpty("this")`), which states that the property `@PossiblyEmpty` holds for `this` *after* the method has returned. The property `@PossiblyEmpty`, i.e. the property that a list is either empty or non-empty, is equivalent to `@NonNull`, since all lists are either empty or non-empty. Thus, the postcondition is `this != null`. While the type checker is unable to verify this at the end of the method, it is trivial to prove with an SMT solver, since `this` and `null` are mapped to distinct values (see Section 3.4.1).

The last two mended type errors are found in the implicit postconditions of the `Order` constructor.

```
public final int witness;
public final @AgedOver(age="witness") Customer customer;
public final @AllowedFor(age="witness") Product product;

// :: error: agedover.contracts.postcondition.not.satisfied
// :: error: allowedfor.contracts.postcondition.not.satisfied
public Order(
  int witness,
  @AgedOver(age="witness") Customer customer,
  @AllowedFor(age="witness") Product product
) {
  this.witness = witness;
  this.customer = customer;
  this.product = product;
  // :: error: initialization.fields.uninitialized
  Packing.pack(this, Order.class);
}
```

If no explicit posttype in the form of an `@EnsuresX` annotation is given for a parameter, the implicit posttype is the declared input type. In other words, if we don't explicitly declare that the property type of a parameter changes as a result of calling the method or constructor, we assume that the original property must still hold.

Here, the verification pipeline must ensure that `customer` and `product` still have the types `@AgedOver(age="witness")` and `@AllowedFor(age="witness")` at the end of the constructor respectively. The code in the constructor neither reassigns the `witness` parameter nor modifies `customer` or `product`, so the types don't change over the course of the constructor. However, because the reflexive relationship between dependent types ($\rho \preceq \rho$) is currently not implemented, the type checker alone cannot determine that the type `@AgedOver(age="witness")` is the same at the beginning as at the end. With the local refinement context, this is again a trivial error to remove for the SMT solver.

We've seen that all 5 of the mended errors were essentially trivial in nature and thus did not require the full potential of SMT solving. Hence, it may instead be interesting to look at some errors that can *not* be mended, and why. The remaining type errors can be put in one of two categories: conditional context, or assignment context.

Conditional context

Examples for this kind of error can, for instance, be found in the `Node` class. A `Node` is a singly-linked list of orders, sorted by price of the associated product.

In a method that inserts a new element at the front of the list, we find the following code:

```
if (this.tail == null) {
    this.tail = new Node(this.head);
} else {
    // :: error: nullness.argument.type.incompatible
    this.tail = new Node(this.head, this.tail);
}
```

The issue here is that the `Node` constructor expects its tail parameter to be non-null, but `this.tail` is not known to be `@NonNull` at the call site. In this case, it's obvious that this error is a false alarm; it is impossible for `this.tail` to be `null` if we find ourselves in the `else` branch of this conditional statement.

However, the type checker currently lacks the ability to recognise conditional context like this. Thus, no branch-dependent information is added to the type context, and, as a consequence, `this.head` is not treated as `@NonNull`. Similar error patterns are found in other places of the case study.

Assignment context

Returning to the `Order` constructor from before, we see one more error at the end of its body: `initialization.fields.uninitialized`. This error appears when we attempt to `pack` an object, i.e. *commit* its fields, but the type system cannot verify that all the fields have their declared types. In this case, `this.customer` must be of type `@AgedOver(age="this.witness")`, and `this.product` must be of type `@AllowedFor(age="this.witness")`. These types mirror the constructor parameter types exactly. The problem is: while we know through basic type inference that `this.customer` and `this.product` have the same *type* as the corresponding parameters, we don't know that the *value* of `this.witness` and `witness` (the parameter) are the same, so no conclusion can be drawn for the fields' declared types.

Most of the other type errors in the case study fall in this category: a variable or field is assigned to, but no direct connection between it and the assigned value is made, leading to a missing link when trying to prove a mending condition later.

To solve this, we would ultimately need to implement a system that models a temporal dimension for variables and fields. That is, when a field or variable is assigned to, its previous value isn't overwritten; instead, the before- and after-states are represented by two distinct variables in SMT. Much like modelling conditionals, modelling assignments would require an integration of program *structure* in our logic, which we currently lack.

4.3. Reduction of redundancy

The analysis so far suggests that the case study is only minimally affected by our SMT-based extension to the Kukicha method. Indeed, no non-trivial assertions were able to be dismissed by our system. This is primarily because few of the type errors in this case study fall in the category of assertions our extension was designed to dismiss. There is, however, one pattern of redundancy in the case study that can be refactored to demonstrate more interesting effects.

We have already shown the constructor declaration of the `Order` class:

```
public Order(
  int witness,
  @AgedOver(age="witness") Customer customer,
  @AllowedFor(age="witness") Product product
)
```

The `witness` parameter here is only intended as an aid for the type checker in cases where a literal is passed as the actual value. For example, the following code is syntactically well-typed:

```
Customer c = new Customer("Bob", 19);
Product p = new Product("DVD", 10, 16);
Order o = new Order(17, c, p);
```

Through viewpoint-adaptation, the expected types for the `Order` constructor arguments here become `@AgedOver(age="17")` (which is a supertype of `@AgedOver(age="19")`) and `@AllowedFor(age="17")` (which is a supertype of `@AllowedFor(age="16")`). The same would apply for any other literal `witness` between 16 and 19. But the witness parameter is semantically redundant, since the combination `@AgedOver(age="witness") customer` and `@AllowedFor(age="witness") product` is equivalent to `@AgedOver(age="product.ageRestriction") customer` or `@AllowedFor(age="customer.age") product`. Thus, we can simplify the `Order` constructor to the following:

```
public Order(
  @AgedOver(age="product.ageRestriction") Customer customer,
  Product product
)
```

Note that not both dependent types are needed, since one implies the other. Now, this code is no longer syntactically well-typed:

```
@AgedOver(age="19") Customer c = new Customer("Bob", 19);
@AllowedFor(age="16") Product p = new Product("DVD", 10, 16);
// :: error: agedover.argument.type.incompatible
Order o = new Order(c, p);
```

The expected type now is `@AgedOver(age="p.ageRestriction")`, which is not related to the actual type `@AgedOver(age="19")` in terms of the property type hierarchy. However, since the local refinement context at this position constrains both `p.ageRestriction` **and** `c.age`, our extension is able to deduce the validity of the required property for `c`, and the type error is removed as a result. This demonstrates the deductive abilities of our system and its understanding of integer arithmetic.

Using the same principle, we were able to restructure the case study's entire client code, i.e. the code interacting with the shop API. Where previously, helper methods and parameters were needed to circumvent type errors, these errors are now dismissed by our extension. The following code leaves no assertions for the deductive verifier:

```
Product product18 = new Product("Louisiana Buzzsaw Carnage", 10, 18);
Customer customer18 = new Customer("Alice", 18);
Shop shop = new Shop();
shop.addOrder(new Order(customer18, product18));
```

```
Product product6 = new Product("Tim & Jeffrey, All Episodes", 10, 6);
shop.addOrder(new Order(customer18, product6));
Customer customer14 = new Customer("Bob", 14);
shop.addOrder(new Order(customer14, product6));
shop.processNextOrder();
shop.processNextOrder();
shop.processNextOrder();
shop.processNextOrder();
```

Explicit property annotations are also not required here due to the local type inference performed by the Checker Framework. The full refactored case study can be found in the `smt/CaseStudyMutableTest` test case in our version of the Property Checker [5].

4.4. Assessment of overall impact

The small size and scope of the case study limits the extent of our evaluation. Nevertheless, it allows us to provide an approximate assessment of our work's impact and, in particular, to answer *RQ2*:

(RQ2) Does the SMT-based implementation of a property type deduction mechanism in the Property Checker lead to a significant reduction of type errors?

Compared to *RQ1*, where our inquiry was about the theoretical extension, we obtained more mixed results here. While many type errors in the case study were resolved as a side-effect of our work on the Property Checker (Section 4.1), the number of errors *mended* via SMT was rather low. As an explanation, we found that the majority of type errors could not be mended due to a lack of symbolic evaluation rules in our system.

On the other hand, we found that our extension allowed us to refactor, and thus improve, portions of code that previously contained redundancies to simplify type checking. For the case study, this also meant that the code consuming the API of the web shop model was left with zero unproven assertions. As a consequence, the corresponding method does not have to be loaded into *KeY* [4] for deductive verification anymore, lowering the overall verification effort.

To summarise, the case study results indicate that our extension does lead to a reduction of type errors, as long as no structural context is required to resolve them. However, we cannot conclusively claim that the reduction is necessarily *significant* in real-world applications, since the case study is too small to make this conclusion. More research with larger case studies is needed to evaluate how often the kinds of errors occur that can be mended by our SMT-based system. Further research could also be used to examine the difference in verification effort more closely, i.e. how our extension changes the actual time and effort developers need to put into verifying a program's safety with the Property Checker.

5. Limitations and future work

The consequences of our work for the concept of property hierarchies could be explored further. While we have shown that a sufficiently powerful SMT solver can be used to infer property hierarchies that technically subsume any manually specified hierarchy (Theorem 1), adjusting the Property Checker implementation of hierarchies was out of scope for this work. Unifying the SMT component and property hierarchies even further could also be an interesting subject for the future, e.g. by revising the original property type rules.

As hinted at already in the evaluation, the local refinement context could be modified to include clauses derived from the structure of the program. Currently, the local refinement context and thus, the constraints for mending conditions, are limited to formulae representing known types, e.g. `x >= && x <= 5`. Using some simple symbolic evaluation rules could lead to much stronger constraints and thus more successful proofs.

One example for this kind of analysis is conditional statements:

```
@P("x") int a = ...;
if (y < x) {
  x = y + 5;
}
```

Here, we know that `x` only changes (and therefore invalidates the type of `a`) if `y < x`. Hence, we could include the type of `a` in the context in a conditional clause:

$\neg(y < x) \to Refin_{P(x)}[subject/a]$

A similar rule would be possible for assignments: when a variable is assigned to, we can add a clause to the refinement context that models the equality between the variable and the right-hand side of the assignment. Care would have to be taken here when variables are assigned multiple times. Each variable in SMT can only ever correspond to the state of a program variable at a specific point in time.

This kind of symbolic evaluation could be limited to a small selection of patterns and types of statements to complement the current implementation. It could also go much further into the direction of deductive verification, e.g. by translating code with property types to an SMT-based intermediary verification language like *Boogie* [24].

A major project for the future could be to develop a type inference system for property types. The Property Checker already supports a very primitive kind of inference through the Checker Framework, which does not take the underlying property hierarchies into consideration. Type inference for refinement types was part of the original paper on liquid types [25], so we know that refinements restricted to the logic of liquid types can generally be inferred. The remaining questions are how this can be transferred to the higher-level concept of qualified properties, and whether something similar can be developed for an object-oriented language like Java.

6. Related Work

The basis for this work is the introduction of the Kukicha method by F. Lanzinger *et al.* [3], which added the packing and uniqueness type systems to an earlier version of property types for immutable data structures [23]. The Java implementation of the Kukicha method is built on the Checker Framework [11].

Logically qualified data types (liquid types for short) are similar to property types in that they extend a given base type system with logical type refinements [25]. The main difference is that type refinements for liquid types are *ad hoc*, while property types associate refinements with named property qualifiers. The concept of liquid types was originally designed for functional programming languages like Haskell [26]. It was also adapted for Java in the *Liquid Java* project [9]. One fundamental property of liquid types is that they are restricted to formulae whose validity can be decided by SMT solving alone [9]. In contrast, our SMT-based extension of the Kukicha method still allows for property types whose validity cannot be definitively decided by an SMT solver. In cases where SMT is not enough, we leave the verification to a deductive verifier.

Our implementation of the Kukicha method for Java, the Property Checker, translates source code to a form that includes annotations in the *Java Modelling Language* (JML) [3]. JML is understood by most deductive verifiers for Java, notably *KeY* [4] and *OpenJML* [8]. KeY offers semi-interactive verification using a rule-based system for an intermediate, logical representation of Java code called *JavaDL*. This core concept does not depend on SMT. However, some components exist that do use SMT, e.g. test case generation [4, Ch. 12] and support for floating point arithmetic [27]. Furthermore, SMT solving can optionally be used to close open proof goals. That is, in places where KeY's sequent calculus is unable to find a proof, the goal can be translated to SMT and passed to Z3 [21] for an attempt to automatically resolve the issue. This constitutes a mapping between KeY's JavaDL and SMT similar to our mapping between a subset of Java expressions and SMT. However, our mapping happens at a higher level: it is integrated directly into the Property Checker and built on the property type abstraction, while KeY needs to analyse the annotated source code and perform symbolic evaluation before it can use SMT to discharge assertions.

On the other hand, OpenJML takes a fully-automated approach to deductive verification and heavily relies on SMT for all its logical deductions [8]. Like KeY, its scope is much broader than that of our work. It uses more complex reasoning based on the program structure, whereas we limit ourselves to information derived from the higher-level property type system. Put simply, the purpose of our extension to Kukicha is to minimise the time spent in external deductive verifiers, and conversely maximise the amount of property assertions verified within the Property Checker.

7. Conclusion

To summarise, we established that validity of property refinements in domains consistent with language semantics is a sufficient condition for semantic well-typedness. Based on this observation, we showed how property type hierarchies can be inferred, and subsequently how relationships between dependent types can be decided. While it is limited compared to deductive verification, the local refinement context can provide constraints that lead to the successful verification of property types that cannot be verified using the existing type rules.

We went on to implement these concepts in practice by extending the Java-based Property Checker application. We improved several aspects of the existing type system implementation in the process, most notably dependent type invalidation. To decide the validity of property refinements, we created a mapping from Java expressions to SMT formulae in a many-sorted logic, leveraging the SMT-LIB standard to pass those formulae to existing SMT solvers.

Finally, we found in a small case study that our work led to some improvements. First, the accuracy of the type system increased as a result of incidental changes. Second, our extended version of the Property Checker was indeed able to verify certain property types that could not be verified without deductive verification before.

In the introduction, we set out to answer two research questions: one asking whether the Kukicha method could be extended with a validity-based approach to property type verification, and the other asking whether its implementation would lead to a more powerful Property Checker. We were able to give a clear positive answer to the first question, and a more cautiously positive one to the second. We were unable to draw broad conclusions about the benefits of our SMT-based extension in practice due to the limited scope and size of the case study we used in our evaluation. However, we did find that our extension increases the accuracy of the property type system and we demonstrated that it can lead to better code.

Bibliography

[1] L. d. Moura and S. Ullrich, ‘The Lean 4 Theorem Prover and Programming Language’, in *Automated Deduction – CADE 28*, Springer International Publishing, 2021, pp. 625–635. doi: 10.1007/978-3-030-79876-5_37.

[2] ‘2024 Stack Overflow Developer Survey’. Accessed: Feb. 27, 2025. [Online]. Available: https://survey.stackoverflow.co/2024/

[3] F. Lanzinger, J. Bachmeier, M. Ulbrich, and W. Dietl, ‘Kukicha – Efficient Yet Powerful Refinement Types for Object-Oriented Languages’, 2025.

[4] W. Ahrendt, B. Beckert, R. Bubel, R. Hähnle, P. H. Schmitt, and M. Ulbrich, Eds., *Deductive Software Verification – The KeY Book*. Springer International Publishing, 2016. doi: 10.1007/978-3-319-49812-6.

[5] J. Zuber, ‘SMT-Based Verification of Property Types: Property Checker and Checker Framework’. Zenodo, Mar. 2025. doi: 10.5281/zenodo.15005210.

[6] A. Church, ‘An Unsolvable Problem of Elementary Number Theory’, *American Journal of Mathematics*, vol. 58, no. 2, p. 345, Apr. 1936, doi: 10.2307/2371045.

[7] W. Ackermann, *Solvable Cases of the Decision Problem*, 3rd print. in Studies in Logic and the Foundations of Mathematics. North-Holland Publishing Company, 1968.

[8] D. R. Cok, ‘Does your software do what it should? User guide to specification and verification with the Java Modeling Language and OpenJML’. Jul. 2022. Accessed: Dec. 30, 2024. [Online]. Available: https://www.openjml.org/documentation/OpenJMLUser Guide.pdf

[9] C. Gamboa, P. Canelas, C. Timperley, and A. Fonseca, ‘Usability-Oriented Design of Liquid Types for Java’, in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, IEEE, May 2023, pp. 1520–1532. doi: 10.1109/icse48619.2023.00132.

[10] N. Vazou, P. M. Rondon, and R. Jhala, ‘Abstract Refinement Types’, in *Programming Languages and Systems*, Springer Berlin Heidelberg, 2013, pp. 209–228. doi: 10.1007/978-3-642-37036-6_13.

[11] ‘The Checker Framework Manual: Custom pluggable types for Java’. Accessed: Dec. 28, 2024. [Online]. Available: https://checkerframework.org/manual

[12] D. Baier, D. Beyer, and K. Friedberger, ‘JavaSMT 3: Interacting with SMT Solvers in Java’, in *Computer Aided Verification*, Springer International Publishing, 2021, pp. 195–208. doi: 10.1007/978-3-030-81688-9_9.

[13] C. Barrett, P. Fontaine, and C. Tinelli, ‘The SMT-LIB Standard: Version 2.7’. Accessed: Mar. 11, 2025. [Online]. Available: https://smt-lib.org/

[14] J. Gosling *et al.*, ‘The Java® Language Specification: Java SE 21 Edition’, Aug. 2023. Accessed: Dec. 30, 2024. [Online]. Available: https://docs.oracle.com/javase/specs/jls/se21/html/index.html

[15] R. T. Boute, ‘The Euclidean definition of the functions div and mod’, *ACM Transactions on Programming Languages and Systems*, vol. 14, no. 2, pp. 127–144, Apr. 1992, doi: 10.1145/128861.128862.

[16] M. Brain, C. Tinelli, P. Ruemmer, and T. Wahl, ‘An Automatable Formal Semantics for IEEE-754 Floating-Point Arithmetic’, in *2015 IEEE 22nd Symposium on Computer Arithmetic*, IEEE, Jun. 2015, pp. 160–167. doi: 10.1109/arith.2015.26.

[17] ‘IEEE Standard for Floating-Point Arithmetic’. pp. 1–84, 2019. doi: 10.1109/IEEESTD.2019.8766229.

[18] H. Wang, ‘Logic of many-sorted theories’, *Journal of Symbolic Logic*, vol. 17, no. 2, pp. 105–116, Jun. 1952, doi: 10.2307/2266241.

[19] R. Stansifer, ‘Presburger’s Article on Integer Airthmetic: Remarks and Translation’, Sep. 1984. Accessed: Mar. 05, 2025. [Online]. Available: https://web.archive.org/web/20230527121900/https://cs.fit.edu/~ryan/papers/presburger.pdf

[20] K. Gödel, ‘Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I’, *Monatshefte für Mathematik und Physik*, no. 1, pp. 173–198, Dec. 1931, doi: 10.1007/bf01700692.

[21] L. de Moura and N. Bjørner, ‘Z3: An Efficient SMT Solver’, in *Tools and Algorithms for the Construction and Analysis of Systems*, Springer Berlin Heidelberg, 2008, pp. 337–340. doi: 10.1007/978-3-540-78800-3_24.

[22] J. Christ, J. Hoenicke, and A. Nutz, ‘SMTInterpol: An Interpolating SMT Solver’, in *Model Checking Software*, Springer Berlin Heidelberg, 2012, pp. 248–254. doi: 10.1007/978-3-642-31759-0_19.

[23] F. Lanzinger, A. Weigl, M. Ulbrich, and W. Dietl, ‘Scalability and precision by combining expressive type systems and deductive verification’, *Proceedings of the ACM on Programming Languages*, vol. 5, no. OOPSLA, pp. 1–29, Oct. 2021, doi: 10.1145/3485520.

[24] K. R. M. Leino, ‘This is Boogie 2’, Jun. 2008. Accessed: Feb. 28, 2025. [Online]. Available: https://www.microsoft.com/en-us/research/publication/this-is-boogie-2-2/

[25] P. M. Rondon, M. Kawaguci, and R. Jhala, ‘Liquid types’, in *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, in PLDI ’08. ACM, Jun. 2008. doi: 10.1145/1375581.1375602.

[26] N. Vazou, E. L. Seidel, and R. Jhala, ‘LiquidHaskell: experience with refinement types in the real world’, in *Proceedings of the 2014 ACM SIGPLAN symposium on Haskell*, in ICFP’14. ACM, Sep. 2014, pp. 39–51. doi: 10.1145/2633357.2633366.

[27] R. Abbasi, J. Schiffl, E. Darulova, M. Ulbrich, and W. Ahrendt, ‘Combining rule- and SMT-based reasoning for verifying floating-point Java programs in KeY’, *International Journal on Software Tools for Technology Transfer*, vol. 25, no. 2, pp. 185–204, Mar. 2023, doi: 10.1007/s10009-022-00691-x.