Scalable and Precise Refinement Types for Imperative Languages

Florian Lanzinger*[1], Joshua Bachmeier[2], Mattias Ulbrich[1], and Werner Dietl**[3]

[1] Karlsruhe Institute of Technology, Karlsruhe, Germany
[2] FZI Research Center for Information Technology, Karlsruhe, Germany
[3] University of Waterloo, Waterloo, Canada
lanzinger@kit.edu

Abstract. In formal verification, there is a dichotomy between tools which are scalable and easy to use but imprecise, like most pluggable type systems, and tools which are expressive and precise but badly scalable and difficult to use, like deductive verification. Our previous research to create a formal link between refinement types and deductive verification allows programmers to use a scalable type system for most of the program, while automatically generating specifications for a deductive verifier for the difficult-to-prove parts. However, this is currently limited to immutable objects. In this work, we thus present an approach which combines uniqueness and refinement type systems in an imperative language. Unlike existing similar approaches, which limit refinements such that they cannot contain any reference-typed program variables, our approach allows more general refinements, because even if a type constraint is not provable in the type system itself, it can be translated and proven by a deductive verifier.

Keywords: Pluggable type systems, Deductive verification, Refinement types, Ownership types

1 Introduction

There are many ways to prove a program's correctness, each with their own strengths and weaknesses. In deductive verification, a formal specification describing the expected behavior needs to be written and then proven using some partially interactive tool. Both the specification languages and the proof tools are often difficult to use for non-experts. In addition, the approach is time-consuming

* This work was supported by funding from the topic Engineering Secure Systems of the Helmholtz Association (HGF) and by KASTEL Security Research Labs. It was also supported by a Mitacs Globalink Research Award for the project "Scalable Verification of Imperative Programs."

** This work was supported by the KIT International Excellence Fellowships Program with funds granted to the University of Excellence concept of Karlsruhe Institute of Technology.

and scales badly with program size and complexity. Pluggable type systems [2], which annotate variables and sub-programs with types that describe their expected values and behavior, offer a light-weight alternative. However, most of these type systems only check for a conservative approximation of the formal guarantee they are designed for.

In our previous research [8], we presented a bridge between the two methods to combine the advantages of both: The developer first uses pluggable type systems to establish formal guarantees. If the type checker cannot show the well-typedness of the complete program, it emits a translated program in which all unproven typing constraints are turned into formal specifications for a deductive verifier, while all proven constraints are turned into auxiliary assumptions. Unlike existing refinement type systems, this combined approach allows us to state and prove refinements beyond the capabilities of a type checker or SMT solver while still retaining the benefits of scalable type checking where possible.

However, this is so far limited to immutable objects. Here, we present an extension of the approach to mutable objects by leveraging a uniqueness type system to deal with aliasing. In addition to uniqueness types, we use a *packing type system* based on an approach by Leino and Müller [11] to allow consistent updates in the presence of dependent types. We have formalized the approach and are currently working on a proof and implementation. Similar combinations of ownership/uniqueness and refinement types in imperative languages exist [10,12,15], but they limit refinements such that they cannot contain any reference-typed program variables. We can allow more general dependent refinements, because even if a type constraint cannot be shown to hold by the syntactic type rules, it can still be translated into a specification for the deductive verifier.

2 Combining Refinement Type Systems and Deductive Verification

```
boolean is_leq(@NonNull VarInfo v1, @NonNull VarInfo v2) {
  @Nullable Invariant inv = null; @Nullable Slice slice = null;
  slice = findSlice(v1, v2);
  if (slice != null) {
    inv = instantiate(slice);
  }
  if (inv != null) {
    boolean found = slice.is_inv_true(inv);
    return found;
  }
  return false;
}
```

Fig. 1. A false positive in the Nullness Checker.

```
//@ requires v1 != null && v2 != null;
boolean is_leq(VarInfo v1, VarInfo v2) {
  //@ assume v1 != null && v2 != null;
  Invariant inv = null; Slice slice = null;
  slice = findSlice(v1, v2);
  if (slice != null) {
    //@ assume slice != null;
    inv = instantiate(slice);
  }
  if (inv != null) {
    //@ assume inv != null;
    //@ assert slice != null;
    boolean found = slice.is_inv_true(inv);
    return found;
  }
  return false;
}
```

Fig. 2. JML translation of Figure 1

This section summarizes our previous work [8]. Figure 1 is an example taken from that paper; it shows a slightly modified code snippet from the Daikon Invariant Generator [5], which uses the Nullness Checker of the Checker Framework [3], a framework for pluggable Java type systems. The checker detects a possible null-pointer exception for the method invocation on `slice`. This is a false positive because the implementation ensures that if `inv` is non-null, `slice` is also non-null. Our approach allows us to use a deductive verifier to avoid the false positive being reported: We associate each type qualifier with a *refinement*, a logical formula expressing its semantics. Here, `NonNull`'s refinement is $\texttt{subject} \neq \texttt{null}$, where `subject` stands for the typed variable, and `Nullable`'s refinement is `true`. Since the former formula implies the latter, we are allowed to define the subtyping hierarchy $\texttt{NonNull} \preceq \texttt{Nullable}$. We then translate the type constraints into specifications in the Java Modeling Language (JML) [9], as seen in Figure 2. There, instead of working with the type qualifiers and their predefined hierarchy, we work directly with the underlying formulas. The type constraint that the checker was unable to show is turned into an assertion, while the other constraints are turned into assumptions. This JML translation can be given to a deductive verification tool like KeY [1]. If all assertions in the translated program are valid, then we know that the putative type error is indeed a false positive.

Using our approach, only the parts of a program's central logic which establish and change the type properties in complex ways have to be considered for deductive verification, while the high-level parts which mostly just preserve the properties or rely on lower-level methods can be discharged by the type checker, which significantly lowers the verification overhead [7,8].

3 Refinement Types for Mutable Objects

This section summarizes our current work on extending what was presented in Section 2 to work with mutable objects. The two main issues we must consider are aliasing, for which we introduce a uniqueness type system in Section 3.1, and consistent updates in the presence of dependent types, for which we introduce a packing type system in Section 3.2. In Section 3.3, we then state the basic ideas needed to formalize and prove our approach.

3.1 Combining Uniqueness and Refinement Types

```
@MinLength(5) List a; @MaxLength(5) List b;
@MinLength(b.length) List c;
void foo() { a.insert(42); }
```

Fig. 3. Aliasing issue with refinement types.

To be able to apply refinement types to mutable objects, we need some way to restrict reference aliasing. Consider the example in Figure 3. It contains the type qualifiers `MinLength`(n) and `MaxLength`(n) with the refinements `subject.length` $\geq n$ and `subject.length` $\leq n$ respectively. The method call `a.insert(42)` seems to be well-typed if we only consider a's type; but if a and b are aliases, it violates b's type. In addition, it may violate c's dependent type.

There are many type systems we could use to alleviate this problem. We settled on a simple system with the hierarchy `Unique` $\preceq$ `MaybeAliased` $\preceq$ `ReadOnly`, where `MaybeAliased` references may have other `MaybeAliased` aliases, but `Unique` references can only have `ReadOnly` aliases, which can never be used to mutate the object and may never occur in a refinement. Our system also supports ownership transfer mechanisms such as borrowing: A method argument has both an input and an output type, so for example a method with the signature `foo(@Unique -> @Unique List arg)` receives a unique reference, but must preserve the uniqueness so it can later be returned to the caller.

This system is quite restrictive. For example, it is impossible to have a well-typed mutable cyclic list. However, this restrictiveness has one major advantage that will be explained in Section 3.2. In the future, we may investigate using a more powerful type system like Universe Types [4]. Another possible avenue is to do with uniqueness types what we did with refinement types, i.e., define a translation into a specification language for deductive verifiers, such that ownership structures too complex to be proven by the type system can still be verified. The RustBelt project [6] already does something similar for Rust's ownership types.

In any case, our uniqueness types solve the aliasing problem with refinement types, as long as we abide by the following restrictions: Refinements may only appear on unique references. A refinement may only depend on other unique references in the current scope. A reference may only be mutated or transferred if doing so cannot violate any refinement dependent on it. E.g., if we make every reference in Figure 3 unique, the method call becomes well-typed: We know that a is unique and that no refinement in the current scope depends on it[4]; thus no other reference's type is violated.

3.2 Consistent Updates with Packing Types

```
class List {
  @NonNegative int length;
  @Unique @Length(this.length) Node head;
  public void insert(@Unique -> @Unique List this,
                     int newDatum) {
    head = new Node(newDatum, head); ++length; }
```

Fig. 4. Keeping the fields consistent is impossible.

However, Figure 4 shows a problem with our approach so far. No matter in what order we execute the two statements in the insert method, we will temporarily violate `head`'s type!

To alleviate this, we use a variant of the approach by Leino and Müller [11]: For an object o that is *packed* up to the class τ, all fields defined in τ and its superclasses must respect their declared types. In addition, these fields are unassignable and immutable. Fields defined in subclasses of τ in contrast can be reassigned and mutated and need not respect their declared types. An object's packing state can be changed via statements of the form `pack` o `as` τ and `unpack` o `from` τ. Packing an object is only allowed if all involved fields respect their declared types. Unlike Leino and Müller, who use special run-time fields to remember an object's packing state, we use a pluggable type system, i.e., a reference is packed up to τ if it has the type qualifier `Packed`(τ). To ensure that every non-read-only reference to the same object has the same packing type (a read-only reference's packing type is irrelevant, as it cannot be mutated or appear in refinements anyway), we only allow (un-)packing unique references. This means that the packing type system now serves double duty as an immutability type system: Since maybe-aliased references cannot be (un-)packed, such references with type `Packed`(τ) are immutable up to τ; thus we can use their state up to τ in refinements.

The problem in our example can be solved by inserting `unpack this from List` at the beginning and `pack this as List` at the end of the insert method. That pack statement is well-typed if the type checker can show that every field

[4] Actually, we cannot know whether this is true in the presence of subclassing, but this is dealt with in the following section.

respects its declared type at that point in the program. If it cannot, then, as usual, an assertion to that effect can be inserted into the program to be shown by a deductive verifier.

But suppose that our list has additional fields (e.g., tail, cache, etc.). Then to show the well-typedness of the pack statement, we would have to prove that every single field respects its declared type, even though we have only changed two. This is where our restrictive uniqueness type system comes in useful: A unique reference that has not been transferred cannot have been mutated; thus it is still in the same state it was in when the receiver was unpacked.

This packing type system allows us to relax the restrictions on refinements from the preceding subsection. Now, the refinement of a field or local variable in a class τ is allowed to depend on the abstract state $A(\mathtt{this}, \tau)$, which includes:

1. Every field declared in τ or a superclass.
2. For every `Unique` or `MaybeAliased` field f declared with type $\mathtt{Packed}(\tau_f)$ in τ or a superclass: $A(\mathtt{this}.f, \tau_f)$.

3.3 Formalization and Proof Idea

For the proof, we are considering an approach similar to the one by Timany et al. [14] who differentiate between *syntactically well-typed* and *semantically well-typed* statements: The former holds if the statement respects the syntactic typing rules. The latter holds if the statement preserves well-typed states (in our case, that it preserves all refinement formulas, as well as the uniqueness and packing properties), even if this cannot be shown by the syntactic type checker. This formal framework fits nicely with our approach of encoding properties that cannot be shown by the type checker as assertions.

4 Conclusion and Outlook

We presented an approach which combines the scalability of type systems with the precision of deductive verification by translating those type constraints which a type checker was unable to show into specifications for a deductive verifier. By leveraging additional guarantees given by a uniqueness and a packing type system, we can apply this approach to languages with mutable objects.

We have formalized the approach and are working on an implementation and proof. We will build on the implementation of our previous work [8], which used the Checker Framework [3] and KeY [1]. For the proof, we are considering either using the Iris framework by Timany et al. [14], or Steinhöfel's work on extending KeY with abstract programs [13], which has the advantage that KeY already contains a complete formalization of our target language Java, but the disadvantage that KeY's dynamic frames are much more cumbersome than Iris's separation logic to reason about ownership and framing properties. We also plan to evaluate the approach by using it to verify a piece of software that uses the Checker Framework: Using our approach, we should be able to discharge any suppressed warnings using KeY; also, our combined approach should be easier and faster than using KeY alone.

References

1. Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P.H., Ulbrich, M. (eds.): Deductive Software Verification - The KeY Book - From Theory to Practice, Lecture Notes in Computer Science, vol. 10001. Springer (2016). `https://doi.org/10.1007/978-3-319-49812-6`
2. Bracha, G.: Pluggable type systems. In: OOPSLA'04 Workshop on Revival of Dynamic Languages (Oct 2004)
3. Dietl, W., Dietzel, S., Ernst, M.D., Muslu, K., Schiller, T.: Building and using pluggable type-checkers. In: Proceedings of the 33rd International Conference on Software Engineering. pp. 681–690. ICSE 2011, Association for Computing Machinery (05 2011). `https://doi.org/10.1145/1985793.1985889`
4. Dietl, W., Ernst, M.D., Müller, P.: Tunable Static Inference for Generic Universe Types. In: Mezini, M. (ed.) European Conference on Object-Oriented Programming (ECOOP). pp. 333–357. Springer Berlin Heidelberg (Jul 2011). `https://doi.org/10.1007/978-3-642-22655-7_16`
5. Ernst, M.D., Perkins, J.H., Guo, P.J., McCamant, S., Pacheco, C., Tschantz, M.S., Xiao, C.: The Daikon system for dynamic detection of likely invariants. Science of Computer Programming **69**(1–3), 35–45 (Dec 2007)
6. Jung, R., Jourdan, J.H., Krebbers, R., Dreyer, D.: RustBelt: Securing the foundations of the rust programming language. Proceedings of the ACM on Programming Languages **2**(POPL) (Dec 2017). `https://doi.org/10.1145/3158154`
7. Klamroth, J., Lanzinger, F., Pfeifer, W., Ulbrich, M.: The Karlsruhe Java Verification Suite, pp. 290–312. Springer International Publishing (Jul 2022). `https://doi.org/10.1007/978-3-031-08166-8_14`, `https://doi.org/10.1007/978-3-031-08166-8_14`
8. Lanzinger, F., Weigl, A., Ulbrich, M., Dietl, W.: Scalability and precision by combining expressive type systems and deductive verification. Proc. ACM Program. Lang. **5**(OOPSLA) (oct 2021). `https://doi.org/10.1145/3485520`
9. Leavens, G.T., Poll, E., Clifton, C., Cheon, Y., Ruby, C., Cok, D., Müller, P., Kiniry, J., Chalin, P., Zimmerman, D.M., Dietl, W.: JML reference manual (May 2013), `http://www.eecs.ucf.edu/~leavens/JML//refman/jmlrefman.pdf`, revision 2344
10. Lehmann, N., Geller, A., Barthe, G., Vazou, N., Jhala, R.: Flux: Liquid types for Rust (2022). `https://doi.org/10.48550/ARXIV.2207.04034`, `https://arxiv.org/abs/2207.04034`
11. Leino, K.R.M., Müller, P.: Object invariants in dynamic contexts. In: Odersky, M. (ed.) ECOOP 2004 – Object-Oriented Programming. pp. 491–515. Springer Berlin Heidelberg, Berlin, Heidelberg (2004)
12. Sammler, M., Lepigre, R., Krebbers, R., Memarian, K., Dreyer, D., Garg, D.: Refinedc: Automating the foundational verification of C code with refined ownership types. In: Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation. p. 158–174. PLDI 2021, Association for Computing Machinery, New York, NY, USA (2021). `https://doi.org/10.1145/3453483.3454036`
13. Steinhöfel, D.: Abstract Execution: Automatically Proving Infinitely Many Programs. Ph.D. thesis, Technische Universität, Darmstadt (2020). `https://doi.org/https://doi.org/10.25534/tuprints-00008540`, `http://tuprints.ulb.tu-darmstadt.de/8540/`
14. Timany, A., Krebbers, R., Dreyer, D., Birkedal, L.: A logical approach to type soundness (2022), `https://iris-project.org/pdfs/2022-submitted-logical-type-soundness.pdf`

15. Toman, J., Siqi, R., Suenaga, K., Igarashi, A., Kobayashi, N.: ConSORT: Context- and flow-sensitive ownership refinement types for imperative programs. In: Müller, P. (ed.) Programming Languages and Systems. pp. 684–714. Springer International Publishing, Cham (2020). https://doi.org/10.1007/978-3-030-44914-8_25