Primrose: Selecting Container Data Types by their Properties

Xueying Qin[a], Liam O'Connor[a], and Michel Steuwer[a]

a The University of Edinburgh, Scotland, United Kingdom

Abstract

Context Container data types are ubiquitous in computer programming, enabling developers to efficiently store and process collections of data with an easy-to-use programming interface. Many programming languages offer a variety of container implementations in their standard libraries based on data structures offering different capabilities and performance characteristics.

Inquiry Choosing the *best* container for an application is not always straightforward, as performance characteristics can change drastically in different scenarios, and as real-world performance is not always correlated to theoretical complexity.

Approach We present Primrose, a language-agnostic tool for selecting the best performing valid container implementation from a set of container data types that satisfy *properties* given by application developers. Primrose automatically selects the set of valid container implementations for which the *library specifications*, written by the developers of container libraries, satisfies the specified properties. Finally, Primrose ranks the valid library implementations based on their runtime performance.

Knowledge With Primrose, application developers can specify the expected behaviour of a container as a type refinement with *semantic properties*, e.g., if the container should only contain unique values (such as a set) or should satisfy the LIFO property of a stack. Semantic properties nicely complement *syntactic properties* (i.e., traits, interfaces, or type classes), together allowing developers to specify a container's programming interface *and* behaviour without committing to a concrete implementation.

Grounding We present our prototype implementation of Primrose that preprocesses annotated Rust code, selecting the best performing container implementation. The design of Primrose is, however, language-agnostic, and is easy to integrate into other programming languages that support container data types and traits, interfaces, or type classes. Our implementation encodes properties and library specifications into verification conditions in Rosette, an interface for SMT solvers, which determines the set of valid container implementations. We evaluate Primrose by specifying several container implementations, and measuring the time taken to select valid implementations for various combinations of properties with the solver. We automatically validate that container implementations confirm to their library specifications via property-based testing.

Importance This work provides a novel approach to bring abstract modelling and specification of container types directly into the programmer's workflow. Instead of selecting concrete container implementations, application programmers can now work on the level of specification, merely stating the behaviours they require from their container types, and the best implementation can be selected automatically.

ACM CCS 2012

- **Software and its engineering → Functionality; Software functional properties;**
- *Theory of computation → Program specifications;*

Keywords Container Data Types, Properties, Data Abstraction, Performance

The Art, Science, and Engineering of Programming

Perspective The Art of Programming

Area of Submission Programming language, Model-based development

1 Introduction

Container data types, such as sets, lists, and trees, represent collections of data ubiquitous in everyday programming [9]. Virtually all programming languages provide a variety of different container implementations in their standard libraries.

Much work has been done to design better abstractions, improve performance and verify correctness for container data types. However, a crucial problem for application developers using containers still exists: when choosing a container data type, application developers are forced to select a concrete implementation that comes with certain theoretical complexity and practical performance tradeoffs.

For example, consider representing a mathematical set, i.e., where each element should occur at most once. In C++, we must choose between `std::set`, usually implemented as red-black trees [3], and `std::unordered_set`, implemented as a hash table. The hash-based implementation was added to the C++ standard in 2011, as the C++ standard has strict complexity requirements preventing the ordinary `std::set` to be implemented as the (often faster) hash table. Many blog posts and discussions [2, 5, 11, 23, 35] report on the performance of various C++ containers, showing the community's interest and the need for external guidance that the language itself does not provide.

In other languages, the situation is similar. Rust provides two container implementations, `HashSet` and `BTreeSet`, expecting application developers to make an explicit choice between them. Scala's complex collection library features abstract interfaces, such as the `Set` trait, abstracting over many implementations such as `HashSet` and `TreeSet`. But when creating an instance of `Set`, a default `HashSet` implementation is chosen regardless of the suitability of this implementation choice for the usage pattern of the application.

These examples demonstrate a general problem: Application developers are forced to *overspecify*, by having to select a concrete implementation, where we generally would like application developers to be shielded from low-level implementation details. Application developers should primarily care about the *abstract behaviour* of the containers in their application, and not how this is achieved. The compiler, or a dedicated tool, should identify those containers that satisfy their functional requirements, and select the best implementation automatically.

In this paper, we propose such an automated tool: Primrose, which allows application developers to specify the expected behaviours and programming interfaces of containers as *properties*. *Syntactic properties* specify the required programming interface of the container and are expressed as traits of the underlying programming language. *Semantic properties* specify the expected behaviour of the container and are written as logical predicates used as refinements of the container type. Primrose automatically selects the set of valid implementations for which the *library specifications*, written by the library developers as pre- and post-conditions of the container operations, satisfy the specified syntactic and semantic properties using an SMT solver. Finally, Primrose ranks the valid library implementations based on their runtime performance.

In this work, we apply verification and formal methods techniques, including refinement types, formal library specifications, and SMT solvers, in an innovative way to raise the level of abstraction for developers, freeing them from the burden of choosing container implementations, and to improve the performance of applications.

To summarize, this paper makes the following contributions:

- We present Primrose (Sec. 3), a language-agnostic tool for selecting valid container implementations (Sec. 6) based on *properties* (Sec. 4) used to describe their behaviour and programming interface, and ranking them based on their performance.
- We show a new application of refinement types (Sec. 4) not—as previous work did—for verification purposes, but to raise the level of abstraction for developers and to improve the runtime performance of applications with container data types.
- We develop a new methodology to specify container libraries (Sec. 5), amenable to our selection process, making use of existing formal methods work such as data abstraction and Hoare logic.
- We show the feasibility of Primrose, selecting container implementations that satisfy various properties from a Rust library of eight container types with library specifications. We validate container implementations against specifications and evaluate the efficiency of the selection process (Sec. 7).

2 Motivation

Suppose as part of a larger application we want to find and store all the elements of a larger collection, but without duplicates. We might, for example, use the result of this function to count the number of unique elements or process the elements further, now with the guarantee that each element in the returned collection is unique.

An easy way to implement this is to return a container that only permits unique elements. We might think of a *set*, but as discussed in section 1, this requires a choice: Which implementation of the abstract idea of a mathematical set should we use?

Figure 1a shows a Rust code snippet computing a container `uniqueElements` that contains the unique elements of the original `input` sequence. The application developer must choose a concrete container implementation, such as `HashSet` in line 1, but other valid choices would be Rust's `BTreeSet` (line 2), or perhaps a custom `UniqueVect` (line 3) container, which stores all elements in a vector but ensures there are no duplicates,

```
1 type Set<I> = HashSet<I>;
2 // type Set<I> = BTreeSet<I>;
3 // type Set<I> = UniqueVect<I>;
4 // type Set<I> = FancySetImpl<I>;
5 // type Set<I> = HashMultiSet<I>; ???
6
7 let mut uniqueElements = Set::new();
8 for val in input.iter()  {
9     uniqueElements.insert(val);  }
```

(a) In Rust, application developers must choose a concrete container implementation with potentially surprising performance implications.

```
1 property unique {
2   c –> for–all–elems c (\a –>
3                   (unique–count? a c)) };
4 type UniqueCon<I> = {
5   c <:  ContainerT | unique c };
6
7 let mut uniqueElements = UniqueCon::new();
8 for val in input.iter()  {
9     uniqueElements.insert(val);  }
```

(b) Using Primrose, developers describe the container's expected behaviour via *properties* and the best valid implementation is selected.

Figure 1 Selecting the unique elements of a sequence by inserting the elements into a *set*.

or some other FancySetImplementations (line 4). Whether a container implementation is *valid* is determined by the application developer's *functional requirements*. Our uniqueness requirement, for example, is not met by the Rust HashMultiSet (line 5).

Many programming techniques exist to abstract over multiple concrete implementations of a general concept. In object-oriented languages, *abstract classes* enable hiding multiple implementations behind a common interface. Similar features exist in other languages under different names, such as, *traits* (e.g., in Rust and Scala), *protocols* (e.g., in Swift), *interfaces* (e.g., in Java), and *type classes* (e.g., in Haskell). All these techniques allow developers to use multiple concrete implementations, such as HashSet and BTreeSet, with a single abstract type, which we might call Set. However, these types are deliberately *abstract*, meaning that we *cannot* instantiate them directly: When creating such a type, a developer must commit to a specific concrete implementation, requiring the developer to look underneath layers of abstraction to make an informed decision. Thus, these abstraction techniques do not free developers from considering low level details and they are not powerful enough to express *semantic* requirements: developer cannot specify their functional requirements directly, but merely provide a common *syntax* enabling the use of multiple implementations. With such abstract container type Set, we cannot express that each concrete implementation is required to contain no duplicate elements. Similarly, with an abstract type Stack, we cannot state that the last-in-first-out property is respected by the push and pop operations.

Figure 1b shows the same problem of selecting unique elements, but expressed using Primrose. Application developers specify their requirements—in this case, that the container must contain unique elements—as a *semantic property* in lines 1–3. This semantic property is expressed as a logical predicate that is used to *refine* the container data type in lines 4 and 5. Refinement types have long been used as a technique for program verification—including container types [33]. Here, we use refinement types in a new way, allowing programmers to express the expected behaviour of a container, and freeing them from having to make a (potentially difficult) implementation choice. The remaining code remains unchanged: we can simply use the refined type in line 7. Primrose preprocesses the code from figure 1b, identifies all valid container

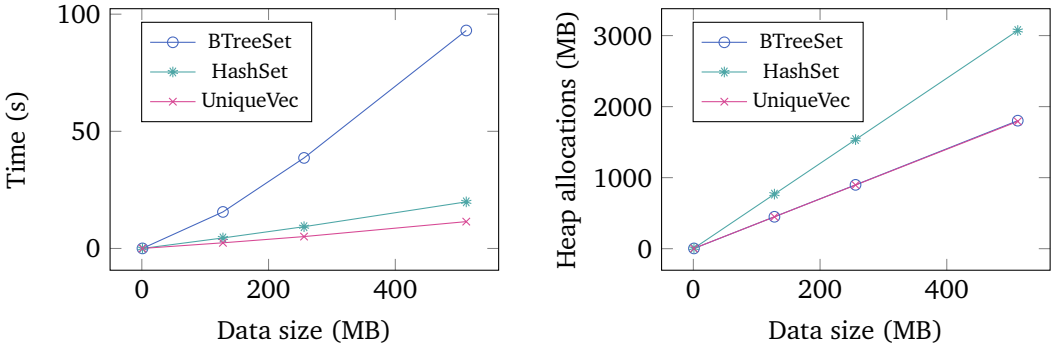


Figure 2 Runtime performance (left) and memory consumption (right) of three container implementations for storing unique elements of an input sequence from figure 1a. The custom UniqueVec implementation ensures elements to be unique lazily on access. It is the fastest implementation, outperforming HashSet and BTreeSet from the Rust standard library, while consuming less memory than HashSet.

implementations from a library of containers, and generates a program equivalent to figure 1a with the best container implementation inserted automatically.

But which is the *best* container implementation? This depends on the non-functional requirements of the application: Often developers care about fast runtime performance, also, for example, an application might require a low memory footprint. Figure 2 shows the performance and memory consumption for three different implementation choices. Perhaps surprisingly, a custom `UniqueVec` implementation that uses a vector and lazily ensures that the stored elements are unique, by sorting the vector and removing duplicates on access outperforms the Rust built-in containers `HashSet` and `BTreeSet`. In addition, it is also the best choice for machines with limited memory. Choosing the best container implementation is not always straightforward, particularly as theoretical complexity of operations can sometimes be misleading in the presence of practical effects such as cache-friendliness. Using Primrose, developers do not have to worry about choosing an implementation which is incorrect or has subpar performance.

3 Overview

Figure 3 gives an overview of the design of the Primrose selection tool. Using Primrose, the application developer writes code in terms of an abstract type, and a *property specification* describing the syntactic and semantic properties they expect this type to satisfy. The syntactic properties take the form of traits and the semantic properties take the form of type refinements. To write a program, the application developer only

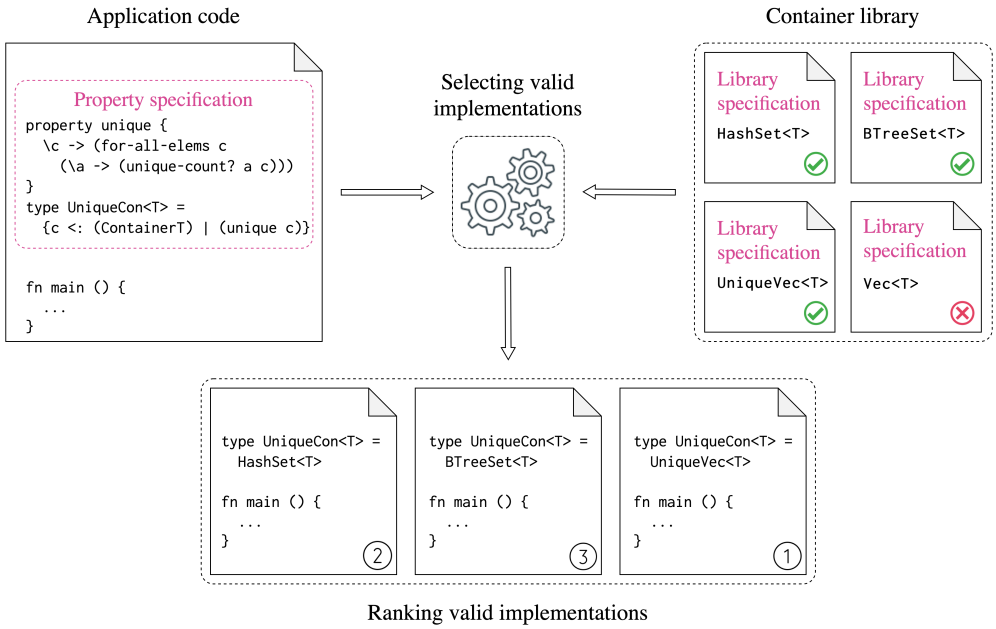


Figure 3 The workflow of Primrose: *Property specifications* (top left), written and used by the application developer, are used to check which *library specifications* (top right), written by library developers, satisfy them. Valid implementations (marked with a green check marks), are then ranked by their performance (bottom).

specify what properties must be satisfied by the required container, and does not have to commit to a particular implementation. In figure 3, the developer specifies that they require a container (the syntactic property `ContainerT`) where all elements are unique (the semantic property `unique`). We discuss properties in detail in section 4.

Given this code as input, Primrose will, acting as a preprocessor, generate copies of the input code where the abstract type is instantiated into a valid concrete implementation that satisfies the expected properties. It determines which implementations are valid by consulting *library specifications*, which are provided by library developers. These specifications abstract over concrete container implementations and provide a summary of their externally observable semantics. For each implementation, the library specification contains the pre- and post-conditions of each operation in terms of an abstract list model. We discuss these specifications in more detail in section 5.

In our example in figure 3, the library specification of the `Vec<T>` type indicates that it is not a suitable choice for `UniqueCon<T>`, as it does not satisfy the required semantic property `unique`. We use a satisfiability modulo theories (SMT) solver for this selection process, specifically using the solver-aided programming language Rosette [31, 32] interacting with Z3 solver. We discuss the selection process in section 6.

Figure 3 shows at the bottom a simplified version of the generated programs. In our implementation, we ensure that only the container operations that the application developer specifies with syntactic properties are accessible in the generated program. Our current prototype of Primrose focuses on ensuring the functional correctness of selecting container implementations based on desired properties. Nevertheless, we have implemented a simple process that ranks valid implementations by their runtime performance. Rankings by other non-functional metrics could easily be added to our design. We provide discussion about code generation and ranking in section 6.4.

Portability of Primrose Currently, we choose Rust as the target language to implement our idea. Application developers write the property specifications as a part of their Rust programs and Primrose generates Rust code after processing specifications. However, Primrose could easily be ported to many other languages, since property specifications, library specifications, and the process of selecting implementations are all language-agnostic and not attached to Rust's particular type system or language features. Adapting property specifications into other languages only requires such languages to have a construct similar to Rust's traits, such as traits in Scala and interfaces in Java, allowing us to model syntactic properties. It would be straightforward to add new backends to Primrose to generate code in these languages. Our library specifications are, by design, an abstraction over implementation details, describing the intended semantics of container operations without respect to their implementation. This means we can trivially adapt these specifications to container libraries from other languages, so long as our specifications remain an abstraction of the new implementations. Thus, we anticipate that Primrose could easily be adapted to produce code in any language with sufficient support for data abstraction, such as Java, Scala, Swift, or C++.

4 Property Specifications

The application developer specifies the desired behaviours of their required container with a *property specification* that consists of *semantic properties*, which refine the container type by a predicate, and *syntactic properties*, which in Rust are traits specifying the operations that must be supported by the container and their types.

The property specification of the type UniqueCon from the example in figure 3 is:

```
property unique { \c -> (for-all-elems c (\a -> (unique-count? a c))) }
type UniqueCon<T> = {c <: (ContainerT) | (unique c)}
```

We first define the semantic property unique using a *predicate*. In our specification language, such predicates have type $Con\langle\tau\rangle \to Bool$, where $Con\langle\tau\rangle$ is a placeholder that is resolved into a concrete container type by the selection process. The combinator for-all-elems is part of a library enabling to write predicates for individual container elements. The predicate unique-count? holds iff the given element occurs exactly once in the container. These combinators and predicates are explained in Section 4.2.

With the defined semantic property unique, we can then declare the container type UniqueCon<T>. The first part of the declaration specifies the syntactic property that must be satisfied by the container type, in the form of the trait ContainerT. Specifically, c <: (ContainerT) says that the type of the container c must implement the trait ContainerT, which specifies a set of basic container operations. The second part of the declaration *refines* our container type by the predicate unique, stating that the property must be invariant across all container operations. Properties may also be composed. For multiple syntactic properties, we specify a list of traits (c <: (T1, T2)) that the container type implements. For multiple semantic properties, we use conjunction, i.e. ((p1 c) and (p2 c)).

Figure 4 shows the syntax of property specifications in Primrose. Formally, the specification language is a variant of the polymorphic λ-calculus [14, 24], with restrictions on the use of polymorphism to enable implicit type inference [16, 22]. This type system guarantees termination, making specifications easier to analyse and straightforward to translate into SMT verification conditions.

4.1 Syntactic Properties as Traits

In our Primrose prototype, we encode syntactic properties as Rust traits, specifying the operations needed by the application developer to interact with a container. Traits

$$
\begin{array}{rrcl}
\text{Literals} & l & := & \mathit{true} \mid \mathit{false} \\
\text{Terms} & t & := & l \mid x \mid \lambda x.\, t \mid t\ t \\
\text{Refinement} & r & := & t \mid r \wedge r \\
\text{Container Type Declarations} & c & := & \{v <: B \mid r\} \\
\text{Simple Types} & \sigma & := & \mathit{Bool} \mid T \mid \mathit{Con}\langle\sigma\rangle \\
\text{Types} & \tau & := & \sigma \mid \tau \to \tau \mid \forall T <: B.\, \tau \\
\text{Bounds} & B & := & \mathit{trait\_name} \mid B\ ,\ B
\end{array}
$$

Figure 4 The syntax of property specifications

are defined in Rust and lifted into our property specification language. For instance, the trait ContainerT introduced above is implemented as:

```
pub trait ContainerT<T> {
  fn len(&self) -> usize;
  fn contains(&self, x: &T) -> bool;
  fn is_empty(&self) -> bool;
  fn insert(&mut self, elt: T);
  fn clear(&mut self);
  fn remove(&mut self, elt: T) -> Option<T>;
}
```

By writing c <: ContainerT, the application developer indicates that they expect the container type selected by Primrose to include implementations for all operations in the trait ContainerT. Thus, after executing Primrose, UniqueCon<T> will be resolved into a concrete container type that implements the trait ContainerT.

As mentioned, we can also declare a container type that satisfies multiple syntactic properties. For instance, suppose that in addition to ContainerT, we would like our container to also satisfy the syntactic property IndexableT:

```
pub trait IndexableT<T> {
  fn first(&self) -> Option<&T>;
  fn last(&self) -> Option<&T>;
  fn nth(&self, n: usize) -> Option<&T>;
}
```

With just ContainerT, there is no way to observe the *ordering* of elements in the container, but with IndexableT there is, as we can now select elements based on their position. By composing our new syntactic property IndexableT with ContainerT we can now specify a container of unique elements where the order can be observed:

```
type UniqueIndexableCon<T> = {c <: (ContainerT, IndexableT) | (unique c)}
```

Semantic properties, such as unique, must be invariant across all operations from all syntactic properties required of the container.

4.2 Semantic Properties as Predicates

As mentioned, semantic properties are predicates that are used to construct refinements for container types; each declared container type in the form $\{v <: B \mid r\}$ is a *refinement type*, i.e. a type circumscribed by a logical predicate [12]. Refinement types are also present in programming languages like Liquid Haskell and F*, where they are used to facilitate verification of program correctness. For instance, in Liquid Haskell, we may define a refinement type UniqueList representing a list of unique elements as:

```
{-@ measure unique @-}
unique :: (Ord a) => [a] -> Bool
unique [] = True
unique (x:xs) = unique xs && not (S.member x (elts xs))
{-@ type UniqueList a = {v:[a] | unique v} @-}
```

While our syntax for type refinements strongly resembles Liquid Haskell, our refinement types are slightly different, and serve a different purpose. Firstly, Liquid Haskell's refinements are attached to a *concrete type*, in this case a list (written [a]), whereas our refinements are attached to an abstract container type, which is then

resolved by Primrose into a concrete implementation. Secondly, Liquid Haskell uses type refinements for the purpose of *correctness*: If a list is declared to have type UniqueList, the Liquid Haskell verifier will check that it satisfies the predicate unique. For example, it will report an error at compile time if given a list that contains duplicates. Our work instead uses type refinements to specify the semantic requirements of the application developer to guide selection of valid concrete implementations. Once all valid implementations have been found, Primrose simply selects the implementation providing the best performance for the application developer. In short, rather than to aid verification, we use refinement types to help application developers optimise their programs. We give more details on the selection process in section 6.

Combinators and Predicate Functions Demonstrated by our examples, Primrose provides a set of combinators and predicate functions to facilitate writing of property specifications. These combinators and predicate functions are defined in Rosette and then imported into our property specification language. In the semantic property unique, the combinator for–all–elems is used to specify that the predicate unique–count? must hold for all elements inside the container. The type of the combinator for–all–elems is $Con\langle\tau\rangle \to (\tau \to Bool) \to Bool$, meaning this combinator takes in two arguments, the first of which is a container and the second of which is a predicate on the elements of that container, and eventually returns a boolean value.

For the purposes of checking, we represent containers $Con\langle\tau\rangle$ abstractly in Rosette as lists. We discuss this list abstraction and justify it in Section 5. This means that we can implement our for–all–elems combinator straightforwardly with a list fold operation:

```
(define (for–all–elems c fn)
    (foldl  elem–and #t (map (lambda (a) (fn a)) c)))
```

We also provide some combinators for applying *relations* between elements in a container. For instance, for-all-consecutive-pairs:

$$\text{for-all-consecutive-pairs} \;:\; Con\langle\tau\rangle \to (\tau \to \tau \to Bool) \to Bool \tag{1}$$

Unlike for–all–elems, this combinator is given a binary relation between elements, and checks that this relation holds between any two consecutive elements in our container.

With this combinator and the predicates geq? and leq?, we can define properties like ascending and descending, which specify particular orderings of elements in a container:

```
property ascending { \c –> (for–all–consecutive–pairs c leq?) }
property descending { \c –> (for–all–consecutive–pairs c geq?) }
```

Beside the set of combinators and predicate functions predefined in Primrose, application developers may also provide customised functions by providing Rosette definitions and importing them into our property specification language.

Composition of semantic properties As shown in figure 4, we can compose semantic properties in a container type declaration with conjunction. For instance, to declare a container type with elements arranged in *strictly* ascending order, i.e., both unique and ascending properties must hold, we can write the following:

```
type StrictlyAscendingCon<T> = {c <: (ContainerT) |  ((unique c)  and (ascending c))}
```

This conjunction is straightforwardly translated into a conjunction operation in Rosette.

4.3 The Interaction between Semantic and Syntactic Properties

All semantic properties we have seen so far have been invariants across all operations, but some semantic properties depend on specific operations given by syntactic properties. For instance, when specifying a stack container type providing operations `push` and `pop` with the expected last-in-first-out (LIFO) property. Firstly, we define a trait specifying operations `push` and `pop`, namely `StackT`:

Listing 1 The trait `StackT` specifying operations `push` and `pop`

```
pub trait StackT<T> {
  fn push(&mut self, elt: T);
  fn pop(&mut self) -> Option<T>;
}
```

Secondly, we define the semantic property `lifo` for containers that implement `StackT`:

Listing 2 The semantic property LIFO

```
property lifo { \c <: StackT -> (forall \x. pop (push c x) == x) }
```

Unlike previously, this semantic property includes a requirement that the given container implement the trait `StackT`, which enables us to refer to the operations `pop` and `push` inside the semantic property. In this definition, `forall` is a combinator with type:

$$\texttt{forall} \ : \ (\forall x.x \rightarrow \mathit{Bool}) \rightarrow \mathit{Bool} \tag{2}$$

This combinator is implemented with the `forall` procedure defined in Rosette's library, which serves as a construct for creating universally quantified formulae.

Armed with the trait `StackT` and the semantic property `lifo`, we can combine all these elements and declare our stack type as follows:

```
type StackCon<T> = {c <: (ContainerT, StackT) | (lifo c)}
```

In the next section, we will discuss how library developers write specifications for their container implementations.

5 Library Specifications

It is not feasible to select container implementations directly by analysing their Rust source code and checking if they satisfy the properties specified by the application developer. Rust is a Turing-complete, general purpose programming language with a complex semantics, and its container libraries are typically highly optimised, making extensive use of unsafe code. Doing such a broad analysis precisely and automatically is very hard even for the most advanced of static analyses. Instead, we write *library specifications* which abstract over the Rust implementations, providing a clear definition of intended semantics of each operation, without respect to performance or implementation details. This approach allows us to select container implementations by simply checking their library specifications, rather than their full implementations, against the properties specified by the application developer. Moreover, using specifications which are abstracted from implementations makes Primrose easy to repurpose

for programming languages other than Rust, as the same specifications would apply, with minimal or no modification, to container libraries written in any other language.

Since these library specifications form a *functional correctness* specification for each operation, we can use these specifications to ensure the correctness of our library implementations and the soundness of our library specifications for the selection process, by encoding these specifications into *property based testings*, which validate container implementations against our library specifications (section 7.1). Such specifications could also be used in future as the basis of full functional correctness verification with a verification framework for Rust [21], but this is out of scope for Primrose.

5.1 The Basic Design of Library Specifications

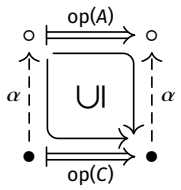
Library specifications of concrete container implementations are developed based on Hoare logic [17]. For each concrete container implementation, we provide a set of *Hoare triples*, one for each operation. A Hoare triple of the form $\{\phi\}$ op $\{\psi\}$ states that if the *precondition* ϕ holds and the operation op is executed, then the *postcondition* ψ will hold. These conditions are predicates on the state of the program. In our case, the state contains the container, plus any other inputs and outputs of the operation op.

As mentioned in section 3, we model the container as a list in Primrose's library specifications. Such list is a model to convey the intended semantics, and does not proscribe anything about the implementation — the implementation is free to represent data in any chosen structure. For example, a set data type may be implemented with a binary search tree, but will still be specified with a list. These model lists are a simple abstraction, easy to analyse, with which all container operations can be specified.

Soundness of Library Specifications It is important that these library specifications are *sound*, in the sense that they fully capture all possible executions of the concrete implementation. Otherwise, the library specification would not be an accurate representation of the implementation's semantics, and Primrose could select a data type that does not satisfy the required semantic properties. More formally, a proof of functional correctness using these specifications would take the form of a data refinement [25], where each value of the concrete container type is related to our list model by an *abstraction function* α, and our specification on lists is shown to contain all possible behaviours of the concrete implementation using a *forward simulation*:

$$\alpha^{-1}; \mathsf{op}(C) \subseteq \mathsf{op}(A); \alpha^{-1}$$

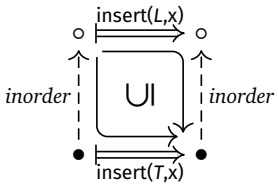
(where ; is forward composition of relations)



Here, op(C) denotes the concrete implementation of our operation op, represented as a relation from inputs to outputs. The abstract operation op(A) is the maximal relation satisfying the Hoare triple given in our library specification, and α is a suitable abstraction function that flattens a concrete container into a list.

If a forward simulation is shown for all operations, we can conclude that each possible execution with the concrete container has a corresponding execution with an abstract list, thus the specification accurately captures the implementation's semantics.

For instance, a binary search tree T can be abstracted to a sorted list L by an abstraction function *inorder* that does an in-order traversal. For each operation interacting with T, there exists a corresponding operation at the abstract level defined using L. Take the operation insert(T, x), which inserts an element x into a binary search tree T. We can abstract such an operation to insert(L, x) which inserts x at the right location in a sorted list. The relation between these two operations is shown by this diagram:



In this work, we specified four container implementations from Rust's standard library (Vec, LinkedList, HashSet, BTreeSet) and four custom container implementations (SortedVec, LazySortedVec, UniqueVec, LazyUniqueVec) by abstracting them into a list model. As we discuss in section 5.5, library specifications abstract over some implementation details, and, thus, Vec and LinkedList share the same specifications, as do the eager and lazy SortedVec and UniqueVec implementations. For each specification, we define a suitable abstraction function for forward simulation which, while not needed for selection, is used for property-based testing to justify the correctness of our specifications.

Completeness of Library Specifications Soundness of library specifications, which we ensure via property-based testing, is crucial to guarantee that Primrose does not select implementations which do not satisfy required semantic properties. Moreover, *completeness* of library specifications is also important. Without completeness, Primrose could possibly rule out perfectly valid implementations because it cannot prove that the required semantic properties are preserved for an incompletely-specified operation.

Our approach easily ensures completeness when each operation is specified by a *deterministic* model operation. Soundness states that every execution of the concrete implementation has a corresponding execution in the abstract operation, while determinism states that such correspondence is one-to-one, i.e., each abstract execution also has a corresponding concrete one. Thus, just as soundness states that each property established for an abstract operation applies also (via the inverse of the abstraction function α^{-1}) to a concrete implementation, completeness states that each property established for a concrete implementation applies (via the abstraction function α) to the abstract operation. With both completeness and soundness, we ensure that *all* valid implementations and *only* the valid implementations are selected by Primrose.

There are many other availiable approaches for modelling library specifications, for instance, the axiomatic approach used in algebraic specifications for abstract data types [36], specifying the behaviour of operations as a set of equational axioms that relate various operations. However, it is hard to ensure the completeness of algebraic specifications, as it is hard capture all behaviours of all operations by a set of equations.

5.2 The Library Specification of A LinkedList

Rust's LinkedList is a doubly-linked list. The abstraction function to convert it into a logic list is straightforward: Collect all nodes' values with previous and next pointers.

Firstly, we specify the insertion operation, LinkedList::insert, whose type signature is:

```
fn insert(&mut self, elt: T) {...}
```

Since variables in Rosette are immutable, in the corresponding abstract insertion operation, we alter the type to return a new list instead of altering the list in-place[1]:

```
abs-insert: List<T> -> T -> List<T>
```

We can then provide the specification of LinkedList::insert with respect to its corresponding abstract operation, the maximal relation satisfying the Hoare triple:

$$\{xs_0.\ \text{true}\}\ \text{abs-insert}\ \{xs_0\ x\ xs.\ xs = \text{model-insert}\ xs_0\ x\} \tag{3}$$

Here, xs_0 refers to the initial value of the container and xs to the resultant container, and x is the element we insert. The function model-insert is defined in Rosette on lists:

```
(define (model-insert xs x) (append xs (list x)))
```

The postcondition states that we expect applying the insertion operation to a container to produce the same result as the model-insert function. In library specifications, defining such *model operations* is a common technique to simplify writing postconditions.

Similarly, we also provide the specification for the operation LinkedList::contains:

```
fn contains(&self, x: &T) -> bool {...}
```

In our corresponding abstract operation, in addition to the boolean value indicating whether the given element x is present or not, the input container is also returned, as we would like to express the input container is not mutable, its value remains unchanged after this operation. Also, since the underlying value with type T is given by an immutable reference &T, in the abstract operation we treat the immutable reference &T as simply T. The signature of the abstract operation is shown below:

Listing 3 The signature of the abstract operation corresponding to LinkedList::contains

```
abs-contains: List<T> -> T -> (List<T>, bool)
```

The Hoare triple that serves as the specification of LinkedList::contains is:

$$\{xs_0.\ \text{true}\}\ \text{abs-contains}\ \{xs_0\ x\ xs\ r.\ (xs, r) = \text{model-contains}\ xs_0\ x\} \tag{4}$$

Note that in this specification, the model operation model-contains defined in listing 4 has the same type signature as the abstract operation shown in listing 3. It also returns a pair of values: the output list, which is always equal to the input list, and a boolean value indicating if the element is present in the list.

Listing 4 The model operation for checking an element's containment

```
(define (model-contains xs x)
  (cond [(list? (member x xs)) (cons xs #t)]
        [else (cons xs #f)]))
```

[1] Rosette is untyped, but this is morally the type signature.

Because model−contains returns the unchanged list, it specifies that the LinkedList::contains operation should not change the list.

The library specification of the list removal operation is slightly more complicated, we use T? to denote that a type may be null to express Rust's Option<T> type, which is the return type of LinkedList::remove. The type signature of LinkedList::remove is shown below:

```
fn remove(&mut self, x: T) −> Option<T> {...}
```

This operation removes the first occurrence of an element from the given linked list and returns it. If the linked list does not contain the element, None is returned and the list remains unchanged. The signature of the corresponding abstract operation is:

```
abs−remove: List<T> −> T −> (List<T>, T?)
```

The model removal operation has the same signature as the abstract operation. We return null in Rosette for the None case:

```
(define (model−remove xs x)
  (cond [(list?   (member x xs)) (cons (remove x xs) x)]
        [else  (cons xs  null)]))
```

Again, we return a pair of the resulting list and the element being removed. Then we provide the library specification of LinkedList::remove:

$$\{xs_0.\ \text{true}\}\ \text{abs-remove}\ \{xs_0\ x\ xs\ r.\ (xs, r) = \text{model-remove}\ xs_0\ x\} \tag{5}$$

To provide a complete specification of LinkedList, the library developer must ensure that each operation of the LinkedList is specified by a trait, and for each operation in each trait the LinkedList implements, specifications similar to the above are provided.

5.3 The Library Specification of A BTreeSet

For the LinkedList it is intuitive to use a logic list as a model, as they are both lists. However, even for non-linear structures such as trees, we can still use logic lists as a model. Rust's BTreeSet is a set implemented using a b-tree. All elements are unique and arranged in ascending order. Thus, our list model of the b-tree is simply a sorted list in ascending order, where uniqueness of elements is preserved. The abstraction function α that converts the BTreeSet to our list model is simply an in-order traversal.

Our first example is again the specification of the insertion operation with signature:

```
pub fn insert(&mut self, value: T) {...}
```

The signature of the abstract insert operation on our model lists is the same as for LinkedList::insert. The specification of abs−insert for BTreeSet, however, differs from that of LinkedList, as we must maintain ordering and uniqueness of elements:

$$\{xs_0.\ xs_0 = \text{dedup}\ (\text{sort}\ xs_0\ <)\}\ \text{abs-insert}\ \{xs_0\ x\ xs.\ xs = \text{model-insert}\ xs_0\ x\} \tag{6}$$

As before, x is the element to be inserted, and xs_0 and xs are lists modelling the container (via the in-order traversal function α) before and after the abs−insert operation respectively. We place an assertion $xs_0 = \text{dedup}\ (\text{sort}\ xs_0\ <)$ in the precondition requiring that the model xs_0 to be a sorted list of unique elements. While this precondition should always be satisfied by an in-order traversal of a valid b-tree, we do not want our abstraction to constrain the implementation's behaviour if the data invariants of the b-tree are violated — given a malformed b-tree, the implementation should be

free to return any result. Because the semantics of abs–insert are the maximal relation satisfying this specification, this abstract operation contains all possible behaviours of the concrete implementation if this precondition is violated. The model–insert here is simply an insertion operation defined on a sorted list of unique elements:

```
(define (model–insert xs x) (dedup (sort (append xs (list x)) <)))
```

We can also provide specifications for abstract operations that observe the ordering of elements in a BTreeSet, such as those operations from the IndexableT trait, since there is a one-to-one correspondence between each element's position in a BTreeSet and its position in the model list abstracted from the BTreeSet. For instance, we provide the specification of the operation BTreeSet::first, which is the operation obtaining the first (and also the minimal) element of a BTreeSet with signature:

```
fn first(&self) –> Option<&T> {...}
```

We again provide the signature of its corresponding abstract operation:

```
abs–first: List<T> –> (List<T>, T?)
```

Like LinkedList::contains in listing 3, this type includes a returned list, as Primrose does not consider the immutability of &self in the Rust type signature above. We again include the requirement that the container is unchanged in the specification:

$$\{xs_0.\ xs_0 = \mathsf{dedup}\ (\mathsf{sort}\ xs_0\ <)\}\ \mathsf{abs\text{-}first}\ \{xs_0\ xs\ x.\ (xs, x) = \mathsf{model\text{-}first}\ xs_0\} \tag{7}$$

Here, model–first is defined as a function that returns the first element of the list, is present, along with the list itself:

```
(define (model–first xs) (cond [(null? xs) (cons xs null)] [else (cons xs (first xs))]))
```

As before, our precondition includes the assumption that the model xs_0 abstracted from the BTreeSet contains unique elements that are sorted in ascending order.

5.4 The Library Specification of A HashSet

A tree implementation of a set maintains its elements in a fixed ascending order, and the ordering of our abstract list model simply reflects the ordering of the elements in the tree. However, some container implementations do not have a fixed ordering of elements. For instance, the HashSet in Rust is a set implementation using a hash algorithm which is randomly seeded. Despite the implementation storing elements in an unspecified order, we may still safely use a sorted, ascending list of unique elements as our abstract model of a HashSet: Our abstraction function α merely collects all elements from the HashSet into a list and then sorts them into ascending order.

Since the ordering of elements in our list is now different from the ordering of elements in the HashSet, the developer may specify properties relating to the ordering of elements, such as ascending, that are not satisfied by the implementation, but are trivially satisfied by the abstraction function. This would lead to HashSet being considered a valid choice for an ascending container. However, Primrose prevents this by the checking of syntactic properties. The HashSet type does not implement any trait with operations that allow the ordering of its elements to be observed. Therefore, in applications for which the ordering of elements is important, HashSet is never a valid choice. The selection process of valid implementations according to traits is discussed in Section 6.1.

For the operations defined on HashSet and BTreeSet, such as insert, remove and contains, the specifications of both implementations are identical—after all, the only observable difference between the implementations is performance—but the specification for HashSet lacks operations that observe the ordering of its elements, such as first or last.

5.5 Abstracting Over Implementation Details with Library Specifications

Since the basic container operations of both HashSet and BTreeSet have the same externally observable behaviour, we can use the same specifications for both implementations. There are many such cases where specifications can be re-used: For instance, we provide two implementations of an ascending vector: SortedVec and LazySortedVec. SortedVec maintains the ascending order of elements inside the vector on insertion (*eager*) and LazySortedVec instead sorts elements whenever the vector is accessed (*lazy*). Since both implementations share the same externally observable behaviour, we use the same model for both implementations: A list with elements sorted in ascending order. Also, their operations are specified with the same set of model operations. For the eager implementation, the abstraction function α simply collects all its elements into a list. For the lazy implementation, in addition to collecting all elements into a list, the abstraction function α also sorts elements into ascending order.

6 Selecting and Ranking Implementations

Before ranking container implementations by performance or other non-functional metrics, Primrose must first identify all implementations that comply with the property specifications provided by the application developer. This selection process comprises two steps: Firstly, Primrose selects all container implementations that satisfy the required *syntactic* properties. Secondly, from the implementations selected in the first step, it selects the ones whose library specifications match the required *semantic* properties. The first step is accomplished by simply gathering all types that implement the required Rust traits. For the second step, we check semantic properties against library specifications using the SMT solver in the backend of Rosette.

6.1 Selecting Container Implementations Satisfying Syntactic Properties

The first step of selecting valid implementations is to select concrete container implementations from the library that satisfy required syntactic properties in a property specification, which is straightforward. Primrose simply picks concrete container implementations that implement the traits required by the property specifications.

For instance, suppose that in a property specification, an application developer requires a container type implementing traits ContainerT and IndexableT, the elements of which are sorted in ascending order:

Listing 5 Property specification composing properties: ascending, ContainerT and IndexableT

```
property ascending { \c -> (for-all-consecutive-pairs c leq?) }
type AscendingIndexableCon<T> = {c <: (ContainerT, IndexableT) | (ascending c)}
```

In Rust's collections library, we have concrete container implementations Vec, LinkedList, BTreeSet and HashSet, where Vec, LinkedList and BTreeSet implement both required traits while HashSet does not implement the trait IndexableT. Clearly, HashSet does not satisfy all required syntactic properties. Therefore, HashSet is ruled out as a possible implementations for AscendingIndexableCon<T>. The implementation for AscendingIndexableCon<T> is then selected from the remaining Vec, LinkedList and BTreeSet types by checking if the library specifications satisfy the required semantic property, ascending.

6.2 Selecting Container Implementations Satisfying Semantic Properties

After gathering container implementations with required syntactic properties, Primrose then selects the ones that satisfy the required semantic properties from these candidates. As discussed in section 5, our library specifications abstract over the concrete container implementations, describing their externally observable semantics in a compact and tractable format. Primrose performs this selection process by encoding the property specifications as verification conditions against the candidates' library specifications in Rosette, to be discharged by an SMT solver in Rosette's backend.

In order to generate the required verification conditions, Primrose must first translate the required semantic properties, given in the specification language of Primrose, into definitions in Rosette that can be used by the solver. The container type Con<T> is resolved into the model type used in our library specifications, i.e., a logic list. For instance, the generated code according to the property ascending from listing 5 is:

```
1 (define ascending (lambda (c) (for-all-consecutive-pairs c leq?)))
```

With these Rosette definitions, Primrose can now generate verification conditions. As an example, suppose we consider BTreeSet as a candidate for the AscendingIndexableCon<T> type, and thus Primrose must generate verification conditions for the semantic property ascending. Primrose checks that the semantic property ascending is an invariant held across each operation defined for BTreeSet. For instance, for the BTreeSet's insertion operation, specified by (6) in section 5.3, it checks that the property ascending is preserved by any execution that satisfies its precondition and its postcondition:

$$\forall\ xs_0\ xs\ x.\ \frac{xs_0 = \mathsf{dedup}\ (\mathsf{sort}\ xs_0\ <) \qquad xs = \mathsf{model\text{-}insert}\ xs_0\ x}{\mathsf{ascending}\ xs_0 \implies \mathsf{ascending}\ xs}$$

$$(\text{where: } \exists\ xs_0.\ \mathsf{ascending}\ xs_0 \wedge xs_0 = \mathsf{dedup}\ (\mathsf{sort}\ xs_0\ <))$$

Figure 5 The rule for checking the operation BTreeSet::insert against ascending

Recall that xs_0 and xs are model lists abstracted from the BTreeSet, specifically, xs_0 is the model for the input BTreeSet, and xs is the model for the resulting BTreeSet of BTreeSet::insert. The model operation model-insert specifies the behaviour of BTreeSet::insert's corresponding abstract operation. Given the rule shown in figure 5, the solver attempts to find a counterexample, i.e., for all input models xs_0 that satisfy the semantic property ascending, the solver tries to find a resulting model of the operation that does not satisfy the property. If there is no such counterexample found, the solver will conclude that the operation BTreeSet::insert satisfies the property ascending.

This search for a counterexample is parameterised by a *model size*, which denotes the maximum size of the input list xs_0 considered by the solver. This parameter is configurable by the application developer using Primrose, and its impact on Primrose's selection time is discussed further in section 7.2.

This rule also contains a side condition stating that there should be no contradiction between the required semantic property and the precondition of the operation. This side condition is important for ensuring that the solver does not search for a counterexample in an empty search space. Without the side condition, this rule is unsound. For example, suppose that the application developer specifies a semantic property requiring the container to have at least two elements that are sorted in strictly descending order. Clearly, the precondition of BTreeSet::insert would contradict this formula, as it implies elements are sorted in strictly ascending order. However, if we were to check its library specification of its operations against this semantic property without taking care of this side condition, the solver will conclude that BTreeSet is a valid choice for this property, since it cannot find a counterexample that violates this rule: The contradiction in the assumptions makes it vacuously true. The absence of counterexamples is not because the property is preserved by the operation, but rather because the property could never hold in the first place. This contradiction provides the solver an empty search space to find a counterexample, and thus none is found. In short, our side condition requires that there exists at least one model that satisfies both the precondition of the operation and the required semantic property.

In general, the library specification of each operation takes the form:

$$\{\phi(xs_0, \vec{u})\}\ \mathsf{op}\ \{\psi(xs_0, xs, \vec{v})\} \tag{8}$$

where xs_0 is the (abstract list model of the) input container and xs is the result of the operation op. The sets of variables $\vec{u}$ and $\vec{v}$ denote any additional variables involved in the specification, such as additional inputs or outputs to the operation. The general form of the verification condition Primrose generates for the SMT solver, to check if an operation op satisfies a property P, is given in Figure 6.

$$\forall\ xs_0\ xs\ \vec{u}\ \vec{v}.\ \frac{\phi(xs_0, \vec{u}) \qquad \psi(xs, \vec{v})}{P(xs_0) \Rightarrow P(xs)} \quad (\text{where: } \exists\ xs_0\ \vec{u}.\ P(xs_0) \wedge \phi(xs_0, \vec{u}))$$

Figure 6 The rule for checking an operation against a property

For our BTreeSet example, Primrose checks these verification conditions for each operation of ContainerT and IndexableT—the traits implemented by BTreeSet. Because the property ascending is satisfied by all operations, Primrose concludes that the BTreeSet is a valid implementation choice for the required container type AscendingIndexableCon<T>.

The same checks are also run for the other two candidates that satisfy the required syntactic properties, specifically Vec and LinkedList, but they do not satisfy the required semantic property ascending. Therefore, Primrose concludes that amongst the three concrete implementations that satisfy the required syntactic properties, only the BTreeSet implementation satisfies the semantic property ascending, and hence it is the only valid implementation for the required container type AscendingIndexableCon<T>.

6.3 Handling Interactions Between Semantic and Syntactic Properties

Recall our property specification of a stack container StackCon<T> from section 4.3. The operations push and pop specified in the trait StackT (listing 1) are made available to the semantic property lifo (listing 2). In this section, we discuss how Primrose selects library implementations for such cases.

Firstly, Primrose generates the definition of semantic property lifo in Rosette, where the operations push and pop are now replaced with their model operations:

```
(define lifo  (lambda (c) (forall  (list  x)  (equal? (cdr  (model-pop (model-push c x))) x))))
```

Note that since model-pop returns a pair (List<T>, T?) and the definition of lifo only involves the element being popped, we use cdr to get the element, which is the second element in the returned pair. The specific model operations model-pop and model-push are supplied to this definition for each candidate type considered by Primrose. Recall that our library specifications state that these model operations exactly specify the intended behaviour of every library operation, which means that these model operations can be used here to express assertions about the interaction between operations such as push and pop. Such assertions will, by virtue of forward simulation, also apply to the concrete implementations of the data type.

To illustrate the selection process, suppose a library developer provides two implementations that implement push and pop. The first one is a last-in-first-out implementation, where the library specification of push and pop is:

$$\{xs_0.\ \mathsf{true}\}\ \mathsf{abs\text{-}push}_1\ \{xs_0\ x\ xs.\ xs = \mathsf{model\text{-}push}\ xs_0\ x\} \tag{9}$$

$$\{xs_0.\ \mathsf{true}\}\ \mathsf{abs\text{-}pop}_1\ \{xs_0\ xs\ x.\ (xs, x) = \mathsf{model\text{-}pop}\ xs_0\} \tag{10}$$

And the model operations are defined as:

```
(define (model-push xs x) (append xs (list x)))
(define (model-pop xs) (cond [(null? xs) (cons xs null)]
                             [else  (cons (take  xs  (- (length  xs)  1))  (last  xs))]))
```

With these two model operations, the solver can verify that this library specification satisfies the semantic property lifo.

By contrast, the second implementation is a first-in-first-out implementation. The library specification of push and pop appears similar:

$$\{xs_0.\ \mathsf{true}\}\ \mathsf{abs\text{-}push}_2\ \{xs_0\ x\ xs.\ xs = \mathsf{model\text{-}push}\ xs_0\ x\} \tag{11}$$

$$\{xs_0.\ \mathsf{true}\}\ \mathsf{abs\text{-}pop}_2\ \{xs_0\ xs\ x.\ (xs, x) = \mathsf{model\text{-}pop}\ xs_0\} \tag{12}$$

However, the model operations have different semantics:

```
(define (model-push xs x) (append (list x) xs))
(define (model-pop xs) (cond [(null? xs) (cons xs null)]
                             [else  (cons (take  xs  (- (length  xs)  1))  (last  xs))]))
```

With these two model operations, the solver concludes that this library specification does not satisfy the semantic property lifo, and Primrose does not consider this implementation as a valid choice for the container StackCon<T>.

6.4 Code Generation and Ranking Implementations by Performance

Once Primrose has selected the valid container implementations, it will generate a Rust program for each valid candidate by resolving the property specification into the selected container implementation. In figure 3 we show a simplified version of generated programs where property specifications are directly replaced with concrete implementations, in practice Primrose carefully generates Rust's trait objects to encapsulate the concrete implementation and exposing only those operations in Rust traits which are specified as syntactic properties.

As a proof-of-concept implementation, the current Primrose prototype ranks the generated Rust code for each valid implementation by executing all candidates and measuring their runtime on some test input data. We anticipate adopting more sophisticated ranking techniques, such as the once discussed in the related work, in the future. Our existing prototype of Primrose focuses on enabling application developers to specify their functional requirements, and automating the selection of valid container implementations.

7 Evaluation

For Primrose to be feasible for use as a programming tool, it must be practical to ensure that our library specifications are sound abstractions of our container implementations, and the selection process itself must not take a prohibitively long time. Our evaluation demonstrates feasibility in both of these aspects. All measurements are conducted on a MacBook Pro with 32 GB of RAM and a 2.4 GHz 8-Core Intel Core i9 processor.

7.1 Correctness of Libraries and their Specifications

To ensure the selected implementations are correct, we validate our Rust container library implementations against the library specifications using property-based testing [8]. We use the framework proptest [1] for encoding and performing the tests.

Firstly, we encode the model list with its operations in Rust. Specifically, we encode the model list from Rosette as an immutable `ConsList` [29] in Rust, along with all its operations. Then we implement the abstraction function α for each container implementation, and, like Chen et al. [6, 7], we encode the forward simulation obligation for the library specification of each operation as assertions in a test.

For each test, 100 test inputs are randomly generated. For our library with eight container implementations, in total 7200 inputs are tested in 7.315 seconds. We conclude that with the existing testing framework, we are able to validate the functional correctness of our container implementations w.r.t. our library specifications efficiently, ensuring that implementations selected by Primrose are correct.

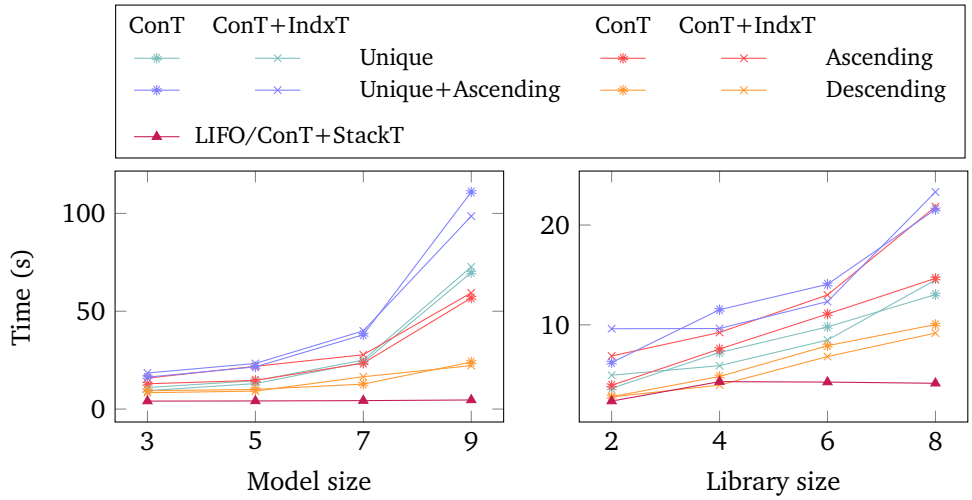


Figure 7 Primrose's efficiency of selecting implementations for different properties

7.2 Evaluation of Primrose's Selection Time

The efficiency of the SMT-based selection time is mainly determined by two factors: the *model size* and the *library size*, which together define the search space in which the solver attempts to find a counterexample. If a counterexample is found, Primrose will conclude that the library specification does not satisfy the required semantic property. The model size is the length of the input model list to the abstract operation. The library size is the number of container implementations from which we select.

Figure 7 shows the measurements of Primrose's selection time. The left side shows that the selection time, for a fixed library size of eight implementations, increases with the model size. The right side shows that for a fixed model size of five, the selection time also increases (albeit at a smaller rate) when the library size is increased.

The complexity of the property specifications and the number of satisfying implementations are also factors that affect the efficiency of the selection, since they determine how difficult it is for the solver to find a counterexample. For example, since the definition of lifo has constant complexity, the model size and library size do not affect its selection time as much as for properties with high polynomial complexity such as unique and ascending. None of our example containers satisfy the property descending. As SMT solvers are faster at finding a counterexample than exhaustively proving that no counterexample exists, the selection time for descending is faster than for ascending, despite both properties having the same algorithmic complexity.

The selection is always completed in less than 30 seconds with a model size of 3 and the full library of 8 implementations. Thus, we consider Primrose to be a practically feasible tool, especially as it does not need to be invoked on every compiler run.

An increase in model size raises selection time quickly, but in practice, we do not believe that a model size of more than five is required to admit counterexamples for most conceivable semantic properties that the application developer may write. Increasing the number of container implementations in the library results in a linear increase in selection time, allowing for Primrose to be used for medium-size libraries.

8 Related Work

Refinement types Refinement types, first introduced for ML [12], are types enriched with logical predicates from an SMT-decidable logic, allowing programmers to express rich logical constraints in the type system and automatically check them. Refinement types have recently been implemented in languages such as Haskell [33, 34] and F* [30], supporting very rich specifications suitable for verifying the correctness of programs. While the syntax of Liquid Haskell inspires our design of the syntax of property specifications, we use refinement types not for verification, but for data abstraction, allowing application developers to specify their semantic requirements for the selection process.

Abstract data types and formal methods Existing work in algebraic specifications [15, 36] provide a formal definition of abstract data types where the semantics of operations are specified with a set of equational axioms. By contrast, our library specifications are model-based. As mentioned in section 5.1, this allows us to easily ensure completeness of library specifications. There exist many formal modelling tools that facilitate model-based specification of abstract data types and software systems more generally, for example Z [28], VDM [19], and most recently Alloy [18]. While these tools allow application developers to formally analyse and explore the software design space, including formal reasoning about abstract data types, they work purely on the level of models and do not typically connect to actual code, as Primrose does.

Performance-oriented selection techniques Many techniques for design space exploration, particularly machine learning techniques, have been applied in compilers [13] to selected performance optimization techniques [4] using various characteristics as features that are then used to rank the performance of multiple implementations [27]. Many dynamic selection techniques have been developed for assisting the selection of performant containers [10, 20, 26, 37]. None of these techniques, however, use refinement types or semantic properties, and they have no general scheme to allow application developers to specify desired behaviour. These techniques purely focus on selecting between multiple, pre-known, valid container implementations. Such techniques could be incorporated into Primrose's ranking process, and are highly complementary with our work.

9 Conclusion

We have applied techniques from verification and formal methods in a new way, raising the level of abstraction by freeing developers from the burden of choosing concrete container implementations. Instead, application developers can specify their expected behaviour using semantic properties—a highly general abstraction technique. We provide a methodology to specify container libraries with library specifications, and describe our mechanism to check semantic properties against these specifications using SMT solvers. We implement Primrose for Rust and specify eight Rust container implementations. We show that Primrose is a practical tool that can be feasibly integrated into a programmer's workflow.

References

[1] AltSysRq. *Proptest: A Rust property testing framework*. Accessed Sep. 2022. 2022. URL: https://github.com/altsysrq/proptest.

[2] Martin Ankerl. *Hashmaps Benchmarks – Finding the Fastest; Memory Efficient Hashmap*. Accessed Sep. 2022. 2019. URL: https://martin.ankerl.com/2019/04/01/hashmap-benchmarks-01-overview/.

[3] Rudolf Bayer. "Symmetric Binary B-Trees: Data Structure and Maintenance Algorithms". In: *Acta Informatica* 1 (1972), pages 290–306. DOI: 10.1007/BF00289509. URL: https://doi.org/10.1007/BF00289509.

[4] John Cavazos, Grigori Fursin, Felix Agakov, Edwin Bonilla, Michael F. P. O'Boyle, and Olivier Temam. "Rapidly Selecting Good Compiler Optimizations Using Performance Counters". In: *Proceedings of the International Symposium on Code Generation and Optimization*. CGO '07. USA: IEEE Computer Society, 2007, pages 185–197. ISBN: 0769527647. DOI: 10.1109/CGO.2007.32. URL: https://doi.org/10.1109/CGO.2007.32.

[5] Paul Cechner. *vector vs map performance confusion*. Accessed Sep. 2022. 2014. URL: https://stackoverflow.com/questions/24542936/vector-vs-map-performance-confusion.

[6] Zilin Chen, Liam O'Connor, Gabriele Keller, Gerwin Klein, and Gernot Heiser. "The Cogent Case for Property-Based Testing". In: *Proceedings of the 9th Workshop on Programming Languages and Operating Systems*. PLOS'17. Shanghai, China: Association for Computing Machinery, 2017, pages 1–7. ISBN: 9781450351539. DOI: 10.1145/3144555.3144556.

[7] Zilin Chen, Christine Rizkallah, Liam O'Connor, Partha Susarla, Gerwin Klein, Gernot Heiser, and Gabriele Keller. "Property-based Testing: Climbing the Stairway to Verification". In: *Proceedings of the 15th ACM SIGPLAN International Conference on Software Language Engineering*. SLE 2022. To appear. New York, NY, USA: Association for Computing Machinery, 2022.

[8] Koen Claessen and John Hughes. "QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs". In: *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*. ICFP '00. New York, NY, USA: Association for Computing Machinery, 2000, pages 268–279. ISBN: 158113202-6. DOI: 10.1145/351240.351266.

[9] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, 3rd Edition*. MIT Press, 2009. ISBN: 978-0-262-03384-8. URL: http://mitpress.mit.edu/books/introduction-algorithms.

[10] Diego Costa and Artur Andrzejak. "CollectionSwitch: A Framework for Efficient and Dynamic Collection Selection". In: *Proceedings of the 2018 International Symposium on Code Generation and Optimization*. CGO 2018. Vienna, Austria: Association for Computing Machinery, 2018, pages 16–26. ISBN: 9781450356176. DOI: 10.1145/3168825. URL: https://doi.org/10.1145/3168825.

[11] Edouard. *Using C++ containers efficiently*. Accessed Sep. 2022. 2020. URL: https://blog.quasar.ai/using-c-containers-efficiently.

[12] Tim Freeman and Frank Pfenning. "Refinement Types for ML". In: *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*. PLDI '91. Toronto, Ontario, Canada: Association for Computing Machinery, 1991, pages 268–277. ISBN: 0897914287. DOI: 10.1145/113445.113468. URL: https://doi.org/10.1145/113445.113468.

[13] Grigori Fursin, Yuriy Kashnikov, Abdul Wahid Memon, Zbigniew Chamski, Olivier Temam, Mircea Namolaru, Elad Yom-Tov, Bilha Mendelson, Ayal Zaks, Eric Courtois, et al. "Milepost gcc: Machine learning enabled self-tuning compiler". In: *International journal of parallel programming* 39.3 (2011), pages 296–327.

[14] Jean-Yves Girard. "The System F of Variable Types, Fifteen Years Later". In: *Theor. Comput. Sci.* 45.2 (1986), pages 159–192. DOI: 10.1016/0304-3975(86)90044-7. URL: https://doi.org/10.1016/0304-3975(86)90044-7.

[15] John Guttag. "Abstract Data Types and the Development of Data Structures". In: *Proceedings of the 1976 Conference on Data: Abstraction, Definition and Structure*. Salt Lake City, Utah, USA: Association for Computing Machinery, 1976, page 72. ISBN: 9781450378987. DOI: 10.1145/800237.807124. URL: https://doi.org/10.1145/800237.807124.

[16] Roger Hindley. "The principal type-scheme of an object in combinatory logic". In: *Transactions of the American Mathematical Society* 146 (1969), pages 29–60.

[17] C. A. R. Hoare. "An Axiomatic Basis for Computer Programming". In: *Commun. ACM* 12.10 (Oct. 1969), pages 576–580. ISSN: 0001-0782. DOI: 10.1145/363235.363259. URL: https://doi.org/10.1145/363235.363259.

[18] Daniel Jackson. *Software Abstractions: Logic, Language and Analysis*. MIT Press, 2006. ISBN: 978-0262017152.

[19] Cliff B. Jones. *Systematic Software Development Using VDM (2nd Ed.)* USA: Prentice-Hall, Inc., 1990. ISBN: 0138807337.

[20] Changhee Jung, Silvius Rus, Brian P. Railing, Nathan Clark, and Santosh Pande. "Brainy: Effective Selection of Data Structures". In: *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '11. San Jose, California, USA: Association for Computing Machinery, 2011, pages 86–97. ISBN: 9781450306638. DOI: 10.1145/1993498.1993509. URL: https://doi.org/10.1145/1993498.1993509.

[21] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. "RustBelt: Securing the Foundations of the Rust Programming Language". In: *Proc. ACM Program. Lang.* 2.POPL (Dec. 2017). DOI: 10.1145/3158154. URL: https://doi.org/10.1145/3158154.

[22] Robin Milner. "A theory of type polymorphism in programming". In: *Journal of Computer and System Sciences* 17.3 (1978), pages 348–375. ISSN: 0022-0000. DOI: https://doi.org/10.1016/0022-0000(78)90014-4. URL: https://www.sciencedirect.com/science/article/pii/0022000078900144.

[23] Douglas A. H. Orr. *Finding unique items - hash vs sort?* Accessed Sep. 2022. 2019. URL: https://douglasorr.github.io/2019-09-hash-vs-sort/article.html.

[24] John C. Reynolds. "Towards a theory of type structure". In: *Programming Symposium, Proceedings Colloque sur la Programmation, Paris, France, April 9-11, 1974*. Edited by Bernard J. Robinet. Volume 19. Lecture Notes in Computer Science. Springer, 1974, pages 408–423. DOI: 10.1007/3-540-06859-7_148. URL: https://doi.org/10.1007/3-540-06859-7%5C_148.

[25] Willem-Paul de Roever and Kai Engelhardt. "Properties of Simulation". In: *Data Refinement: Model-Oriented Proof Methods and their Comparison*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1998, pages 73–89. DOI: 10.1017/CBO9780511663079.005.

[26] Ohad Shacham, Martin Vechev, and Eran Yahav. "Chameleon: Adaptive Selection of Collections". In: *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '09. Dublin, Ireland: Association for Computing Machinery, 2009, pages 408–418. ISBN: 9781605583921. DOI: 10.1145/1542476.1542522. URL: https://doi.org/10.1145/1542476.1542522.

[27] Norbert Siegmund, Sergiy S. Kolesnikov, Christian Kästner, Sven Apel, Don S. Batory, Marko Rosenmüller, and Gunter Saake. "Predicting performance via automated feature-interaction detection". In: *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*. Edited by Martin Glinz, Gail C. Murphy, and Mauro Pezzè. IEEE Computer Society, 2012, pages 167–177. DOI: 10.1109/ICSE.2012.6227196. URL: https://doi.org/10.1109/ICSE.2012.6227196.

[28] J. M. Spivey. *The Z Notation: A Reference Manual*. USA: Prentice-Hall, Inc., 1989. ISBN: 013983768X.

[29] Bodil Stokke. *im conslist: A Rust cons-list implementation*. Accessed Sep. 2022. 2022. URL: https://docs.rs/im/10.2.0/im/conslist/struct.ConsList.html.

[30] Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoue, and Santiago Zanella-Béguelin. "Dependent Types and Multi-Monadic Effects in F*". In: *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '16. St. Petersburg, FL, USA: Association for Computing Machinery, 2016, pages 256–270. ISBN: 9781450335492. DOI: 10.1145/2837614.2837655. URL: https://doi.org/10.1145/2837614.2837655.

[31] Emina Torlak and Rastislav Bodik. “A Lightweight Symbolic Virtual Machine for Solver-Aided Host Languages”. In: *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’14. Edinburgh, United Kingdom: Association for Computing Machinery, 2014, pages 530–541. ISBN: 9781450327848. DOI: 10.1145/2594291.2594340. URL: https://doi.org/10.1145/2594291.2594340.

[32] Emina Torlak and Rastislav Bodík. “Growing solver-aided languages with rosette”. In: *ACM Symposium on New Ideas in Programming and Reflections on Software, Onward! 2013, part of SPLASH ’13, Indianapolis, IN, USA, October 26-31, 2013*. Edited by Antony L. Hosking, Patrick Th. Eugster, and Robert Hirschfeld. ACM, 2013, pages 135–152. DOI: 10.1145/2509578.2509586. URL: https://doi.org/10.1145/2509578.2509586.

[33] Niki Vazou, Patrick Maxim Rondon, and Ranjit Jhala. “Abstract Refinement Types”. In: *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*. Edited by Matthias Felleisen and Philippa Gardner. Volume 7792. Lecture Notes in Computer Science. Springer, 2013, pages 209–228. DOI: 10.1007/978-3-642-37036-6_13. URL: https://doi.org/10.1007/978-3-642-37036-6%5C_13.

[34] Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton-Jones. “Refinement Types for Haskell”. In: *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*. ICFP ’14. Gothenburg, Sweden: Association for Computing Machinery, 2014, pages 269–282. ISBN: 9781450328739. DOI: 10.1145/2628136.2628161. URL: https://doi.org/10.1145/2628136.2628161.

[35] Baptiste Wicht. *C++ benchmark – std::vector VS std::list VS std::deque*. Accessed Sep. 2022. 2012. URL: https://baptiste-wicht.com/posts/2012/12/cpp-benchmark-vector-list-deque.html.

[36] Martin Wirsing. “Algebraic Specification”. In: *Formal Models and Semantics*. Edited by Jan Van Leeuwen. Handbook of Theoretical Computer Science. Amsterdam: Elsevier, 1990, pages 675–788. ISBN: 978-0-444-88074-1. DOI: https://doi.org/10.1016/B978-0-444-88074-1.50018-4. URL: https://www.sciencedirect.com/science/article/pii/B9780444880741500184.

[37] Guoqing Xu. “CoCo: Sound and Adaptive Replacement of Java Collections”. In: *ECOOP 2013 - Object-Oriented Programming - 27th European Conference, Montpellier, France, July 1-5, 2013. Proceedings*. Edited by Giuseppe Castagna. Volume 7920. Lecture Notes in Computer Science. Springer, 2013, pages 1–26. DOI: 10.1007/978-3-642-39038-8_1. URL: https://doi.org/10.1007/978-3-642-39038-8%5C_1.

About the authors

Xueying Qin is a PhD student at the University of Edinburgh. Contact her at xueying.qin@ed.ac.uk.

Liam O'Connor is a lecturer in Programming Languages for Trustworthy Systems at the University of Edinburgh. His research focuses on combining formal methods techniques with practical programming languages and tools. Contact him at l.oconnor@ed.ac.uk.

Michel Steuwer is a lecturer in Compilers and Programming Languages at the University of Edinburgh. His research on compiler design and domain-specific languages aims to drastically simplify the programming of complex parallel hardware devices while improving performance and efficiency. Contact him at michel.steuwer@ed.ac.uk.