Primrose: Selecting Container Data Types by their Properties

Xueying Qin

The University of Edinburgh

Table of Contents

Introduction

An Existing Problem of Container Types

- **Problem**: programmers have to choose a concrete implementation of a container type which is overly specific.
- **Drawbacks**: the chosen concrete implementation might not provide the desirable performance and portability of programs is limited.

An Existing Problem of Container Types - Example

When we write a program to gather a collection of unique elements and then to lookup a certain one:

What we have to write:

```
fn main() {
  let mut c = BTreeSet::<u32>::new();
  ...
}
/* OR */
fn main() {
  let mut c = HashSet::<u32>::new();
  ...
}
...
```

Figure 1: We have to choose a concrete implementation of a unique container when writing the program

What we want to write:

```
fn main() {
  let mut c = UniqueCon::<u32>::new();
  let data = raw_data();
  for val in data.iter() {
    c.insert(*val);
  }
  c.contains(&1024);
}
```

Figure 2: What we need is a container type representing a collection of unique elements in the program

Performance Benchmarks of Example Programs

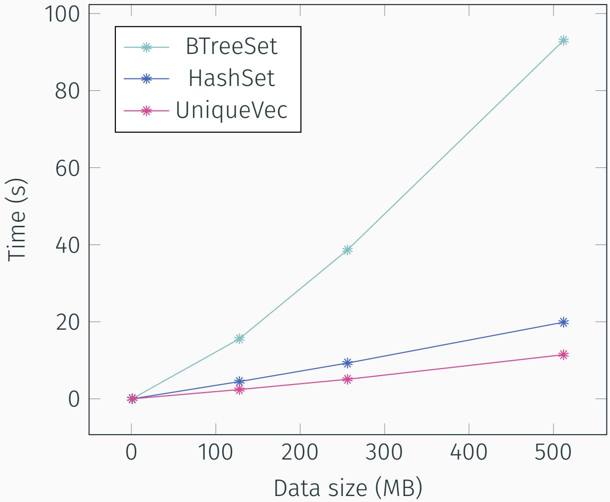

Figure 3: Different choices of concrete container implementations result in different performance

Observations

- What is actually needed in this program is a container of unique elements;
- However a programmer has to commit to a concrete container implementation when writing the program;
- For a program requiring many consecutive insertions, if the programmer chooses a `BTreetSet` or `HashSet` as the implementation of the container of unique elements, this program will not achieve the best performance.

Existing Approach Trying to Address This Problem - Abstract Data Types

- Abstract Data Types (ADTs): defined from the perspective of users, as a class of objects that is characterised by operations available for the objects.
- ADTs in practice: in some PLs, an ADT can be modelled as an interface allowing different underlying implementations.
- **Limitations**:
    - The default implementation choice is always the one optimised for the average case; it **does not provide the desirable performance** or memory efficiency for all usage cases.
    - ADTs are **insufficient for describing** the programmer's **expected behaviours** of the container in a program in a way that can be used by a compiler for selecting a best implementation (i.e., fastest, least memory consumption).

Our Proposed Design

- Application programmers specify expected behaviours of a container in a program instead of how the container is implemented as ***property specifications***;
  - A *syntactic property* specifies operations to interact with a container:
    - **We model it as a trait**
    - E.g., iterable - elements can be accessed by an iterator
  - A *semantic property* specifies the desired behaviours of existing operations:
    - **We model it as a logic predicate refines a container type**
    - E.g., unique - no duplicated elements in a container, it does not introduce any new operation
- Primrose, our pre-processing tool, selects all implementations from a container library where their ***library specifications*** match the desired property specifications;
- Primrose then chooses a best implementation (i.e., fastest, least memory consumption) for the program.

Overview of Primrose

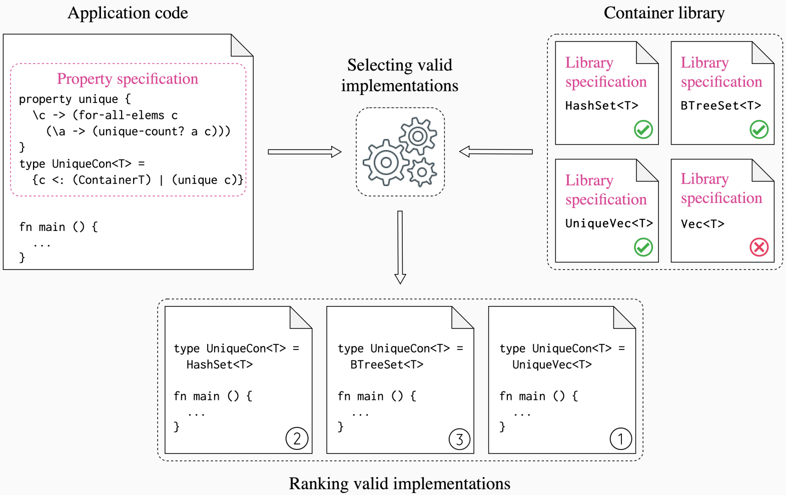

Figure 4: The workflow of the Primrose selection tool

Example Usage - User Program

To write a program, instead of picking a concrete container implementation, a programmer only need to specify what syntactic and semantic properties should be satisfied by the required container type as below:

```
property unique {
  \c -> (for_all_elems c (\a -> (is_unique_count a c)))
}
type UniqueCon<T> = {c impl (ContainerT) | (unique c)}

fn main () {
  let mut c = UniqueCon::<u32>::new();
  let data = raw_data();
  for val in data.iter() {
    c.insert(*val);
  }
  c.contains(&1024);
}
```

Figure 5: Example of a user program

Example Usage - Generated Programs

Our compiler will select all container implementations which satisfies the property specification from the library. For each selected container implementation, a program like below is generated, replacing the declared type `UniqueCon<T>` with the implementation:

```
type UniqueCon<T> = library::UniqueVec<T>;

fn main () {
  let mut c = UniqueCon::<u32>::new();
  let data = raw_data();
  for val in data.iter() {
    c.insert(*val);
  }
  c.contains(&1024);
}
```

Figure 6: One of generated programs with a possible library container implementation choice

Property Specification

Overview

In the above example, a programmer specifies the expected behaviours of the container in the program as *property specifications*:

```
property unique {
  \c -> (for_all_elems c (\a -> (is_unique_count a c)))
}
type UniqueCon<T> = {c impl (ContainerT) | (unique c)}
```

Figure 7: Example of a property specification

It says, the desired container type `UniqueCon<T>` need implement all operations in declared in the trait `ContainerT` and contain no duplicated element.

Semantic Properties as Type-Level Refinements

- Semantic property declaration:

```
property unique { \c -> (for_all_elems c (\a -> (is_unique_count a c))) }
```

  - Type of `unique`: $Con\langle T\rangle \rightarrow Bool$
  - $Con\langle T\rangle$ is a placeholder container type that will be resolved into concrete container types in the library;
  - `for_all_elems` is a predefined *combinator* for encoding semantic properties as measurements held by elements inside a container; `is_unique_count` is a predefined measurement function.
- Semantic property as refinement in the container type declaration:

```
type UniqueCon<T> = {c impl (ContainerT) | (unique c)}
```

  - The declared type $UniqueCon\langle T\rangle$ is a placeholder container type $Con\langle T\rangle$ refined by the property `unique`;
  - It will be resolved into concrete container types that satisfy the property `unique`.

More Semantic Properties and Their Compositions

- Other than `unique`, we can also encode other semantic properties with different combinators:
  - Elements inside a container are sorted in **ascending** order:

    ```
    property ascending { \c -> (for_all_consecutive_pairs c leq) }
    ```

  - Elements inside a container are sorted in **descending** order:

    ```
    property descending { \c -> (for_all_consecutive_pairs c geq) }
    ```

- We can also compose semantic properties in a container type declaration:

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

Syntactic Properties as Bounds of Container Types

- Syntactic properties are modelled as *traits* in Rust, specifying operations to interact with the container.
- Syntactic property as bound in the container type declaration:

```
type UniqueCon<T> = {c impl (ContainerT) | (unique c)}
```

  - The trait `ContainerT` is a syntactic property;
  - The declared type *UniqueCon*$\langle T \rangle$ is a placeholder container type *Con*$\langle T \rangle$ providing operations specified by `ContainerT`;
  - It will be resolved into concrete container types that implement `ContainerT`.

How Syntactic Properties Interact with Semantic Properties

Some semantic properties are defined on operations specified by syntactic properties. For example:

- on top of the basic `ContainerT` trait, we define a `StackT` trait providing `push` and `pop` operations:

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

- then the semantic property last-in-first-out (LIFO) can be defined as:

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

- to specify a container implements `StackT` with the property LIFO:

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

Library Specification

Why We Need Library Specifications

- It is hard to select container implementations from the library by checking if concrete implementations satisfy property specifications;
- We need library specifications which **abstract over implementations**, allowing us to:
  - **verify** if a **container implementation** satisfies its library specification;
  - **select container implementations** by checking if their library specifications satisfy property specifications.

The Design of Library Specifications

Library specifications of concrete container implementations are developed based on Hoare logic. For each concrete container implementation, we provide a set of *Hoare triples*, one for each operation:

$$\{\phi\}\ \mathsf{op}\ \{\psi\}$$

- If the *precondition* ϕ holds and the operation op is executed, then the *postcondition* ψ will hold.
- We define the precondition and postcondition of each operation in terms of an *abstract list model*.

The Soundness of Library Specifications (1)

- It is important that library specifications are *sound*, i.e., capture all possible executions of the concrete implementation;
- The proof of the functional correctness takes the form of a data refinement, where
  - each value of the concrete container type is related to our list model by an *abstraction function* α,
  - our specification on lists is shown to contain all possible behaviours of the concrete implementation using a *forward simulation*:

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

(where ; is forward composition of relations)

$$\begin{tikzcd} \circ \arrow[r, Rightarrow, "\mathrm{op}(A)"] & \circ \\ \bullet \arrow[u, dashed, "\alpha"] \arrow[r, Rightarrow, "\mathrm{op}(C)"'] \arrow[ur, phantom, "\subseteq"] & \bullet \arrow[u, dashed, "\alpha"'] \end{tikzcd}$$

Figure 8: Forward simulation

The Soundness of Library Specifications (2)

If such a forward simulation is shown for all of our container operations, each possible execution involving the concrete container has a corresponding execution involving an abstract list, and thus the specification accurately captures the semantics of our implementation.

For example, for the insertion operation of the `BTreeSet` from Rust collection library, which is a set implementation based on a B-Tree, with signature:

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

Figure 9: Signature of `BTreeSet::insert`

Example - The Library Specification of `BTreeSet`'s Insertion Operation (1)

- The model logic list is a list of unique elements which are sorted in ascending order;
- We abstract this `BTreeSet` in to a the list model by applying an abstraction function *inorder* that does a in-order traversal;
- The corresponding abstract operation defined on the list has (moral) type signature:

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

Figure 10: Signature of `BTreeSet::insert`'s corresponding abstract operation

Example - The Library Specification of `BTreeSet`'s Insertion Operation (2)

We write its library specification:

$$\{xs_0.\ xs_0 = \texttt{remove-duplicates}\ (\texttt{sort}\ xs_0\ \texttt{<})\}\ \texttt{abs-insert}\ \{xs_0\ x\ xs.\ xs = \texttt{model-insert}\ xs_0\ x\}$$

Figure 11: Specification of `BTreeSet::insert`

Where the `list-insert` is the insertion function defined on a logic list which is unqiue and sorted in ascending order:

```
(define (model-insert xs x) (remove-duplicates (sort (append xs (list x)) <)))
```

Example - Verifying An Operation's Implementation Satisfying Its Specification

To check if the `BTreeSet`'s insertion operation satisfies the specification:

1. **assume the precondition holds**, `y0` is a unique logic list that is sorted in ascending order
2. **check if the postcondition holds**, i.e., if `ys` equal to the result of the corresponding insertion operation defined on a logic list

Since we can verify each operation implementation of a container w.r.t its library specification, we represent each container by its library specification in the implementation selection process.

Selecting Valid Implementations

Overview

Two steps of selecting implementations from the library:

1. Selecting container types in the library which implement specified syntactic properties (traits);
2. Selecting container types of which the library specification match the semantic property specification.

The first step can be simply handled by checking whether a container implementation implements the traits specified in a property specification, we mainly discuss the second step here.

The Process of Selecting Valid Implementations (1)

For each semantic property in a property specification and each library specification:

1. check there is no contradiction between the semantic property and each precondition in the library specification;
2. assume the semantic property holds before each operation;
3. check if the resulting logic list of each operation still satisfies the semantic property.

This process is implemented using a SMT solver (Z3), more specifically, we interact with the solver using Rosette, which is a solver-aided programming language.

The Process of Selecting Valid Implementations (2)

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

$$\{\phi(xs_0, \vec{u})\} \text{ op } \{\psi(xs_0, xs, \vec{v})\}$$

- The general form of the verification condition Primrose generates for the SMT solver, to check if an operation op satisfies a property P:

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

Figure 12: The rule for checking an operation against a property

Example - Checking If `BTreeSet` is A Valid Unique Container Implementation

Recall the introduced property specification of a unique container:

```
property unique {
  \c -> (for_all_elems c (\a -> (is_unique_count a c)))
}
type UniqueCon<T> = {c impl (ContainerT) | (unique c)}
```

Given the `BTreeSet` implements the trait `ContainerT`, we need to check if for each operation of `ContainerT` implemented by the `BTreeSet`, the property `unique` holds.

Example - Checking the Library Specification of `insert`

Recall the introduced library specification of the `BTreeSet`'s insertion operation:

$$\{xs_0.\ xs_0 = \texttt{remove-duplicates}\ (\texttt{sort}\ xs_0\ \texttt{<})\}\ \texttt{abs-insert}\ \{xs_0\ x\ xs.\ xs = \texttt{model-insert}\ xs_0\ x\}$$

We check this specification against the property `unique` in a SMT solver according to:

$$\forall\ xs_0\ xs\ x.\ \frac{xs_0 = \texttt{remove-duplicates}\ (\texttt{sort}\ xs_0\ \texttt{<}) \qquad xs = \texttt{model-insert}\ xs_0\ x}{\texttt{unique}\ xs_0 \Rightarrow \texttt{unique}\ xs}$$

$$(\text{where: } \exists\ xs_0.\ \texttt{unique}\ xs_0 \land xs_0 = \texttt{remove-duplicates}\ (\texttt{sort}\ xs_0\ \texttt{<}))$$

Example - Checking If `BTreeSet` is A Valid Unique Container Implementation

- For each operation of `ContainerT` implemented by the `BTreeSet`, a similar checking process is performed.
- If the property `unique` holds through all operations, the `BTreeSet` is a valid implementation choice for the required unique container `UniqueCon`.

Future Work

Future Work

- Design a better ranking technique, allowing the best container implementation (in terms of run-time performance, memory footprints etc.) for a program to be selected;
- Formally verify the container implementations in a library satisfy their library specifications.

Thank you \\(^-^)/

Xueying Qin [xueying.qin@ed.ac.uk]