Studies Concerning the Meaning of Computer Programs

Xueying Qin (秦雪莹)

The University of Edinburgh

March 20, 2024

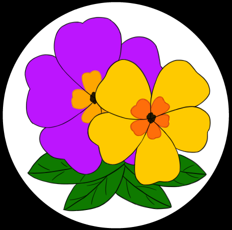
Introduction

In my three-year short research journey, I have studies three projects concerning the **meaning** of computer programs, specifically:

- How we **model** and **understand** programs;
- How we effectively **communicate** what we want computers to do in terms of programs with computers;
- How we **reason about** the behaviours of programs.

Three Conceptual Questions and Three Projects

- How to design a better abstraction mechanism that allows programmers to effectively express *what* they want a computer to do via some declarative yet accurate *specifications* instead of *how* a computer should accomplish a task via some concrete *implementations*?
  - Primrose: Selecting Container Data Types by Their Properties
- How do we characterise the relationship between the *syntax* and *semantics* of programming languages?
  - Shoggoth: A Formal Foundation for Strategic Rewriting
- How do we intuitively understand *distributed programs* using the same conceptual model as *monolithic programs*?
  - Oxidising Remote Procedural Calls



Primrose: Selecting Container Data Types by Their Properties

Xueying Qin[1]
Liam O'Connor[1], Michel Steuwer[1 2]
[1] The University of Edinburgh [2] Technische Universität Berlin

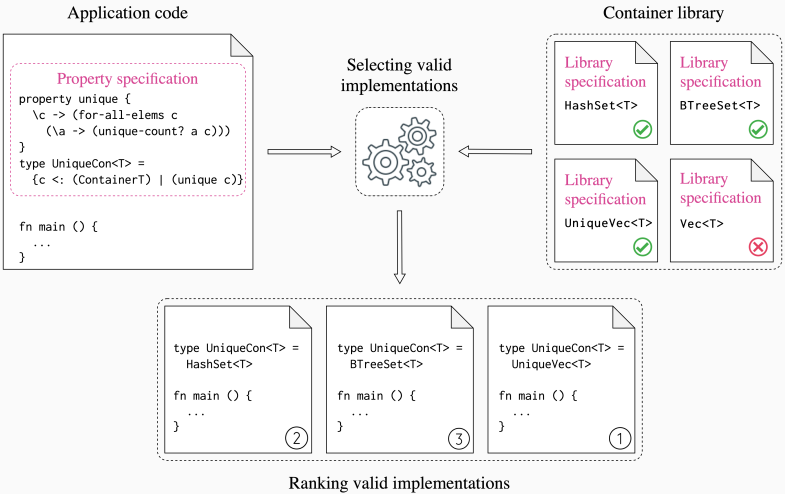
Motivation: An Existing Problem of Container Types

- **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.

The Design of Primrose

- 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 refining 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



Application code

Property specification

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

```
fn main () {
  ...
}
```

Selecting valid implementations

Container library

Library specification `HashSet<T>` ✓

Library specification `BTreeSet<T>` ✓

Library specification `UniqueVec<T>` ✓

Library specification `Vec<T>` ✗

```
type UniqueCon<T> =
  HashSet<T>

fn main () {
  ...
}
```

②

```
type UniqueCon<T> =
  BTreeSet<T>

fn main () {
  ...
}
```

③

```
type UniqueCon<T> =
  UniqueVec<T>

fn main () {
  ...
}
```

①

Ranking valid implementations

Figure 1: The workflow of the Primrose selection tool

Property Specifications and Library Specifications

Property specification

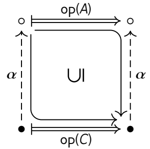
```
property unique {  \c  -> (for_all_elems   c (\a  -> (is_unique_count  a c)))}
type UniqueCon<T> = {c impl (ContainerT) |  (unique  c)}
```

- Semantic property `unique` as refinement
- Syntactic property `ContainerT` as trait

Library specification

- Taking the form of a **Hoare triple** for each operation, defined w.r.t. a list model: $\{\phi\}\ \mathsf{op}(A)\ \{\psi\}$
- The (absract) list model and the concrete implementation forms a **forward simulation**:

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



- Example: library specification for `BTreeSet` insertion:

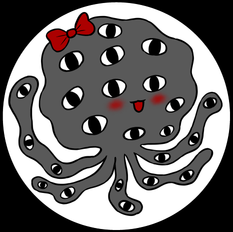
$$\{xs_0.\ xs_0 = \texttt{remove-duplicates}\ (\texttt{sort}\ xs_0\ <)\}\ \texttt{abs-insert}\ \{xs_0\ x\ xs.\ xs = \texttt{model-insert}\ xs_0\ x\}$$

Selecting Valid Implementations

- Selecting container types in the library which implement specified syntactic properties (traits);
- Selecting container types of which the library specification match the semantic property specification, using the **Z3 SMT solver**:
  - check there is no contradiction between the semantic property and each precondition in the library specification;
  - assume the semantic property holds before each operation;
  - check if the resulting model list of each operation still satisfies the semantic property.

Summary of Our Contributions

- We show a **new application of refinement types** 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**, 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, validate container implementations against specifications and evaluate the efficiency of the selection process.
- Paper available (‹Programming› 2023): https://doi.org/10.22152/programming-journal.org/2023/7/11



Shoggoth:

A Formal Foundation for Strategic Rewriting

Xueying Qin[1]
Liam O'Connor[1], Rob van Glabbeek[1 3],
Peter Höfner[2], Ohad Kammar[1], Michel Steuwer[1 4]

[1] The University of Edinburgh
[2] Australian National University
[3] UNSW
[4] Technische Universität Berlin

Motivation: Importance of Strategic Rewriting Languages

- Strategic rewriting languages provide programmers with **combinators** and **generic traversals** that allow them to:
  - control the application of rewrite rules
  - reuse rewrite rules
- Many application areas: program optimisation (ELEVATE [Hagedorn et al., 2020]), writing interpreter/compiler for DSLs (Spoofax/Stratego [Visser, 2001]) etc.
- However, there is a lack of formal treatment.

Overview of Strategic Rewriting Languages

Atomic strategy

An atomic strategy is a *rewrite rule*:

$add_{com} : a + b \rightsquigarrow b + a \quad add_{id} : 0 + a \rightsquigarrow a$

$mult_{com} : a * b \rightsquigarrow b * a$

$mapFusion : map\ f\ (map\ g\ xs) \rightsquigarrow map\ (f \circ g)\ xs$

Composed strategy

$add_{com}\ ;\ add_{id} \qquad add_{com} \mathrel{<\!+} mult_{com}$

$repeat(mapFusion)$

Strategy combinators

Strategy combinators compose strategies together and controls the application of atomic strategies:

$s_1\ ;\ s_2$ sequential composition, apply s_1 then s_2

$s_1 \mathrel{<\!+} s_2$ left choice, if fail to apply s_1 then s_2

$repeat(s)$ keep applying s until inapplicable

Introduction to System S and Expressions to be Rewritten

System S

System S [Visser and el Abidine Benaissa, 1998], the core calculus of strategic rewriting languages like ELEVATE [Hagedorn et al., 2020], Stratego [Visser, 2001] and Strafunski [Kaiser and Lämmel, 2009], contains **atomic strategies** (rewrite rules), **strategy combinators** which compose strategies and **traversals** that traverse the expression AST.

Expression

The expressions being rewritten by strategies are in the form of:

$$\mathit{Expressions}(\mathbb{E}) \quad e := \quad \mathit{Leaf} \mid \overset{n}{\widehat{e \quad e}}$$

Syntax of Strategies

Strategy

Strategy$(\mathbb{S})$ $\quad s \;:=\;$ SKIP (Always succeeds) | ABORT (Always results in error)

| *atomic* (Atomic strategy)

| $s_1 \;;\; s_2$ (Sequential composition)

| $s_1 <\!+\; s_2$ (Left choice)

| $s_1 <\!+\!> t_2$ (Nondeterministic choice)

| *one(s)* (Apply *s* to one child, nondeterministic)

| *some(s)* (Apply *s* to as many children as possible, nondeterministic)

| *all(s)* (Apply *s* to all children, nondeterministic)

| X (Variable)

| $\mu X.s$ (Fixed-point operator)

Importance of A Formal Understanding of Strategic Rewriting Languages

Strategies can go wrong

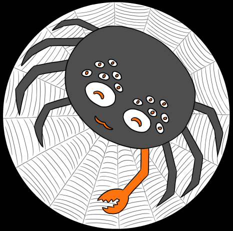
- **Result in error** - an atomic strategy is not defined for certain expressions or strategies are not well composed, for example: $add_{com} \,;\, mult_{com}$
- **Do not terminate** - for example: $repeat(\mathrm{SKIP})$
- **Do not rewrite an expression into desired form**

Existing formal work is not sufficient

- Big-step operational semantics of System S without modelling divergence [Visser and el Abidine Benaissa, 1998].
- Weakest preconditional calculus for System S using computational tree logic (CTL) [Kieburtz, 2001]. It has following issues:
  - not expressive enough to reason about nondeterminism in traversals
  - problematic fixed-point operator construction
  - soundness of the calculus is not proven

Summary of Our Contributions

- We provide the formal semantics of System S, including both **denotational** and **operational** models.
    - Featuring **nondeterminism**, **errors**, and **divergence**.
    - Proving these two semantics models are **equivalent** via computational soundness and adequacy,
- We provide the **weakest precondition calculus** for the strategic rewriting language.
    - Proving its soundness w.r.t. the denotational semantics.
- We demonstrate how to use the weakest precondition calculus to **prove properties** of strategic rewriting.
- All formalised semantics and calculus as well as proofs are mechanised in Isabelle/HOL.
- Paper available (POPL 2024): https://doi.org/10.1145/3633211



Oxidising Remote Procedure Calls

Xueying Qin[1] Dan Ghica[2]

[1] The University of Edinburgh [2] Huawei Central Software Institute

Motivation: Challenges in Existing Remote Procedure Call (RPC) Design

- In distributed computing, a remote procedure call (RPC) allows a method invocation to be executed on another computer on a shared network. Such a remote method invocation has the same coding as a local invocation, without the programmer explicitly coding the details for the remote interaction.
- It is hard to support **location transparency**, i.e., in most existing frameworks (e.g., Java RMI), remote invocations do not have **the same semantics** as local invocations.
- **Memory management** is hard in a distributed setting, for example, distributed garbage collection is complicated.

A New Remote Procedure Call Design: Universal Method Invocation

We design a **universal method invocation (UMI)** library in Rust, where our remote invocations have **the same semantics** as local invocations.

Why UMI?

- It allows applications to be **migrated from a monolithic design** to a **distributed architecture** without massive changes to source code or the needs of high-level expertise in microservices.
- It gives support for **advanced optimisations** such as profile guided optimisation, which are not viable if changing the deployment requires extensive re-coding.

Why Rust?

- Rust is high-level system programming language which **guarantees memory safety** and **prevents data races** by its *ownership* rules for memory management and *borrow checker* for tracking object lifetime of all references in a program during compilation.
- Since Rust has semantics that guarantees memory safety, we can extend such guarantees to the distributed computing setting, allowing our UMI framework to provide safe remote method invocations.

Example: Deploying A Monolithic Program to Multiple Nodes

Key Idea: **location transparency** — a method invocation on a remote object preserves the semantics of the method invocation on a local object.

```
#[gen_remote]
struct A { arg: u32 }

#[gen_remote]
impl A {
  new(arg: u32) -> A {
    A {arg: arg}
  }
  fun_owned(&self, a: A) {...}
  fun_imm(&self, &a: A) {...}
  fun_mut(&self, &mut a: A) {...}
  .....
}
```

```
fn main() {
  let a_remote = remote!(addr, A::new(10));

  let a_local1 = A::new(1);
  let a_local2 = A::new(2);
  let mut a_local3 = A::new(3);

  a_local1.fun_imm(&a_local2);
  a_remote.fun_imm(&a_local2);
}
```

Argument Passing Semantics

Variable Binding Type	**Passing Semantics**
owned	pass by copy/ move
immutable reference	pass by reference
mutable reference	pass by mutable reference

Examples:

```
fn main() {
  ...
  a_remote.fun_owned(a_local1);    // pass by copy/move
  a_remote.fun_imm(&a_local2);     // pass by reference
  a_remote.fun_mut(&mut a_local3); // pass by mutable reference
}
```

Distributed Resource Management

On a UMI server, we use a table with the same lifetime of the server to identify and manage local resources involved in remote computations.

- Rust **borrow checker** is generalised to work over the multiple nodes and handle remote allocation, access, modification and deallocation.
- If **a variable is created locally**, it will be put into the table once it is passed into a remote computation. The entry will not be removed (this table will ensure it is not deallocated) until the remote computation finishes.
- If **a variable is created via a remote call**, it will be put into table on creation. It will be deallocated when its remote owner decides to drop it.

Summary of Our Contributions

- We provide a usable **Rust implementation of the UMI framework**.
- We formalise the **structural operational semantics** for a core calculus of monolithic and distributed Rust programs.
- We prove a **location transparency theorem**: With the UMI framework, when a monolithic program is deployed to multiple nodes, its semantics is preserved.
- The research results are currently under preparation to be submitted to a conference.

Finale

Three Years, Three Conceptual Questions and Three Projects

- How to design a better abstraction mechanism that allows programmers to effectively express *what* they want a computer to do via some declarative yet accurate *specifications* instead of *how* a computer should accomplish a task via some concrete *implementations*?
  - Primrose: Selecting Container Data Types by Their Properties
- How do we characterise the relationship between the *syntax* and *semantics* of programming languages?
  - Shoggoth: A Formal Foundation for Strategic Rewriting
- How do we intuitively understand *distributed programs* using the same conceptual model as *monolithic programs*?
  - Oxidising Remote Procedural Calls

Thank you!

Xueying Qin [xueying.qin@ed.ac.uk]
[https://xyunknown.github.io]

Hagedorn, B., Lenfers, J., Kœhler, T., Qin, X., Gorlatch, S., and Steuwer, M. (2020).
Achieving high-performance the functional way: A functional pearl on expressing high-performance optimizations as rewrite strategies.
Proc. ACM Program. Lang., 4(ICFP).

Kaiser, M. and Lämmel, R. (2009).
An isabelle/hol-based model of stratego-like traversal strategies.
In *Proceedings of the 11th ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*, PPDP '09, page 93–104, New York, NY, USA. Association for Computing Machinery.

Kieburtz, R. B. (2001).
A logic for rewriting strategies.
Electronic Notes in Theoretical Computer Science, 58(2):138–154.
STRATEGIES 2001, 4th International Workshop on Strategies in Automated Deduction - Selected Papers (in connection with IJCAR 2001).

Visser, E. (2001).
Stratego: A language for program transformation based on rewriting strategies system description of stratego 0.5.
In Middeldorp, A., editor, *Rewriting Techniques and Applications*, pages 357–361, Berlin, Heidelberg. Springer Berlin Heidelberg.

Visser, E. and el Abidine Benaissa, Z. (1998).
A core language for rewriting.
Electronic Notes in Theoretical Computer Science, 15:422–441.
International Workshop on Rewriting Logic and its Applications.