Empirical Soundness of Gradual Verification

Jan-Paul V. Ramos-Dávila[1] [jvr34@cornell.edu]
Advised by: Jenna Wise[2], Ian McCormack[2], Dr. Joshua Sunshine[2], Dr. Jonathan Aldrich[2]
Cornell University[1]
Institute for Software Research @ Carnegie Mellon University[2]

Background

λ *Static verification* techniques do not provide good support for incrementality.

λ *Dynamic verification* approaches cannot provide static guarantees.

λ *Gradual verification* bridges this gap, supporting incrementality by allowing the user to specify a given program as much as they want, with a formal guarantee of verifiability. The *gradual guarantee* states that verifiability and reducibility are **monotone with respect to precision**.

```
//Gradually Verified Example
void withdraw(Account* account)
//@requires acc(account->balance) &&
account->balance >= 5
//@ensures acc(account->balance) &&
account->balance >= 0
{
  account->balance -= 5;
}

// ? allows the verifier to assume anything
necessary to satisfy the withdraw precon
void withdraw(Account* account)
//@requires ? && acc(account->balance);
//@ensures ? && acc(account->balance) &&
account->balance >= 0
{
  if(account->balance <= 100)
    withdraw(account);
}
```

Motivation

λ *A simple issue*: Our verifier stores this information `{x.f->t} , {t == 2}` where **`t`** is an abstract variable.

`? && x.f == 2 { }, { t == 2 }` : Loses relationship between location **`x.f`** and **`t`**, hence there is no mapping of **`x.f->t`**.

`assert x.f == 2` : Creates a fresh abstract variable **`t'`** for **`x.f`** and so **`t' == 2`** is asserted against **`{t == 2 }`**, so tool ends up creating a run-time check for **`t' == 2`**, which is information the tool knows but isn't aware that it knows.

λ *A simple fix*: The example in *Background* uses *accessibility predicates* to denote the ownership of heap locations. The system currently verifies accessibility predicates at runtime by tracking and updating a set of owned heap locations. Therefore, **our fix adds this mapping `x.f -> t` onto the heap**.

```
struct Test
{
  int f;
  int g;
};


void issue(struct Test *x)
//@requires ? && x->f == 2;
//@ensures ? && x->f == 2;
{
  x->f = 2;
  x->g = 1;
}
```

Runtime Checks *before* fix

```
[info] Runtime checks required for
GenericNode(x.Test$g):
[info]   if true: acc(x.Test$g, write)
[info] Runtime checks required for
GenericNode(x.Test$f):
[info]   if true: acc(x.Test$f, write)
[info] Runtime checks required for
GenericNode(acc(x.Test$f, write)):
[info]   if true: !(x == null)
```

Runtime Checks *after* fix

```
[info] Runtime checks required for
GenericNode(x.Test$g):
[info]   if true: acc(x.Test$g, write)
```

Evaluating Soundness

λ Our current version of the verifier does not evaluate the *soundness* of a given solution.

> *Soundness: the verifier's ability to catch all bugs/violations of a given specification.*

λ **Lightweight Formal Methods via Property Based Testing** is the inspiration behind this framework. It allows us to test the properties of examples and compare the results of the PBT to the results of our tool.

Framework

Reference model language
Gradual specification language in Gradual C0

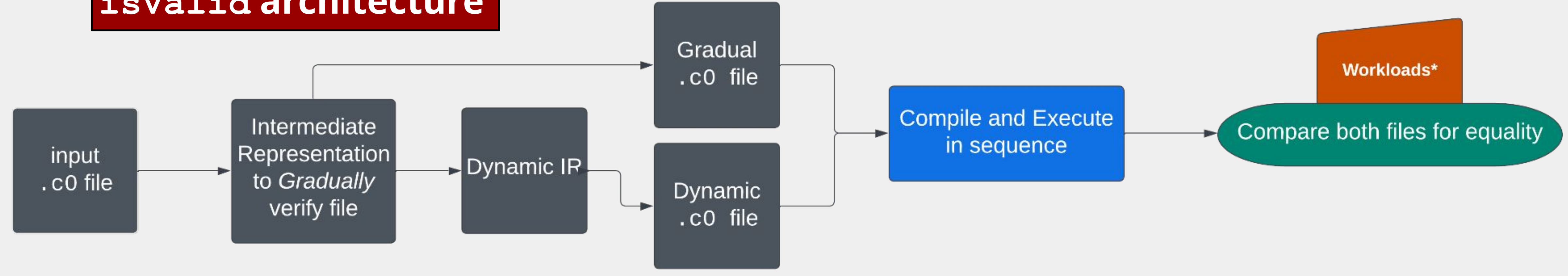
Checker
`isValid` method

Input Generator
QuickCheck type generator for specific examples. These examples are **not** supposed to verify correctly, they need to either have incorrect specifications or incorrect implementations.

Architecture

λ Currently, we have set up system benchmarking as a GitHub Actions script that automatically populates a database with the current instance of the benchmark when the `gvc0` repository receives a pull request. This is the same idea behind the `isValid` function. In this pipeline, however, `isValid` is just an intermediate step for the overall testing framework, **we still have to verify verifier soundness generally!**

`isValid` architecture



Future Work

λ For the **Input Generator**, we need a strong set of benchmark examples. We already have examples that are correct—verify correctly, correct specification, and correct implementation. We now want to *non-trivially break* those examples by coming up with incorrect specifications and incorrect implementations. This is a **QuickCheck** tool.

λ Modify the `main` function in the test file to generate input to the functions that get at the unsoundness. Currently, `main` randomly generates (Lists/BSTs) for execution.

λ Evaluate the empirical tool's *completeness/soundness* with a formal soundness evaluation of the gradual verifier.