Gradual Verification
An End-to-End Implementation

Hemant Gouni
University of Minnesota, Twin Cities

Conrad Zimmerman
Brown University

Motivation

- Static verification requires full specification of program behavior.
- Gradual verification allows writing specification incrementally, reducing the specification burden required to verify important components.
- We present the first implementation of gradual verification in an executable language.

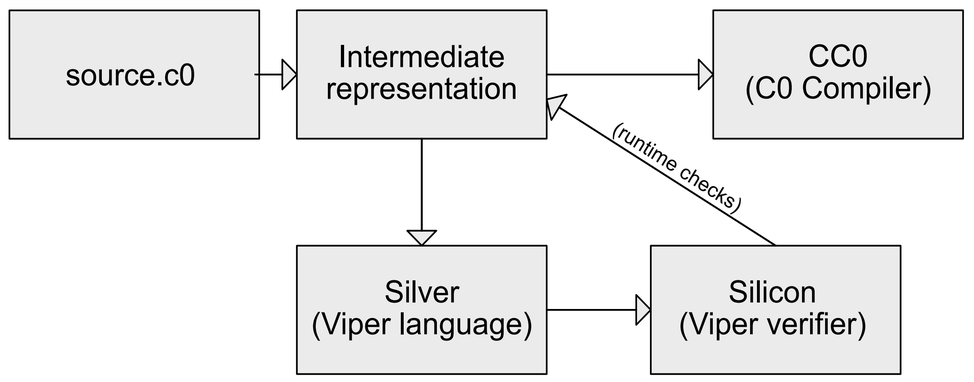
Example

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

void monthEnd(Account* account)
  /*@ requires ? &&
      acc(account->balance); @*/
  /*@ ensures ? &&
      acc(account->balance) &&
      account->balance >= 0; @*/
{
  if (account->balance <= 100)
    withdrawFee(account);
}
```

C0 → Viper → C0

- C0: a safe subset of C with support for dynamically-verified specifications.
- Viper: a program verification framework with frontends which statically verify code in various languages.
- Gradual Viper: A fork of Viper with support for gradual verification.
- Our frontend compiles C0 to Gradual Viper, statically verifies the specifications, and outputs generated C0 code which includes dynamic verification of assumptions made by the gradual verifier.



Side-effects are reasoned about statically using Implicit Dynamic Frames (IDF), where access to memory locations is specified using `acc(object->field)`.

Implementing gradual verification required supporting the dynamic verification of IDF concepts.

Program states are represented by the verifier as formulas in a resource logic.

Static information at the end of `withdrawFee`:

```
acc(account->balance) &&
  account->balance >= 0 &&
  account->balance == old(account->balance) - 5
```

`?` allows the verifier to assume anything necessary to complete proofs.

Assumption in order to statisfy the precondition of `withdrawFee`:

```
acc(account->balance) &&
  account->balance >= 5
```

Wherever specifications are strengthened by the verifier, dynamic checks are inserted into the compiled program to ensure proper behavior at runtime:

```
assert(account->balance >= 5);
```

Conclusion

- Unique challenges in dynamically verifying resource logic specifications.
- Implemented first platform that allows gradual verification to be used on real programs.
- Evaluating the runtime costs and usability of gradual verification is left for future work.