COMP 4161

NICTA Advanced Course

Advanced Topics in Software Verification

Gerwin Klein, June Andronick, Toby Murray, Rafal Kolanski,

+ Thomas Sewell

$$\{P'\} \ \ldots \ \{Q'\}$$
$$\Downarrow$$
$$\{P\} \ \ldots \ \{Q\}$$

Previously in this series

- Program verification, Hoare logic and invariants.
- Real C programs
  - Side effects.
  - Types (fixed-width words, arrays, structs)
  - C memory (pointers, heap representation)
  - Control flow (for, break, continue, return, etc)
  - Undefined execution (null pointers etc, Simpl Guard)
  - VCG
- C/SIMPL/VCG alternatives
  - State monads & equalities
  - AutoCorres

This time

Today's lecture will not be a chaotic collection of demos.

Instead, we will cover some theory behind the mechanisms we've seen:

- Deep and shallow embeddings, computation on functions
- Varieties of Monads
- Abstraction and Refinement

Deep and Shallow Embeddings

We've seen a few examples of program encodings in Isabelle/HOL.

A deep embedding encodes the **syntax** of the program.

A **shallow** embedding uses features of the host logic (e.g. Isabelle/HOL's λ and function type) to encode the semantics of the program.

Deep and Shallow Embeddings

NICTA

We've seen a few examples of program encodings in Isabelle/HOL.

A deep embedding encodes the **syntax** of the program.

A **shallow** embedding uses features of the host logic (e.g. Isabelle/HOL's λ and function type) to encode the semantics of the program.

This means that semantically equivalent programs cannot be distinguished.

Examples of Embeddings

- The simple imperative languages we've seen are deeply embedded
- The state monad language is shallowly embedded
- The SIMPL language is 80% deeply embedded

Advantages of Shallow Embeddings

Advantages of shallow embeddings:

- Standard language features (case statements, variable passing) don't have to be reinvented.
- No need for an "executor" to convert syntax to semantics.
- Equivalent programs are equal, so equality-driven tools (like Isabelle/HOL's simplifier) can be applied.

Advantages of Shallow Embeddings

NICTA

Advantages of shallow embeddings:

- Standard language features (case statements, variable passing) don't have to be reinvented.

f >>= $(\lambda\ x.$if x then g else $h)$
do $x \leftarrow f$; if x then g else h od

- No need for an "executor" to convert syntax to semantics.
- Equivalent programs are equal, so equality-driven tools (like Isabelle/HOL's simplifier) can be applied.

Advantages of Shallow Embeddings

NICTA

Advantages of shallow embeddings:

- Standard language features (case statements, variable passing) don't have to be reinvented.

$f >>= (\lambda\, x.\text{if } x \text{ then } g \text{ else } h)$
do $x \leftarrow f$; if x then g else h od

- No need for an "executor" to convert syntax to semantics.
- Equivalent programs are equal, so equality-driven tools (like Isabelle/HOL's simplifier) can be applied.

$(f >>= g) >>= h = f >>= (\lambda\, x.\ g\ x >>= h)$
do (do $x \leftarrow f$; $g\ x$ od; $h\ x$ od) $=$ do $x \leftarrow f$; $y \leftarrow gx$; hy od

Advantages of Deep Embeddings

The advantage of a deep embedding is that computations on the program can be defined within the logic.

The SIMPL Hoare logic VCG is defined as a term in Isabelle/HOL
$vcg :: (\sigma \text{ set}) \Rightarrow (\sigma \text{ com}) \Rightarrow (\sigma \text{ set}) \Rightarrow \text{bool}$

The Hoare rules are definitions
$vcg\ Pre\ (\text{Basic } f)\ Post \equiv (\forall s \in Pre. f\ s \in Post)$

The VCG is proven to be sound and complete. This would not be possible for a shallowly embedded language.

Grandiose observations

The SIMPL language is a poor example of a deeply embedded language. The statement structure is deeply embedded, but all the expressions are shallow.

For instance, you can't define (in Isabelle/HOL) a program that collects all the references to some variable.

The deep/shallow embedding issue exists in functional languages too. Should functions in a Haskell EDSL have the function type? This becomes an tradeoff between efficiency and flexibility.

Grandiose observations

The SIMPL language is a poor example of a deeply embedded language. The statement structure is deeply embedded, but all the expressions are shallow.

For instance, you can't define (in Isabelle/HOL) a program that collects all the references to some variable.

The deep/shallow embedding issue exists in functional languages too. Should functions in a Haskell EDSL have the function type? This becomes an tradeoff between efficiency and flexibility.

The exception to the rule is LISP.

On Monads

If you use Haskell, at some point you should think about the Monad concept.

Monads are programs for which the handy do-notation makes sense.

Different kinds of monads have different meanings for ";".

```
do
 x <- f 1;
 y <- g x 3 x;
 unless (y > 2) panic;
 z <- h x y;
 case z of
   Nothing -> return ()
   Just v -> commit_value v
```

Monad Demo

What kinds of monads can we think of?

DEMO

Abstraction and Refinement

The AutoCorres tool produces a simplified version of a C/SIMPL program. We can prove that this simplified program has desired properties.

AutoCorres also proves that the C/SIMPL program is a **refinement** of the simplified monadic program.

Refinement is the property that all observations of a concrete program p_c are also visible on an abstract program p_a, or that properties proven of p_a hold of p_c.

Refinement

Formally, refinement, $p_c \sqsubseteq p_a$, states that, for all traces of p_c there exists a trace of p_a for which the observable parts of the state are the same.

Different models of computation come with different observations and refinement orders.

Refinement is equivalent to Hoare triple implication
$\forall\ P\ Q\ trs. \{P\}\ p_a\ _{trs}\{Q\} \longrightarrow \{P\}\ p_c\ _{trs}\{Q\}$

Refinement vs Abstraction

We are doing **abstraction**, the opposite of refinement. We started with a concrete C program and semantics.

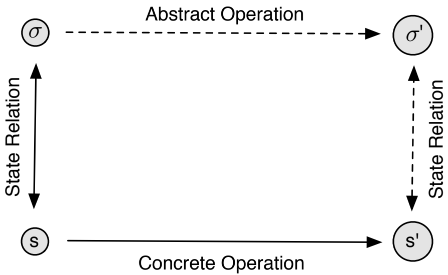
This is backwards compared to the formal software engineering approach.

The formal idea is to start with a specification and derive an implementation via refinement.

Simulation

Refinement is often proven by **forward simulation**.
A trace of p_c is related to one of p_a step by step.

Given a state relation on the states of p_a and p_c and related states s_a and s_c, we prove that every step forward to s'_c has a related state s'_a.



Abstraction, Refinement and Hoare Triples

By abstraction we're showing that p_a can be used as a stand-in for p_c.

This is related to our Hoare triple proofs.

$\{P\}\ p_c\ \{Q\} \equiv p_c \sqsubseteq (\lambda s.\text{if } s \in P \text{ then choose } Q \text{ else choose } \mathfrak{U})$

Hoare triples also transport down refinement.

Safety Properties

Refinement and Hoare triples are related to each other because they characterise **safety properties.** Safety properties are properties programs have if they never enter certain unsafe states.

Liveness properties are satisfied by programs if something good eventually happens.

Confidentiality properties or **information flow properties** require the observers of a program not to learn private information. These are paired with **integrity properties** (which are safety properties) to give **security properties.**

Safety properties are probably the important ones.

Next time

The next lecture will leave program properties behind and focus on proof methods.