COMP3141

Software System Design and Implementation

Lecture 8: Static Analysis, Phantom Types

Zoltan A. Kocsis
University of New South Wales

Term 2 2022

Announcements

- **Assignment 2 Due:** 02 Aug 2022
- A milestone: so far, only technical difficulties with marking, beyond our control.
- But today: I made a mistake with marking Exercise 4.

My Ex04 mistake

Some of you (I know about 3 people) lost a mark on Exercise 4 Part 2, with the following:

```
*** Failed! (after 1 test):
Exception:
  Test.hs:(93,1)-(98,44): Non-exhaustive patterns in
                          function equals
Null
Null
```

Those of you who implemented any "null folding" in your smart constructors, e.g.

```
     node Null Null = Null
```

lost a mark because of this.

Your Ex04 mistake

You complained about this on the forums, and – unfortunately for me – I thought I made a mistake in the marking scripts.
This morning, when I looked at the spec:

Ex04 spec

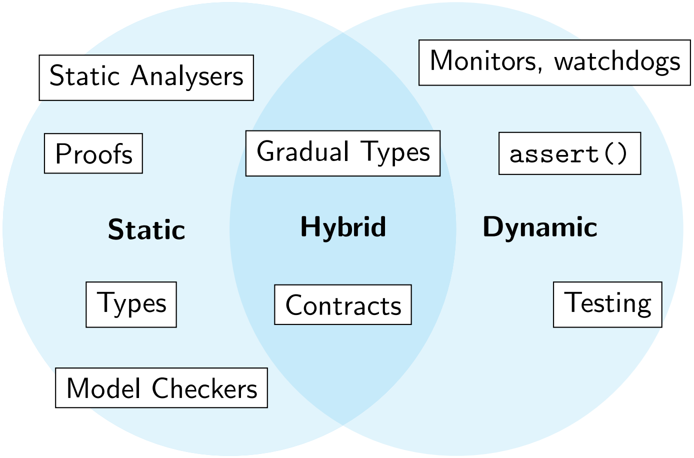
Then, define two smart constructors, which **behave exactly like their counterparts** `Leaf` and `Node`, but automatically calculate the `NodeInfo` values so as to maintain the invariants.

I realized that I didn't. Those who implemented "null folding" had wrong solutions, and should have lost this mark fair and square.

My Ex04 mistake

Unfortunately for me – but fortunately for you, I already changed the marking scripts by the time I realized this, and instructed James to mark the exercise again. *This* was my mistake. Since I don't want to force James to re-mark the exercise yet again, I will accept the null-folding solution, and those of you who implemented it will find that they have an extra mark compared to yesterday's results, even though their solution was wrong.

Methods of Assurance



Static means of assurance analyse a program without running it.

Static vs. Dynamic

- Static checks can be exhaustive.

Exhaustivity

An exhaustive check is a check that is able to analyse all possible executions of a program.

- **However**, some properties cannot be checked statically in general (halting problem), or are intractable to feasibly check statically (state space explosion).
- Dynamic checks cannot be exhaustive, but can be used to check some properties where static methods are unsuitable.

Compiler Integration

Most static and all dynamic methods of assurance are not integrated into the compilation process.

- You can compile and run your program even if it fails tests.
- You can change your program to diverge from your model checker model.
- Your proofs can diverge from your implementation.

Types

Because types are integrated into the compiler, they cannot diverge from the source code. This means that type signatures are a kind of machine-checked documentation for your code.

Static Checks are Possible

Theorem (H. G. Rice)

All non-trivial properties of partial computable functions $\mathbb{N} \to \mathbb{N}$ are *undecidable*. A property is non-trivial if it is neither true for every partial computable function, nor false for every partial computable function.

When you have a property of a program, it may be:

- **semantic**: one about the function computed by the program (does the program terminate for all inputs, does it return 2 for any input, etc.)
- **syntactic**: e.g. does the program contain an if-then-else statement etc.

Syntactic properties may be decidable; by Rice's theorem semantic ones aren't. But syntactic properties can imply semantic ones (no loops, no recursive calls implies terminating).

Types

Types are the most widely used kind of formal verification in programming today.

- They are checked automatically by the compiler.
- They can be extended to encompass properties and proof systems with very high expressivity (covered next week).
- They are an exhaustive analysis.

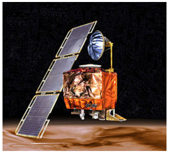
In the next two week, we'll look at techniques to encode various correctness conditions inside Haskell's type system.

Phantom Types

We'll start with Phantom Types.

Units of Measure

In 1999, badly written software confusing units of measure (U.S. Customary unit of force Pounds and SI/Metric unit of force Newtons) caused the Mars Climate Orbiter to burn up on atmospheric entry.



Demo 1: Units of Measure

Phantom Types

Definition

A phantom type is a data type that has a type parameter which does not occur in the type of any argument to any of its constructor.

Examples:

```haskell
data DoubleUnit u = DoubleUnit Double
data NestedList r a = NestedList [[a]]
```

Non-examples:

```haskell
  data Maybe a = Nothing | Just a
  data NamedMaybe e = NM String (Maybe e)
```

Borderline but non-example:

```haskell
data StringWith r = Nil | Cons Char (StringWith r)
```

Phantom Types: Uses

Use cases:

- We can use this parameter to track what data invariants have been established about a value.
- We can use this parameter to track information about the representation (e.g. units of measure).
- There are some non-use-cases where regular old data types are preferable: the "database IDs" example you see all over the Internet is one such.

Demo 2: Student IDs

Datatype Promotion

```
data UG
data PG
data StudentID x = ZID Int
```

Defining empty data types for our tags is untyped. We can have `StudentID UG`, but also `StudentID String`.

Recall

Haskell types themselves have types, called kinds. Can we make the kind of our tag types more precise than `*`?

The `DataKinds` language extension lets us use data types as kinds:

```
{-# LANGUAGE DataKinds, KindSignatures #-}
data Stream = UG | PG
data StudentID (x :: Stream) = SID Int
-- rest as before
```

Making Illegal States Unrepresentable

Demo 3: Using Phantom Types (File IO, Read Write mode)
Demo 4: Type Golf (Soccer Plays)

FIN

1. **Thanks!**
2. The quiz is due 23:59 Thursday, 27 July 2022.
3. The exercise is due 09:10 Thursday, 27 June 2022.
4. The assignment is due 23:59 Tuesday 02 Aug 2022.