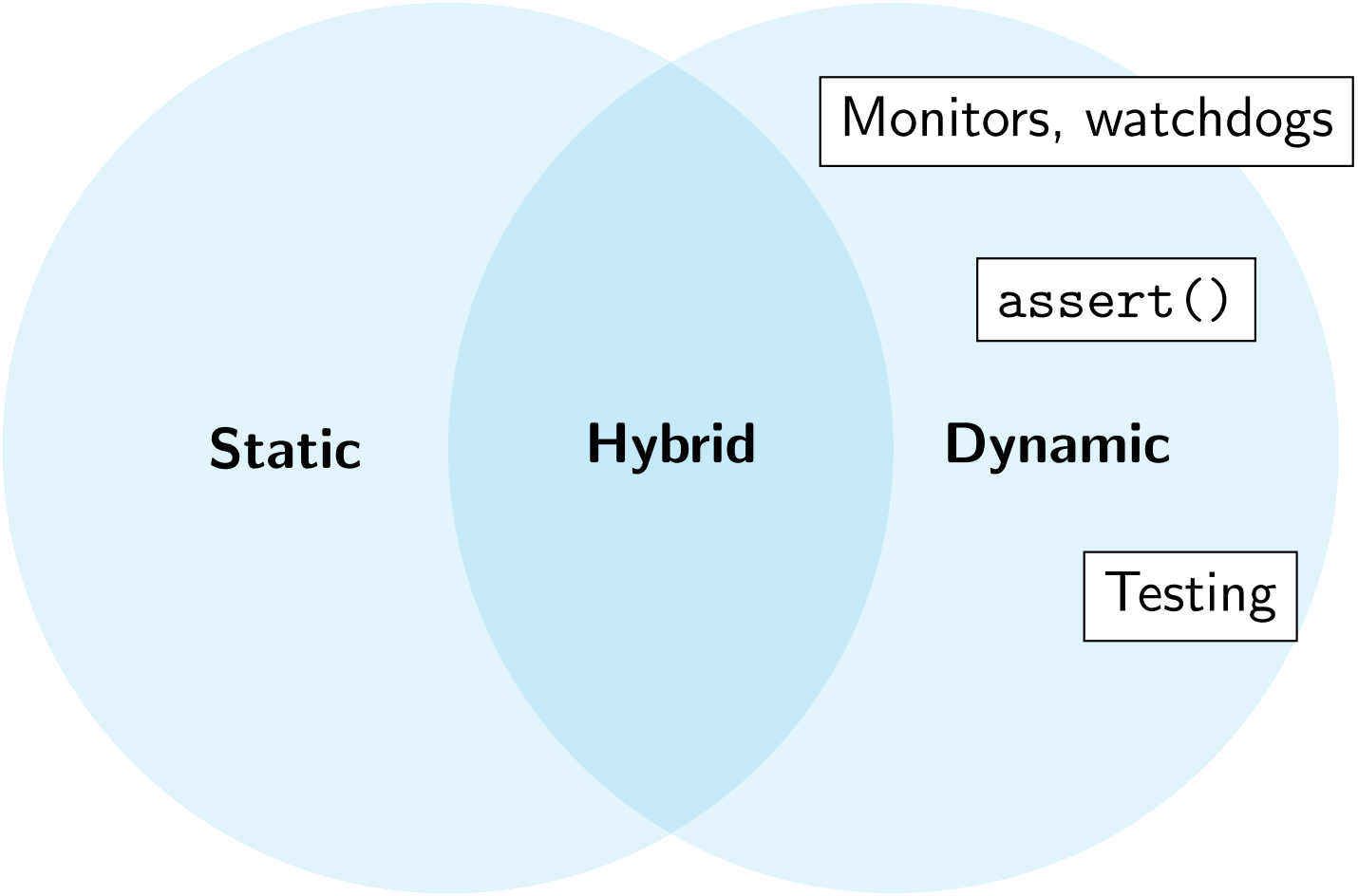
Software System Design and Implementation

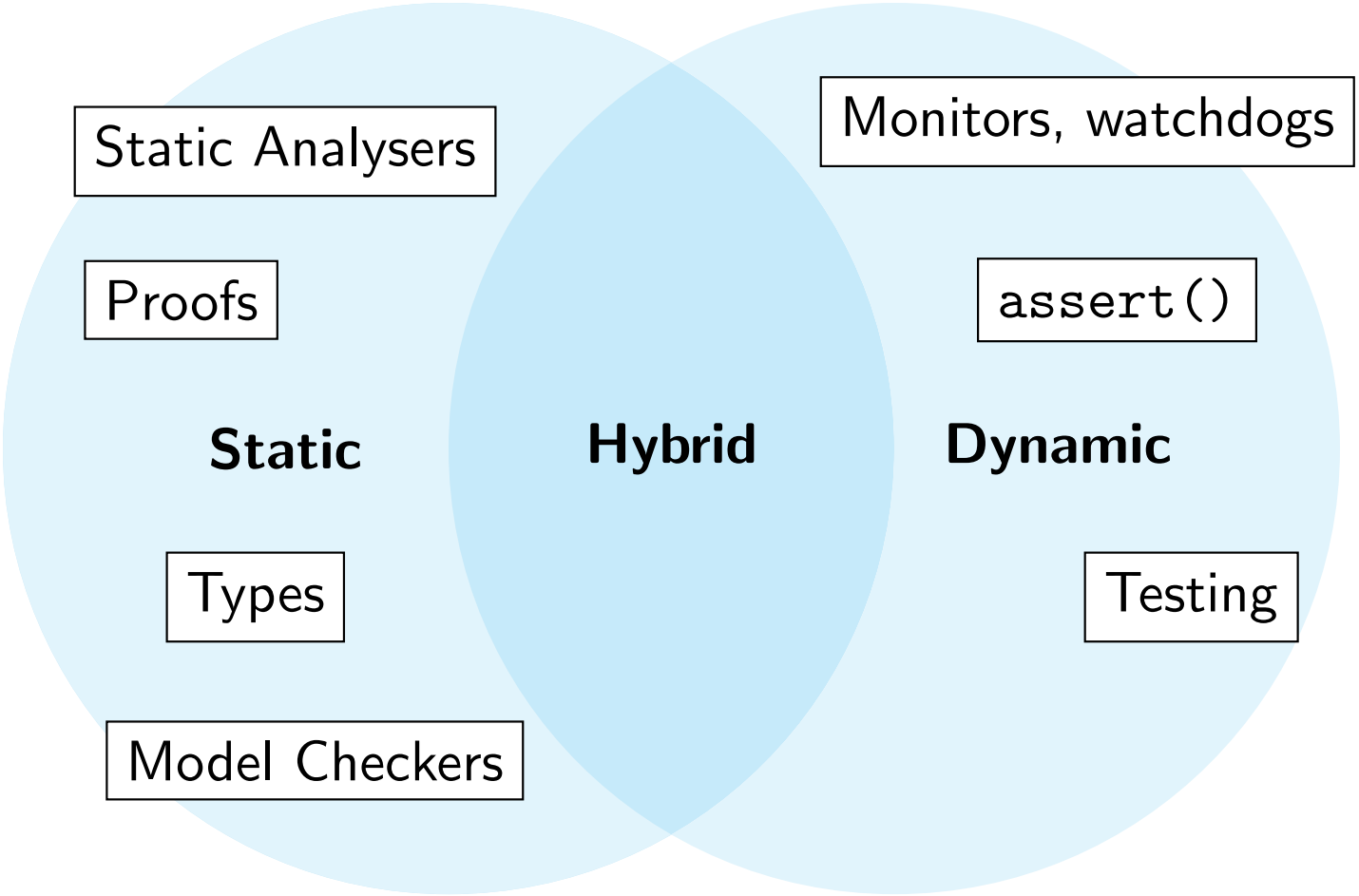
Lecture 8: Static Analysis, Phantom Types

Johannes Åman Pohjola
University of New South Wales
Term 2 2023

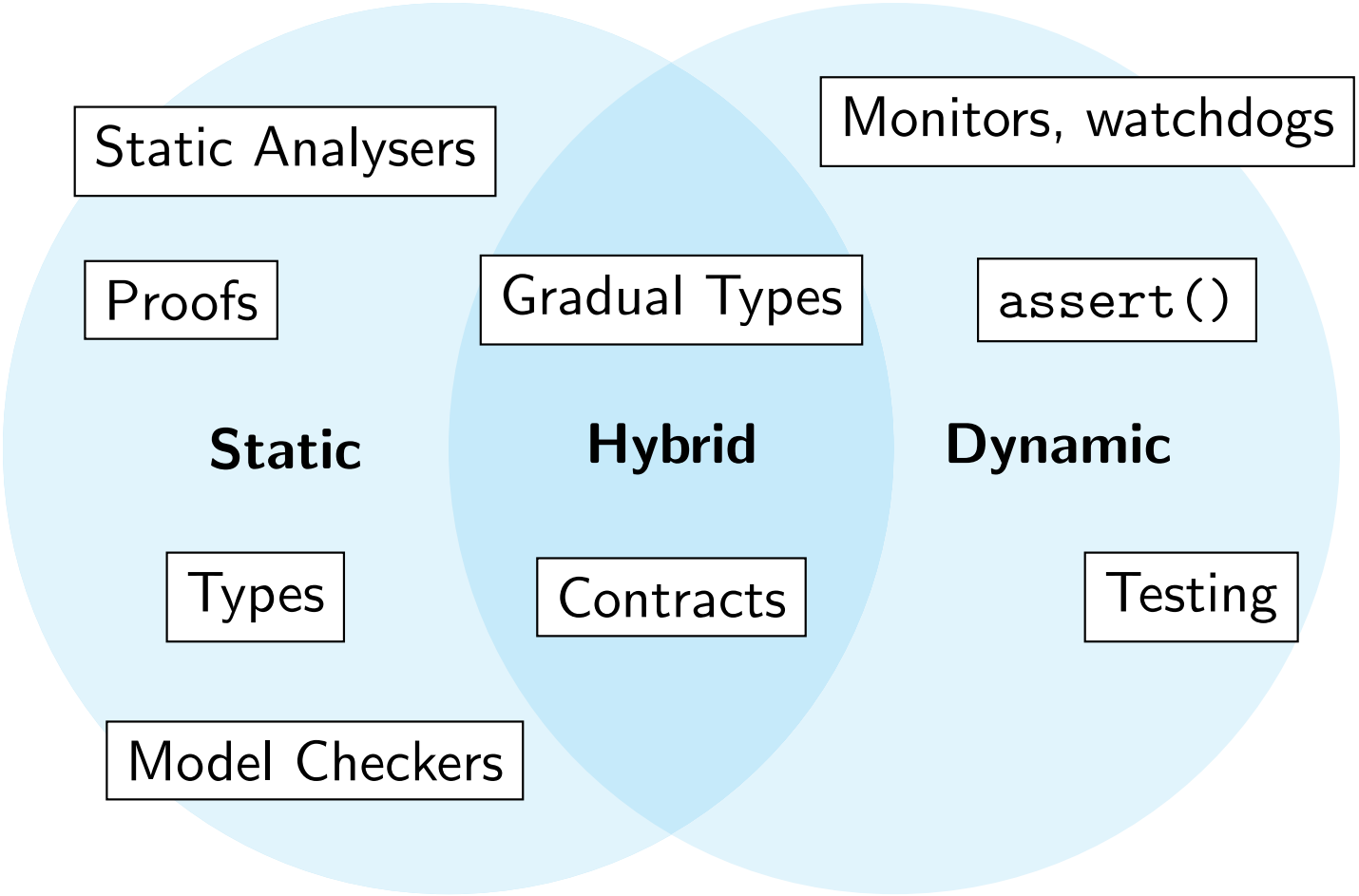
Methods of Assurance



Methods of Assurance



Methods of Assurance



Static means of assurance analyse a program without running it.

Static vs. Dynamic

- Static checks can be exhaustive.

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.

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.

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.

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.
- Your proofs can diverge from your implementation.

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.
- 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.

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**: about the function computed by the program
- **syntactic**: about the program text

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**: about the function computed by the program
- **syntactic**: about the program text

Syntactic properties may be decidable; by Rice's theorem semantic ones aren't. But syntactic properties can imply semantic properties.

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.

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.

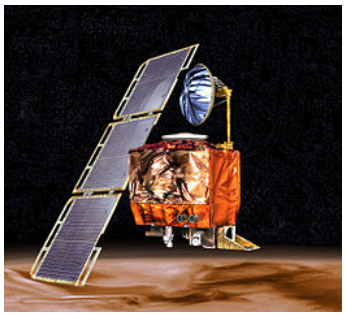
In the next two weeks, we'll look at techniques to encode various correctness conditions inside Haskell's type system.

Phantom Types

We'll start with Phantom Types.

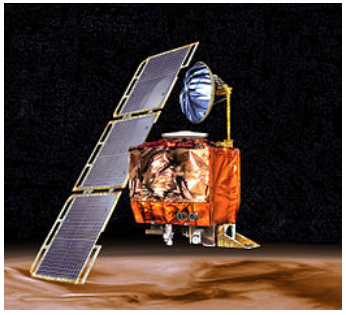
Units of Measure

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.



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.

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:

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

Non-examples:

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

Borderline but non-example:

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

Phantom Types

- We can use this parameter to track what data invariants have been established about a value.

Phantom Types

- 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).

Phantom Types

- 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
```

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`.

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 $*$?

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

If time, more demos!

FIN

1. **Thanks!**