

# Program Synthesis

---

An overview of logic-based approaches

Ricardo Brancas

# Presentation Outline

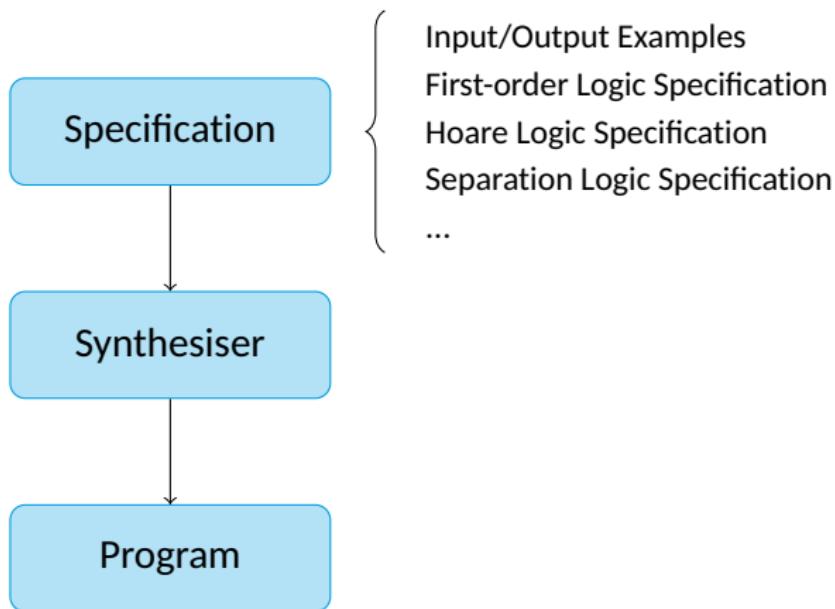
- 1. What is it?**
2. Logic Background
3. A very simple synthesiser
4. A modern synthesiser: Synquid

# Program Synthesis

Specification

Program

# Program Synthesis



# Formal Verification

How can we check if a program is correct (wrt. to the specification)?

# Formal Verification

How can we check if a program is correct (wrt. to the specification)?

We can use a logic formulation.

# Presentation Outline

1. What is it?
2. Logic Background
3. A very simple synthesiser
4. A modern synthesiser: Synquid

# First-order Logic I

## First-order Logic (FOL)

A superset of propositional logic, adding predicates, functions and quantifiers.

## Propositional Logic

$$P \wedge Q$$

$$Q \implies P \vee R$$

# First-order Logic I

## First-order Logic (FOL)

A superset of propositional logic, adding predicates, functions and quantifiers.

## First-order Logic

$\forall x. \text{Panda}(x) \implies \text{Mammal}(x)$

$\neg \exists x. \text{Panda}(x) \wedge \neg \text{Panda}(\text{father-of}(x))$

# First-order Logic II

## Validity & Satisfiability

A formula is **satisfiable** if it is True for some assignment to the variables/functions.

It is **valid** if it is True for all possible assignments.

# First-order Logic II

## Validity & Satisfiability

A formula is **satisfiable** if it is True for some assignment to the variables/functions.

It is **valid** if it is True for all possible assignments.

## FOL Decidability

First-order logic is undecidable.

# Decidability

## Decidable Problem

A problem is **decidable** iff there is a way of deriving the correct answer.

## Undecidable Problem

Complementary, if a problem is **undecidable** then it is *provably impossible* to create an algorithm that always derives the correct answer.

# Decidability

## Decidable Problem

A problem is **decidable** iff there is a way of deriving the correct answer.

## Undecidable Problem

Complementary, if a problem is **undecidable** then it is *provably impossible* to create an algorithm that always derives the correct answer.

As such, it is impossible to determine if any given formula is valid.  
So how can we solve this?

# Satisfiability Modulo Theories

## Satisfiability Modulo Theory (SMT)

A decision problem on **decidable subsets** of first-order logic.  
Such subsets are called **theories**.

### Theory: Presburger arithmetic - $\{ \mathbb{N}, + \}$

$$3x + 2y < 3$$

$$x + y + z = 45$$

# Satisfiability Modulo Theories

## Satisfiability Modulo Theory (SMT)

A decision problem on **decidable subsets** of first-order logic.  
Such subsets are called **theories**.

### Theory: Presburger arithmetic - $\{ \mathbb{N}, + \}$

$$3x + 2y < 3$$

$$x + y + z = 45$$

### Theory: Equality with Uninterpreted Functions

$$f(b) = d \wedge f(a) = d \wedge a = d$$

## Satisfiability Modulo Theories - Models

A model can be seen as a mapping from variables to constants/functions, and represents a solution of the formula.

Example: Model for  $\{3x + 2y < 3\}$

$\{x \mapsto 0, y \mapsto 1\}$

# Satisfiability Modulo Theories - Models

A model can be seen as a mapping from variables to constants/functions, and represents a solution of the formula.

Example: Model for  $\{3x + 2y < 3\}$

$$\{x \mapsto 0, y \mapsto 1\}$$

Example: Model for  $\{f(b) = d \wedge f(a) = d \wedge a = d\}$

$$\{a \mapsto *_1, b \mapsto *_2, d \mapsto *_1, f \mapsto \lambda x. *_1\}$$

# Other Useful Theories for PL

- Bit vectors
- Arrays
- Pointer logic
- Quantified Theories
- ...

Different theories can also be combined while retaining decidability.

# Formal Verification

How can we check if a program is correct (wrt. to the specification)?

We can use a logic formulation.

## Simple Hoare Triple

```
{ X = 3 }
Y ::= X - 2;;
X ::= X - 1
{ X = 2 ∧ Y = 1}
```

# Formal Verification

How can we check if a program is correct (wrt. to the specification)?

We can use a logic formulation.

## Simple Hoare Triple

```
{ X = 3 }  
Y := X - 2;;  
X := X - 1  
{ X = 2 ∧ Y = 1 }
```

## SMT Formula

$$\begin{aligned} & X_0 = 3 \\ \wedge \quad & Y_0 = (X_0 - 2) \\ \wedge \quad & X_1 = (X_0 - 1) \\ \wedge \quad & X_1 = 2 \wedge Y_0 = 1 \end{aligned}$$

# Formal Verification

How can we check if a program is correct (wrt. to the specification)?

We can use a logic formulation.

## Simple Hoare Triple

```
{ X = 3 }
Y := X - 2;;
X := X - 1
{ X = 2 ∧ Y = 1}
```

## SMT Formula

$$\begin{aligned} & X_0 = 3 \\ \wedge \quad & Y_0 = (X_0 - 2) \\ \wedge \quad & X_1 = (X_0 - 1) \\ \wedge \quad & X_1 = 2 \wedge Y_0 = 1 \end{aligned}$$

We say that a program is correct, if the corresponding SMT formula is satisfiable.

# Presentation Outline

1. What is it?
2. Logic Background
3. A very simple synthesiser
4. A modern synthesiser: Synquid

# Generating Programs

Now we can check if a given program is correct. But how do we generate them?

# Generating Programs

Now we can check if a given program is correct. But how do we generate them?

## Enumeration

We can use the grammar of the language to generate (enumerate) all possible programs.

# Generating Programs

Now we can check if a given program is correct. But how do we generate them?

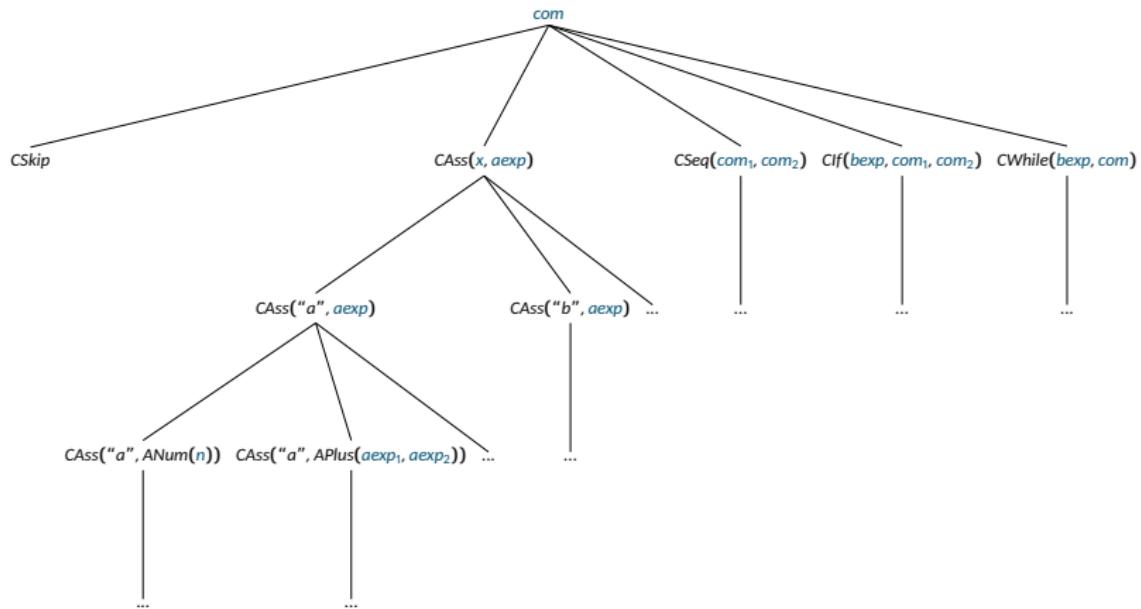
## Enumeration

We can use the grammar of the language to generate (enumerate) all possible programs.

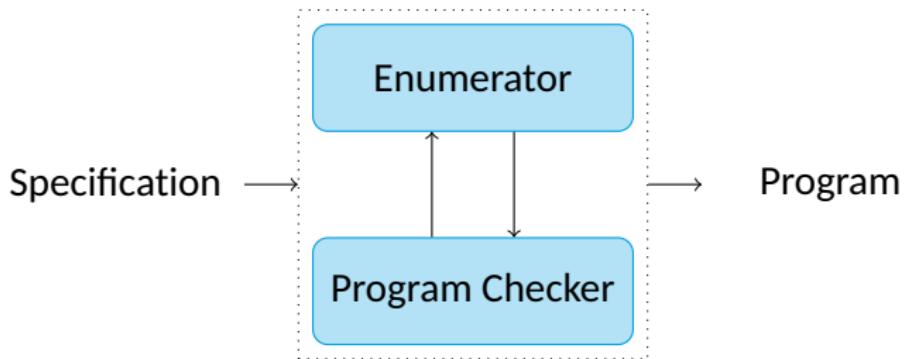
## Problem

The space of possible programs of fixed length, is exponentially large. It is impossible to just check all programs.

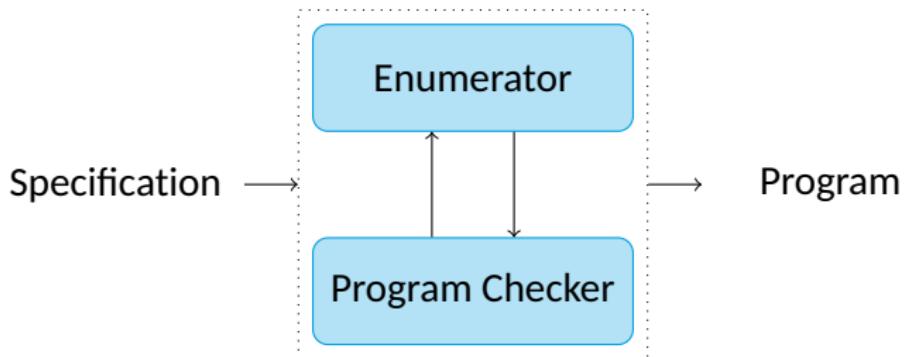
# Generating Programs (Imp language)



# The simplest synthesiser



# The simplest synthesiser



## Problem

This idea is very simple, but it has a big problem: most of the time is spent testing very similar programs that will never lead to a solution.

# Possible Improvements

- Domain Specific Languages (DSL)
- Partial evaluation
- Conflict-driven learning
- Type theory
- Synthetic separation logic
- ...

# Presentation Outline

1. What is it?
2. Logic Background
3. A very simple synthesiser
4. A modern synthesiser: Synquid

## Synquid [1]

Synquid is a modern synthesiser that uses *polymorphic refinement* types in order to prune the search space and make deductions about the program.



Nadia Polikarpova

# Logically Qualified Data Types

## Liquid Types [2]

Liquid Types is a way for automatically deriving refinement types.

### Refinement types

A combination of a “regular type” and a **refinement**.

A refinement is a logic restriction on the values of the type.

# Logically Qualified Data Types

## Liquid Types [2]

Liquid Types is a way for automatically deriving refinement types.

### Refinement types

A combination of a “regular type” and a **refinement**.

A refinement is a logic restriction on the values of the type.

### Example

```
i :: {ν:Int | 1 ≤ ν ∧ ν ≤ 99}
```

$\nu$  is the notation used to represent the value.

## Example: replicate

Taking a number  $n$  and some object  $x$  of type  $\alpha$ ,  
return a list containing  $n$  copies of  $x$ .

### Specification

```
replicate :: n:Nat → x:α → {ν>List α | len ν = n}  
replicate = ??
```

# Example: replicate

## Specification

```
replicate :: n:Nat → x:α → {ν>List α | len ν = n}  
replicate = ??
```

## Auxiliary components

```
zero :: {ν:Int | ν = 0}  
inc :: x:Int → {ν:Int | ν = x+1}  
dec :: x:Int → {ν:Int | ν = x-1}  
leq :: x:Int → y:Int → {ν:Bool | ν = x≤y}  
neq :: x:Int → y:Int → {ν:Bool | ν = x≠y}
```

# References I

- [1] Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. "Program Synthesis from Polymorphic Refinement Types". In: *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '16. New York, NY, USA: ACM, 2016, pp. 522–538. ISBN: 978-1-4503-4261-2. DOI: [10.1145/2908080.2908093](https://doi.org/10.1145/2908080.2908093).
- [2] Patrick M. Rondon, Ming Kawaguci, and Ranjit Jhala. "Liquid Types". In: *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '08. New York, NY, USA: ACM, 2008, pp. 159–169. ISBN: 978-1-59593-860-2. DOI: [10.1145/1375581.1375602](https://doi.org/10.1145/1375581.1375602).