ETH zürich

Program synthesis in the visual programming environment Algot

Daniel Nezamabadi
Bachelor thesis
24. August 2022, Zurich

Executive Summary

- Motivation
  - Programming is important in computer science education [1], but difficult to learn [3, 4]
  - Functional programming is said to be especially difficult [5]
- Goal
  - Develop an educational tool to introduce students to functional programming
- Observation
  - Syntax is one source of difficulties in learning to program [6]
- Idea
  - Overcome the syntax barrier using visual programming and programming-by-demonstration
- Execution
  - Develop a prototype of a visual programming environment based on Algot [9] and programming-by-demonstration
- Results
  - Prototype has potential, but requires significant improvements to its Graphical User Interface (GUI)

Contents

Introduction

- Programming is
  - …important (in computer science education)
  - …diverse
  - …difficult to learn (especially functional programming)

  ⇒ Let’s make a learning tool for functional programming!

- Syntax is a source of difficulties: Syntax barrier

- How can we reduce the syntax barrier?

Introduction

- Well-known block programming environment: Scratch [7, 8]
- How does it work?
  - Programming primitives are represented as blocks
  - Combining (compatible) blocks forms a program
- But users are (still) explicitly constructing a program
  - Can we do better?

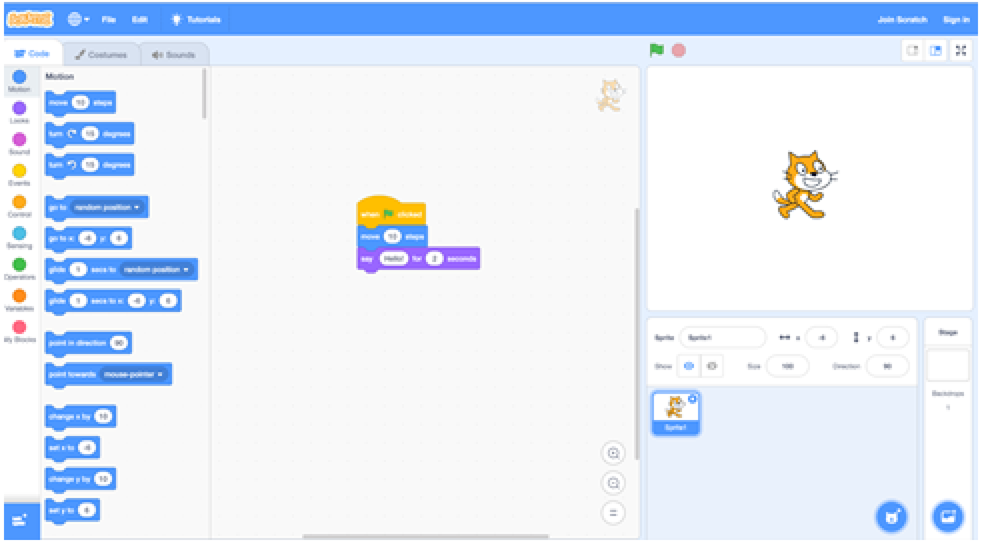

Fig. 1: Scratch User Interface

Introduction

- Visual programming language Algot [9]
  - Programming-by-demonstration: Programming is done by demonstrating examples
  - “The program state should always be visible to the user” [9] (p. 2)
  - “Operations of the program share the same syntactic and semantic meaning [whenever appropriate]” [9] (p. 3)

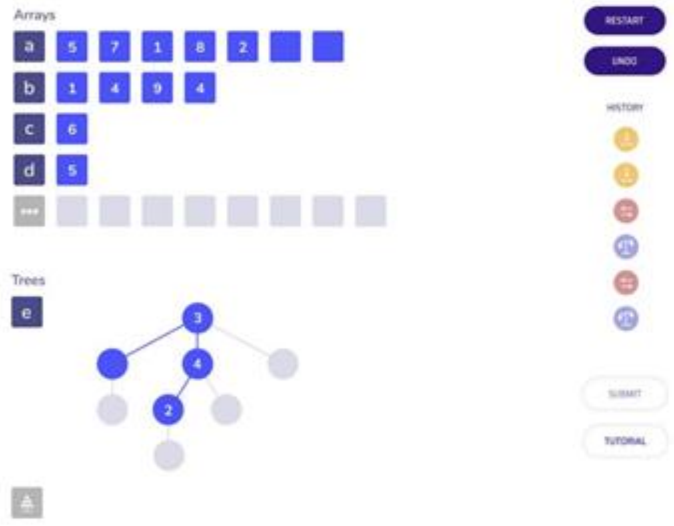

Fig. 2: Algot User Interface [9]

Introduction

- Our work: Prototype a visual programming environment based on
  - the first two design principles of Algot
  - programming-by-demonstration to define custom functions

- Purpose: Introduce students to functional programming, without using any syntax

Contents

Plan

- Implement the system from scratch in Python
- Reference point for functional programming: Haskell
- Why Haskell?
  - Well-known functional programming language
  - Static type system
  - Everything can be understood as a (mathematical) function
    ⇒ facilitates reasoning about programs

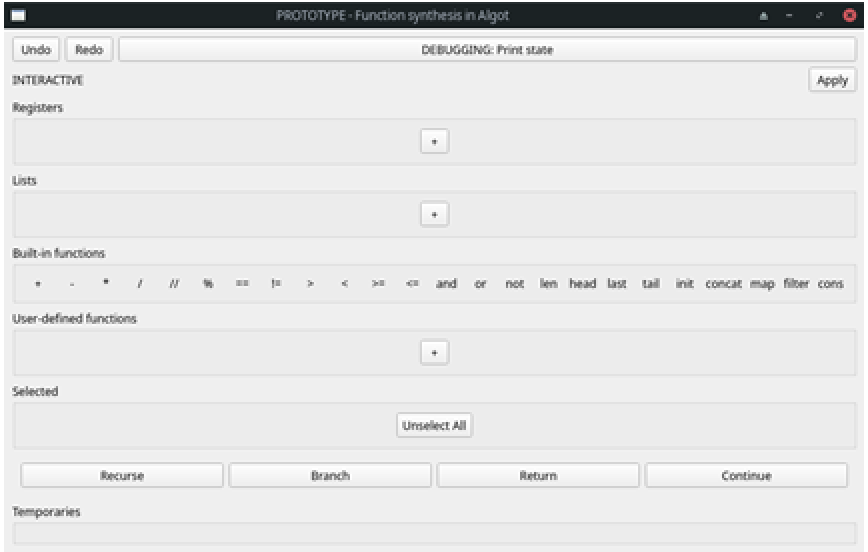

Fig. 3: Prototype user interface

Features

Goals

Following functionality should be supported:

- Values
  - Can be used as inputs to functions
  - Primitive values: number (`Num`) and booleans (`Bool`)
  - Lists of either numbers (`[Num]`) or booleans (`[Bool]`)
  - Functions

- Return types: `Num`, `Bool`, `[Num]` and `[Bool]`

Features

Goals

- Combine existing functions into new functions
- Branching
- Recursion

⇒ User should be able to synthesize many functions from Haskell’s Prelude [11]

Contents

Contents

Interactive mode

Function application

- Process of combining values

- Can be divided into two major steps
  - Type checking
  - Computing the return value

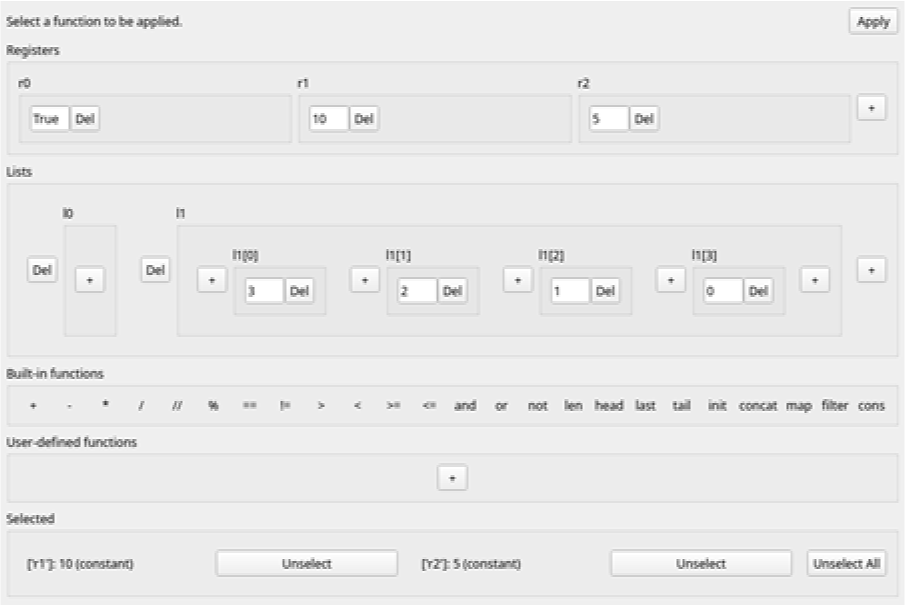

Fig. 4: System prompting the user to select a function to be applied

Interactive mode

Type	Description
`Num`	Numbers (both integers and floats)
`Bool`	Booleans (`True` and `False`)
`[Num]`	List of numbers
`[Bool]`	List of booleans
`a -> b`	Function with one input of type a and output of type b
`a -> b -> c`	Function with two inputs of type a and then b, and output of type c

Table 1: Overview of supported types

Interactive mode

Type checking

- Purpose: Makes sure function application makes sense

- High level algorithm:
    1. Verify number of expected and actual inputs is the same
    2. For every input, verify that the expected and passed input type “match”

- When do types “match”?

Interactive mode

Type checking

1. Verify that #expected arguments = #passed argument = n
2. Generate the set of constraints G = {expected type$_i$ = passed type$_i$ | $1 \leq i \leq n$}
3. Apply unification algorithm

Type checking succeeds ⇔ unification algorithm succeeds

Expected types

1. `a`
2. `Bool`

Passed types

1. `Num`
2. `Bool`

Constraints

`a = Num`

`Bool = Bool`

Unifier

`a = Num`

Type checking succeeds

Fig. 5: Example type checking

Contents

Demonstration mode

Function application

- What should the system infer if the user demonstrates 2 + 2?
  - Many possibilities: 2+2, 2+a, a+2, a+a, a+b, `(f a b)`, …

- Essentially, need to answer two questions:
  - Is the value constant or variable?
  - Which values are represented by the same variable?

- Can the system automatically infer these properties?

Demonstration mode

Function application

- Procedure to automatically infer whether a value is constant or variable:[1]
  - Assume that values are constant
  - Ask user to demonstrate another example
  - For every value that has changed, update the property from constant to variable
- Problem: requires a lot of effort from user
- Alternative: User explicitly differentiates between constant and variable values

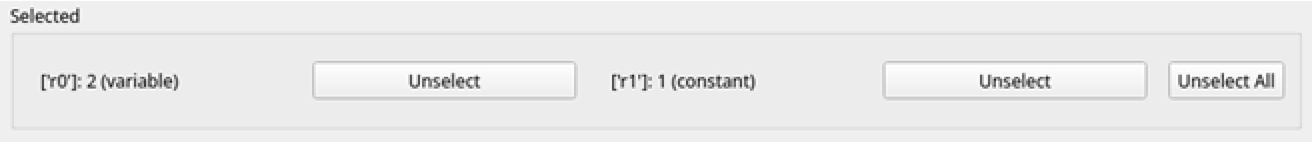

Fig. 6: Representation of a selected variable and constant in the GUI

1. The following described procedure is similar to ideas presented in the section about version space encoding in [13]

Demonstration mode

Type inference

- How can we infer the type of the function to be synthesized?
  - Naive approach: Inferred inputs define the type signature of a function

- Problem: Does not allow type variables in the type of user-defined functions
  ⇒ limits generality of functions

- Can we do better?

Demonstration mode

Type inference

- Key observations:
  - Type signature encodes a set of constraints
  - Constraints are hardcoded or generated while demonstrating examples

- Resulting algorithm:
  - Initialize variables with unique type variables, constants with the type of their value
  - Collect constraints generated by operations
  - Find the most general unifier/type assignment using unification

Demonstration mode

Type inference (Example)

Inputs:

in0, in1, in2

Instructions:

temp0 = + in0 in1

temp1 = in2 temp0

return temp1

Constraints:

`w0 -> w1 -> w3 = Num -> Num -> Num`

`w3 -> w4 = w2`

`w_out = w4`

variable	abstract type
in0	w0
in1	w1
in2	w2
temp0	w3
temp1	w4

Table 2: Mapping from names to abstract types

Add constraint `w_sig = w0 -> w1 -> w2 -> w_out` and apply unification
⇒ Resulting type signature: `w_sig = Num -> Num -> (Num -> w_out) -> w_out`

Contents

Evaluation

Setup

- Qualitative study with two participants:
  - Computer science student (6th semester) with experience in functional programming
  - Electrical engineering student, no experience in functional programming

- Participants were asked to answer questions and solve tasks using a prototype of the system

Evaluation

Results

- Most issues are regarding the GUI (which wasn't the focus of this work)
  - Lack of "documentation"
    ⇒ Good tutorial is important!
  - Lack of information during synthesis
    ⇒ Expose more information in an easy-to-understand way
  - Awkward function application
    ⇒ Use more interactive components

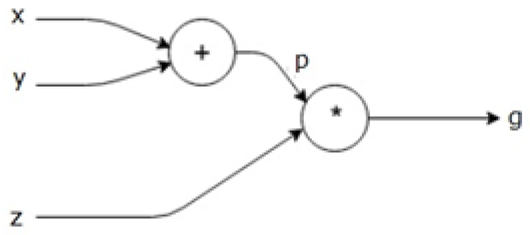

Fig. 7: Example for computational graph

- Both participants think that the tool can be useful to help students get a better understanding of programming, in particular functional programming

Image source for Fig. 7: https://www.tutorialspoint.com/python_deep_learning/python_deep_learning_computational_graphs.htm

Demo

Questions and Discussion

Discussion Starters

- Is this style of programming-by-demonstration useful beyond teaching programming?

- How effective is programming-by-demonstration (going to be) for teaching programming?

- What are alternative or supplementary approaches to teaching programming?

Bibliography

[1] Richard H. Austing, Bruce H. Barnes, Della T. Bonnette, Gerald L. Engel, and Gordon Stokes. Curriculum '78: Recommendations for the undergraduate program in computer science— a report of the acm curriculum committee on computer science. Commun. ACM, 22(3):147–166, mar 1979

[2] Elena Bolshakova. Programming paradigms in computer science education. 2005

[3] Tony Jenkins. On the difficulty of learning to program. In Proceedings of the 3rd Annual Conference of the LTSN Centre for Information and Computer Sciences, volume 4, pages 53–58. Citeseer, 2002

[4] Anthony Robins, Janet Rountree, and Nathan Rountree. Learning and teaching programming: A review and discussion. Computer Science Education, 13(2):137–172, 2003

[5] Abdullah Khanfor and Ye Yang. An overview of practical impacts of functional programming. In 2017 24th Asia-Pacific Software Engineering Conference Workshops (APSECW), pages 50–54, 2017

[6] Paul Denny, Andrew Luxton-Reilly, Ewan Tempero, and Jacob Hendrickx. Understanding the syntax barrier for novices. In Proceedings of the 16th Annual Joint Conference on Innovation and Technology in Computer Science Education, ITiCSE '11, page 208–212, New York, NY, USA, 2011. Association for Computing Machinery

[7] Scratch. `https://scratch.mit.edu/`. [Online; accessed 15-June-2022]

[8] John Maloney, Mitchel Resnick, Natalie Rusk, Brian Silverman, and Evelyn Eastmond. The scratch programming language and environment. ACM Trans. Comput. Educ., 10(4), nov 2010

[9] Sverrir Thorgeirsson and Zhendong Su. Algot: An educational programming language with human-intuitive visual syntax. In 2021 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC), pages 1–5, 2021

Bibliography

[10] Aaron Krauss. Programming concepts: Static vs. dynamic type checking. https://thecodeboss.dev/2015/11/programming-concepts-static-vs-dynamic-type-checking/, 2015. [Online; accessed 05-July-2022]

[11] The University of Glasgow. Prelude. https://hackage.haskell.org/package/base-4.16.1.0/docs/Prelude.html. [Online; last accessed 9-June-2022].

[12] The University of Glasgow. Source code: module GHC.List. https://hackage.haskell.org/package/base-4.9.0.0/docs/src/GHC.List.html. [Online; last accessed 9-June-2022].

[13] Tessa A Lau and Daniel S Weld. Programming by demonstration: An inductive learning formulation. In Proceedings of the 4th international conference on Intelligent user interfaces, pages 145–152, 1998.