ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Program synthesis in the visual programming environment Algot

Bachelor Thesis

Daniel Nezamabadi

August 19, 2022

Supervisors: Prof. Dr. Zhendong Su, Sverrir Thorgeirsson
Department of Computer Science, ETH Zürich

Acknowledgements

First, I would like to thank Sverrir Thorgeirsson for his guidance and support, making this thesis a pleasant experience. I also want to thank Prof. Dr. Zhendong Su for his valuable feedback, especially regarding the potential connection between this work and concolic execution. Finally, I would like to thank Diego de los Santos Gausí and Richard Willke for taking their time to participate in the study and offering valuable feedback.

Abstract

Programming, especially functional programming, is considered to be difficult. One difficulty for beginner programmers is the syntax of "classical" programming languages. In this thesis, we develop a prototype of a visual programming environment based on Algot and the programming-by-demonstration paradigm combined with the style of functional programming to introduce beginner programmers to functional programming without them having to learn the syntax of a classical functional programming language. To evaluate our prototype, we conduct a qualitative study with two participants. Our results show that while the graphical user interface of the prototype requires more work, the prototype has the potential to become a valuable tool for introducing students to the key ideas of functional programming.

Contents

Chapter 1

Introduction

Programming is a core component of computer science education [2], which is why programming courses are part of undergraduate computer science programs. These usually include an introductory programming course and a basic data structure course, usually referred to as "CS1" and "CS2", respectively [5].

At the same time, programming is considered to be difficult to learn [6, 19]. Students struggle significantly with the syntax of programming languages [4], which could hinder them from learning and practising more general concepts like problem-solving.

One approach to teaching programming is using visual programming, for example, block programming. In block programming environments like Scratch [1, 12], programming primitives are represented as blocks that can be combined with other (compatible) blocks, forming a program. While this seems like an attractive approach to reducing the impact of barriers rooted in syntax, we argue that we can do better.

For example, to add two variables in a block programming environment together, the user would need to drag the blocks representing the respective variables onto the block representing the + operation, meaning that the user is still explicitly constructing the program. An alternative would be that the program is automatically constructed. This way, barriers rooted in syntax should effectively be eliminated, as the user never explicitly constructs a program. One way to automatically construct a program would be to infer it from one or more examples demonstrated by the user. This approach is known as programming-by-demonstration and is used as a basis for the visual programming language Algot [20]. According to Thorgeirsson et al., Algot is based on four design principles. In particular, the first two design principles state that "the program state should always be visible to the user", and that "operations of the program share the same syntactic and semantic

meaning [whenever appropriate]" [20] (p. 2, 3).

However, programming is not limited to a single paradigm. Bolshakova states that there are four main paradigms (imperative, functional, logic and object-oriented), each with its strengths and weaknesses. Furthermore, she argues that modern programming requires programmers to know a wide range of programming techniques and hence familiarity with different programming paradigms [3].

One of these programming paradigms is functional programming, which "is generally considered to be difficult to learn and master than another [sic] programming paradigms such as object-oriented programming" [8] (p. 50). This motivates the development of a tool to introduce students to the functional programming paradigm.

To achieve this goal, we develop a prototype of a visual programming environment with support for user-defined functions based on the first two design principles of Algot and the programming-by-demonstration paradigm combined with the functional programming style. More specifically, we use the programming-by-demonstration paradigm to automatically construct programs in the style of pure functional programming languages like Haskell.

We argue that the resulting system can be a helpful tool to introduce students who have a basic understanding of programming concepts like branching and recursion to key ideas from functional programming without requiring them to learn the syntax of a functional programming language like Haskell. By adding additional resources inside the tool, we think the system could also be helpful for people without any programming experience. Such additional resources could be, for example, tutorials which elaborate on concepts like branching, recursion, and the system itself.

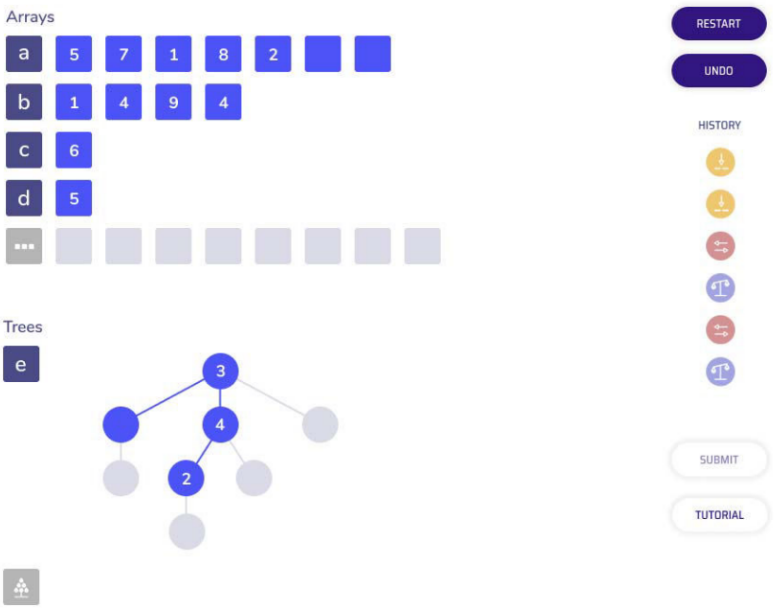

Figure 1.1: "A screenshot of the Algot system. The picture shows a snapshot of a program construction where the user has instantiated several arrays and a tree." [20] (p. 1)

Chapter 2

System description

In this chapter, we will outline the implementation of the system and motivate the various design choices that have been made.

The focus of this thesis is not to directly modify Algot. Instead, we take the first two design principles of Algot and implement a new system from scratch in Python. Recall the first two design principles of Algot mentioned in Chapter 1: "The program state should always be visible to the user" and "operations of the program share the same syntactic and semantic meaning [whenever appropriate]" [20] (p. 2, 3). This way, we are not restricted by the previous implementation. In case the implementation of the system proves to be useful, it (or at least parts of it) can be added to Algot afterwards.

2.1 Foundation

As already mentioned in Chapter 1, we will allow the definition of user-defined functions by implementing a system which maps the programming-by-example paradigm to a pure functional programming style as implemented by the programming language Haskell as transparent to the user as possible. This will be the foundation of our system. Haskell is a well-known representative of pure functional programming languages, which we will use throughout the thesis as a guide for our design decisions and to give examples for code and implementations using an actual (pure) functional programming language.

We choose to map the examples given by the user to a pure functional programming style as implemented by Haskell over an imperative programming style due to the following benefits: Firstly, functions in Haskell are side-effect free. This means that unlike imperative programming languages like C or Java, functions in Haskell may not change any "external state". We define "external state" as any state not explicitly passed to the function as an input.

```c
#include <stdio.h>
int count = 0;

int add(int a, int b) {
    if(count >= 3) {
        // User needs to buy a new license to continue using
        ↪ add
        return 0;
    }

    count += 1;
    return a + b;
}
```

Program Code 1: C function with side-effects

The function `int add(int a, int b)` in Program Code 1 is not side-effect free, because every time the function `add` is used, the variable `count` is incremented, even though it is not explicitly passed as an input. In this example, the value of `count` influences the function's behaviour. In particular, calling `add` with the same inputs `a` and `b` does not guarantee that the output will always be the same, since if `count` is larger than 2, the function will always return 0 instead of `a+b`. Hence, it is usually not enough to only consider the function and its inputs to understand a function's output in a programming language with side effects. Instead, the context, i.e. the implicitly passed state in which it is executed, also needs to be considered. Additionally, it is possible that other functions also depend on the value of `count` by reading and writing to it, leading to potentially complex interactions. This makes reasoning about functions, including function synthesis, in programming styles with side-effects hard.

The second benefit of Haskell is that variables are immutable – once the value of a variable is defined, it cannot be changed. This makes reasoning significantly easier, as we do not have to worry about how the value of a variable may change over time. Note that the variables in Program Code 1 are not immutable, as the value of `count` can be and also is modified.

The third benefit of Haskell is that most, if not all, constructs can be understood as a function. Imperative programming languages usually have constructs like while/for-loops, while Haskell does not: loops are expressed using recursion. Thus, there are fewer constructs to implement and reason about while maintaining the same amount of expressivity. We will return to this fact in Section 2.5.4 on page 28 with a concrete example of a for-loop in C expressed using recursion in Haskell.

The final benefit of Haskell that we will mention here is its static type system. For a statically-typed language, the types of objects are known at compile time, meaning it is possible to check whether any typing rules are violated before the program is run [9]. Typing rules ensure that functions are applied to values that make sense: for example, a static type system could reject programs that try to add a number to a string at compile time. In our case, the system automatically infers the type of any values the user produces and ultimately the type of the synthesized function while they are demonstrating one or more examples. This can help users to avoid demonstrating examples that do not make sense.

2.2 Features

In the end, our system should support the following functionality:

- Values, which can be used as inputs to functions
  - Basic primary/primitive values (number and booleans) and data structures (lists of either numbers or booleans)
  - Functions
- Return types: `Num`, `Bool`, `[Num]` and `[Bool]`, where we use `[X]` to denote the type of a list of `X`s
- Combining built-in and user-defined functions to create new functions
- Branching
- Recursion

To simplify the implementation, we also define some restrictions:

- Functions are not valid return values
- No support for partial function application: whenever a function is used, all of its inputs have to be provided
- No support for user-defined data types

Most of the supported features are basic programming concepts. For example, branching, recursion, and combining functions are essential in imperative and functional programming languages. However, we do not support loops as a separate construct since they are not part of pure functional programming languages like Haskell. This forces users of the system to think using recursion, which is typical for functional programming languages. Additionally, while the current iteration of the system does not support functions as return values, it supports functions as input values, which is also a key feature of functional programming languages. Although imperative programming languages like C allow passing functions to other functions using function

pointers, we do not think someone with only basic programming knowledge would necessarily be aware of this fact.

Thus, in mapping the programming-by-demonstration paradigm to the functional programming paradigm, we expose and, in some cases, even force the user to use features common in functional instead of imperative programming languages. Together with the fact that these features are not placed behind any syntax barriers, we think that our tool can help introduce students to some of the key ideas in functional programming.

In the end, users should be able to synthesize many functions from Haskell's Prelude [16] by demonstrating a concrete example in a functional style. To clarify how our system can be a helpful tool in helping students get familiar with thinking "functionally", we compare possible implementations of `map` in an imperative programming language and a functional programming language.

`map` is a function which takes a function and a list as inputs and applies the passed function to every element in the list. Someone who has mainly been exposed to imperative programming languages would most likely implement `map` by looping through the input list and applying the passed function to every element, storing the result in the same or a new list. However, as already mentioned, this is not possible in Haskell and, by extension, our system since there is no construct for loops.

Instead, we take a more functional approach using recursion. Assume that the function we want to apply is called `f`. First, we check whether the list is empty. If the list is empty, we are done and return the empty list. If there is at least one element in the list, we "remove" the first element of the list and apply `f` to it. After "removing" the first element of the list, we have a strictly smaller list to which we recursively apply `map`. The result will be a list where `f` has been applied to every element in this smaller list. Finally, the element on which we have applied `f` separately is "added back" to the smaller list on which we have recursively applied `map`, and the result is returned.

Note that the procedure using recursion is quite different from the one using a loop and might not be immediately clear to people unfamiliar with functional programming. Nonetheless, using recursion this way is common in functional programming.

The user would execute these high-level steps in our system on a concrete example. The implementation of this procedure in Haskell is given in Program Code 2 on the next page. Arguably, the previously described procedure might not be immediately clear from Program Code 2 to someone unfamiliar with Haskell's syntax. For instance, they might be wondering what the meaning of `_` or `:` is. `:` in particular might be confusing since `(x:xs)` as an argument means that a list passed as an input is decomposed into its

```
map :: (a -> b) -> [a] -> [b]
map _ []     = []
map f (x:xs) = f x : map f xs
```

Program Code 2: Implementation map from [17]

first element called `x`, and the rest of the list called `xs`. On the other hand, `f x : map f xs` combines an element `f x` and a list `map f xs` into a new list, which starts with `f x` and is followed by the elements in `map f xs`, preserving their order. We think that the barrier posed by the syntax of "classical" functional programming languages hinders students from grasping more fundamental ideas like expressing loops using recursion.

By implementing a functional interface for visual programming and the programming-by-demonstration paradigm, we argue that we remove most, if not all, of the syntax barrier, allowing students to concentrate their efforts on learning more fundamental functional programming concepts. Additionally, by only exposing a functional programming interface, students are "forced" to think functionally. Considering this, we believe our system can be a valuable tool to help students expand the range of programming techniques they are familiar with.

2.3 Plan

Two criteria influence design choices regarding managing and interacting with the state: the difficulty of inferring information from examples to synthesize the function the user is trying to define and the ease of use. Note that there is a relation between these two criteria: if we burden the user with all the work, then the implementation of the system becomes easy, but usability decreases. However, forcing the system to infer everything is also problematic, as this may also result in less usability. We will see this phenomenon in more detail in Section 2.5.1 on page 17.

As a first step, we divide the system into two major parts and look at them separately: an interactive part, where the user can apply existing functions to values in the state, and a demonstration part, where the user demonstrates examples to synthesize a new function. These parts are mirrored in the system by different modes. The need for this differentiation will hopefully become clear throughout this chapter. Finally, we will elaborate on how the results of function applications are computed.

2.4 Interactive mode

To implement the interactive mode, we will need to decide what the user can interact with and how they can do so when they are not synthesizing a

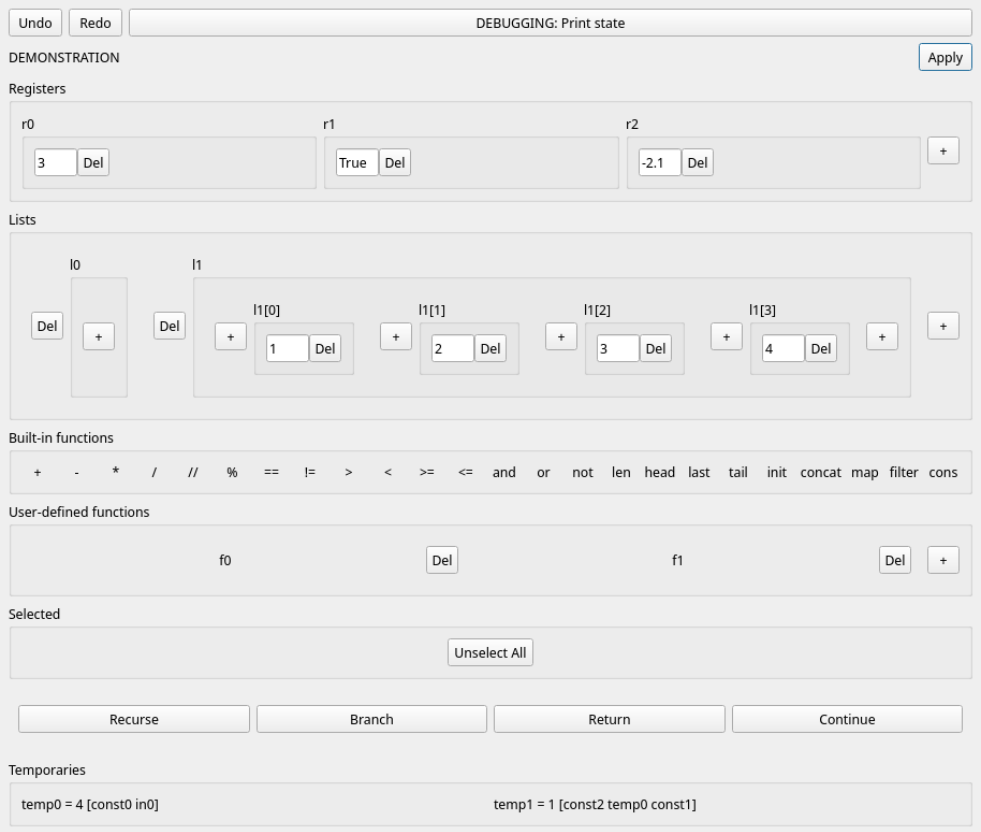

Figure 2.1: Overview of the graphical user interface (demonstration mode)

function. For this purpose, we divide the state in interactive mode into three major parts: registers, lists and functions. Finally, we will discuss how these elements can be combined into new ones using function application.

2.4.1 Registers

Registers hold numbers or booleans. In a programming language like Java, registers would represent variables like `int foo;` `boolean bar;` or `float foobar;`. Internally, registers are stored as a map from unique register names to numbers or booleans. Note that the notion of numbers includes both integers and floating point numbers. The user can create, delete and update registers without any restrictions in interactive mode.

This idea of registers does not exist in Algot, where values are stored in data structures like lists or trees. To understand the motivation behind adding a new kind of "data structure" consider the following situation: Assume that the system stores all values inside of lists. If the user selects the first value of a list, it is unclear whether they mean the value itself (case A) or the first

Figure 2.2: Representation of registers in the GUI

element of the list (case B). In case A, it is enough for the system to select the value itself. However, in case B the system needs to select the list, as the notion of "the first element of the list" does not make sense without a list.

This is also reflected in the resulting types: in case A, we would be selecting a value of type `a`, while in case B, we would be selecting a list of type `[a]`. This difference is significant as we want to utilize a static type system. Thus, having registers which hold values separately from lists, which also contain values, allows us to differentiate between these two cases cleanly.

This difference between lists or arrays and values is also made in languages like Java: we either have a single value `int foo;` or a reference to an array `int[] bar;`.

This is also the first instance where we require the user to give the system explicit information by requiring them to differentiate between a value and a list of values instead of forcing the system to infer it. Arguably, this barely increases the user's mental load while significantly improving the clarity of the system and, by extension, ease of function inference.

While the current implementation only takes into account whether a list was selected and ignores which element from the list was selected for the sake of simplicity, the notion of registers being separate entities from lists is quite intuitive and allows future implementations of the system to take into account which element from the list was selected.

2.4.2 Lists

Lists hold multiple items of the same type, namely numbers or booleans. This is a straightforward and intuitive data structure, which is why it is ubiquitous in modern programming languages in some form or another: arrays are supported by C and Java, while Java, Python and Haskell support lists. The difference between lists and arrays is that, in contrast to arrays, the size of a list can change. Like registers, lists are stored using a map from unique list names to lists internally, and the user can create, delete and update lists without any restrictions in interactive mode. Additionally, the user can insert, delete and update elements within a list.

To access an arbitrary element of a list during a demonstration, the user can define a new function using list functions like `head` and `tail` together

```haskell
-- We assume that n is a valid, non-negative index for xs
nth xs n
    | n == 0 = head xs
    | otherwise = nth (tail xs) (n-1)
```

Program Code 3: Haskell implementation of accessing a list by an index. Adapted `(!!)` implementation from [18]

Figure 2.3: Representation of lists in the GUI

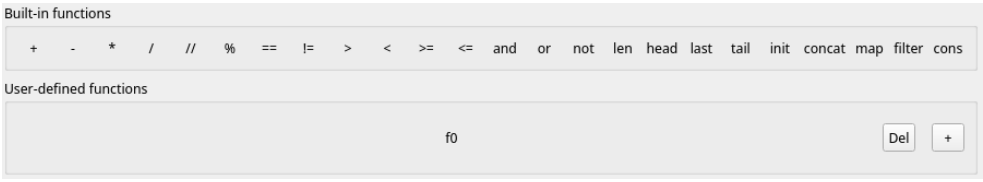

Figure 2.4: Representation of functions in the GUI

with recursion. `head` returns the first element of a list and `tail` returns a list without its first element.

Using functions like `head` and `tail` together with recursion to manipulate lists is a common approach in Haskell; see, for example, Program Code 3. Many of the implemented built-in functions are taken from the Haskell Prelude [16].

2.4.3 Functions

Functions take zero or more input values and return exactly one output value. Valid input types are numbers, booleans, lists and functions, while the output types are restricted to numbers, booleans and lists. In particular, functions may not return functions. Like registers and lists, internally, they are implemented using a map from unique function names to functions. The system provides built-in functions like arithmetic, boolean, and list operations to enable and facilitate the synthesis of user-defined functions. Users may delete functions they have defined or create new ones using the programming-by-demonstration paradigm. We will postpone the discussion of how function synthesis is achieved to Section 2.5 on page 17.

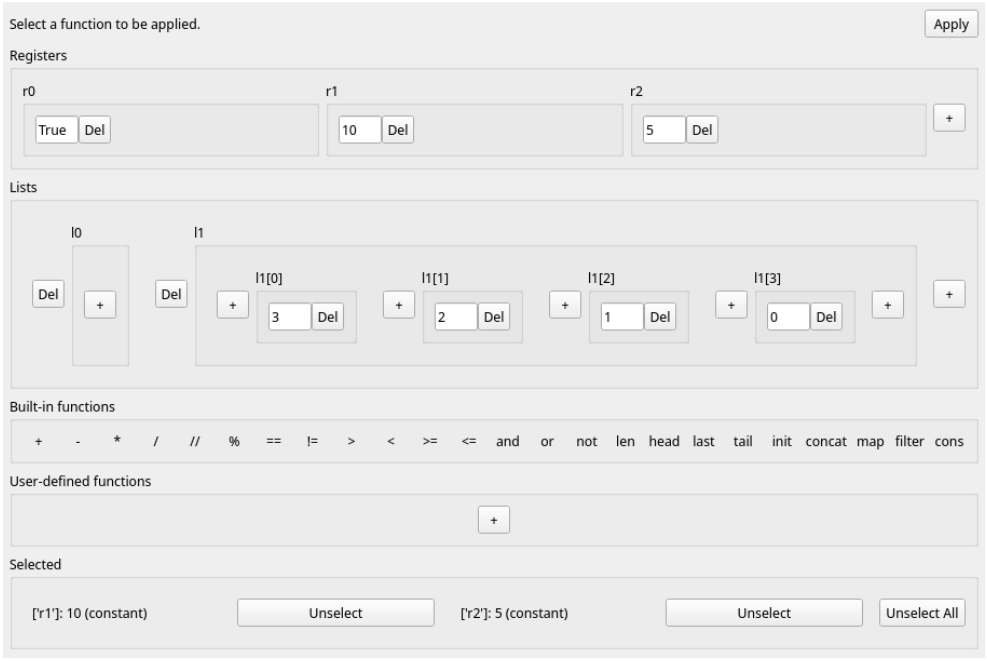

Figure 2.5: System prompting the user to select a function to be applied

2.4.4 Function application

To combine existing values into new ones, the user selects existing ones by clicking on their name and then selects a function to be applied after pressing "Apply". Function application can be divided into two major steps: type checking and computing the return value.

Type checking

First, we lay out the types supported by the system. Numbers and booleans are represented by the types `Num` and `Bool` respectively. The type of a list is represented by `[a]`, where `a` is the type of the values stored inside of the list. Since the system only supports lists of either numbers or booleans, `a` can only be either `Num` or `Bool` and never the type of a function or a list of values. A function, which takes an input of type `a` and returns a value of type `b` is denoted by `a -> b`.

For function types of the form `a -> b`, we understand every type besides the last type as an input to the function. For example, `a -> b -> c` would be a function which takes a value of type `a` as its first input, a value of type `b` as its second input and finally returns a value of type `c`. Because `->` associates to the right, `a -> b -> c` is equivalent to `a -> (b -> c)`, which can be interpreted as a function taking a value of type `a` as an input and returning a function of type `b -> c`. Thus, if we were to only pass an input of

type `a` to a function with type `a -> b -> c`, we would receive a new function of type `b -> c`. This is known as partial application.

The system's current implementation does not directly support partial application, as it would require us to handle functions as return values, which is not supported by the system; see Section 2.4.3 on page 12. Nonetheless, the user can achieve the same result by defining a new function, which applies the function to be partially applied to selected constants and variables. After reading Section 2.5 on page 17, the details of this procedure should become clear.

Note that the system also implements the notion of a type variable, which is usually represented by lowercase letters like `a`, `b` and `c`. A type variable is a placeholder for a type. In general, the only restriction for type variables is that type variables with the same name must represent the same type. However, we may add more restrictions, e.g. only allowing lists of numbers or booleans, to restrict the scope of the system.

To motivate their usefulness, consider the following situation: Assume that we want to define a function which returns the n-th element of a list. As this operation is independent of the type of values the list holds, we would like exactly one function implementing this operation. Without type variables, this would not be possible. This is because a function can only have one type, forcing us to define two functions with types `[Num] -> Num` and `[Bool] -> Bool`. By using type variables, we can define a single function of type `[a] -> a`.

All types can be interpreted as functions taking zero, one or more arguments. For example, `Num` can be interpreted as the function `Num()`, `a` as `Var("a")`, `[b]` as `List(Var("b"))` and `x -> Bool` as `App(Var("x"), Bool())`. This idea is used by unification, the main algorithm involved in type checking.

The type of a value is determined by type inference. Values of type `int` and `float` are inferred to be of type `Num` and values of type `bool` are inferred to be of type `Bool`. Note that `int`, `float` and `bool` are Python types, which is the programming language in which the system is implemented. The system itself uses the types `Num` and `Bool`. The type of a list is determined by the type of the values inside it: if a list contains `int`s and `float`s, its type is inferred to be `[Num]`. An empty list is inferred to be of type `[a]`, as its type is not fully determined due to it being empty. Internally, lists are stored as Python lists. The type of built-in functions is hard-coded, while the type of custom functions is inferred during function synthesis; see Section 2.5.1 on page 17.

At this point, we have established what types are supported by the system and how we can infer these types, with the exception of the type of user-

defined functions. Next, we will motivate and explain the implementation of type checking in the system.

As mentioned in Section 2.1, the primary purpose of type checking is to ensure that functions are not used in a way that does not make sense: for example, applying a function that only takes three arguments to 5 arguments does not make sense. Similarly, applying a function which increments a number by 1 to a boolean does not make sense either.

Without type variables, type checking would be a simple two-step process: First, check whether the number of expected and passed arguments matches. Then, check whether the type of the i-th passed argument exactly matches the expected type for the i-th argument, where $1 \leq i \leq$ number of arguments. Type checking fails if and only if any of these steps fail.

Since type variables are part of the system, type checking is not as straightforward. In particular, the notion of when types match is not clear. A naive definition would be that a type variable matches any type. However, this definition is wrong as it ignores the fact that type variables with the same name must represent the same type. Consider the following example: Let `f` be a function with type `a -> a -> b`. That is, `f` takes two inputs of the same type and returns a value with a potentially different type. If we were to use the naive definition that any type matches with a type variable, type checking would allow arguments of type `Num` followed by `Bool`, which is incorrect.

Instead, we use unification [21, 13]: Given a set of constraints, a unification algorithm returns a unifier, which in our case specifies an assignment for type variables. Assume that we want to unify the set of constraints {`a -> a -> b = Num -> Num -> c`}. Applying the decomposition rule [21, 13] results in the set of constraints {`a = Num`, `b = c`}, which is a valid assignment. It follows immediately that {`a = Num`, `b = Num`, `c = Num`} is also a valid assignment. However, while the assignment {`a = Num`, `b = c`} does not assign a specific type to `b` and `c`, {`a = Num`, `b = Num`, `c = Num`} restricts `b` and `c` to be `Num`. Hence, {`a = Num`, `b = Num`, `c = Num`} is less general than {`a = Num`, `b = c`}. The algorithm described in [21, 13] always returns the most general unifier. An implementation of a unification algorithm returning the most general unifier can be found in the thesis' code repository.

Finally, we express type checking as a unification problem by unifying the constraint *type of function to be applied without output = argument signature*. We define the argument signature to be the types of the passed inputs combined using `->`. For example, a passed input of type `Num` followed by `Bool` results in the argument signature `Num -> Bool`.

Previous implementations unified a different constraint, namely *type of function to be applied = argument signature*. Note that in this case, the type of the output of the function is not removed. Additionally, the argument signature

is defined as the types of the passed inputs together with a "fresh" type variables combined using `->`. Thus, the argument signature of the previous example would be `Num -> Bool -> z0`, where `z0` is an unused type variable, instead of `Num -> Bool`. The additional type variable should match the output type of the function, resulting in successful unification. This implementation, however, is incorrect: Let `f` be a function with type signature `a -> b -> c`, and let `Num -> z0` be the inferred argument signature, where `z0` is the added type variable. We would expect unification, and therefore type checking, to fail, as the argument signature is missing the second input of type `b`. But since an unused type variable can, in general, match any other type, `z0` will match `b -> c` during unification, resulting in the system missing this type violation.

We will wrap up the discussion about type checking by detailing how the argument signature is inferred from concrete values. Due to the existence of type variables, taking the inferred type of each value is not enough. Assume that we pass a value of type `a` followed by a value of type `b` to a function of type `b -> a -> c`. In isolation, the type `b -> a -> c` does not enforce any constraints on its inputs or outputs. Thus, we would expect that unification does not introduce any constraints beyond that the input types of the function and argument signature have to match. However, applying the rule *type of function to be applied without output = argument signature* results in the constraint `a = b`, i.e. the types of the values passed as an input must be the same. This issue is caused by using the same type variables, namely `a` and `b`, across different, independent types, which can lead to problems when they are combined. In lambda calculus, this problem is known as variable capture and is solved by alpha conversion. In this case, alpha conversion works by replacing type variables with different, unused type variables, which should not change the meaning of the types [11, 7]. In particular, we can convert the type of the function to be applied from `b -> a -> c` into `d -> e -> f` by applying alpha conversion. Now, unification will not introduce any new constraints beyond the input types having to match between function and argument signature, i.e. `a = d`, `b = e`. Note that similar issues can arise if we pass multiple empty lists as inputs, as all of them might be of the type `[a]`, leading to a name collision on the type variable `a`.

After inferring the type of each value and applying alpha conversion such that no name collisions can occur, we combine these types using `->`. For example, let `t1`, `t2` and `t3` by the types of the inputs we have inferred and "alpha converted". Then, the resulting type of the argument signature is `t1 -> t2 -> t3`.

If type checking succeeds, the system will continue with computing the function application's return value. We will look at this in more detail in Section 2.6 on page 30.

2.5 Demonstration mode

To synthesize a function, we need to infer its type and synthesize instructions representing its computation. In other words, we need to infer symbolic relations between concrete values using one or more examples, which has some parallels to concolic execution. Thus, in addition to allowing branching, recursion and returning in user-defined functions, we also need a different implementation of function application. This motivates a new mode separate from interactive mode, in which function synthesis takes place: Demonstration mode.

2.5.1 Function application

We will start with implementing function application in demonstration mode, as function application lies at the heart of functional programming languages. Imagine the following scenario: the user demonstrates an example to generate a new function. In this example, the user selects one or two registers which hold the value `2` and adds them together, i.e. `2+2`. Assuming that the type signature of `+` is `Num -> Num -> Num`, type checking will succeed, since `2` is of type `Num`. The system now needs to decide on how to generalize this example: `2+2` (no generalization), `2+x`, `x+2` (one input, one constant), `x+x` (one input, no constant) and `x+y` (two inputs, no constant). Essentially, for every value that the user selects, the system needs to determine two properties: whether the value is constant ("Does the user mean the selected constant or is it just a stand-in for a variable?") and whether the value is coupled to another one ("Are two or more values always the same, i.e. are they represented by the same variable?").

There are two ways in which the decision can be made: either the user explicitly differentiates between these cases, or the system tries to infer these properties on its own.

The system could infer whether a value is constant or variable by first assuming that it is a constant and then asking the user to demonstrate another example. The system updates the property from constant to variable for every value that has changed, as a constant cannot take on different values. For instance, if the first example provided by the user is 2+2 and the second one is 2+3, then the system would determine the first value to be the constant 2 and the second value to be a variable. A similar idea can be found in the section about version space encoding in [10]. However, while we can be sure that the second value is a variable, we cannot be sure that the first value truly is a constant, as it could be possible that the first value is also a variable and it is just a coincidence that the first value was 2 among all examples. The system can only be sure if the user demonstrates all possible input values for the first value. This is unfavourable, as the number of possible input

Figure 2.6: Representation of a selected variable and constant in the GUI

values may be huge. Additionally, this also relies on the user knowing all possible input values and guaranteeing to the system that examples with all possible input values have been demonstrated, which is similar to the user explicitly telling the system whether a value is a constant or variable but with the added work of demonstrating possibly many examples. Even if we give up on the guarantee of whether a constant truly is a constant, this scheme still requires the user to demonstrate at least two examples for every function application, which is undesirable.

Instead, the system requires the user to differentiate between constant and variable values explicitly. In a graphical user interface (GUI), selecting a constant could correspond to right-clicking a value, while selecting a variable could correspond to left-clicking.

To infer whether two values are coupled, the system could follow a similar procedure as for determining whether a value is a constant or variable: if variables are always demonstrated to have the same value, then the system infers that the values are coupled, i.e. that they are the same variable. However, this procedure suffers from the same issues described above. We would need a demonstration for every possible combination of input values and the guarantee that all relevant combinations have been demonstrated. Instead, the system infers that values are coupled if and only if the user selects the same register/list/function.

Note that functions which are being applied can also be variable or constant. This is useful if the user wants to define a function like `map`. `map` takes a list `l` and a function `f` as inputs and applies `f` to every element in `l`. For example, the result of `map [True, False] not` would be `[False, True]`. If we do not allow the application of variable functions, we would not be able to define `map` in its general form. Instead, for every function `f` we would require a new function, which lifts `f` from being used on values to lists of values. Determining whether a function being applied is variable or constant is done in a similar fashion as determining whether other values are variable or constant: left-clicking the function in the GUI after pressing "Apply" implies that the function is a variable, while right-clicking implies that the function is a constant.

Let the global state be defined as the set of values accessible in both interactive and demonstration mode, thus excluding temporaries. We will describe temporaries later in this section. Whenever we pass an argument to a

function, we assume that the argument's value cannot change while executing a function and, by extension, while demonstrating an example. One way to ensure this would be to always allow modifications to the global state, but instead of selecting inputs from the global state directly and applying functions to them, the user selects values and explicitly copies them as separate inputs to the function, where the user cannot modify them anymore. However, this method creates more work for the user (e.g. explicitly declaring values as inputs and copying them), while the resulting advantages are unclear. Instead, suppose a value from the global state is used as an input for a function that is being generated. In that case, we do not allow modifications to that value before finishing function synthesis. Conversely, if a value from the global state is not an input to the function that is being generated, it can be modified (including deletion) without restriction. In particular, the user can always create new registers and lists.

It is worth noting that modifications to the global state are not recorded during a demonstration. For example, if the user deletes a list in the global state while creating a function, using the function after it has been synthesized will never delete anything in the global state. This is because these changes are invisible to the function, as it cannot "see" the global state, reinforcing the idea that the generated functions are side-effect free.

Internally, the system updates two maps whenever the user applies a function: one map is responsible for mapping registers/lists/functions to input names like `in0`, which keeps track of coupled values. In particular, this map determines which register/list/function represents which input. Another map is responsible for mapping names of constants like `const0` to actual values so that instructions can refer to all values by a name.

At this point, given a function application demonstrated by the user, the system knows both the concrete values of the inputs and the function being applied, as well as whether they are variables or constants and thus their internal names. As a next step, the system stores an instruction of the form `(temp_name, [f x0 x1 ..])`, where `temp_name` is a unique name for the temporary representing the result of the function application, `f` is the name of the function being applied, and `x0, x1, ..` are the names of the inputs to the function being applied. Additionally, the system computes, stores, and displays the actual value of the temporary such that the user can continue to use concrete values during a demonstration.

Since functions require a type, the system must also infer a type for user-defined functions. One approach is to let the inputs inferred from the demonstration define the function's type signature. That is, if during a demonstration the function takes two numbers as input and returns a number, the system infers that the function type is `Num -> Num -> Num`. While this works well if values are only stored in registers, this approach breaks down

as soon as we introduce lists.

Assume that the user wants to define a function that returns the first two elements of a list as a new list. To do this, the user would demonstrate an example on a concrete list with concrete values, for example, a list of numbers. If the input types define the type signature, the new function would expect a list of numbers, even though the operation could also be applied to a list of booleans. This would mean that the user has to define a new function by demonstrating the same example on a boolean list. While defining two functions might not seem bad, it does not consider more complicated ones.

Assume that the user wants to synthesize the function `map` which has the type signature `[a] -> (a -> b) -> [b]`. As both `a` and `b` represent types of values that can be stored in lists, they must either be `Num` or `Bool`. Thus, to define `map` for every possible type, the user would need to synthesize 2 * 2 = 4 different functions. In general, the number of functions can be determined by

$$\prod_{\text{x : x is a type variable}} \text{number of possible types for x}$$

Instead of letting concrete inputs define the function's type signature, the system tries to infer the most general function signature which still encodes all relevant constraints. More specifically, in our example of returning the first two elements of a list as a new list, the system should infer the type signature `[a] -> [a]`. Note that the type signature `a -> b` would be more general but does not encode any useful information: `a -> b` implies that the input and output could be anything, while we know that the input and output are lists of the same type.

A key observation is that the type signature of a function is only used during function application to make sure that a function is applied to values with types that make sense. That is, type checking prevents us from calling `+`, which adds two numbers together, on, for example, a number and a list. In other words, the type of `+` (namely `Num -> Num -> Num`) enforces the constraint that both of its inputs have to be numbers. Thus, whenever the system infers the type signature of a function, it must ensure that it enforces all necessary constraints for any operation within that function. In other words, if the type check for applying a function succeeds, all type checks that happen "within" that function must also succeed. We also observe that the output type of a function should be sound. That is, if a function returns a number, its output type should be `Num` and not `Bool`. With these observations, we can devise a strategy to infer the most general function type.

Firstly, we implement a mapping from names like `inX`, `constX`, `tempX` to abstract types, where the abstract type is the type used to infer the type signature of the function to be synthesized. We refer to the actual type of a

value as its concrete type. The abstract type of inputs and temporaries is a new type variable. In contrast, the abstract type of constants is the concrete type after applying alpha conversion, such that type variables cannot be captured accidentally, as explained before.

Secondly, we collect constraints. For example, if type checking for a function application during demonstration succeeds, the system stores the constraint *abstract type of applied function = abstract types of arguments* `->` *abstract type of assigned temporary*. In general, if not stated otherwise, we combine multiple types using `->`. For example, `a` and `b` would be combined into `a -> b`. Here, we apply this procedure to combine the abstract types of the arguments. We also keep track of the constraints introduced by returning a value (see Section 2.5.3 on page 24). Finally, after collecting all constraints generated by function applications and returning a value, we add the constraint *w_sig = abstract types of inputs* `->` *w_out* and run the unification algorithm. If unification succeeds, `w_sig` is equal to the inferred type signature. While we do not formally prove that this type signature is the most general one respecting all constraints, these properties should follow from using a unification algorithm that returns the most general unifier, assuming we have collected all constraints.

However, collecting all constraints before running the unification algorithm might be too late in some cases. Assume that a function has multiple return points, for example, due to branching. At the first return point, a number is returned, while at the second return point, a boolean is returned. This, however, is a contradiction, as the return type of a function cannot simultaneously be a boolean and a number, leading to the failure of the unification algorithm. Note that contradictions are not restricted to return types but are also possible on input types. For example, assume that we increment an input by 1 in one branch and apply `not` to the same input in another branch. This would imply that the input is simultaneously a boolean and a number – a contradiction. If unification is delayed until the end, any work done by the user starting from the point of generating a contradiction is wasted. Instead, we could improve the system by running the unification algorithm every time a constraint is added. In case unification fails, changes are rolled back. This should result in less wasted effort for the user.

2.5.2 Branching

At this point, the system is capable of building functions using function composition. To properly support lists, we will need to add branching support since an operation like returning the first two elements of a list as a new list needs to check whether the passed list contains zero, one or more elements and adapt its behaviour appropriately. Branching is also required

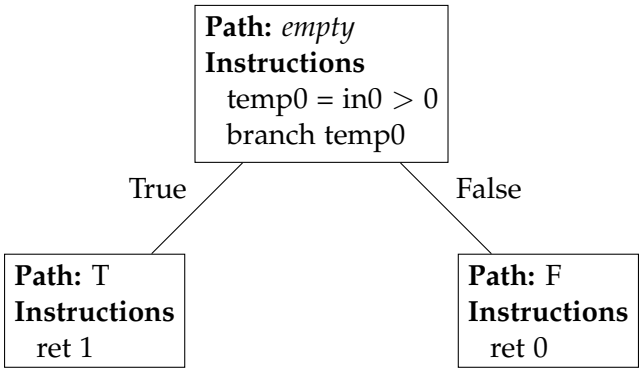

Figure 2.7: Simplified representation of the tree generated during function synthesis

for recursion support, as we need a way to differentiate between the base case and the "recursive step" [22] of the recursion.

We start with an example: Assume that the user wants to define a function which returns different numbers depending on the value of its input. If the input x is greater than 0, i.e. the condition $x > 0$ is true, the function should return 1. Otherwise, the function should return 0. If the system is only shown one branch, for example, where the condition is true, it has no way of knowing what to synthesize for the branch where the condition does not hold. Thus, the user will need to provide two examples in total: one example where the input is larger than 0 and the function returns 1 after checking the condition, and another example where the input is less than or equal to 0 and the function returns 0 after checking the condition. Note that this implies that the input's value has to change between examples (e.g. from 3 to -2 or vice versa). This motivates a third and final mode, called between mode, which we will explain later.

We keep track of these examples and their inferred instructions using a tree. In this case, the root of the tree contains the list of instructions required for comparing the input to 0 and branching based on that result. The children of the root are leaves with lists of instructions to return either 0 or 1.

Each node of the tree contains a list of instructions, which includes precisely one control flow instruction placed at the end. The list of instructions is always either fully executed or not at all. This definition of a node is similar to the notion of basic blocks within the control flow graph of a program. For non-leaf nodes, the final control flow instruction must be `branch cnd_name`, where `cnd_name` is the name for a boolean value which is not a constant; for example, the name of an input or temporary. Constants are not allowed as branch conditions, as the system expects each branch to require a new example where the value of `cnd_name` has changed. However, a constant is, by definition, either always true or always false – it cannot

change. Hence, branching on a constant does not make sense, and it is the user's responsibility to avoid it. For leaf nodes, the final control flow instruction must be `ret ret_name`, where `ret_name` is the name of the value to be returned, for example, the name of an input, temporary or constant. All leaf nodes must end with a return because a function must always return a value if it terminates. We will explain returning in more detail in Section 2.5.3 on the next page. In addition to a list of instructions and possible references to child nodes, nodes also contain a representation of the path from the root to themselves. Finally, we keep track of the position of the next instruction to be added to the tree.

The demonstration of the previously described function might go like this: the user selects a register containing 3 and a register containing 0 and applies $>$, adding a new instruction at the next position in the tree. Then, the user selects the resulting boolean and, for example, presses a button labelled "Branch". This leads to the creation of a new node, which is a child of the root, and the appropriate update to the position of the next instruction. Since 3 is strictly greater than 0, the user will select the number 1 and return it, which adds a return instruction to the tree. In a GUI, returning can be achieved by pressing the "Return" button. At this point, the system will inform the user that another example where the first condition is false is required. The system switches into between mode, allowing modifications to the state with some restrictions. When the user is done modifying the state by changing the value of the register holding 3 to -1, they signal to the system that they want to continue by pressing another button. Following that, the system resets the position of the next instruction to be the root and expects the user to demonstrate another example, such that the newly inferred instructions match with previously inferred instructions. In other words, the system expects the user to demonstrate the comparison followed by branching again. As soon as the user branches to an unseen branch, i.e. when the user branches on the result of $-1 > 0$ (false), there are no more expected instructions, and the user may freely continue their example. In this case, they would select the number 0 and return it.

Note that this procedure forces the user to demonstrate parts that stay the same between different examples multiple times. This is by design since it forces the user to think through the entire example again. If, instead, the system automatically determined the state following the expected instructions without user input, the user may not be able to follow the steps executed by the system, which may make it hard or impossible to continue the demonstration of the new part of an example. Nonetheless, there are ways in which this process can be made easier for the user: instead of forcing the user to demonstrate the static part of an example again, the system could display the expected instruction, and the user would only need to confirm it as the next instruction. How the expected instruction is presented to the

user is essential: simply showing the instruction how it is stored internally may not be intuitive to the user and partially defeats the purpose of getting familiar with the concepts of functional programming without learning the syntax of a programming language. Instead, a (small) computational graph could represent the expected instruction.

In addition to using computational graphs to represent expected instructions intuitively, we can also use them to visualize how the value of a given temporary is obtained. For example, when the user hovers over a temporary in the GUI, the system could display an appropriate computational graph.

2.5.3 Returning

On a high level, returning is straightforward: the user selects the value they want to return and returns it by pressing a button labelled "Return" in the GUI. As briefly mentioned before, this leads to the instruction `ret ret_name` being added to the tree, resulting in a leaf node. Additionally, the constraint $ret\_type = w\_out$ is added to the constraints, where `ret_type` is the abstract type of the return value and `w_out` is the output type of the function which is being inferred.

The subsequent steps depend on whether the system expects more examples from the user: if no more examples are expected, the function is synthesized by saving the tree of instructions and inferring the function's type signature using unification, as described in Section 2.5.1 on page 17, and the system switches back to interactive mode, where the user can use the newly synthesized function. In case more examples are expected, the system prepares the state for another example and switches into between mode, where the user can modify the state similar to interactive mode, but with some exceptions. We will postpone the discussion about between mode to later in this section.

To determine whether the user has to demonstrate another example, we make the following observation: whenever a node has exactly one child, the user has provided exactly one example where the branching condition is either true or false. Hence, another example is required where the branching condition is different. Otherwise, the node is a leaf, or both branches are already known. We use this observation in Program Code 4 on the next page, which returns the remaining examples that the system requires.

Assuming that more examples are required, the system needs to prepare the state for the demonstration of a new example. Firstly, the "pointer" to the position of the next instruction is set to the first instruction in the root node. Secondly, we clear the mapping from temporary names to their associated expressions and values, since at the beginning of an example, no instruction has been "executed" yet and hence no temporary should be available. However, we need to be careful that we do not reuse temporary

```
Path = list[str]

def remaining_examples(self) -> list[Path]:
    """
    Returns a list of paths, which represents the (additional)
↪   examples required for function synthesis

    Examples
    --------
    [["T", "F"], ["F", "F"]] means that we will need two
↪   examples:
    One where the first condition is True and the second False,
↪   and another one where both conditions are false.
    """

    true_empty = self._true is None
    false_empty = self._false is None

    if not true_empty and not false_empty:  # Both are not None
    ↪   => Go deeper
        return self._true.remaining_examples() +
        ↪   self._false.remaining_examples()
    elif true_empty and not false_empty:
        return [self._path + ["T"]] +
        ↪   self._false.remaining_examples()
    elif not true_empty and false_empty:
        return self._true.remaining_examples() + [self._path +
        ↪   ["F"]]
    elif true_empty and false_empty:  # Both are None => There
    ↪   is no branching
        return []
```

Program Code 4: Algorithm to determine the required examples (Python)

names for new, "unexpected" instructions. Otherwise, we may end up in an invalid state where the abstract type of a temporary is falsely captured by both old and new (unrelated) constraints. For example, assume that `temp0` has the abstract type `a0` and the constraint `a0 = Num` was introduced by a previous example. If the system reuses `temp0` and introduces the constraint `a0 = Bool`, unification will fail even if the different constraints for `temp0` are never forced to hold simultaneously. This sitation could occur if, for example, `temp0` is defined in two separate branches. On the other hand, if the system always generates new temporaries, inferred instructions would always differ from expected instructions in their temporary name. Hence, the system needs to deal with these scenarios by keeping track of which names have already been used and whether the user is demonstrating old and expected or new instructions.

In contrast to temporaries, we do not clear the set of constraints or the mapping from names to abstract types, as they are required for unification to infer the type signature of the synthesized function. Additionally, we do not clear the mapping from registers/lists/functions to input names such that we can keep track of inferred inputs of the function across multiple examples.

Generally, a function is expected to be applied to all of its expected arguments, regardless of whether they are used. Even if the user knows that some arguments are never used for a certain set of concrete input values, they cannot just drop those "unnecessary" arguments. However, the user might only use a subset of the final set of arguments while demonstrating a concrete example. For example, assume that the user is defining a conditional map `cmap b f g l`, which takes a boolean `b`, two functions `f` and `g`, and a list `l`. Depending on the value of `b`, `cmap` applies either `f` or `g` to `l`. A demonstration of this function would involve two examples: one, where after branching on `b` the user returns `map f l`, and another one where after branching on `b` the user returns `map g l`. In the first example the user only uses the inputs `b`, `f` and `l`, while in the second example the user only uses the inputs `b`, `g` and `l`. Note that the user never uses `f` and `g` simultaneously. As long as no example uses all the inputs of the final function, it is not enough to consider examples in isolation to infer the final set of arguments.

A naive approach would be to force the user to always use all inputs in an example by introducing dummy instructions. However, this would be inconvenient for the user. Instead, the system remembers the input registers/lists/functions from the previous example. If the user selects registers/lists/functions that have acted as an input in previous examples, the system infers that the user is selecting the same inputs again. A "new" register/list/function is inferred to be a new input when used. This is essentially the same mechanism used to determine coupled values within an example. However, this approach also has issues. Assume that the user wants to create

a function that determines whether a number is even and returns different values based on this information. First, the user selects a register and checks whether the value it holds is even and branches based on the result. In the second example, the user needs to select the same register again; otherwise, the system infers a new input and thus an instruction that is slightly different from the expected one. Because the notion of input is not exposed to the user, they may not know that they must select the same register again if they want to select the same input. It is also possible that the user forgets the mapping from registers/lists/functions to inputs at some point. We can handle this problem by exposing this mapping to the user by adding a small note telling the user whether a value is the first, second, etc., input under the values in question.

Between the demonstration of examples, the user may want to change the value of an input. For instance, if the user demonstrates examples to synthesize a conditional map, they must change the input boolean from True to False (or vice versa) between examples. However, because the system remembers inputs across different examples, the user may not modify the state in arbitrary ways. This motivates a third mode called between mode, as mentioned before.

Between mode is similar to interactive mode with restrictions. For example, the user may create any value except a new function in between mode. This is because creating a function can have two interpretations: creating a function independent of the current demonstration, or creating a nested function, i.e. a function definition within a function definition. If the user wants to create a function independent from the current demonstration, it makes more sense for the user first to abort the current synthesis. While the current system implementation does not support aborting, adding this functionality is straightforward. Since we already generate snapshots for every valid action the user takes, we only need to "mark" the snapshot directly before function synthesis, add a button to abort function synthesis and restore the snapshot in case the abort button is pressed. Nested functions are out of the scope of this thesis.

The only other restriction is that deleting registers/lists/functions which are being used is not allowed. To understand why, recall that we use under which name a value is stored in the state to determine the corresponding input. If the user were to delete this value, the associated name would also be removed from the system. Hence, there would be no way to access the associated input anymore. While it is true that this is not a problem if that input is not going to be used anymore, it is an opportunity for the user to "shoot themselves in the foot" without offering any significant benefit. Deleting elements from a list in use is possible, as we always select an entire list and not list elements. Updating and selecting values and applying functions are

```haskell
data MyList a = EmptyList | Cons a (MyList a)
```

Program Code 5: Custom list definition in Haskell using algebraic data types [15]

```c
int main() {
    int acc = 0;
    for (int i = 0; i < 10; i++) {
        acc += 1;
    }
    return acc;
}
```

Program Code 6: Adding 1 to acc ten times in C using a loop

analogous to the interactive case.

2.5.4 Recursion

When we say that a function is recursive, we mean that in the definition of the function, the function applies itself to some arguments. Usually, these arguments represent a simpler instance of the problem. For example, summing up the elements in a list containing three numbers is the same as adding the value of the first element to the sum of the list without its first element, which contains two elements. To ensure that the function does not call itself infinitely often or runs into issues like taking an element from an empty list, the definition of a recursive function contains a base case where it does not recursively apply itself. In our example of summing up the elements in a list, the base case would be returning 0 for the sum of elements inside an empty list.

Recursive functions are incredibly useful for data structures that we can define recursively. Recursive data structures allow us to combine instances of a data structure, possibly together with other elements, into a new instance of the same data structure. For example, we can define a list as an empty list or an element added to the beginning of a list. One can imagine the elements as layers around a core represented by the empty list. In Haskell, the definition of this data structure could look like in Program Code 5.

Beyond processing lists, recursion is capable of simulating loops which are available in imperative programming languages (compare Program Code 6 and Program Code 7 on the facing page).

In Program Code 7 on the next page, we can understand `n` as the number of times we want to execute the body, `i` as how often we have executed the loop and `acc` as keeping track of the state of the loop body, such that we can update it in the next "iteration". We can employ the same strategy we

```
increment n i acc
    | i < n = increment n (i+1) (acc+1)
    | i >= n = acc

increment 10 0 0
```

Program Code 7: "Adding" 1 to acc ten times in Haskell using recursion

```
data MyNat = Zero | PlusOne MyNat
```

Program Code 8: Custom natural number definition in Haskell using algebraic data types

used to define lists as a recursive data structure to define a recursive data structure representing natural numbers (Program Code 8).

In a GUI, the user can recursively apply a function they are synthesizing by selecting the arguments and then pressing a special button called "Recurse", resulting in the instruction `self in0 in1 ...`. However, applying an incomplete function to itself poses some challenges. For one, it might not be possible to generate the result as the function is incomplete. This can occur, for example, if the recursive step is demonstrated before the base case. In general, the system attempts to calculate the value. If it gets stuck, i.e. the instruction stream had ended before a return statement was reached, no value is returned (represented by `None` in Python).

The other issue is that an incomplete function, including its function signature, may change while demonstrating an example. For example, assume that the current number of arguments is inferred to be three. The user then recursively applies the function to three arguments. After this recursive application, the user may introduce a new input. Thus, the function signature demands four arguments instead of three. The system must handle this case for every recursive call in the final instruction tree. Otherwise, recursive calls may be missing arguments, leading to type errors.

As already discussed in the section about returning, a naive and inconvenient approach would be to force the user always to select all currently known and future inputs in an example. Instead, we observe that new inputs are always added at the end of the function signature. We will illustrate the resulting procedure with an example. Suppose a function has three inputs at some point. In that case, a recursive call at that point must have also received three inputs, resulting in the instruction `self x y z`, where `x`, `y` and `z` are the names representing the arguments to the recursive call. When a new input, for example, `inX`, is added, we can retroactively add it to the recursive call `self x y z` by changing the instruction to `self x y z inX`. Note that we do not need to add a type constraint, as the name and thus the type of the input to the function being synthesized and the input to the recursive

call are by definition the same, resulting in the trivial constraint *type of inX* = *type of inX*. This approach only requires the user to pass all currently known inputs to a recursive call instead of all currently known and future inputs.

We will now explore whether it is possible to weaken this assumption even further, i.e. whether it is possible for the user to only pass a subset of the currently known inputs. For this purpose we return to the example of the conditional map `cmap b xs f g`, where `b` is of type `Bool`, `xs` is of type `[a]` and `f` and `g` are functions of type `a -> b`. In a demonstration, the user would first branch based on the value of `b` and then demonstrate the standard map implementation. Assume that `b` is True. If `xs` is non-empty, the user takes the first element from `xs`, applies `f` to it and prepends the result to `self b (tail xs) f`. Note that in this first example, the user does not use `g`. Since the user branches on the value of `b`, they will need to demonstrate another example for when `b` is false. The demonstration will be very similar to the demonstration for the branch where `b` is True, as the only thing that changes is the function to be applied. However, if the user only applies the function recursively to the inputs `b`, `(tail xs)` and `g`, the system faces the following problem: the function expects inputs `b`, `xs`, `f` and `g`, but only `b`, `xs` and `g` were passed. This is different from the case where we assume that users always pass all known inputs to recursive calls since, in that case, the system knows that a missing input is the last input to a function. However, in this case, the system is missing the second to last argument `f`. This raises the question of whether the system should infer `self b (tail xs) f g` or `self b (tail xs) g f` as the recursive call. While in this example, it would be fair to assume that the user meant `self b (tail xs) f g`, it is possible that the user meant `self b (tail xs) g f`, as it is also a valid function application. Thus, it is not possible to only pass a subset of arguments and expect the system to be able to infer a unique recursive function application.

When we recursively apply a function, the abstract types of the currently known inputs of the function need to match the types of the arguments passed to the recursive call, while the abstract type of the temporary in which the result is stored needs to match the type of the output of the function (`w_out`). We can ensure this by adding the following constraint: *abstract types of currently known inputs (combined with `->`) `->` `w_out` = argument signature `->` abstract type of temporary in which the result is stored*.

2.6 Computing results

To prepare the computation of the result of a function application, the system first builds the initial context, which includes checking whether the received input types match the expected ones and adding a mapping from the names of inputs and constants to concrete values. Both built-in and user-defined

```
def compute(self, args: list[Value]) -> Value:
    context: dict[str, Value] = self.input_context(args)

    match self.builtin:  # Name of the built-in function
        case "+":
            return context["in0"] + context["in1"]
        ...
```

Program Code 9: Implementation of built-in functions

Table 2.1: Overview of instructions generated by the system

Description	Instruction	Python representation
Apply function `f` is to `x`, `y` and `z`, and bind the result to `temp0`	`temp0 = f x y z`	`("temp0", ["f", "x", "y", "z"])`
Branch on boolean `b`	`branch b`	`(None, ["branch", "b"])`
Return value of `temp0`	`ret temp0`	`(None, ["ret", "temp0"])`
Recursively apply function to `x`, `y` and `z`, and bind the result to `temp1`	`temp1 = self x y z`	`("temp1", ["self", "x", "y", "z"])`

functions use this initial context to compute the final result.

2.6.1 Built-in functions

Built-in functions use the initial context to get the value of the inputs and directly compute the result using a corresponding Python implementation (Program Code 9).

2.6.2 User-defined functions

User-defined functions extend the initial context by executing instructions generated during function synthesis until a return instruction is reached, which returns the value of a name in the context. For an overview of the supported instructions and the implementation for executing those instructions, see Table 2.1 and Program Code 10, respectively.

```
def compute(self, args: list[Value]) -> Value:
    context: dict[str, Value] = self.initial_context(args)

    # Keep track of instruction to be executed
    block_counter: int = 0
    current_node: Tree = self._instructions

    while True:
        tmp_name, expr =
        ↪  current_node.get_instruction(block_counter)

        if expr[0] == "ret":
            return context[expr[1]]

        elif expr[0] == "branch":
            cnd_val = context[expr[1]]
            current_node = current_node.get_true() if cnd_val
            ↪  else current_node.get_false()
            block_counter = 0

        elif expr[0] == "self":
            f_args = [context[arg] for arg in expr[1:]]
            context[tmp_name] = self.compute(f_args)

            block_counter += 1

        elif expr[0] in context:
            f = context[expr[0]]
            f_args = [context[arg] for arg in expr[1:]]

            context[tmp_name] = f.compute(f_args)

            block_counter += 1
```

Program Code 10: Implementation of ”interpreter” for user-defined functions

Chapter 3

Evaluation

We conducted a small-scale qualitative study with two participants to evaluate our implementation. Each participant answered a set of questions and solved tasks using the developed prototype. Even with only two participants, we cover a broad spectrum of experience in programming, in particular, functional programming: while one of the participants is a computer science student in their sixth semester and has taken courses with significant functional programming content, the other participant is an electrical engineering student with no functional programming experience.

We summarise the findings and propose improvements and future research directions that might be interesting to explore. We invite the reader to take a look at Appendix A on page 37 for more details about the study.

Most, if not all, issues are regarding the graphical user interface, which was not the focus of this work. Some of the observed problems have straightforward solutions. For example, participants tried to create registers containing strings, which is not supported by the system. Informing the user while they create a register that only numbers and booleans are supported can resolve this issue. Many problems stem from this lack of "documentation": for example, the participants did not expect labels to be clickable. Additionally, they did not initially understand how to synthesize functions using the programming-by-demonstration paradigm.

Another problem was that the participants sometimes did not know what the built-in functions "do" or how to use them, i.e. which arguments are required and in which order. Hence, we think that a tutorial introducing users to the tool is essential to make it useful. The idea of tutorials was also suggested by one of the participants. A tutorial should include general information about the system like what the programming-by-demonstration paradigm is and specific information regarding the GUI, e.g. how to select values as constants and variables, how to apply a function to selected values, the meaning of the

buttons and so on.

Additionally, a significant amount of issues are caused by the lack of information during function synthesis. The lack of information during function synthesis differs from the lack of "documentation" mentioned above. Before, we meant a lack of general information, i.e. information that is not specific to a demonstration. In this case, we mean a lack of information regarding specific demonstrations. Participants often got lost on what they needed to do, especially if the program to be synthesized contained branches. One participant, for example, tried to demonstrate the second branch of a program using a register different from the one used in the first example. However, either this won't work because the system expects the user to use the previously used register, or it will lead to the system synthesizing a function with an additional input. We can alleviate this problem by explicitly exposing the internal mapping between registers/lists/functions to inputs to the user. One straightforward solution would be adding text like "input 0" to the register/list/function which is considered to be the first input.

Another point which led to confusion was the meaning of generated temporaries. Sometimes, the participants forgot what computation the temporaries represented, as currently they more or less directly expose the internally stored instruction, for example, `temp0 = const1 in0 const0`. A fix to this problem would be to replace constants and inputs with their actual values and differentiate between them using different formatting: constants would not be formatted, while inputs could be formatted using italics. Another solution would be to visualize the computation using a computational graph. In fact, generating a computational graph for the entire example or program might be helpful.

In addition to the inputs not being clear to the participants, branching often posed a significant challenge to them as well: in particular, they did not know that they had to repeat previously demonstrated "instructions" when demonstrating a new example. Even when they knew this, they sometimes got lost and did not know what instruction was expected next. Finally, at least one participant was occasionally unsure whether the "Branch" button worked.

To make branching easier for the user, the system could show the expected instruction to the user and ask for confirmation to demonstrate it automatically, instead of forcing the users to demonstrate it themselves. This way, the burden on the user is reduced. The displayed expected instructions must be easy to understand, which can be achieved by presenting a textual description to the user instead of the internal representation. More specifically, instead of the system showing `branch temp0` to the user, it may show "Branch on the condition temp0 (currently True)" to the user instead. Another way to help the user in these cases could be to show them the complete program

tree: recall that the system generates a binary tree representing the program and keeps track of its current position in it. To expose this information to the user, we could replace the instructions stored inside the tree with a textual description as described just now, remove any unnecessary information like the paths to the nodes and visualize the current position using an arrow pointing at the correct location in the tree.

Beyond these issues, there are other things about the GUI that we can improve. Firstly, both participants had a lot of problems with function application. The process of selecting inputs first, pressing "Apply" and then selecting the function to be applied was perceived as unintuitive. Additionally, the GUI does not support aborting function application or synthesis, which forces the user to press the undo button one or more times instead of aborting with a single button press. One may even argue that this could be considered a syntax barrier. Nonetheless, even if it were a syntax barrier, it is probably less significant than the syntax barrier of "industry-standard" programming languages. Suppose the GUI were more "interactive" instead of filled with buttons. In that case, we could alleviate these issues by replacing function application with a drag-and-drop procedure: the user applies a function by dragging and dropping the inputs to a node representing the function to be applied.

Secondly, some events are easily missed in the GUI: for example, after the user returns from an example, the system informs them about the necessary examples in a prompt in the top left corner of the window. The participants often missed this prompt. Similarly, one participant pressed the "+" button for user-defined functions multiple times, as they did not realize that they had already changed into demonstration mode. One way to alleviate these issues would be to, for example, use pop-up windows or different colours. However, the specifics heavily depend on what the final implementation should look like.

Finally, we think that there is more that we can do in regards to implementing the second design point of Algot ("operations of the program share the same syntactic and semantic meaning [whenever appropriate]" [20] (p. 3)) in this system. For example, instead of selecting a list and then applying `head` and `tail` to it, we think that it would be a lot more intuitive to allow users to use "scissors": the user selects the "scissors" from the user interface, and "cuts" the list into two parts.

In general, we think that combining programming-by-demonstration and visual programming can be an effective tool for helping students better understand functional programming. In the study, the participants shared this opinion. The main issues exposed in this study are regarding the graphical user interface, which was not the focus of the thesis. While we have proposed solutions to many of the observed problems, we think more

research should be done, including more rigorous user studies. This opinion is shared by one of the participants.

Beyond improving the GUI, it might be interesting to make the system more powerful by allowing the definition of user-defined data types like algebraic data types. We leave the exploration of this idea to future work.

Appendix A

Appendix

A.1 Study procedure

Pre-Study Interview

- Do you have programming experience? If yes, please elaborate (Which programming languages are you familiar with? For how long? How would you rate your proficiency [in general and with specific languages]?).
- Do you have experience with functional programming? If yes, please elaborate (Which programming languages are you familiar with? For how long? How would you rate your proficiency [in general and with specific languages]?).
- Do you have a background in Computer Science? If yes, please elaborate.

Study

1. We shortly explain what the system is: a visual programming environment where the user can synthesize functions by demonstrating examples.
2. Give the user time to play around with the programming environment.
3. Ask the user about their expectations regarding the system, particularly what they think the system needs from them and what the most complicated function is that the system can generate.
4. Tell the user to define a function that takes the average of two numbers. [Simple function composition]

5. Tell the user to define a function that returns 1 if a number is even and 0 otherwise. [Conditionals]
6. Tell the user to define a function that calculates the n-th Fibonacci number (f(0) = 0, f(1) = 1, f(n) = f(n-1) + f(n-2) [14]) [Recursion]
7. Tell the user to define map for a general function [Recursion, Lists, Function as input]

During the study, the user is observed. We take note of any questions they have and the answers they are given. Additionally, if the user seems to be stuck, we actively ask them whether/where they have a problem.

Notes: The explanation in the beginning is kept short on purpose. This way, it should be easier to identify parts of the system that are unintuitive.

Post-Study Interview

- How did you solve the tasks? Did you have a strategy?
- Do you feel like this helped you get a better understanding of programming, in particular functional programming? Why (not)?
- Do you think that such an implementation can be used to help students get a better understanding of programming, in particular functional programming? Why (not)?
- Regardless of the implementation you interacted with just now, do you think that the approach of visual programming + programming-by-demonstration is a good idea in the first place? Why (not)?
- In general, what do you think are the strengths of this implementation (if any)?
- In general, what do you think are the weaknesses of this implementation?
- Ask for any other comments

A.2 Study Results A

Length of session: around 2 to 2.5 hours

In general, red indicates that the text was generated by the participant, while blue indicates that the text was generated by us.

Pre-Study Interview

- Do you have programming experience? If yes, please elaborate (Which programming languages are you familiar with? For how long? How would you rate your proficiency [in general and with specific languages]?).
  - Rough understanding of Python (1 yr)
  - More experienced in C++ low-level applications (1.5 yrs)
  - Simulink and Matlab, general controllers and mechatronics applications (0.5 yrs)
- Do you have experience with functional programming? If yes, please elaborate (Which programming languages are you familiar with? For how long? How would you rate your proficiency [in general and with specific languages]?).

  None
- Do you have a background in Computer Science? If yes, please elaborate.
  - Electrical engineer
  - Experience in robotics in the fields mechatronics and controls as well as computer vision

Study

1. We shortly explain what the system is: a visual programming environment where the user can synthesize functions by demonstrating examples.
2. Give the user time to play around with the programming environment.
   a. Participant tried to create a string in register (only numbers and booleans are supported)
   b. Participant tried to add true as register instead of True
   c. Participant pressed + button for user-defined functions multiple times

   d. Participant struggles to figure out how to do function application
   e. Participant struggled to figure out how to do selection (Did not know that labels are clickable)
   f. Participant tried to select values directly from a list
   g. Participant got stuck in "Select a function to be applied."
   h. Participant tried to add more than two integers together at the same time
   i. Participant is unsure what some of the built-in functions do (in particular those which are more common in functional programming languages like Haskell, e.g., cons, map, filter)

3. Ask the user about their expectations regarding the system, in particular, what they think the system needs from them and what the most complicated function is that the system can generate.

   It needs to know the exact inputs and order of operations the user wants to have. With the simple elements, I think the user is able to even create complexer programs, if executed correctly

4. Tell the user to define a function that takes the average of two numbers. [Simple function composition]

   a. Participant forgot to press the + button under user-defined functions before demonstrating an example
   b. Reminded the participant that they could define new functions by demonstrating examples (similar to what they did in "interactive" mode)
   c. Participant tried to define all operations at the same time (e.g., selected 4, +, 4, /, 2) - told them to do the demonstration step by step
   d. Told the user about the difference between selecting something as constant or variable (including functions to be applied) and how to do it in the GUI
   e. Trying out the newly synthesized function resulted in a unification error. After having the participant undo steps step-by-step, we found out that they accidentally selected 2 as a variable instead of a constant.
   f. Participant got stuck in "Select a function to be applied." again

5. Tell the user to define a function that returns 1 if a number is even and 0 otherwise. [Conditionals]

   a. Participant forgot to press the + button under user-defined functions before demonstrating an example again

   b. Explained to the participant how "Branch" works: selecting a conditional branch and then branching tells the program that everything that follows depends on the selected condition

   c. Participant was not sure whether the branch actually worked (i.e. whether the "if" got generated)

   d. Participant did not understand the prompt "Prepare state for a path from `[['T']]` and press continue." – Explained using a tree: branching generates a fork in a tree, and one example corresponds to one path through the tree. Thus, to fully define the function, another example needs to be shown for the other path through the tree generated by the fork.

   e. Participant tried to demonstrate the other path/example using a different (input) register – Explained that they need to use the same register, as this is what the system assumes.

   f. After these explanations, they did not struggle with demonstrating the second branch, including that they had to start the examples "from the top", i.e. that they also need to demonstrate previously demonstrated parts in this task at least.

6. Tell the user to define a function that calculates the n-th Fibonacci number (f(0) = 0, f(1) = 1, f(n) = f(n-1) + f(n-2) [14]) [Recursion]

   a. Participant forgot to press the + button under user-defined functions before demonstrating an example again

   b. Participant forgot to press "Continue" after finishing setting up the state for the next example

   c. Participant forgot to check whether the input is equal to 0 (what they did in the previous example) before checking whether the input is equal to 1.

   d. Participant started a new attempt

   e. Participant forgot to check whether the input is less than 1 (what they did in the previous example) before checking whether the input is equal to 1.

   f. Meaning of the prompt "Prepare state for a path from ..." was unclear to the participant

   g. Participant asked whether they had to demonstrate previously demonstrated conditions/branches in new examples and, if yes, in

which order they had to be demonstrated. – Answered that previously demonstrated conditions/branches need to be demonstrated again in the same order.

h. When it came to recursion, we told the participant to not get stuck thinking about how it can be done with the system, but rather that they should think about how they would do it on a high level

i. Participant asked whether they could define a function with a function – Answered with no.

j. Participant got stuck when f(0) = 0 and f(1) = 1, since at that point they just added 0 and 1 together directly (i.e., they demonstrated (n-1) + (n-2)) instead of using recursion – Told them to explicitly recursively apply the function to n-1 and n-2

k. Participant got stuck in "Select a function to be applied." again

7. Tell the user to define map for a general function [Recursion, Lists, Function as input]

   a. Told the participant to really think about it on a high level using recursion

   b. Participant came up with the idea to pop the first element and apply function

   c. After the participant got stuck, we explained the idea of how to solve this task recursively: apply the function to the first element of the list, recursively apply the function to the list without its first element, and combine the results by adding the first element on which the function has been applied, to the beginning of the result of the recursion. The reason they got stuck is that they were not familiar with "map" or thinking "functionally/recursively".

   d. Participant mentioned (while) loops and wondered how they would work – We did not go into details as a lot of time had passed already, and this way of thinking might be considered more advanced/suited to imperative programming

   e. Demonstrated the program together, as they struggled with the "recursive algorithm" for defining map and did not have functions like tail and cons in mind

   f. Participant selected the function to be applied as a constant instead of a variable since they did not know that map can apply arbitrary functions as long as the types match

   g. Participant was confused about `None` as a value resulting from function application – We explained to them that the system could

not compute the result sometimes as it cannot continue the recursion (e.g. due to a missing base case)

h. Participant forgot that they had to check whether a list was not empty, which is what they did in the previous example

i. Participant forgot to select a function when applying map during interactive mode (after synthesizing it by demonstration)

Post-Study Interview

- How did you solve the tasks? Did you have a strategy?

  Since I mostly worked in oop, I tried working the same way, using if and else to get around the different cases each function can reach

- Do you feel like this helped you get a better understanding of programming, in particular functional programming? Why (not)?

  Definitely. As a person who has not worked with functional programming yet, it was a good introduction into the general concept, without having to learn a specific programming language. I personally did not have any experience in functional programming prior, and was able to understand the basic concepts within only a few hours with the explanations of Mr Nezamabadi. The program might need a readme file or a short introduction (from a perspective of someone with no prior experience) to get started. I was first occupied with understanding how to use the program in general and what I can do with it in a broad sense. However, I was new to the concept of functional programming as a whole, which has played a major role in my personal performance. Without the help from Mr Nezamabadi I think that I would have taken significantly longer with the understanding of functional programming. It still helped a lot having something to try out in parallel and getting hands on experience with self chosen numbers for the examples and get a feeling of how everything worked

- Do you think that such an implementation can be used to help students get a better understanding of programming, in particular functional programming? Why (not)?

  Yes. The program helped me understand the basic concepts within a few hours. It's still only the basics that I managed to understand, but with more practice and more tasks and examples, I am certain that it will ease the learning process for many students that are just beginning their first moves in functional programming. It might only serve its purpose for programs with smaller data sizes, since the declaration of each variable and list can become a bit tedious when having to deal with long lists or many variables.

- Regardless of the implementation you interacted with just now, do you think that the approach of visual programming + programming-by-demonstration is a good idea in the first place? Why (not)?

  For beginners, yes. There is a reason why we do learning by demonstration even in robotics: Because it is easy to understand. Easing the learning process is always a good idea, since it enables a wider pool of people to get started with what you are doing or trying to achieve.

- In general, what do you think are the strengths of this implementation (if any)?

  In my opinion, going through your code with a numbers example is always a good idea in the first place. It makes the programmer think about how the program would see the code from its perspective. Learning this perspective of the program is crucial not only for functional programming but for all programs in general. Having these things being shown by demonstrating concrete examples and directly telling the program what it should do step by step makes the whole learning process way easier than learning it with a "real"/industry-used programming language.

- In general, what do you think are the weaknesses of this implementation?

  The user still has to learn how to use the program itself. With that I mean knowing which buttons to press in which order and which inputs and outputs need to be chosen in each step. A tutorial would be ideal to have for creating your first function, in order to get started with any subsequent function creations.

- Ask for any other comments

  Empty

A.3 Study Results B

Length of session: around 2 hours

In general, red indicates that the text was generated by the participant, while blue indicates that the text was generated by us.

Pre-Study Interview

- Do you have programming experience? If yes, please elaborate (Which programming languages are you familiar with? For how long? How would you rate your proficiency [in general and with specific languages]?).

  I do have programming experience. I am familiar with Java, C, Ocaml, Python, Haskell and a bit of Rust. I would say I am not proficient in any of them, however. I have been "programming" since I was 14 or so but mostly for fun and I have never maintained any serious project. I do however have a couple of small "projects".

- Do you have experience with functional programming? If yes, please elaborate (Which programming languages are you familiar with? For how long? How would you rate your proficiency [in general and with specific languages]?).

  I have experience with Haskell and Ocaml from university courses but I would not say that I am proficient in any of them as I only used them for a couple of months. I do think they are very interesting and could probably become proficient in them if I had a reason or more time.

- Do you have a background in Computer Science? If yes, please elaborate.

  I do. I am currently studying CS at ETH (6. Semester Bachelor).

Study

1. We shortly explain what the system is: a visual programming environment where the user can synthesize functions by demonstrating examples.
2. Give the user time to play around with the programming environment.
   a. Participant did not expect registers/lists to be clickable
   b. Participant found it confusing that the prompt for values does not specify what type of values can be entered (for example, numbers or booleans)

c. It was not really clear to the participant how function application works in the system

d. The different modes of the system were not clear to the participant, i.e. that they exist, what their functionality is and how the system switches between them

3. Ask the user about their expectations regarding the system, in particular, what they think the system needs from them and what the most complicated function is that the system can generate.

   Note: We accidentally told the participant that the system could also generate functions like map (since they asked) before they answered this question. They also asked whether functions are first-class citizens – Answered with yes.

   The system apparently provides limited support for data types (only int and bool) from what I can see. Therefore one is not able to define custom algebraic data types or similar. It seems to require some inputs in the form of registers that can be assigned to values and a selection/definition of a function that shall operate on the registers. If the system allows for function composition and recursion then it would be quite powerful (at least on first sight).

4. Tell the user to define a function that takes the average of two numbers. [Simple function composition]

   a. Reminded the participant that they could do programming-by-demonstration

   b. Participant was confused about function application: they selected `temp0 / r5` instead of first selecting `temp0` and `r5` followed by selecting `/` after pressing "Apply". – We reminded them how functions are applied.

   c. Participant did not know how to return values

   d. Participant did not realize that they had to select values as variables or as constants and ended up selecting everything as variables – Told the participant about variables and constants and how to differentiate between these by left- and right-clicking the labels in the GUI

   e. Participant was confused about function application again

   f. Participant found right-clicking the labels pretty confusing

5. Tell the user to define a function that adds a number to itself [Coupling]

Note: This task was only performed by this participant, as it is something that we came up with during the study to test the intuitiveness of coupled inputs.

   a. Participant forgot how to create a new function

   b. Participant solved this task without further problems: they intuitively knew that one register corresponds to one input

6. Tell the user to define a function that returns 1 if a number is even and 0 otherwise. [Conditionals]

   a. Participant struggled with function application: they tend to select for example `4`, `/` and `2`, instead of selecting `4` and `2` followed by selecting `/` after pressing "Apply". Additionally, aborting a function application is awkward as it requires users to press "Undo" one or more times (for example, if they created registers after pressing "Apply").

   b. Branching was unclear to the participant, in particular, what it does in the system, how it works, where they are in the tree/program and what happens after it

   c. Participant forgot what they were doing. They said that this was caused by things disappearing: for example, after adding two inputs together, the only information that remains is something like `temp0 = const0 in0 in1`, which probably is not informative enough

   d. The resulting function required a function as input. This was probably caused by the participant accidentally selecting functions to be applied as variables instead of constants.

7. Tell the user to define a function that calculates the n-th Fibonacci number (f(0) = 0, f(1) = 1, f(n) = f(n-1) + f(n-2) [14]) [Recursion]

   a. Semantics of branching in the context of examples were very unclear to the participant, especially when doing nested branching

   b. Participant was completely lost in regards to where they were in the tree/example/program, i.e. which branch they were in and whether they applied "Branch" already

   c. Participant was confused about the definition of the Fibonacci sequence

8. Tell the user to define map for a general function [Recursion, Lists, Function as input]

   a. Participant was not sure about the arguments (and their order) of map – They had to select map and look at the type signature

b. Told the participant that it is probably easier to demonstrate map using a boolean list, as that way they do not need to define a new function to use with map – They decided to use a number list anyway

c. Participant asked whether partial application is supported – Answered that partial application is not supported and that they can create a new function instead

d. Participant thought that the user-defined map was going to use a "hard-coded" function – Reminded them that function applications can be selected as variable

e. Participant was unsure of how to implement map, as they had not done functional programming in a while

f. Participant was confused by the "Select a function to be applied" mode

g. Participant forgot that map takes the function that should be applied to the list as an input during recursion

h. Participant made a mistake in the order of inputs (i.e. whether the list or function is the first argument to the function) when trying to use recursion

i. Participant forgot to add the base case

j. Participant forgot that they need to check the length of the list again (i.e. that they need to demonstrate previously demonstrated parts again)

k. Participant asked how `cons` worked

l. Participant added the wrong value to the beginning of a list because they got lost on what the generated temporaries mean and where they come from

Post-Study Interview

- How did you solve the tasks? Did you have a strategy?

  Once I found out that I had to show examples, this is what I did. I had to keep track of the branches in my head.

- Do you feel like this helped you get a better understanding of programming, in particular functional programming? Why (not)?

  Not really, I thought this was more confusing than just programming by hand, only because of the GUI because it hides a lot of information

regarding already demonstrated example steps. I find that it is quite intuitive that you can just "show examples" but it is hard to do so when you forget at which step of the branch you are or what intermediate values you already created (because you cannot see what function creates the intermediate value).

- Do you think that such an implementation can be used to help students get a better understanding of programming, in particular functional programming? Why (not)?

  I do think it can be very useful because one of the main difficulties in functional programming is the large amount of abstraction and being able to "program by example" could give students a more intuitive introduction to this different paradigm.

- Regardless of the implementation you interacted with just now, do you think that the approach of visual programming + programming-by-demonstration is a good idea in the first place? Why (not)?

  I do think it is a great idea, however, implementing a GUI that is good enough might be very challenging.

- In general, what do you think are the strengths of this implementation (if any)?

  It feels like the "backend" is sound, however, the GUI does not allow it to achieve its full potential.

- In general, what do you think are the weaknesses of this implementation?

  The general intuitiveness of the GUI is not really there yet. I feel like the user should be presented with more information about the current "state" of the implementation. Also it is not clear what the buttons do.

- Ask for any other comments

  In closing, this idea is really interesting and I think it can have huge potential, however designing a good user interface might be the main challenge which will require large amounts of investment in terms of development time and user studies to make it match its potential.

Bibliography

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

[2] 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.

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

[4] 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.

[5] Matthew Hertz. What do "cs1" and "cs2" mean? investigating differences in the early courses. In *Proceedings of the 41st ACM Technical Symposium on Computer Science Education*, SIGCSE '10, page 199–203, New York, NY, USA, 2010. Association for Computing Machinery.

[6] 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.

[7] Kevin Sookocheff. Alpha conversion. https://sookocheff.com/post/fp/alpha-conversion/, 2018. [Online; accessed 14-June-2022].

[8] 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.

[9] 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].

[10] 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.

[11] Madhavan Mukund. Variable capture. https://www.cmi.ac.in/~madhavan/courses/pl2009/lecturenotes/lecture-notes/node85.html, 2004. [Online; accessed 14-June-2022].

[12] 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.

[13] Alberto Martelli and Ugo Montanari. An efficient unification algorithm. *ACM Trans. Program. Lang. Syst.*, 4(2):258–282, apr 1982.

[14] MathIsFun.com. Fibonacci sequence. https://www.mathsisfun.com/numbers/fibonacci-sequence.html, 2020. [Online; accessed 07-July-2022].

[15] Miran Lipovača. Learn You a Haskell for Great Good! - Making Our Own Types and Typeclasses. http://learnyouahaskell.com/making-our-own-types-and-typeclasses, 2011. [Online; accessed 15-June-2022] (Year 2011 is considering the book, and not the website).

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

[17] The University of Glasgow. Source code: module GHC.Base, map. https://hackage.haskell.org/package/base-4.11.0.0/docs/src/GHC.Base.html#map. [Online; last accessed 15-July-2022].

[18] 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].

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

[20] 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.

[21] Wikipedia contributors. Unification (computer science) — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Unification_(computer_science)&oldid=1087312821, 2022. [Online; accessed 10-June-2022].

[22] MIT with contributions from: Saman Amarasinghe, Adam Chlipala, Srini Devadas, Michael Ernst, Max Goldman, John Guttag, Daniel Jackson, Rob Miller, Martin Rinard, and Armando Solar-Lezama. Reading 14: Recursion. https://web.mit.edu/6.005/www/fa16/classes/14-recursion/, 2016. [Online; accessed 14-June-2022].