Matthew Kerr and Lane Thompson

25 February 2020

CS 381 - W2020

Simperative Design Document

Introduction

Language Name - Simperative

Language Paradigm - Imperative

Unique - The Simperative programming language has a few unique features. First, all programs are fully divided into functions, which are named lists of commands. The first function is considered “Main” and is executed immediately upon loading a program; the other functions are considered helper functions, and they are loaded into the Env variable, where they can be called from anywhere in the program. Second, rather than having a static list of parameter types for each function, when a function is called, parameters are passed in via the Env type, so any number of variables can be declared, and even functions can be defined. Since the Simperative language has dynamic scope, these new functions and variables will be made visible anywhere in the program.

Design

What features does your language include? Be clear about how you satisfied the constraints of the feature menu.

- 1 - Basic data types and operations: Simperative includes the Boolean Expression type and Integer Expression Type, which have the Semantic Domain of Boolean/Integer respectively. These types include boolean and integer literals, as well as some operations involving booleans, integers, and strings. The BoolExpr type includes a comparison between two int expressions, a comparison between two string expressions and logic performed on two boolean expressions, and the IntExpr type includes addition, subtraction, and multiplication of integer expressions.
    - Level: Core , because basic expressions are all included as haskell type definitions.

- 2 - Conditionals: If conditional statements are fully supported by the Simperative language. The CMD type contains the If constructor, which takes 3 parameters, a boolean expression to determine which command to run, and two commands. If the boolean expression resolves to true, the first command is run, otherwise, the second command is run
    - Level: Core, because If statements are included in the haskell CMD type deinition
- 3 - Recursion/loops: Simperative contains two procedures for looping over code. The first method is via recursion. The Simperative language supports recursively calling functions in order to simulate a loop to perform similar actions multiple times. The second method is via while loops. The CMD type contains the While constructor, which takes two parameters, a boolean expression, and a CMD to run repeatedly while the boolean expression resolves to true.
    - Level: Core, because While loops and Function calls are included in the haskell CMD type definition
- 4 - Variables/local names: Simperative defines an Env type, which contains four different arrays: an array of int variables, an array of bool variables, an array of String variables, and an array of Functions. When a Simperative program is executed, the Env variable is operated on by CMDs one at a time until all of the CMDs in the "Main" function have been run. During this time, variables can be declared and assigned values, which adds them to the Env variable, and they can also be referenced, which searches the Env for a variable name and gets the value of that variable.
    - Level: Core, because the ability to store variables is in the Env type, the ability to modify variables is in the CMD type, and the ability to reference variables is in either the IntExpr, BoolExpr, or StrExpr type.
- 5 - Procedures/functions with arguments: Functions are usually defined in the Function Array that is part of a Program, which will inizialize them into the function array in the Env variable. Functions can be run by name from a CMD, which searches the current Env for the function name, and then executes that function. When a function is called, it also needs a list of parameters, which is a separate Env variable that contains all of the new parameters that are to be inserted into the Env. Since Env variables can contain functions, this means that another valid way of declaring a function is to place a function in the function array of the Env variable that is being passed as a parameter to a function being executed. This will load the function into the Env, making it possible for the function to be called from anywhere from that point forward.

    - Level: Core, because Functions are defined in a Haskell type, they are loaded into the environment by being placed in a Program or Env constructor, and they are called using the Exec form of the CMD type.

Extra Features

- 1 - Strings and operations - Simperative includes a string expression type that can hold String Literals and Variables, and supports concatenation of two strings. A comparison of two strings for equality is supported in the boolean expression type.
    - Level: Core, because string expressions are defined in a Haskell type
- 2 - Static Type System - Expressions are split into Boolean, Integer, and String expressions. We have built type checker functions as part of the compilation process to check all of the expressions for type errors before the program is executed.
    - Level: Syntactic Sugar, because all of our type checking is done with Haskell functions, and it is possible to build and run a program in Simperative without explicitly checking for type.

What are the safety properties of your language? If you implemented a static type system, describe it here. Otherwise, describe what kinds of errors can occur in your language and how you handle them.

- Simperative is a statically typed language. This means that we have type checker functions that will ensure that all types in a program are valid and that there will not be any type errors during execution. Additionally, we have runtime checks in place for two other types of errors, referencing a variable that has not been defined and calling a function that has not been defined. The semantic domain (Env type) contains Error messages in it, and calling a variable that does not exist yet or running a function that does not exist yet will result in the Program returning a relevant error message depending on which error was encountered.

Implementation

What semantic domains did you choose for your language? How did you decide on these?

- Simperative uses an Env type as its semantic domain. This type will hold an Error Message (if the program encountered an error), or lists of all int variables, all bool variables, and all string variables if the program did not reach an error.

The Env variable also contains Function information so that it is accessible throughout the program. However, the function list is not considered to be a part of the semantic domain, so at the end of a program when semantics are output to the console, the function list is omitted. Running semantic functions on either the Program, Function, or Cmd types will return an Env. Running semantic functions on IntExpr will result in the Maybe Int type, BoolExpr will result in Maybe Bool, and StrExpr will result in Maybe String

Are there any unique/interesting aspects of your implementation you'd like to describe?

- Simperative is a dynamically scoped language, which means that all variables and functions in a Simperative program are considered to be global and public, so they are accessible from anywhere within the program. It also means that if an external variable is referenced, that variable will contain the last value during execution, not during compilation. This choice was made for two reasons. First, so that functions can be stored in the Env variable and accessed by any command anywhere within a program. Second, so that functions can work with the primary Env variable, which is the Semantic Domain of the language, and the changes to the environment made in a function will affect the Environment outside the function.
- Additionally, because parameters are passed to a function via the Env variable, this means that function definitions can also be passed as parameters to functions, in which case they will be introduced to the Full Scope of the Program, and will be accessible anywhere. We have added restrictions, however, that a function must already be in the Env variable to be called, so it is not valid to declare a function in the parameters of its own reference, and attempting to do this will return an Error for referencing a function that does not exist.