COMP2212 – Programming Language Concepts

Group members:

Daniel Braghis, ddb1u20@soton.ac.uk

Alin Formusatii, aaf1u20@soton.ac.uk

Bagir Bazarov, bb1u20@soton.ac.uk

May 4th, 2023

Introduction to BitGrid

BitGrid is a domain-specific programming language for specifying tiling patterns. The syntax is based on a python-like pseudocode with elements from C. The language syntax and features where optimized for the narrow task of manipulating unidimensional and bidimensional arrays that represent a tile pattern made up of 0s and 1s.

BitGrid features a static variable environment with strong typing allowing arithmetic, boolean and other types of expressions to be bound to variable names. Each variable has a global scope and is final.

Expressions currently return four types currently supported by the language: Bool, Int, Range and Tile. The special types - Range and Tile represent a range with a start-end value used in iteration and a valid tile construct respectively.

BitGrid does not support NULL values, each expression is expected to return one of the four types to be considered valid.

Users may use C-style comments in their application using the "//" symbol at the start of the line to comment. Inline commenting is allowed.

The language comes bundled with a compiler that parses and runs the type-checking before running the program and redirecting the output to stdout or std error in case of an error.

BitGrid Execution Model

Before running, a BitGrid program first is analyzed by a compile-time unit comprised of a lexer unit, parser unit and a static type checker unit.

Firstly, the lexer removes comments and whitespaces from the input and returns a parse tree made of identified tokens. The parser then looks for valid sentences in the BitGrid grammar to generate an abstract syntax tree. In case a token is unidentified, or the sentence is not part of the grammar a perse error is printed to stderr in this format:

    Parse error at (<row>, <column>)

Note that the parser does some high lever type checking expecting specific types of sentences in the grammar and will report them as parse errors.

The static type checker is then run at compile time to make sure the program is well-typed and reports the final type returned by the program (this is because

BitGrid programs can only have one output). There are several types of errors that can occur when type checking is done, we will discuss them in a later section. No type errors can occur at runtime.

After the program passes the type checking, the result of parsing is passed to the run-time unit for execution. The executor makes sure the results of operations are valid and can be assigned to the respective type, keeps track of variable bindings and that an operation can be performed. There are multiple run-time errors that can occur at this stage and we will cover them in the next sections.

BitGrid Programs

The BitGrid program structure is loosely based on the following BNF:

<PROGRAM> ::= <FINAL> | <ASSIGNMENTS> <FINAL>

<FINAL> ::= <TYPE> "final" "=" <EXPRESSION> ";"

<ASSIGNMENTS> ::= <TYPE> <STRING> "=" <VAR> ";" | <TYPE> <STRING> "=" <VAR> ";" <ASSIGNMENTS>

<VAR> ::= <STRING> | <ARITH> | <BOOLEAN> | <RANGE> | <OPERATION>

<ARR> ::= "[" <BINARIES> "]" | "[" "[" <BINARIES> "]" <ARRS> "]"

<ARRS> ::= "," "[" <BINARIES> "]" | "," "[" <BINARIES> "]" <ARRS>

<BINARIES> ::= <INT> | <INT> "," <BINARIES>

<OPERATION> ::= hRepeat <RANGE> <OPERATION> | vRepeat <RANGE> <OPERATION>
| hAdd <OPERATION> <OPERATION> | vAdd <OPERATION> <OPERATION> | "(" <OPERATION> ")"
| rot90 <OPERATION> | rot180 <OPERATION> | rot270 <OPERATION> | grow <ARITH> <OPERATION>
| hReflect <OPERATION> | vReflect <OPERATION> | blank <ARITH> <ARITH> | not <OPERATION>
| and <OPERATION> <OPERATION> | <ARR> | <BOOLEAN> ? <OPERATION> : <OPERATION>
| subtile <ARITH> <ARITH> <ARITH> <OPERATION>

<BOOLEAN> ::= "(" <BOOLEAN> ")" | <BOOLEAN> "&&" <BOOLEAN> | <BOOLEAN> "||" <BOOLEAN>
| <ARITH> "<" <ARITH> | <ARITH> "<=" <ARITH> | <ARITH> ">" <ARITH> | <ARITH> "=>" <ARITH> |
| <ARITH> "==" <ARITH> | <BOOL>

<ARITH> ::= <INT> | <ARITH> "+" <ARITH> | <ARITH> "-" <ARITH> | "(" <ARITH> ")"

<TYPE> ::= Int | Tile | Bool | Range

<RANGE> ::= <ARITH> ".." <ARITH>

<BOOL> := true | false

<INT> ::= <DIGITS> | <DIGIT> <INT>

<DIGIT> ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 <DIGITS> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

This BNF was later expanded on to cover all terminals with no shift-reduce conflicts and accommodate the final feature set of the language.

A BitGrid program consists of a sequence of sentences delimited by a semicolon. All the sentences are assignments of different types. All the sentences are processed in the order they are present in the program. All the sentences follow this general structure:

<Type> <variableName> = <sequence of operations returning the Type> ";"

Below is a breakdown of all the different types of sentences present in the BitGrid language:

The Final assignment

The "final" variable name is a special reserved one in the BitGrid language. Every valid program in the language must end with a sentence assigning a value to the "final" variable. This variable is the one printed on the stdout at the end of the program execution. There can't be any other sentences after the final assignment. The final variable can be of any valid type.

Other assignments

To keep BitGrid programs readable and easy to comprehend, the language allows binding an expression to a variable. The expression can have several types: arithmetic expression, boolean expression, tile expression or an if-then-else expression. These expressions can have other expressions nested inside them as long as the correct type is returned.

The arithmetic expression

This expression always returns an Int type. It allows operations on numbers: addition, subtraction, division, and multiplication.

The boolean expression

The expression always returns a Bool type. It allows boolean arithmetic operations: comparing arithmetic expressions and boolean algebra operations "and" and "or".

The range expression

This expression always returns a Range type. It defines an Int with an upper and lower bound. This types allows iteration in a language that only has final variables.

The tile expression

This expression always returns a Tile type. A valid tile can only be represented as a unidimensional or bidimensional array containing comma delimited 0s and 1s. For a bidimensional tile the length of each row must be the same. Any operation on tiles must return a valid tile and the run-time executor checks for it. The user can define their own tile using square brackets like in other programming languages as long as it is valid.

How assignments are type-checked

BitGrid has a more rigid grammar than other languages expecting certain types of expressions when nesting expressions and reports inconsistencies at parse time. This of course can't account for all possible type errors (for example checking correct types when using variables). Thus, the type checker builds the type environment and does a complete check. Each assignment starts with a BitGrid type name that is the expected one for the expression that is assigned to it.

How assignments are evaluated

BitGrid adopts an eager evaluation strategy that will evaluate the expression at assignment and then add the pair (var_name, value) to the variables environment. Variables have to be assigned at declaration. All assignments are evaluated in the order they appear in the program line-by-line and added to the environment in that order with the "final" assignment coming last. At the end of execution, the pair ("final", value) is used to output the result to stdout.

BitGrid Errors

As previously stated, there several types of errors that can occur at compile time and execution. Below is a breakdown:

Compile-time errors:

Parse error – when the sentence is not part of the BitGrid grammar. Prints the column and row in the file it occurs.

Missing binding – the variable used has not been defined or it has not yet been evaluated and added to the environment.

Couldn't match type for assignment – The variable type and expression type that is assigned to it are mismatched.

Couldn't match type <Type> for <Operation> with <Type> - Reports the expected type or types for an expression and the actual types provided instead.

Invalid tile construct – the tile defined is not a valid tile allowed in the BitGrid language.

Incompatible type with <Type> - Reports an unexpected type error.

Run-time errors:

Invalid range values – The Range type expects the first value to be lower than the second value for the range.

Unidimensional with bidimensional tile operation error – The operation can't be performed as the two tiles have different sizes.

Can't take subtile of size – The subtile is outside the input tile bounds.

Invalid resulting tile construct – The resulting tile after operation is invalid.

Incompatible <Operation> tiles – The operation can't be performed with the given tiles.

BitGrid Iteration and Syntax Sugar

It is important to note that BitGrid does not have a classic way of iterating over a range of values. Instead, the hRepeat and vRepeat operations have slight changes in their behaviour at run-time depending on the type given in their first argument that defines the range and return the resulting tile.

The user defines a Range explicitly (arith..arith) or chooses to use a syntax sugar by defining or using a variable of just type Int. In the first case the program repeats the operation N times where N is the difference between the two values and in the second case it just executes it N times where N is the value of the arithmetic expression given.

However, if the user wants to define and use a variable inside the scope of the iteration, they must define the variable of type Range separately and then give it as the first argument of the operation. Now they can use the variable name inside the scope of the operation.

Also, note that this can be done only once as no assignments are allowed inside the iteration.