GRAMPA: Generalized Rick And Morty ProgrAmming

Contact: [email]@grinnell.edu

Reilly Grant
Grinnell College
[grantrei]

Tristan Knoth
Grinnell College
[knothtri17]

Chris Kottke
Grinnell College
[kottkech17]

Abstract

Parallel computing is an important way to process over very large problems. However, developing parallel applications is extremely difficult. We develop Generalized Rick And Morty ProgrAmming (GRAMPA), an esoteric imperative programming language supporting a simple forking model in order to simulate parallelism and introduce students shared memory and other rudimentary parallel computing concepts. The language supports only a limited set of commands, and includes syntax is based on the popular cartoon Rick and Morty in order to present parallel computing concepts in a fun and accessible manner.

1. Introduction

As parallelism becomes the most important paradigm for large-scale information processing, developing parallel applications is becoming an increasingly important skill for any developer. Students who begin to think about splitting problems up between processors earlier in their computer science education will potentially see more success learning more sophisticated parallelism paradigms later on. To this end, the popular TV show Rick and Morty presents the perfect medium through which to introduce students to rudimentary parallelism concepts. We develop GRAMPA, a simple Turing Complete imperative language with syntax based on references to Rick and Morty that supports a simple model of forking across shared memory. In Rick and Morty, the main characters travel between dimensions. The show's clear conceptual connections to multithreading may help alleviate the pain of learning to parallelize simple algorithms. GRAMPA's syntax is intended to look something like written natural language.

2. Prior Work

GRAMPA relies on a number of existing technologies, particularly the Haskell Parsec library. Our parser is built in Haskell using Parsec, which allows users to combine parsers via monads. There are also already a number of languages supporting parallel computing. C, for example, supports multithreading. OpenMP is a widely used API facilitating parallelism within a node, while MPI is often used to enable parallelism amongst the nodes of a network. Haskell itself has libraries to achieve parallelism. GRAMPA will utilize concepts from these all of these technologies.

3. Examples

An example program using two threads to find all prime numbers less than 1000 is in appendix A. One thread processes half the odd numbers (1, 5, 9, ...) and the other processes the other half of the odd numbers (3, 7, 11 ...). The program demonstrates almost all of GRAMPA's features.

4. System

GRAMPA is a Turing-Complete imperative language simulating parallelism via a simple forking model. GRAMPA includes integers, booleans, and a variety of arithmetic and boolean operations. GRAMPA also allows if statements, while loops, variable declaration, and printing. These features, while relatively limited in scope, offer functionality sufficient for exploring some parallel computing concepts. GRAMPA's key feature is its simulated parallelism. In GRAMPA, one can mark different blocks of code as "universes", and when executing the main universe, fork computation to include other universes and thus simulate multithreaded execution. The universes share variables in a sort of shared memory, and can use this to manually lock threads. The universes do not actually execute in parallel. Instead, GRAMPA executes one statement at a time from each code block, alternating between all of the universes currently in the execution pool. A more detailed explanation of the implementation of these ideas follows.

4.1 Parsing GRAMPA

Using Parsec, we can build a recursive descent parser for GRAMPA by combining parsers for different types of expressions and statements in our language. Given a grammar for our language, we can define parsers for each variable in the grammar and thus recursively parse the entire language.

The following Context-Free Grammar defines the GRAMPA syntax, and indicates the parsing hierarchy. The variable STRING refers to any string consisting only of chars, and INT refers to any integer.

$$S \rightarrow UNIV$$
$$UNIV \rightarrow \text{universe } STRING\ STMT \text{ destroy universe } \mid UNIV\ UNIV$$
$$STMT \rightarrow PORTAL \mid IF \mid DECL \mid PRINT \mid WHILE \mid STMT\ ST$$
$$EXPR \rightarrow OP1$$
$$OP1 \rightarrow AND \mid OR \mid OP2$$
$$OP2 \rightarrow NUMEQ \mid NUMLT \mid NUMGT \mid OP3$$
$$OP3 \rightarrow ADD \mid SUB \mid OP4$$
$$OP4 \rightarrow MUL \mid DIV \mid MOD \mid TERM$$

CONF 'yy Month d–d, 20yy, City, ST, Country

ACM 978-1-nnnn-nnnn-n/yy/mm. . . $15.00
DOI: http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

$TERM \rightarrow BASE \mid PARENA \mid PARENB \mid STRING$

$BASE \rightarrow INT \mid BOOL$

$BOOL \rightarrow$ right | wrong

$PORTAL \rightarrow$ lets grab our $STRING$ and portal out of here

$IF \rightarrow$ if $OP1$ then $STMT$ otherwise $STMT$ wubulubadubdub

$DECL \rightarrow STRING$ means $EXPR$

$PRINT \rightarrow$ show me $STRING$

$WHILE \rightarrow$ while $OP1$ do this for grandpa $STMT$ thanks Summer

$AND \rightarrow OP2$ and $OP1$

$OR \rightarrow OP2$ or $OP1$

$NUMEQ \rightarrow OP3$ is the same as $OP3$

$NUMLT \rightarrow OP3$ is less than $OP3$

$NUMGT \rightarrow OP3$ is greater than $OP3$

$ADD \rightarrow OP4$ plus $OP3$

$SUB \rightarrow OP4$ minus $OP3$

$MUL \rightarrow TERM$ times $OP4$

$DIV \rightarrow TERM$ divided by $OP4$

$MOD \rightarrow TERM$ mod $OP4$

$PARENA \rightarrow$ you gotta $OP3$ Morty

$PARENB \rightarrow$ you gotta $OP1$ Morty

Given the context-free grammar above, we define parsers for each of the individual substitution rules. For example, to parse a multiplication, we look for a $TERM$ on the left side of the expression, a "times" to indicate that we are multiplying two expressions, and an $OP4$ on the right hand side of the expression. In Haskell, this is implemented as follows:

Algorithm 1: Multiplication Parser

```
whitespace
e1 ← termParser
whitespace
string "times"
whitespace
e2 ← op4Parser
whitespace
return $ EBin Mul e1 e2
```

The sequence of instructions above is wrapped in a "do" block to create a parser for multiplication. `whitespace` is a parser designed to consume all whitespace. As such, our language is completely whitespace insensitive as long as one separates commands by any amount of whitespace. `string` is a parser built into Parsec, which parses a specific string. In this way, we combine parsers for different types of expressions and recursively parse the entire document.

Parsec also gives the user the option to "try" a parser. If one uses "try" when calling a specific parser, the parser will only consume input if it successfully parses the entire expression. Thus, if it encounters an error it will attempt again to parse the exact same string for a different type of expression. This feature allows us to combine parsers for different types of expressions quite easily. For example, we parse statements, which have a number of forms, in the following way:

```
stmt = try sPortal <|> try sIf <|> try sDec <|>
try sPrint <|> try sWhile
```

Thus, if the parser does not find an "if" statement, it can move on to look for a variable declaration or a print statement, for example. "Trying" different parsers in this way allows us to combine parsers like the multiplication parser described above and recursively parse the entire context-free grammar.

As a final note, one should observe that our order of operations is implicitly embedded in the grammar, and therefore the parser. The relevant portion of the grammar begins with the $EXPR \rightarrow OP1$ substitution. Here, we parse operations in ascending precedence. We parse the terminals and parenthesized expressions last in order to ensure that they are leafs in the parse tree and are therefore evaluated first.

Our parsing methodology leads to a few idiosyncrasies in GRAMPA. For example, if statements must execute statements in both the true and false cases. In other languages, programmers can often simply omit the "else" case if it is unnecessary. However, when writing GRAMPA code, if one does not want anything to execute in the "else" case, one must include a dummy instruction, such as assigning a never-used variable.

4.2 Left Associativity

Parsing a file using the process described above results in a right-associative parse tree. This means that the expression $1 + 2 + 3$ is parsed as $1+(2+3)$ rather than $(1+2)+3$, which is what one would expect. In many cases this difference is nontrivial. For example, we need to ensure that $16/4/2$ evaluates to $(16/4)/2 = 2$ rather than $16/(4/2) = 8$. To rectify this problem, we transform the abstract syntax tree(AST) generated by our parser to reorder operations and guarantee left-associativity. While we originally attempted to simply generate a left-associative Abstract Syntax Tree (AST), the subtleties of Parsec meant that it was easier to simply transform the AST after parsing. The algorithim we used was based on the following observation. Consider the expression $1 - 2 - 3$. We see that with right associativity, that this is parsed as $(1 - (2 - 3)$, and with left associativity, that it is instead parsed $((1 - 2) - 3)$. As a tree, this is represented as

We see that this can be represented generally as a change of (a op1 (b op2 c)) to ((a op1 b) op2 c). This tree transformation is represented as

We also see that if we repeatedly apply this transformation until the right child of the top node is either parentheses, or a literal, and then recurse on all children, we will have transformed a right-associative parse tree into a left-associative parse tree.

One additional note is that to preserve order of operations, addition and subtraction treat a right child which is multiplication or division node as a literal, in that the structure tree remains the same, except for recursing on the right child. After parsing the entire program, we map this function over all statements in all universes.

4.3 GRAMPA Code Generation and Execution

Behind the scene's, code generation and execution in GRAMPA relies on a variety of data types and abstractions. It is relatively simple to understand code generation by examining some of the relevant algebraic data types.

`Stmt`:

```
SDecl :: String → Exp → Stmt
SWhile :: Exp → [Stmt] → Stmt
SIf :: Exp → [Stmt] → [Stmt] → Stmt
SPrint :: String → Stmt
SPortal :: String → Stmt
```

The `Exp` data type contains constructors for literals, binary operations, parentheses, and variables. A program in GRAMPA is defined by the following data types, which we will also use as terminology when discussing technical details of the functionality.

```
type Prog = [Stmt]
type Env = [(String, Value)]
type Block = (String, Prog)
type Multi = [Block]
type Print = [String]
```

As described previously, a GRAMPA program is simply a list of imperative statements. In order to keep track of variables, we must also maintain an environment, called `Env`, mapping variable names (strings) to their values (boolean or integer literals). During execution, there is only a single environment, so different threads have access to the same variables, mimicking the concept of shared memory. A `Block` maps strings to programs in order to facilitate parallelism. The user can declare and name different blocks, using the "universe" syntax, which are then associated with their name in order to enable forking. The multiverse, called `Multi`, is an array of blocks, and contains all the information on all programs that can be executed by forking.

Finally, the `Print` data type is essentially a print buffer. As Haskell is a purely functional language, it is much easier to implement printing via a buffer rather than printing during the evaluation. Any print statement simply appends a string onto the buffer, and at the end of execution the entire buffer prints.

The actual code execution comes from two functions: `stepProg` and `stepUni`. `stepProg` takes a `Multi`, an `Env`, a `Print`, and an array of `Progs`. The program array is the programs of each block currently being executed. `stepProg` executes one instruction from each program in turn, continuing until all programs have been completely processed. If it has to execute a "portal" instruction, it simply searches for the relevant program in the `Multi` and appends it to the list of programs currently being executed. `stepUni` is basically a wrapper for `stepProg`. `stepProg` works by pattern matching for the following cases and responding accordingly:

Program array is empty: Execution is done

Head of program list is empty: Continue stepping through remaining programs (tail of program list)

Executing portal instruction: Search for the relevant block, and append its program to the list of programs being executed.

Executing other instruction: Execute instruction, update environment, print buffer, and program list accordingly

Executing a single instruction returns an environment, a list of statements, and any updates to the print buffer. Executing an if statement, for example, requires first evaluating the boolean expression and subsequently returning a set of statements, depending on what the boolean expression evaluated to. Evaluating a while loop is slightly more subtle. We evaluate the boolean expression, and if it evaluates to true, we append another identical while loop statement onto the body of the loop and return that. The other types of statements all evaluate fairly intuitively.

4.4 Parallelism in GRAMPA

Parallelism is an important concept and often is difficult to grasp for people who are new to it. People who are used to standard functional and imperative styles are often tripped up by the indeterminacy that parallelism introduces. In order to aid users who are unaccustomed to parallelism and its pitfalls and to cohere with the style of GRAMPA, we have implemented a mechanism for simulated parallelism which allows users to run programs on an indeterminate number of parallel threads. By simulating threads rather than actually implementing multithreading, we allow the user to explore this concept in a relatively consequence free environment. Furthermore, by controlling how the simulated multithreading works, we can present a simplistic representation of the concept that does not require the user to fight with more advanced topics such as context switching, locks, and signals.

Each parallel thread is represented by a different universe where the first universe listed is initially run in a single threaded environment. Universes are defined as follows:

```
universe name [STMT] destroy universe
```

In order to initiate multithreaded executions of multiple universes and essentially fork the program execution, the following Portal invocation is used:

```
lets grab our universe name and portal out of here
```

Upon executing this invocation, execution of the new universe will follow in parallel with the current executions. Rather than implementing true parallelism however, a mechanism similar to forking in C is used. In order to run this, the statement list of each universe currently being executed is stored in a list. In each step, the head statement of the head universe is popped and executed. Following this, the head universe is moved to the tail of the list. In this way, universes are cycled through and one universe will execute one instruction after all others have also executed one instruction. This workflow is changed slightly when the current executed instruction is a Portal instruction. In this case, the portal instruction returns the statement list of the new universe and this is appended to the end of the universe list followed by the universe which called the Portal instruction. When a universe has no more instructions, it is removed from the universe list. When there are no more universes in the list, execution ends.

Importantly, because all variables declared have multiversal scope, different universes can interact with and alter variables declared and used in other universes. This creates the potential for race conditions and allows users to implement their own basic locks in order to avoid these race conditions. An example of this can be found in code.txt. In addition to finding all prime numbers less than 1,000, this program also finds the sum of all of these primes. We can do this by declaring a variable sum and adding each prime we find to it. Note however that we cannot simply print sum once universe one finishes as universe two may still be processing; rather we must wait until all universes have terminated. To implement this, we introduce the variable `uTwoNotDone` which is initialized to right. At the end of universe one, we create a while loop which constantly checks the value of `uTwoNotDone`. If `uTwoNotDone` is still right, we perform an action which amounts to doing nothing and check again. At the end of universe two, we set `uTwoNotDone` to wrong in order to mark that universe two has finished process-

ing. At this point, universe one escapes its loop and prints the value of sum.

5. Conclusion and Reflection

Working in Haskell was generally very helpful in completing this project. Pattern-matching is an incredibly useful language feature, and played an important role in the overall conciseness of our code. In general, our Haskell implementation is much more concise than a similar implementation in a different language. Also, Haskell lends itself incredibly well to working with trees. It is very easy to work with the program once it is parsed into the Abstract Syntax Tree.

However, working in Haskell did make certain parts of the project more difficult. In particular, learning to use Parsec and actually parsing were both particularly difficult. Supporting infix operators made the process much more complex. Finally,

Looking forwards, there are a number of ways to improve GRAMPA. For one thing, the language itself is still somewhat limited. While it is Turing-Complete, adding features like arrays, floating point numbers, and functions would make it much more useful. We would also like to add type-checking to the compilation process. It is currently incredibly difficult to debug GRAMPA code, and type-checking would go a long way towards aiding this process (and would not be too difficult to implement). Finally, the original aim of the project was to allow the programmer to verify their parallel computations via linear temporal logic. While we did not attain this goal, as implementing the language proved more difficult than anticipated, we have the parallel computing infrastructure to perhaps implement some linear temporal logic in the future.

Acknowledgments

Special thanks to professor Peter-Michael Osera for his consistent help when we had no idea what we were doing.

References

A. Appendix

```
IMPORTANT: GRAMPA does NOT support commenting
--declares a universe "one". Note that this universe will be the universe that executes initially
universe one

   --1 and 2 are prime, but since these are edge cases, they are handled manually
   numOne means 1
   numTwo means 2

   --Adds numOne and numTwo to the print buffer (note that only variables can be added to the print buffer
   show me numOne
   show me numTwo

   --This boolean will keep track of a second concurrent execution in order to avoid race conditions. Think
of it like a pthread_mutex_t in a rudimentary C lock. Instead of True and False, GRAMPA uses right and wrong
respectively in order to make value judgements on all statements.
   uTwoNotDone means right
   --The sum of the 1 and 2 edge cases. This variable will be updated by both threads simultaneously. This
is easy to do since all variables have multiversal scope. Furthermore, because our program only simulates
multithreading, users do not need to attend to the bureaucracy of locking, setting mutex condition variables,
unlocking, and signaling conditions in order to add to this sum.
   sum means 3

   --This line signals to GRAMPA that a "concurrent" thread executing the code held in universe two be started.
   lets grab our two and portal out of here

   --iOne is the counter for that checks every other odd number between 3 and 1000 for being prime. iOne and
iTwo need to be specified due to the universal scope of variables.
   iOne means 3

   --Basic while loop
   while iOne is less than 1000 do this for grandpa

   --jOne checks if any number between 3 and (iOne - 1) divides iOne evenly
      jOne means 3

   --This boolean is initialized to True and gets set to False if any number evenly divides iOne
      notFoundOne means right

   --This is another basic while loop. Note that you gotta ... Morty is equivalent to (...). In this case it
is not necessary, but is included to demonstrate a feature of the language.
      while you gotta jOne is less than iOne and notFoundOne Morty do this for grandpa

   --This checks if iOne can be evenly divide by jOne. Note that equality is checked by the key expression
"is the same as".
         if iOne mod jOne is the same as 0 then
            notFoundOne means wrong
         otherwise
--GRAMPA requires that both the left and right side of an if statement be filled. This means that do nothing
statements like the following are required. Such is the price we pay for simplicity.
            notFoundOne means right
--All If statements must be ended by the keyword "wubalubadubdub".
         wubalubadubdub

         jOne means 2 plus jOne

   --The phrase "thanks Summer" marks the end of a while loop.
      thanks Summer

   --If iOne is prime, print it and add its value to the current sum.
      if notFoundOne then
         show me iOne
         sum means sum plus iOne
      otherwise
         notFoundOne means right
```

```
      wubalubadubdub

      iOne means 4 plus iOne
   thanks Summer

   --This is a basic lock for sum. Because we cannot assume that universe two has terminated by this point,
we cannot simply print sum; rather we must wait until universe two 'signals' to us that it is done by marking
uTwoNotDone as False.
   while uTwoNotDone do this for grandpa
      doNothing means 0
   thanks Summer

   show me sum

destroy universe

universe two

   iTwo means 5

   while iTwo is less than 1000 do this for grandpa

      jTwo means 3

      notFoundTwo means right

      while jTwo is less than iTwo and notFoundTwo do this for grandpa

         if iTwo mod jTwo is the same as 0 then
            notFoundTwo means wrong
         otherwise
            notFoundTwo means right
         wubalubadubdub

         jTwo means 2 plus jTwo

      thanks Summer

      if notFoundTwo then
         show me iTwo
         sum means sum plus iTwo
      otherwise
         notFoundTwo means right
      wubalubadubdub

      iTwo means 4 plus iTwo

   thanks Summer

   --Signal that universe two has finished adding to sum.
   uTwoNotDone means wrong

destroy universe
```