Solution to a Simplified Halting Problem*

Douglas Martins[1], Emmanuel Podestá Jr.[1], Gustavo Zambonin[1]

[1]Departamento de Informática e Estatística, Universidade Federal de Santa Catarina
88040-900, Florianópolis, Brazil

{marcelino.douglas,emmanuel.podesta,gustavo.zambonin}@posgrad.ufsc.br

1. Introduction

The halting problem is well known in computability theory. In 1937, a groundbreaking result related to this problem was proved by Turing [Turing 1937]. In modern terms, the proof states that an algorithm, which use is to determine whether a Turing machine processing a given program stops, cannot exist. More precisely, this problem can be defined as a decision problem and determines if a program will continue running or not based on a generic machine-program input pair. Formally, it can be defined as follows. Consider a Turing machine, that accepts as input a machine M and an aptly encoded program w, *i.e.* $Halt_{TM} = \{(M, w) \mid M \text{ halts with } w\}$. Additionally, let H be a machine that solves $Halt_{TM}$ if, for any input (M, w), H either accepts or rejects.

We cannot build such a machine H, hence $Halt_{TM}$ is undecidable. We give a short proof in the sequence. Suppose that machine H exists and decides $Halt_{TM}$, and let <A> be a codification of a Turing machine A. We introduce a new machine B that takes A and runs H with $(A, <A>)$, and halts if and only if H rejects. Moreover, consider that B receives as input <B>, and will run H with $(B, <B>)$. This scenario has two outcomes: (i) if H accepts, then B halts on input <B>. However, by definition, B will enter in loop if H accepts. Hence, B on input <B> does not halt; and (ii) if H rejects, then B does not halt on input <B>. However, by definition, B will terminate if H rejects. Hence, B halts on input <B>. Therefore, the halting problem is undecidable.

As an alternative, we can use reduction to prove that a problem is undecidable. More precisely, a problem A is reducible to B if B can be used to solve A. Hence, if A is undecidable, is enough to show that a solution in B can be used to decide A, unveiling a contradiction, and thus demonstrating that B is undecidable. Based on this property, if the halting problem was shown to be decidable, several problems would also be decidable. For instance, Goldbach's conjecture and calculation of the Kolmogorov complexity of a program.

We can simplify the halting problem to transform it into decidable, by means of adding restrictions to the possible machine-program pairs. In the following sections, we will define and solve an instance of a simplified halting problem. Moreover, details about the implementation and a proof of the problem decidability will be unveiled.

2. Simplified Halting Problem

It is known that there exists no algorithm which can determine if any program halts or not, as seen above. However, there exist deciders that can know if strings belong to a certain formal language. This is exactly the set of recursive languages. For instance, every

*As submitted to the INE410113 class (Theory of Computation). Source code.

context-free language is decidable [Sipser 2012, Theorem 4.9]. Many other examples of languages that always halt exist. Such is the case of what we call the Simplified Halting Problem (SHP), described by Demasi [Demasi 2013] in an assembly-like format. We briefly cover its details and prove that it is decidable after.

The main characteristics of SHP lie in their limited syntax and arithmetic operations. Commands are divided into three groups: (i) usual conditional branches that compare two integers and a copy operation; (ii) arithmetic calculations modulo 1000, with the additional restriction that dividing and taking the modulo of zero is equal to zero itself; (iii) looping constructs in the form of a `CALL` command, and program return `RET`, that accept one argument. Further, there are ten registers that compose the "memory", where `R0` and `R9` receive input and place the output from the third group of commands, respectively. Finally, programs have an upper bound of a hundred lines.

Note that there is only one method of looping, that is, recursive. This is exactly what the `CALL` command achieves. As such, to execute programs of this language, it is imperative to maintain a recursion stack and "program counters". Moreover, all "memory" is preserved between recursion calls, except for the last register. `R1` to `R8` can be seen as scratchpad registers. We present below an example of source code for SHP and execute it step-by-step. Note that syntactic limitations such as delimitation of conditional branches and uppercase enforcement is not discussed. The first line informs that the program has 8 lines and that its first input is 9.

```
8 9
IFEQ R0,0
RET 1
ENDIF
MOV R1,R0
SUB R1,1
CALL R1
MUL R9,2
RET R9
```

We will start by comparing equality of `R0` $= 9$ and 0 and returning 1 if so. This is the stop condition of the recursive loop. Then, we copy `R0` to `R1`, and decrement `R1`. We save the "memory" state and start a new execution of the program with `R1` $= 8$ as `R0`. This is repeated until the stopping condition is reached, and thus, the recursion stack has nine entries. When `RET 1` is executed, `R9` $= 1$ and the "program counter" moves to the `MUL` instruction, since it is directly after `CALL`. Then, `R9` is doubled and returned, once again to the line below `CALL`, since it was the origin of the recursion.

We can see that this is repeated as long as there are entries in the stack. Since the stopping condition returns 1 and that is doubled repeatedly, the point of this program is to recursively calculate the n-th power of 2, where n is the recursion depth and, indeed, the input of the program.

Now, we turn to the question on whether SHP is decidable. Evidently, the language has a finite number of configurations, due to its limited arithmetic operations and memory bank. This behaviour is independent of the instruction that modified the program state, since a given configuration can be reached from different lines. By this reasoning, it is enough to check that the number of lines executed is greater than the number of possible configurations. This is akin to say that the language is as powerful as a linear bounded

automaton.

This strategy, albeit correct, will not yield the necessary performance on simple programs such as `CALL R0; RET R0`. The state for this program actually never changes, and still, to identify its non-halting behaviour, one would have to go through all the possible program configurations. However, note that if a configuration is reached more than once, this means exactly that an infinite loop exists. That is, a state that leads to itself will never exit this connection. The pigeonhole principle is the formalisation of this notion.

Identifying state repetitions is a simpler approach, since it needs not to look at all the possible states for a program. Since it cannot have more than 100 lines, the only instruction that can alter this behaviour semantically is `CALL`, by means of its looping capabilities. Hence, infinite loops only happen when two configurations are equal, exactly when those instructions are executed. Thus, it is only necessary to check for situations of this case.

With this information in mind, we need only to define what is such a configuration in the context of SHP. Recall that no additional information on the depth level or line number is needed. Then, we define the "memory state", that is, the sequence of registers `R0` to `R9`, as the configuration. By means of checking if a "memory state" has already occurred within former `CALL` instructions, one can identify if an infinite loop will occur. Ergo, it is always possible to know whether a program will halt or not. Thus, SHP is decidable.

3. Implementation

The full source code for the SHP language parsing and execution unit can be read in Appendix A, stripped of comments for brevity. We shortly discuss it in the sequence. Data structures were carefully chosen to prevent data access overhead. As such, unordered maps and sets are heavily featured, since their underlying STL implementation is a red-black tree. Additionally, hashing functions were created, to enable insertion of custom structures into these containers.

We parse the input with regular expression rules created from the description of the problem [Demasi 2013] and check for any syntax errors. We must note that, while the program description says that arithmetic is limited to modular arithmetic modulo 1000, there exists a wrong input program that features exactly this number. Hence, the program has a workaround to accept this variable. Further, the input value limit of 100 is also not respected — we believe it is a typo and should be $0 \leq n < 1000$. Helper functions that translate values from the "memory" and parse each program line into a valid "instruction" are given shortly below.

Lines 115–159 present the function that loops over an entire program and evaluates its lines, following the control flow if needed. It takes into account a "program counter" and a recursion stack, essential to simulate the inner works of the toy language. If instructions are arithmetic or conditional, a function map is invoked to decide what should be executed. Further, in the case of arithmetic functions, recall that operations have to follow a set of rules that, for instance, allow division by zero.

The most complex behaviour of the code, however, is met when parsing instruc-

tions that enter and exit recursion states. An optimisation is added here, that features memoisation of previously calculated recursion states. This allows the code to pass the $\frac{1}{8}$ of a second time target, since there are input programs that calculate large Fibonacci numbers, and it is known that if intermediary results are not memoised, this calculation will take a very long time. Evidently, the recursion stack is modified, the "program counter" is reset, and values are returned as usual.

Within this code, the strategy for preventing infinite loops is also implemented. It is enough to compare if a recursion state has already occurred, as discussed above. Note that the hashing function used in this comparison was not carefully constructed as to prevent collisions, but it seems to distribute all "memory" states distinctly. Finally, once the instruction is parsed, the "program counter" is incremented and the program may return if it finished its execution.

All programs are executed and results are printed according to the problem definition. After determining its correct parsing and execution of the toy language, the code takes 30 ms to calculate the entirety of the input. Parsing instructions uses a considerable amount of time, but this cannot be memoised, *e.g.* `ADD R1,R2` may appear in two programs, but have completely different meanings depending on the contents of the "memory" at each "program counter" state.

Using simpler string structures, manual translation of strings to integers, and substituting the "memory" from a map to an array lead the code to execute in 4 ms. Correctly emulating recursion loops, as well as these performance improvements, were the main obstacles. Further optimisations can be achieved if large parts of the code is rewritten, for instance, removing function maps and regular expressions. Ergo, to prevent reduction of code readability and maintenance, we leave it as is.

References

[Demasi 2013] Demasi, P. (2013). The Halting Problem, URI Online Judge Problem 1405. Accessed: 09 nov 2018.

[Sipser 2012] Sipser, M. (2012). *Introduction to Theory of Computation*. 3rd edition.

[Turing 1937] Turing, A. M. (1937). On Computable Numbers, with an Application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, s2-42(1):230–265.

A. C++ implementation of SHP

```
#include <iostream>
#include <regex>
#include <unordered_map>
#include <unordered_set>

static const uint16_t constexpr MAX_INT = 1000;
static const uint16_t constexpr pow10[3] = {1, 10, 100};

using bank = std::array<uint16_t, 10>;
using instruction = std::tuple<std::string_view, uint8_t, uint16_t, uint16_t>;
template <typename T>
using func_map_type =
    std::unordered_map<std::string_view, std::function<T(uint16_t, uint16_t)>>;
using cond_func_map_type = func_map_type<bool>;
using arith_func_map_type = func_map_type<uint16_t>;

// Provides a way to use an array of integers as a key in unordered containers.
struct bank_hash {
  std::size_t operator()(const bank &b) const {
    uint32_t summation = 0;
```

```
    for (const uint16_t reg : b) {
      summation = summation * 31 + std::hash<uint16_t>{}(reg);
    }
    return std::hash<uint32_t>{}(summation);
  }
};

// In this language, a program consists of a sequence of lines, a memory
// consisting of ten registers, markers that represent how a conditional should
// modify the program counter, and a set of old memory states at each CALL, to
// identify infinite loops.
struct program {
  std::vector<std::string> lines;
  std::unordered_map<uint32_t, uint32_t> cond_markers;
  bank reg_bank = {0};
  std::unordered_set<bank, bank_hash> recursion_states;
};

// Creates a structure resembling a program, according to the definition below,
// doing so through regular expressions to match lines and check for syntax
// errors.
program load_program(std::istream &file, const std::regex &match) {
  program p{};

  try {
    uint32_t number_lines = 0, current = 0;
    std::stack<uint32_t> cond_matches;

    file >> number_lines >> p.reg_bank[0];

    // end of input with multiple programs
    if (number_lines == 0 && p.reg_bank[0] == 0) {
      return p;
    }

    // problem restrictions
    if (number_lines > 100 || p.reg_bank[0] >= 1000) {
      exit(EXIT_FAILURE);
    }

    // ignore rest of line
    file.ignore(std::numeric_limits<std::streamsize>::max(), '\n');

    std::string line;
    while (current != number_lines && getline(file, line) &&
           std::regex_match(line, match)) {
      p.lines.push_back(line);
      current = p.lines.size();
      if (line == "ENDIF") {
        p.cond_markers[cond_matches.top() - 1] = current - 1;
        cond_matches.pop();
      } else if (line.find("IF") != std::string::npos) {
        p.cond_markers.emplace(std::make_pair(current - 1, 0));
        cond_matches.push(current);
      }
    }

    bool unbalanced_ifs = !cond_matches.empty();
    bool last_line_not_ret =
        (current != 0) && p.lines.back().find("RET") == std::string::npos;

    if (unbalanced_ifs || last_line_not_ret) {
      exit(EXIT_FAILURE);
    }
  } catch (const std::exception &e) {
    exit(EXIT_FAILURE);
  }

  return p;
}

// Gives the value for a register or numerical string.
uint16_t parse_arg(const program &p, const std::string_view &s) {
  if (s.empty()) {
    return -1;
  }
  if (s[0] == 'R') {
    return p.reg_bank[s[1] - '0'];
  }
  std::string::size_type len = s.size();
  uint16_t n = 0, i = len;
  // pretend we're atoi
  while ((i--) != 0u) {
    n += pow10[i] * (s[len - i - 1] - '0');
  }
  return n;
}

// Converts a program line into a tuple of strings and integers, representing
// the opcode, first, second and third arguments, respectively, if applicable.
instruction parse_inst(const program &p, const uint32_t n) {
  const std::string_view s = p.lines[n];
  const std::string::size_type space = s.find(' '), comma = s.find(',');

  // binary instruction if a comma is present
```

```
  if (comma != std::string::npos) {
    return std::make_tuple(
        s.substr(0, space), s.substr(space + 1, comma - space - 1)[1] - '0',
        parse_arg(p, s.substr(space + 1, comma - space - 1)),
        parse_arg(p, s.substr(comma + 1, std::string::npos)));
  }

  // unary instruction if a space is present
  if (space != std::string::npos) {
    return std::make_tuple(s.substr(0, space), s.substr(space + 1)[1] - '0',
                           parse_arg(p, s.substr(space + 1)), -1);
  }

  // endif keyword, degenerate case
  return std::make_tuple(s, -1, -1, -1);
}

// Processes the program according to the language description. Arithmetic and
// conditional instructions are calculated through function map lookups. CALL
// and RET are unique in their behavior and cannot be easily isolated.
uint16_t evaluate(program &p, cond_func_map_type &cond_functions,
                  arith_func_map_type &arith_functions) {
  uint32_t current = 0;
  uint16_t arg1, arg2;
  uint8_t output;
  std::stack<std::pair<uint32_t, bank>> stack;
  std::unordered_map<uint16_t, uint16_t> memoization;
  std::string_view command;

  while (current != p.lines.size()) {
    std::tie(command, output, arg1, arg2) = parse_inst(p, current);

    if (command == "CALL") {
      if (memoization.find(arg1) != memoization.end()) {
        // memoize CALL evaluations and return early if argument is known
        p.reg_bank[9] = memoization[arg1];
      } else {
        // push recursion to stack and keep track of program counter and memory
        stack.push(std::make_pair(current, p.reg_bank));
        current = -1;
        p.reg_bank[0] = arg1;
        if (p.recursion_states.find(p.reg_bank) == p.recursion_states.end()) {
          // if memory state has already happened, an infinite loop will occur
          p.recursion_states.insert(p.reg_bank);
        } else {
          return MAX_INT;
        }
      }
    } else if (command == "RET") {
      // save to memoization if return is still not there
      memoization[p.reg_bank[0]] = p.reg_bank[9] = arg1;
      if (static_cast<uint8_t>(!stack.empty()) != 0u) {
        std::tie(current, p.reg_bank) = stack.top();
        p.reg_bank[9] = arg1;
        stack.pop();
      } else {
        return p.reg_bank[9];
      }
    } else if (command[0] == 'I') {
      current = cond_functions[command](arg1, arg2) ? current
                                                    : p.cond_markers[current];
    } else if (command[0] != 'E') {
      p.reg_bank[output] = arith_functions[command](arg1, arg2) % MAX_INT;
    }
    current++;
  }

  return p.reg_bank[9];
}

int32_t main() {
  const std::string integers("([1-9]?[0-9]{1,2})|1000"), registers("R[0-9]"),
      operand("(" + integers + "|" + registers + ")"), no_arg_key("ENDIF"),
      call_key("(CALL|RET)"), valid_arith_key("(MO[DV]|ADD|SUB|MUL|DIV)"),
      valid_cond_key("(IF(N?EQ|GE?|LE?))"),
      valid_call_line(call_key + " " + operand),
      valid_arith_line(valid_arith_key + " " + registers + "," + operand),
      valid_cond_line(valid_cond_key + " " + operand + "," + operand);

  const std::regex valid_lines(no_arg_key + "|" + valid_call_line + "|" +
                                   valid_arith_line + "|" + valid_cond_line,
                               std::regex_constants::optimize);

  cond_func_map_type cond_functions = {
      {"IFEQ", std::equal_to<>()}, {"IFNEQ", std::not_equal_to<>()},
      {"IFG", std::greater<>()},   {"IFGE", std::greater_equal<>()},
      {"IFL", std::less<>()},      {"IFLE", std::less_equal<>()},
  };

  arith_func_map_type arith_functions = {
      {"ADD", std::plus<>()},
      {"SUB",
       [](auto a, auto b) { return (a - b == -1) ? MAX_INT - 1 : a - b; }},
      {"MUL", std::multiplies<>()},
      {"DIV", [](auto a, auto b) { return (b == 0) ? 0 : a / b; }},
```

```
        {"MOD", [](auto a, auto b) { return (b == 0) ? 0 : a % b; }},
        {"MOV", [](auto /*a*/, auto b) { return b; }},
    };

    program p{};
    do {
      p = load_program(std::cin, valid_lines);
      uint16_t result = evaluate(p, cond_functions, arith_functions);
      if (result == MAX_INT) {
        std::cout << "*" << std::endl;
      } else if (static_cast<uint8_t>(!p.lines.empty()) != 0u) {
        std::cout << result << std::endl;
      }
    } while (static_cast<uint8_t>(!p.lines.empty()) != 0u);

    exit(EXIT_SUCCESS);
}
```