Chapter 1

Preface

1.1 Motivation

The halting problem is undecidable in general, however this property is often abused to deduce that for all programs. The intent of this project is to explore some context in which the halting property *is* decidable, and to analyze how useful this indeed is.

1.2 Expectations of the reader

The reader is expected to have a background in computer science on a graduate level or higher. In particular, it is expected that the reader is familiar with basic concepts of compilers, computability and complexity, which are subject to basic undergraduate courses at the state of writing. Furthermore, the reader is expected to be familiar with discrete mathematics and the concepts of functional programming languages. Ideally, the reader should know at least one purely functional programming language.

For those still in doubt, it is expected that the following terms can be used without definiiton:

- Algorithm, Recursion, Induction, Big O Notation
- Regular Expressions (`preg` syntax)
- Backus-Naur Form
- Turing Machine, Halting Problem
- List, Binary tree, Head, Tail

Chapter 2

On the general uncomputability of the halting problem

2.1 Computable problems and effective procedures

A computable problem is a problem that can be solved by an effective procedure.

A problem can be solved by an effective procedure iff the effective procedure is well-defined for the entire problem domain[1], and iff passing a value from the domain as input to the procedure *eventually* yields a correct result (to the problem) as output of the procedure. That is, an effective procedure can solve a problem if it computes an injective partial function that associates the problem domain with the range of solutions to the problem.

An effective procedure is discrete, in the sense that computing the said function cannot take an infinite amount of time. To do this, an effective procedure makes use of a finite sequence of steps that themselves are discrete. This has a few inevitable consequences for the input and output values, namely that they themselves must be discrete and that there must be a discrete number of them[2].

Proof. An infinite value cannot be processed nor produced by a finite sequence of discrete steps. □

An effective procedure is also deterministic, in the sense that passing the same input value always yields the same output value. This means that all of the steps of the procedure that are relevant to it's output[3] are themselves deterministic.

Proof. If a procedure made use of a stochastic process to yield a result, that stochastic process would have to yield the output for the same input if the global deterministic property of the procedure is to be withheld. This is clearly absurd. □

In effect, a procedure can be said to comprise of a finite sequence of other procedures, which themselves may comprise of other procedures, however, all procedures eventually bottom out, in that a finite sequence of composite procedures can always be replaced by a finite sequence of basic procedures that are implemented in underlying hardware.

- effective procedure
- effectively decidable
- effectively enumerable

[1] Invalid inputs are, in this instance, irrelevant.
[2] A finite sequence of discrete values can be trivially encoded as a single discrete value.
[3] All other steps can be omitted without loss of generality.

2.2 Enumerability

2.2.1 Enumerable sets

Enumerable sets, or equivalently countable or recursively enumerable sets, are sets that can be put into a one-to-one correspondence to the set of natural numbers $\mathbb{N}$, more specifically:

Definition 0.1. *An enumerable set is either the empty set or a set who's elements can placed in a sequence s.t. each element gets a consecutive number from the set of natural numbers* $\mathbb{N}$.

2.2.2 Decidability

Definition 0.2. *A problems is decidable if there exists an algorithm that for any input event*

- Recursively enumerable – countable sets
- Co-recursively enumerable

2.3 Cantor's diagonalization

Cantor's diagonalization argument is a useful argument for proving unenumerability of a set and hence it's uncomputability.

The original proof shows that the set of infinite bit-sequences is not enumerable.

Proof. Assume that sequence S is an infinite sequence of infinite sequences of bits. The claim is that regardless of the number of bit-sequences in S it is always possible to construct a bit-sequence not contained in S.

Such a sequence can be represented as a table:

Such a sequence is constructable by taking the complements of the elements along the diagonal of all

□

2.4 The halting problem

2.5 Rice's statement

2.6 Primitive recursion

All primitive recursive programs terminate.

2.7 Introduction to size-change termination

The size change termination .. why values should be well-founded

2.8 The language to be defined

The soft version.

Chapter 3

Language

3.1 The language `D`

In the following chapter the language `D`[1] is described in terms of an extended Backus-Naur form[2]. It is described in the simplest possible terms, that is, extraneous syntactical sugar and basic terms are left out of the core language definition. Instead, these are defined as necessary in the latter chapters.

3.1.1 General properties

The intent of the language is for it be used to explain concepts such as size-change termination. One of the fundamental concepts required of the language of application is that it's datatypes are well-founded. That is, any subset S of the range of values of some well-defined type has a value s s.t. $\forall s' \in S\ s \leq s'$. This makes it ideal to chose some oversimplistic data type structure rather than an army of basic types. Besides, an apropriately defined basic data type should be able to represent arbitrarily complex data values.

The language is initially first-order since the size-change termination principle is first described for first-order programs later on in this work. However, the language is designed so that it is easy to turn it into a high-level language without much effort. This may prove necessary as we try to expand size-change termination to higher-order programs.

The language is a call-by-value and purely functional to avoid any problems that could arise from regarding lazy programs or where the notion of a global state of the machine is relevant. Simply put, this is done to ensure elegance of further proof with the help of the language.

3.1.2 Data & Functions

The language `D` is untyped, and represents all data in terms of *unlabeled ordered binary trees*. Such a tree is recursively defined as a finite set of nodes which is either empty or consists of a single node with two trees as it's left and right child, respectively.

`D` represents an empty tree with the atom `0`. Node constrution is done with the right-associative infix binary operator '`.`', within expressions. Node destruction is done with the exact same operator, except that it is done while pattern matching an argument list to a parameter list of a function declaration. Conventional braces can be used to override the right-associativity of the '`.`' operator in both cases.

Hence, the grammar for expressions and function declarations is defined as follows:

[1] The choice of the letter `D` bares no special meaning.

[2] The extension lends some constructs from regular expressions to achieve a more concise dialect. The extension is described in further detail in Appendix A.

$$\texttt{<expression>} ::= \texttt{<value>}\ (\ \text{‘}\texttt{.}\text{’}\ \texttt{<expression>}\ )\ ? \tag{3.1}$$

$$\texttt{<value>} ::= \text{‘}\texttt{0}\text{’} \mid \text{‘}\texttt{(}\text{’}\ \texttt{<braces>}\ \text{‘}\texttt{)}\text{’} \mid \texttt{<variable-name>} \tag{3.2}$$

$$\texttt{<braces>} ::= \texttt{<expression>} \mid \texttt{<application>} \tag{3.3}$$

$$\texttt{<application>} ::= \texttt{<function-name>}\ \texttt{<expression>}^{+} \tag{3.4}$$

$$\texttt{<function>} ::= \texttt{<function-name>}\ \texttt{<pattern>}^{+}\ \text{‘}\texttt{:=}\text{’}\ \texttt{<expression>} \tag{3.5}$$

$$\texttt{<pattern>} ::= \texttt{<pattern-value>}\ (\ \text{‘}\texttt{.}\text{’}\ \texttt{<pattern>}\ )\ ? \tag{3.6}$$

$$\texttt{<pattern-value>} ::= \text{‘}\texttt{0}\text{’} \mid \text{‘}\texttt{\_}\text{’} \mid \text{‘}\texttt{(}\text{’}\ \texttt{<pattern>}\ \text{‘}\texttt{)}\text{’} \mid \texttt{<variable-name>} \tag{3.7}$$

The term ‘_’ in <pattern-value> is the conventional wildcard operator – it indicates a value that is irrelevant to the function declaration. Multiple wildcards in the parameter list indicate possibly different value arguments, while multiple occurances of the same variable name in the parameter list are disallowed.

It is worth noting that the sets <function-name> and <variable-name> are disjoint, but are otherwise both defined by the nonterminal <name>:

$$\texttt{<name>} ::= [\text{‘}\texttt{a}\text{’}\texttt{-}\text{‘}\texttt{z}\text{’}]\ (\ [\text{‘}\texttt{-}\text{’}\ \text{‘}\texttt{a}\text{’}\texttt{-}\text{‘}\texttt{z}\text{’}]^{*}\ [\text{‘}\texttt{a}\text{’}\texttt{-}\text{‘}\texttt{z}\text{’}]\ )? \tag{3.8}$$

3.1.3 Size

Althought the language is already complete, it would prove useful for further analysis to define the notion of size, and hence the equality and order of values.

We define the size of a value to be the number of nodes in the tree that represents that value.

Hence, the tree `0` has the value 0, the tree `0.0` has the value 1, and the tree `0.0.0` has the value 2 as does it's symmetrical equivalent, `(0.0).0`.

This allows us to define the, otherwise built-in, function `less` in a primitive recursive fashion as follows:

```
less 0 0 = 0
less _._ 0 = 0
less 0 _._ = 0.0
less AR.AL BR.BL = or (and (less AR BR) (less AL BL))
```

This definition indicates that we choose for the empty tree to represent the value *false*, and for the tree `0.0` to represent the value *true*. We'll keep the definition even more generic, and let the *nonempty* tree represent the value *true*, as shall become useful when we define the higher-order function `if` (§ 3.1.5/7).

Since the values begin at 0 and grow at the rate of 1 ... we can define it as syntactic sugar and use nonnegative integers where ...

In addition to defining the actual data type we need to specify how we're going to reason about it. Specifically, the questions of equality and order of values constructed in this manner have to be answered.

For all intents and purposes, we can let the *absolute value* of such a tree-structured value be equal to $n-1$, where n is the number of leafs in the tree. Hence, the tree `0` denotes 0, `0.0` denotes 1, `0.0.0` denotes 2 and so on.

The choice of this data representation yields the following properties for the construction and destruction operators:

Lemma 0.1. *Construction of value yields a value strictly greater than either of it's constituents. Specifically, the absolute value of the new value is the sum of the absolute values of the constituents.*

Lemma 0.2. *Destruction of a value yields a pair of values who's absolute values are strictly less than the absolute value of the original value.*

3.1.4 Programs

Programs are defined in a conventional functional context and without mutual recursion, namely:

$$\texttt{<program>} ::= \texttt{<function>}^{*}\ \texttt{<expression>} \tag{3.9}$$

The order of the function definitions does matter wrt. pattern matching in so far as those defined before are attempted first, if the match fails, the next function with the same signature[3] is attempted.

Note, that we let the number of function definitions be zero as an `<expression>` is a valid program as well. More generally, the program can be thought of as a constant function, where the actual `<expression>` simply has access to some predefined functions defined by the function definitions in the program.

3.1.5 Built-in high-order functions

Although `D` is initially a first-order language, we will ignore that limitation for a bit and define a few higher-order functions to provide some syntactical sugar to the language. Beyond the discussion in this section, these higher-order functions should be regarded as `D` built-ins.

Branching

In the following definition, the variable names `true` and `false` refer to expressions to be executed in either case.

```
if 0 _ false := false
if _._ _ true := true
```

As you can see, we employ the C convention that any value other than 0 is a "truthy" value, and the expression `true` is returned.

Although the call-by-value nature of the language does not allow for short-circuiting the if-statements defined in such a way, this shouldn't be any impediment to further analysis.

3.1.6 Sample programs

As an illustration of the language syntax, the following program reverses a tree:

```
reverse 0 := 0
reverse left.right := (reverse right).(reverse left)
```

The following program computes the Fibonacci number `n`:

```
fibonacci 0 x y := 0
fibonacci 0.0 x y := y
fibonacci n x y := fibonacci (minus n 0.0) y (add x y)
```

3.2 Semantics

In the following section, the operational semantics of the language `D` are defined in terms of structured operaitonal semantics[3]. Table 3.1/8 specifies most[4] of the syntactical elements used to define the semantic reduction rules below.

In addition to the notation specified in the table, we'll make use of Haskell-like list comprehension when dealing with lists of elements. For instance, $[e]$ refers to a list of expressions, and $[e'|e]$ refers to a list of expressions that starts with the expression e' and is followed by the list of expressions, e. Also a bit alike Haskell, in both cases above, e is used as both a type and a variable.

[3]In this case comprising of the name of the function and it's arity.

[4]The rest is discussed further below and in § 3.2.1/8 .

Notation	Description
e	expression
v	value
n	variable name
p	pattern
0	the atom 0
$\cdot$	'.'
σ	memory
$\sigma[n]$	the value of variable n in memory, returns some v

Table 3.1: Some of the syntactical elements used in the reduction rules for D.

3.2.1 Memory

For the sake of an elegant notation, we'll define the notion of memory as a set of stacks, one for each variable in the program. If a variable n has an empty stack, it is undefined, otherwise the value of the variable (in the present scope) is the value at the top of the corresponding stack.

This model of memory allows us to deal with abitrary scope[5] in a rather elegant matter, where we simply push a new value onto the corresponding stack when we enter a nested scope and pop off the corresponding stacks when exiting a nested scope. Hence, visiting a nested scope has the following operational semantics wrt. σ:

$$\sigma \longrightarrow \sigma(n) \leftarrow v \longrightarrow \sigma$$

Following the conventions of structured operational semantics, the value of an element, such as σ, does not change throughout a reduction rule. So in the above example, the starting σ is equivalent to the final σ.

3.2.2 Evaluation

D has only one operator, namely '.', which is a right-associative binary operator, hence expressions are evaluated using the following triplet of rules:

$$\frac{\langle e',\, \sigma\rangle \longrightarrow \langle e'',\, \sigma\rangle}{\langle e \cdot e',\, \sigma\rangle \longrightarrow \langle e \cdot e'',\, \sigma\rangle} \tag{3.10}$$

$$\frac{\langle e,\, \sigma\rangle \longrightarrow \langle e',\, \sigma\rangle}{\langle e \cdot v,\, \sigma\rangle \longrightarrow \langle e' \cdot v,\, \sigma\rangle} \tag{3.11}$$

$$\langle v \cdot v',\, \sigma\rangle \longrightarrow \langle v'',\, \sigma\rangle \qquad (\text{where } v'' = v \cdot v') \tag{3.12}$$

Variables

As expressions may contain variable names, we need a way to retrieve the values of variables from memory:

$$\langle n,\, \sigma\rangle \longrightarrow \langle \sigma[n],\, \sigma\rangle \tag{3.13}$$

Application

Function application is left-associative and has variable (constant at run time) arity of at least one. The arity of the application depends on the declaration that the function specifier, f, points to. Hence, we begin by evaluating the function specifier itself to some expression λ and a pattern list $[p]$, that is, it's corresponding function declaration:

[5] Although first-order D can make little use of that.

$$\frac{\langle f,\ \sigma\rangle \longrightarrow \langle\langle\lambda,\ [p]\rangle,\ \sigma\rangle}{\langle\langle f,\ [e]\rangle,\ \sigma\rangle \longrightarrow \langle\langle\langle\lambda,\ [p]\rangle,\ [e]\rangle,\ \sigma\rangle} \tag{3.14}$$

Note, in a first-order context λ bares no special meaning, however, the letter is carefully chosen to aid further extension of `D` to it's higher-order sibling.

Followed by evaluation of the pattern and expression lists:

$$\frac{\langle p',\ \sigma\rangle \longrightarrow \langle p'',\ \sigma\rangle \wedge \langle e',\ \sigma\rangle \longrightarrow \langle e'',\ \sigma\rangle}{\langle p\cdot p',\ e\cdot e',\ \sigma\rangle \longrightarrow \langle p\cdot p'',\ e\cdot e'',\ \sigma\rangle} \tag{3.15}$$

$$\frac{\langle e',\ \sigma\rangle \longrightarrow^{*} \langle 0,\ \sigma\rangle}{\langle p\cdot 0,\ e\cdot e',\ \sigma\rangle \longrightarrow \langle p,\ e,\ \sigma\rangle} \tag{3.16}$$

$$\frac{\langle e',\ \sigma\rangle \longrightarrow^{*} \langle v',\ \sigma\rangle}{\langle p\cdot n,\ e\cdot e',\ \sigma\rangle \longrightarrow \langle p,\ e,\ \sigma(n) \leftarrow v'\rangle} \tag{3.17}$$

We complete application by evaluating λ with the new memory state:

$$\langle\lambda,\ \sigma([n]) \leftarrow [v]\rangle \longrightarrow \langle v',\ \sigma\rangle \tag{3.18}$$

Note, we return to the original σ once λ is evaluated.

Bibliography

[1] P. Naur (ed.), Revised Report on the Algorithmic Language ALGOL 60; CACM, Vol. 6, p. 1; The Computer Journal, Vol. 9, p. 349; Num. Math., Vol. 4, p. 420. (1963); Section 1.1.

[2] A. M. Turing, On computable numbers with an application to the Entscheidungsproblem; Proceedings of the London Mathematical Society, 42(2):230-265, (1936).

[3] Gordon D. Plotkin, A Structural Approach to Operational Semantics, Journal of Logic and Algebraic Programming (2004) Volume: 60-61, Issue: January, Publisher: Citeseer, Pages: 17-139.

Appendix A

Extended-BNF

This report makes use of an extended version of the Backus-Naur form (BNF). This appendix is provided to cover the extensions employed in the report. This is done because there is seemingly no universally acknowledged extension, unlike there is a universally acknowledged Backus-Naur form, namely the one used in the ALGOL 60 Reference Manual[1].

A.1 What's in common with the original BNF

The following parts are in-common with the original Backus-Naur form:

Construct	Description
$< \ldots >$	A metalinguistic variable, aka. a nonterminal.
::=	Definition symbol
$\vert$	Alternation symbol

Table A.1: Constructs in common with the original BNF.

In the original BNF, everything else represents itself, aka. a terminal. This is not preserved in this extension – all terminals are encapsulated into single quotes.

A.2 Constructs borrowed from regular expressions.

The use of single quotes around all terminals allows us to give characters such as $($, $)$, $]$, $]$, $*$, $+$, and * special meaning, namely:

Construct	Meaning
$(\ldots)$	Entity group
$[\ldots]$	Character group
-	Character range
$*$	0-∞ repetition
$+$	1-∞ repetition
?	0-1 repetition

Table A.2: Constructs borrowed from regular expressions.

An entity group is a shorthand for an auxiliary nonterminal declaration. This means, for instance, that using the alternation symbol within it would mean an alternation of entity sequences within the entity group rather than the entire declaration that contains the entity group.

A character group may only contain single character terminals and an alternation of the terminals is implied from their mere sequence. It is identical to an auxiliary single character nonterminal declaration. A character range binary operator can be used to shorten a given character group, e.g. [‘a’-‘z’] implies the list of characters from ‘a’ to ‘z’ in the ASCII table. Moreover, a character range is the only operator allowed in a character group.

Applying the repetition operators to either the closing brace of an entity group or the closing bracket of a character group has the same effect as applying the repetition operator to their respective hypothetical auxiliary declarations.

A.3 Nonterminals as sets and conditional declarations

Another extension to the original BNF is the ability to use nonterminals as sets in declaration conditions. For example, if the two nonterminals, <type-name> and <constructor-name>, are both declared in terms of the <literal> nonterminal, but type names and constructor names should not intersect in a given program, then we can append the following condition to one or both declarations:

$$\text{s.t. } \texttt{<type-name>} \cap \texttt{<constructor-name>} \equiv \emptyset$$

Where the shorthand s.t. stands for “such that”. This implies that the nonterminals <type-name> and <constructor-name> represent the sets of character sequences that end up associated with the respective nonterminals for any given program, and can be used in conjunction with regular set notation.