Chapter 1

On the general uncomputability of the halting problem

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.

In effect, a procedure can be said to comprise of a finite sequence of other procedures, some basic, others composite, which it executes in some order in order to produce an output value.

Such a definition has no sense of "scope", which is unnecessary for a purely functional definition as relevant variables can be passed down the chain as necessary. This "passing down" however, is impractical

The procedure input is a continuous machine state, the output is a discrete change of the continuous state. And yet, the values that a procedure may use to produce it's result are all discrete.

Instructions that the computer can execute may have no effect, but such instructions are seldom of any particular interest. More amusing instructions alter some sort of state that the computer withkeeps across instructions.

An instruction commands a machine to perform a discrete and deterministic alternation of the it's own state. For the simplest of purposes, the machine state can comprise of the index of the currently executing instruction, known as the program counter, the current state of the machine memory (tape), and the finite sequence of instructions that the machine is using as it's program.

The machine state comprises of the state of the tape, the current position on tape and the program.

finite, discrete and deterministic instructions, that is, without infinite, continous or stochastic processes. Procedures hence consume a value as *input* and produce a value as *output*. These values are finite in the sense that they can be represented as a finite sequence of bits. No input or output values

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

can in this sense be represented by empty sequences. If conversely, multiple values are to serve as either input or output, a simple value composition scheme can be devised:

Suppose a binary digit representation of all values. Then the longest possible value in the value composition will be of some finite length. All other values can be padded from the left with 0's to fit that length, and the sequence can begin with an additional number (of same width) indicating the number of elements in the composite.

How do we know how wide the first number is?

They operate on finite values, and the finiteness of values, individual instructions and the instruction sequence means that the effective procedure eventually halts or goes into an infinite loop for any given input.

The instruction sequence, is by it's sequential nature enumerable. Initial program execution would be to iterate through the sequence in a particular direction, e.g. top-down. In this process we implicitly make use of a "program counter", which is a value that points to the index of the next instruction to execute. When the program counter moves past the end of the instruction sequence, the procedure halts.

In this model, a "loop" is constructable by having a special *jump* instruction that changes the program counter to some value less than the current program counter.

A "loop" can be constructed by a sequence of such instructions, due it's sequential nature being enumerable, and with the existance of a special *jump* instruction, that changes the

Clearly, a procedure that eventually halts and returns the correct output for every input in the problem domain, solves the problem.

Procedures take in an input value and produce an output value. Multiple values can be represented as a single value via. a pairing function.

1.1 Computation environment

Finitely many different instructions

A procedure is a finite length sequence of instructions, i.e. a countable set that can be enumerated.

Every instruction is discrete, as in there are no continuous processes.

Every instruction is deterministic, as in there are no stochastic processes.

1.2 Effectiveness

Effective procedure

Effectively enumerable

Effectively decidable

Recursively enumerable – countable sets

Co-recursively enumerable

1.3 The halting problem

1.4 Rice's statement

Chapter 2

Language

2.1 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.

The language `D` is *initially* a purely functional, first-order, explicitly typed, call-by-value language.

Programs are defined as follows:

$$\texttt{<program>} ::= \texttt{<type>}^{*}\ \texttt{<function>}^{*}\ \texttt{<expression>} \tag{2.1}$$

Types are declared by means of infinite algebraic datatypes:

$$\begin{aligned}
\texttt{<type>} &::= \texttt{<type-name>}\ \texttt{`:='}\ \texttt{<constructor>}\ (\texttt{`|'}\ \texttt{<constructor>})^{*} && (2.2)\\
\texttt{<type-name>} &::= \texttt{<literal>} && (2.3)\\
\texttt{<constructor>} &::= \texttt{<constructor-name>}\ (\texttt{`of'}\ \texttt{<type-name>})? && (2.4)\\
\texttt{<constructor-name>} &::= \texttt{<literal>} && (2.5)
\end{aligned}$$

Functions (and their arguments) are explicitly typed, consume at least one argument and always yield a value as output:

$$\begin{aligned}
\texttt{<function>} &::= \texttt{<type-name>}\ \texttt{<function-name>}\ \texttt{<argument>}^{+}\ \texttt{`:='}\ \texttt{<expression>} && (2.6)\\
\texttt{<function-name>} &::= \texttt{<literal>} && (2.7)\\
\texttt{<argument>} &::= \texttt{<type-name>}\ \texttt{<argument-name>} && (2.8)\\
\texttt{<argument-name>} &::= \texttt{<literal>} && (2.9)
\end{aligned}$$

All of the $\texttt{<}x\texttt{-name>}$ declarations above make use of the `<literal>` nonterminal. For simplicity we'll let this be a lowercased dash-separated sequence of a-z characters:

$$\texttt{<literal>} ::= [\texttt{`a'-`z'}]\ (\ [\texttt{`-'}\ \texttt{`a'-`z'}]^{*}\ [\texttt{`a'-`z'}]\ )? \tag{2.10}$$

It is important that the above $\texttt{<}x\texttt{-name>}$s are distinct from one another. We'll avoid cluttering up the grammar and simply state the following:

[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.

$$\texttt{<type-name>} \cap \texttt{<constructor-name>} \cap \texttt{<function-name>} \cap \texttt{<argument-name>} \equiv \emptyset$$

Last but not least, the `<expression>` itself:

$$\texttt{<expression>} ::= \texttt{<function-call>} \mid \texttt{<constructor-call>} \tag{2.11}$$

$$\texttt{<function-call>} ::= \texttt{<function-name>}\ \text{‘}\texttt{(}\text{’}\ \texttt{<expression>}^{+}\ \text{‘}\texttt{)}\text{’} \tag{2.12}$$

$$\texttt{<constructor-call>} ::= \texttt{<constructor-name>}\ (\text{‘}\texttt{(}\text{’}\ \texttt{<constructor-call>}\ \text{‘}\texttt{)}\text{’})? \tag{2.13}$$

Since this is an eager language, the number of expressions that a function call takes must be the exact number of arguments defined for that function, and due to it being typed, their types must correspond.

If this definition should otherwise prove impractical in further affairs, such as lack of basic types or basic functions, these will be introduced as needed instead of them being included in the core grammar.

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.

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
$\mid$	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.