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
- Discrete mathematics

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 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 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

D

For the purposes of describing size-change termination we'll consider a language `D`. The following chapter describes the syntax and semantics of the language.

3.1 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.

3.2 Data

D is a simple language where the emphasis is on the sizes of data. Hence, the way that data values are constructed does not have to be particularly practical, but all values have to be well founded and easily comparable.

The language D is untyped, and represents all data in terms of *unlabeled ordered binary trees*, henceforth referred to as simply, *binary trees*. Such a tree is recursively defined as follows:

Definition 3. *A binary tree is a set that is either empty, heneceforth refferred to as a leaf or simply* 0*, or contains a single unlabeled node with two binary trees as it's left and right child, henceforth simply reffered to as a node. We'll refer to the set of all possible values in D as* $\mathbb{B}$*.*

To operate on such trees we'll require a few primitives. Namely, a representation of leafs, recursive construction and destruction of nodes, as well as a way to tell nodes and leafs apart. Most of these will be derived in the operational semantics of D[1], however, they will make use of the following primitive function:

Definition 4. *The function* $\cdot : \mathbb{B} \times \mathbb{B} \to \mathbb{B}$ *constructs a node with the two arguments as it's left and right child, respectively. We'll refer to this function, as well as the operator* $\cdot$ *in general, as "cons".*

[1] See § 3.4 (6) .

3.3 Syntax

We describe the syntax of D in terms of an extended Backus-Naur form[2]. This is a core syntax definition, and other, more practical, syntactical features may be defined later on as needed. The initial non-terminal is `<program>`.

$$\texttt{<program>} ::= \texttt{<declaration>}^* \texttt{<expression>} \tag{3.1}$$
$$\texttt{<expression>} ::= \texttt{<element>} \; (\; \texttt{`.'} \; \texttt{<expression>} \;) \; ? \tag{3.2}$$
$$\texttt{<element>} ::= \texttt{`0'} \mid \texttt{`('} \; \texttt{<element>} \; \texttt{`)'} \mid \texttt{<name>} \mid \texttt{<application>} \tag{3.3}$$
$$\texttt{<application>} ::= \texttt{<name>} \; \texttt{<expression>}^+ \tag{3.4}$$
$$\texttt{<declaration>} ::= \texttt{<name>} \; \texttt{<pattern>}^+ \; \texttt{`:='} \; \texttt{<expression>} \tag{3.5}$$
$$\texttt{<pattern>} ::= \texttt{<pattern-value>} \; (\; \texttt{`.'} \; \texttt{<pattern>} \;) \; ? \tag{3.6}$$
$$\texttt{<pattern-value>} ::= \texttt{`0'} \mid \texttt{`\_'} \mid \texttt{`('} \; \texttt{<pattern>} \; \texttt{`)'} \mid \texttt{<name>} \tag{3.7}$$
$$\texttt{<name>} ::= [\texttt{`a'-`z'}] \; (\; [\texttt{`-'} \; \texttt{`a'-`z'}]^* \; [\texttt{`a'-`z'}] \;)? \tag{3.8}$$

0-ary declarations are disallowed to avoid having to deal with constants in general.

The term `'_'` in `<pattern-value>` is the conventional wildcard operator; it indicates a value that won't used by the declaration, but allows us to keep the same declaration signature. We hence define the *signature* of a declaration as follows:

Definition 5. *A declaration signature in D consists of the function name and the number of parameters it has.*

We'll adopt the Erlang-like notation when talking about function signatures, i.e. if we have a function `less` having two arguments in it's signature, we'll refer to it as `less/2`.

3.4 Semantics

Revise the context of an expression within a function call, it should always be the context upon entering the function call! Or even better, the context when the function was defined!

Perhaps pattern matching must be exhaustive in general.

Every subsequent definition must be strictly less specific than the former.

In the following section we describe the semantics of D using a form of structured operational semantics. The syntax used to define the reduction rules is largely equivalent to the Aarhus report**[?]**, but differs slightly[3].

The syntax aside, Table 3.1 (7) defines a few shorthands for various constructs in various contexts. We'll use these both when talking about the semantics as well as in futher proofs. Additionally, we'll let the atoms 0 and _ represent themselves in the reduction rules.

3.4.1 The memory model

Memory is considered in terms of a set of value stacks, σ. Every stack has a unique identifier $n \in \mathbb{N}$, that is, each variable gets a value stack. This renders σ countably infinite since $\mathbb{N}$ is countably infinite.

As we enter a new scope, we bind a variable to a value, that is, we push that value on top of the corresponding stack. We pop the value off the corresponding stack as we leave the scope at the entry to which the variable was bound.

An expression at a certain scope depth only has access to variables at the same scope depth. This is to ensure static scope. We won't adhere to this problem explicitly in the semantics, but instead ask you to simply keep it in mind.

[2]The extension lends some constructs from regular expressions to achieve a more concise dialect. The extension is described in detail in Appendix A.1 (17) .

[3]The syntax applied here is described in further detail in Appendix A.2 (18) .

Description	I	P	A
Expression	x	X	$\mathbb{X}$
Element (of an expression)	e	E	$\mathbb{E}$
Pattern	p	P	$\mathbb{P}$
Value(binary tree)	b	B	$\mathbb{B}$
Name	n	N	$\mathbb{N}$

Table 3.1: Overview of some of the shorthands used in this text. The column **A** refers to all possible instances of the given construct, i.e. $\mathbb{B}$ reffers to all constructable values in D. The column **P** refers to all the instances of the given construct in a given program, i.e. N reffers to all the names in a given program. The column **I** reffers to specific instances of the given constructs, i.e. x reffers to a particular expression.

Functions and variables

Due to D being a first-order language, we should make sure to separate the function and variable spaces. We'll represent these by ϕ and γ, respectively.

Whenever we use σ, ϕ or γ in set notation, we imply the sets of the names of functions and variables, and not the stacks themselves corresponding to those names. Hence, $\sigma = \phi \cup \gamma$, and to keep D first-order we add the limitation that $\phi \cap \gamma = \emptyset$.

Making D higher order

The only change that this would require is to let $\phi = \gamma = \sigma$.

3.4.2 Function declarations

Assuming that as a part of the semantic analysis all `<declaration>` with the same name are grouped into the set $\langle nF \rangle$

A declaration with a name n, a *non-empty* pattern list P and an expression e is stored in the function space ϕ:

$$\frac{\langle \phi(n) \mapsto \langle P, x, \phi \rangle \rangle \to \phi'}{\langle n, P, x, \phi \rangle \to \phi'} \tag{3.9}$$

3.4.3 Expression evaluation

An expression x is either the element e, or a construction of an element e_1 with another expression x_1. That is, the binary infix operator $\cdot$ is right-associative, and has the following operational semantics:

$$\frac{\langle \textsc{Single}, x, \sigma \rangle \to \langle v, \sigma \rangle \vee \langle \textsc{Chain}, x, \sigma \rangle \to \langle v, \sigma \rangle}{\langle x, \sigma \rangle \to \langle v, \sigma \rangle} \tag{3.10}$$

$$\frac{x \to e \wedge \langle e, \sigma \rangle \to \langle v, \sigma \rangle}{\langle \textsc{Single}, x, \sigma \rangle \to \langle v, \sigma \rangle} \tag{3.11}$$

$$\frac{x \Rightarrow e_1 \cdot x_1 \wedge \langle e_1, \sigma \rangle \to \langle v_1, \sigma \rangle \wedge \langle x_1, \sigma \rangle \to \langle v_2, \sigma \rangle}{\langle \textsc{Chain}, x, \sigma \rangle \to \langle v, \sigma \rangle} \quad (\text{where } v_1 \cdot v_2 = v) \tag{3.12}$$

3.4.4 Element evaluation

According to the syntax specification, an element of an expression can either be the atom 0, or an application. We'd like to distinguish between variables and functions, and we do that

$$\frac{(e \Rightarrow 0 \wedge v \equiv 0) \vee \dfrac{e \Rightarrow n}{\beta(n) \Rightarrow v} \vee \dfrac{e \Rightarrow \langle n, X \rangle}{\langle n, X, \sigma \rangle \Rightarrow \langle v, \sigma \rangle}}{\langle e, \sigma \rangle \Rightarrow \langle v, \sigma \rangle} \tag{3.13}$$

3.4.5 Function application

$$\frac{\frac{\frac{\langle n, \phi\rangle \Rightarrow \langle P, x, \phi\rangle}{\langle P, X, \sigma\rangle \Rightarrow \sigma'}}{\langle x, \sigma'\rangle \Rightarrow \langle v, \sigma'\rangle}}{\langle n, X, \sigma\rangle \Rightarrow \langle v, \sigma\rangle} \tag{3.14}$$

3.4.6 Pattern matching

$$\frac{\frac{\langle P_{head}, X_{head}, \sigma\rangle \Rightarrow \sigma''}{\langle P_{tail}, X_{tail}, \sigma''\rangle \Rightarrow \sigma'}}{\langle P, X, \sigma\rangle \Rightarrow \sigma'} \tag{3.15}$$

$$\frac{\langle \mathrm{I}, p, x, \sigma\rangle \Rightarrow \langle p', x', \sigma'\rangle \vee \langle \mathrm{Z}, p, x, \sigma\rangle \Rightarrow \langle p', x', \sigma'\rangle \vee \langle \mathrm{N}, p, x, \sigma\rangle \Rightarrow \langle p', x', \sigma'\rangle \vee \langle \mathrm{P}, p, x, \sigma\rangle \Rightarrow \langle p', x', \sigma'\rangle}{\langle p, x, \sigma\rangle \Rightarrow \langle p', x', \sigma'\rangle} \tag{3.16}$$

For the sake of an elegant notation, we'll override the function $\cdot$ for patterns.

Definition 6. *A pattern is an unlabeled of binary tree which is either empty or consists of an unlabeled node with a 0, _, name, or a pattern as it's left and right child.*

Definition 7. *Let the set of all possible patterns be denoted by* $\mathbb{P}$.

Definition 8. *The function* $\cdot : \mathbb{P} \times \mathbb{P} \to \mathbb{P}$ *constructs a pattern node with the two arguments as it's left and right child, respectively.*

$$\frac{p \Rightarrow \_ \cdot p' \wedge x \Rightarrow e \cdot x' \wedge \sigma \Rightarrow \sigma'}{\langle \textsc{Underscore}, p, x, \sigma\rangle \Rightarrow \langle p', x', \sigma'\rangle} \tag{3.17}$$

$$\frac{p \Rightarrow 0 \cdot p' \wedge x \Rightarrow e \cdot x' \wedge \sigma \Rightarrow \sigma'}{\langle \textsc{Zero}, p, x, \sigma\rangle \Rightarrow \langle p', x', \sigma'\rangle} \tag{3.18}$$

$$\frac{\frac{\frac{p \Rightarrow n \cdot p' \wedge x \Rightarrow e \cdot x'}{\langle e, \sigma\rangle \Rightarrow \langle v, \sigma\rangle}}{\langle \sigma(n) \leftarrow v\rangle \Rightarrow \sigma'}}{\langle \textsc{Name}, p, x, \sigma\rangle \Rightarrow \langle p', x', \sigma'\rangle} \tag{3.19}$$

$$\frac{\frac{p \Rightarrow p'' \cdot p' \wedge x \Rightarrow x'' \cdot x'}{\langle p'', x'', \sigma\rangle \Rightarrow \sigma'}}{\langle \textsc{Pattern}, p, x, \sigma\rangle \Rightarrow \langle p', x', \sigma'\rangle} \tag{3.20}$$

3.5 User input

To be able to write more interesting programs, we'll define the primitive function `input/0` that can yield literally any valid D value.

3.6 Size

For the purposes of talking about size-change termination, we also need to define the notion of size, and be sure to do so in such a way so that all possible data values are well-founded.

Definition 9. *Size of a value in D is the number of nodes in the tree representing that value.*

The "well-foundedness" of D's data values, given such a definition can be argued for by proving a bijective relation between $\mathbb{B}$ and $\mathbb{N}$. This would imply that we can define the relation $<$ on D's data values, which we know to be well-founded.

We start by formally proving that Definition 9 (8) yields a many-to-one mapping of D's data values to the natural numbers.

First, we prove, by induction, that any natural number can be represented in D:

Proof.

Base The atom 0 has no nodes, and hence represents the value 0.

Assumption If we can represent the $n \in \mathbb{N}$ in D, then we can also represent the number $n + 1 \in \mathbb{N}$.

Induction Let n be represented by some binary tree A, then $n + 1$ can be represented by $0 \cdot A$.

□

Second, we prove, also by induction, that any value in D has one and only one representation in $\mathbb{N}$.

Proof.

Base The atom 0 has no nodes, and hence corresponds only to the value 0.

Assumption

1. If the binary tree A has only one representation $n \in \mathbb{N}$, then $|0 \cdot A| \equiv n + 1$ and $|A \cdot 0| \equiv n + 1$.
2. If the binary tree A has only one representation $n \in \mathbb{N}$, and the binary tree B has only one representation $m \in \mathbb{N}$, then $|A \cdot B| \equiv n + m + 1$ and $|B \cdot A| \equiv n + m + 1$.

Induction By definition of the binary function $\cdot$, any given node A with left child A_{left} and right child A_{right} has the size:

$$|A| = 1 + \left|A_{left}\right| + \left|A_{right}\right|$$

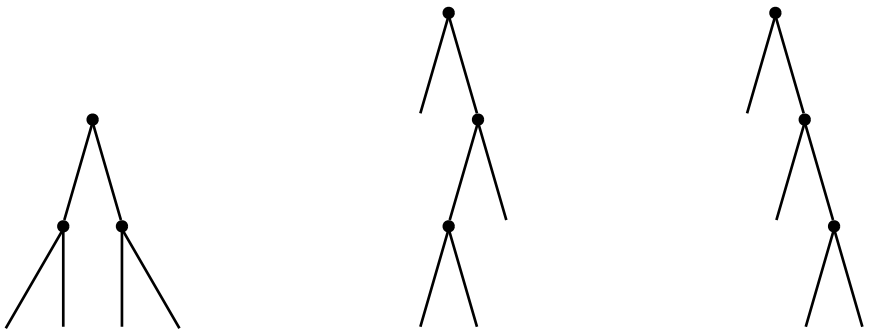
Hence, any value in D must have one and only one representation in $\mathbb{N}$.

□

Definition 9 (8) *almost* allows us to devise an algorithm to compare the sizes of data values. The problem withstanding is that two different values can have rather diverging tree representations. Hence, comparing them, using only the operations defined in § 3.4 (6) , is seemingly impossible unless we initially, or along the way, transform the binary trees being compared into some sort of a *standard representation*. We'll define this representation, recursively, as follows:

Definition 10. *A binary tree in standard representation is a binary tree that either is a leaf or a node having a leaf as it's left child and a binary tree in standard representation as it's right child.*

Intuitively, a binary tree in standard representation is just a tree that only descends along the right side. Comparing the sizes of two trees in this representation is just a matter of walking the descending in the two trees simultaneously, until one of them, or both, bottom out. If there is a tree that bottoms out strictly before another, that is the lesser tree by Definition 9 (8) . Figure 3.1 (10) showcases some examples.



(a) Not in standard representation (b) Not in standard representation (c) Standard representation

Figure 3.1: Three trees of various shapes but equal size.

3.6.1 `normalize/1`

```
normalize a = normalize-aux a 0 0

normalize-aux 0     0     an = an
normalize-aux 0     bl.br an = normalize-aux bl br    an
normalize-aux 0.ar  b     an = normalize-aux ar b     0.an
normalize-aux al.0  b     an = normalize-aux al b     0.an
normalize-aux al.ar b     an = normalize-aux ar al.b  0.an
```

Correctness

`normalize/1` makes use of an auxiliary procedure, `normalize-aux/3`, for which we can provide the following argument descriptions:

1. The tree to be normalized.
2. An auxiliary tree.
3. A normalized tree.

The idea of the algorithm is to move right-wise down the tree to be normalized, constructing an auxiliary tree containing all left-wise child nodes, if any.

The return value is the normalized tree, i.e. the third argument. Hence, we must increase the size of the normalized tree each time we move right-wise down the tree to be normalized.

Once we reach the right-most leaf of the tree to be normalized we return the normalized tree if the auxiliary tree is empty. Otherwise, we normalize the right child of the auxilary tree, with the left child of the auxiliary as the new auxiliary tree, and the normalized tree constructed thus far as the initial normalized tree.

Time complexity

Coming soon..

Space complexity

Coming soon..

3.6.2 `less/2`

We'll define the function `normalize/1` further below to transform any D value into it's standard representation. For now we'll assume that we have such a function in scope and define `less/2` for determining whether the value of the first argument is strictly less than the value of the second argument.

In order to define such a boolean-valued function we need a convention for representing the boolean values *true* and $false$ in D. We'll adopt the C-like convention:

Definition 11. *A $false$ value is represented by a leaf tree. A true value is represented by a non-leaf tree, i.e. a node.*

We're now ready to define the function `less/2`:

```
less a b := normalized-less (normalize a) (normalize b)

normalized-less 0 b := b
normalized-less _ 0 := 0
normalized-less _.a _.b := normalized-less a b
```

Correctness

Proof. Given Definition 10 (9) , and the assumption that NORMALIZE(A) computes the standard representation of A, we know the following:

1. $|A| \equiv |\text{NORMALIZE}(A)|$.
2. We'll walk through all the nodes if we perform a recursive right-child-walk starting at A.
3. The same holds for B.

It is also easy to see from lines **??**:**??** that NORMALIZEDLESS stops as soon as we reach the "bottom" of either A or B.

Given Definition 9 (8) , $A < B$ iff it bottoms out before B, that is, we reach an instance of the recursion where both $IsLeaf(A)$ and $IsNode(B)$ hold. In all other cases $A \geq B$, the cases specifically are:

- $IsLeaf(A)$ and $IsLeaf(B)$, then $|A| \equiv |B|$.
- $IsNode(A)$ and $IsLeaf(B)$, then $|A| > |B|$

Last but not least, due to all data values being finite, eventually one of the trees does bottom out. □

Time complexity

Given that the binary trees A and B are in standard representation when we enter the auxiliary procedure, NORMALIZEDLESS, it is fairly easy to get an upper bound on the running time of NORMALIZEDLESS itself.

Indeed, the running time of NORMALIZEDLESS itself is $O(\text{MAX}(|A|, |B|))$, since we just walk down the trees until one of them bottoms out.

We haven't yet defined the procedure NORMALIZE yet. Hence, the only thing that we can say about the running time of LESS in general is that it is $O(\text{NORMALIZE}(A) + \text{NORMALIZE}(B) + \text{MAX}(|A|, |B|))$.

Space complexity

Coming soon..

3.7 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.

3.8 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`:
Assume that the argument is

```
fibonacci n = fibonacci-aux (normalize n) 0 0

fibonacci-aux 0 x y := 0
fibonacci-aux 0.0 x y := y
fibonacci-aux 0.n x y := fibonacci-aux n y (add x y)
```

Note: The return value is not normalized.

Chapter 4

The size-change termination principle

The size-change termination analysis builds upon the idea of flow analysis of programs. In general, flow analysis aimes to answer the question, "What can we say about a given point in a program without regard to the execution path taken to that point?". A "point" in a computer program is in this case a primitive operation such as an assignment, or a conditional branch, etc.

The idea is then to construct a graph where such points are nodes, and the arcs in between them represent a transfer of control between the primitive operations, that would otherwise occur under the execution of the program. For instance, there is a transition to the first operation of a true or false branch all depending on the value of the forthcoming branch condition. Such graphs are hence reffered to as *control flow graphs*.

With such a graph at hand, various optimization algorithms can be devised to traverse the graph and deduce certain proprties, such as reoccuring primitive operations on otherwise static variables[**?**], etc.

As an example, consider the program in Listing 4.1 (13) . Figure 4.1 (13) represents the control flow graph for this program.

Listing 4.1: A simple, down-counting loop in D.

```
1 f 0 := 0
2 f x._ := f x
3
4 f input
```

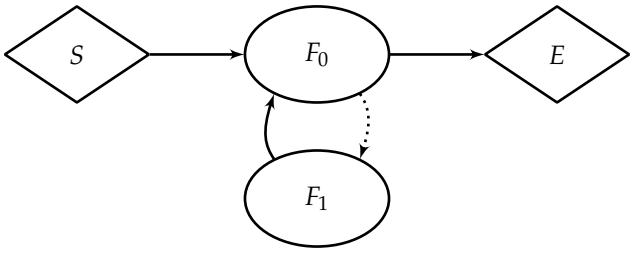


Figure 4.1: A control flow graph for the program defined in Listing 4.1 (13) .

The notation used here is unconventional and tailored for describing programs written in D. Refer to Appendix A.3 (20) for a detailed explanation of the notation. There are however, a few things worth about the notation for further discussion.

Every case of every function declaration gets a node and a program-wide unique identifier. The identifier is the initial function name followed by it's index among the cases of that particular function, enumerated top-to-bottom, starting with 0.

Each such "case node", except the last one, has two outgoing edges, the true edge, marked with a filled line, and a false edge, marked with a dotted line. The very last case has only one outgoing edge

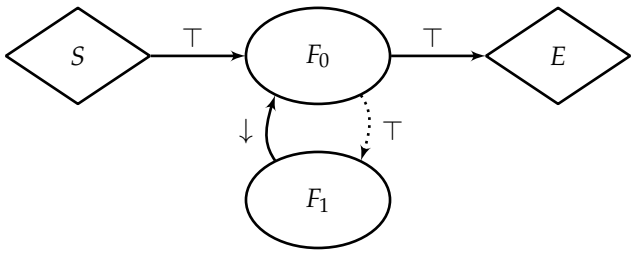


Figure 4.2: An edge-labelled control flow graph for the program defined in Listing 4.1 (13) . Transitions that cause a strict decrease in some value are labelled ↓. False transitions from declaration cases (dotted edges) never change any values.

– the true edge. This is because the last case *cannot* fail by the definition of D, where cases must be in order of strictly decreasing specificity and be exhaustive in total.

Hence, when talking back and forth between listings and control flow graphs it may proove useful to label the cases of a function declaration in the listing as well. Listing 4.2 (14) corresponds to a labelled version of Listing 4.1 (13) .

Listing 4.2: A labelled version of Listing 4.1 (13) .

```
1 F0: f 0 := 0
2 F1: f x._ := f x
3
4 f input
```

The size-change termination principle can be stated in the following way: *the program terminates, if every infinite control flow sequence causes an infinite descent in a well-founded data value*[**?**]. Graphically, a cycle in the control flow graph is an infinite control flow sequence.

The notation used in Figure 4.1 (13) indicates that the transition from $F_0 \to E$ is chosen over the transition $F_0 \to F_1$ as soon as the pattern of F_0 matches. What's unclear from the figure is whether a value initially not matching F_0's pattern is ever mutated enough to do so by the infinite control flow sequence $F_0 \to F_1 \to F_0$.

A derrivable property from the program text is that the value sent as the argument in the transition $F_1 \to F_0$ is *always* strictly less than the value sent as the argument in the preceeding transition, $F_0 \to F_1$. Additionally, we can universally state that *false transitions from case nodes by definition can't change any values*. Hence, the infinite control flow sequence $F_0 \to F_1 \to F_0$ strictly decreases a well founded value in every iteration of the control flow cycle. This means that eventually the value bottoms out at 0 which would validate the F_0 pattern and lead the program to terminate. In general, we can state the following:

Lemma 0.1. *An infinite control flow sequnece infinitely decreases a well founded value if all the transitions in the corresponding control flow graph cycle either strictly decrease a value or keep all values unchanged, and at least one transition strictcly decreases a value.*

Figure 4.2 (14) showcases an edge-labelled control flow graph for Listing 4.1 (13) . Transitions for which we can *safely* say that a value is decreased we mark with the symbol ↓, transitions for which we can *safely* say that no value is increased we mark with the symbol ⊤. The graph is rather verbose and in future graphs we'll omit ⊤ labels for nodes that universally can't increase a value, i.e. $S \to F_0$, $F_0 \to E$ and $F_0 \to F_1$.

We can further proove the soundness of the size-change termination principle.

Proof. Assume for the sake of contradiction that there exists an infinite control flow sequence that infinitely decreases a well-founded data value. Assume furthermore that the program does not terminate. Then, some variable has to decrease indefinately, which contradicts with the definition of well-founded values. □

Unfortunately, this particular program can be trivially unfolded into a loop program, so the point of using size-change termination analysis may seem superflous. Indeed, the only change wrt. the control

flow diagram is the notaiton used in the diagram. For such programs, the termination property is easily derrivable[**?**].

Note, that the control is not backpropagated in Figure 4.2 (14) , as if tail recursion is superimposed. In general we won't bother with backpropagation of control as that is of not particular interest to us. What is of interest is "how deep the rabithole goes" before we backpropagate. This is because if a function call terminates than so does the expression containing it due to the way D is defined. Hence, the control flow graphs presented in this text might just as well be considered as abstract call graphs, where the concrete arguments of the function calls are abstracted away, and we merely consider how they change from call to call. Indeed, the problem of termination analysis can be rephrased as the problem of determining whether the static call graph of a program, i.e. the one containing the concrete function arguments, is finite.

Let us a consider a more complex program, one that can't be unfolded into a loop program, namely the `reverse` program from § 3.8 (12) .

Listing 4.3: A simple, down-counting loop in D.

```
1 R0: reverse 0 := 0
2 R1: reverse left.right := (reverse right).(reverse left)
3
4 reverse input
```

Appendix A

Notation

The following appendix describes the notation used throughout this text for various concepts.

A.1 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[**?**].

A.1.1 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.1.2 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.1.3 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.

A.2 The structured operational semantics used in this work

The following section describes the syntax used in this text to describe the operational semantics of the language D. The syntax is inspired by [**?**], but differs slightly.

A.2.1 Some general properties

- Rules should be read in increasing order of equation number.
- If some rule with a lower equation number makes use of an undefined reduction rule, it is because the reduction rule is defined under some higher equation number.
- Rules can be defined in terms of themselves, i.e. they can be recursive, even mutually recursive.

A.2.2 Atoms

To keep the rules clear and concise we'll make use of atoms to subdivide a rule into subrules and distinguish those rules from the rest. If you're familiar with Prolog, this shouldn't be particularly new to you.

For instance, a chained expression x may have the following semantics:

$$\frac{\langle \textsc{Single}, x, \sigma\rangle \to \langle v, \sigma\rangle \vee \langle \textsc{Chain}, x, \sigma\rangle \to \langle v, \sigma\rangle}{\langle x, \sigma\rangle \to \langle v, \sigma\rangle} \tag{A.1}$$

This means that either the rule corresponding to the single element expression ($\langle \textsc{Single}, x, \sigma\rangle \to \langle v, \sigma\rangle$) validates, or the rule corresponding to the element followed by another expression ($\langle \textsc{Chain}, x, \sigma\rangle \to \langle v, \sigma\rangle$) does.

Atoms are used in both propositions and conclusions of rules. For instance, A.2 defines one of the subrules to the above rule.

A.2.3 The proposition operators

The $\Rightarrow$ operator

The notation used in [**?**] does not make use of atoms[1], but instead leaves the reader stranded guessing which rule to apply next. This is derivable from the language syntax, so usually this is isn't a problem. For instance, if an expression is either an if-statement or a while-loop we wouldn't find a summoning rule for expressions, but rather "orphan rules" like the following:

$$\frac{\cdots}{\langle \textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2, \sigma\rangle \longrightarrow \cdots}$$

$$\frac{\cdots}{\langle \textbf{while } e \textbf{ do } c, \sigma\rangle \longrightarrow \cdots}$$

In the notation used in this text we define a summoning rule first, such as A.1, and use atoms to subdivide that rule into subrules. The subrules are then defined further down, such as A.2. However, we still need a way to distinguish between things like if-statements and for-loops, or in the case of the running example elements and expressions.

Hence, the first part of the proposition of a subrule will often begin with a "rule" that uses the $\Rightarrow$ operator. For instance, $x \Rightarrow e$ means that the expression x that we're considering really is just a single element, or $x \Rightarrow e \cdot x'$ means that the expression x that we're considering really is a construction of an element e and some other expression x'.

The $\rightarrow$ operator

[**?**] uses the operator $\longrightarrow$ to indicate a transition. Since we will blend this operator with other binary operators like $\wedge$ and $\vee$, and wish for the transition to have higher precedence[2], it is visually more appropriate to use the $\rightarrow$ operator, since that keeps the vertical space between the operators roughly the same as between the operators $\wedge$ and $\vee$.

The $\wedge$ operator

The $\wedge$ operator is used as a conventional *and* operator to combine multiple rules that must hold in a proposition. The left-to-right evaluation order is superimposed on the binary operator such that the ending values of the left hand rule can be used in the right hand rule. For instance, in the following rule, the value e resulting from validating the left side of the $\wedge$ operator is carried over to the right side of the operator and used in another rule.

$$\frac{x \Rightarrow e \wedge \langle e, \sigma\rangle \rightarrow \langle v, \sigma\rangle}{\langle \textsc{Single}, x, \sigma\rangle \rightarrow \langle v, \sigma\rangle} \tag{A.2}$$

The $\vee$ operator

The $\vee$ operator is used as a conventional short-circuited *or* operator. That is, a left-to-right evaluation order is also superimposed but evaluation stops as soon as one of the operands holds.

Operator precedence

To avoid ambiguity, and having to surcome to using parentheses we'll define the precedences of the possible operators in the prepositions of rules. Elements with higher precedence are hence considered first.

1. $\vee$
2. $\wedge$

[1] See Appendix A.2.2 (18) .
[2] See Appendix A.2.3 (19) .

3. $\rightarrow$

A.3 Control flow graphs

A.3.1 The starting and ending point

These are denoted by diamonds with the letter S and E respectively. Every program has a starting point, but not every program may have an ending point, indeed a non-terminating program may either not have an ending point or that point may be otherwise unreachable.

A.3.2 Cases of a function declaration

A full function declaration in D is a series of uniformly named and arried function declarations where each declaration represents a case