Chapter 1

Introduction

1.1 Motivation

The generic halting problem, or the *Entscheidungsproblem*, was formulated well before the invention of the modern computer. It was formulated at a time when many mathematicians believed that they could formalize all of mathematics and use algorithmic means to formally prove all statements within that formal system. The problem can be stated as follows:

Definition 1. *Given the set of all possible programs P, find a program $p \in P$, that can for any $p' \in P$, within a finite amount of time return* `halts` *or* `doesn't halt`*, depending on whether p' eventually stops or runs indefinitely, respectively.*

While the concept of a program remains to be formally defined, an important part of that definition is that it is a finite sequence of discrete, terminating steps. Hence, the problem can be restated as determining whether the given program contains program flow cycles that loop indefinitely.

Alan A. Turing and Alonzo D. Church developed separate proofs for the infeasibility of such a program almost simultaneously in 1937. Turing's proof however, would become the one more widely recognised, although they are mutually reduceable to one another.

However, the fact that termination checking is infeasible *in general*, has unfortunately become an easy excuse for many to claim that the property is *always* undecidable.

The motivation behind this project is to examine some of the contexts in which the halting property *is* decidable in a matter that is both sound and complete. To those unfamiliar with logic, a *sound* proof is a proof that produces the correct result for any query, and a *complete* proof is a proof that always terminates.

To do this for a generic program[1] we need to slightly relax the definition of the halting problem allowing for the answer `unknown` to be returned. The goal is then to reduce the number of programs in P for which the termination checking program returns the result `unknown`.

Definition 2. *Given the set of all possible programs P, find a program $p \in P$, that can for any $p' \in P$, within a finite amount of time, either give up and return* `unknown`*, or return* `halts` *or* `doesn't halt`*, depending on whether p' eventually stops or runs indefinitely, respectively. Find a p such that the number of $p' \in P$ for which p returns* `unknown` *is minimized.*

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 at the present state of writing, are subject to basic undergraduate courses in computer science. Furthermore, the reader is expected to be familiar with discrete mathematics and the

[1] A term that also remains to be formally defined.

basic concepts of functional programming languages. Ideally, the reader should be well familiar with at least one purely functional programming language such as ML or Haskell.

In summary, the following concepts are used without definition:

- Algorithm.
- Function, pattern matching, loop, recursion.
- Induction, variant, invariant.
- Big-O notation.
- Regular Expressions (`preg` syntax).
- Backus-Naur Form, structured operational semantics.
- Turing machine, the halting problem.
- List, head, tail.
- Basic discrete mathematics.
- Basic graph theory.

1.3 Chapter overview

Chapter 2

Chapter 3

Chapter 4

Chapter 5

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 3. *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 4. *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

The language Δ

For the purposes of describing size-change termination we'll consider a language Δ. 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

Δ 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 Δ 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 5. *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* Δ *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 Δ[1], however, they will make use of the following primitive function:

Definition 6. *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 Δ 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 7. *A declaration signature in Δ 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!

Allow mutual recursion

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 Δ 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 further 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 (23) .

[3]The syntax applied here is described in further detail in Appendix A.2 (24) .

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 Δ. 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 Δ 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 Δ first-order we add the limitation that $\phi \cap \gamma = \emptyset$.

Making Δ 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 8. *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 9. *Let the set of all possible patterns be denoted by* $\mathbb{P}$*.*

Definition 10. *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 Δ 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 11. *Size of a value in* Δ *is the number of nodes in the tree representing that value.*

The "well-foundedness" of Δ'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 Δ's data values, which we know to be well-founded.

We start by formally proving that Definition 11 (8) yields a many-to-one mapping of Δ's data values to the natural numbers.

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

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 Δ, 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 Δ 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:

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

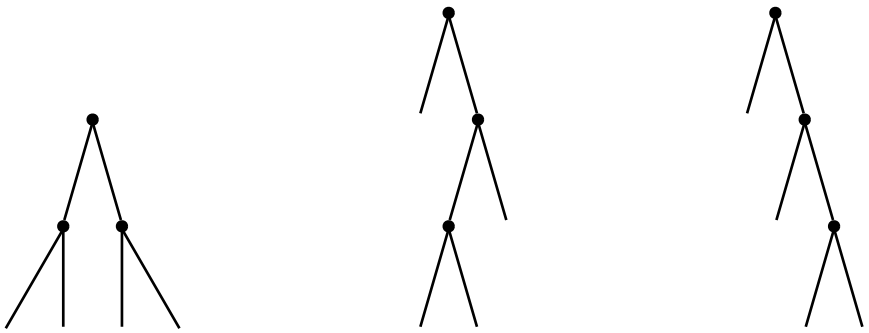
Hence, any value in Δ must have one and only one representation in $\mathbb{N}$.

□

Definition 11 (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 12. *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 11 (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 Δ 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 Δ. We'll adopt the C-like convention:

Definition 13. *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`:

Listing 3.1: A definition of the `less/2` function.

```
1 less a b := normalized-less (normalize a) (normalize b)
2
3 normalized-less 0 b := b
4 normalized-less _ 0 := 0
5 normalized-less _.a _.b := normalized-less a b
```

Correctness

Proof. Given Definition 12 (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 11 (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 $O\left(\text{MAX}\left(|A|, |B|\right)\right)$, 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 $O\left(\text{NORMALIZE}(A) + \text{NORMALIZE}(B) + \text{MAX}\left(|A|, |B|\right)\right)$.

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

Size-change termination

The size-change termination analysis builds upon the idea of flow analysis of programs. In general, flow analysis aims 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, a condition 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. Such a node may have variable in-degree and out-degree from one given primitive. For instance, a condition branch would usually have two possible transfers of control depending on the outcome of the condition. Hence, it serves useful to label arcs depending on when they are taken. The conditions should clearly not overlap to avoid non-determination.

Such graphs are referred to as *control flow graphs*.

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

4.1 Control flow graphs in Δ

4.1.1 Start and end nodes

Every control flow graph has a start and an end node. These nodes do not explicitly represent control primitives, but rather the start and end of a program. *A program cannot be started nor ended more than once*. The start node is labelled S and has out-degree 1 and in-degree 0. The end node is labelled E and has out-degree 0, but variable in-degree, i.e. a program can be ended in more than one way.

The control-flow graph for the empty program is hence:

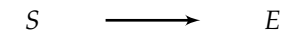


Figure 4.1: A control flow graph for the empty Δ program.

4.1.2 Function clauses

While node construction and destruction are primitive operations in Δ, we'll refrain ourselves from delving into such details in the control flow graphs of our programs. Indeed because *node construction and destruction always terminates*. Instead, we'll let a *function clause* define a point in a program.

The expression of the clause can thereafter make calls to its enclosing[1], or some other function. Such calls are represented by transfer of control, that is, arcs. Disregarding the cases where a function clause

[1] We say that a function *consists* of function clauses and a function clause is *enclosed* in a function.

expression makes multiple calls to the same function with different arguments, these arcs need not be disjunctively labelled since all of these transitions happen unconditionally as a result of evaluating the expression. More specifically, *we consider the order of evaluation to be insignificant*, and hence undeserving of labelling. We further discuss the reasons for this below.

If calls are separated by node construction, the order in which those calls are made is definitely insignificant. For instance, consider the expression `(f a).(g b)`, where `f` and `g` are some well-defined functions, `f` $\neq$ `g`, and `a` and `b` are some bound variables. It makes no difference to the final result which of the calls, `f a` and `g b`, is evaluated first. Indeed, they can be evaluated in parallel, and we would still get the same result. This is easy to see for any nested construction of results of function calls, as in e.g. `(f a).0.(g b)`.

On the other hand, the syntax and semantics of Δ allow for function calls to be nested as in e.g. the expression `(f (g a) (h b))`, where `h` is also some well-defined function and is pairwise unequal to `f` and `g`. While the order of evaluation of `g a` and `h b` is *insignificant* wrt. to one another, as with function calls separated by construction, the order of evaluation of these two subexpressions wrt. to the call to function `f`, *is significant to the result*, and *might* be significant to termination analysis in general. However, we'll regard this as insignificant for the time being for mere simplicity. We'll come back to the question of whether size-change termination analysis can benefit from regarding this as significant later on.

We can now draw a control flow graph for the program define in Listing 4.1 (14) as shown in Figure 4.2 (14) .

Listing 4.1: A sample Δ program, always returning `0.0.0`.

```
1 f x y := x.y
2 g _ := 0
3 h _ := 0
4 i x y := (f ((h y).(g x)) (h y))
5 i input input
```

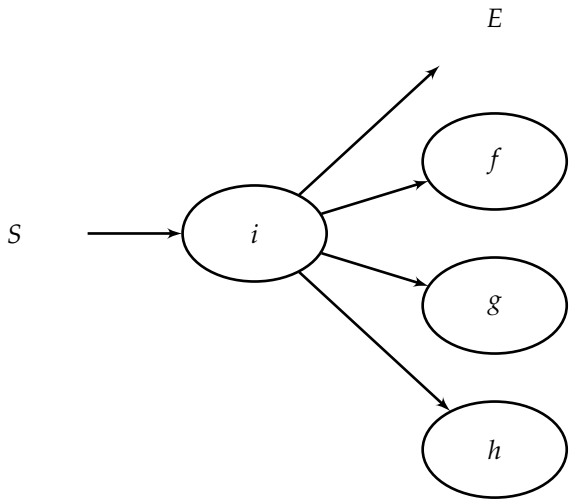


Figure 4.2: A control flow graph for the Δ program in Listing 4.1 (14) . The graph does not explicitly specify back-propagation of control, if any.

4.1.3 Call cycles

A call cycle occurs when there is a cyclical transition of control between the nodes of a control flow graph. I.e. when there is a cycle in the control flow graph.

Lemma 0.1. *We're concerned with call cycles in control flow graphs since non-termination cannot occur if not for an infinite control flow cycle.*

Proof. If a program has a control flow graph with no cycles and does not terminate, then one of the primitive operations, i.e. construction, destruction, comparison or binding, does not terminate, which is certainly absurd given the semantics of Δ. □

Call cycles in Δ can occur in recursive or mutually recursive function clauses.

We will henceforth refer to function clauses with recursive calls as *recursive clauses* and their counterparts, i.e. the base clauses of a function declaration, *terminal clauses*.

4.1.4 Disregarding back-propagation

It is worth noting that in Figure 4.2 (14) , the clauses that make no function calls have out-degree 0. Technically, these functions *do transfer control* – back to the callee. We may refer to this process as *back-propagation of control*. While considering back-propagation is seemingly important to a concept that bases itself on the changes in the sizes of the program values, we're only concerned with call cycles.

The thing with back-propagation is that forward-propagation after back-propagation of a call cannot occur due to the way Δ is defined. Hence, what we are really concerned with is, "how deep the rabbit hole goes", before we back-propagate, as back-propagation superimplies termination of the function we're back-propagating out of.

4.1.5 Dropping the start and end nodes

The disregard of the back-propagation of control forces us to either redefine the transition from the start node and the transitions to the end node. This is because neither of these transitions are ever back-propagated, while all other transitions *must be* back-propagated if the program terminates.

Alternatively, disregard of back-propagation allows us to drop these nodes completely and concentrate on the clauses and explicit calls within the clause expressions. Hence, start and end nodes will not appear in any further graphs.

4.1.6 Control flow graphs vs. abstract static call graphs

Disregard of back-propagation allows us to consider control flow graphs presented in this text as mere *abstract static call graphs*, henceforth referred to simply as, *call graphs*. The abstraction applied to these graphs compared to regular static call graphs is that the concrete arguments of the function calls are not considered, and we merely consider how these values can change in size from for a given function call. Interestingly, the problem of termination analysis can be rephrased as the problem of determining whether the regular static call graph of a program, i.e. the one containing the concrete function arguments, is finite.

4.1.7 Multiple calls to the same function

Up until now we've only regarded expressions that don't make calls to the same function with varying arguments. This is because these calls have to be disjunctively labelled for the purposes of our analysis, because the use of varying arguments *may* mean varying decrease (or increase), in values for the different calls within the expression. For this purpose we'll disjunctively label *all* the calls within an expression, if necessary, but remember that this has nothing to do with evaluation order as has been discussed above.

This allows us to draw a control flow graph, or equivalently, a call graph, for the program in Listing 4.2 (15) . Here, we've already disjunctively labelled all of the calls in the expressions. This call graph is drawn in Figure 4.3 (16) .

Listing 4.2: A sample Δ program, always returning `(0.x).(0.y)`, where `x` and `y` are arbitrary Δ values supplied by the user.

```
1 f x y := x.y
2 g x := 0.x
3 i x y := (0: f (1: g x) (2: g y))
```

```
4 i input input
```

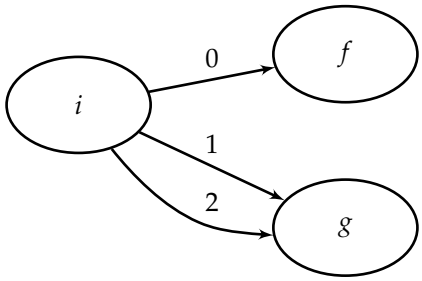


Figure 4.3: A control flow graph for the Δ program in Listing 4.2 (15) .

4.1.8 Multiple clauses

If function clauses are nodes, and the function calls within the expressions of the function clauses are unconditional transitions, what exactly happens if the arguments supplied to the function clause fail to match the pattern declaration for the clause?

The semantics of Δ tell us to make an unconditional transition to the immediately next clause of the function. There is at most one such transition for any clause, and the last clause of a function declaration cannot fail to pattern match[2].

We'll refer to these transitions as *fail transitions* and visually mark them with a dotted line rather than a filled line. We need this way of visually distinguishing fail transitions from the rest since they are conditionally different, in that for any clause with a fail transition, either the fail transition is chosen, or all the non-fail transitions are chosen simultaneously.

Before we can draw the call graph we also need a way to distinguish the clauses of a function wrt. the program text. We decide to enumerate the clauses top-to-bottom starting with 0. Sometimes we'll annotate the program text with these unique labels for each clause to make the call graph more readable.

Hence, we can now draw the call graph for the program defined in Listing 4.3 (16) as shown in Figure 4.4 (16) .

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

```
1 f0: f 0 := 0
2 f1: f x._ := f x
3
4 f input
```

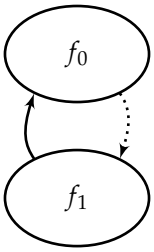


Figure 4.4: A control flow graph for the program defined in Listing 4.3 (16) .

For a more complex example, let's consider the call graph for the program `reverse` introduced in § 3.8 (12) . The program is repeated in annotated form in Listing 4.4 (17) , and its corresponding call graph is shown in Figure 4.5 (17) .

[2] See § 3.4 (6) .

Listing 4.4: An annotated version of the program `reverse` introduced in § 3.8 (12) .

```
r0: reverse 0 := 0
r1: reverse left.right := (0: reverse right).(1: reverse left)

reverse input
```

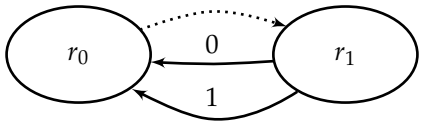


Figure 4.5: A control flow graph for the Δ program in Listing 4.4 (17) .

4.2 Size-change termination principle

Consider the program in Listing 4.3 (16) and it's corresponding call graph in Figure 4.4 (16) . Without any further information about the control transitions, the program seemingly loops indefinitely. However, there are some things that we can deduce about the control transitions.

Lemma 0.2. *If we can deduce for every control flow cycle in a prorgram that it reduces a value of well-founded data-type on each iteration of the cycle, then the value must eventually bottom out and the program must terminate.*

Proof. Assume for the sake of contradiction that a program that reduces a value of a well-founded data type in each call cycle does not terminate. Then, either the value reduces indefinitely, which is a contradiction to the well-foundedness of its data type, or some noncyclic call sequence causes an infinite loop, also an absurdity due to the definition of Δ. □

That is the *size-change termination principle*. All values in Δ are inherently well-founded so what remains to be shown is how we can deduce from a call cycle whether it reduces a value on each iteration.

Lemma 0.3. *A control flow cycle reduces a value on each iteration if at least one of the participating control transitions reduces the value and all other control transitions do not increase that value.*

Proof. If a value is not reduced in a cycle, it either stays the same or is increased. If it is increased, then at least one control transition must've increased the value, an absurdity. If it stays the same then none of the participating control transitions have neither increased nor decreased the value, also an absurdity. □

By the definition of call graphs, function clauses participate as nodes in a call cycle. A control transition is a directed edge between two function clauses where one clause is the *source* and the other is the *destination*.

We can analyze how a value changes it's size through a call sequence by analyzing the size relation between the variables bound in the source and the variables bound in the destination of every control transition.

Definition 14. *The relation* $\Upsilon : C_{caller} \times N_{caller} \times C_{callee} \times N_{callee} \to \{\bot, <, \leq\}$ *is defined to be the size relation between the caller* (C_{caller}) *and callee* (C_{callee}) *clauses in* Δ *where* N_{caller} *are the names of the variables bound in the caller, and* N_{callee} *are the names of the variables bound in the callee. Note, that we are only concerned with reductions and non-increases in size, all other relationships are marked by the no relationship symbol* ⊥*. Initially, the relationship between all the clauses and their variables is* ⊥*.*

The construction of the relationship Υ for a given transition depends first and foremost on whether that transition is a fail or success transition.

4.2.1 Fail transitions

A fail transition occurs if the values passed to a given clause to match it's pattern. If the values fail to match the pattern, no variables are bound and hence no change in values can occur. The values are simply passed along as they were to the next clause of the function declaration.

Lemma 0.4. *Fail transitions are transitive in the sense that the relationship between the variables bound in the source and the variables bound in the destination is the same regardless of the number of fail transitions in the path between the source and the destination.*

Proof. Follows from the semantics of Δ. □

Hence all fail transitions in the Υ relation return the result $\leq$ for all variable pairs.

Note, that due to Δ being first order and statically scoped, the variable space is always initially empty when a function clause begins pattern matching.

4.2.2 Success transitions

Since Δ is a call-by-value language, when a function call is encountered, the source evaluates the arguments of the function call and generates some *values* before giving up control.

The values may hence be a nested construction of some concrete values, values bound to variables in the source, and results of nested function calls. Without further regard of nested function calls, this implies that a *size relation* can be deduced between the variables bound in the source and the values that result from an evaluation of the function call arguments.

Of course, we cannot deduce a precise size displacement as the values of the bound variables may initially be *unknown* at compile time[3]. However, we can deduce a *safe* displacement estimate, such that it is less than or equal to the actual displacement in terms of absolute value. For instance, if the expression `a.b` appears as a function call argument, where `a` and `b` are some bound variables with unknown values, and this argument evaluates to some value v, then we can *safely* say that $v > \texttt{a}$, $v > \texttt{b}$, $v \geq \texttt{a.0}$ and $v \geq \texttt{0.b}$.

We decide to ignore the nested function calls because this would imply a more complex static analysis of the program. Specifically, we're unable to say anything about the result of the nested function call from the scope of the source clause alone. Instead, we treat results from nested function calls simply as variables with *unknown* values. We also make sure to keep these variables separate from the bound variables as there is no relationship to draw between these "variables" and the variables bound in the destination[4].

Continuing on with the example above, i.e. having the size relations $\{v > \texttt{a}, v > \texttt{b}, v \geq \texttt{a.0}, v \geq \texttt{0.b}\}$, assume that the destination clause has the corresponding pattern `x.y`. The question henceforth is how do we draw the relationship that $\texttt{a} \equiv \texttt{x}$ and $\texttt{b} \equiv \texttt{y}$, or perhaps simply that the control transition neither decreases nor increases any values. We can perform a corresponding analysis on the pattern declaration and deduce the set of conditions that will hold after pattern matching succeeds, indeed, $\{v > \texttt{x}, v > \texttt{y}, v \geq \texttt{x.0}, v \geq \texttt{0.y}\}$. The participation of x in the same kind of relations as a and the participation of y in the same kind of relations as b does not alone indicate their respective equivalence, since the actual property that $v \equiv \texttt{a.b}$ is lost.

On the other hand, if we had to formally define the relation that had to be built between the variables bound in the source and the values that the function call arguments evaluated to, this would be a relation between values and some kind of "abstract patterns".

To simplify the entire process, instead of deducing actual size relations between the variables bound in the source and the values that the function arguments evaluate to, we can simply turn the function argument into the abstract pattern to begin with. The actual size relations are hence withkept and can be deduced at a later stage in the process.

[3] Although some values can be deduced via static analysis of the program, others can come in from the outside world via the 0-ary function `input` at run time.

[4] While this information may be useful for dead-code elimination and other forms of static analysis, this is of little importance to size-change termination.

For instance, consider the more complex program `reverse` in Listing 4.4 (17) and it's corresponding control flow graph in Figure 4.5 (17) . The abstract pattern for the success transition 0 is simply `right` with the variable space consisting merely of the variable `right` with an unknown value. The abstract pattern is matched to the actual pattern `left.right`, and for the sake of avoiding ambiguity let us refer to that pattern as simply `a.b`. Hence, the variable is destructed into two children, `a` and `b` such that `a` $<$ `right` and `b` $<$ `right`.

Definition 15. *The relation* $\Upsilon : C_{caller} \times N_{caller} \times C_{callee} \times B_{callee} \to \{>, \}$*, between the values of the bound variables of the function clause and the values that the arguments of the function call evaluate to.* $\uparrow$ *indicates an increase,* $\Uparrow$ *a nondecrease.*

Lemma 0.5.

This is followed by the source transmitting the resulting values to the destination, i.e. performing a control transition and the destination pattern matching the incoming values, the result of which may be binding of variables in the destination clause.

Variables (or names) are bound to values, hence the relation ∇ is a injective relation between variable names, where each mapping is annotated with the change in size indicating a lower bound on the difference between the values of the variables.

In the following sections we discuss how we can deduce the lower bound on the difference between the values of the variables.

4.2.3 Pattern matching

Definition 16. *For the sake of further discussion we'll regard the underscore pattern simply as a uniquely named variable that is not used in the expression of the function clause, i.e. it is also bound to a value.*

Lemma 0.6. *If the single pattern declaration includes a variable name, we can safely state that if some variable is bound as a result of pattern matching the corresponding argument, the value that variable is bound to is less than or equal to the value of the original argument.*

Proof. There is no construction operator for a pattern declaration. □

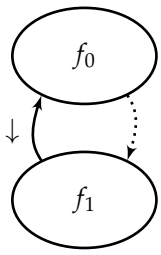
Lemma 0.7. *If a destruction operator participates in a pattern declaration, then any variable bound as a result of pattern matching the corresponding argument will be bound to a value strictly less than the value of the original argument.*

Proof. A variable can be bound to the actual value of the argument iff the entire pattern is just a variable name. If the pattern contains at least one destruction operator, the syntax rules of Δ cause for that operation to be performed first when pattern matching the argument. Hence, any value that can be bound thereafter has a value with at least one node less than the original value of the argument. □

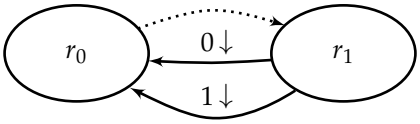
Hence, we can deduce from Listing 4.3 (16) , that when f_1 makes a call to f_0 it does so with a value strictly less then it's own argument, i.e. the transition $f_1 \to f_0$ strictly decreases a value. Visually we will mark this with a $\downarrow$. The Lemmas 0.6 (19) and 0.7 (19) can be used to deduce the same sort of relationship for the transitions $r_1 \xrightarrow{0,1} r_0$ for Listing 4.4 (17) . These observations are summarised in Figure 4.6 (20) .

4.2.4 Calls to multivariate functions

The call graph notation used thus far has only been used for describing calls to unary functions. As an example of a multivariate function, we may consider the function `normalized-less/2`, introduced in § 3.6.2 (11) . We use this function to define the program in Listing 4.5 (20) . The corresponding call graph is shown in Figure 4.7 (20) .



(a) The program in Listing 4.3 (16) .



(b) The program in Listing 4.4 (17) .

Figure 4.6: Call graphs with annotated edges for various programs.

Listing 4.5: A sample program with a multivariate function.

```
n0: normalized-less 0 b := b
n1: normalized-less _ 0 := 0
n2: normalized-less _.a _.b := normalized-less a b
normalized-less input input
```

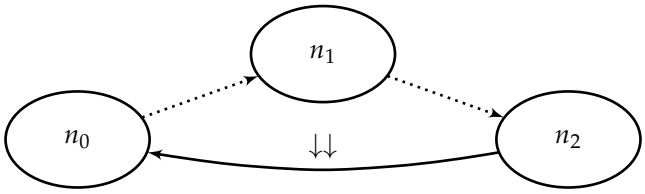


Figure 4.7: A control flow graph for the program defined in Listing 4.5 (20) .

The notation is straightforward, the juxtaposition of the $\downarrow$ indicates the size change of the respective arguments, read left to right as in the function clause definition.

4.2.5 Nonincreasing transitions

There are cases where for a given transition in a call cycle, we can't tell whether the sizes are strictly decreased or remain the same, but we can definitely say that there is *no increase* in the sizes of variables. As an example, consider the program in Listing 4.6 (20) .

Listing 4.6: The binary function g has a call cycle with nonincreasing sizes in variables.

```
g0: g 0 0 = 0
g1: g _.a b._ = g 0.a b.0
g input input
```

For the recursive clause g_1, it is unclear whether the sizes of the variables are decreased in the transition $g_1 \to g_0$, or not. Specifically, if the arguments to g are of the form `0._` and `_.0`, respectively, the size is *not* decreased by the call. We'll denote such transitions by the symbol $\Downarrow$. We can now draw the call graph for the program in Listing 4.6 (20) as in Figure 4.8 (21) .

4.2.6 Increasing transitions

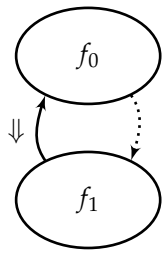


Figure 4.8: An annotated call graph for the program in Listing 4.6 (20) .

Listing 4.7: The function `infinite-join/2` infinitely joins..

```
f0: f a b = f a.b b.a
f input 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 Δ. 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 (24) .
[2] See Appendix A.2.3 (25) .

3. $\rightarrow$