# Termination analysis of first order programs

Oleksandr Shturmov

January $10^{th}$, 2012

# Contents

# Chapter 1

# Introduction

## 1.1 Motivation

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

**Definition 1.1.1.** *Given the set of all possible programs with finite program texts $P$, and all possible finite inputs $V$, find a program $p \in P$, that can for any $p' \in P$, and any input $v \in V$, within a finite amount of time, return either "halts" or "doesn't halt", depending on whether $p'$, given argument $v$, eventually stops or runs indefinitely, respectively.*

An important part of this definition is that a program is a finite sequence of discrete, terminating steps. Hence, the problem can be restated as determining whether the given program contains any 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 already in 1936/1937. Turing's proof[Turing, 1936] however, would live to become the one more widely recognised, although they are mutually reducible 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.

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 problem instances for which the result "unknown" is returned.

**Definition 1.1.2.** *Given the set of all possible programs with finite program texts $P$, and all possible finite inputs $V$, find a program $p \in P$, that can for any $p' \in P$, and any input $v \in V$, within a finite amount of time, return either "halts", "doesn't halt", or "unknown", depending on whether $p'$, given argument $v$, eventually stops, runs indefinitely, or neither can be decided. Find a $p$ such that the number of problem instances for which "unknown" is returned, is minimized.*

## 1.2 About the title

The original title of this project read "Sound and complete termination analysis of higher order programs". The title and the accompanying synopsis were overenthusiastic, and this text does not touch upon the problems related to higher order programs, although size-change termination has been mildly extended to such programs as well [Jones and Bohr, 2004], [Sereni and Jones, 2005].

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

What's more, "sound and complete" termination analysis is undecidable in general as per Definition 1.1.1 (5). Hence, whatever conclusions that could've been reached, would've either been unsound or incomplete. Instead we turn our attention to methods that are sound and complete if the halting problem is defined as in Definition 1.1.2 (5).

This definition also allows for a trivial implementation, in particular, one that returns "unknown" for any given program. While such a solution is valid, it is not particularly useful.

## 1.3 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, discrete mathematics, and basic concepts of functional programming languages (Monads excluded). All of these topics, at the present state of writing, are subject to basic undergraduate courses in computer science. Ideally, the reader should be well familiar with at least one purely functional programming language such as ML or Haskell.

More exemplary, the following concepts shouldn't frighten you:

- Algorithm, Big-O notation.
- Function, expression, pattern, pattern matching, loop, recursion.
- Induction, variant, invariant.
- Regular expressions.
- Backus-Naur Form, structured operational semantics.
- Turing machine, the halting problem.
- Set, list, tuple, head, tail.
- Graph, node, edge, path, cycle.

## 1.4 Preliminaries

To avoid ambiguity and to aid some of the discussions below, we provide the following definitions.

**Definition 1.4.1.** *Let $\mathbb{N}^0$ denote the set of nonnegative integers and let $\mathbb{N}$ denote the set of positive integers.*

**Definition 1.4.2.** *When dealing with variables that are insignificant to some definition, lemma, theorem, etc. we might simply denote them as _, which carries over the conventional "wildcard" meaning from functional languages.*

**Definition 1.4.3.** *When dealing with lists, aka. finite ordered sequences, we'll adopt the following notational constructs:*

1. *Given a list $L$ and a possibly infinite set $S$, we say that $L \subset S$, if $L$ consists solely of elements also contained in $S$.*
2. *Given a list $L$, $|L| \in \mathbb{N}^0$ and denotes the length of $L$.*
3. *Any given list $L$ is the ordered sequence $l_1, l_2, \cdots, l_{|L|}$.*
4. *Given a list $L$ and an element $l$, we say that $l \in L$ if $l$ is one of $l_1, l_2, \cdots, l_{|L|}$.*
5. *Lists may be nested, hence, given an element $e$ and a list $l$, we say that $e \Subset l$ if $e$ is contained in either $l$ or one of its nested lists.*
6. *Given the lists $L$ and $L'$ we say that $L = L'$ iff $|L| = |L'|$ and $\forall\, i \in \{i \mid i \in \mathbb{N} \wedge i \leq |L|\}\ l_i = l'_i$.*

7. *Given a list* $L = l_1, l_2, \cdots, l_{|L|}$, $L_{head}$ *refers to* $l_1$, *and* $L_{tail}$ *refers to the sequence* $l_2, l_3, \cdots, l_{|L|}$.
8. $\emptyset$ *denotes the empty list.*
9. $[S]$, *where* $S$ *is some set, denotes a list of elements from the set* $S$.
10. $[term \mid variables \in spaces, precondition]$ *denotes a finite sequence where each element is constructed from evaluating the "term", containing the given "variables" are in the given "spaces", and fulfilling the "precondition". This is reminiscent of conventional list comprehension.*

For lists, we need not necessarily know the size, hence we often refer to lists of some particular known size as tuples.

**Definition 1.4.4.** *A tuple is a sequence of a known size, represented as a comma-separated list enclosed in* $\langle\rangle$. *A tuple definition has the form* $\langle x_1, x_2, \ldots, x_n\rangle : S_1, S_2, \ldots, S_n$, *meaning* $x_1 \in S_1, x_2 \in S_2, \ldots, x_n \in S_n$, *where* $n \in \mathbb{N}^0$.

## 1.5 Chapter overview

**Chapter 2** This chapter recaps some general concepts from computability theory, defining the concept of Turing machines and proving the general halting problem undecidable.

**Chapter 3** This chapter formally introduces a Turing-complete language called $\Delta$, that will be used to aid the discussions in latter chapters. We discuss data representation in $\Delta$, define the syntax and semantics for the language, provide some sample programs and show that $\Delta$ is Turing-complete.

**Chapter 4** This chapter describes the size change termination principle and describes how the technique can be applied to $\Delta$ programs. While the chapter seeks to describe the concepts known from [Lee et al., 2001] it does so with a heavy reliance on $\Delta$, so it is recommended to be familiar with Chapter 3.

**Chapter 5** This chapter proposes a small extension to size-change termination, dubbed shape-change termination, that allows to determine the halting property for a slightly wider class of programs. This chapter exploits many of the definitions in both Chapters 3 & 4, so it is recommended to have read those beforehand.

# Chapter 2

# Computability and 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 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]. This is because an infinite value clearly 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 its output[3] are themselves deterministic. This is easy to see, by a sort of cut-and-paste proof. In particular, if a procedure made use of a stochastic process to yield a deterministic result, the stochastic process would have to yield the same output for the same input in order for the global determinism property to be withheld, which is clearly absurd.

## 2.2 Enumerability

**Definition 2.2.1.** *A set $S$ is enumerable if there exists a bijective function $f : S \longrightarrow \mathbb{N}$.*

## 2.3 Turing machine

Alan Turing was one of the first to introduce a powerful, yet simplistic computational model, in the form of what is now known as a *Turing machine*[4].

A Turing machine is a machine that has unlimited and unrestricted memory in the form of a one dimensional tape that can be extended infinitely in both directions, known as left and right. The tape itself is a series of symbols of a finite alphabet. Furthermore, the Turing machine has a read/write head that at any given point in time is located over a single symbol on the tape. The head can hence read in

[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.
[4] Although various definitions exist, this one was inspired by http://plato.stanford.edu/entries/turing-machine/.

the symbol and either overwrite it, move one symbol to the left, or move one symbol to the right, both in one computational step.

The actual action to perform is defined in terms of a transition table which is a finite list of of the 4-tuples $\langle \lambda_0, \sigma, \lambda_1, \omega \rangle : \Lambda \times \Sigma \times \Lambda \times \Omega$, where $\Lambda$ is the set of states of the machine (some non-existent), $\Sigma$ is the alphabet of the machine, and hence the tape, typically $\{0, 1\}$, and $\Omega$ is the action table of the machine. The action table is typically the set $\{\leftarrow, \rightarrow, 0, 1\}$, denoting a left-wise and right-wise move, as well as write 0 and 1, respectively. At any given point in time a Turing machine is in a state, say $\lambda_0$. Such a state has one and only one transition, namely $\langle \lambda_0, \sigma, \lambda_1, \omega \rangle$. If the symbol on the tape is equal to $\sigma$, the action $\omega$ is performed and regardless of the symbol, the machine moves into state $\lambda_1$.

Last but not least, the Turing machine has an initial state, and often an accepting and rejecting state. We'll keep the definition simple and let any state that is undefined in the transition table yield a halting of the machine. The value which the read/write head is located over, can hence represent an accepting or rejecting state as needed.

## 2.4 Computational equivalence

**Definition 2.4.1.** *We say that two languages are computationally equivalent if they both can compute the same class of functions.*

## 2.5 The halting problem

**Theorem 2.5.1.** *There does not exist a Turing machine $H(M, x)$ that given an arbitrary Turing machine $M$ and an arbitrary input $x$ determines whether $M$ halts for $x$.*

*Proof.* Assume for the sake of contradiction that there exists a Turing machine $H(M, x)$, that given a Turing machine $M$ decides whether $M$ halts on input $x$, that is

$$H(M, x) = \begin{cases} true & M \text{ halts on } x \\ false & M \text{ does not halt on } x. \end{cases}$$

This means that we can construct another Turing machine $F$, which calls $H(M, M)$ and depending on the outcome, yields the opposite result, i.e.

$$F(M) = \begin{cases} true & \text{if } H(M, M) \rightsquigarrow false \\ false & \text{if } H(M, M) \rightsquigarrow true. \end{cases}$$

Consider now running the Turing machine $F$ with itself as input, either result is absurd, and hence neither the Turing machine $F$ nor $H$ can exist. □

## 2.6 Introduction to size-change termination

In the chapters that follow we will describe the size-change termination principle as in [Lee et al., 2001], and extend it slightly. The general idea behind the technique is to construct a control flow graph for a given program and consider the cycles in that graph.

If every control flow cycle reduces a value of a well-founded data type on every iteration, i.e. monotonically decreases the value, then, rather intuitively, the program must terminate. In particular, a well-founded data type is a data type where any set of values has a minimal element. The technique hence relies on the fact such a value eventually bottoms out and the cycle, unable to decrease the value any further, must inevitably terminate.

# Chapter 3

# The language Δ

The goal of this work is to describe a few automated termination analysis techniques, and in particular, size-change termination. In order to allow for the following chapters to retain a modest level of abstraction to the Turing machine, such that the techniques are described for an environment that is modestly applicable to solving moderate programming problems, a Turing complete language Δ is introduced.

## 3.1 The goal

The intent of the language is two-fold, (1) aid the descriptions of automated termination analysis techniques in latter chapters, and (2) be relatively expressive.

Expressiveness of a language is a rather subjective and domain-driven concept. First and foremost, expressiveness depends on the initial intended domain of the language. Of course, Turing complete languages are known to be universally applicable, however, some languages are just more fine tuned to solving some problems, while others are better tuned to solving other problems.

Δ is a language with very few primitive operations but is expressive enough to write the Fibonacci and Ackermann functions in an elegant way. To do this, Δ borrows some syntax and semantics from purely functional languages such as ML or Haskell. Hence, programs in Δ make heavy use of pattern matching and recursion to achieve branching and looping, some of the constructs required for a language to be Turing complete.

Unlike ML and Haskell, Δ is a language that completely disregards the concepts of abstract data structures and types. Hence, many data driven programs will be hard to write in Δ. Of course, this is not to say that data flow analysis is irrelevant to termination analysis as such, on the contrary, it is key to size-change termination. It is because of this prime importance of data flow to termination analysis, that data value representation is kept to its almost lowest possible denominator. This keeps the analysis clean of rather irrelevant abstract data structure fiddling. What's more, any methods developed for Δ can be extended and used in a language with types and abstract data structures, as long as it is computationally equivalent to Δ.

Also, unlike most purely functional languages, Δ is a first-order, call-by-value language. This is done in part to adhere to the general flow of [Lee et al., 2001], and in part to keep the analysis simple at first. Higher-order constructs impose difficulties when deducing changes in size, and evaluation strategies other than call-by-value impose a similar sort of difficulties.

## 3.2 Data

We chose to keep Δ untyped.

**Definition 3.2.1.** *Let* $\mathbb{B}$ *denote the infinite set of values representable in* Δ.

**Definition 3.2.2.** *Let the data type* $T$ *range over the set* $\mathbb{B}$.

The automated termination analysis techniques discussed in latter chapters will rely heavily on the well-foundedness of the language's data types.

**Definition 3.2.3.** *Let $f : \mathbb{B} \to \mathbb{N}^0$ be a surjective function. $T$ is well-founded iff*

$$\forall B \subseteq \mathbb{B} \left( B \neq \emptyset \to \exists\, b' \in B \forall\, b \in B \backslash \{b'\}\ f(b') < f(b) \right).$$

**Definition 3.2.4.** *A value $b \in \mathbb{B}$ is a either $\emptyset$ or $\left\langle b_{left}, b_{right} \right\rangle : \mathbb{B} \times \mathbb{B}$.*

Definition 3.2.4 (12) implies that we chose to represent all values in $\Delta$ in terms of *unlabeled ordered binary trees*, henceforth referred to as simply *binary trees*. We refer to empty binary trees, i.e. $\emptyset$, as *leafs*. For any nonempty binary tree $b = \left\langle b_{left}, b_{right} \right\rangle : \mathbb{B} \times \mathbb{B}$, we refer to $b$ as a *node*, $b_{left}$ as the *left child* of $b$, and $b_{right}$ as the *right child* of $b$.

**Definition 3.2.5.** *We represent a leaf with the atom 0, and define the binary relation $\cdot$ to be the set* $\left\{ \left\langle b_{left}, b_{right} \right\rangle \mid b_{left}, b_{right} \in \mathbb{B} \right\}$.

We'll sometimes refer to the $\cdot$ operator as "cons".

**Theorem 3.2.1.** *The set $\mathbb{B}$ is infinite.*

*Proof.* Follows from Definition 3.2.4 (12). □

### 3.2.1 Size

Definition 3.2.3 (12) made use of a surjective function $f : \mathbb{B} \to \mathbb{N}^0$. To ensure well-foundedness of $\Delta$'s data values we need to define such a function or equivalently, define a notion of the size of a data value in $\Delta$.

**Definition 3.2.6.** *The size of a value $b \in \mathbb{B}$ is the number of nodes in the value.*

**Theorem 3.2.2.** *There exists a surjective function $f : \mathbb{B} \to \mathbb{N}^0$.*

*Proof.* The proof is two-fold. First, we prove by induction that any $n \in \mathbb{N}^0$ can be represented in $\Delta$:

**Base case** A leaf has no nodes, and hence represents the value 0.

**Assumption** If we can represent $n \in \mathbb{N}^0$ in $\Delta$, then we can also represent $n + 1 \in \mathbb{N}^0$ in $\Delta$.

**Induction** Let $n$ be represented by some $b \in \mathbb{B}$, then $n + 1$ can be represented by $0 \cdot b$.

Second, by Definition 3.2.4 (12), any $b \in \mathbb{B}$ has one and only one number of nodes, hence it has one and only one representation $n \in \mathbb{N}^0$, indeed the number of nodes. □

**Corollary 3.2.3.** *$T$ is well-founded.*

*Proof.* Follows from Definition 3.2.3 (12) and Theorem 3.2.2 (12). □

**Definition 3.2.7.** *Definition 3.2.6 (12) allows us to WLOG overload the binary operators $=, <, >, \geq, \leq, +$ and $-$, defined over $\mathbb{N}^0 \times \mathbb{N}^0$, for $\mathbb{B} \times \mathbb{B}$.*

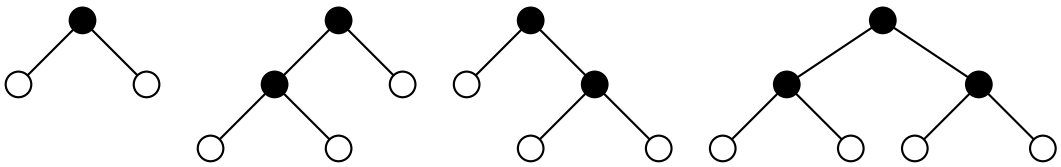

**Figure 3.1:** A few sample value visualizations having the sizes 1, 2, 2, and 3 respectively.

### 3.2.2 Visual representation

To provide for a more comprehensible discussion, we'll sometimes visualize the values we're dealing with. Figure 3.1 (13) shows a few sample value visualizations.

**Definition 3.2.8.** *We visually represent values in* $\mathbb{B}$ *using a common convention of drawing binary trees, with the root at the top and the subtrees drawn in an ordered manner in a downward direction. Nodes are represented by filled dots while leafs are represented by hollow ones. We use the Reingold-Tilford algorithm[Reingold and Tilford, 1981] for laying out the trees.*

Although visually, a strict increase is usually associated with an upwards direction, and a strict decrease is usually associated with a downwards direction, this definition implies the exact opposite. A strict increase in value would imply more nodes and hence a downward extension of the binary tree along one of the branches, while a strict decrease in value would imply fewer nodes and hence and upward contraction of the binary tree in one of the branches.

### 3.2.3 Shapes

A shape is an abstract description of a value in $\Delta$. Shapes *describe* values, and values *match* shapes, in particular, a shape describes a value iff the value matches the shape.

**Definition 3.2.9.** *We refer to the set of all possible shapes as* $\mathbb{S}$.

**Definition 3.2.10.** *A shape* $s \in \mathbb{S}$ *is either* $\emptyset$, $\left\langle s_{left}, s_{right} \right\rangle : \mathbb{S} \times \mathbb{S}$, *or* $\langle \triangle \rangle$.

We refer to $\emptyset \in \mathbb{S}$ as the leaf shape, $\langle \triangle \rangle \in \mathbb{S}$ as the triangle shape, and any $\left\langle s_{left}, s_{right} \right\rangle \in \mathbb{S}$ as a node shape.

**Definition 3.2.11.** *We refer to the leaf shape as* 0 *and the triangle shape merely as some* $s \in \mathbb{S}$. *We also overload the binary relation* $\cdot$ *with the set* $\left\{ \left\langle s_{left}, s_{right} \right\rangle \mid s_{left}, s_{right} \in \mathbb{S} \right\}$.

**Definition 3.2.12.** *We define the binary relation* $\succ$, *read "matches", ranging over* $\mathbb{B} \times \mathbb{S}$ *using the following subdefinitions:*

1. $\forall\, b \in \mathbb{B}\ b \succ \langle \triangle \rangle$.
2. $(0 \succ \emptyset) \wedge (\forall\, b \in \mathbb{B} \backslash \{0\}\ b \not\succ \emptyset)$.
3. $\forall\, s = \left\langle s_{left}, s_{right} \right\rangle \in \mathbb{S}\ \forall\, b = \left\langle b_{left}, b_{right} \right\rangle \in \mathbb{B}\ \left( \left( b_{left} \succ s_{left} \wedge b_{right} \succ s_{right} \right) \rightarrow b \succ s \right)$.

**Lemma 3.2.4.** *Any shape that contains a triangle shape in its binary tree, describes infinitely many values.*

*Proof.* Follows directly from Definition 3.2.12 (13) and Theorem 3.2.1 (12). □

As with values, it might prove beneficial to the discussion to visualize the shapes. Figure 3.2 (14) shows a few visualizations of shapes.

**Definition 3.2.13.** *Generally we'll visually represent shapes as we represent values. Triangle shapes will be represented with hollow triangles.*

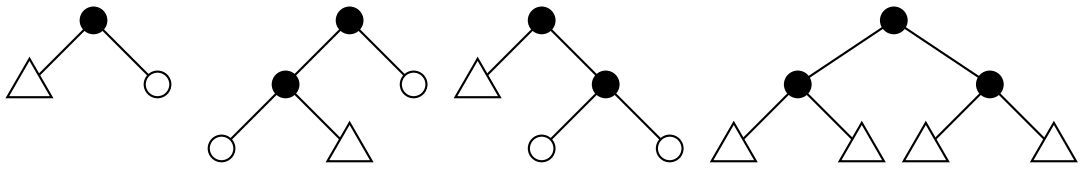

**Figure 3.2:** A few shape visualization examples. The leftmost shape describes values that are nodes with leafs as right children and any trees as left children. The two leftmost values in Figure 3.1 (13) match this shape.

**Definition 3.2.14.** *Let $f : \mathbb{S} \to \mathbb{B}$ be a surjective function that given a shape $s \in \mathbb{S}$ transforms it into a $b \in \mathbb{B}$ by replicating the binary tree, except that any $\langle\triangle\rangle$ is replaced by 0. We overload the binary relation $\succ$ with the set $\{\langle s_1, s_2\rangle \mid s_1, s_2 \in \mathbb{S}, f(s_1) \succ s_2\}$.*

**Lemma 3.2.5.** $\forall\, s_1, s_2 \in \mathbb{S}\,((s_1 \neq s_2 \wedge s_1 \succ s_2) \to s_2 \not\succ s_1)$

*Proof.* If $s_1 \neq s_2$ and $s_1 \succ s_2$, then by Definition 3.2.12 (13), $s_1$ must have more nodes than $s_2$, and by Definition 3.2.12 (13), a shape with more nodes cannot match a shape with fewer nodes. □

## 3.3 Syntax

We describe the syntax of $\Delta$ in terms of an extended Backus-Naur form[1]. 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>`.

```
<program>         ::= <clause>* <expression>                                  (3.1)
<expression>      ::= <element> ( '.' <expression> ) ?                        (3.2)
<element>         ::= '0' | '(' <element> ')' | <name> | <application>        (3.3)
<application>     ::= <name> <expression>+                                    (3.4)
<clause>          ::= <name> <pattern>+ ':=' <expression>                     (3.5)
<pattern>         ::= <pattern-element> ( '.' <pattern> ) ?                   (3.6)
<pattern-element> ::= '0' | '_' | '(' <pattern> ')' | <name>                  (3.7)
<name>            ::= ['a'-'z'] ( ['-' 'a'-'z']* ['a'-'z'] )?                 (3.8)
```

**Definition 3.3.1.** *Table 3.1 (15) defines shorthands for various language constructs. We'll often refer to these in further discussions. Additionally, we'll let the atoms 0 and _ represent themselves.*

**Definition 3.3.2.** *For any given $v \in \mathbb{V}$ and $P \subset \mathbb{P}$, we say that $v \in P$ if $v$ occurs in some $p \in P$.*

**Definition 3.3.3.** *A clause $c \in \mathbb{C}$ is a tuple $\langle v, P, x\rangle$, where $v \in \mathbb{V}$ is the name of the clause, $P \subset \mathbb{P}$ is a non-empty list of patterns of the clause, and $x \in \mathbb{X}$ is the expression of the clause. $P$ is ordered by occurrence of the patterns in the program text.*

**Definition 3.3.4.** *We say that a clause $c = \langle v, P, x\rangle$ "accepts" an argument list $B$ iff $|P| = |B|$ and $\forall\, \{i \mid 0 \geq i < |P|\}\ b_i \in B \wedge p_i \in P \wedge b_i \succ p_i$.*

**Definition 3.3.5.** *A function $f \in \mathbb{F}$ is a tuple $\langle v, C\rangle$, where $v \in \mathbb{V}$ is the name of the function, and $C \subset \mathbb{C}$ is the non-empty list of clauses of the function. It must hold for $C$ that $\forall\, c \in C\ \ (c = \langle v_c, P_c, x_c\rangle \wedge v_c = v)$ and $\forall\, c_1, c_2 \in C\ \ (c_1 = \langle v_1, P_1, x_1\rangle \wedge c_2 = \langle v_2, P_2, x_2\rangle \wedge |P_1| = |P_2|)$. $C$ is ordered by occurrence of the clauses in the program text.*

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

[1] The extension lends some constructs from regular expressions to achieve a more concise dialect. The extension is described in detail in Appendix A.1 (43).

| Description | Instance | Finite list | Space |
|---|---|---|---|
| Expression | $x$ | $X$ | $\mathbb{X}$ |
| Element (of an expression) | $e$ | $E$ | $\mathbb{E}$ |
| Function | $f$ | $F$ | $\mathbb{F}$ |
| Clause | $c$ | $C$ | $\mathbb{C}$ |
| Pattern | $p$ | $P$ | $\mathbb{P}$ |
| Value | $b$ | $B$ | $\mathbb{B}$ |
| Name | $v$ | $V$ | $\mathbb{V}$ |
| Program | $r$ | $R$ | $\mathbb{R}$ |

**Table 3.1:** Shorthands for various language constructs for use in latter discussions. We provide shorthands for an instance, a list, and the space of a construct. For instance, $x$ is some particular expression, $X$ is some particular list of expressions, and $\mathbb{X}$ is the set of all possible expressions.

**Definition 3.3.6.** *A signature of some function $f = \langle v, C\rangle$ is the tuple $\langle v, |P|\rangle$, s.t. $\forall\, c \in C_f\ |P_c| = |P|$. We'll adopt the Erlang notation when talking about function signatures, i.e. if we have a function `less` that takes in two parameters, we'll refer to it as `less/2`.*

We assume for it to be fairly simple to construct the set $F$ of a given program $r$ given the set of clauses $C$ derived during syntactic analysis of the program text.

**Definition 3.3.7.** *A program $r$ is a tuple $\langle F, x\rangle$, where $F \subset \mathbb{F}$ is the list of functions defined in program $r$, and $x$ is the expression of program $r$.*

**Definition 3.3.8.** *A function call is a tuple $\langle v, X\rangle$, where $v \in \mathbb{V}$ is the name of the callee, and $X \subset \mathbb{X}$ is a non-empty list of arguments for the function call, ordered by occurrence of the expressions in the program text.*

0-ary clauses are disallowed to avoid having to deal with constants in general. The term `'_'` in `<pattern-element>` is the conventional wildcard operator; it indicates a value that won't used in the clause expression, but some value has to be there for an argument to match the pattern. Furthermore, as will be clear from the semantics, multiple occurrences of `'_'` in a clause pattern list does not indicate that the same value has to be in place for each `'_'`.

**Definition 3.3.9.** *When describing various values and patterns in definitions, theorems, proofs, etc. we'll sometimes make use of _ to denote parts of the value or pattern that are irrelevant to the said definition, theorem, proof, etc.*

### 3.3.1 Patterns constitute shapes

The nonterminal declarations for patterns, in particular 3.6 and 3.7, indicate that a pattern are equatable to shapes.

**Definition 3.3.10.** *The pattern `'0'` corresponds to the leaf shape. The patterns `'_'` and `<name>` correspond to triangle shapes. Any pattern `a.b` corresponds to the shape $a \cdot b$ iff the pattern `a` corresponds to the shape $a$ and the pattern `b` corresponds to the shape $b$.*

**Definition 3.3.11.** *$\forall\, p \in \mathbb{P}\ \forall\, s \in \mathbb{S}$ $p = s$ iff $p$ corresponds to $s$ as by Definition 3.3.10 (15).*

**Definition 3.3.12.** *We overload the binary relation $\succ$ with the set*
$\{\langle p_1, p_2\rangle \mid p_1, p_2 \in \mathbb{P}, s_1, s_2 \in \mathbb{S} \wedge p_1 = s_1 \wedge p_2 = s_2\}$.

### 3.3.2 Unary functions from multivariate functions

The patterns of a clause as well as the arguments of a function call get special treatment in $\Delta$ in that they according to Definition 3.3.3 (14) and Definition 3.3.8 (15) are ordered by their occurrence in the program text. This order is important to make sure that the appropriate argument is matched against the appropriate pattern.

While this is setup is practical for the programmer, it is of no use to us due to Theorem 3.3.1 (16). In latter discussions, this particular theorem allows us to keep to unary functions, and regard the extension to multivariate functions as a fairly simple matter.

**Theorem 3.3.1.** *Any multivariate function in $\Delta$ can be represented with a unary function.*

*Proof.* Given a multivariate function $f = \langle v, C \rangle$:

1. For each clause $c \in C$, where $c = \langle v, P, x \rangle$, replace the pattern list $P$ with $P' = \{p\}$. Construct $p$ by initially letting $p = 0$, and folding left-wise over $P$, performing $p = p \cdot p'$ for each $p' \in P$.
2. For each call $\langle v, X \rangle$ to function $f$, replace $X$ with the set $X' = \{x\}$, where $x$ has been constructed in a manner equivalent to the pattern $p$ above.

It is easy to see that both the constructed patterns and expressions are indeed valid patterns and expressions, and that $f$ hence becomes a unary function. □

As this transformation is relatively simple to perform, we redefine the generic clause tuple to have but one pattern in place of a list.

**Definition 3.3.13.** *We redefine the clause $c$ to be the tuple $\langle v, p, x \rangle$, where $v \in \mathbb{V}$ is the name of the clause, $p \in \mathbb{P}$ is the pattern of the clause, and $x \in \mathbb{X}$ is the expression of the clause.*

**Definition 3.3.14.** *We redefine a function call to be the tuple $\langle v, x \rangle$, where $v \in \mathbb{V}$ is the name of the callee, and $x \in \mathbb{X}$ is the argument to the (always unary) callee.*

## 3.4 Semantics

In the following section we describe the semantics of $\Delta$ using a form of structured operational semantics. The syntax used to define the reduction rules is largely equivalent to the Aarhus report[Plotkin, 2004], but differs slightly[2].

### 3.4.1 The memory model

**Definition 3.4.1.** *The memory is a binary relation $\sigma$, which is the set $\{(v, b) \mid v \in \mathbb{V} \wedge b \in \mathbb{B}\}$.*

To keep $\Delta$ first order we distinguish between the function space and variable space.

**Definition 3.4.2.** *Let $\phi \subseteq \sigma$ represent the function space and let $\beta \subseteq \sigma$ represent the variable space, what's more, $\phi \cup \beta = \sigma$. When we refer to $\sigma$, $\phi$ or $\beta$ in set notation, we refer merely to the names of the variables, hence to keep $\Delta$ first order, let $\phi \cap \beta = \emptyset$.*

#### Making $\Delta$ higher order

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

### 3.4.2 Program evaluation

Given a program $r = \langle F, x \rangle$, we apply the following semantics:

$$\frac{\langle F, \emptyset \rangle \to \phi_1 \wedge \sigma = \phi_1 \wedge \langle x, \sigma \rangle \to \langle b, \sigma \rangle}{\langle F, x \rangle \to b} \tag{3.9}$$

[2]The syntax applied here is described in further detail in Appendix A.2 (44).

### 3.4.3 Function declarations

Given a list of functions $F$, and some function space $\phi$, we apply the following semantics:

$$\frac{(F = \emptyset \wedge \phi = \phi_1) \vee (F_{head} = \langle v, C\rangle \wedge \phi_2 = \phi[v] \mapsto \langle C, \phi\rangle \wedge \langle F_{tail}, \phi_2\rangle \to \phi_1)}{\langle F, \phi\rangle \to \phi_1} \tag{3.10}$$

The fact that the active function space is saved together with the list of clauses for any given function should intentionally indicate that $\Delta$ is *statically scoped*.

### 3.4.4 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:

$$\frac{\langle \textsc{Single}, x, \sigma\rangle \to b \vee \langle \textsc{Chain}, x, \sigma\rangle \to b}{\langle x, \sigma\rangle \to b} \tag{3.11}$$

$$\frac{x = e \wedge \langle e, \sigma\rangle \to b}{\langle \textsc{Single}, x, \sigma\rangle \to b} \tag{3.12}$$

$$\frac{x = e_1 \cdot x_1 \wedge \langle e_1, \sigma\rangle \to b_1 \wedge \langle x_1, \sigma\rangle \to b_2}{\langle \textsc{Chain}, x, \sigma\rangle \to b} \quad (\text{where } b_1 \cdot b_2 = b) \tag{3.13}$$

### 3.4.5 Element evaluation

According to the syntax specification, an element of an expression can either be the atom 0, a variable, an expression (in parentheses), or an application.

$$\frac{\langle \textsc{Zero}, e, \sigma\rangle \vee \langle \textsc{Expression}, e, \sigma\rangle \vee \langle \textsc{Variable}, e, \sigma\rangle \vee \langle \textsc{Application}, e, \sigma\rangle}{\langle e, \sigma\rangle \to \langle b, \sigma\rangle} \tag{3.14}$$

$$\frac{e = 0 \wedge b = 0}{\langle \textsc{Zero}, e, \sigma\rangle \to b} \tag{3.15}$$

$$\frac{e = x \wedge \langle x, \sigma\rangle \to b}{\langle \textsc{Expression}, e, \sigma\rangle \to b} \tag{3.16}$$

$$\frac{e = v \wedge \beta[v] = b}{\langle \textsc{Variable}, e, \sigma\rangle \to b} \tag{3.17}$$

$$\frac{e = \langle v, x\rangle \wedge \phi[v] = \langle C, \phi_1\rangle \wedge \langle x, \sigma\rangle \to b_{arg} \wedge \langle C, b_{arg}, 0, \phi_1\rangle \to b}{\langle \textsc{Application}, e, \sigma\rangle \to b} \tag{3.18}$$

### 3.4.6 Clause matching

We would like to ensure that pattern matching is exhaustive for any function definition. This is to avoid programs that terminate due to inexhaustive pattern matching.

**Definition 3.4.3.** *Given a function $f = \langle v, C\rangle$, it must hold that $\forall\, b \in \mathbb{B}\ \exists\ \langle v, p, x\rangle \in C\ b \succ p$.*

This definition allows us to define the following semantics for clause evaluation:

$$\frac{(C = \emptyset \wedge b = b_{acc}) \vee \big(C_{head} = \langle v, p, x\rangle \wedge b_{arg} \succ p \wedge \langle x, \phi\rangle \to \langle b, \phi\rangle\big) \vee \big(\langle C_{tail}, b_{arg}, 0, \phi\rangle \to b\big)}{\big\langle C, b_{arg}, b_{acc}, \phi\big\rangle \to b} \tag{3.19}$$

## 3.5 Built-in functions

In the following section we define a few built-in Δ functions. Some of them will be defined in terms of Δ itself.

### 3.5.1 Input

To be able to write more interesting programs, we'll define the primitive function `input/0` that can yield any valid Δ value. This is the only non-deterministic, 0-ary function in Δ.

### 3.5.2 Boolean operations

**Definition 3.5.1.** *We'll adopt the C-convention of letting any non-zero value represent a true value, and any zero value to represent a false value.*

Given the definition above, we define the often useful functions `and/2`, `or/2` and `not/1` in Listing 3.1 (18), Listing 3.2 (18) and Listing 3.3 (18), respectively. Note, that due to Δ being a call-by-value language, `or/2` is *not* shortcircuited as conventionally is the case.

```
1 and _._ _._ := 0.0
2 and _ _ = 0
```

**Listing 3.1:** The function `and/2`.

```
1 or 0 0 := 0
2 or _ _ := 0.0
```

**Listing 3.2:** The function `or/2`.

```
1 not 0 := 0.0
2 not _._ := 0
```

**Listing 3.3:** The function `not/1`.

### 3.5.3 Comparison

There are many imaginable programs that rely on some value being less, more, or equal to some other value. Since no such primitives are available we have to define such comparisons ourselves. However, there is seemingly no *elegant* way of figuring out how the sizes of two arbitrary values in Δ differ, using Δ itself. For this purpose we define the concept of a *normalized* value.

**Definition 3.5.2.** *A binary tree in normalized form is a binary tree that either is a leaf, or a node having a leaf as its left child and a binary tree in standard representation as its right child.*

Visually, a binary tree in standard representation is just a tree that only descends along the right-hand side. Refer to Figure 3.3 (18) for an example of a value in Δ and its normalized form.

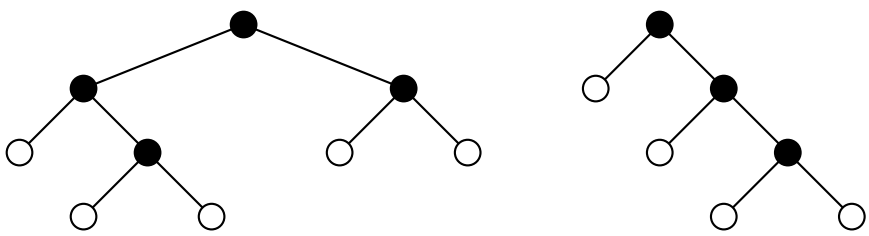

**Figure 3.3:** A sample value $b \in \mathbb{B}$ to the left, and its normalized form to the right.

Definition 3.5.2 (18) allows us to define the function `normalize/1` that normalizes a value. We've done this Listing 3.4 (19).

```
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
```

**Listing 3.4:** The function `normalize/1` turns any value $b \in \mathbb{B}$ into its normal form.

Comparing the sizes of two trees in this representation is just a matter of walking down two normalized trees simultaneously, until one of them, or both, bottoms out. If there is a tree that bottoms out strictly before another, that is the lesser value by Definition 3.2.6 (12). This allows us to define the functions `less/2` and `equal-size/2`[3] which we do in Listing 3.5 (19) and Listing 3.6 (19), respectively.

```
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
```

**Listing 3.5:** The function `less/2` yields true if the first argument is less than the second and false otherwise.

```
equal-size a b := normalized-equal-size (normalize a) (normalize b)

normalized-equal-size 0 0 := 0.0
normalized-equal-size _ 0 := 0
normalized-equal-size 0 _ := 0
normalized-equal-size _.a _.b := normalized-equal-size a b
```

**Listing 3.6:** The function `equal-size/2` function.

### 3.5.4 Increase & decrease

As with comparison, there are many imaginable programs that increase or decrease values. An increase in the number of nodes is trivial, as shown by the function `increase/1` in Listing 3.7 (19).

```
increase a := 0.a
```

**Listing 3.7:** The function `increase/1` increases a value by 1.

A decrease of a value on the other hand, requires normalization of the value and a right-wise walk down the tree until the bottom-most node is reached, after which the node is removed. What's more, $\Delta$ has no overflow and no negative values, so we must take care of what we do with the value 0, which hence cannot be decreased. We decide to let `decrease 0` yield `0`. All this is summarized in Listing 3.8 (19).

```
decrease 0 := 0
decrease a := normalized-decrease (normalize a)

normalized-decrease 0.0 := 0
normalized-decrease a.b := a.(normalized-decrease b)
```

**Listing 3.8:** The function `decrease/1` decreases a value 1, unless that value is 0, in which case nothing is done.

[3] We add the `-size` suffix in order to reserve the name `equal` for a function that compares two values in $\Delta$ by their actual tree structure rather than the number of nodes.

## 3.6 Sample programs

As an illustration of the language syntax, take a look at the programs in Listing 3.9 (20), Listing 3.10 (20) and Listing 3.11 (20).

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

reverse input
```

**Listing 3.9:** A program that reverses the order of the nodes of some supplied tree.

```
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)

fibonacci input
```

**Listing 3.10:** A program that computes the $n^{th}$ fibonacci when supplied with some $n$.

```
ackermann 0 n := 0.n
ackermann a.b 0 := ackermann (decrease a.b) 1
ackermann a.b c.d := ackermann (decrease a.b) (ackermann a.b (decrease c.d))

ackermann input input
```

**Listing 3.11:** The Ackermann-Péter function.

## 3.7 Turing-completeness of $\Delta$

We show that $\Delta$ is Turing-complete by showing that we can simulate a universal Turing machine, that is a Turing machine that can simulate any Turing machine on any input value.

As a consequence of the discussion in § 2.3 (9), we can describe any Turing machine in terms of its transition table, i.e. list of 4-tuples $\langle \lambda_0, \sigma, \lambda_1, \omega \rangle : \Lambda \times \Sigma \times \Lambda \times \Omega$. If we can show that $\Lambda$, $\Sigma$ and $\Omega$ have definitions that are subsets of $\mathbb{B}$, any such list of tuples can be represented as a $\Delta$ value.

**Definition 3.7.1.** *A list of values $L = [b_i \mid b_i \in \mathbb{B}, 0 < i \leq |L|]$ in $\Delta$ is represented as the value $b_1 \cdot b_2 \cdots b_n$.*

This superimposes an indexing on the values $b_1, b_2, \ldots, b_n$ using normalized $\Delta$ values. In particular, the element $b_i$ where $0 < i \leq |L|$ has the same location in the binary tree $b_l$ as the left child of the bottom-most node of the value $i \in \mathbb{B}$ in normalized form.

**Definition 3.7.2.** *Let $M'$ denote a list representing a Turing machine transition table, where each tuple has the form $\langle \sigma, \lambda, \omega \rangle$. Let $M \in \mathbb{B}$ be the $\Delta$ representation of $M'$. The set of possible states is hence $\{n(i) \mid i \in \mathbb{B}, 0 < i \leq |M|\} \subset \Lambda$, where $n$ is the normalization function from Listing 3.4 (19).*

$\Lambda$ is a superset of the set of possible states to allow transitions to undefined states, the effect of which is a halting of the machine.

**Definition 3.7.3.** *The possible symbols, in the set $\Sigma$, are denoted as follows:*

1. `0.0`*, denoting the value* 0.
2. `0.0.0`*, denoting the value* 1.

**Definition 3.7.4.** *The possible actions, in the set $\Omega$, are denoted as follows:*

1. `0.(0.0)`*, denoting the action "move right on the tape".*

2. `(0.0).0`, *denoting the action "move left on the tape".*
3. `0`, *denoting the action "write* 0 *at current position on the tape".*
4. `0.0`, *denoting the action "write* 1 *at the current position on the tape".*

Let us assume that we have a function `find/2` at our disposal, that takes in a normalized value $i \in \mathbb{B}$ and a list $l \in \mathbb{B}$, and returns either the element $l_i$, if $i$ is a valid index in the list $l$, and 0 otherwise. Then Listing 3.12 (21) is a possible implementation of a universal Turing machine in $\Delta$.

```
1  utm _ 0 _             := 0
2  utm m s 0             := utm s 0.(0.0).0 m
3  utm m s 0.0           := utm s 0.(0.0).0 m
4  utm m s 0.r           := utm s 0.r.0 m
5  urm m s l.0           := urm s 0.l.0 m
6  utm m s l.(0.0).r     := utm-interpret m l.v.r (find s m)
7  utm m s l.(0.0.0).r   := utm-interpret m l.v.r (find s m)
8  utm m s l.v.r         := utm m s (l.v).r
9
10 utm-interpret _ _ 0 := 0
11
12 utm-interpret m l.(0.0).r (0.0).s.0 := utm m s l.(0.0).r
13 utm-interpret m l.(0.0).r (0.0).s.(0.0) := utm m s l.(0.0.0).r
14
15 utm-interpret m l.(0.0).r (0.0).s.(0.(0.0)) := utm m s (l.(0.0)).r
16 utm-interpret m l.(0.0).r (0.0).s.((0.0).0) := utm m s l.((0.0).r)
17
18 utm-interpret m l.(0.0.0).r (0.0.0).s.0 := utm m s l.(0.0).r
19 utm-interpret m l.(0.0.0).r (0.0.0).s.(0.0) := utm m s l.(0.0.0).r
20
21 utm-interpret m l.(0.0.0).r (0.0.0).s.(0.(0.0)) := utm m s (l.(0.0.0)).r
22 utm-interpret m l.(0.0.0).r (0.0.0).s.((0.0).0) := utm m s l.((0.0.0).r)
23
24 utm-interpret m t _.s._ := utm m s t
```

**Listing 3.12:** A universal Turing machine in $\Delta$. `m` stands for $M$, `s` stands for $\lambda$, `t` stands for tape, and `l` and `r` stand for left and right, respectively.

Line 8 is there merely to make sure that if the initial tape is somehow invalid, then it doesn't break the universal Turing machine. Line 24, is there, in part, due to a similar reason, to ensure that if an invalid transition table is given, this does not break the universal Turing machine.

The function `utm/3` takes in a transition table `m`, initial state `s`, and initial tape `t`, while the function `utm-interpret/3` takes in a transition table `m` the tape `t` and some tuple of the form $\langle \sigma, \lambda, \omega \rangle : \Sigma \times \Lambda \times \Omega$. Lines 1 and 10 are in place to ensure that the universal Turing machine halts on any invalid state.

The code should otherwise be fairly self-explanatory given the definitions above, with perhaps the exception of how we handle the tape, that in this case, expands indefinitely in both directions. In particular, the end of the tape is marked with a `0`, which should explain the somewhat unusual Definition 3.7.3 (20).

# 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. 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.

Such graphs are referred to as *control flow graphs*. With a control flow graph at hand, various optimization algorithms can be devised to traverse the graph and deduce certain properties, such as e.g. reoccurring primitive operations on otherwise static variables[Kildall, 1973].

## 4.1 Control flow graphs (or call graphs) in $\Delta$

### 4.1.1 Start and end nodes

Conventionally, a 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. Clearly, a program cannot be started nor ended more than once, and hence the start node, has out-degree 1 and in-degree 0, while the end node has out-degree 0, and (initially) variable in-degree since a program can be ended in more than one way. For reasons that will become apparent in later on, we chose to disregard the start and end nodes completely.

### 4.1.2 Function clauses

While node construction and destruction are primitive operations in $\Delta$, we'll refrain ourselves from delving into such details in the control flow graphs of our programs. This is because by the semantics of $\Delta$, node construction and destruction always terminates. Instead, we'll let function clauses define primitive program points. The expression of a given clause can make calls to its enclosing, or some other function. Such calls are represented by transfer of control, that is, edges between clauses.

**Definition 4.1.1.** *Given a program $r = \langle F, x\rangle$ we define a control flow graph $G = \langle C, E\rangle$, where*

$$C = \left\{c \mid f = \left\langle v, C_f \right\rangle \in F \wedge c \in C_f\right\},$$

*and*

$$E = \{\langle c_s, c_t, x\rangle \mid c_s = \langle v_s, p_s, x_s\rangle \in C \wedge c_t = \langle v_t, \_, \_\rangle \in C \wedge x \in \mathbb{X} \wedge \langle v_t, x\rangle \Subset x_s\}.$$

**Definition 4.1.2.** *Given a control transition graph $G = \langle C, E \rangle$, we refer to a directed edge $e \in E$ as a call. We refer to $G$ as a call graph and given any call $\langle c_s, c_t, x \rangle \in C$, we say that $c_s$ is the* source *clause, $c_t$ is the* target *clause, and $x$ is the call argument.*

**Lemma 4.1.1.** *Given a call graph $G = \langle C, E \rangle$, $C$ and $E$ are both finite.*

*Proof.* Follows from the semantics of $\Delta$ and Definition 4.1.1 (23). □

**Definition 4.1.3.** *Given a call graph $G = \langle C, E \rangle$ we refer to the list $\langle c_1, c_2, \_ \rangle, \langle c_2, c_3, \_ \rangle, \ldots, \langle c_{n-1}, c_n, \_ \rangle \in E$ as a "path".*

Definition 4.1.1 (23) might strike you as odd, since by the semantics of $\Delta$, when we make a function call to some function $f$, we will iteratively consider the list of clauses contained in the function, looking for one which accepts the argument. While this is true, we will merely concern ourselves with those control transitions, where a change in the values of the program variables can occur. The transitions between the clauses of a single definition, which occur when a function call argument is matched to a clause, call them *fail transitions*, do not change the value of the argument but merely propagate it. Hence, fail transitions are irrelevant to our analysis, so long as we have an edge from the source clause to each possible target clause, which is exactly what Definition 4.1.1 (23) states.

We use a triplet to represent a call in order to ensure that calls with different arguments get different edges. This is important to clauses with expressions where multiple calls to the same function are made. In particular, the different calls might modify the values in different ways.

### 4.1.3 Order of evaluation

Definition 4.1.1 (23) indicates that we disregard the order of evaluation of the arguments. This too, is intentional. In particular, if all the call cycles terminate, then so will all the evaluations. We show this with a few examples below and prove formally when we discuss Theorem 4.2.1 (27).

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, $\texttt{f} \neq \texttt{g}$, 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 $\Delta$ 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`. The order of evaluation of `(g a)` and `(h b)` is insignificant wrt. to each another, as with function calls separated by construction. However, 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 initially regard this as insignificant for mere simplicity.

### 4.1.4 An example

We can now draw a control flow graph for the program define in Listing 4.1 (24) as shown in Figure 4.1 (25).

```
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
```

**Listing 4.1:** A sample $\Delta$ program, always returning `0.0.0`.

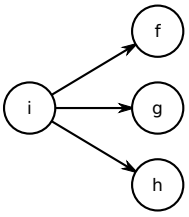


**Figure 4.1:** A control flow graph for the $\Delta$ program in Listing 4.1 (24). The graph does not explicitly specify back-propagation of control, if any.

### 4.1.5 Disregarding back-propagation

It is worth noting that in Figure 4.1 (25), the clauses that make no function calls have out-degree 0. Technically, these functions do transfer control, in particular, 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 won't be concerned with any exact values.

The thing with back-propagation is that forward-propagation after back-propagation of a call cannot occur due to the way $\Delta$ 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.

### 4.1.6 Call cycles

**Definition 4.1.4.** *Given a call graph $G = \langle C, E \rangle$, a call cycle is a path $z = \langle c_1, c_2, \_ \rangle, \langle c_2, c_3, \_ \rangle, \ldots, \langle c_{n-1}, c_n, \_ \rangle \in E$ s.t. $c_1 = c_n$.*

**Theorem 4.1.2.** *Given program $r = \langle F, x \rangle$ has a corresponding acyclic graph $G = \langle C, E \rangle$, then the program terminates.*

*Proof.* Assume for the sake of contradiction that the program does not terminate. Then one of the acyclic paths in the graph must not terminate. Since the set $E$ is finite by Lemma 4.1.1 (24), then any path in $G$ must be finite. Hence, one of the primitive operations, i.e. construction, destruction, comparison, binding, function call, etc., must not terminate. By the semantics of $\Delta$ this is absurd. □

**Definition 4.1.5.** *Given a call graph $G = \langle C, E \rangle$, we refer to the set of calls that participate in a call cycle as recursive clauses and all other clauses as terminal clauses.*

**Definition 4.1.6.** *If a program choses some edge $e$ over another edge $e'$, we say that the program branches off to a new path starting with edge $e$.*

**Theorem 4.1.3.** *If a program branch takes a path $k$ consisting solely of terminal clauses, the branch terminates.*

*Proof.* If a program branches off to a path $k$ and $k$ consists solely of terminal clauses, then the program may be regarded as having branched off into a new program with a call graph consisting solely of the clauses visited by $k$ and edges in $k$. Such a call graph is acyclic due to Definition 4.1.5 (25), and by Theorem 4.1.2 (25) is finite. The branch must therefore terminate. □

**Corollary 4.1.4.** *A program terminates if all of its branches terminate.*

Corollary 4.1.4 (25) allows us to disregard all the clauses of a call graph that do not participate in call cycles.

**Definition 4.1.7.** *Given a call graph $G = \langle C, E \rangle$, and the set of call cycles $Z$ in $G$, we define a call graph $G^r = \langle C^r, E^r \rangle$ to be the "recursive call graph", where $C^r = \{c \mid c \in C \wedge c \Subset Z\}$ and $E^r = \{e \mid e \in E \wedge e \Subset Z\}$. WLOG, all call graphs we refer to from this point on will be recursive call graphs.*

### 4.1.7 Relation of call graphs to conventional static call graphs

Conventionally, a static call graph is a graph over all the calls made by a program at runtime. Such a graph has an infinite number of edges for any nonterminating program in Δ. Hence, deducing the halting property can be rephrased as determining whether the static call graph has an infinite number of edges.

### 4.1.8 Visualization

When describing size-change termination we'll often revert to examples. Here we will make use of a few conventions wrt. listings and graphs s.t. a call graph for any given program is easy to visualize.

#### Multiple calls to the same function

An edge $e \in E$ in a call graph $G = \langle C, E \rangle$ was defined to be the tuple $\langle c_1, c_2, x \rangle : C \times C \times \mathbb{X}$. While this ensures to distinguish calls to the same function with different arguments from the same expression, it is not particularly friendly to the eye to visually annotate each edge with an expression.

Instead, we adopt the convention of disjunctively labelling all the function calls of an expression as in [Lee et al., 2001]. We'll do this both in listings and in visualizations. Refer to Listing 4.2 (26) and Figure 4.2 (26) for an example.

```
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
```

**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.

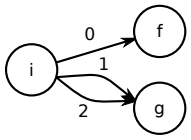


**Figure 4.2:** A control flow graph for the Δ program in Listing 4.2 (26).

#### Multiple clauses

As multiple clauses denote different nodes in the call graph, we would also like to visually distinguish the nodes while keeping a relation to the original listing. Hence, each clause of a function in the program listing will be labeled with a label prefixed with the function name and postfixed with the clause index starting at 0. Refer to Listing 4.3 (26) and its corresponding call graph in Figure 4.3 (26) for an example.

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

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

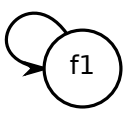


**Figure 4.3:** A control flow graph for the program defined in Listing 4.3 (26).

As a more complex example, consider the call graph for the program `reverse` introduced in § 3.6 (20). The program is repeated in annotated form in Listing 4.4 (27), and its corresponding call graph is shown in Figure 4.4 (27).

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

**Listing 4.4:** An annotated version of the program `reverse` introduced in § 3.6 (20).

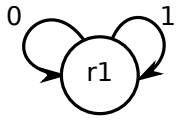


**Figure 4.4:** A control flow graph for the Δ program in Listing 4.4 (27).

## 4.2 Size-change termination principle

Consider the program in Listing 4.3 (26) and its corresponding call graph in Figure 4.3 (26). Without any further information about the calls, the program seemingly loops indefinitely. However, there are some things that we can deduce about the control transitions.

**Theorem 4.2.1.** *If every call cycle in a given program reduces a value of a 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*[Lee et al., 2001]. 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.

**Definition 4.2.1.** *For a given call graph $G = \langle C, E \rangle$, let a size relation be the set*

$$\Phi = \left\{ \langle c_s, c_t, x, v_s, v_t, \rho \rangle \;\middle|\; \begin{array}{ll} & \langle c_s, c_t, x \rangle \in E \\ \wedge & c_t = \langle \_, p_t, \_ \rangle \in C \\ \wedge & v_s, v_t \in \mathbb{V} \\ \wedge & v_s \Subset x \wedge v_t \Subset p_t \\ \wedge & \rho \in \{\bot, <, \leq\} \end{array} \right\}$$

This definition implies that for each call cycle there may be multiple variables[1] to ensure reduction for. However, as there is only a finite number of variables in any given pattern and any given expression, we can define these as separate cycles.

**Definition 4.2.2.** *Given a recursive call graph $G^r = \langle C^r, E^r \rangle$, let $G^u = \langle C^u, E^u \rangle$ be a "unary recursive call graph", where*

$$E^u = \left\{ \langle c_s, c_t, x, v_s, v_t \rangle \;\middle|\; \begin{array}{ll} & \langle c_s, c_t, x \rangle \in E^r \\ \wedge & c_t = \langle \_, p_t, \_ \rangle \in C^r \\ \wedge & v_s, v_t \in \mathbb{V} \\ \wedge & v_s \Subset x \wedge v_t \Subset p_t \end{array} \right\},$$

*and $C^u = \{c \mid c \in C^r \wedge c \Subset E^u\}$.*

**Theorem 4.2.2.** *A unary recursive call graph $G^u$ has a finite number of edges.*

*Proof.* Follows from Definition 4.2.2 (27) and the semantics of Δ. □

[1] We'll define what we mean by a variable more formally in Definition 4.2.4 (28).

Definition 4.2.2 (27) forces us to redefine the concept of a call cycle, as a sequence of clauses may have multiple cycles in $G^u$.

**Definition 4.2.3.** *Given a recursive call graph $G^r = \langle C^r, E^r \rangle$, let $Z^r$ denote the set of call cycles in $G^r$, and $Z^u$ the set of call cycles in $G^u$, then*

$$Z^u = \bigcup_{z^r \in Z^r} \left\{ \langle c_s, c_t, x, v_s, v_t \rangle \;\middle|\; \begin{array}{ll} & \langle c_s, c_t, x \rangle \in z^r \\ \wedge & c_t = \langle \_, p_t, \_ \rangle \in C^r \\ \wedge & v_s, v_t \in \mathbb{V} \\ \wedge & v_s \Subset x \wedge v_t \Subset p_t \end{array} \right\}.$$

Definition 4.2.2 (27) allows us now to more formally define what we've thus far meant as a variable that changes value from call to call.

**Definition 4.2.4.** *Given a unary recursive call graph $G^u$, every call cycle $z$ in $G^u$, changes exactly one variable, call it "cycle variable", or $v_z$.*

Hence, while a variable may have different names in different clauses, we've defined an overall variable for every call cycle, and want to ensure that this variable is reduced in every iteration of the call cycle.

**Theorem 4.2.3.** *We can WLOG to termination analysis limit our attention to programs that bind at most one variable in every clause.*

*Proof.* Let a recursive call graph $G^r$ have a set of call cycles $Z^r$, and a corresponding unary recursive call graph $G^u$ with a set of call cycles $Z^u$. By Definition 4.2.3 (28) and Definition 4.2.1 (27), every call cycle in $Z^r$ reduces a value iff every call cycle in $Z^u$ reduces a value. We can hence limit our attention to deducing if each call cycle in $Z^u$ reduces a value. □

**Definition 4.2.5.** *Let $G^1$ denote a recursive call graph of a program where each clause binds at most one variable.*

Given Theorem 4.2.3 (28) we can return to the old definition of a call graph as per Definition 4.1.7 (25) and call cycle as per Definition 4.1.4 (25). This demands a simplification of the size relation $\Phi$.

**Definition 4.2.6.** *For a given $G^1 = \langle C, E \rangle$ of a program, let a size relation be the set*

$$\Phi^1 = \{ \langle c_s, c_t, x, \rho \rangle \mid \langle c_s, c_t, x \rangle \in E \wedge \rho \in \{\bot, <, \leq\} \}.$$

**Definition 4.2.7.** *Given a call graph $G$ with a call cycle $z$, a call $\langle c_s, c_t, x \rangle \in z$ is,*

1. *A "decreasing call" iff $\langle c_s, c_t, x, < \rangle \in \Phi^1$.*
2. *A "nonincreasing call" iff $\langle c_s, c_t, x, \leq \rangle \in \Phi^1$.*
3. *An "undteremined call" iff $\langle c_s, c_t, x, \bot \rangle \in \Phi^1$.*

**Lemma 4.2.4.** *A call cycle $z = \langle c_1, c_2 \rangle, \langle c_2, c_3 \rangle, \ldots, \langle c_{n-1}, c_n \rangle$ reduces a value on each iteration iff*

$$\left( \forall\ \langle c_i, c_j \rangle \in z\ \left( \langle c_i, c_j, \_, < \rangle \in \Phi^1 \right) \vee \left( \langle c_i, c_j, \_, \leq \rangle \in \Phi^1 \right) \right) \wedge \left( \exists\ \langle c_i, c_j \rangle \in z\ \langle c_i, c_j, \_, < \rangle \in \Phi^1 \right).$$

*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 call must've increased the value, which is an absurdity. If it stays the same then none of the participating control transitions have neither increased nor decreased the value, also an absurdity. □

### 4.2.1 Deducing $\Phi^1$

**Definition 4.2.8.** *Given a call graph* $G^1 = \left\langle C^1, E^1 \right\rangle$ *and a call* $\langle c_s = \langle \_, p_s, \_ \rangle, c_t = \langle \_, p_t, \_ \rangle, x \rangle \in E^1$, *let* $v_{p_s}$ *denote the variable s.t.* $v_{p_s} \in \mathbb{V} \wedge v_{p_s} \Subset p_s$, *and let* $v_{p_t}$ *denote the variable s.t.* $v_{p_t} \in \mathbb{V} \wedge v_{p_t} \Subset p_t$. *If some* $p \in \mathbb{P}$ *contains no variables, then* $\forall\, v \in \mathbb{V}\ v \neq v_p$.

**Definition 4.2.9.** *Given a call graph* $G^1 = \left\langle C^1, E^1 \right\rangle$ *and a call* $\langle \_, \_, x \rangle \in E^1$, *let* $b_x \in \mathbb{B}$ *denote the actual value x evaluates to.*

This is a three-step process, for any given call $\langle c_s, c_t, x \rangle \in E^1$, (1) deduce a size relation between $v_{p_s}$ and $b_x$, (2) deduce a size relation between $b_x$ and $v_{p_t}$, (3) given (1) and (2), deduce a size relation between $v_{p_s}$ and $v_{p_t}$, allowing us to update the relevant $\Phi^1$.

In the discussions below we assume to be considering a single call $e \in E^1$, hence the variables $c_s$, $c_t$, $x$, $b_x$, $p_s$, $p_t$, $v_{p_s}$, $v_{p_t}$ and $V_x$ are available without further details.

#### Source to argument

Since $\Delta$ is a call-by-value language, the source evaluates $x$, and generates some $b_x \in \mathbb{B}$ as the actual argument for the call. The expression $x$ is by definition a nested construction of either some concrete value, some variable $v_{p_s}$, or a nested function call. Without further regard of nested function calls, this implies that a size relation can be deduced between the $v_{p_s}$ and $b_x$.

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 $c_s$ 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 auxiliary variables and the variable $v_{p_t}$.

Due to nested functions calls being represented as variables with unknown values, a precise size displacement between the bound variable $v_{p_s}$ and the generated argument $b$ is not feasible. However, we can deduce a *safe* displacement estimate.

**Definition 4.2.10.** *A safe displacement estimate between the values* $b_1, b_2 \in \mathbb{B}$ *is a value* $n \in \mathbb{N}$ *s.t.* $b_1 + n \leq b_2$.

**Definition 4.2.11.** *Given a call* $\langle c_s, c_t, x \rangle$, *we construct the expression* $p_x$ *where we replace all* $\langle \_, \_ \rangle \Subset x$ *with an auxiliary variable* $v \neq v_{p_s}$. *We group those auxiliary variables in the set* $V_x$, *that is,*
$V_x = \left\{ v \mid v \in \mathbb{V} \wedge v \Subset p_x \wedge v \neq v_{p_s} \right\}$.

**Lemma 4.2.5.** *Given a call* $\langle c_s, c_t, x \rangle$, *the expression* $p_x$ *is interchangeable with a pattern. Hence, we say that* $p_x \in \mathbb{P}$.

*Proof.* The expression $p_x$ contains no function calls. Function calls is what syntactically distinguishes expressions from patterns by the semantics of $\Delta$. □

For instance, consider an expression $x$ such as `(g c.(f c))`, where $v_{p_s} =$ `c`. Assume that we're considering the call to the function `g`. By Definition 4.2.11 (29), we replace the expression `c.(f c))` with an expression like `c.d`, and let $V_x = \{$`d`$\}$. When this argument evaluates to some value $b_x \in \mathbb{B}$, then we can deduce the set of safe displacement estimates $\{b_x >$ `c`$\}$.

Consider now a target clause with a pattern like `e.0`. The question henceforth is how do we draw the relationship that `c` $=$ `e`. In other words, that the call neither decreases nor increases the call cycle value. We can perform a corresponding analysis on the pattern declaration and deduce the set of conditions that will hold if pattern matching succeeds, indeed, $\{b_x \geq$ `e`$\}$. The participation of `c` in the same kind of relations as `e`, does not alone imply that `c` $=$ `e`, since the property that $b_x =$ `c.d`, where `d` $\geq 0$, is lost.

On the other hand, if we had to formally define the relation that had to be drawn between $v_{p_s}$ and $b_x$ for any given $\langle c_s, c_t, x \rangle$, this would be a relation between $b_s$ and some sort of "abstract patterns", as e.g. $b_s \geq$ `c.0` where `c` $\geq 0$.

To simplify the entire process, instead of deducing actual size relation between $v_{p_s}$ and $b_x$, we can simply turn $x$ into such an abstract pattern to begin with (Definition 4.2.11 (29)). The actual size relations are hence kept and can be deduced at a later stage in the process.

**Source to target**

In the section below we will focus on defining a set of rules that allow us to deduce a relation between $v_{p_t}$ and $v_{p_t}$, directly given some $p_x$ which has been constructed as by Definition 4.2.11 (29).

**Definition 4.2.12.** *Initially, let* $\langle c_s, c_t, x, \bot \rangle \in \Phi^1$.

In the discussion below, we use $\Phi^1$ in a manner similar to the state $\sigma$ in the definition of the semantics of $\Delta$ in § 3.4 (16). However, $\Phi^1$ is now a binary memory element, requiring both a source name and a target name (in that order).

In particular, we would like a set of rules for modifying $\Phi^1$, given some argument expression $p_x$ and some target pattern $p_t$. We start by defining a summoning rule, dividing the rules up into sub-rules:

$$\frac{\begin{array}{ll} & \left\langle \mathrm{A}, p_x, p_t, \Phi^1 \right\rangle \to \Phi_1^1 \\ \vee & \left\langle \mathrm{B}, p_x, p_t, \Phi^1 \right\rangle \to \Phi_1^1 \\ \vee & \left\langle \mathrm{C}, p_x, p_t, \Phi^1 \right\rangle \to \Phi_1^1 \\ \vee & \left\langle \mathrm{D}, p_x, p_t, \Phi^1 \right\rangle \to \Phi_1^1 \\ \vee & \left\langle \mathrm{E}, p_x, p_t, \Phi^1 \right\rangle \to \Phi_1^1 \end{array}}{\left\langle p_x, p_t, \Phi^1 \right\rangle \to \Phi_1^1} \tag{4.1}$$

One of the simpler cases is when the abstract pattern $p_x$ is simply `0`, or $v \in V_x$. Since $v_{p_s}$ does not participate in $p_x$, then no relation needs to be drawn between $v_{p_s}$ and $v_{p_t}$ because no such relation exists.

$$\frac{\left(p_x = 0 \vee \left(p_x = v \wedge v \neq v_{p_s}\right)\right) \wedge \Phi^1 \to \Phi_1^1}{\langle \mathrm{A}, p_x, p_t, \Phi^1 \rangle \to \Phi_1^1} \tag{4.2}$$

This has a symmetrical case. Indeed when $p_t$ is neither a destruction, nor any name $v_{p_t}$, that is, it is either `_` or `0`. In this case, no relation needs to be drawn between $v_{p_t}$ and $v_{p_s}$ as such a relationship is absent.

$$\frac{(p_t = 0 \vee p_t = \_) \wedge \Phi^1 \to \Phi_1^1}{\langle \mathrm{B}, p_x, p_t, \Phi \rangle \to \Phi_1^1} \tag{4.3}$$

If $p_t$ is the name pattern $v_{p_t}$, the matters get a bit more complicated:

1. If $p_x$ is some node, then all the variables that occur in $p_x$, will be strictly less than $v_{p_t}$ by the semantics of $\Delta$. However, we are not concerned with this relation, as this indeed would be an increasing call rather than a decreasing or nonincreasing call.
2. If $p_x$ is $v_{p_s}$, then the values of these corresponding variables will be *equivalent*. However, we're not concerned with exact equivalence, and simply mark this relationship with the weaker, but still sound relation, $\leq$:

$$\frac{p_t = v_{p_t} \wedge p_x = v_{p_s} \wedge \left\langle \Phi^1 \left(c_s, c_t, x\right) \mapsto \leq \right\rangle \to \Phi_1^1}{\langle \mathrm{C}, p_x, p_t, \Phi^1 \rangle \to \Phi_1^1} \tag{4.4}$$

If $p_t$ is a destruction and $p_x = v_{p_x}$, then we can safely say that all the variables that occur in $p_t$, are all strictly less than the variable in $v_{p_x}$:

$$\frac{p_t = p_{t_1} \cdot p_{t_2} \wedge p_x = v_{p_s} \wedge v_{p_t} \Subset p_t \wedge \left\langle \Phi^1 \left(c_s, c_t, x\right) \mapsto < \right\rangle \to \phi_1^1}{\langle \mathrm{D}, p_x, p_t, \Phi^1 \rangle \to \Phi_1^1} \tag{4.5}$$

If both $p^t$ and $p^s$ are a destructions, then the following recursive rule applies:

$$\frac{p_t = p_{t_1} \cdot p_{t_2} \wedge p_s = p_{x_1} \cdot p_{x_2} \wedge \left\langle p_{t_1}, p_{x_1}, \Phi^1 \right\rangle \to \Phi_2^1 \wedge \left\langle p_{t_2}, p_{x_2}, \Phi_2^1 \right\rangle \to \Phi_1^1}{\langle \mathrm{E}, p_x, p_t, \Phi^1 \rangle \to \Phi_1^1} \tag{4.6}$$

## 4.3 Graph annotation

Given the discussion in § 4.2 (27), we can now deduce the calls from the recursive clause of the program in Listing 4.4 (27) are both decreasing, yielding the conclusion that the program `reverese` terminates.

Graphically, we won't define any special syntax for functions that have clauses where more than one variable is bound, instead, we will merely define a syntax for the set of size relations $\{<, \leq, \perp\}$. Figure 4.5 (31) shows an example.

**Definition 4.3.1.** *Given a call graph* $G^1 = \left\langle C^1, E^1 \right\rangle$*, and its deduced size relation* $\Phi^1$*, let the graph* $G^s = \left\langle C^1, \Phi^1 \right\rangle$ *be known as a size-change graph. Note, that by the rules defined in § 4.2.1 (30) and Definition 4.2.1 (27),*

$$\left(\forall\ \langle c_s, c_t, x\rangle \in E^1\ \exists\ \langle c_s, c_t, x, \_\rangle \in \Phi^1\right) \wedge \left(\exists\ \langle c_s, c_t, x, \rho\rangle \in \Phi^1 \longrightarrow \nexists \left\langle c_s, c_t, x, \rho'\right\rangle \in \Phi^1\ \rho \neq \rho'\right).$$

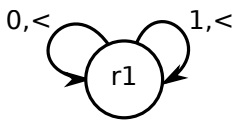


**Figure 4.5:** A size-change graph for the program in Listing 4.4 (27).

## 4.4 The algorithm

We define the size-change termination algorithm as follows:

**Definition 4.4.1.** *Given a program r and its corresponding size-change graph G, yield "halts" if all the call cycles in G are monotonically decreasing, and "unknown" otherwise.*

Although it may seem odd to speak of termination of a method for which no particular implementation has been specified, we'll regard the finiteness of the call graph of a program as a proof of the termination of any method that takes a finite amount of time to build the call graph, and takes a finite amount of time to analyze the size-change property of any possible cycle in the call graph.

**Theorem 4.4.1.** *Size-change termination is a sound and complete solution to Definition 1.1.2 (5).*

*Proof.* Follows from Theorem 4.2.1 (27) and Theorem 4.2.2 (27). □

# Chapter 5

# Shape-Change Termination

Size-change termination, despite being simple is powerful enough to determine the halting property for a large class of programs. Many authors have extended the principle to even wider classes of programs, e.g. extending it to handle initially not well-founded data types [Avery, 2006], applying it in imperative programming languages [Berdine et al., 2006], etc.

One trouble with size-change termination as described in the previous chapter is Lemma 4.2.4 (28). This lemma makes size-change termination weak in the sense that the overall shape changes in a given call cycle are *not* considered, and instead, call cycles are constrained to monotonically decreasing call cycles. However, there may be programs that have calls, or even call cycles, that in terms of size, increase a value for a finite amount of time, until some condition is met, or as in the case of $\Delta$, a value has some particular shape.

Consider the program in Listing 5.1 (33) as an example of a program for which regular size-change termination is unable to determine the halting property, while the property itself would seem fairly simple to deduce. This is a sample program where some value is increased in terms of size in a call cycle, but only until the value matches a certain shape, the shape required by a terminal clause.

```
1 f0: f a.b.c.d := a
2 f1: f a := f a.0
3 f input
```

**Listing 5.1:** A terminating program with a call cycle where a value is temporarily increased.

The extension proposed in this chapter is to be able to determine the halting property for such a class of programs without reducing size of the class of programs for which size-change termination can already deduce the halting property.

In the discussion below we continue the assumptions from § 4 (23). In particular, all clauses in a program are unary and bind at most one variable. Also, we can safely disregard terminal clauses in call graphs.

## 5.1 The class of programs considered

Before we can speak of extending size-change termination to determine the halting property for programs in the same class as Listing 5.1 (33), we need to formally define that class.

Actual conditions in $\Delta$ can only be expressed in terms of patterns in function clauses. Hence, we disregard programs that rely on equality or size comparison conditions for termination, since this type of programs will often already be covered by regular size-change termination, and if not, they at the very least come down to recursive pattern matching.

As an example of a program where size-change termination is already prevalent, consider a program that finds the $n^{th}$ Fibonacci number as the one already presented in § 3.6 (20). The function `fibonacci-aux` seemingly increases a value until a condition is met, in particular, until we count one of the arguments down to `0`. However, due to the fact that we count that we decrease the value of that

argument in *recursive* clause of the `fibonacci-aux` functions, the halting property is certainly already deducible by regular size-change termination.

Instead, we turn our attention to simpler programs, ones that rely solely on conditions defined in terms of patterns. Consider again the program in Listing 5.1 (33). The function `f` has only one terminal clause, the one that accepts a shape as in Figure 5.1 (34). If the function argument has any other shape, i.e. either a shape as in Figure 5.2 (34), Figure 5.3 (34) or Figure 5.4 (34), then the recursive clause $f_1$ is chosen. For any argument $b \in \mathbb{B}$, the clause $f_1$ replaces the right-most child of the value, which is always `0`, with a node.

For instance, the smallest possible argument $b \in \mathbb{B}$ is `0`. If passed such a value, $f_1$ transforms it into a value that has a shape that corresponds to Figure 5.3 (34), which in turn transforms the value into one that matches Figure 5.4 (34), which in turn transforms the value into one that matches Figure 5.1 (34), i.e. the terminal clause. What's more, there are infinitely many other values that will match the shape Figure 5.3 (34), and for each of them, the clause $f_1$ will transform them into values that match Figure 5.4 (34), which will transform them into values that match Figure 5.1 (34).

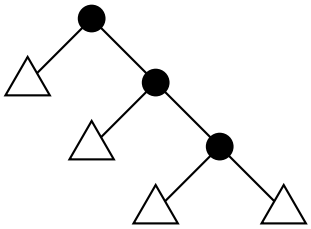

**Figure 5.1:** The shape that the clause $f_0$ in Listing 5.1 (33) will accept.



**Figure 5.2:** The pattern `0`.

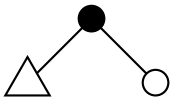

**Figure 5.3:** The pattern `a.0`.

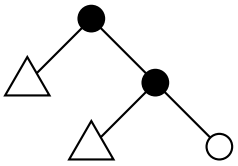

**Figure 5.4:** The pattern `a.b.0`.

We shall henceforth say that a clause such as $f_1$ changes the shape of any argument $b$ to *eventually* match the shape corresponding to a pattern of the terminal clause $f_0$. The task then becomes to determine for each call cycle in a program whether it changes the shape of the argument to eventually match some terminal clause.

## 5.2 Preliminaries

Before we continue with this extension we can make a few important observations based on the semantics of function clauses in $\Delta$.

### 5.2.1 Deducing leafs

The `.` operator in the patterns of function clauses in $\Delta$ is right-associative. Hence, a pattern of the form `a.b.c.d` is the same as `a.(b.(c.d))`. This implies that we can always construct a parenthesized version of any valid pattern, indeed this is required to keep the syntax unambiguous. This associativity can be overridden by the conventional use of parentheses, e.g. a pattern like `(a.b).c.d` is the same as `(a.b).(c.d)`.

Consider the function defined in Listing 5.2 (35). If $f_0$ and $f_1$ both fail to accept some argument $b \in \mathbb{B}$, then $b$ must match the pattern $0 \cdot b'$, where $b' \geq 0$, that is, on entry to $f_2$, `d` is *always* bound to `0`, and `e` is always bound to some $b' \geq 0$.

*Proof.* Otherwise, either $f_0$ or $f_1$ would've matched. □

```
1 f0: f 0 := 0
2 f1: f (a.b).c := 0
3 f2: f d.e := 0
```

**Listing 5.2:** A sample program for showing 0-deduction.

Such a deduction is not always unambiguous as the function in Listing 5.3 (35) exhibits. Here, if $g_0$ and $g_1$ both fail to accept some argument $b \in \mathbb{B}$, then the shape of $b$ is either $0 \cdot b'$ or $b' \cdot 0$ where $b \geq 0$. However, one thing is certain, and that is that $b$ can't have the shape $b' \cdot b''$ where $b' > 0$ and $b'' > 0$.

```
1 g0: g 0 := 0
2 g1: g (a.b).(c.d) := 0
3 g2: g e.f := 0
```

**Listing 5.3:** A sample program where 0-deduction is ambiguous.

### 5.2.2 Disjoint shapes

**Definition 5.2.1.** *Given two shapes, $s_1, s_2 \in \mathbb{S}$, we say that $s_1$ and $s_2$ are disjoint, or $s_1 \cap s_2 = \emptyset$, iff given $B_1 = \{b \mid b \in \mathbb{B} \wedge b \succ s_1\}$ and $B_2 = \{b \mid b \in \mathbb{B} \wedge b \succ s_2\}$ it holds that $B_1 \cap B_2 = \emptyset$.*

**Definition 5.2.2.** *Given a shape $s \in \mathbb{S}$, let $S_s^d = \left\{ s^d \mid s^d \in \mathbb{S} \wedge s \cap s^d = \emptyset \wedge \forall\, s_1^d \in S_s^d \backslash \{s^d\}\ s^d \cap s_1^d = \emptyset \right\}$ be the set of shapes disjoint with $s$ and with each other.*

**Theorem 5.2.1.** *Given a shape $s \in \mathbb{S}$, the set $S_s^d$, is finite.*

*Proof.* The proof is two-fold,

1. Given a shape $s \in \mathbb{S}$, there is a shape $s' \in \mathbb{S}$ for every leaf and every node in $s$ s.t. $s \cap s' = \emptyset$. In particular, for every leaf in $s$, there is a shape $s'$, that is otherwise equal to $s$, but in place of the particular leaf in $s$, it has a node with two triangles as children. Likewise, for every node in $s$, there is a shape $s'$, that is otherwise equal to $s$, but in place of the particular node in $s$, there is a leaf. Any other shapes wouldn't be disjoint with either $s$ or the shapes already considered. It is easy to see that all such $s' \in \mathbb{S}$ are also pairwise disjoint.
2. For any given shape $s \in \mathbb{S}$ the number of nodes and leafs is finite by Definition 3.2.10 (13).

□

**Definition 5.2.3.** *Given a pairwise disjoint set of shapes $S_1$, and another pairwise disjoint set of shapes $S_2$, let*

$$S_1 \Cup S_2 = \left\{ s \;\middle|\; \begin{array}{ll} & \left(s \in S_1 \wedge \left(\exists\, s' \in S_2\ s \cap s' \neq \emptyset \longrightarrow s \succ s'\right)\right) \\ \vee & \left(s \in S_2 \wedge \left(\exists\, s' \in S_1\ s \cap s' \neq \emptyset \longrightarrow s \succ s'\right)\right) \end{array} \right\}.$$

In particular, the set $S_1 \Cup S_2$ is the union of the two sets where for any pair of patterns that match one another, the one with the most nodes and leafs is chosen.

### 5.2.3 Plausible shapes

**Definition 5.2.4.** *We say that a variable $v \in \mathbb{V}$ with some value $b \in \mathbb{B}$ has a set of plausible shapes $S_v$ iff $\exists\, s \in S_v\ b \succ s \wedge \left(\forall\, s' \in S_v \backslash \{s\}\ b \not\succ s'\right)$.*

**Corollary 5.2.2.** *By Definition 3.2.12 (13), given a variable $v \in \mathbb{V}$ with some unknown value $b \in \mathbb{B}$, the set of plausible shapes $S_v = \{\langle \triangle \rangle\}$.*

**Corollary 5.2.3.** *Given a clause $\langle v, p, x \rangle \in \mathbb{C}$, and a value $b \in \mathbb{B}$, if $b \succ p$, then $S = \{p\}$*[1].

[1] Patterns are shapes as Definition 3.3.11 (15).

Consider henceforth a function call $f = \langle v, C \rangle$. Given a tuple $\langle i, j \rangle$ from the set $\{\langle i, j \rangle \mid 0 < i < |C| \wedge j = i + 1\}$, consider also the clauses $c_i = \langle v_i, p_i, x_i \rangle, c_j = \langle v_j, p_j, x_j \rangle \in C$. Let $B_i = \{b \mid b \in \mathbb{B} \wedge b \succ p_i\}$ and $B_j = \{b \mid b \in \mathbb{B} \wedge b \succ p_j\}$ denote the sets of values that the clauses $c_i$ and $c_j$ accept, respectively.

**Corollary 5.2.4.** *Given the semantics of $\Delta$, i.e. that clause $c_i$ will be considered before $c_j$, we can deduce that if $c_j$ indeed accepts a given value $b \in \mathbb{B}$, that $c_i$ hence rejected, the set of plausible values for $b$ must be $B_j - B_i$.*

**Corollary 5.2.5.** *Given a set $S_i^d$ of clauses disjoint with the shape corresponding to pattern $p_i$, let $S'_c = S_i^d \Cup p_j$ and $S_c = S'_c - \{s \,|\, s \in S'_c \wedge s \not\succ p_j\}$. Due to Corollary 5.2.4 (36), the clause $c_j$ could've been expressed in terms of the list of the following list of clauses:*

$$C_j = \langle v_j, p_1, x_j \rangle, \langle v_j, p_2, x_j \rangle, \ldots, \langle v_j, p_n, x_j \rangle,$$

*where and $p_1, p_2, \ldots, p_n \subset S_c$ and $n = |S_c|$.*

**Theorem 5.2.6.** *A function $f = \langle v, C \rangle$ can be defined in such a way that $B_i \cap B_j = \emptyset$.*

*Proof.* Follows from the semantics of $\Delta$, Definition 3.4.3 (17) and Corollary 5.2.5 (36). □

Given that all clauses are unary and patterns correspond to shapes, we henceforth say that clauses, like the shapes that their patterns represent can be "disjoint".

**Definition 5.2.5.** *Given the clauses $c_1 = \langle \_, p_1, \_ \rangle, c_2 = \langle \_, p_2, \_ \rangle \in \mathbb{C}$, we say $c_1 \cap c_2 = \emptyset$ iff $p_1 \cap p_2 = \emptyset$.*

**Definition 5.2.6.** *All programs henceforth considered have function definitions with clause lists $C = c_1, c_2, \ldots, c_n$ s.t. $\forall\ \langle i, j \rangle \in \{\langle i, j \rangle \mid 0 < i < n \wedge i < j \leq n\}\ c_i \cap c_j = \emptyset$.*

**Lemma 5.2.7.** *Given a program $r = \langle F, x \rangle$, it can be transformed into $r' = \langle F', x' \rangle$, s.t. $|F'| = 1$.*

*Proof.*

1. Let $id : \mathbb{V} \to \mathbb{B}$ be a bijective function that yields a unique $b \in \mathbb{B}$ for a unique $v \in \mathbb{V}$. The existence of such a function can be argued for by the fact that both $b \in \mathbb{B}$ and $v \in \mathbb{V}$ are finite sequences of finite alphabets. We'll omit the formal proof as the property is fairly easy to see.

2. Let $unite : \mathbb{V} \times \mathbb{X} \to \mathbb{X}$ be a bijective function that given a name $v \in \mathbb{V}$ and an expression $x \in \mathbb{X}$ replaces all $\langle v_a, x_a \rangle \Subset x$ with the tuple $\langle v, id(v_a) \cdot x_a \rangle$.

3. Let $F' = \{\langle v, C \rangle\}$, where $v \in \mathbb{V}$ is some arbitrary name, and

$$C = \left\{ \langle v, p_c, x_c \rangle \;\middle|\; \begin{array}{ll} & \left\langle v_f, C_f \right\rangle \in F \\ \wedge & \left\langle \_, p_{f_c}, x_{f_c} \right\rangle \in C_f \\ \wedge & p_c = id(v_f) \cdot p_{f_c} \\ \wedge & x_c = unite(v, x_{f_c}) \end{array} \right\}$$

□

**Definition 5.2.7.** *All programs considered henceforth can WLOG be considered to be programs with a single function, or simply $r = \langle C, x \rangle : [\mathbb{C}] \times \mathbb{X}$.*

## 5.3 Shape-change termination

In the next section, unless otherwise stated, we consider recursive call graphs as defined in Definition 4.1.7 (25), albeit for programs as discussed in Definition 5.2.6 (36) and Definition 5.2.7 (36).

**Definition 5.3.1.** *Given a program $r = \langle F, x \rangle$, where $|F| = 1$, we define a "shape-change graph" to be the recursive call graph as by Definition 4.1.7 (25). We also define a "shape cycle" to be a cycle in that graph as by Definition 4.1.4 (25).*

While this recycling of definitions might seem odd at first, the difference is the underlying program and hence its call graph. The underlying program has exactly one function, and every clause of that function has a unique a pattern, i.e. shape. Hence, the name shape cycle, as a cycle between the clauses of the program now indeed constitutes a shape cycle, where we start at some given shape and stop at that same shape.

Definition 5.3.1 (37) allows us to refer to a cycle variable as by Definition 4.2.4 (28).

**Definition 5.3.2.** *Given a shape cycle $z = \langle c_1, c_2, \_ \rangle, \langle c_2, c_3, \_ \rangle, \ldots, \langle c_{n-1}, c_n, \_ \rangle$, where $c_1 = c_n$. Let $v_{p_1}$ denote the name of the variable bound in clause $c_1$. We say that the cycle $z$ has a terminal branch if the program takes a path $z' \neq z$ whenever $c_1$ is called with an argument $b \in \mathbb{B}$ s.t. $v_{p_1}$ is bound to 0.*

**Theorem 5.3.1.** *Given a shape-change graph $G$, a program terminates iff all shape cycles monotonically decrease their respective cycle variables and have a terminal branch.*

*Proof.* If a cycle $z = \langle c_1, c_2, \_ \rangle, \langle c_2, c_3, \_ \rangle, \ldots, \langle c_{n-1}, c_n, \_ \rangle$ monotonically decreases the cycle variable, then eventually, the clause $c_1$ will be called with a value $b \in \mathbb{B}$ s.t. $v_{p_1}$ is bound to 0. If $c_1$, hence branches off to a path $z' \neq z$, i.e. if $z$ is a terminal branch, then the cycle itself has terminated. □

## 5.4 The algorithm

We define the shape-change termination algorithm as follows:

**Definition 5.4.1.** *Given a program $r$ and its corresponding shape-change graph $G$, yield "halts" if all the call cycles in $G$ are monotonically decreasing, and "unknown" otherwise.*

**Theorem 5.4.1.** *Shape-change termination is sound and complete as per Definition 1.1.2 (5).*

*Proof.* It is easy to see from the discussion above that the shape-change graph has a finite number of nodes and edges, the method is complete by Theorem 4.2.1 (27). □

# Chapter 6

# Conclusions and possible future work

Size-change termination has been moderately extended to handle call cycles that do not monotonically decrease, but eventually terminate due to primitive boolean conditions eventually becoming fulfilled.

The technique has been dubbed shape-change termination but in the end comes down to transforming the call graph of the program enough to ensure that no such cycles can exist, i.e. all cycles either monotonically increase, monotonically decrease or keep the values of the variables the same. After this is done, the technique basically comes down to good old size-change termination.

What's more, shape-change termination is an extension that was natural to $\Delta$, and size-change termination would've sufficed in a more low-level language with a bit more constraints on the programmer. We see this reassured as assumptions reducing the complexity of the language are employed throughout Chapters 4 & 5.

The down side of both techniques, or rather, the remaining down side of size-change termination, are some of the following:

- Dead code isn't handled very well.
- Some programs might terminate for some variable ranges, while they don't for others, this isn't handled very well either.

In general, one particularly big problem is that as soon as monotonically nondecreasing call cycle is spotted the method yields the result "unknown", regardless of whether that call cycle is reachable or not.

The descriptions of both size-change termination and shape-change termination evolve around the language $\Delta$, which allowed both for shortcuts due to language definition and bias towards language-specific intricacies. However, as $\Delta$ is Turing complete, the techniques should be universally applicable in general.

An actual implementation of either size-change termination or shape-change termination has not been provided although it has been covered in e.g. [Lee et al., 2001]. We can imagine a method for shape-change termination where we explore the possible shapes of the original input values as we examine the branches of the program from its initial expression. That is, given a clause in a shape graph, we know what sort of shape the clause argument must have had if the clause had accepted. Furthermore, all function calls have some expressions, i.e. compositions of e.g. subsets of the clause argument, allowing us to draw out how the original argument should've looked if it matched some clause as a result of the function call as well. In this way, we could eliminate some of the problems posed by unreachable clauses and deduce some variable ranges for which the program does or does not terminate. Most likely, other problems would propagate. This method resembles righteously another known method of automated termination proving, in particular, term rewriting [Arts and Giesl, 2000], [Barendregt et al., 1987], [Thiemann and Giesl, 2003].

# Bibliography


[Arts and Giesl, 2000] Arts, T. and Giesl, J. (2000). Termination of term rewriting using dependency pairs. Theoretical Computer Science *236*, 133 – 178.

[Avery, 2006] Avery, J. (2006). Size-Change Termination and Bound Analysis. In Functional and Logic Programming, (Hagiya, M. and Wadler, P., eds), vol. 3945, of Lecture Notes in Computer Science pp. 192–207. Springer Berlin / Heidelberg.

[Barendregt et al., 1987] Barendregt, H., van Eekelen, M., Glauert, J., Kennaway, J., Plasmeijer, M. and Sleep, M. (1987). Term graph rewriting. In PARLE Parallel Architectures and Languages Europe, (de Bakker, J., Nijman, A. and Treleaven, P., eds), vol. 259, of Lecture Notes in Computer Science pp. 141–158. Springer Berlin / Heidelberg.

[Berdine et al., 2006] Berdine, J., Cook, B., Distefano, D. and O'Hearn, P. (2006). Automatic Termination Proofs for Programs with Shape-Shifting Heaps. In Computer Aided Verification, (Ball, T. and Jones, R., eds), vol. 4144, of Lecture Notes in Computer Science pp. 386–400. Springer Berlin / Heidelberg.

[Jones and Bohr, 2004] Jones, N. and Bohr, N. (2004). Termination Analysis of the Untyped Lambda Calculus. In Rewriting Techniques and Applications, (van Oostrom, V., ed.), vol. 3091, of Lecture Notes in Computer Science pp. 1–23. Springer Berlin / Heidelberg.

[Kildall, 1973] Kildall, G. A. (1973). A unified approach to global program optimization. Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages , 194–206.

[Lee et al., 2001] Lee, C. S., Jones, N. D. and Ben-Amram, A. M. (2001). The size-change principle for program termination. SIGPLAN Not. *36*, 81–92.

[Naur et al., 1963] Naur, P., Backus, J. W., Bauer, F., Green, J., Katz, C., McCarthy, J., Perlis, A. J., Rutishauser, H., Samuelson, K., Vauquois, B., Wegstein, J., van Wijngaarden, A. and Woodger, M. (1963). Revised Report on the Algorithmic Language ALGOL 60. Technical report.

[Plotkin, 2004] Plotkin, G. D. (2004). A Structural Approach to Operational Semantics. Journal of Logic and Algebraic Programming *60-61*, 17–139.

[Reingold and Tilford, 1981] Reingold, E. M. and Tilford, J. S. (1981). Tidier Drawings of Trees. IEEE Trans. Softw. Eng. *7*, 223–228.

[Sereni and Jones, 2005] Sereni, D. and Jones, N. (2005). Termination Analysis of Higher-Order Functional Programs. In Programming Languages and Systems, (Yi, K., ed.), vol. 3780, of Lecture Notes in Computer Science pp. 281–297. Springer Berlin / Heidelberg.

[Thiemann and Giesl, 2003] Thiemann, R. and Giesl, J. (2003). Size-Change Termination for Term Rewriting. In Rewriting Techniques and Applications, (Nieuwenhuis, R., ed.), vol. 2706, of Lecture Notes in Computer Science pp. 264–278. Springer Berlin / Heidelberg.

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

# 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[Naur et al., 1963].

### 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-$\infty$ repetition |
| $+$ | 1-$\infty$ 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 text

The following section describes the syntax used in this text to describe the operational semantics of the language $\Delta$. The syntax is inspired by [Plotkin, 2004], 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 $=$ operator

The notation used in [Plotkin, 2004] 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 $=$ operator. For instance, $x = e$ means that the expression $x$ that we're considering really is just a single element, or $x = e \cdot x'$ 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

[Plotkin, 2004] 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.

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

#### 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 revert to using parentheses we'll define the precedences of the possible operators in the prepositions of rules. Operators with higher precedence are bind tighter:

1. $\vee$
2. $\wedge$
3. $\rightarrow$

[1] See Appendix A.2.2 (44).
[2] See Appendix A.2.3 (45).