# Chapter 1

# Introduction

## 1.1 Motivation

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

**Definition 1.1.1.** *Given the set of all possible programs $P$, find a program $p \in P$, that can for any $p' \in P$, within a finite amount of time return* `halts` *or* `doesn't halt`*, depending on whether $p'$ eventually stops or runs indefinitely, respectively.*

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

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

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

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

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

**Definition 1.1.2.** *Given the set of all possible programs $P$, find a program $p \in P$, that can for any $p' \in P$, within a finite amount of time, either give up and return* `unknown`*, or return* `halts` *or* `doesn't halt`*, depending on whether $p'$ eventually stops or runs indefinitely, respectively. Find a $p$ such that the number of $p' \in P$ for which $p$ returns* `unknown` *is minimized.*

## 1.2 Expectations of the reader

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

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

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

In summary, the following concepts are used without definition:

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

## 1.3 Preliminaries

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

**Definition 1.3.1.** *Let $\mathbb{N}^0$ denote the set of nonnegative integers and let $\mathbb{N}$ denote the set of positive integers.*

**Definition 1.3.2.** *When dealing with lists, aka. finite ordered sequences, we'll adopt the following notation:*

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. *$[term \mid variables \in spaces, precondition]$ 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 1.3.3.** *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 $n \in \mathbb{N}^0$.*

## 1.4 Chapter overview

**Chapter 2**

**Chapter 3**

**Chapter 4**

**Chapter 5**

# Chapter 2

# On the general uncomputability of the halting problem

## 2.1 Computational equivalence

We say that two languages are computationally equivalent if they both can compute the same class of functions. We make use of this concept over regular Turing completeness because there exist computable functions uncomputable by a universal Turing machine, and there is seemingly no proof that all computable functions are computable by a Turing machine.

## 2.2 Computable problems and effective procedures

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

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

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

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

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

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

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

[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.

- effective procedure
- effectively decidable
- effectively enumerable

## 2.3 Enumerability

### 2.3.1 Enumerable sets

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

**Definition 2.3.1.** *An enumerable set is either the empty set or a set who's elements can placed in a sequence s.t. each element gets a consecutive number from the set of natural numbers* $\mathbb{N}$.

### 2.3.2 Decidability

**Definition 2.3.2.** *A problems is decidable if there exists an algorithm that for any input event*

- Recursively enumerable – countable sets
- Co-recursively enumerable

## 2.4 Cantor's diagonalization

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

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

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

Such a sequence can be represented as a table:

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

□

## 2.5 The halting problem

## 2.6 Rice's statement

## 2.7 Primitive recursion

All primitive recursive programs terminate.

## 2.8 Introduction to size-change termination

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

## 2.9 The language to be defined

The soft version.

# Chapter 3

# The language $\Delta$

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 $\Delta$ 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.

$\Delta$ 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, $\Delta$ borrows some syntax and semantics from purely functional languages such as ML or Haskell. Hence, programs in $\Delta$ 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, $\Delta$ 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 $\Delta$. 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 $\Delta$ can be extended and used in a language with types and abstract data structures, as long as it is computationally equivalent to $\Delta$.

Also, unlike most purely functional languages, $\Delta$ is a first-order, call-by-value language. This is done in part to adhere to the general flow of [**?**], 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 $\Delta$ untyped.

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

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

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

**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 (8) 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 (8) . □

### 3.2.1 Size

Definition 3.2.3 (8) made use of a surjective function $f : \mathbb{B} \rightarrow \mathbb{N}^0$. 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 $f : \mathbb{B} \rightarrow \mathbb{N}^0$.*

*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 (8) , 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 (8) and Theorem 3.2.2 (8) . □

**Definition 3.2.7.** *Definition 3.2.6 (8) allows us to wlog overload the binary operators $=$, $<$, $>$, $\geq$ and $\leq$, defined over $\mathbb{N}^0 \times \mathbb{N}^0$, for $\mathbb{B} \times \mathbb{B}$.*

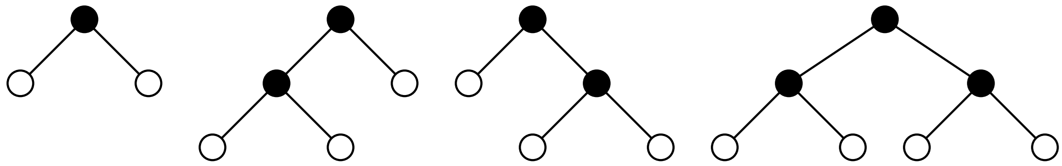


**Figure 3.1:** A few sample value visualizations having the sizes 1, 2, 2, and 3 respectively.

### 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 (9) 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[**?**] 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 (9) and Theorem 3.2.1 (8) . □

As with values, it might prove beneficial to the discussion to visualize the shapes. Figure 3.2 (10) 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.*

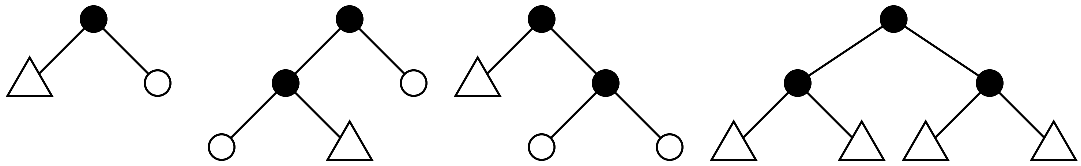


**Figure 3.2:** A few shape visualization examples. The leftmost shape describes values that are nodes with leafs as right children and any trees as left children. The two leftmost values in Figure 3.1 (9) match this shape.

**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 (9) , $s_1$ must have more nodes than $s_2$, and by Definition 3.2.12 (9) , a shape with more nodes cannot match a shape with fewer nodes. □

## 3.3 Syntax

We describe the syntax of Δ 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>.

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

**Definition 3.3.1.** *Table 3.1 (11) 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 (39) .

| 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 (11) .*

**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 (10) and Definition 3.3.8 (11) 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 (12) . 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. $\square$

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[**?**], 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 (40) .

### 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:

$$\frac{\langle \text{SINGLE}, x, \sigma\rangle \to b \vee \langle \text{CHAIN}, x, \sigma\rangle \to b}{\langle x, \sigma\rangle \to b} \tag{3.11}$$

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

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

### 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.

$$\frac{\langle \text{ZERO}, e, \sigma\rangle \vee \langle \text{EXPRESSION}, e, \sigma\rangle \vee \langle \text{VARIABLE}, e, \sigma\rangle \vee \langle \text{APPLICATION}, e, \sigma\rangle}{\langle e, \sigma\rangle \to \langle b, \sigma\rangle} \tag{3.14}$$

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

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

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

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

### 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:

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

## 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 (14), Listing 3.2 (14) and Listing 3.3 (14), respectively. Note, that due to Δ being a call-by-value language, `or/2` is *not* shortcircuited as conventionally is the case.

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

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

```
or 0 0 := 0
or _ _ := 0.0
```

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

```
not 0 := 0.0
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 it's left child and a binary tree in standard representation as it's 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 (14) for an example of a value in Δ and its normalized form.

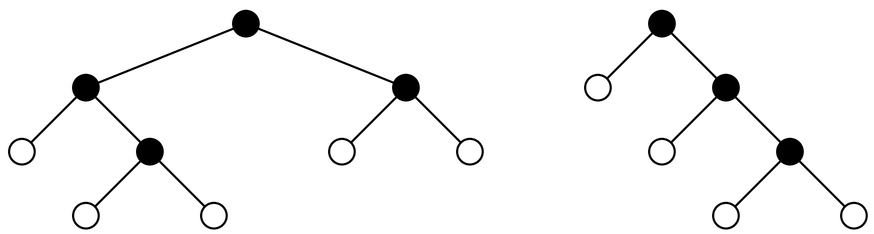


**Figure 3.3:** A sample value $b \in \mathbb{B}$ to the left, and its normalized form to the right.

Definition 3.5.2 (14) allows us to define the function `normalize/1` that normalizes a value. We've done this Listing 3.4 (15).

```
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 (8) . This allows us to define the functions `less/2` and `equal-size/2`[3] which we do in Listing 3.5 (15) and Listing 3.6 (15) , 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 (15) .

```
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 (15) .

```
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 (16), Listing 3.10 (16) and Listing 3.11 (16).

```
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[4].

We can describe any Turing machine in terms of its transition table, which is a list of 4-tuples $\langle \lambda_0, \sigma, \lambda_1, \omega \rangle : \Lambda \times \Sigma \times \Lambda \times \Omega$, where $\Lambda$ is the finite set of states of the machine (some non-existent), $\Sigma$ is the alphabet of the machine, typically $\{0, 1\}$, and $\Omega$ is the action table of the machine. Given 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 (15).*

$\Lambda$ 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.

[4]The notation and type of universal Turing machine used here was inspired by http://plato.stanford.edu/entries/turing-machine/.

**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 (17) 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 1.0         := urm s 0.1.0 m
6  utm m s 1.(0.0).r   := utm-interpret m 1.v.r (find s m)
7  utm m s 1.(0.0.0).r := utm-interpret m 1.v.r (find s m)
8  utm m s 1.v.r       := utm m s (1.v).r
9
10 utm-interpret _ _ 0 := 0
11
12 utm-interpret m 1.(0.0).r (0.0).s.0 := utm m s 1.(0.0).r
13 utm-interpret m 1.(0.0).r (0.0).s.(0.0) := utm m s 1.(0.0.0).r
14
15 utm-interpret m 1.(0.0).r (0.0).s.(0.(0.0)) := utm m s (1.(0.0)).r
16 utm-interpret m 1.(0.0).r (0.0).s.((0.0).0) := utm m s 1.((0.0).r)
17
18 utm-interpret m 1.(0.0.0).r (0.0.0).s.0 := utm m s 1.(0.0).r
19 utm-interpret m 1.(0.0.0).r (0.0.0).s.(0.0) := utm m s 1.(0.0.0).r
20
21 utm-interpret m 1.(0.0.0).r (0.0.0).s.(0.(0.0)) := utm m s (1.(0.0.0)).r
22 utm-interpret m 1.(0.0.0).r (0.0.0).s.((0.0).0) := utm m s 1.((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 (16) .

# 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[**?**].

## 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. Indeed 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, arcs.

**Definition 4.1.1.** *A control flow graph for a given program $r = \langle F, x \rangle$, with the set of clauses*

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

*is a graph with the set of nodes $C$, and the set of directed edges*

$$E = \{\langle c_1, c_2, x\rangle \mid c_1 = \langle v_1, p_1, x_1\rangle \in C \wedge \\ c_2 = \langle v_2, \_, \_\rangle \in C \wedge \\ x \in \mathbb{X} \wedge \langle v_2, x\rangle \Subset x_1\}.$$

Control flow graphs of this sort can more intuitively be referred to as *call graphs*.

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

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

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

We can now draw a control flow graph for the program define in Listing 4.1 (20) as shown in Figure 4.1 (21) .

```
f x y := x.y
g _ := 0
h _ := 0
i x y := (f ((h y).(g x)) (h y))
i input input
```

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

### 4.1.3 Call cycles

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

**Lemma 4.1.1.** *We're concerned with call cycles in control flow graphs since non-termination cannot occur if not for an infinite control flow cycle.*

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

Call cycles in $\Delta$ can occur in recursive or mutually recursive function clauses.

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

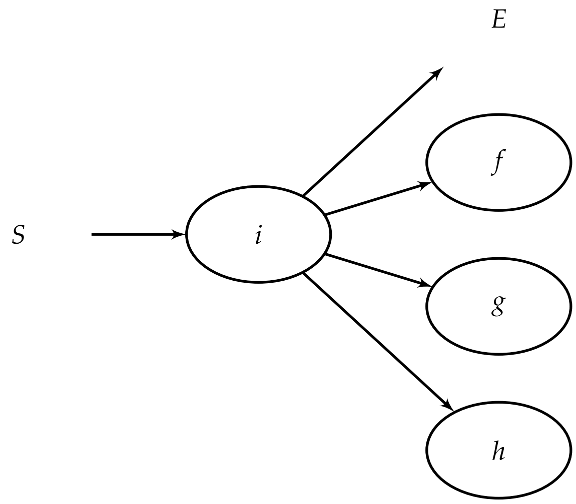



**Figure 4.1:** A control flow graph for the Δ program in Listing 4.1 (20) . The graph does not explicitly specify back-propagation of control, if any.

### 4.1.4 Disregarding back-propagation

It is worth noting that in Figure 4.1 (21) , the clauses that make no function calls have out-degree 0. Technically, these functions *do transfer control* – back to the callee. We may refer to this process as *back-propagation of control*. While considering back-propagation is seemingly important to a concept that bases itself on the changes in the sizes of the program values, we're only concerned with call cycles.

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

### 4.1.5 Dropping the start and end nodes

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

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

### 4.1.6 Control flow graphs vs. abstract static call graphs

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

### 4.1.7 Multiple calls to the same function

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

This allows us to draw a control flow graph, or equivalently, a call graph, for the program in Listing 4.2 (22) . Here, we've already disjunctively labelled all of the calls in the expressions. This call graph is drawn in Figure 4.2 (22) .

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

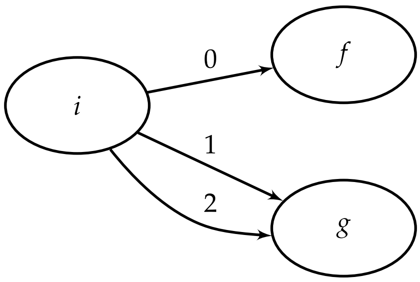



**Figure 4.2:** A control flow graph for the Δ program in Listing 4.2 (22) .

### 4.1.8 Multiple clauses

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

The semantics of Δ tell us to make an unconditional transition to the immediately next clause of the function. There is at most one such transition for any clause, and the last clause of a function declaration cannot fail to pattern match[1].

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

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

Hence, we can now draw the call graph for the program defined in Listing 4.3 (22) as shown in Figure 4.3 (23) .

```
f0: f 0 := 0
f1: f x._ := f x

f input
```

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

For a more complex example, let's consider the call graph for the program `reverse` introduced in § 3.6 (16) . The program is repeated in annotated form in Listing 4.4 (23) , and its corresponding call graph is shown in Figure 4.4 (23) .

[1]See § 3.4 (12) .

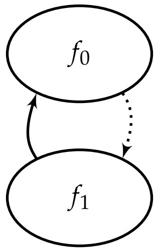



**Figure 4.3:** A control flow graph for the program defined in Listing 4.3 (22) .

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

reverse input
```

**Listing 4.4:** An annotated version of the program `reverse` introduced in § 3.6 (16) .

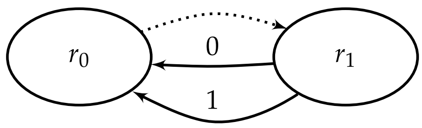



**Figure 4.4:** A control flow graph for the Δ program in Listing 4.4 (23) .

### 4.1.9 Deeply nested function calls

Blah

## 4.2 Size-change termination principle

Consider the program in Listing 4.3 (22) and it's corresponding call graph in Figure 4.3 (23) . Without any further information about the control transitions, the program seemingly loops indefinitely. However, there are some things that we can deduce about the control transitions.

**Lemma 4.2.1.** *If we can deduce for every control flow cycle in a prorgram that it reduces a value of well-founded data-type on each iteration of the cycle, then the value must eventually bottom out and the program must terminate.*

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

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

**Lemma 4.2.2.** *A control flow cycle reduces a value on each iteration if at least one of the participating control transitions reduces the value and all other control transitions do not increase that value.*

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

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

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

**Definition 4.2.1.** *The relation $\Phi : C_{caller} \times C_{callee} \times N_{caller} \times N_{callee} \to \{\perp, <, \leq\}$ is defined to be the size relation between the caller and callee clauses in $\Delta$ where $N_{caller}$ are the names of the variables bound in the caller, and $N_{callee}$ are the names of the variables bound in the callee. Note, that we are only concerned with reductions and non-increases in size, all other relationships are marked by the no relationship symbol $\perp$. Initially, the relationship between all the clauses and their variables is $\perp$.*

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

### 4.2.1 Fail transitions

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

**Lemma 4.2.3.** *Fail transitions are transitive in the sense that the relationship between the variables bound in the source and the variables bound in the target is the same regardless of the number of fail transitions in the path between the source and the target.*

*Proof.* Follows from the semantics of $\Delta$. □

We are not concerned with exact equivalence, hence all fail transitions in the $\Phi$ relation return the relation $\leq$ for all variable pairs.

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

### 4.2.2 Success transitions

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

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

Of course, we cannot deduce a precise size displacement as the values of the bound variables may initially be *unknown* at compile time[2]. However, we can deduce a *safe* displacement estimate, such that it is less than or equal to the actual displacement in terms of absolute value. For instance, if the expression `a.b` appears as a function call argument, where `a` and `b` are some bound variables with unknown values, and this argument evaluates to some value $v$, then we can *safely* say that $v > \texttt{a}$, $v > \texttt{b}$, $v \geq \texttt{a.0}$ and $v \geq \texttt{0.b}$.

We decide to ignore the nested function calls because this would imply a more complex static analysis of the program. Specifically, we're unable to say anything about the result of the nested function call from the scope of the source clause alone. Instead, we treat results from nested function calls simply as variables with *unknown* values. We also make sure to keep these variables separate from the bound variables as there is no relationship to draw between these "variables" and the variables bound in the target[3].

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

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

More formally, given a function argument as the expression $x$, we construct the expression $x^s$ where we replace all first-level nested function calls[4] by auxiliary variables. We group all those auxiliary variables into the set of variable names $N^s_{calls}$ and all the remaining variables into the set $N^s_{vars}$. Furthermore we construct the auxiliary variable names in such way that $N^s_{vars} \cup N^s_{calls} = \emptyset$. Hence, we obtain the tuple $(x^c, N^s_{vars}, N^s_{calls})$.

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

On the other hand, if we had to formally define the relation that had to be built between the variables bound in the source and the values that the function call arguments evaluated to, this would be a relation between values and some kind of "abstract patterns", as e.g. $v \geq \texttt{0.b}$.

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

Indeed, the tuple $(x^s, N^s_{vars}, N^s_{calls})$ constitutes such an abstract pattern already, since the expression $x^s$, contains no function calls and hence syntactically matches a pattern in $\Delta$[5]. We henceforth refer to such an expression as $p^s$[6]. Given a clause with the pattern $p^t$[7], we can easily deduce the set $N^t_{vars}$, which is the set of variable names used in $p$. Our task is then to deduce a size relation between the variables in the sets $N^s_{vars}$ and $N^t_{vars}$ given the tuples $(p^s, N^s_{vars}, N^s_{calls})$ and $(p^t, N^t_{vars})$.

### 4.2.3 Pattern matching

Let the function $\phi : \mathbb{N} \times \mathbb{N} \to \{<, \leq, \perp\}$ denote the function $\lambda N^t, N^s.\Phi\left(C^t, C^s, N^t, N^s\right)$. In the following section we will discuss the rules involved in deducing the function $\phi$, that is, the function $\Phi$ for some given source and target of a success transition.

For this purpose we will regard the tuples $(P^s, N^s_{vars})$ and $(P^t, N^t_{vars})$, of a given success transition, where $P^s$ is the list of abstract patterns derived from the function arguments in the source, and $P^t$ is the list of corresponding actual patterns in the target. Furthermore, let $N^s_{vars}$ and $N^t_{vars}$ be unary functions of the type $\mathbb{P} \to \mathbb{N}^*$, accepting a pattern and yielding the variable names that are contained both in the input pattern and the sets $N^s_{vars}$ and $N^t_{vars}$, respectively.

In the following analysis we will look at but one instance of the lists $P^s$ and $P^t$, namely the abstract pattern $p^s$ from the source and its corresponding actual pattern in the declaration, $p^t$. In total, however, this process has to be repeated for each such pair given the sets $P^s$ and $P^t$, iteratively extending the definition of the relation $\phi$ to all variables bound in the sets $N^s_{vars}$ and $N^t_{vars}$.

We initially define $\phi$ to yield the value $\perp$ for all arguments. We will continuously modify this definition as we process $p^s$ and $p^t$. We denote this within the semantics in a manner similar to the state $\sigma$ in the semantics[8]. However, $\phi$ is now a binary "memory", requiring both a target name and a source name (in that order). For simplicity, we will borrow some suger coding from the matlab notation which allows us to provide a collection in place of a single element and let the runtime apply the given function to each element in the collection. For instance, we might write that $\phi\left(N^t_{vars}(p^t), n^s\right) \mapsto <$, meaning that all the target variables used in $p^t$ are strictly less than the source variable $n^s$.

We now define a summoning rule, dividing the rules up into sub-rules:

[4]Nested function calls of nested function calls are hence considered irrelevant to the derivation of the size relation of the top-level call, however, they may become relevant as we derive the size relations of the corresponding nested calls.

[5]See § 3.3 (10) if you're uncertain.

[6]Where $s$ stands for *source*.

[7]Where $t$ stands for *target*.

[8]See § 3.4 (12) .

$$\frac{\langle \mathrm{A}, p^t, p^s, \phi \rangle \to \phi' \lor \langle \mathrm{B}, p^t, p^s, \phi \rangle \to \phi' \lor \langle \mathrm{C}, p^t, p^s, \phi \rangle \to \phi' \lor \langle \mathrm{D}, p^t, p^s, \phi \rangle \to \phi' \lor \langle \mathrm{E}, p^t, p^s, \phi \rangle \to \phi'}{\langle p^t, p^s, \phi \rangle \to \phi'} \tag{4.1}$$

One of the simpler cases is when the abstract pattern $p^s$ is simply 0, or some name $n^s$, and $n^s \in N^s_{calls}$. Since no variables bound in the source participate in $p^s$, then no relations need to be drawn to any of the target variables that might appear in the corresponding $p^t$. Hence, $\phi$ need not be modified.

$$\frac{(p^s \to 0 \lor (p^s \to n^s \land n^s \notin N^s_{vars})) \land \phi \to \phi'}{\langle \mathrm{A}, p^t, p^s, \phi \rangle \to \phi'} \tag{4.2}$$

This has a symmetrical case. Indeed when $p^t$ is neither a destruction, nor any name $n^t$, that is, it is _ or 0. This pattern contains no variables, and hence no relations need to be drawn from any of the variables that might appear in the corresponding $p^s$. Hence, $\phi$ need not be modified in such a case either.

$$\frac{\left(p^t \to 0 \lor p^t \to \_\right) \land \phi \to \phi'}{\langle \mathrm{B}, p^t, p^s, \phi \rangle \to \phi'} \tag{4.3}$$

If $p^t$ is the name pattern $n^t$, the matters get a bit more complicated:

1. If $p^s$ is some node, then all the variables that occur in $p^s$, i.e. $N^s_{vars}(p^s)$, will all be strictly less than $n^t$ by the semantics of $\Delta$. However, we are not concerned with this relation, as we would like to know when a value is decreased from source to target, and not, as in this case, increased.

2. If $p^s$ is also some name pattern $n^s$, and $n^s \in N^s_{vars}$, 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, $\leq$:

$$\frac{p^t \to n^t \land p^s \to n^s \land n^s \in N^s_{vars} \land \langle \phi \left(n^t, n^s\right) \mapsto \leq \rangle \to \phi'}{\langle \mathrm{C}, p^t, p^s, \phi \rangle \to \phi'} \tag{4.4}$$

If $p^t$ is a destruction and $p^s$ is the variable name $n^s$, then we can safely say that all the variables that occur in $p^t$, i.e. $N^t_{vars}(p^t)$, are all strictly less than the variable in $n^s$:

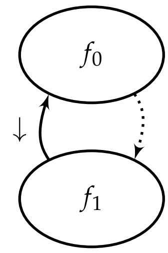
$$\frac{p^t \to p^t_1 \cdot p^t_2 \land p^s \to n^s \land n^s \in N^s_{vars} \land \langle \phi \left(N^t_{vars}(p^t), n^s\right) \mapsto < \rangle \to \phi'}{\langle \mathrm{D}, p^t, p^s, \phi \rangle \to \phi'} \tag{4.5}$$

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

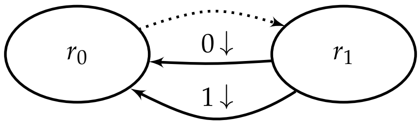
$$\frac{p^t \to p^t_1 \cdot p^t_2 \land p^s \to p^s_1 \cdot p^s_2 \land \langle p^t_1, p^s_1, \phi \rangle \to \phi'' \land \langle p^t_2, p^s_2, \phi'' \rangle \to \phi'}{\langle \mathrm{E}, p^t, p^s, \phi \rangle \to \phi'} \tag{4.6}$$

## 4.3 Graph annotation

Hence, we can deduce from Listing 4.3 (22) , that when $f_1$ makes a call to $f_0$ it does so with a value strictly less then it's own argument, i.e. the transition $f_1 \to f_0$ strictly decreases a value. Visually we will mark this with a $\downarrow$. The Lemmas **??** (**??**) and **??** (**??**) can be used to deduce the same sort of relationship for the transitions $r_1 \xrightarrow{0,1} r_0$ for Listing 4.4 (23) . These observations are summarised in Figure 4.5 (27) .




(a) The program in Listing 4.3 (22) .




(b) The program in Listing 4.4 (23) .

**Figure 4.5:** Call graphs with annotated edges for various programs.

### 4.3.1 Calls to multivariate functions

The call graph notation used thus far has only been used for describing calls to unary functions. As an example of a multivariate function, we may consider the function `normalized-less/2`, introduced in § **??** (**??**) . We use this function to define the program in Listing 4.5 (27) . The corresponding call graph is shown in Figure 4.6 (27) .

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

**Listing 4.5:** A sample program with a multivariate function.

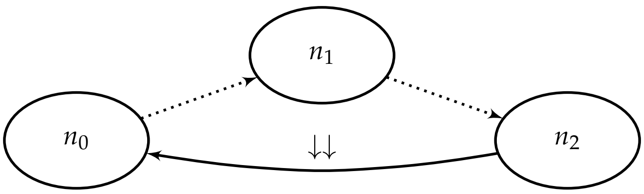



**Figure 4.6:** A control flow graph for the program defined in Listing 4.5 (27) .

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

### 4.3.2 Nonincreasing transitions

There are cases where for a given transition in a call cycle, we can't tell whether the sizes are strictly decreased or remain the same, but we can definitely say that there is *no increase* in the sizes of variables. As an example, consider the program in Listing 4.6 (27) .

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

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

For the recursive clause $g_1$, it is unclear whether the sizes of the variables are decreased in the transition $g_1 \rightarrow g_0$, or not. Specifically, if the arguments to `g` are of the form `0._` and `_.0`, respectively, the size is *not* decreased by the call. We'll denote such transitions by the symbol $\Downarrow$. We can now draw the call graph for the program in Listing 4.6 (27) as in Figure 4.7 (28) .

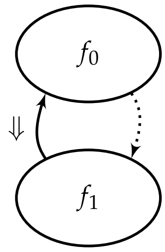



**Figure 4.7:** An annotated call graph for the program in Listing 4.6 (27) .

### 4.3.3 Increasing transitions

```
f0: f a b = f a.b b.a
f input input
```

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

# Chapter 5

# Shape-Change Termination

One trouble with size-change termination as described in the previous chapter is Lemma 4.2.2 (23) . 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, the calls of any call cycle are constrained to nonincreasing calls. 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 $\Delta$, it has some particular shape.

Consider the program in Listing 5.1 (29) 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.

```
f0: f a.b.c.d := a
f1: f a := f a.0
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.

## 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 (29) , 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 (16) . 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 (29) . The function `f` has only one terminal clause, the one that accepts a shape as in Figure 5.1 (30) . If the function argument has any other shape, i.e. either a shape as in Figure 5.2 (30) , Figure 5.3 (30) or Figure 5.4 (30) , 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 (30) , which in turn transforms the value into one that matches Figure 5.4 (30) , which in turn transforms the value into one that matches Figure 5.1 (30) , i.e. the terminal clause. What's more, there are infinitely many other values that will match the shape Figure 5.3 (30) , and for each of them, the clause $f_1$ will transform them into values that match Figure 5.4 (30) , which will transform them into values that match Figure 5.1 (30) .

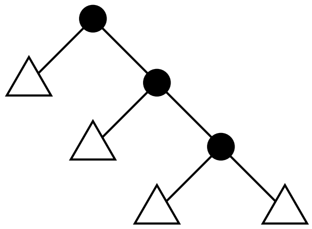


**Figure 5.1:** The shape that the clause $f_0$ in Listing 5.1 (29) will accept.

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

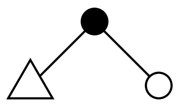


**Figure 5.3:** The pattern `a.0`.

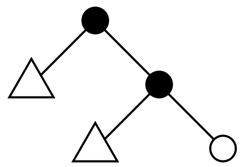


**Figure 5.4:** The pattern `a.b.0`.

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 Prerequisites

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 (30) . 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. □

```
f0: f 0 := 0
f1: f (a.b).c := 0
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 (31) 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 $b'' > 0$.

```
g0: g 0 := 0
g1: g (a.b).(c.d) := 0
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, $s_1 \in \mathbb{S}$ and $s_2 \in \mathbb{S}$, 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 $B_1 \cap B_2 = \emptyset$.*

**Theorem 5.2.1.** *Given a shape $s' \in \mathbb{S}$, the set $\{s \mid s \in \mathbb{S} \wedge s \cap s' = \emptyset\}$, that is also pairwise disjoint, 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, 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 pairwise disjoint.

2. For any given shape $s' \in \mathbb{S}$ the number of nodes and leafs is finite by Definition **??** (**??**) .

□

## 5.3 Shape-change termination

**Definition 5.3.1.** *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 5.3.1.** *By Definition **??** (**??**) , 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 5.3.2.** *Given any clause $\langle v, p, x\rangle \in \mathbb{C}$ and a value $b \in \mathbb{B}$, if $b \succ p$, then $S = \{s\}$ is a plausible set of shapes for $b$, where $s$ is the shape corresponding to pattern $p$.*

**Corollary 5.3.3.** *Consider any pair of clauses $c_1 = \langle v_1, p_1, x_1\rangle, c_2 = \langle v_2, p_2, x_2\rangle \in \mathbb{C}$, with the plausible value sets $S_1 = \{b \mid b \in \mathbb{B} \wedge b \succ p_1\}$ and $S_2 = \{b \mid b \in \mathbb{B} \wedge b \succ p_2\}$. If for some $b \in \mathbb{B}$, $c_1$ is considered before $c_2$, and $c_1$ fails to accept $b$, then the set of plausible shapes for $b$ can be said to be $S_1$*

By Definition 3.3.13 (12) , all clauses are unary, and hence a clause can be said to specify a shape. By Definition 3.3.5 (10) , a function call contains a finite list of clauses.

**Theorem 5.3.4.** *Given a function $f = \langle v, C\rangle$,*

*Proof.* The shapes specified by a function are derived from the patterns of its clauses. □

## 5.4 Nice

Every function can be defined in terms of clauses with disjoint shapes (perhaps duplicating some clauses).

Let the set $S_{f_{init}}$ denote the initial (finite) set of shapes that the function $f \in \mathbb{F}$ considers. We say that a function considers a set of shapes $S \subset \mathbb{S}$, if the clauses of the function (now disjoint) use exactly the patterns that would constitute $S$.

We define a program $r = \langle F, x \rangle$ as a network of nodes with the node set $\left\{ s \mid f \in F \wedge s \in S_{f_{init}} \right\}$.

We define an explorative algorithm that at any given stage is in one of the nodes of the program. At any given

Let $s \in \mathbb{S}$ be some current variable shape. For any call from $s_f \in S_{f_{init}}$ to some function $g \in \mathbb{F}$, draw edges from $s_f$ to every shape in $\left\{ s_g \mid s_g \in S_{g_{init}} \wedge s \succ s_g \right\}$. Extend the present variable shape if applicable. If we're in shape $s_f \in S_{f_{init}}$ and follow a transition to shape $s_g \in S_{g_{init}}$, we must check back with the path taken to get to this $s'$, whether it has previously already been in the $s_g$ shape. If so, the question merely is whether

## 5.5 Rest

If there is a finite number of shapes for any triangle, there is a finite number of shapes for any shape.

Instead of constructing a call graph with every clause in a program as a node, we construct a call graph with every shape of every function call as a node. This builds upon the idea that there is a finite number of disjoint shapes deducible from the clauses of any function definition, which renders the number of nodes in the graph finite. The number of shapes for any one clause may vary, and hence there will be multiple edges from caller to its callees, rather than one edge to the first clause of the callee as seen with regular size-change termination.

The nodes are iteratively derived starting with the initial clause $c_{main}$, drawing the edges and callee shapes as function calls are encountered, i.e. we use an exploration technique to build the graph.

### 5.5.1 Annotation

In the sections that follow we'll make use of an extended call graph syntax. The intent of the call graphs in this chapter is to show how a shape changes in a control transition from source to target. Indeed, the call graphs are no longer call graphs but *shape change graphs*.

#### Success & fail transitions

There no longer needs to be drawn a distinction between the two since a value of some shape will match exactly one choice. In some cases, where a deduction of a value is fairly evolved this clause may be deduced, in other cases, the success & fail transitions will be regarded as one and the same.

### 5.5.2 Patterns and shapes

For any unknown value, we may assume only one shape, the tree. Hence, given a program like the one in Listing 5.1 (29) , we start analyzing the function `f` by assuming nothing about the input argument, i.e. annotating it with a $\triangle$. The value may match either clause, but it will match exactly one. If the value matches a clause, that indicates that the value has a certain shape, indeed this is what a pattern declaration is – a shape specification, or a requirement, if you wish. The position of the clause enclosing the pattern wrt. to other clauses can be used to reduce the singleton set of possible values to a set containing multiple, more concrete shapes. All in all, multiple clauses may be chosen, and as already discussed for clause $f_1$, multiple shapes can be deduced for a given clause, we consider *all* possible shapes. Figure 5.5 (33) illustrates this initial step.

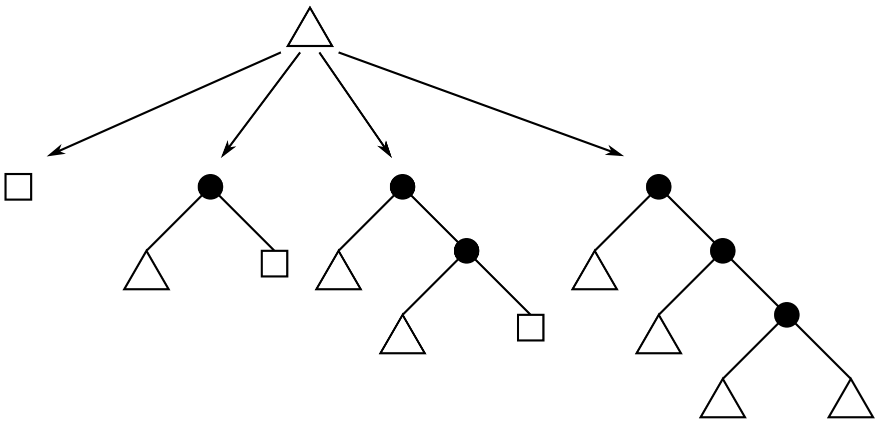


**Figure 5.5:** Initializing shape analysis for the program in Listing 5.1 (29) .

**Deducing shapes from patterns**

Shape deduction is an iterative, and initially non-terminating process for any non-terminating program. We begin by introducing the rules that allow us to deduce shapes from patterns and call sequences, and make the method terminating at a later stage.

The method itself builds upon the idea that the sort of iterative pattern matching that goes on when a function call is made, can be used to deduce information about the shape of the value. Here, both failure to match a series of patterns as well as the success to match some pattern eventually are useful. In particular, the series of failed pattern matches indicate which shapes the value *does not* have, which in turn allows us to assemble a finite set of shapes, one of which *must* match the value[1]. We use this information as we look at the subsequent patterns where patterns themselves might not be descriptive enough. For instance, consider how failure to match a node allowed us to deduce a leaf, although the subsequent pattern itself indicated simply a tree shape in § 5.2.1 (30) .

**Lemma 5.5.1.** *If a value $v$ matches the pattern $p$ of the first clause $c$ of a function declaration $f$, then only one possible shape of $v$ is known, indeed it is the one represented by $p$.*

*Proof.* The first clause of a function declaration is the first clause considered by $\Delta$'s runtime when a function call is made. No *other* clauses, and hence patterns, have been attempted at this point. There is therefore no information about the shape of the value other than $p$ itself. Furthermore, if $v$ matches, then it certainly has the shape indicated by $p$, otherwise, $v$ wouldn't have matched $p$. □

The following are a few definitions and lemma are useful when talking about shapes.

**Definition 5.5.1.** *Given two sets of shapes, $X$ and $Y$, we let the operation $X \Cup Y$ denote an operation where the two sets are joined into one, and the pairs of shapes that are not pairwise disjoint in the final set are joined with each other such that if $\exists\, s_1, s_2 \in X \Cup Y : s_1 \curlyvee s_2$, then $s_1$ is removed from $X \Cup Y$.*

**Lemma 5.5.2.** *Given a shape, there is a finite number of disjoint shapes that are disjoint with that shape.*

*Proof.* A shape in $\Delta$ is recursively defined in terms of an unlabeled binary tree, where neither nodes nor leafs have labels, and it is the nodes and leafs themselves that constitute a "shape" together with trees, that match either trees, nodes or leafs.

Given a shape $A$, there is a corresponding shape $B$ for every leaf, and every node in $A$, such that $A$ and $B$ are disjoint. In particular, for every leaf in $A$, $B$ can be given a node with two trees as children, and for every node in $A$, $B$ can be given a leaf. In either case, $A$ and $B$ end up being disjoint. What's more, every $B$ is concerned with a different leaf or node, and no leaf or node in $A$ is ever replaced by a tree, this indicates that all the shapes $B$ are disjoint wrt. one another as well. The parts in shape $A$ that

[1]If a pattern matches a value then the value matches the pattern, and vice versa.

remain untouched are the trees and there are no converse constructs to trees as they match both trees, nodes and leafs.

Since in any given shape there is a finite number of nodes and leafs, any shape has a finite number of disjoint shapes that are disjoint with that shape. □

The following lemma indicates that for any function call there will finitely many shape branches.

**Lemma 5.5.3.** *Every pattern in a function definition, if matched, indicates that the value has one of a finite number of disjoint shapes.*

*Proof.* Any pattern is a shape specification, hence, for single-clause functions this is true due to Lemma 5.5.1 (33) .

Given a multiple-clause function with two consecutive clauses $c_1$ and $c_2$ with patterns $p_1$ and $p_2$, the presence of $p_1$ reduces the space of values which can reach $p_2$. Let $\overline{X}$ denote the pairwise disjoint set of shapes that are disjoint with the shape defined by $p_1$. If $p_1$ fails to match, then the incoming value must have one of the shapes in $\overline{X}$, and by the definition of $\Delta$, any such shape must be accepted by exactly one of the consecutive clauses. Let $Y$ denote the set of shapes defined by $p_2$. Then, if $p_2$ matches followed by $p_1$ failing to match, the shape of the incoming value will be in the set $\overline{X} \Cup Y$.

Since $\overline{X}$ and $Y$ are finite sets by Lemma 5.5.2 (33) , the $\overline{X} \Cup Y$ must be finite as well. □

## 5.6 A description

### 5.6.1 Relationship between value shape and bound variables.

### 5.6.2 Shape transitions

Actual shape transitions occur within the body of a clause.

#### Nested function calls

Unlike regular size-change termination we won't disregard the outcome of nested function calls, but instead consider the shape shifting that we can deduce from such calls. Rather intuitively, an expression evaluation terminates if all of the nested function calls terminate. Hence, for any expression it would require to start with the "deepest most" nested call and work our way up the call stack from there.

#### Between calls

Consider a unary clause $c$ with a pattern specification $p$ and expression $x$. Assume that the pattern $p$ matches some unknown value $v$ and the expression $x$ is hence evaluated with some variables $N$ bound to pairwise disjoint values, which are strict subsets of $v$, or $v$ itself. Due to Lemma 5.5.3 (34) , when $x$ is evaluated there is a finite set of value shapes that we've deduced for $v$ due to the fact that it matched $p$. The variables in the set $N$, without further details, can only be assumed to all be trees.

When a nested function call is encountered in $x$ we compute a *safe* shape approximation of all the arguments to the function before considering what shape shift the actual call might bring about, not least because that depends on the approximation of the shapes of the arguments.

Any function call argument in $\Delta$ is an expression. An expression is a nested construction of either concrete values, bound variables or nested function calls. Given concrete values, and what we've thus far said about the variables bound in $c$, we can deduce the most concrete shape specification without knowledge of what the initially unknown $v$ is.

However, nested function calls, again complicate the matters, indeed because that implies that the shape of the nested function call has to be determined *before* the shape of the parent function call can be determined. We could've ignored nested function calls and merely safely assumed the shape of the value returned by the call was a tree, but that would've been rather useless to the derivation of the shape returned by clause $c$. Instead, we follow this stack of nested function calls in an expression (without actually following the calls), and eventually a base function call that has no function calls in its arguments. Assume we've reached that nested function call and wish to determine the halting property for that particular call.

With the finite set of shapes for the arguments we consider the disjoint shapes of the call destination, and hence determine which clauses may be taken. Note, of course, that this might be *all* the clauses of the called function.

We repeat this process, branching out in all the possible shapes and clauses taken. We certainly terminate whenever a value is passed to a terminating clause. The question hence is what do we do when we reach a loop.

A loop in this case is a shape shifting loop, where a value of a certain shape input into a certain function leads to a call cycle that *seemingly* retains the shape of the value. At runtime, if this call cycle is taken, it is certain that the change that has occurred in the value, if any, has happened in one of the triangles of the shape specification, since otherwise the shapes wouldn't have matched and the cycle wouldn't have occurred.

Such cases is where original size change termination comes in handy. If despite retaining the shape, value is actually decreased in every iteration of the cycle, the triangle eventually reduces to `0`.

A function call can at most possibly shift the shape to any of the possible shapes in a given function.

## 5.7 Termination & Soundness

**Lemma 5.7.1.** *Any function declaration accepts a finite number of shapes for every parameter.*

*Proof.* Any function in $\Delta$ consists of a finite number of clauses and by Lemma 5.5.3 (34) , every clause accepts a finite number of shapes for every parameter. □

**Lemma 5.7.2.** *Any function accepts any valid $\Delta$ value as any parameter.*

*Proof.* Follows from the semantics of $\Delta$. □

**Definition 5.7.1.** *A shape transition source and target are different from a function call source and target in that a function function may elicit one of several shape transitions.*

**Lemma 5.7.3.** *Any possible function call in a program has at least one and at most all of the shapes of a target function as its shape transition targets.*

*Proof.* By Lemma 5.7.2 (35) any value is matched by at least one of the shapes of a given function. Any value matches the triangle shape, and by Theorem **??** (**??**) , the patterns for a particular parameter in summation can match any value, i.e. in total correspond to the triangle shape. □

**Theorem 5.7.4.** *A shape shift network can be constructed for any valid program in finite time.*

*Proof.* Any given program has a finite number of functions. By Lemma 5.7.1 (35) the program may hence initially be considered as finite forest of shapes. Furthermore, by Lemma 5.7.3 (35) any pair of shapes can be directly connected by at most one directed edge. Hence, the said forest has a finite number of edges. The algorithm is sound if all the edges possibly taken by an executing program are in place when it terminates. Hence, if we terminate the branches of the algorithm whenever they reach a terminal clause or enclose a loop, eventually the algorithm must terminate. □

When a function call is made, a list of expressions, each consisting of nested constructions of concrete values, bound variables and nested function calls, is evaluated. For each such call, for each expression in the argument list, we would like a relationship to be drawn between the target input arguments (for which we already have some shape information)

If there is a cycle where we start from one of the disjoint shapes and come back to that shape, then the change must've occurred in one of the triangles. If the value was not decreased (size change termination), then the value must've been increased, or remained unchanged, and this is an infinite loop.

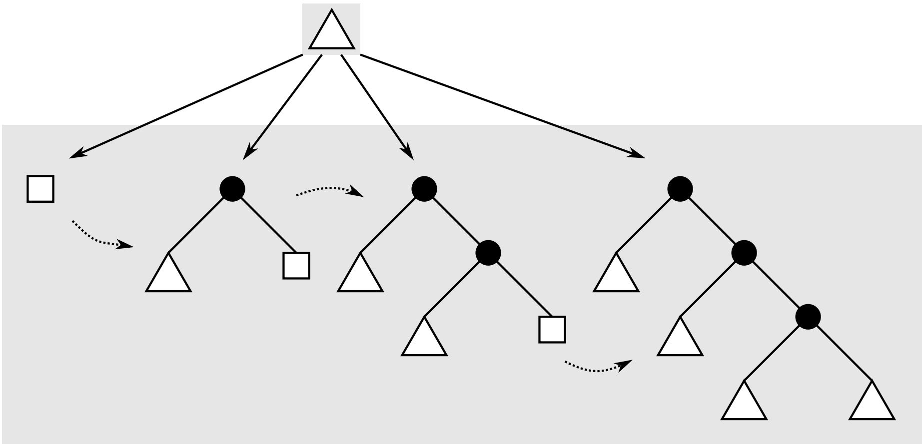


**Figure 5.6:** Nice

*Proof.* Otherwise, there wouldn't have been a loop. This is because the shapes are disjoint, so a change in a triangle can by no means make the value match another shape, otherwise the shapes indeed are *not* disjoint. □

## 5.8 Notes

Size change termination is indeed an abstraction of what we would like to do. Given that $p_0 \curlyvee p_1$, we know that if any

A program can be rewritten into a single function where an extra parameter is added to each function, to distinguish the functions and each function gets a number of auxiliary parameters which are ignored in general. This way, a program of various functions can be transformed into a program consisting of a single function with many clauses, subsets of which represent the individual functions of the original program.

The problem is thus reduced to checking the halting property for some single arbitrary function.

Any function has a number of clauses, at least one. Any terminating program has at least one terminal clause. The point of checking the halting property is hence to deduce that any possible recursive clause that is taken, alters the shape of its arguments in such a way that the call cycle eventually reduces the value towards one, or several of the base cases.

The benefit of this method over original size change termination is that it allows for some control transitions to increase values, as long as the overall cycle size and shape mutation goes towards a some base case.

As much information about the shape of the (changing) value has to be withkept across calls, unlike original size-change termination that allows to discard shape information from the previous clause.

Next, we make the observation that several functions may participate in a call cycle. and in particular, a subset of the recursive clauses may participate in a call cycle. The call cycle describes a terminating loop if every control transition is nonincreasing and at least one is increasing, or the loop shape shifts one of the values towards a base of one of the participating functions.

### 5.8.1 Recursive and terminal clauses

**Lemma 5.8.1.** *A program terminates if all the functions terminate.*

*Proof.* Assume for the sake of contradiction that this is does not hold. That would imply that one of Δ's primitives does not terminate, which is absurd. □

**Lemma 5.8.2.** *A function terminates if all the recursive clauses of the function definition participate in call cycles that shape shift the input value towards a terminal clause of the function definition after each iteration.*

*Proof.* □

If the value is decreased, the shape distortion need not be withkept since the shape information that is deducible from here is by no means useful for call cycle analysis, only size decrease is.

```
c_0: count x 0 := x
c_1: count x y.z := count (count 0.x y) z
```

```
f ((0.a).(0.b)).(0.(0.c)) :=
f a := f a.0
```

```
equal x y := n-equal (normalize x) (normalize y)
n-equal 0 _._ := 0
n-equal _._ := 0
n-equal 0 0 = 0.0
n-equal a.b c.d = and (n-equal a c) (n-equal b d)
```

Let $A$ denote the set of values that $p_2$ can match without regard to $p_1$, let $B$ denote the set of values that $p_1$ can match, and let $C$ denote the set of values that $p_2$ can come to match if $p_1$ failed. Since $p_1 \curlyvee p_2$, then we know that $B \subset A$, $C \subset A$, $B \cap C = \emptyset$ and $C = A - B$.

We say that a value "has" a shape and "it is of" a shape.

# Appendix A

# Notation

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

## A.1 Extended-BNF

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

### A.1.1 What's in common with the original BNF

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

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

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

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

### A.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 work

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

### A.2.1 Some general properties

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

### A.2.2 Atoms

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

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

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

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

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

### A.2.3 The proposition operators

**The $\Rightarrow$ operator**

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

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

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

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

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

**The $\rightarrow$ operator**

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

**The $\wedge$ operator**

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

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

**The $\vee$ operator**

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

**Operator precedence**

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

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

[1]See Appendix A.2.2 (40) .
[2]See Appendix A.2.3 (41) .

3. $\rightarrow$