Chapter 1

Introduction

1.1 Motivation

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

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

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

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

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

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

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

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

1.2 Expectations of the reader

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

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

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

In summary, the following concepts are used without definition:

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

1.3 Chapter overview

Chapter 2

Chapter 3

Chapter 4

Chapter 5

Chapter 2

On the general uncomputability of the halting problem

2.1 Computable problems and effective procedures

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

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

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

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

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

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

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

- effective procedure
- effectively decidable
- effectively enumerable

[1] Invalid inputs are, in this instance, irrelevant.

[2] A finite sequence of discrete values can be trivially encoded as a single discrete value.

[3] All other steps can be omitted without loss of generality.

2.2 Enumerability

2.2.1 Enumerable sets

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

Definition 3. *An enumerable set is either the empty set or a set who's elements can placed in a sequence s.t. each element gets a consecutive number from the set of natural numbers* $\mathbb{N}$.

2.2.2 Decidability

Definition 4. *A problems is decidable if there exists an algorithm that for any input event*

- Recursively enumerable – countable sets
- Co-recursively enumerable

2.3 Cantor's diagonalization

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

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

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

Such a sequence can be represented as a table:

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

□

2.4 The halting problem

2.5 Rice's statement

2.6 Primitive recursion

All primitive recursive programs terminate.

2.7 Introduction to size-change termination

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

2.8 The language to be defined

The soft version.

Chapter 3

The language Δ

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

The intent of the language is hence two-fold, (1) aid the descriptions of the automated termination analysis techniques in further chapters and (2) be relatively modern in the subjective sense of expressiveness.

Expressiveness of a language depends on its initially intended domains. 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 for solving other problems, hence the domain-specific and subjective term of expressiveness.

Δ is a language that disregards the aspect of abstract data structures. Hence, many data driven programs *will* be hard to write in Δ due to, for instance, the complete absence of types. This is of course, relative. Various aspects of the data flow of a program can be important to various automated termination analysis techniques, in particular size-change termination. Hence, Δ is not completely useless and does have data and even simple ways of analyzing the sizes and shapes of its values and branching depending on the outcome of such analyses.

3.1 General properties

One of the fundamental concepts required of the language of application is that it's datatypes are well-founded. That is, any subset S of the range of values of some well-defined type has a value s s.t. $\forall s' \in S\ s \leq s'$. This makes it ideal to chose some oversimplistic data type structure rather than an army of basic types. Besides, an apropriately defined basic data type should be able to represent arbitrarily complex data values.

The language is initially first-order since the size-change termination principle is first described for first-order programs later on in this work. However, the language is designed so that it is easy to turn it into a high-level language without much effort. This may prove necessary as we try to expand size-change termination to higher-order programs.

The language is a call-by-value and purely functional to avoid any problems that could arise from regarding lazy programs or where the notion of a global state of the machine is relevant. Simply put, this is done to ensure elegance of further proof with the help of the language.

The language Δ is also Turing complete in the sense that it can model the Turing machine.

3.2 Data

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

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

Definition 5. *A binary tree is a set that is either empty, henceforth referred to as a leaf or simply* 0*, or contains a single unlabeled node with two binary trees as it's left and right child, henceforth simply referred to as a node. We'll refer to the set of all possible values in* Δ *as* $\mathbb{B}$.

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

Definition 6. *The function* $\cdot : \mathbb{B} \times \mathbb{B} \rightarrow \mathbb{B}$ *constructs a node with the two arguments as it's left and right child, respectively. We'll refer to this function, as well as the operator* $\cdot$ *in general, as "cons".*

Sometimes we'll refer to the *shape* of a data value. A shape specification starts at the root of a value, and specifies a few some immediate nodes or leafs, leaving some sub-values unspecified. This will often be expressed either in graphical binary tree notation, or using the syntax of Δ for constructing binary trees. In either case there will be sub-trees who's actual structure is irrelevant. When using Δ's notation we'll make use of auxiliary variables for such sub-trees, and when using graphical binary tree notation, we'll make use of the conventional triangle. As an example, consider Figure 3.1 (6) .

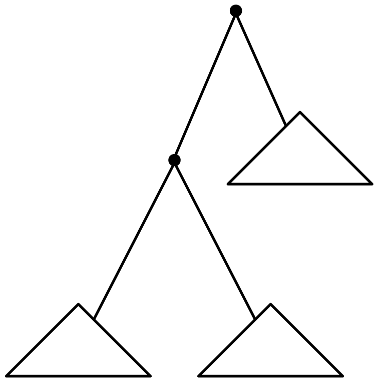


Figure 3.1: Representation of a value consisting of a node with a node as its left child. The triangles represent sub-trees who's actual structure is irrelevant to the shape specification.

3.3 Syntax

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

[1] See § 3.4 (7) .

[2] The extension lends some constructs from regular expressions to achieve a more concise dialect. The extension is described in detail in Appendix A.1 (31) .

$$\texttt{<program>} ::= \texttt{<declaration>}^{*}\ \texttt{<expression>} \tag{3.1}$$

$$\texttt{<expression>} ::= \texttt{<element>}\ (\ \text{‘}\texttt{.}\text{’}\ \texttt{<expression>}\ )\ ? \tag{3.2}$$

$$\texttt{<element>} ::= \text{‘}\texttt{0}\text{’} \mid \text{‘}\texttt{(}\text{’}\ \texttt{<element>}\ \text{‘}\texttt{)}\text{’} \mid \texttt{<name>} \mid \texttt{<application>} \tag{3.3}$$

$$\texttt{<application>} ::= \texttt{<name>}\ \texttt{<expression>}^{+} \tag{3.4}$$

$$\texttt{<declaration>} ::= \texttt{<name>}\ \texttt{<pattern>}^{+}\ \text{‘}\texttt{:=}\text{’}\ \texttt{<expression>} \tag{3.5}$$

$$\texttt{<pattern>} ::= \texttt{<pattern-value>}\ (\ \text{‘}\texttt{.}\text{’}\ \texttt{<pattern>}\ )\ ? \tag{3.6}$$

$$\texttt{<pattern-value>} ::= \text{‘}\texttt{0}\text{’} \mid \text{‘}\texttt{\_}\text{’} \mid \text{‘}\texttt{(}\text{’}\ \texttt{<pattern>}\ \text{‘}\texttt{)}\text{’} \mid \texttt{<name>} \tag{3.7}$$

$$\texttt{<name>} ::= [\text{‘}\texttt{a}\text{’}\texttt{-}\text{‘}\texttt{z}\text{’}]\ (\ [\text{‘}\texttt{-}\text{’}\ \text{‘}\texttt{a}\text{’}\texttt{-}\text{‘}\texttt{z}\text{’}]^{*}\ [\text{‘}\texttt{a}\text{’}\texttt{-}\text{‘}\texttt{z}\text{’}]\ )? \tag{3.8}$$

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

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

Definition 7. *A declaration signature in Δ consists of the function name and the number of parameters it has.*

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

3.4 Semantics

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

Allow mutual recursion

Perhaps pattern matching must be exhaustive in general.

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

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

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

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

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

3.4.1 The memory model

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

[3] The syntax applied here is described in further detail in Appendix A.2 (32) .

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

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

Functions and variables

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

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

Making Δ higher order

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

3.4.2 Function declarations

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

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

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

3.4.3 Expression evaluation

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

$$\frac{\langle \text{SINGLE}, x, \sigma \rangle \rightarrow \langle v, \sigma \rangle \vee \langle \text{CHAIN}, x, \sigma \rangle \rightarrow \langle v, \sigma \rangle}{\langle x, \sigma \rangle \rightarrow \langle v, \sigma \rangle} \tag{3.10}$$

$$\frac{x \rightarrow e \wedge \langle e, \sigma \rangle \rightarrow \langle v, \sigma \rangle}{\langle \text{SINGLE}, x, \sigma \rangle \rightarrow \langle v, \sigma \rangle} \tag{3.11}$$

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

3.4.4 Element evaluation

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

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

3.4.5 Function application

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

3.4.6 Pattern matching

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

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

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

Definition 8. *A pattern is an unlabeled of binary tree which is either empty or consists of an unlabeled node with a* 0, _, *name, or a pattern as it's left and right child.*

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

Definition 10. *The function* $\cdot : \mathbb{P} \times \mathbb{P} \rightarrow \mathbb{P}$ *constructs a pattern node with the two arguments as it's left and right child, respectively.*

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

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

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

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

3.5 User input

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

3.6 Size

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

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

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

We start by formally proving that Definition 11 (9) yields a many-to-one mapping of Δ's data values to the natural numbers.

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

Proof.

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

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

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

□

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

Proof.

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

Assumption

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

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

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

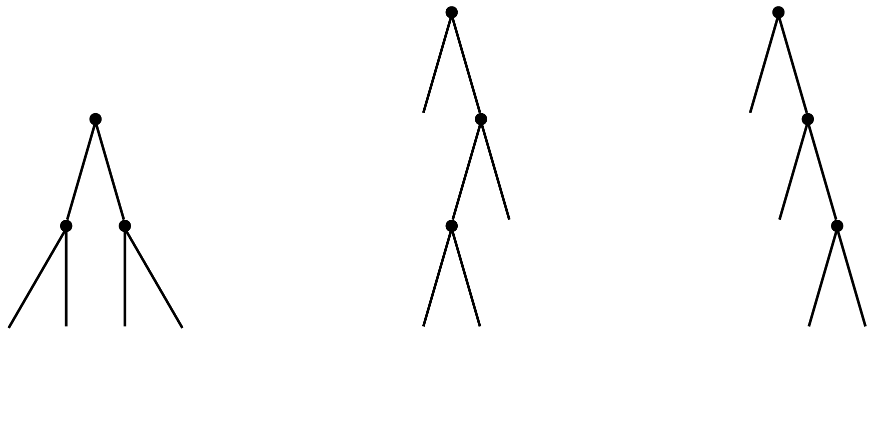
Hence, any value in Δ must have one and only one representation in $\mathbb{N}$.

□

Definition 11 (9) *almost* allows us to devise an algorithm to compare the sizes of data values. The problem withstanding is that two different values can have rather diverging tree representations. Hence, comparing them, using only the operations defined in § 3.4 (7) , is seemingly impossible unless we initially, or along the way, transform the binary trees being compared into some sort of a *standard representation*. We'll define this representation, recursively, as follows:

Definition 12. *A binary tree in standard representation is a binary tree that either is a leaf or a node having a leaf as it's left child and a binary tree in standard representation as it's right child.*

Intuitively, a binary tree in standard representation is just a tree that only descends along the right side. Comparing the sizes of two trees in this representation is just a matter of walking the descending in the two trees simultaneously, until one of them, or both, bottom out. If there is a tree that bottoms out strictly before another, that is the lesser tree by Definition 11 (9) . Figure 3.2 (11) showcases some examples.



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

Figure 3.2: Three trees of various shapes but equal size.

3.6.1 `normalize/1`

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

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

Correctness

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

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

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

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

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

Time complexity

Coming soon..

Space complexity

Coming soon..

3.6.2 `less/2`

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

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

Definition 13. *A false value is represented by a leaf tree. A true value is represented by a non-leaf tree, i.e. a node.*

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

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

Correctness

Proof. Given Definition 12 (10) , and the assumption that Normalize(A) computes the standard representation of A, we know the following:

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

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

Given Definition 11 (9) , $A < B$ iff it bottoms out before B, that is, we reach an instance of the recursion where both $IsLeaf(A)$ and $IsNode(B)$ hold. In all other cases $A \geq B$, the cases specifically are:

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

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

□

Time complexity

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

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

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

Space complexity

Coming soon..

3.7 Built-in high-order functions

Although D is initially a first-order language, we will ignore that limitation for a bit and define a few higher-order functions to provide some syntactical sugar to the language. Beyond the discussion in this section, these higher-order functions should be regarded as D built-ins.

Branching

In the following definition, the variable names true and false refer to expressions to be executed in either case.

```
if 0 _ false := false
if _._ _ true := true
```

As you can see, we employ the C convention that any value other than 0 is a "truthy" value, and the expression true is returned.

Although the call-by-value nature of the language does not allow for short-circuiting the if-statements defined in such a way, this shouldn't be any impediment to further analysis.

3.7.1 Boolean operations

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

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

3.8 Sample programs

As an illustration of the language syntax, the following program reverses a tree:

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

The following program computes the Fibonacci number n:

Assume that the argument is

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

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

Note: The return value is not normalized.

Chapter 4

Size-change termination

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

The idea is then to construct a graph where such points are nodes, and the arcs in between them represent a transfer of control between the primitive operations, that would otherwise occur under the execution of the program. Such a node may have variable in-degree and out-degree from one given primitive. For instance, a condition branch would usually have two possible transfers of control depending on the outcome of the condition. Hence, it serves useful to label arcs depending on when they are taken. The conditions should clearly not overlap to avoid non-determination.

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

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

4.1 Control flow graphs in Δ

4.1.1 Start and end nodes

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

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

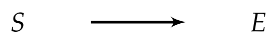


Figure 4.1: A control flow graph for the empty Δ program.

4.1.2 Function clauses

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

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

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

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

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

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

We can now draw a control flow graph for the program define in Listing 4.1 (16) as shown in Figure 4.2 (16) .

```
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 Δ program, always returning `0.0.0`.

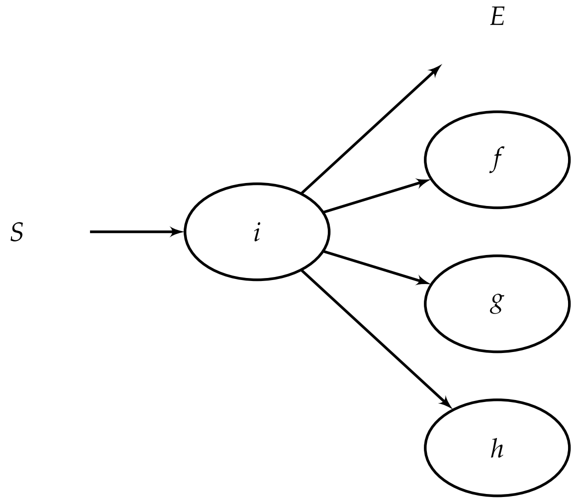


Figure 4.2: A control flow graph for the Δ program in Listing 4.1 (16) . The graph does not explicitly specify back-propagation of control, if any.

4.1.3 Call cycles

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

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

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

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

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

4.1.4 Disregarding back-propagation

It is worth noting that in Figure 4.2 (16) , 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 (17) . Here, we've already disjunctively labelled all of the calls in the expressions. This call graph is drawn in Figure 4.3 (18) .

```
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.

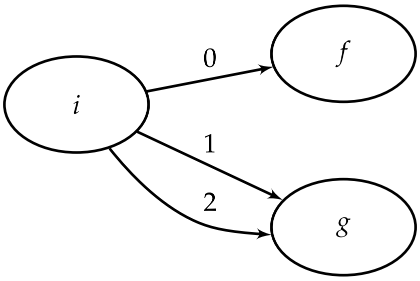


Figure 4.3: A control flow graph for the Δ program in Listing 4.2 (17) .

4.1.8 Multiple clauses

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

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

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

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

Hence, we can now draw the call graph for the program defined in Listing 4.3 (18) as shown in Figure 4.4 (18) .

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

f input
```

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

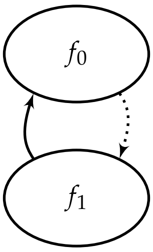


Figure 4.4: A control flow graph for the program defined in Listing 4.3 (18) .

For a more complex example, let's consider the call graph for the program `reverse` introduced in § 3.8 (13) . The program is repeated in annotated form in Listing 4.4 (18) , and its corresponding call graph is shown in Figure 4.5 (19) .

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

```

[2] See § 3.4 (7) .

```
reverse input
```

Listing 4.4: An annotated version of the program `reverse` introduced in § 3.8 (13) .

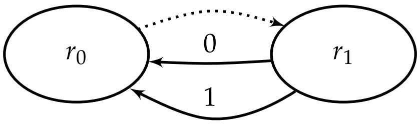


Figure 4.5: A control flow graph for the Δ program in Listing 4.4 (18) .

4.1.9 Deeply nested function calls

Blah

4.2 Size-change termination principle

Consider the program in Listing 4.3 (18) and it's corresponding call graph in Figure 4.4 (18) . Without any further information about the control transitions, the program seemingly loops indefinitely. However, there are some things that we can deduce about the control transitions.

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

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

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

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

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

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

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

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

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

4.2.1 Fail transitions

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

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

Proof. Follows from the semantics of Δ. □

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

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

4.2.2 Success transitions

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

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

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

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

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[5] 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 $\{v > \mathtt{a}, v > \mathtt{b}, v \geq \mathtt{a.0}, v \geq \mathtt{0.b}\}$, assume that the destination clause has the corresponding pattern `x.y`. The question henceforth is how do we draw the relationship that $\mathtt{a} \equiv \mathtt{x}$ and $\mathtt{b} \equiv \mathtt{y}$, 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, $\{v > \mathtt{x}, v > \mathtt{y}, v \geq \mathtt{x.0}, v \geq \mathtt{0.y}\}$. 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 $v \equiv \mathtt{a.b}$ is lost.

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

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

[5] 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.

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. $v \geq$ `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 Δ[6]. We henceforth refer to such an expression as p^s[7]. Given a clause with the pattern p^d[8], we can easily deduce the set N^d_{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^d_{vars} given the tuples $(p^s, N^s_{vars}, N^s_{calls})$ and (p^d, N^d_{vars}).

4.2.3 Pattern matching

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

For this purpose we will regard the tuples (P^s, N^s_{vars}) and (P^d, N^d_{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^d is the list of corresponding actual patterns in the destination. Furthermore, let N^s_{vars} and N^d_{vars} be unary functions of the type $\mathbb{P} \rightarrow \mathbb{N}^*$, accepting a pattern and yielding the variable names that are contained both in the input pattern and the sets N^s_{vars} and N^d_{vars}, respectively.

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

We initially define ϕ to yield the value $\perp$ for all arguments. We will continuously modify this definition as we process p^s and p^d. We denote this within the semantics in a manner similar to the state σ in the semantics[9]. However, ϕ is now a binary "memory", requiring both a destination 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 $\phi\left(N^d_{vars}(p^d), n^s\right) \mapsto <$, meaning that all the destination variables used in p^d are strictly less than the source variable n^s.

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

$$\frac{\left\langle \mathrm{A}, p^d, p^s, \phi \right\rangle \rightarrow \phi' \vee \left\langle \mathrm{B}, p^d, p^s, \phi \right\rangle \rightarrow \phi' \vee \left\langle \mathrm{C}, p^d, p^s, \phi \right\rangle \rightarrow \phi' \vee \left\langle \mathrm{D}, p^d, p^s, \phi \right\rangle \rightarrow \phi' \vee \left\langle \mathrm{E}, p^d, p^s, \phi \right\rangle \rightarrow \phi'}{\left\langle p^d, p^s, \phi \right\rangle \rightarrow \phi'} \tag{4.1}$$

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 destination variables that might appear in the corresponding p^d. Hence, ϕ need not be modified.

$$\frac{(p^s \rightarrow 0 \vee (p^s \rightarrow n^s \wedge n^s \notin N^s_{vars})) \wedge \phi \rightarrow \phi'}{\left\langle \mathrm{A}, p^d, p^s, \phi \right\rangle \rightarrow \phi'} \tag{4.2}$$

[6] See § 3.3 (6) if you're uncertain.

[7] Where s stands for *source*.

[8] Where d stands for *destination*.

[9] See § 3.4 (7) .

This has a symmetrical case. Indeed when p^d is neither a destruction, nor any name n^d, 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, ϕ need not be modified in such a case either.

$$\frac{\left(p^d \to 0 \lor p^d \to \_\right) \land \phi \to \phi'}{\left\langle \mathrm{B}, p^d, p^s, \phi \right\rangle \to \phi'} \tag{4.3}$$

If p^d is the name pattern n^d, 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^d by the semantics of Δ. However, we are not concerned with this relation, as we would like to know when a value is decreased from source to destination, 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, $\leq$:

$$\frac{p^d \to n^d \land p^s \to n^s \land n^s \in N^s_{vars} \land \left\langle \phi\left(n^d, n^s\right) \mapsto \leq \right\rangle \to \phi'}{\left\langle \mathrm{C}, p^d, p^s, \phi \right\rangle \to \phi'} \tag{4.4}$$

If p^d 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^d, i.e. $N^d_{vars}(p^d)$, are all strictly less than the variable in n^s:

$$\frac{p^d \to p^d_1 \cdot p^d_2 \land p^s \to n^s \land n^s \in N^s_{vars} \land \left\langle \phi\left(N^d_{vars}(p^d), n^s\right) \mapsto < \right\rangle \to \phi'}{\left\langle \mathrm{D}, p^d, p^s, \phi \right\rangle \to \phi'} \tag{4.5}$$

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

$$\frac{p^d \to p^d_1 \cdot p^d_2 \land p^s \to p^s_1 \cdot p^s_2 \land \left\langle p^d_1, p^s_1, \phi \right\rangle \to \phi'' \land \left\langle p^d_2, p^s_2, \phi'' \right\rangle \to \phi'}{\left\langle \mathrm{E}, p^d, p^s, \phi \right\rangle \to \phi'} \tag{4.6}$$

4.3 Graph annotation

Hence, we can deduce from Listing 4.3 (18) , 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 ↓. 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 (18) . These observations are summarised in Figure 4.6 (23) .

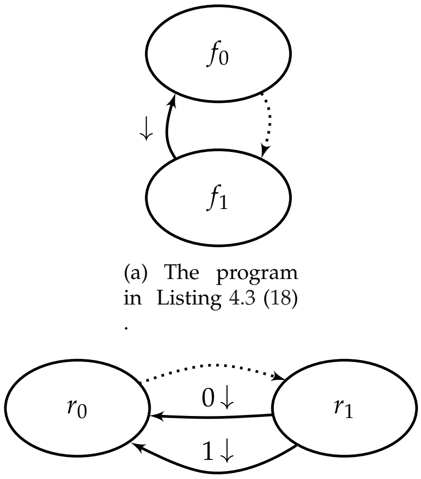
4.3.1 Calls to multivariate functions

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 § 3.6.2 (12) . We use this function to define the program in Listing 4.5 (22) . The corresponding call graph is shown in Figure 4.7 (23) .

```
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.

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.



(a) The program in Listing 4.3 (18)
.

(b) The program in Listing 4.4 (18) .

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

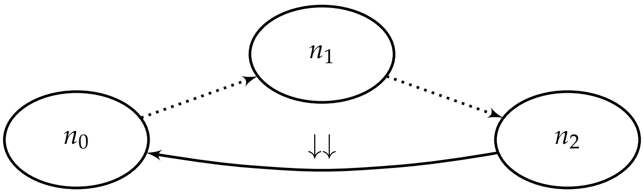


Figure 4.7: A control flow graph for the program defined in Listing 4.5 (22) .

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 (23) .

```
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 $g_1 \to g_0$, or not. Specifically, if the arguments to g are of the form `0._` and `_.0`, respectively, the size is *not* decreased by the call. We'll denote such transitions by the symbol $\Downarrow$. We can now draw the call graph for the program in Listing 4.6 (23) as in Figure 4.8 (23) .

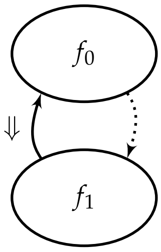


Figure 4.8: An annotated call graph for the program in Listing 4.6 (23) .

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

Extending size-change termination

One trouble with size-change termination as described in the previous chapter is Lemma 0.3 (19) . This lemma makes size-change termination weak in the sense that the overall “shape shifting” in a given call cycle is *not* considered, and instead, the individual control transitions of a cycle are constrained to transitions that do not increase values. However, there may be programs that have control transitions or even control transition cycles that increase a value until some condition is met.

Consider the program in Listing 5.1 (25) 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.

```
$f_0$: f a.b.c.d := a
$f_1$: 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

Before we can speak of extending size-change termination to determine the halting property for programs in the same class as Listing 5.1 (25) , we need to formally define that class.

In particular, the idea behind this extension is to be able to deduce the halting property for programs that either decrease a value in each iteration of a call cycle, or increase it until a condition is met. Actual conditions in Δ can only be expressed in terms of patterns in function clauses.

Hence, we swiftly 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 § 3.8 (13) . The function `fibonacci-aux` seemingly increases a value until a condition is met, in particular, that we count down to `0`. However, due to the fact that we count down by 1 in *every* recursive clause of the `fibonacci-aux` function, 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 patterns. Consider therefore the program in Listing 5.1 (25) . The main function of the program has only one terminal clause, the one that accepts a shape as in Figure 5.1 (26) . If the incoming value v has any other shape, i.e. either a shape as in Figure 5.2 (26) , Figure 5.3 (26) or Figure 5.4 (26) , then the recursive clause f_1 is chosen. For any given value v, the clause f_1 replaces the right-most child of the value, which is always `0`, with a node.

For instance, the smallest possible input value v is `0`. If passed such a value, f_1 transforms it into a value that has a shape that corresponds to Figure 5.3 (26) , which in turn transforms the value into one that matches Figure 5.4 (26) , which in turn transforms the value into one that matches Figure 5.1 (26) , i.e. the terminal clause. What's more, there are infinitely many other values that will match the shape Figure 5.3 (26) , and for each of them, the clause f_1 will transform them into values that match Figure 5.4 (26) , which will transform them into values that match Figure 5.1 (26) .

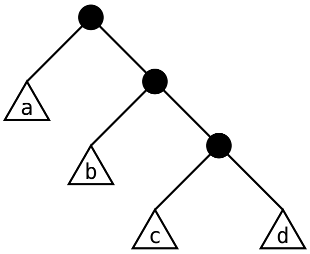


Figure 5.1: The shape that the clause f_0 in Listing 5.1 (25) will accept.

Figure 5.2: The pattern `0`.

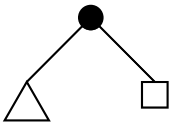


Figure 5.3: The pattern `a.0`.

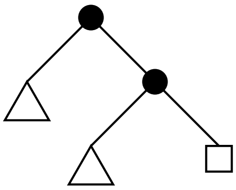


Figure 5.4: The pattern `a.b.0`.

We shall henceforth say that a clause such as f_1 *shape shifts* the input value v to *eventually* match the shape required by the terminal clause f_0. The task then becomes to determine for each call cycle in a program whether it shape shifts any given initial argument to eventually match a 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 Δ.

5.2.1 Deducing `0`

The `.` operator in the patterns of function clauses in Δ 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, s.t. a pattern like `(a.b).c.d` is the same as `(a.b).(c.d)`.

Consider the function defined in Listing 5.2 (26) . If f_0 and f_1 fail to match some input value v, then v must be of the shape `0.x`, that is, on entry to f_2, `d` is *always* bound to `0`, and `e` is always bound to some value $v' \geq 0$.

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 (27) exhibits. Here, if g_0 and g_1 fail to match some input value v, then the shape of the v is either `0.x` or `x.0` where `x` ≥ 0. However, one thing is certain, and that is that v can't have the shape `y.z` where `y` > 0 and `z` > 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 Patterns and shapes

In the following we shall regard only unary clauses, but the lemmas apply equally to multivariate clauses.

Lemma 0.5. *Given a valid function definition, a value of any shape matches one and exactly one clause.*

Proof. We know that given two consecutive unary clauses c_1 and c_2, having the patterns p_1 and p_2, $p_1 \curlyvee p_2$. This superimplies that any shape that can be matched by p_1 will also be matched by p_2, however, given the semantics of Δ, c_2 will not be considered if p_1 matches. □

5.3 The extension

5.3.1 Annotation

5.3.2 Patterns and shapes

For any value coming from the outside world, i.e. via the `input` function, we may assume only the shape $\triangle$. Hence, given a program like Listing 5.1 (25) , we start analyzing the function `f` by assuming nothing about the input argument, i.e. annotate it with $\triangle$. The value may match either clause, but it will match exactly one. When the value matches a clause that indicates that a value has a certain shape, indeed this is what the pattern of the clause is – a shape specification. 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* the possible shapes. Figure 5.5 (27) illustrates this initial step.

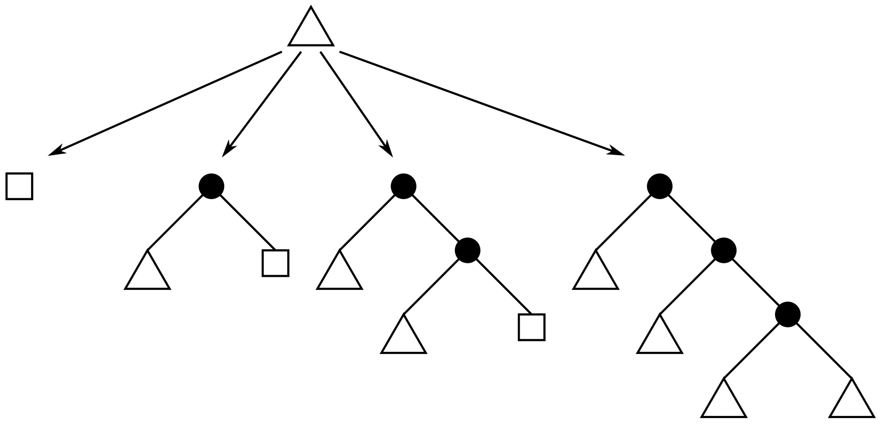


Figure 5.5: Initializing shape analysis for the program in Listing 5.1 (25) .

Lemma 0.6. *If the first clause of a function definition matches a value, it indicates that the value is of exactly one shape, the one discretely indicated by the corresponding pattern declaration.*

Proof. The first clause is the first one considered by Δ's runtime when a function call is made. No other patterns have been attempted at this point, and hence, no deductions about the shape of the value have been made. If the pattern matches, then it certainly has the shape defined by the corresponding pattern declaration, otherwise, the value wouldn't have matched the pattern. □

Definition 15. *Given two shapes, A and B, we say that they are disjoint if the sets of values matched by A and B are disjoint.*

Definition 16. *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 $\exists\, s_1, s_2 \in X \Cup Y : s_1 \Upsilon s_2$, then s_1 is removed from $C \Cup Y$.*

Lemma 0.7. *Given a shape, there is a finite number of disjoint shapes that are disjoint with that shape.*

Proof. A shape in Δ 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 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. □

Lemma 0.8. *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 0.6 (27) .

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 Δ, 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 0.7 (28) , the $\overline{X} \Cup Y$ must be finite as well. □

5.4 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.

5.4.1 Recursive and terminal clauses

Lemma 0.9. *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 0.10. *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.

```
c0: count x 0 := x
c1: 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 $C = A - B$.

Appendix A

Notation

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

A.1 Extended-BNF

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

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

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

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

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

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

A.1.2 Constructs borrowed from regular expressions.

The use of single quotes around all terminals allows us to give characters such as (,),],], $*$, $+$, and * special meaning, namely:

Construct	**Meaning**
$(\ldots)$	Entity group
$[\ldots]$	Character group
-	Character range
$*$	0-∞ repetition
$+$	1-∞ repetition
?	0-1 repetition

Table A.2: Constructs borrowed from regular expressions.

An entity group is a shorthand for an auxiliary nonterminal declaration. This means, for instance, that using the alternation symbol within it would mean an alternation of entity sequences within the entity group rather than the entire declaration that contains the entity group.

A character group may only contain single character terminals and an alternation of the terminals is implied from their mere sequence. It is identical to an auxiliary single character nonterminal declaration. A character range binary operator can be used to shorten a given character group, e.g. [`‘a’-‘z’`] implies the list of characters from `‘a’` to `‘z’` in the ASCII table. Moreover, a character range is the only operator allowed in a character group.

Applying the repetition operators to either the closing brace of an entity group or the closing bracket of a character group has the same effect as applying the repetition operator to their respective hypothetical auxiliary declarations.

A.1.3 Nonterminals as sets and conditional declarations

Another extension to the original BNF is the ability to use nonterminals as sets in declaration conditions. For example, if the two nonterminals, `<type-name>` and `<constructor-name>`, are both declared in terms of the `<literal>` nonterminal, but type names and constructor names should not intersect in a given program, then we can append the following condition to one or both declarations:

$$\text{s.t. } \texttt{<type-name>} \cap \texttt{<constructor-name>} \equiv \emptyset$$

Where the shorthand s.t. stands for “such that”. This implies that the nonterminals `<type-name>` and `<constructor-name>` represent the sets of character sequences that end up associated with the respective nonterminals for any given program, and can be used in conjunction with regular set notation.

A.2 The structured operational semantics used in this work

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

A.2.1 Some general properties

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

A.2.2 Atoms

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

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

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

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 ⇒ 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 ⇒ 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 → operator

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

The ∧ operator

The ∧ 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 ∧ 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 ∨ operator

The ∨ 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. ∨

2. ∧

[1] See Appendix A.2.2 (32) .

[2] See Appendix A.2.3 (33) .

3. →