Proving Termination of Pointer Programs on Top of Symbolic Execution

Ira Fesefeldt

Advisor:
Christoph Matheja

Examiners:
Prof. Thomas Noll
Prof. Joost-Pieter Katoen

A thesis presented for the degree of
Master of Science

Chair for Software Modeling and Verification
RWTH Aachen
Germany

Communicated by Prof. Joost-Pieter Katoen.

Contents

Abstract **v**

Declaration of Originality **vii**

1 Introduction **1**
1.1 Informal Example . . . 2
1.2 Related Work . . . 4
1.3 Outline . . . 5

2 Hypergraph Transition Systems **7**
2.1 Hypergraphs . . . 7
2.2 Concrete Hypergraph Transition System . . . 12
2.3 Hyperedge-Replacement Grammars . . . 16
2.4 Abstract Hypergraph Transition System . . . 22

3 Predicates **31**
3.1 Compatible Predicates . . . 31
3.2 Monotonic Directed Predicates . . . 34
3.3 Terminating Predicates . . . 36
3.4 Relative Index . . . 37
3.5 Index Abstraction Rules . . . 40

4 Predicate Abstraction Analysis **47**
4.1 Abstracting Predicates . . . 47
4.1.1 Assigning Domain . . . 48
4.1.2 Transfer Function . . . 49
4.1.3 Overall Analysis . . . 55
4.1.4 Widening . . . 55
4.2 Soundness of Overapproximating Predicates . . . 56
4.2.1 Soundness of Materialisation . . . 57
4.2.2 Soundness of Hypergraph Operations . . . 58
4.2.3 Soundness of Abstraction . . . 59
4.2.4 Soundness of Combination . . . 60

5 Proving Termination **63**
5.1 Decreasing Chains using one Hyperedge . . . 63
5.2 Decreasing Chains using a Set of Hyperedges . . . 67
5.3 Proving Overall Termination . . . 69

6 Inequality Predicates **71**
6.1 Integers and Relative Integers 72
6.2 Inequality Compatible 76
6.3 A Tree Example 79

7 Conclusion **89**
7.1 Further Work 90

A Set Theory **93**

B Monotonic Data Flow Analysis **99**

C Additionally Proofs **103**
C.1 Analysis Proofs 103
C.2 Inequality Proofs 105

Abstract

We present a static analysis to prove termination of pointer programs. Our analysis builds upon an existing symbolic execution, which, for a given program and a graph grammar guiding abstraction and concretisation, overapproximates program executions. While the resulting abstract transition system enables verification of partial correctness properties, e.g. preservation of simple data structures invariants, it loses the precise number of loop iterations or method invocations. To combat this loss, we propose an abstraction refinement that preserves certain variants. In particular, the refined transition system is capable of tracking how the size of data structures changes in any loop iteration. This allows us to synthesise a suitable ranking function.

Declaration of Originality

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

....
date (signature)

Chapter 1

Introduction

The halting problem, i.e. the question whether a given program terminates on a given input, appeared around 1950-1960 in the literature [Cop04, p.40]. It was probably first mentioned by Kleene in *Introduction to Metamathematics*, but he only covered it briefly [Kle52, p.382]. Davis then proved in his work *Computability and Unsolvability* from 1958 that "there exists a Turing machine whose halting problem is recursively unsolvable"[Dav58, p.70]. Here, the halting problem was associated with a fixed Turing machine and an arbitrary input. Today we often describe the halting problem as an arbitrary Turing machine and an arbitrary input.

In this thesis, we do not consider Turing machines but *pointer programs*. We introduce a technique to prove the termination of pointer programs. A *pointer* is an object that stores a *memory address*. By *dereferencing* a pointer, we can access the value at the memory address of the pointer. Saving pointers inside memory allows the construction of complex *data structures*, e.g. lists, trees, graphs. Many modern programming languages allow the use of pointers, although some restrict possible operations, e.g. the addition of a pointer with a constant. Nevertheless, programming with pointers is still hard and especially avoiding potential unwanted infinite computations is hard. As the halting problem on pointer programs is unfortunately undecidable, we give a technique that is only capable of proving that a program is indeed terminating for a given set of valid memory configurations (e.g. memory configurations that encode a list), but is not able to prove the opposite.

We present a technique that uses *hypergraphs* to represent memory configurations. Hypergraphs are an extension of graphs, which allow the use of *hyperedges*, i.e. edges that can connect any number of vertices instead of only two. To describe a set of memory configurations, we use *graph grammars* which allow hyperedges to be replaced by a hypergraph. Graph grammars can be used to describe a vast range of data structures. This also allows us to abstract a set of infinitely many hypergraphs by one hypergraph.

We use this *graph abstraction* to abstract the *operational semantics* of a pointer program. The operational semantics of a program is a transition system that describes every possible run of the program. We also call the operational semantics of a program the *concrete transition system*. Although this transition system might be infinite, we hope to get a finite transition system that abstracts the concrete transition system. This abstraction may yield an information loss

and especially may introduce loops that do not correspond to any infinite run. To prove that these loops are indeed not corresponding to any infinite run, we synthesise *ranking functions*. A ranking function usually maps the states of the concrete transition system to positive numbers, where every state must have a higher number then its successor states. If a ranking function exists, the concrete transition system does not have any infinite run, and thus the program terminates on the input(s).

We synthesise a ranking function by annotating the hypergraphs of the abstract transition system with *indices*. The indices describe a property of the data structure that often leads to ranking functions. As an example, for a list data structure, we may use its size, and for a tree data structure, we may use its tree height. If we calculate that the annotated indices are decreasing after one loop-iteration, we then conclude that there exists a ranking function. However, we do not construct the ranking explicitly but prove the lack of corresponding infinite paths in the concrete transition system for a loop in the abstract transition system instead.

1.1 Informal Example

To give a rough sketch of how the technique developed in this thesis works, we consider a small program and prove its termination. The example program P is depicted in Example 1.1. P iterates over a list and removes every visited list object. It removes an object in three steps. First it saves the object's address of the current pointer `list` in the temporal variable `t`, then it sets the current pointer `list` to its object successor `list.next`, and lastly, the old object saved in `t` is removed. As soon as only one object is left, the while-loop terminates, and the program stops. We assume that the input for this program is a list of size j. In the following, we informally explain how to prove this observation using our technique.

Example 1.1: A program P that removes the list saved in the variable `list`

```
while(list.next != null){
        t := list;
        list := list.next;
        remove(t);
}
```

Concrete Transition System We define the semantics of P in terms of the concrete transition system in Figure 1.1. Every state represents one iteration of the while-loop. We encode the states as graphs, which model the memory configuration before the current iteration. The start state consists of a graph H_j representing a list with j elements. As one iteration removes one element from the list after one iteration $j-1$ elements are left in the list, resulting in the graph H_{j-i}. These steps repeat until we reach the state with the graph H_1, where only 1 element is left. The last element has a next selector to `null` and thus here the run stops. We do not fix the value j but instead want to investigate if the program terminates for every j.

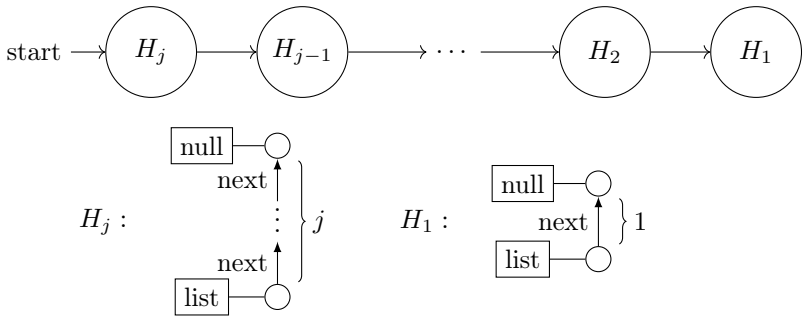


Figure 1.1: The Concrete transition system for the program P, for which the list length is decreasing in every iteration.

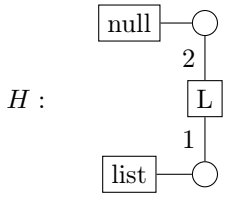


Figure 1.2: The abstracted graph H, which abstracts all H_i by replacing the list with a placeholder edge labelled L.

Abstracted Graphs Since the list can be arbitrarily large, we abstract the list by a placeholder. We abstract a graph H_i by graph H, as depicted in Figure 1.2. In graph H, the nodes where `list` is attached and where `null` is attached remain and are then connected by an edge with label L that represents a list (especially a list of size i). Later, We discuss how to define these placeholders as nonterminals in a graph-grammar to formally define the correspondence between the placeholder and the set of lists. Using this placeholder, the graphs $H_j, \ldots H_1$ are transformed into H. However, H represents not only every graph $H_j, \ldots H_1$ but also graphs that represent a list of any size. This abstraction leads to a loss of size information of the graphs but gains us the method to abstract an infinite set of graphs.

Abstract Transition System We generate an abstract transition system by replacing every graph in the concrete transition system from Figure 1.1. The abstract transition system then only consists of the state with the graph H. To also abstract the transition $H_i \to H_{i-1}$, we also need to add a self-loop to this state. The transition system also shows that the data structure remains a list after every iteration. We lose the information that the concrete transition system does not have infinite paths. Moreover, we lose size information of every graph, which would help us to prove the termination. These losses are accepted, to be able to compute a finite abstraction of the concrete transition system.

Indices and Predicates To regain the size information, we annotate the hyperedges with indices representing relative size information. These indices

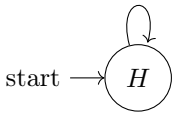


Figure 1.3: The abstract transition system for program P, which consists only of a state with the graph H and a self loop.

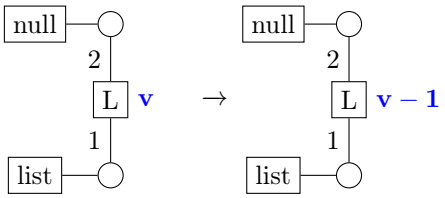


Figure 1.4: The annotation of the self loop $H \to H$ in the abstract transition system in Figure 1.3.

are a sum of variables and an integer. Variables are used to set the indices in a relative context. We assign the edge with label L afresh variable $\mathbf{v}$. We can interpret the variable $\mathbf{v}$ as the size of the list before the current iteration. As the current iteration is arbitrary, this may be any integer. We use this variable to compute the change after an iteration. To give indices a meaning, we formalise predicates that are *compatible* with the graph grammar used to express the meaning of the edge with label L. We formalise a predicate that describes the size of a list in our example. This predicates takes an integer and a graph and holds if and only if the graph is indeed a list and the integer is the graph's list size.

Annotations in Graphs To compute the change of the graph's list size, we annotate the edge labelled L with a variable $\mathbf{v}$. Then we analyse what happens during this transition. The result of the computation is depicted in Figure 1.4. The pointer `list` moves to the next object, and the last object is removed. The size of the list should be decreased by one, and thus, relative to the previous iteration, the list now has size $\mathbf{v} - \mathbf{1}$. We annotate the graph after the transition with the value $\mathbf{v} - \mathbf{1}$. As the index is decreasing, we conclude that every iteration decreases this index by at least 1. We conclude furthermore that after at most $\mathbf{v}$ many iterations, the list has size 0. However, the program would have an exception if the list is empty, i.e. the program has to stop before that.

1.2 Related Work

This thesis is based on a graph guided symbolic execution which aims to abstract the semantics of a pointer program. The operational semantics of pointer programs can be abstracted via hyperedge-replacement grammars for which forward derivations yield an *abstraction* and backward derivations yield a *materialisation* [HJKN15]. We now look at similar approaches to extend either the

same abstraction method or similar abstraction methods.

Indexed Grammars Another approach to extend this graph-based symbolic execution is to extend the grammar to an *indexed grammar*. These indexed grammars are used to abstract *relational properties*, e.g. the difference of two subtrees in a balanced binary tree [AJMN18]. Similarly to our approach, the grammar rules are extended to formalise the change of the indices after derivations. The indices are assigned while the symbolic execution is computed and not in the form of post-processing. Furthermore, these indices often only allow invariant properties, i.e. properties that do not change after a loop. Our approach, however, aims to give an upper bound of a variant index using a post-processing algorithm that annotates the hyperedges in an already computed abstract transition system to prove the termination of the program.

Separation Logic Separation Logic is a logic which uses additionally to the commonly known *logical disjunction* $\vee$ the *separating conjunction* $*$. The separation conjunction expresses that the left and the right sides are defined in separate domains, i.e. the right side is not allowed to share the same values for variables then the left side and vice-versa [BDP11]. Separating conjunction happens to be quite useful to describe pointer-arithmetic memories. Separation logic is related to graph-based abstraction, as there exists a correspondence between a fragment of separation logic and a fragment of hyperedge-replacement grammars [Jan17].

Indexed Separation Logic Indexed separation logic aims to proof statements on *indexed predicates* [CDNQ12]. While not designed to be used to prove termination, it is similar to this thesis' approach that it assigns indices to the predicates (which correspond to hyperedges) and using folding and unfolding mechanism (which corresponds to forward and backward derivations respectively) to change the indices while computing a proof. The folding and unfolding mechanism may need additional invariants, which this thesis does not require.

Proving Termination with Separation Logic Another approach using separation logic uses the *terminating statement* $\downarrow$ and additionally *proof-rules* for this statement to formulate a cyclic proof that shows termination of a program [BBC08]. This technique does not use any indices and still is capable of proving the termination of some programs by cleverly using the proof rules. Every termination statement is formulated as a sequence consisting of a logical description of the memory (e.g. "$list(x, \mathbf{null})$") on the left side and the termination symbol $\downarrow$ on the right side with the current program point i attached to the sequence symbol. For example, such a sequence would be "$list(x, \mathbf{null}) \ \vdash_i \downarrow$". A valid proof in this system would then prove the termination of the loop.

1.3 Outline

This thesis expects some basic knowledge about set theory and static program analysis. Theorems and definitions we use in this thesis are, however, formulated in Appendix A and Appendix B. The *first chapter* of this thesis gives a

rough overview of the topic. In the *second chapter*, we discuss the symbolic execution on which this thesis bases. There we introduce *hypergraphs*, which we use to represent the memory within program states. Afterwards, we discuss the representation of programs as a (concrete) *transition system*. This transition system consists of locations, which represent the program position, and hypergraphs, which represent the program memory. To abstract hypergraphs, we then use hyperedge-replacement grammars. This abstraction yields an abstract hypergraph transition system. The *third chapter* considers *compatible predicates* which we use to give the indices we annotate to hyperedges a meaning. These predicates are defined on indices and hypergraphs and behave well on rules, i.e. indices can be decided after applying forward or backward derivations using the old indices and the derivation rule. In the *fourth chapter*, we discuss how to analyse a given abstract transition system and propagate the indices through the transition system. The propagated indices are then be used to check if the index is decreasing, which we can ultimately use to prove termination. We prove in the *fifth chapter* that a decreasing index proves partial termination for a loop or more precise for a state. Analysing all loops, or more precisely a state for each loop, we can then prove termination for the program. Up to now, all chapters discussed the topic on a relatively abstract level. We change this in the *sixth chapter* by introducing an integer set for the indices and inequalities for compatible predicates. By using a subset of inequalities, we define predicates that satisfy almost all properties, except for compatibility. In the *seventh chapter*, we conclude this thesis and explain what further work remains.

Chapter 2

Hypergraph Transition Systems

In this work, we focus on analysing a graph-based symbolic execution of pointer programs and derive from our proposed analysis whether the program will terminate. We assume that the pointer program is transformed into a feasible abstract transition system by the given symbolic execution. Such an abstract transition system abstracts a concrete transition system that describes the semantics of a pointer program. We will not show how to calculate such an abstract transition system since this has been done before in other literature [Jan17, AJMN18, HJKN15].

We will introduce (concrete) transition systems which states are tuples of a location, which gives the type of the state, and a (hyper-)graph, which represents the memory configuration of the pointer program in this state. The states of the abstract transition system also consist of a location and a hypergraph, but this time the hypergraph describes a language of hypergraphs defined by a graph grammar.

2.1 Hypergraphs

We represent the configuration of the memory by *hypergraphs*. A hypergraph consists of a finite set of *vertices* and a set of *hyperedges*. Hyperedges can (in contrast to standard edges) connect arbitrary many vertices. A function *att* is used to map hyperedges to their *attached* vertices. Every edge connects all attached vertices. The attached vertices are totally ordered (which extends the notion of directed edges to hyperedges). The order is formalised by representing the attached vertices as a *sequence* (compare Appendix A.7). Every hyperedge has a *label* given by the function *label*. The label of the hyperedges comes from a set of symbols Σ. Lastly we have *external vertices*, which will be used later to formalise *hyperedge-replacement grammars*. We use hyperedge-replacement grammars in order to abstract and materialise graphs. In this sense, hypergraphs will represent a set (or more precise a language) of graphs and thus is also able to abstract a set of graphs.

Definition 2.1 (Hypergraph (HG) [Hab92, p.7])
Let Σ be a finite ranked alphabet (see Appendix A.4) with the ranking function rank. We define a (labelled) hypergraph H over Σ as

$$H = (V, E, att, label, ext)$$

where V is a finite set of vertices and E a finite set of hyperedges. The function $att \in E \mapsto V^*$ maps every hyperedge to its sequence of attached vertices. The labelling function $label \in E \mapsto \Sigma$ assigns every hyperedge a symbol in Σ. The ranking of a label determines the number of attached vertices: $\forall e \in E\ rank(label(e)) = |att(e)|$. The external vertices $ext \in V^*$ is a sequence of distinct vertices.

We also write V_H, E_H, att_E, $label_H$ and ext_H to refer to the components of the hypergraph H.

$\mathcal{H}_S$ is the set of all hypergraphs over the symbols $S \subseteq \Sigma$.

Example *The hypergraph $H = (V, E, att, label, ext)$ from Figure 2.1 over the alphabet $\Sigma = \{\mathbf{n}, \mathbf{x}\}$ with $rank(\mathbf{n}) = 2$ and $rank(\mathbf{x}) = 1$ is defined as:*

- $V = \{a, b, c\}$
- $E = \{n_1, n_2, n_3, x'\}$
- $ext = ac$
- $att(n_1) = ab \quad att(n_2) = bc$
  $att(n_3) = ca \quad att(x') = b$
- $label(n_1) = \mathbf{n} \quad label(n_2) = \mathbf{n}$
  $label(n_3) = \mathbf{n} \quad label(x') = \mathbf{x}$

We have especially that $rank(\mathbf{n}) = 2 = att(n_1) = att(n_2) = att(n_3)$ and $rank(\mathbf{x}) = 1 = att(x')$.

We draw hypergraphs graphical by representing vertices as circles and hyperedges as rectangles. External vertices have their position in the sequence of external vertices written in the circle and hyperedges have their label written inside their rectangle. We draw lines between a hyperedge and a vertex if the vertex is attached to the edge. Besides the line, we write the position of the vertex in the edges sequence. If the hyperedge only connects one vertex, we may omit the position. Sometimes (to be precise for so-called *selector* edges [Jan17, p.27ff.]) we decode the numbering in the form of a directed arrow if the hyperedge has only two attached vertices. The side without tip represents the number 1, and the side with tip represents the number 2. If we want to argue about specific vertices or hyperedges, we may denote the name n besides their label in case of hyperedges or their external number (if it has any) for vertices as $n : l$.

If we want to argue about a single hyperedge e, it is necessary to transform the single hyperedge into a hypergraph. This new hypergraph should only consist of the minimum of structure to have the hyperedge e. Minimal structur means that it has only the vertices that are attached to the hyperedge $[att_H(e)]$ (see Appendix A.7 for notation) and only the hyperedge e. We call this hypergraph the *handle of* e. Similarly, the handle of a symbol A is a hyperedge consisting of only one hyperedge with label A and exactly the attached vertices

that match the rank of A. This way the handle always has only the minimal structure for a hyperedge with this label.

Definition 2.2 (Handle [Hab92, p.11][Jan17, p.41])
Let $H \in \mathcal{H}_\Sigma$ be a hypergraph. The handle of a hyperedge $e \in E_H$ is the hypergraph $e^\bullet$ with:

- $V_{e^\bullet} = [att_H(e)]$
- $att_{e^\bullet}(e) = att_H(e)$
- $E_{e^\bullet} = \{e\}$
- $ext_{e^\bullet} = att_H(e)$
- $label_{e^\bullet}(e) = label_H(e)$

The handle of a symbol $A \in \Sigma$ with $rank(A) = m$ is the hypergraph $A^\bullet$ with:

- $V_{A^\bullet} = \{v_1, \dots v_m\}$
- $att_{A^\bullet}(e) = v_1 \dots v_m$
- $E_{A^\bullet} = \{e\}$
- $ext_{A^\bullet} = v_1 \dots v_m$
- $label_{A^\bullet}(e) = A$

The handle $e^\bullet$ of a hyperedge $e \in E_H$ is a special case of a subhypergraph. Subhypergraphs are hypergraphs that consist of a subset of hyperedges and vertices of some other hypergraph.

Definition 2.3 (Subhypergraphs [Hab92, p.12])
Let H and H' by two hypergraphs over a common finite ranked alphabet Σ. Then H' is a subhypergraph of H, denoted by $H' \subseteq H$, if and only if $V_{H'} \subseteq V_H$, $E_{H'} \subseteq E_H$ and for all $e \in E_{H'}$ $att_{H'}(e) = att_H(e)$ and $label_{H'}(e) = label_H(e)$. The components ext_H and $ext_{H'}$ may be arbitrary.

Example *Again consider the hypergraph $H = (V, E, att, label, ext)$ from Figure 2.1 over the alphabet $\Sigma = \{\mathbf{n}, \mathbf{x}\}$ with $rank(\mathbf{n}) = 2$ and $rank(\mathbf{x}) = 1$. The handle $n_1^\bullet$ is the subhypergraph of H defined as:*

- $V_{n_1^\bullet} = \{a, b\}$
- $att_{n_1^\bullet}(n_1) = ab$
- $E_{n_1^\bullet} = \{n_1\}$
- $ext_{n_1^\bullet} = ab$
- $label_{n_1^\bullet}(n_1) = \mathbf{n}$

The handle $\mathbf{n}^\bullet$ is defined as:

- $V_{\mathbf{n}^\bullet} = \{v_1, v_2\}$
- $att_{\mathbf{n}^\bullet}(e) = v_1 v_2$
- $E_{\mathbf{n}^\bullet} = \{e\}$
- $ext_{\mathbf{n}^\bullet} = v_1 v_2$
- $label_{\mathbf{n}^\bullet}(e) = \mathbf{n}$

We partition the alphabet into *terminal* symbols T and *nonterminal* symbols N. To analyse pointer programs, we would split terminal symbols further into

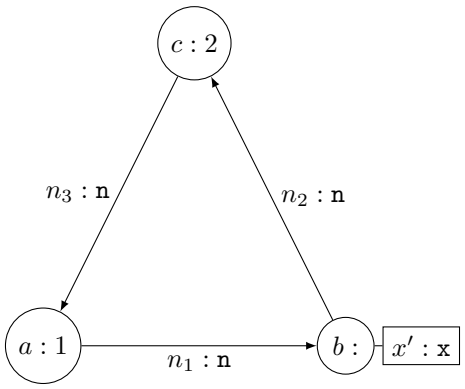


Figure 2.1: A (concrete) hypergraph consisting of three vertices a, b and c connected in a circle by edges n_1, n_2 and n_3 labelled $\mathtt{n}$ and an edge x' for the variable $\mathtt{x}$ pointing to the vertex b.

selectors *Sel* and variables *PVar*. Selectors *Sel* are the selectors of objects in a pointer program. List objects would, for example, have the selector **next**. The variables *PVar* are the pointers present in the program as variables. In this sense, selectors are only allowed to have two attached vertices, which are the object which owns this selector and the object that is the target of this selector. Variables are only allowed to have one attached vertex, which is the object that the variable points to. We will not need the partitioning into selectors and variables, but interpreting the hypergraphs in by thinking of selectors and variables gives a better intuition, and we use this intuition in the following examples. Terminal and nonterminal symbols, on the other hand, are used to categorise hypergraphs into *concrete* (consisting only of hyperedges with terminal labels) and *abstract* (consisting of at least one hyperedge with nonterminal label) hypergraphs.

Definition 2.4 (Concrete/Abstract Hypergraph [Jan17, p.28]) *For an alphabet $\Sigma = T \uplus N$, we call $H = (V, E, att, label, ext)$*

- *concrete if $label(E) \cap N = \emptyset$ and*
- *abstract otherwise.*

We call $E_N = \{e \in E \mid label(e) \in N\}$ the set of nonterminal hyperedges and $E_T = \{e \in E \mid label(e) \in T\}$ the set of terminal hyperedges.

Abstract hypergraphs have hyperedges with nonterminal labels. Such nonterminal labels will later be used to derive a language of graphs with only terminal hyperedges.

Example *The hypergraph in Figure 2.1 over the alphabet $\Sigma = T \uplus N$, $T = \{\mathtt{n}, \mathtt{x}\}$ and $N = \{L\}$, is concrete.The hypergraph in Figure 2.2 over the alphabet Σ is abstract and defined by*

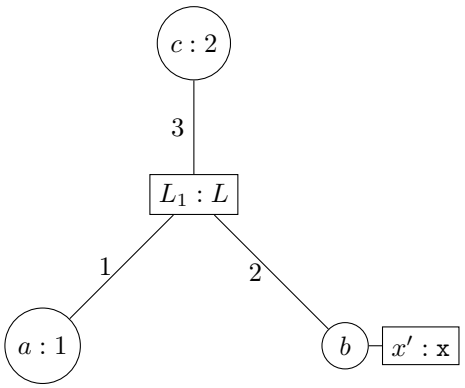


Figure 2.2: An (abstract) hypergraph that we yield by abstraction the graph from Figure 2.1 by replacing the circle with a hyperedge labelled L.

- $V = \{a, b, c\}$
- $E = \{L_1, x'\}$
- $att(e') = b, att(L_1) = abc$
- $label(L_1) = L, label(x') = \mathtt{x}$
- $ext = ac$

We do not distinguish between identical graphs up to the names of their vertices and hyperedges. Therefore we use an isomorphism up to renaming. We consider two hypergraphs equal if their vertices and edges can be renamed in order to yield the other hypergraph. Renaming does, however, not mean, that labels can be different. The labelling of hyperedges still has to be the same.

Definition 2.5 (Isomorphic Hypergraphs [Hab92, p.12])
Two hypergraphs H and H' over a common finite ranked alphabet Σ are considered to be isomorphic if and only if there exist two bijective functions $h_V \in V_H \mapsto V_{H'}$ and $h_E \in E_H \mapsto E_{H'}$ with

$$\forall e \in E_H \ \forall n \in [1, |att_H(e)|] \ h_V(att_H(e)(n)) = att_{H'}(h_E(e))(n),$$
$$\forall e \in E_H \ label_H(e) = label_{H'}(h_E(e)) \text{ and}$$
$$\forall n \in [1, |ext_H|] \ h_V(ext_H(n)) = ext_{H'}(n).$$

Isomorphism Warning

We will from now on consider isomorphic hypergraphs as equal and do not denote the isomorphism h_V and h_E explicitly. Moreover, if two hypergraphs are isomorphic by h_V and h_E, we will write the elements of the hypergraph as if they were equal.

2.2 Concrete Hypergraph Transition System

Transition systems are used to generalise different *models of computation*. Many computations can be transformed into a transition system by their *structural operational semantics* (or short SOS) [Plo04]. We will use such transition systems to describe arbitrary pointer programs. In general, the transition systems are infinite, but we will later introduce *abstract transition systems* that we assume finite, although its corresponding concrete transition system is infinite.

A transition system consists of a set of *states* States, a *transition relation* $\rightarrow$ and a set of *initial states* $Init$. In the literature sometimes a set of atomic propositions and a labelling of states to propositions are also added [BK08, p.20]. As we do not need these in this thesis, we will omit them. The states of the transition system are connected. We call such a connection from state s to s' a transition from s to s'. As we analyse pointer programs, we consider the states of the program as tuples of a location, which defines the current position in the program, and the memory configuration, which we represent as a hypergraph.

Definition 2.6 (Hypergraph Transition System)
We define a hypergraph transition system $TS = (States, Act, \rightarrow, Init)$ *over a set of locations* Loc *and an alphabet* Σ *as*

- $States \subseteq Loc \times \mathcal{H}_\Sigma$,
- $\rightarrow \subseteq States \times Act \times States$ *and*
- $Init \subseteq States$.

We now also introduce *concrete hypergraph transition systems* similar to concrete hypergraphs as in Definition 2.4. If a transition system only consists of concrete hypergraphs, the transition system is concrete. This Definition does not translate to abstract hypergraph transition systems, as they do not need to have at least one abstract hypergraph. Indeed, every concrete hypergraph transition system is an abstract transition system of itself.

Definition 2.7 (Concrete Hypergraph Transition System)
A hypergraph transition system $TS_c = (States_c, Act_c, \rightarrow_c, Init_c)$ *over a set of locations* Loc *and an alphabet* $\Sigma = T \uplus N$ *is concrete if and only if*

$$States_c \subseteq Loc \times \mathcal{H}_T.$$

We write $Stats_c$, Act_c, $\rightarrow_c$ *and* $Init_c$ *to refer to the components of the transition system* TS_c.

A *path* on a transition system is a sequence of states, where every successor is connected by a transition. A path may be finite or infinite. If a path is infinite, the corresponding computation does not *terminate*. The goal of this thesis is to show a technique to prove that no such infinite path exists in a concrete hypergraph transition system by using an abstract hypergraph transition system. We will define abstract hypergraph transition systems later.

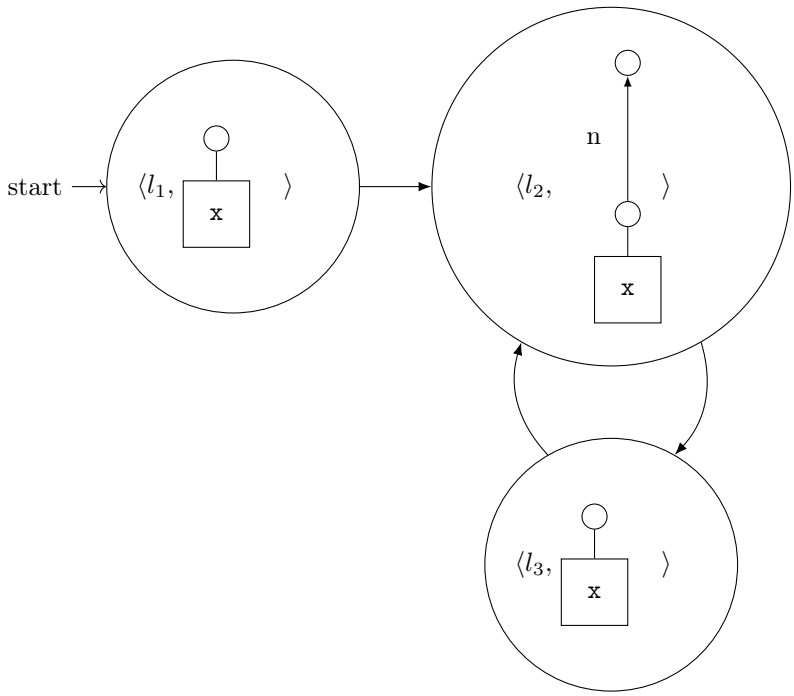


Figure 2.3: A hypergraph transition system consisting of three states and a unique location for each state. The states with location l_2 and l_3 form a circle and the initial state with location l_1 has a transition to this circle.

Definition 2.8 (Path)
The finite or infinite sequence $\pi \in States^* \cup States^\omega$ (see Appendix A.7) is a path in $TS = (States, Act, \to, Init)$ if and only if

$$\forall m \in [2, |\pi|] \; \exists \tau \in Act \; \pi(m-1) \xrightarrow{\tau} \pi(m).$$

We also write $c \to^\pi c'$ if and only if π is a path with $\pi(1) = c$ and $\pi(|\pi|) = c'$.

Example *We consider $TS = (States, Act, \to_c, Init)$ over $Loc = \{l_1, l_2, l_3\}$, $Act = \{\tau\}$ and $\Sigma = T \uplus N$ with $T = \{n\}$ and $N = \emptyset$ as in Figure 2.3, where*

- $States = \{(l_1, H_1), (l_2, H_2), (l_3, H_3)\}$,
- $(l_1, H_1) \xrightarrow{\tau}_c (l_2, H_2)$, $(l_2, H_2) \xrightarrow{\tau}_c (l_3, H_3)$, $(l_3, H_3) \xrightarrow{\tau}_c (l_2, H_2)$ *and*
- $Init = \{(l_1, H_1)\}$.

TS is a concrete hypergraph transition system as every hypergraph is concrete. The sequence $(l_1, H_1), (l_2, H_2), (l_3, H_3), (l_2, H_2) = \pi$ is a path, and we write the path π also as $(l_1, H_1) \to_c^\pi (l_2, H_2)$.

In the next step, we take a closer look at the locations *Loc*. Locations are used to connect the states to the corresponding program. Often some locations are similar to other: every location with an if-statement has a conditional branch, every location with a while-statement has a loop. The type of locations can give meaning to the state. In some programming languages, there are only specific types of locations that can be used to have non-terminating programs, i.e. locations with loop-statements. We call locations that have the potential to lead to a non-terminating program *critical*. For example, in a while program, we would consider all locations that have the while statement critical.

We define critical locations however not on a syntactic base (as we do not introduce a formal syntax for pointer programs), but on the semantics. If the transition system of the program (i.e. its semantics) has an infinite run π, then there exists a location that appears infinitely often on π. The set of critical locations is such that *every* infinite run has at least one infinitely often occurring location that is critical. Every set, including the empty set, is then a set of critical location for every transition system with only finite paths. For arbitrary programs, the minimal set of critical locations is not decidable; thus, we may use a non-minimal set of critical locations. Then we want to prove that every location in the set of critical locations does not occur infinitely often in any infinite path. If in the end, no critical location remains, the transition system only has finite paths. We do this by using an abstract transition system and proving that the states corresponding to the locations do not correspond to infinitely many states on any infinite path in the concrete transition system.

Definition 2.9 (Critical Locations)
Let $TS = (States, \rightarrow, Act, Init)$ be a transition system. A set of critical locations $Crit \subseteq Loc$ ensures that for all infinite paths $\pi \in States^{\omega}$

$$\forall m \in \mathbb{N}\ \exists m' \in \mathbb{N} : \quad m < m' \wedge \exists \langle l, H\rangle \in Crit \times \mathcal{H}\ \pi(m') = \langle l, H\rangle.$$

The set of critical states is

$$States_{Crit} = \{(l, H) \in States \mid l \in Crit\}.$$

Example *We consider the transition system from Figure 2.3. We assume this transition system was generated by the program code P of some while language. A connection between the locations and the program code could look like this:*

- $P(l_1) =$ "$\mathtt{x.n} =$ **new();**"
- $P(l_2) =$ "**foreach(**$\mathtt{x} = \mathtt{x.n}$**)**"
- $P(l_3) =$ "$\mathtt{x.n} =$ **new();**"

Although we did not introduce a formal syntax or even semantics of this while language, we can try to understand what this program is supposed to do. l_1 creates a new object that is the target of the selector $\mathtt{x.n}$. Afterwards, a loop is entered that iterates over the selector $\mathtt{x.n}$ and assigns the object to $\mathtt{x}$. Then the old object that is not reachable any longer is removed (e.g. by a garbage collector). Inside the loop again a new object is created as the target of the selector $\mathtt{n}$. However, any infinite run has to enter the statement **foreach** *infinitely often. Many programming languages have specific keywords that have to be visited infinitely often, to allow a non-terminating run, e.g. while or jump. If we can find these keywords, we consider the resulting set of locations having these keywords the set of critical locations. In this example, the set of critical locations would be $Loc = \{l_2\}$.* **foreach** *is a statement that makes its locations critical. We could, however, choose an even bigger set of critical locations, as every superset of Loc is still a set of critical locations.*

2.3 Hyperedge-Replacement Grammars

In this section, we focus on how we define a particular type of graph grammars, which we can use to formalise languages of hypergraphs. We use a type of replacement systems called *hyperedge-replacement grammars*. These systems work similarly to context-free string grammars [Hab92, p.45ff.]. In context-free string grammars, we have nonterminal and terminal symbols. By using productions, we can replace a nonterminal symbol by a string of symbols. Every possible way of repeatedly replacing nonterminal symbols generates a set of terminal strings, called a language. Hyperedge-replacement grammars work similar in the sense that they nonterminal and terminal symbols, but now the productions consists of a nonterminal symbol and a hypergraph. To derive a hypergraph from a hyperedge-replacement grammar, we use *productions* on a hyperedge. We replace a hyperedge by a production's hypergraph if the hyperedge's label is matching the production's nonterminal. The production rules are defined such that only nonterminal hyperedges are replaceable if the rank of the hypergraph matches the rank of the symbol. If only terminal hyperedges remain, the hypergraph is of the *language* described by the hyperedge-replacement grammar. In literature hyperedge-replacement grammars have, similar to context-free grammars, an *axiom*, that is a hypergraph from which we start with every derivation. We will, however, skip the axiom as we do not need it. We can use hyperedge-replacement grammars to describe an (infinite) language of graphs by a single hypergraph and a hyperedge-replacement grammar. While the grammar is generally fixed, we use different hypergraphs to abstract different sets of the hypergraphs of a set of states.

Definition 2.10 (Hyperedge-Replacement Grammar [Hab92, p.18]) *A hyperedge-replacement grammar (abbreviated as HRG) over the finite ranked alphabet $\Sigma = N \uplus T$ is a system $G = (N, T, P)$ where $P \subseteq N \times \mathcal{H}_\Sigma$ is a finite set of productions over N. For every production rule $(A, H) \in P$ we have $|ext_H| = rank(A)$.*

We notate the following sets of hyperedge-replacement grammars:

- *HRG is the set of all hyperedge-replacement grammars.*
- *HRG_Σ is the set of all hyperedge-replacement grammars over an alphabet Σ.*

Productions of these grammars use the *hyperedge-replacement* operation. In a hyperedge-replacement a hyperedge e from the hypergraph H is replaced by some hypergraph R. The replacement is done by removing the hyperedge e in H and removing the external vertices in R. Then the attachments to R's external vertices are redirected to the vertices that were attached to e in the same order of the external vertices. If a vertex is attached to e, it will be attached after the replacement to all hyperedges there we attached to the external vertex. The sequence of vertices of all attached hyperedges remains the same (by replacing the vertices in the sequence accordingly, if the vertex was an external vertex).

Definition 2.11 (Hyperedge-Replacement [Jan17, p.39])
Let $H, R \in \mathcal{H}_\Sigma$ be two hypergraphs. We assume that $V_H \cap V_R = \emptyset$. If the vertices are not disjoint, we rename vertices in R to make them disjoint. We call $H[\xi/R]$ the replacement of ξ by R in H. We define $H[\xi/R]$ as:

- $V_{H[\xi/R]} = V_H \cup (V_R \setminus [ext_R])$
- $E_{H[\xi/R]} = (E_H \setminus \{\epsilon\}) \uplus E_R$
- $att_{H[\xi/R]}(e)(m) = \begin{cases} att_H(e)(m) & \text{if } e \in E_H \\ att_R(e)(m) & \text{if } e \in E_R \wedge \\ & \quad att_R(e')(m) \notin [ext_R] \\ att_H(\epsilon)(m') & \text{if } e \in E_R \wedge \\ & \quad att_R(e)(m) = ext_R(m') \end{cases}$
- $label_{H[\xi/R]}(e) = \begin{cases} label_H(e) & \text{if } e \in E_H \\ label_R(e) & \text{if } e \in E_R \end{cases}$
- $ext_{H[\xi/R]} = ext_H$

Example *We have the three hypergraphs $H_1, H_2, H_3 \in \mathcal{H}_\Sigma$ depicted in Figure 2.4. Let ξ be the hyperedge with label A in the hypergraph H_1. We replace the hyperedge ξ by the hypergraph H_2. The replacement yields the hypergraph H_3. We notate the replacement of ξ by H_2 as*

$$H_1[\xi/H_2] = H_3.$$

In the replacement, we reattach the external nodes' attachments to ξ's vertices while conserving their order attachment. The non-external vertex in H_2 is thus attached via the hyperedges with label C to the external vertices of H_1. Additionally, a hyperedge with label A is attached to all external vertices.

A derivation is the multiple application of hyperedge-replacements. The productions of a grammar describe how such hyperedge-replacements are allowed to be used. We can replace a hyperedge ξ with a production p by the hypergraph of p if ξ's label matches the nonterminal symbol of p. Afterwards, we can combine these replacements to derivations of arbitrary size. As we do not distinguish between isomorphic hypergraphs, the resulting hypergraph of a derivation may be renamed, but still is isomorphic to the replacement.

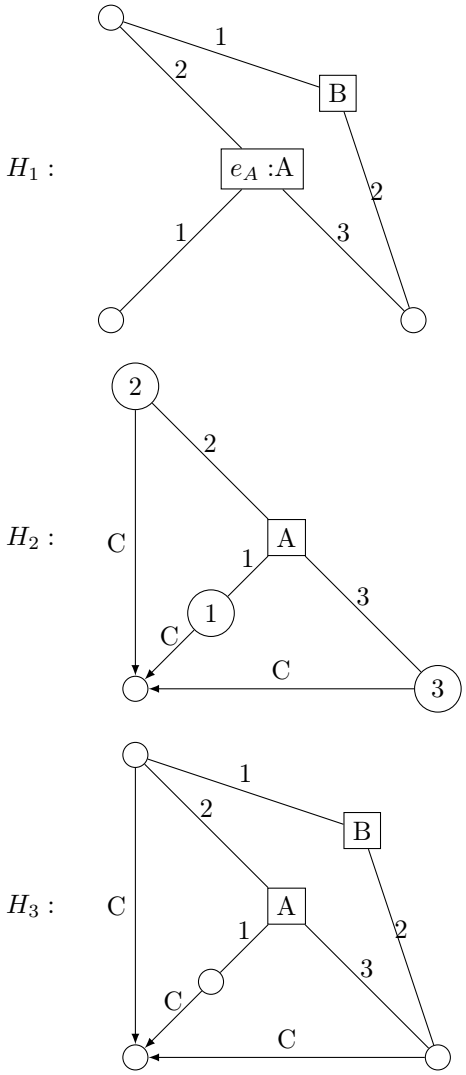


Figure 2.4: Two hypergraphs H_1 and H_2 which we use to generate H_3 by replacing the hyperedge e_A in H_1 by the hypergraph H_2.

Definition 2.12 (Direct Derivation [Hab92, p.17-18])
Let $H, H' \in \mathcal{H}_\Sigma$ be two hypergraphs, $G = (N, T, P)$ a hyperedge-replacement grammar, $p = (A, R) \in P$ be a production and $\epsilon \in E_H$ a nonterminal hyperedge with $label_H(\epsilon) = A$.

H directly (forward) derives H' by p (applied to ξ), written as $H \Rightarrow_{(\xi,p)} H'$, if and only if $H' = H[\xi/R]$. We also have that

$$H \Rightarrow_p H' \text{ iff } \exists \xi \in E_H \; H \Rightarrow_{(\xi,p)} H',$$
$$H \Rightarrow H' \text{ iff } \exists p \in P \; H \Rightarrow_p H'.$$

H directly backwards derives H' by p (applied to ξ), written as $H \Leftarrow_{(\xi,p)} H'$, if and only if H' directly forward derives H by p (applied to ξ). We also have that

$$H \Leftarrow_p H' \text{ iff } \exists \xi \in E_H \; H \Leftarrow_{(\xi,p)} H',$$
$$H \Leftarrow H' \text{ iff } \exists p \in P \; H \Leftarrow_p H'.$$

If we use a sequence of direct derivations, we call the sequence a derivation (of some size). Similar a sequence of backward direct derivations is called a backward derivation (of some size). Usually, we are not interested in the size of the derivation. Instead, often we will prove a property on direct derivations and then conclude that the property also holds on a sequence of direct derivations.

Definition 2.13 (Derivations [Hab92, p.17-18])
A sequence of direct derivations $H_0 \Rightarrow_{(\xi_1,p_1)} \ldots \Rightarrow_{(\xi_m,p_m)} H_m$ is called a (forward) derivation of length m from H_0 to H_m, and we write the derivation as $H_0 \Rightarrow_{(\xi_1,p_1)\ldots(\epsilon_m,p_m)} H_m$. We also have that

$$H_0 \Rightarrow^* H_m \text{ iff } \exists (\xi_1, p_1) \ldots (\xi_m, p_m) \; H_0 \Rightarrow_{(\xi_1,p_1)\ldots(\xi_m,p_m)} H_m.$$

We similarly denote the backward derivation $H_0 \Leftarrow_{(\xi_1,p_1)} \ldots \Leftarrow_{(\xi_m,p_m)} H_m$ as $H_0 \Leftarrow_{(\xi_1,p_1)\ldots(\xi_m,p_m)} H_m$ and have that

$$H_0 \Leftarrow^* H_m \text{ iff } \exists (\xi_1, p_1) \ldots (\xi_m, p_m) \; H_0 \Leftarrow_{(\xi_1,p_1)\ldots(\xi_m,p_m)} H_m.$$

Remark that we notate the production rules $p = (A, R)$ as a number in brackets. Considering the grammar from Example 2.1, we have the productions (1) and (2) and we write for a derivation $H \Rightarrow_{(1)} H'$ if and only if there exists $\xi \in E_H$ such that H derives H' by (1) applied to ξ.

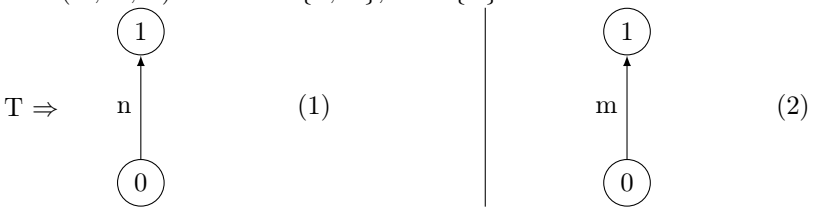
We define the language of a grammar as the application of derivations on the axiom. As we do not have axioms, we instead define the language of a hypergraph and a grammar. As soon as the result of derivations constructs a hypergraph only consisting of terminal hyperedges, the hypergraph is part of the grammar's language.

Definition 2.14 (Hyperedge-Replacement Language [Hab92, p.18]) *The (hyperedge-replacement) language $L_G(H)$ of an hyperedge-replacement grammar $G = (N, T, P)$ and a hypergraph $Z \in \mathcal{H}_\Sigma$ consists of all terminal labelled hypergraphs which can be derived from Z by applying productions of P:*

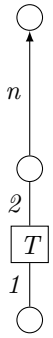
$$L_G(Z) = \{H \in \mathcal{H}_T \mid Z \Rightarrow^* H\}$$

Example 2.1: A grammar that generates a finite language of two hypergraphs and each having only one hyperedge.

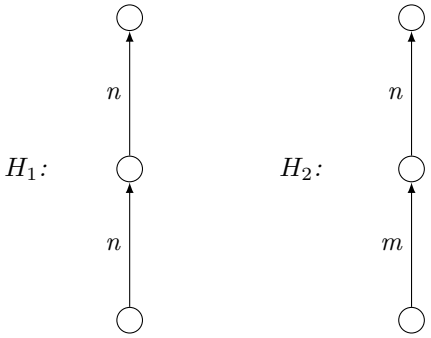
$G = (T, N, P)$ with $T = \{n, m\}$, $N = \{T\}$ and P:



Example *We consider the hyperedge-replacement grammar G from Example 2.1. We define Z as the following hypergraph:*



Then the language $L_G(Z)$ is the set of hypergraphs $\{H_1, H_2\}$ where $Z \Rightarrow_{(1)} H_1$ and $Z \Rightarrow_{(2)} H_2$ and thus yields the hypergraphs:



Sometimes we want to decompose a derivation into the subhypergraphs that reflects the replacement of the hyperedges from the original hypergraph H into

the resulting hypergraph H'. We want to decompose H' such that every subhypergraph represents one hyperedge of H and is the result of the replacement of this hyperedge in the derivations. If we replace a hyperedge e by some hypergraph R, this replacing hypergraph R is the subhypergraph of H' for the hyperedge e. This subhypergraph is called the joint-embedding of e into H' [Hab92, p.45ff.].

Definition 2.15 (Joint-Embedding [Hab92, p.47])
Let $H, H' \in \mathcal{H}_\Sigma$ be two hypergraphs with $E_H = \{e_1, \dots e_m\}$ and

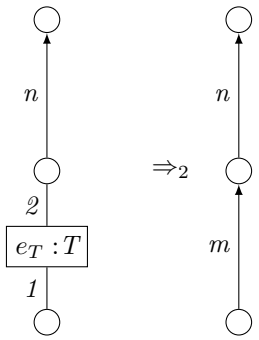
$$H \Rightarrow^* H',$$
$$H[e_1/R_{e_1} \dots e_m/R_{e_m}] = H'.$$

For a hyperedge $e \in E_H$ the subhypergraph $H'(e) \subseteq H'$ such that

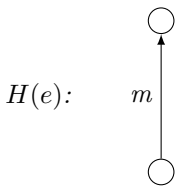
$$H'(e) = R_e$$

is the joint-embedding of e in H'.

Example *We consider the derivation $Z \Rightarrow H$ as following:*



The embedding $H(e_T)$ is the following subhypergraph from H:



2.4 Abstract Hypergraph Transition System

We will now introduce an abstraction for the hypergraph transition systems, namely the *abstract hypergraph transition systems* and the *abstract hyperedge-replacement grammar transition systems*. These are used to abstract a concrete hypergraph transition system into a smaller transition system, while many essential properties are preserved. Notably, it is used to abstract an infinite transition system into a finite transition system. Unfortunately, some properties are also lost. For example, we may lose the information if a state is visited only finitely often.

Our goal is thus first to compute a finite abstract transition system and afterwards use this transition system to conclude that also its concrete transition system has to be finite. Every state in an abstract hypergraph transition system consists of a location and a hypergraph. For a concrete state $\langle l, H\rangle$ that is abstracted by an abstract state $\langle l, H_a\rangle$, the hypergraph H needs to be in the language of the abstract state's hypergraph: $H \in L_G(H_a)$. Additionally, the locations of abstract states and its concrete states have to be the same. The function that maps the concrete states to the abstract states then induces for the abstract transition system the transitions and the set of the initial states. We call this function the *abstraction function* [BK08, p.499].

Definition 2.16 (Hypergraph Abstraction Function)
Let TS_c be a concrete hypergraph transition system and $States_a$ a set of (abstract) states. $\alpha \in States_c \mapsto States_a$ is a hypergraph abstraction function over $G \in HRG_\Sigma$ if and only if

$$\forall \langle l, H_c\rangle \in States_c : \quad \alpha(\langle l, H_c\rangle) = \langle l, H_a\rangle \wedge H_c \in L_G(H_a)$$

To define the transitions, we will use *inference rules*. These rules consist of a line, a *premise* on top of the line and a *conclusion* below the line. These rules can be interpreted as logical reasoning and can be used to define relations formally. If the premise on top holds, this rule concludes that the conclusion below the line holds, too. Infer rules are often used to define transitions, either in case of semantics or in the case of abstracting transitions [Rey98, p.12ff., p.126ff.]. We use the inference to say that if two concrete states c and c' have a transition, then also the abstract states $\alpha(c)$ and $\alpha(c')$ have an *abstract transition*. The abstract transition is indeed allowed to have even more transitions. We say that the abstract transitions are *overapproximating* the concrete transitions. We generate the initial states by applying the abstraction function α to all of the concrete initial states. This abstraction guarantees us that every path in the concrete transition system also has a corresponding path in the abstract transition system. It does, however, not guarantee that only paths in the concrete transition system have a corresponding path in the abstract transition system.

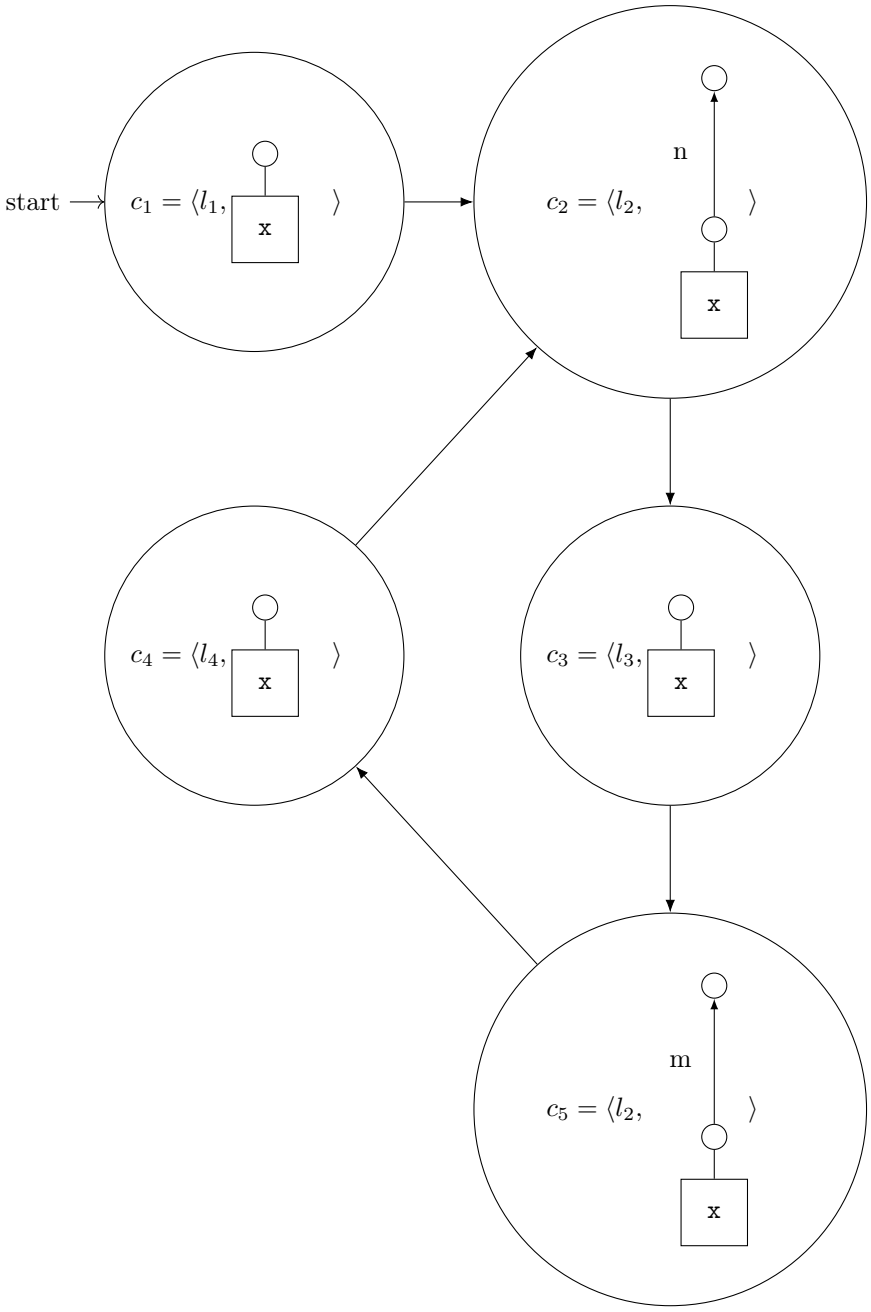


Figure 2.5: An extended version of the hypergraph transition system from Figure 2.3. We extend the circle by introducing two more states c_4 and c_5. The graphs of the states c_3 and c_4 are the same, but their locations are different. The locations of the states c_2 and c_5 are the same, but the graphs are different.

Definition 2.17 (Abstract Transition System [BK08, p.500]) *Let TS_c be a concrete hypergraph transition system. The abstract hypergraph transition system TS_a induced by the hypergraph abstraction function $\alpha \in States_c \mapsto States_a$ on TS is such that:*

- $$\frac{c \xrightarrow{\tau}_c c'}{\alpha(c) \xrightarrow{\tau'}_a \alpha(c')}$$
- $Init_a = \{\alpha(c) \mid c \in Init_c\}$

We write $States_a$, Act_a, $\rightarrow_a$ and $Init_a$ to refer to the components of the transition system TS_a.

Example *Let TS_c be the transition system from Figure 2.5 and TS_a be the transition system from Figure 2.6 over the alphabet $\Sigma = \{n, m, T\}$.*
$TS_c = (States_c, Act_c, \rightarrow_c, I_c)$ is a concrete hypergraph transition system (compare Definition 2.7). We define the hypergraph abstraction function α for G from Example 2.1 as

$$\begin{aligned} &\alpha(c_1) = a_1, && \alpha(c_2) = a_2, \\ &\alpha(c_3) = a_3, && \alpha(c_5) = a_2, \\ &\alpha(c_4) = a_4. \end{aligned}$$

First, we check that α is a hypergraph abstraction function. The only interesting states are c_2 and c_5. We map any other state to a state that has a one-to-one correspondence with its abstract state. We have, however, that

$$\alpha(\langle l_2, H\rangle) = \alpha(\langle l_2, H'\rangle) = \langle l_2, H_a\rangle.$$

The location l_2 is in all these states the same and $L_G(H_a) = \{H, H'\}$ is the language of H_a. Thus α is a hypergraph abstraction function. TS_a is thus an abstract hypergraph transition system of TS_c induced by α. Additionally TS_a describes a path that has no corresponding path in TS_c: a_1, a_2, a_4.

We abstract Pointer programs, and therefore also hypergraph transition systems, often via some form of *shape analysis*. Shape analysis aims to analyse pointer programs by using different abstract structures to encode the state of the memory. In order to transform the abstract shape of the memory, we sometimes may need to make it more concrete. We use *materialisation* operations to make the shape more concrete and *abstraction* operations to abstract the shape again afterwards. We will materialise a hypergraph by applying derivations on nonterminal hyperedges. The resulting hypergraph does not need to be entirely concrete, however. Abstraction operations are then used to reverse the materialisation again. We will use backward derivations on the hypergraph to receive a more abstracted hypergraph.

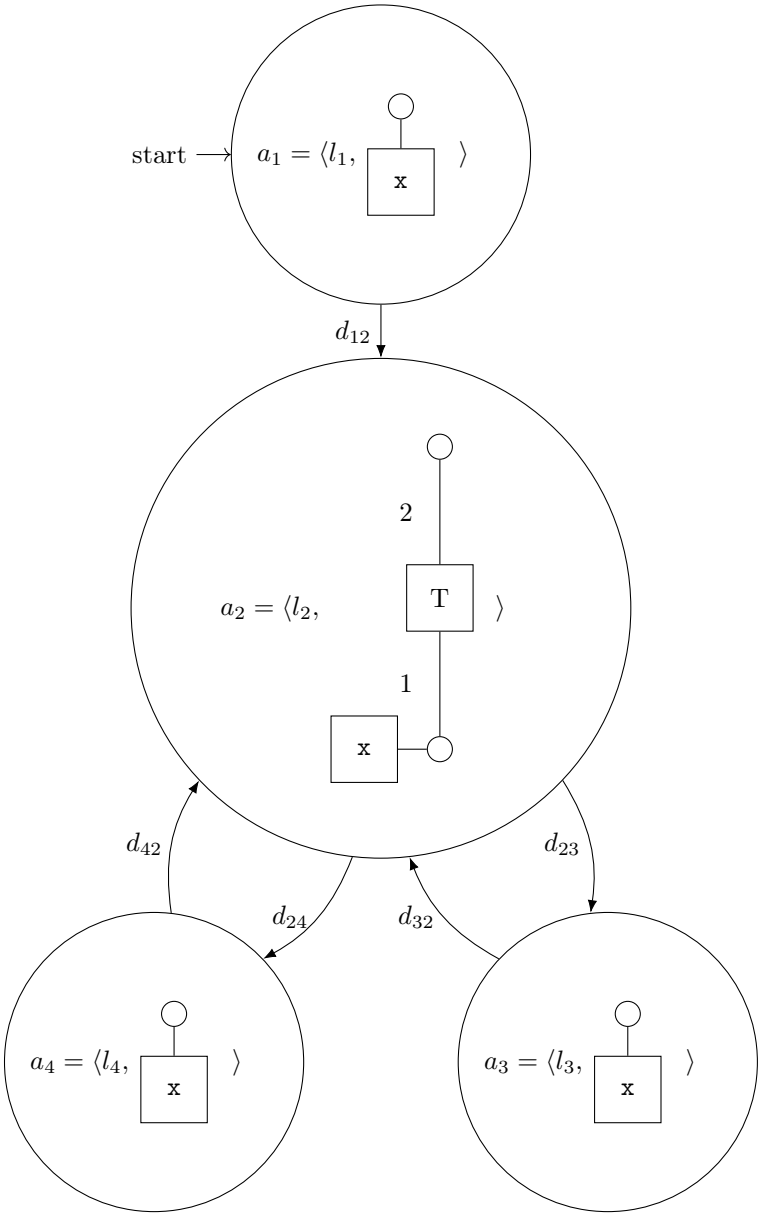


Figure 2.6: An abstract hypergraph transition system for the transition system from Figure 2.5 via the grammar from Example 2.1. The states c_2 and c_5 were combined as their locations are the same and their hypergraphs can be abstracted by the same hypergraph. All transitions have decompositions d as labels.

Definition 2.18 (Materialisation/Abstraction [Jan17, p.41ff.])
Let $G \in HRG_\Sigma$ be a hyperedge-replacement grammar and $H, H' \in \mathcal{H}_\Sigma$ two hypergraphs.

- *We call the forward derivation $H \Rightarrow^* H'$ a materialisation of H to H'.*
- *We call the backward derivation $H \Leftarrow^* H'$ an abstraction of H to H'.*

Between materialisation and abstraction operations,we apply a *concrete hypergraph operation* in order to simulate the concrete transition. For a hypergraph, this means that the operation is *commutable* over deriving the abstract hypergraph to a concrete hypergraph [HJKN15, p.177]. Thus it does not matter if we first apply the operation on the abstract hypergraph and then fully materialise the hypergraph or if we first fully materialise the hypergraph and then apply the operation. We will enforce this property by not allowing to change nonterminal hyperedges. Remark however that an operation may be commutable even if it changes nonterminal hyperedges.

Definition 2.19 (Concrete Hypergraph Operation)
A hypergraph operation $\mu \in \mathcal{H}_\Sigma \mapsto \mathcal{H}_\Sigma$ is concrete if and only if for all $H \in \mathcal{H}_\Sigma$ the subhypergraph H^N exists such that $H^N \subseteq H$, $H^N \subseteq \mu(H)$ and

$$E_{H^N} = \{e \in E_H \mid label_H(e) \in N\},$$
$$E_{H^N} = \{e \in E_{\mu(H)} \mid label_{\mu(H)}(e) \in N\}.$$

We denote $\mu(H) = H'$ also by $H \xrightarrow{\mu} H'$.

Materialisation, concrete hypergraph operations and abstraction can be used to abstract a concrete hypergraph transition systems that describes a pointer program [Jan17]. In such an abstract transition system every transition is composed of a materialisation, then applying a concrete hypergraph operation and lastly abstracting again. If we perform all these three steps correctly, the resulting transition system satisfies all conditions for an abstract hypergraph transition system from Definition 2.17. Such a transition system is called an *abstract hyperedge-replacement grammar transition system* because it uses the derivations of a hyperedge-replacement grammar to materialise and abstract the abstract hypergraph.

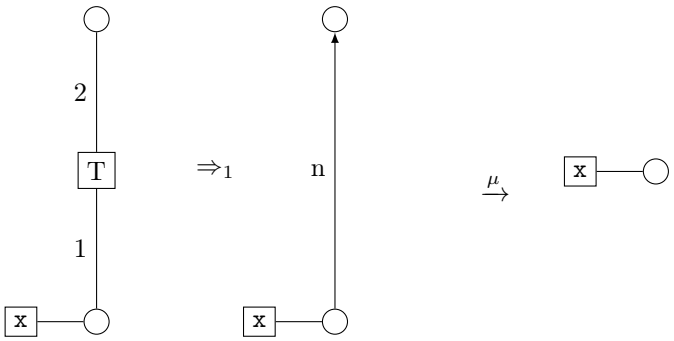


Figure 2.7: The decomposition d_{23} in the abstract transition system in Figure 2.6. The decomposition consists of a forward derivation and a concrete hypergraph operation.

Definition 2.20 (Abstract HRG Transition System)
Let $G \in HRG_\Sigma$ be a hyperedge-replacement grammar. We call the hyperedge transition system TS_a an abstract hyperedge-replacement grammar transition system if and only if

- *TS_a is an abstract hypergraph transition system induced by a hypergraph abstraction function α over G for TS_c*
- *and for all concrete transitions $\langle l, H_T\rangle \to_c \langle l', H'_T\rangle$ exists an abstract transition*

$$\alpha(\langle l, H_T\rangle) = \langle l, H\rangle \xrightarrow{H_m \xrightarrow{\mu} H'_m}_a \langle l, H'_T\rangle = \alpha(\langle l, H'_T\rangle)$$

such that μ is a concrete hypergraph operation and

$$H \Rightarrow^* H_m \xrightarrow{\mu} H'_m \Leftarrow^* H'$$
$$H_T \in L_G(H_m)$$
$$H'_T \in L_G(H'_m).$$

$H_m \xrightarrow{\mu} H'_m$ is called the decomposition of the abstract transition.

Example *We consider the transition systems TS_c from Figure 2.5 and TS_a from Figure 2.6 again. TS_a is an abstract hypergraph transition system of TS_c by some hypergraph abstraction function α for the grammar G in Example 2.1 as already shown before. TS_a is also an abstract hyperedge-replacement grammar transition system. As an example, we consider the transition $a_2 \xrightarrow{d_{23}}_a a_3$. The transition $a_2 \xrightarrow{d_{24}}_a a_4$ is, however, analogous. The decomposition d_{23} is visually shown in Figure 2.7. We derive the nonterminal hyperedge with label T to a terminal hyperedge with label n. The new hypergraph H_m is then concrete, and therefore the whole graph can be changed arbitrarily by μ. Here μ removes one node and the hyperedge with label n. An additional abstraction of the hypergraph is not needed.*

Paths on a concrete transition system always correspond to a path in its abstract transition system. We can generate these by applying the abstraction function on every state in the concrete path. We will introduce a notation to argue about both of these paths at the same time. This notation can be considered as annotating every transition with its abstract counterparts. For a path in the concrete transition system $c \to_c^* c'$ and its abstract counterpart $a \to_a^* a'$, we then argue that the path from c to c' corresponds to the path from a and a'.

Definition 2.21 (Abstract Path)
Let TS_a be an abstract HRG transition system for the concrete hypergraph transition system TS_c. Let TS_a be induced by the hypergraph abstraction function $\alpha \in States_c \mapsto States_a$ with the grammar G.
We write $c \to_c^{c_a, c'_a} c'$ if and only if

$$c = \langle l, H_T \rangle \to_c \langle l', H'_T \rangle = c',$$
$$\alpha(c) = a, \alpha(c') = a'.$$

We write for a path $\pi \in States_a$ $c \to_c^{\pi, a', a''} c'$ if and only if

$$c \to_c^{\pi, a'} c', c' \to_c^{a', a''} c''.$$

Example *Let TS_c be the transition system from Figure 2.5 and TS_a from Figure 2.6 the abstract hypergraph transition system of TS_c induced by an abstraction function α. The abstract path $c_2 \to_c^{a_2, a_3, a_2, a_4, a_2} c_2$ is an abstract path from c_1 to c_2. We can construct the path segments as follows:*

$$\left.\begin{array}{c} c_2 = \langle l, H_T \rangle \to_c \langle l', H'_T \rangle = c_3, \\ \alpha(c_2) = a_2 = \langle l, H \rangle \xrightarrow{H_m \xrightarrow{\mu_1} H'_m}_a \langle l', H'_T \rangle = a_3 = \alpha(c_3), \\ H_T \in L_G(H_m) \text{ and } H'_T \in L_G(H'_m) \end{array}\right\} \Rightarrow c_2 \to_c^{a_2, a_3} c_3$$

with $H \Rightarrow_{(1)} H_m \xrightarrow{\mu_1} H'_m = H'$.

$$\left.\begin{array}{c} c_3 = \langle l, H_T \rangle \to_c \langle l', H'_T \rangle = c_5, \\ \alpha(c_3) = a_3 = \langle l, H \rangle \xrightarrow{H_m \xrightarrow{\mu_2} H'_m}_a \langle l', H' \rangle = a_2 = \alpha(c_5), \\ H_T \in L_G(H_m) \text{ and } H'_T \in L_G(H'_m) \end{array}\right\} \Rightarrow c_3 \to_c^{a_3, a_2} c_5$$

with $H = H_m \xrightarrow{\mu_2} H'_m \Leftarrow_{(2)} H'$.

$$\left.\begin{array}{c} c_5 = \langle l, H_T \rangle \to_c \langle l', H'_T \rangle = c_4, \\ \alpha(c_5) = a_2 = \langle l, H \rangle \xrightarrow{H_m \xrightarrow{\mu_3} H'_m}_a \langle l', H' \rangle = a_4 = \alpha(c_4), \\ H_T \in L_G(H_m) \text{ and } H'_T \in L_G(H'_m) \end{array}\right\} c_5 \to_c^{a_2, a_4} c_4$$

with $H \Rightarrow_{(2)} H_m \xrightarrow{\mu_3} H'_m = H'$.

$$\left.\begin{array}{c} c_4 = \langle l, H_T \rangle \to_c \langle l', H'_T \rangle = c_2, \\ \alpha(c_4) = a_4 = \langle l, H \rangle \xrightarrow{H_m \xrightarrow{\mu_4} H'_m}_a \langle l', H' \rangle = a_2 = \alpha(c_2), \\ H_T \in L_G(H_m) \text{ and } H'_T \in L_G(H'_m) \end{array}\right\} c_4 \to_c^{a_4, a_2} c_2$$

with $H = H_m \xrightarrow{\mu_4} H'_m \Leftarrow_{(1)} H'$. *The path segments can then be combined to:*

$$c_2 \rightarrow_c^{a_2,a_3} c_3 \wedge c_3 \rightarrow_c^{a_3,a_2} c_5 \textit{ follows } c_2 \rightarrow_c^{a_2,a_3,a_2} c_5$$
$$c_2 \rightarrow_c^{a_2,a_3,a_2} c_5 \wedge c_5 \rightarrow_c^{a_2,a_4} c_4 \textit{ follows } c_2 \rightarrow_c^{a_2,a_3,a_2,a_4} c_4$$
$$c_2 \rightarrow_c^{a_2,a_3,a_2,a_4} c_4 \wedge c_4 \rightarrow_c^{a_4,a_2} c_2 \textit{ follows } c_2 \rightarrow_c^{a_2,a_3,a_2,a_4,a_2} c_2$$

Chapter 3

Predicates

To analyse pointer programs, we use an abstract hypergraph which represents a hyperedge-replacement language. In every transition of the abstract hypergraph transition system we materialise the abstract hypergraph if necessary (i.e. use production rules forward) and always try to abstract the hypergraph in the end (i.e. use production rules backwards). Additionally, we now want to filter the language of hypergraphs by a specific property. For example, we only want the hypergraphs that describe an object of a data structure with a given size or only those that are connected. For this, we use predicates, which are compatible with the rules of the grammar and express the desired property. We study a formalism which gives us such compatible predicates [Hab92, p. 128]. These predicates consist of indices. The indices are used to express a variant property. We furthermore use indices that describe a property on subhypergraphs to prove the index of the whole hypergraph. We could, for example, assign every (derivable) hypergraph a size and the indices would represent this size. The size of subhypergraphs could then be used to prove the size of the whole graph (more precise the joint-embeddings). If we want to restrict the hypergraphs to some property, we also need to know how these properties behave on materialisation steps, concrete hypergraph operations and abstraction steps. We will overapproximate the indices after materialisation and abstraction to have a sound restriction. We can, however, calculate index changes after concrete hypergraph operations precisely. In the next sections, we will see how we define these predicates describing feasible properties on hypergraphs.

3.1 Compatible Predicates

First, we introduce a property of predicates called *compatibility*. For a grammar G and a predicate $PROP'$, a (G,PROP')-compatible predicate $PROP$ can be proven by using a derivation of the grammar G and applying $PROP'$ to every direct derivation. Such predicates can be described over the rules of the grammar instead of the hypergraph itself. We use a function called *assign* which assigns every hyperedge in the hypergraph H an index. Indices encode some property that has to hold on the joint-embedding of every hyperedge $e \in E_H$ in $H' \in \mathcal{H}_T$.

Intuitively, compatible predicates $PROP$ are similar to nondeterministic tree automata on a derivation. We can express derivations of hyperedge-replacement

grammars as a derivation tree. Every node in the derivation tree represents a joint-embedding of a hyperedge in the parent node. The leaves consist of concrete hypergraphs, and by replacing them into their parent nodes recursively, the root yields the wholly derived hypergraph. Tree automata are automata that operate bottom-up on a tree [CDG+07]. Every leaf is assigned a state based on the type (symbol) of the leaf. The inner nodes are recursively assigned (by a transition relation) a state based on the type of the inner node and the states assigned to the children. For compatible predicates this work similarly. The index set can be understood as a set of states, a derivation can be understood as the input tree, and PROP' can be interpreted as a transition in the tree automata. The aissgn function and the predicate $PROP$ describe the properties (i.e. indices) and have to hold recursively on every derivation. To prove that for the index i and the hypergraph R that $PROP(i)(R)$ holds, we apply the assign function $assign$ and the hypergraph R to $PROP'$, similar to how we would calculate the state in the tree automata by applying the type of the current node (R) and the states of the children ($assign$) to the transition ($PROP'$).

Definition 3.1 (Compatible Predicate [Hab92, p. 128])
Let $G = (N, T, P, Z) \in HRG_\Sigma$ be a hyperedge-replacement grammar, I be a non-empty index set and

$$PROP \in I \mapsto (\mathcal{H}_T \mapsto \mathbb{B}),$$
$$PROP' \in I \mapsto ((E_R \mapsto I) \times \mathcal{H}_\Sigma \mapsto \mathbb{B})$$

be decidable predicates.

$PROP$ is called (G,PROP')-compatible *over I if and only if for all derivations $A^\bullet \Rightarrow R \Rightarrow^* H \in \mathcal{H}_T$ and all $i \in I$:*

$$PROP(i)(H) \Leftrightarrow \begin{cases} \exists assign \in E_R \mapsto I \; PROP'(i)(assign, R) \\ \quad \wedge \forall e \in E_R \; PROP(assign(e))(H(e)). \end{cases} \tag{3.1}$$

In the literature, compatible predicates $PROP$ are often also defined on a set of grammars C instead of only on one grammar. We can, however, extend our definition to sets. If the predicate $PROP$ is (G, $PROP'$)-compatible for all grammars $G \in C$, then $PROP$ is also (C, $PROP'$)-compatible [Hab92, p. 128]. We will, however, only use one fixed but arbitrary grammar in this thesis. Furthermore, we will later use a partial ordered set for an index set. In this case, for a partially ordered set $(I, \sqsubseteq)$, we will call a predicate $PROP$ (G,PROP')-compatible over $(I, \sqsubseteq)$, where I is the non-empty index set.

Compatible predicates map an index and a hypergraph to a boolean value. We will often argue that a specific hypergraph or the handle of a hyperedge is mapped to *true* by the predicate. To express this, we say that for a predicate with index $PROP(i)$, the hypergraph H satisfies this predicate or that the hyperedge e satisfies this predicate.

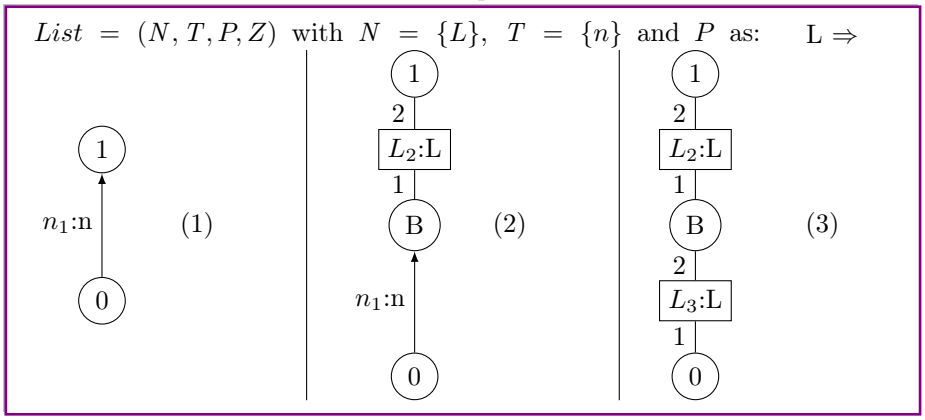
Definition 3.2 (Satisfying Hypergraphs and Edges)
We say a hypergraph H satisfy $PROP(i)$ if and only if $PROP(i)(H)$ holds. We say a hyperedge e satisfy $PROP(i)$ if and only if $PROP(i)(e^\bullet)$ holds.

Using compatible predicates allows us to filter graph languages defined by hyperedge-replacement grammars to only the hypergraphs H that satisfies the predicate for some i, i.e. $PROP(i)(H)$ holds. If the index set is finite, the new language is also a hyperedge-replacement language as shown by Habel [Hab92, p. 139]. For an infinite index set, this does not hold in general as then we can describe a property $SQUARE(H)$ that is satisfied if and only if $\exists n \in \mathbb{N} : |V_H| = n^2$. However, this property can be used to filter languages such that the filter is not a hyperedge-replacement language [Hab92, p. 142].

Definition 3.3 (Filter)
The filter of a language $L \subseteq \mathcal{H}_\Sigma$ to a predicate written as $L \cap PROP(i)$ is defined as

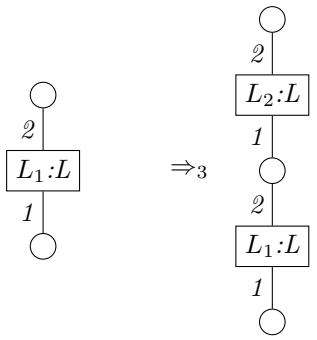
$$L \cap PROP(i) = \{H \in L \mid PROP(i)(H)\}. \tag{3.2}$$

Example 3.1: A grammar for list data structures. The grammar consists of one nonterminal L, one terminal$n$ and three production rules.



Example *We now construct a predicate called $LIST\text{-}SIZE(i)(H)$ over the index set $I = \mathbb{N}$. List is the grammar as defined in Example 3.1. For an index i and a hypergraph H, we say that $LIST\text{-}SIZE(i)(H)$ holds if and only if $|E_H| = i$. $LIST\text{-}SIZE$ does not require that H is a list, but List enforces that only lists can be derived. The list containing two nodes is the list of size 1 instead of 2. We use this size of lists because we assume the last node to be null and thus not a valid node of the list. Then $LIST\text{-}SIZE'(i)(assign, R)$ holds if and only if $\sum_{e \in E_R} assign(e) = i$.*

Every terminal hyperedge n only satisfies $LIST\text{-}SIZE(1)$. In every derivation for the nonterminal hyperedge L to the hypergraph R, the predicate then adds the number of terminal hyperedges derived in the joint-embeddings in R and the number of terminal hyperedges in the current derivation together to yield the current number of terminal edges. We consider the following derivation with the production 3:



We assume that

$$LIST\text{-}SIZE(4)(H(L_2)),$$
$$LIST\text{-}SIZE(3)(H(L_3)).$$

By assuming this, we restrict the possible hypergraphs that are derivable to only the list of at length 4 for L_2 and 3 for L_3. Then we have an assign mapping that maps L_2 and L_3 to their index:

$$assign(L_2) = 4,$$
$$assign(L_3) = 3.$$

This mapping is the only mapping such that $H(L_i)$ satisfies

$$LIST\text{-}SIZE(assign(L_i)).$$

By this assign, this R and that $4 + 3 = 7$ we have that

$$LIST\text{-}SIZE'(7)(assign, R).$$

We add the size, this is the only index and assign combination such that the predicate $LIST\text{-}SIZE'$ holds with R and such that $LIST\text{-}SIZE(assign(e))$ is satisfied by $H(L_2)$ and $H(L_3)$. This number of terminal hyperedges is the only index such that the predicate holds. Without prove, we also observe that the predicate $LIST\text{-}SIZE$ is (List,LIST-SIZE')-compatible.

3.2 Monotonic Directed Predicates

The analysis is based on the idea that we can filter languages of hypergraphs by predicates and indices. After the execution of a statement, we adjust the mapping from hyperedges to indices. For this adjustment, we will overapproximate the index in order to calculate an index such that all computations still have a corresponding path in the abstract, filtered transition system. In order to be able to overapproximate the index, we need an ordering on the index set. The order should intuitively describe how abstract a particular index is. The ordering has to be monotonic in regards to the subset order of the filtered hypergraph language. We also need to have an index for every possible occurring hypergraph in order to assign a start index to every possible hypergraph of a start state in the transition system. We later want to assign every nonterminal a variable with an index. To be able to do so, every hypergraph H has to have an index i such that H satisfies $PROP(i)$. This way, we can assign every hypergraph derived by a nonterminal an index and abstract these indices by variables.

Definition 3.4 (Monotonic Directed Predicates)
A (G,PROP')-compatible predicate PROP over $(I, \sqsubseteq)$ with $G \in HRG_\Sigma$ is monotonic directed *if and only if $(I, \sqsubseteq)$ is a complete lattice (see Appendix A.12) and, for all $A \in \Sigma$ and $i, j \in I$, we have*

$$i \sqsubseteq j \Rightarrow L_G(A^\bullet) \cap PROP(i) \subseteq L_G(A^\bullet) \cap PROP(j), \tag{3.3}$$

$$\forall H \in L_G(A^\bullet)\ \exists i \in I\ PROP(i)(H). \tag{3.4}$$

Example 3.2: Definition of *MOST-LIST-SIZE*. These are upper bounds of the list length for any hypergraph representing a list.

We define the predicate *MOST-LIST-SIZE* (or short *MLS*) over the index set $(\mathbb{Z} \cup \{\infty, -\infty\}, \leq)$. We have for the infinity values in the domain the order $\forall i \in \mathbb{Z}\ -\infty \leq i \leq \infty$. We use however only the positive integers. For the predicate we have

$$MLS(i)(H) \Leftrightarrow |E_H| \leq i,$$
$$MLS'(i)(assign, R) \Leftrightarrow \sum_{e \in E_R} assign(e) \leq i.$$

MLS is monotonic directed and (*List*,MLS')-compatible where *List* (see Example 3.1).

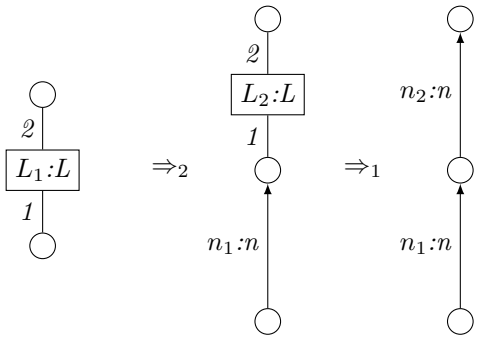
Example *The predicate LIST-SIZE with the index set $(\mathbb{N}, \leq)$ is not monotonic directed since $1 \leq 2$ but*

$$L_{List}(L) \cap PROP(1) \not\subseteq L_{List}(L) \cap PROP(2).$$

Instead, we use the predicate MOST-LIST-SIZE from in Example 3.2. MLS is (List,MLS')-compatible. This predicate satisfies both conditions for monotonic directed:

1. *for $i \leq j$ and $|E_H| \leq i$ we have that $|E_H| \leq i \leq j$*
2. *Every hypergraph has a finite number m of hyperedges; thus every hypergraph satisfies at least $MLS(m)$.*

We consider a derivation for a list given by the grammar from Example 3.1 of size 2 and prove that the list is at most of size 3.



We have:

- $MOST\text{-}LIST\text{-}SIZE(1)(H(n_1))$
- $MOST\text{-}LIST\text{-}SIZE(1)(H(n_2))$

by using $MOST\text{-}LIST\text{-}SIZE'$ to prove an index we get

- $MOST\text{-}LIST\text{-}SIZE(m)(H(L_2))$ *for every index* $m \geq 1$
- $MOST\text{-}LIST\text{-}SIZE(m')(H(L_1))$ *for every index* $m' \geq 2$
- *especially:* $MOST\text{-}LIST\text{-}SIZE(3)(H(L_1))$

If every hypergraph that satisfies $PROP(i)$ also satisfies $PROP(j)$ where $i \sqsubseteq j$, we can abstract i by j. Every hypergraph H has to have at least one index i such that H satisfies $PROP(i)$ and we can abstract indices by $\sqsubseteq$. Thus, we can take the least upper bound of $\sqsubseteq$ on all indices and receive the top element, which has to contain every hypergraph H. Thus $PROP(\top)$ is satisfied by every hypergraph $H \in \mathcal{H}_\Sigma$.

Lemma 3.1
Let $PROP$ be a monotonic directed, (G, $PROP$')-compatible predicate over the index set $(I, \sqsubseteq)$ with $G \in HRG_\Sigma$. Then for all $A \in \Sigma$, it holds that

$$L_G(A^\bullet) = L_G(A^\bullet) \cap PROP(\top). \tag{3.5}$$

Proof First, we prove that $L_G(A^\bullet) \cap PROP(\top) \subseteq L_G(A^\bullet)$

$$\begin{aligned} & L_G(A^\bullet) \cap PROP(\top) \\ = \; & \{H \in L_G(A^\bullet) \mid PROP(\top)(H)\} \\ \subseteq \; & \{H \in L_G(A^\bullet) \mid true\} \\ = \; & L_G(A^\bullet). \end{aligned}$$

■

Now we prove that $L_G(A^\bullet) \subseteq L_G(A^\bullet) \cap PROP(\top)$. Assume $H \in L_G(A^\bullet)$, then by condition (3.4) there exists $i \in I$ with $H \in L_G(A^\bullet) \cap PROP(i)$. Since $i \sqsubseteq \top$, we have by condition (3.3) $L_G(A^\bullet) \cap PROP(i) \subseteq L_G(A^\bullet) \cap PROP(\top)$, thus $H \in L_G(A^\bullet) \cap PROP(\top)$.

3.3 Terminating Predicates

Now we focus on the property of predicates than can be used to prove termination. We will use terminating predicates $PROP$ to show that for an abstract path π, there is no concrete path $c \to_c^\pi c'$ such that the hypergraph of c satisfies the initial conditions (i.e. some $PROP(i)$). For this, we filter the language via an index and a compatible predicate and change the index accordingly to abstraction and materialisation steps. If the index is descending infinitely often, we want to be able to show a contradiction, i.e. that there is *no* hypergraph satisfying $PROP(i)$ before execution and using the hypergraph as an input for the program results in this execution. We say that every infinitely descending chain σ reaches at some point an index t for which exists no hypergraph H that satisfies $PROP(t)$. We call t a terminating index and a language filtered to $PROP(t)$ should result in the empty language. If we conclude that a hypergraph has to satisfy $PROP(t)$, we can further conclude a contradiction.

Definition 3.5 (Terminating Predicate)
Let $PROP$ be a (G,$PROP'$)-compatible predicate over a partially ordered set $(I, \sqsubseteq)$ with $G \in HRG_\Sigma$. We call $PROP$ terminating *if and only if*

$$\begin{aligned} &\forall \sigma \in I^\omega \ \sigma \text{ is a strict descending chain} \\ &\Rightarrow \big(\exists m \in \mathbb{N}\ \forall A \in \Sigma\ L_G(A^\bullet) \cap PROP(\sigma(m)) = \emptyset. \end{aligned} \tag{3.6}$$

We call an index $i \in I$ terminating if and only if

$$\forall A \in \Sigma\ L_G(A^\bullet) \cap PROP(i) = \emptyset. \tag{3.7}$$

Example *$MOST\text{-}LIST\text{-}SIZE$ and $LIST\text{-}SIZE$ are both terminating with the index set $(\mathbb{Z} \cup \{\infty, -\infty\}, \leq)$. In both cases, 0 is a terminating index, but for MLS, $-\infty$ is the least terminating index. We have that*

1. *every strict descending chain σ has a negative number,*
2. *$\forall i \leq 0\ L_{List}(A^\bullet) \cap LIST\text{-}SIZE(i) = L_{List}(A^\bullet) \cap MLS(i) = \emptyset$,*
3. *$L_{List}(A^\bullet) \cap MLS(-\infty) = \emptyset$.*

If now we know that revisiting a specific state would yield a strict descending index after following a specific abstract path (or more precise loop), we know that taking this abstract path infinitely often generates a strict descending chain and thus we will reach a terminating index t. Nevertheless, by ending up in t, the language of the remaining graphs is empty, and thus the path is invalid. Hence, the path can only be taken finitely often, which proves the termination of this abstract path.

3.4 Relative Index

Up to now, we always suggested that we know which index we use. If there are infinitely many possible indices, we would have to test all of these. Since this is not feasible, we instead use *variables* to abstract every possible index and analyse the variable instead. We call the domain using variables *relative index set*. We use functions to map the variables to the *concrete indices* and call these *variable assignments*. Then we map a relative index via a variable assignment to a concrete index. We describe this mapping by the function $[\![\cdot]\!]_\nu$, where ν is a variable assignment. We represent the missing index with the dot, i.e. $[\![\cdot]\!]_\nu$ is a function from relative indices to concrete indices. This relative domain does not need to have all the same restrictions as the concrete domain but inherits some via a connection with the concrete domain. It does not need to be terminating, for example. Thus it can have infinitely descending chains. However, for an infinitely descending chain, the elements have to be mapped to elements that form a finitely descending chain. A variable assigning function can be restricted to a subset of indices $I_\mathcal{V}$. We use this, for example, in order to disallow the use of the greatest element $\top$ and the least element $\bot$ as variable assignments. We dissallow them because they could break other needed properties like monotonicity of $[\![\cdot]\!]_\nu$.

Definition 3.6 (Variable Assignment)
Let PROP be (G,PROP')-compatible over $(I, \sqsubseteq)$ with $G \in HRG_\Sigma$ and Var a set of variables. $I_\mathcal{V} \subseteq I$ is a set such that for all $A \in \Sigma$ and all $H \in L_G(A^\bullet)$

$$\exists i \in I_\mathcal{V}\ PROP(i)(H). \tag{3.8}$$

We call a function $\nu \in Var \mapsto I_\mathcal{V}$ a variable assignment *and the set of variable assignments $\mathcal{V} = Var \mapsto I_\mathcal{V}$.*

Example *A valid set $I_\mathcal{V}$ over the index set $(\mathbb{Z} \cup \{-\infty, \infty\}, \leq)$ for the predicate MOST-LIST-SIZE would be $I_\mathcal{V} = \mathbb{Z} \cup \{-\infty, \infty\}$. Taking the index set is always a valid $I_\mathcal{V}$ set for monotonic directed index sets. However, we do not want to have ∞, $-\infty$ and non-positive numbers as an assignment for a variable, since they break further conditions and is not necessary for $I_\mathcal{V}$. We instead choose $I_\mathcal{V} = \mathbb{N} \setminus \{0\}$.*

With variables that we assign values, we want to have a domain in which we can compute new indices based on the value assigned to the variable. This abstract domain behaves relative to the variables. We call the domain using these variables the *relative index set*. We use a function $[\![\cdot]\!]_\nu$ that maps relative indices to the concrete indices. Thus we have an element for every variable that works as a start element. Usually, this is just the variable itself, i.e. $[\![v]\!]_\nu = \nu(v)$. We also want to express the concrete indices in term of relative indices. These are used to give terminal hyperedges a concrete index. This way, the concrete index set is embedded into the relative index set. The embedding ensures on the one hand that we can be most precise if we assign terminal hyperedges an index, and on the other hand enables us to use widening techniques to map hyperedges to the greatest element $\top$ if we are unable to keep track. In such a case, we want to map the index to the greates element $\top$ and do not restrict the language any longer, as we have seen in Lemma 3.1. Additionally, the function $[\![\cdot]\!]_\nu$ has to behave monotonically in order to analyse descending chains. The descending chains then can be used to prove that if a hyperedge is assigned a terminating element t at some point in the execution, the current run had to terminate.

Definition 3.7 (Relative Index Set)
Let $(I, \sqsubseteq)$ be an index set and Var be a set of variables. We call $(I_{Rel}, \sqsubseteq_{Rel})$ their relative index set *if and only if there exists a function $[\![\cdot]\!]_\nu \in I_{Rel} \times \mathcal{V} \mapsto I$ such that*

$$\forall v \in Var\ \exists ir_v \in I_{Rel}\ \forall \nu \in \mathcal{V}\ [\![ir_v]\!]_\nu = \nu(v) \tag{3.9}$$

$$\forall i \in I\ \exists ir \in I_{Rel}\ \forall \nu \in \mathcal{V}\ [\![ir]\!]_\nu = i \tag{3.10}$$

$$\begin{gathered}\forall \nu \in \mathcal{V}\ \forall ir, jr \in I_{Rel} : \\ ir \sqsubset_{Rel} jr \Rightarrow [\![ir]\!]_\nu \sqsubset [\![jr]\!]_\nu\end{gathered} \tag{3.11}$$

Example *Let the relative index set be $(\mathbb{N}_{Rel}, \leq_{Rel})$ as defined in Example 3.3 for the concrete index set $(\mathbb{Z} \cup \{\infty, -\infty\}, \leq)$ for the predicate MOST-LIST-SIZE*

Example 3.3: Relative Index Set of *MOST-LIST-SIZE*. The integers are extended with variables. The connection between the integers and the relative integers is an evaluation of the variables to a value.

We define the relative index set for *MOST-LIST-SIZE* from Figure 3.2 and the variables Var as $\mathbb{Z}_{Rel} = \text{Var} \times \mathbb{Z} \cup \{\infty, -\infty\}$. We define $\leq_{Rel}$ as

$$\forall ir \in \mathbb{Z}_{Rel} \ -\infty \leq_{Rel} ir \leq_{Rel} \infty$$
$$(vars, z) \leq_{Rel} (vars', z') \Leftrightarrow vars \subseteq vars' \wedge z \leq z'$$

Furthermore we let $[\![\cdot]\!]_\nu$ be

$$[\![(vars, z)]\!]_\nu = z + \sum_{v \in vars} \nu(v)$$
$$[\![-\infty]\!]_\nu = -\infty$$
$$[\![\infty]\!]_\nu = \infty$$

from Example 3.2. Furthermore, let the index set for the variable assignment be $I_{\mathcal{V}} = \mathbb{N} \setminus \{0\}$ *as discussed before. Conditions* (3.9) *and* (3.10) *hold easily, because*

$$\nu(v) = [\![(\{v\}, 0)]\!]_\nu,$$
$$\infty = [\![\infty]\!]_\nu,$$
$$-\infty = [\![-\infty]\!]_\nu,$$
$$i = [\![(\emptyset, i)]\!]_\nu.$$

For condition (3.11)*, we assume that* $ir <_{Rel} jr$ *holds. We make a case distinction on ir's and jr's shape:*

1. $ir \neq \infty$ *and* $jr = \infty$*, thus* $\infty \neq [\![ir]\!]_\nu < [\![jr]\!]_\nu = \infty$.
2. $ir = -\infty$ *and* $jr \neq -\infty$*, thus* $-\infty = [\![ir]\!]_\nu < [\![jr]\!]_\nu \neq -\infty$.
3. $ir = (v_{ir}, z_{ir})$ *and* $jr = (v_{jr}, z_{jr})$*, and* $v_{ir} \subset v_{jr}$ *and* $z_{ir} \leq z_{jr}$*, then for all* $\nu \in \mathcal{V}$ *we have*

$$\sum_{v \in v_{ir}} \nu(v) < \sum_{v \in v_{jr}} \nu(v)$$
$$\Rightarrow z_{ir} + \sum_{v \in v_{ir}} \nu(v) < z_{jr} + \sum_{v \in v_{jr}} \nu(v)$$
$$\Rightarrow [\![ir]\!]_\nu < [\![jr]\!]_\nu.$$

4. $ir = (v_{ir}, z_{ir})$ *and* $jr = (v_{jr}, z_{jr})$*, and* $v_{ir} = v_{jr}$ *and* $z_{ir} < z_{jr}$*, then for all* $\nu \in \mathcal{V}$ *we have*

$$\sum_{v \in v_{ir}} \nu(v) = \sum_{v \in v_{jr}} \nu(v)$$
$$\Rightarrow z_{ir} + \sum_{v \in v_{ir}} \nu(v) < z_{jr} + \sum_{v \in v_{jr}} \nu(v)$$
$$\Rightarrow [\![ir]\!]_\nu < [\![jr]\!]_\nu.$$

3.5 Index Abstraction Rules

Now that we have a relative index set, we also want to be able to apply $PROP'$ from a (G,PROP')-compatible predicate $PROP$ to the relative index set. Deterministic rules furthermore replace the predicate $PROP'$. For every derivation we create two rules: one for a forward derivation (materialisation) and one for a backward derivation (abstraction). The rule for forward derivation $H \Rightarrow_{(e,p)} H'$ with $p = (A, R)$ takes an index of the hyperedge e and maps it to an assign function for the hypergraph R. The rule for the backward derivation $H \Leftarrow_{(e,p)} H'$ with $p = (A, R)$ takes an assign function for the hypergraph R and maps it to an index for the hyperedge e. These rules are called *index abstraction rules*.

We allow a loss of preciseness in order to calculate the indices. Since the predicate $PROP$ is monotonic directed, taking a larger index corresponds to a loss of preciseness and taking a smaller index corresponds to a gain of preciseness. We need to allow this loss and gain in the definitions for the index abstraction rules explicitly as $PROP'$ is not enforced to be monotonic directed too.

Example 3.4: Addition for $(\mathbb{Z}_{Rel}, \leq_{Rel})$ and $(\mathbb{Z} \cup \{-\infty, \infty\}, \leq)$. The relative addition is a combination of an integer addition and an union of variable sets.

We define the addition $+$ on $\mathbb{Z} \cup \{-\infty, \infty\}$ as:

- $\forall i \in \mathbb{Z} \cup \{-\infty, \infty\}\ \infty + i = i + \infty = \infty$
- $\forall i \in \mathbb{Z} \cup \{-\infty\}\ (-\infty) + i = i + (-\infty) = -\infty$
- $\forall i, j \in \mathbb{Z}\ i + j$ as usual

We define the addition $+$ on $\mathbb{Z}_{Rel}$ for $ir, (vs, z), (vs', z') \in \mathbb{N}_{Rel}$:

- $\infty + ir = ir + \infty = \infty$
- $(-\infty) + ir = ir + (-\infty) = -\infty$ if $ir \neq \infty$
- $(vs, z) + (vs', z') = \begin{cases} (vs \cup vs', z + z') & \text{if } vs \cap vs' = \emptyset \\ \infty & \text{else} \end{cases}$

We also shorten the elements of $\mathbb{N}_{Rel}$ without the tuple as follows:

- $z = (\emptyset, z)$
- $z + \sum_{v \in vs} v = (v, z)$

The index abstraction rule for a forward derivation $H \Rightarrow_{(e,p)} H'$ with $p = (A, R)$ has an index i for the nonterminal hyperedge e as an input. In the forward derivation, e is replaced by a hypergraph R. The rule takes the index i and maps it to an index assign function *assign*. We then use that the predicate is monotonic directed in order to reduce the preciseness of i and *assign*. Thus we require that there exists an $j \sqsupseteq i$ for which $PROP'(j)(R, assign)$ holds.

Definition 3.8 (Forward Index Abstraction Rule)
Let PROP be a (G,PROP')-compatible, monotonic directed predicate over $(I, \sqsubseteq)$ with $G = (N, T, P, Z) \in HRG_\Sigma$ and $(I_{Rel}, \sqsubseteq_{Rel})$ the relative index set of $(I, \sqsubseteq)$.

$$\varphi^m_{(A,R)} \in I_{Rel} \mapsto (E_R \mapsto I_{Rel})$$

is a forward index abstraction rule *for the production $(A, R) \in P$ if and only if for all $A^\bullet \Rightarrow R \Rightarrow^* H \in \mathcal{H}_T$, $\nu \in \mathcal{V}$ and $ir \in I_{Rel}$*

$$\begin{aligned} & PROP([\![ir]\!]_\nu)(H) \\ \Rightarrow\ & \exists j \sqsupseteq [\![ir]\!]_\nu\ PROP'(j)([\![\varphi^m_{(A,R)}(ir)]\!]_\nu, R) \\ \wedge\ & \forall e \in E_R\ PROP([\![\varphi^m_{(A,R)}(ir)]\!]_\nu(e))(H(e)), \end{aligned} \tag{3.12}$$

Example 3.5: Forward Index Abstraction Rules for MOST-LIST-SIZE.

Let MLS be the ($List$,MLS')-compatible and monotonic directed predicate from Example 3.2 with the relative index set $(\mathbb{N}_{Rel}, \leq_{Rel})$ from Example 3.3. $\varphi^m_1, \varphi^m_2, \varphi^m_1$ as below are forward index abstraction rules for the productions of $List$ from Example 3.1. We use + for this relative index as in Example 3.4.

- $\varphi^m_1(ir) = \{n_1 \mapsto 1\}$
- $\varphi^m_2(ir) = \{n_1 \mapsto 1, L_2 \mapsto ir - 1\}$
- $\varphi^m_2(ir) = \{L_1 \mapsto ir - 1, L_2 \mapsto ir - 1\}$

Example *We want to prove that the functions from Example 3.5 are indeed forward index abstraction rules. For these functions, we need to prove condition* (3.12) *in Definition 3.8. We denote the edges by their name given in the rules in Example 3.1.*

φ^m_1 *We prove this independent of $[\![ir]\!]_\nu$ as the size of n_1 is 1.*

$$\begin{aligned} & MLS([\![ir]\!]_\nu)(H) \\ \Rightarrow & [\![ir]\!]_\nu \geq 1 \wedge MLS(1)(H(n_1))) && \text{by production 1} \\ \Rightarrow & \begin{cases} [\![ir]\!]_\nu \geq [\![\varphi^m_1(ir)]\!]_\nu(n_1) \\ \wedge MLS([\![\varphi^m_1(ir)]\!]_\nu(n_1))(H(n_1))) \end{cases} && \text{as } \varphi^m_1(ir)(n_1) = 1 \\ \Rightarrow & \begin{cases} [\![ir]\!]_\nu \geq [\![\varphi^m_1(ir)]\!]_\nu \\ \wedge \forall e \in E_R\ MLS([\![\varphi^m_1(ir)]\!]_\nu(e))(H(e)) \end{cases} && \text{as } E_R = \{n_1\} \\ \Rightarrow & \begin{cases} MLS'([\![ir]\!]_\nu)([\![\varphi^m_1(ir)]\!]_\nu, R) \\ \wedge \forall e \in E_R\ MLS([\![\varphi^m_1(ir)]\!]_\nu(e))(H(e)) \end{cases} && \text{replacing by } MLS' \end{aligned}$$

φ_2^m *This time we assume that the size of the list H is at most ir and conclude that as the size of n_1 is 1, the size of L_2 is at most $ir-1$.*

$$
\begin{aligned}
& MLS([\![ir]\!]_\nu)(H) \\
\Rightarrow & \begin{cases} MLS(1)(H(n_1)) \\ \wedge MLS([\![ir-1]\!]_\nu)(H(L_2)) \end{cases} && \text{as explained above} \\
\Rightarrow & \begin{cases} MLS([\![\varphi_2^m(ir)]\!]_\nu(n_1))(H(n_1)) \\ \wedge MLS([\![\varphi_2^m(ir)]\!]_\nu(L_2))(H(L_2)) \end{cases} && \text{replaced by } \varphi_2^m \\
\Rightarrow & \forall e \in E_R \; MLS([\![\varphi_2^m(ir)]\!]_\nu(e))(H(e)) && \text{as } E_R = \{n_1, L_2\} \\
\Rightarrow & \begin{cases} \sum\limits_{e \in E_R} [\![\varphi_2^m(ir)]\!]_\nu(e) = [\![ir]\!]_\nu \\ \wedge \forall e \in E_R MLS([\![\varphi_2^m(ir)]\!]_\nu(e))(H(e)) \end{cases} && \text{as } [\![ir-1]\!]_\nu + 1 = [\![ir]\!]_\nu \\
\Rightarrow & \begin{cases} MLS'([\![ir]\!]_\nu)([\![\varphi_2^m(ir)]\!]_\nu, R) \\ \wedge \forall e \in E_R \; MLS([\![\varphi_2^m(ir)]\!]_\nu(e))(H(e)) \end{cases} && \text{replacing by } MLS'
\end{aligned}
$$

φ_3^m *We again assume that the list is of at most size i. Every list segment has a size of at least 1; thus $i \geq 2$ holds and each list segment hase a length of at most $i-1$.*

$$
\begin{aligned}
& MLS([\![ir]\!]_\nu)(H) \\
\Rightarrow & \begin{cases} \exists j_1, j_2 \in I \; j_1 + j_2 = ir \\ \wedge MLS(j_1)(H(L_1)) \\ \wedge MLS(j_2)(H(L_2)) \end{cases} && \text{see Example 3.2} \\
\Rightarrow & \begin{cases} MLS([\![ir]\!]_\nu - 1)(H(L_1)) \\ \wedge MLS([\![ir]\!]_\nu - 1)(H(L_2)) \end{cases} && \text{as discussed above} \\
\Rightarrow & \begin{cases} MLS([\![ir-1]\!]_\nu)(H(L_1)) \\ \wedge MLS([\![ir-1]\!]_\nu)(H(L_2)) \end{cases} \\
\Rightarrow & \begin{cases} MLS([\![\varphi_3^m(ir)]\!]_\nu(L_1))(H(L_1)) \\ \wedge MLS([\![\varphi_3^m(ir)]\!]_\nu(L_2))(H(L_2)) \end{cases} && \text{replaced by } \varphi_3^m \\
\Rightarrow & \forall e \in E_R \; MLS([\![\varphi_3^m(ir)]\!]_\nu(e))(H(e)) && \text{as } E_R = \{L_1, L_2\} \\
\Rightarrow & \begin{cases} \sum\limits_{e \in E_R} [\![\varphi_3^m(ir)]\!]_\nu(e) = [\![ir-1]\!]_\nu + [\![ir-1]\!]_\nu \geq [\![ir]\!]_\nu \\ \wedge \forall e \in E_R \; MLS([\![\varphi_3^m(ir)]\!]_\nu(e))(H(e)) \end{cases} && \text{as } [\![ir-1]\!]_\nu \geq 1 \\
\Rightarrow & \begin{cases} [\![ir-1]\!]_\nu + [\![ir-1]\!]_\nu \geq [\![ir]\!]_\nu \\ \wedge MLS'([\![ir-1]\!]_\nu + [\![ir-1]\!]_\nu)([\![\varphi_3^m(ir)]\!]_\nu, R) \\ \wedge \forall e \in E_R \; MLS([\![\varphi_3^m(ir)]\!]_\nu(e))(H(e)) \end{cases} && \text{replaced by MLS'}
\end{aligned}
$$

The index abstraction rule for backward derivations $H \Leftarrow_{(e,p)} H'$ with $p = (A, R)$ (i.e. abstraction) on the other hand uses the function $assign$ for the

hypergraph R and maps it to an index i for the hyperedge e. For backward derivations where we reverse the replacement of the hyperedge e by the hypergraph R, we want to find an index for the hyperedge e. Similar to forward derivations we allow loss of preciseness. We will later have the problem, that we only have the information that $\forall e \in E_R\ PROP(assign(e))(H(e))$ holds and not that $PROP'(i)(assign, R)$ holds. We combat this loss of information by forcing the index abstraction rule to generate only indices for which exists an $j \sqsubseteq i$ (prove for more precise index) such that $PROP'(j)(assign, R)$ holds.

Definition 3.9 (Backward Index Abstraction Rule)
Let $PROP$ be a (G,PROP')-compatible, monotonic directed predicate over $(I, \sqsubseteq)$ with $G = (N, T, P, Z) \in HRG_\Sigma$ and $(I_{Rel}, \sqsubseteq_{Rel})$ be the relative index set of $(I, \sqsubseteq)$.

$$\varphi^a_{(A,R)} \in (E_R \mapsto I_{Rel}) \mapsto I_{Rel}$$

is a backward index abstraction rule *for the production $(A, R) \in P$ if and only if for all $A^\bullet \Rightarrow R \Rightarrow^* H$, $\nu \in \mathcal{V}$ and $assign_{Rel} \in E_R \mapsto I_{Rel}$*

$$\begin{aligned} &\exists j \sqsubseteq [\![\varphi^a_{(A,R)}(assign_{Rel})]\!]_\nu\ PROP'(j)([\![assign_{Rel}]\!]_\nu, R) \\ &\quad \wedge \forall e \in E_R\ PROP([\![assign_{Rel}]\!]_\nu(e))(H(e)) \\ &\quad\quad \Rightarrow PROP([\![\varphi^a_{(A,R)}(assign_{Rel})]\!]_\nu)(H), \end{aligned} \tag{3.13}$$

$$\begin{aligned} &\forall e \in E_R\ PROP([\![assign_{Rel}]\!]_\nu(e))(H(e)) \\ &\Rightarrow \exists j \sqsubseteq [\![\varphi^a_{(A,R)}(assign_{Rel})]\!]_\nu\ PROP'(j)([\![assign_{Rel}]\!]_\nu, R). \end{aligned} \tag{3.14}$$

Condition (3.13) is however redundant, as $PROP$ is (G, $PROP'$)-compatible and monotonic directed condition (3.13) follows from the condition (3.14).

Lemma 3.2
In Definition 3.9, the condition (3.13) *follows from condition* (3.14).

Proof If $\forall e \in E_R\ PROP([\![assign_{Rel}]\!]_\nu(e))(H(e))$ does not hold, condition (3.13) holds trivially. Thus assume $\forall e \in E_R\ PROP([\![assign_{Rel}]\!]_\nu(e))(H(e))$ does hold. Let $j \sqsubseteq [\![\varphi^a_{(A,R)}(assign_{Rel})]\!]_\nu$ such that condition (3.14) holds:

$$\begin{aligned} &\forall e \in E_R\ PROP([\![assign_{Rel}]\!]_\nu(e))(H(e)) \\ \overset{(3.14)}{\Rightarrow} &\begin{cases} \forall e \in E_R\ PROP([\![assign_{Rel}]\!]_\nu(e))(H(e)) \\ \wedge PROP'(j)([\![assign_{Rel}]\!]_\nu, R) \end{cases} \\ \overset{(3.1)}{\Rightarrow} &PROP(j)(H) \\ \overset{(3.3)}{\Rightarrow} &PROP([\![\varphi^a_{(A,R)}(assign_{Rel})]\!]_\nu)(H) \end{aligned}$$

■

Example *For the functions φ^a_1, φ^a_2 and φ^a_3, we need to prove condition* (3.14). *Since MLS' is independent of R, we can prove condition* (3.14) *on an arithmetic level. A linearity property of $[\![\cdot]\!]_\nu$ will help us to prove MLS'.*

Lemma 3.3 (Linearity of $[\![\cdot]\!]_\nu$ for MOST-LIST-SIZE)
For all $i, j \in I_{Rel}$ we have $[\![i]\!]_\nu + [\![j]\!]_\nu \leq [\![i + j]\!]_\nu$

Proof See Appendix C.5

Example 3.6: Backward Index Abstraction Rules for MOST-LIST-SIZE.

Let MLS be the ($List$,MLS')-compatible and monotonic directed predicate from Example 3.2 with the relative index set $(\mathbb{N}_{Rel}, \leq_{Rel})$ from Example 3.3. φ_1^a, φ_2^a and φ_1^a as below are backward index abstraction rules for the productions of $List$ from Example 3.1. We use $+$ for this relative index as in Example 3.4.

- $\varphi_1^a(assign_{Rel}) = assign_{Rel}(n_1)$
- $\varphi_2^a(assign_{Rel}) = assign_{Rel}(L_2) + assign_{Rel}(n_1)$
- $\varphi_3^a(assign_{Rel}) = assign_{Rel}(L_1) + assign_{Rel}(L_2)$

φ_1^a *Since φ_1^a is identical to the index of n_1, the condition holds directly.*

$$\begin{aligned} &[\![assign_{Rel}]\!]_\nu(n_1) = [\![\varphi_1^a(assign_{Rel})]\!]_\nu \\ \Rightarrow &[\![assign_{Rel}]\!]_\nu(n_1) \leq [\![\varphi_1^a(assign_{Rel})]\!]_\nu \\ \Rightarrow &MLS'([\![\varphi_1^a(assign_{Rel})]\!]_\nu)([\![assign_{Rel}]\!]_\nu, R) \end{aligned}$$

φ_2^a *By linearity, we can shift all relative indices inside of $[\![\cdot]\!]_\nu$.*

$$\begin{aligned} &[\![assign_{Rel}(n_1)]\!]_\nu + [\![assign_{Rel}(L_2)]\!]_\nu \\ \overset{Lemma\ 3.3}{\leq} &[\![assign_{Rel}(n_1) + assign_{Rel}(L_2)]\!]_\nu \\ = &[\![\varphi_2^a(assign_{Rel})]\!]_\nu \\ \Rightarrow &MLS'([\![\varphi_2^a(assign_{Rel})]\!]_\nu)([\![assign_{Rel}]\!]_\nu, R) \end{aligned}$$

φ_3^a *Again we use the linearity to shift all relative indices inside of $[\![\cdot]\!]_\nu$.*

$$\begin{aligned} &[\![assign_{Rel}(L_1)]\!]_\nu + [\![assign_{Rel}(L_2)]\!]_\nu \\ \overset{Lemma\ 3.3}{\leq} &[\![assign_{Rel}(L_2) + assign_{Rel}(L_2)]\!]_\nu \\ = &[\![\varphi_3^a(assign_{Rel})]\!]_\nu \\ \Rightarrow &MLS'([\![\varphi_3^a(assign_{Rel})]\!]_\nu)([\![assign_{Rel}]\!]_\nu, R) \end{aligned}$$

By combining now all forward and backward index abstraction rules, we get a set of index abstraction rules that can be used to approximate $PROP'$ with a relative index. We use these rules later to define a transfer function that transfers relative indices on an abstract hypergraph transition system. As these rules are now deterministic, the transfer function will also behave deterministically.

Definition 3.10 (Index Abstraction Rules)
Let $PROP$ be a (G,PROP')-compatible, monotonic directed predicate over $(I, \sqsubseteq)$ with $G = (N, T, P, Z) \in HRG_\Sigma$ and $(I_{Rel}, \sqsubseteq_{Rel})$ the relative index set of $(I, \sqsubseteq)$. The index abstraction rules *for all $A \in \Sigma$ and every rule $(A, R) \in P$ are the computable functions*

$$\varphi^a_{(A,R)} \in (E_R \mapsto I_{Rel}) \mapsto I_{Rel}$$
$$\varphi^m_{(A,R)} \in I_{Rel} \mapsto (E_R \mapsto I_{Rel})$$

where $\varphi^a_{(A,R)}$ is a backward index abstraction rule for (A, R) and $\varphi^m_{(A,R)}$ is a forward index abstraction rule for (A, R).

In general, the rules are not most precise and thus are not feasible to prove $PROP(i)(H)$ for any i. Condition (3.14) is additionally needed since we may not be able to prove this for every R and every *assign* automatically. Thus the rule then only gives us indices for which $PROP'$ is already satisfied. To sum up, we only need to prove the conditions (3.12) and (3.14) in order to prove a set of index abstraction rules. We also remark that in condition (3.12) $PROP'$ can be skipped. $PROP'$ is used to prove that the structure of R together with assign proves an index i, but the transfer function later is only interested in that for all hyperedges $e \in E_R$ $PROP(assign(e))(H(e))$ holds and thus may ignore the structure of R.

Chapter 4

Predicate Abstraction Analysis

We assume for the following analysis that we have an abstract HRG transition system TS_a of concrete hypergraph transition system TS_c. We use hyperedge-replacement grammars to abstract the hypergraphs in TS_c. Furthermore, we use predicates which are compatible with the grammar to filter the abstract hypergraphs. By filtering the hypergraph language to an arbitrary but fixed index of the predicate, we are then able to propagate the index further down the abstract transition system. The propagation is be done via a monotonic data flow analysis framework (see Appendix B), which is a standard framework for static program analysis[NRNH99, p. 63]. A data flow analysis describes an equation system, for which every state in the transition system (or the flow graph of a program [NRNH99, p. 5ff.]) is interpreted as labels. The initial label is assigned some extremal value of the abstract domain, which is then propagated through the labels via transfer functions and combined with the least upper bound. The data flow analysis defines an equation system for which we compute a solution, i.e. the (least) fixpoint. The solution of the data flow analysis then gives an upper bound on the index after transitioning through the transition system until either a final state or the state which we started in is reached. We are interested in the value of the index after reaching the state in which we started. If this index is smaller than the extremal value we started with, we then can conclude that it is not possible to visit the state infinitely often, since this would yield a descending chain of indices. However, infinitely descending chains would yield a terminating index, which also terminates the path.

4.1 Abstracting Predicates

Now we focus on analysing one state. We introduce a data flow analysis that abstracts the indices of a predicate $PROP$. Our input for the analysis is the state in question, the abstract HRG transition system over a hyperedge-replacement grammar G and a (G,PROP')-compatible and monotonic directed predicate $PROP$ with the relative index set $(I_{Rel}, \sqsubseteq_{Rel})$ and index abstraction rules $\varphi^a_{(A,R)}$ and $\varphi^m_{(A,R)}$. We discussed these predicates and index abstraction rules in Chap-

ter 3. The idea is to initialise the abstract hypergraph with variables for each nonterminal hyperedge. Afterwards, we propagate the relative index via φ_r^a and φ_r^m to all states in the abstract transition system. We define all components of the data flow analysis in the following in multiple steps. For the labels and data flow relation of the data flow analysis, we use the abstract transition system with its state space and transition relation with a small change. We replace q by two labels q_0 for outgoing transitions (and as an initial/extremal label) and q_f for ingoing transitions (and as a final label). We can catch the index after just one iteration (or one cycle) when again reaching q. The new label q_0 is used as the initial label. For the domain of the analysis, we use a mapping of hyperedges to the relative index set $(I_{Rel}, \sqsubseteq_{Rel})$. For the extremal value ι, we will use afresh variable for every hyperedge with a nonterminal label and concrete indices for hyperedges with a terminal label. Finally, we construct the transfer function $\dot{\varphi}$ by using the index abstraction rules φ^a and φ^m.

Definition 4.1 (Untangled Flow Graph)
Let TS_a be an abstract hypergraph transition system and $q \in States_a$ a state. We define the flow graph of TS_a that untangles q as (Lab, Ex, F) with

- $Lab = (States_a \setminus \{q\}) \cup \{q_0, q_f\}$
- $Ex = \{q_0\}$
- $$F = \begin{array}{lll} & \{(s, s') & |s \to_a s' \wedge s \neq q \wedge s' \neq q\} \\ \cup & \{(q_0, s') & |s \to_a s' \wedge s = q \wedge s' \neq q\} \\ \cup & \{(s, q_f) & |s \to_a s' \wedge s \neq q \wedge s' = q\} \\ \cup & \{(q_0, q_f) & |s \to_a s' \wedge s = q \wedge s' = q\} \end{array}$$

4.1.1 Assigning Domain

In this section, we focus on the domain $(Assigns_{Rel}, \dot{\sqsubseteq}_{Rel})$. $Assigns_{Rel}$ is the domain of functions (or assign mappings) that maps every hyperedge of (all) hypergraphs to a relative index. The partial order $\dot{\sqsubseteq}_{Rel}$ of the domain is the pointwise application of the ordering from the relative index set. Since $(I_{Rel}, \sqsubseteq_{Rel})$ is already a complete lattice, $(Assigns_{Rel}, \dot{\sqsubseteq}_{Rel})$ is also a complete lattice.

Definition 4.2 (Assigning Domain)
Let $(I_{Rel}, \sqsubseteq_{Rel})$ be a relative index set and E a set of hyperedges. Then we define $Assigns_{Rel}$ as

$$Assigns_{Rel} = (E \mapsto I_{Rel}) \tag{4.1}$$

and $\dot{\sqsubseteq}_{Rel}$ as

$$\begin{array}{c} assign_{Rel} \;\dot{\sqsubseteq}_{Rel}\; assign'_{Rel} \\ \Leftrightarrow \forall e \in E\; assign_{Rel}(e) \sqsubseteq_{Rel} assign'_{Rel}(e). \end{array} \tag{4.2}$$

Lemma 4.1 (Assigning Domain is a Complete Lattice)
The assigning domain $(Assigns_{Rel}, \dot{\sqsubseteq}_{Rel})$ from Definition 4.2 is a complete lattice.

PROOF See Appendix C.1.

Example *We use the relative index domain* $(\mathbb{N}_{Rel}, \leq_{Rel})$ *from Example 3.3. Then the assigning domain of this relative index domain is* $(E \mapsto \mathbb{N}_{Rel}, \dot{\leq}_{Rel})$. *For* $E = \{1,2,3,4\}$ *and* $Var = \{x_1, x_2, x_3\}$ *we have that*

$$\begin{array}{rl} & \{1 \mapsto 1, \quad 2 \mapsto 1, \quad 3 \mapsto x_2, \quad 4 \mapsto x_3\} \\ \sqsubseteq^E & \{1 \mapsto 2 + x_1, \quad 2 \mapsto 1, \quad 3 \mapsto 10 + x_2, \quad 4 \mapsto x_3\} \end{array}$$

$$\begin{array}{rl} & \{1 \mapsto 1, \quad 2 \mapsto 1, \quad 3 \mapsto x_2, \quad 4 \mapsto x_3\} \\ \not\sqsubseteq^E & \{1 \mapsto 0, \quad 2 \mapsto 1, \quad 3 \mapsto \infty, \quad 4 \mapsto x_2 + x_3\} \end{array}$$

Furthermore, the least upper bound of these three functions would be

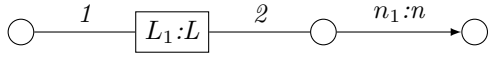
$$\{1 \mapsto 2 + x_1, 2 \mapsto 1, 3 \mapsto \infty, 4 \mapsto x_2 + x_3\}.$$

Next, we introduce how we define the extremal value ι. Here we use condition (3.10) from Definition 3.7 ensuring that $[\![\cdot]\!]_\nu$ behaves like a variable assignment for a particular relative index. ι assigns the nonterminal hyperedges these variable indices and terminal hyperedges e an index i for which $PROP(i)(e^\bullet)$ holds. We always assume that we can compute such an index (and at least we can always assign the greatest element $\top$ by Lemma 3.1). Since we can compute such an index, it is not necessary to save indices for the terminal hyperedges, as we would compute them as needed.

Definition 4.3 (Extremal Value)
Let $(I_{Rel}, \sqsubseteq_{Rel})$ *be a relative index set and* $Var = \{v_e | e \in E_H \wedge label(e) \in N\}$ *the set of variables for the set of hyperedges* E_H *in the hypergraph* H *from* $q = \langle loc, H \rangle$. *Then we define* ι *as:*

$$\iota(e) = \begin{cases} ir_e \text{ with } [\![ir_e]\!]_\nu = \nu(v_e) & \text{if } label(e) \in N \wedge e \in E_H \\ ir \text{ with } PROP([\![ir]\!]_\nu)(e^\bullet) & \text{if } label(e) \in T \wedge e \in E_H \\ \bot & \text{else.} \end{cases} \tag{4.3}$$

Example *Let* $(\mathbb{N}_{Rel}, \sqsubseteq_{Rel})$ *be the relative index set from Example 3.3 and the hypergraph for* $q = \langle loc, H_0 \rangle$ *as:*

H_0 : 

Then ι *is defined as*

$$\iota(e) = \begin{cases} v_{L_1} & \text{if } e = L_1 \\ 1 & \text{if } e = n_1 \\ 0 & \text{else.} \end{cases}$$

4.1.2 Transfer Function

We split the transfer functions into three functions for materialisation, concrete hypergraph operation and abstraction and combine them again in the end. For abstract HRG transition systems, we can decompose a transition into these three

steps. Before every hypergraph operation, the current hypergraph is materialised. Then the concrete hypergraph operation is executed on the materialised hypergraph. After the operation, the hypergraph is abstracted again. We will use the index abstraction rules from Definition 3.10 to deal with materialisation and abstraction. The hypergraph operation affects only hyperedges with a terminal label; thus, we need to adjust only these. We can, fortunately, adjust terminal hyperedges on a concrete level. If new hyperedges are introduced in an operation step, we assign them a new valid index.

In the first paragraph, we focus on the materialisation, that is a forward derivation $H \Rightarrow_{rs} H_\gamma$ where rs is a sequence of direct forward derivations. Then the concrete hypergraph operation $H_\gamma \xrightarrow{\mu} H'_\gamma$ is handled in the second paragraph. Afterwards, we also need to adjust the index for the abstraction $H'_\gamma \Leftarrow_{rs'} H'$ where rs' is a sequence of direct backward derivations as we see in the third paragraph. In the last paragraph, we combine these three functions into one function.

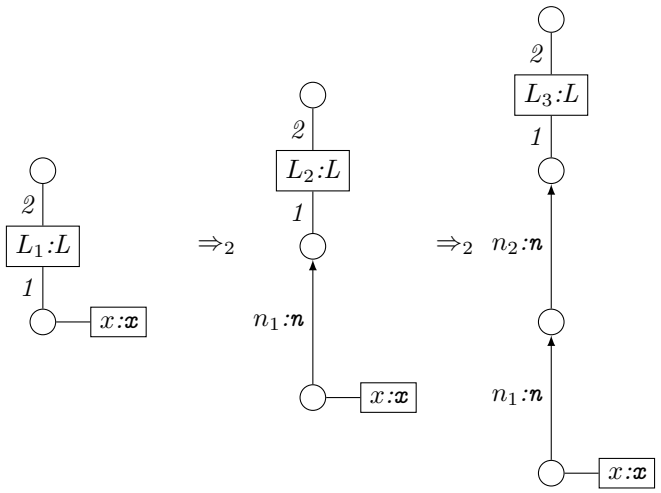
Materialisation We define the transfer function for a materialisation $H \Rightarrow_{rs} H_\gamma$ as applying the rules for every single derivation one after another. For single derivations, we use the index abstraction rule $\varphi^m_{(A,R)}$ as a transfer function.

Definition 4.4 (Materialisation Transfer Function)
Let $G \in HRG_\Sigma$ and $\varphi^m_{(A,R)}$ be index abstraction rules for all $(A,R) \in P$.For $H \Rightarrow_{r_1} \ldots \Rightarrow_{r_n} H'$ we define the materialisation transfer function $\dot{\varphi}^m_{r_1 \ldots r_n} \in Assigns_{Rel} \mapsto Assigns_{Rel}$:

$$\dot{\varphi}^m_{(\xi,(A,R))}(assign_{Rel})(e) = \begin{cases} assign_{Rel}(e) & \text{if } e \in E_H \setminus \{\xi\} \\ \varphi^m_{(A,R)}(assign_{Rel})(e) & \text{if } e \in E_R \\ \bot & \text{else} \end{cases}$$

$$\dot{\varphi}^m_{r_1,r_2,\ldots r_n}(assign_{Rel}) = \dot{\varphi}^m_{r_2,\ldots r_n}(\dot{\varphi}^m_{r_1}(assign_{Rel})).$$

Example *For the following example, we use the hyperedge-replacement grammar List from Example 3.1 and use the index abstraction rules from Example 3.5 and Example 3.6. We first take an example materialisation and assigning function and then show how $\dot{\varphi}^m$ maps this function. Remark now that although the hyperedge x is not a list, we assign it the index 1. MLS does only count the hyperedges (the grammar List enforces the list structure), and thus $MLS(1)(x^\bullet)$ holds. Let the derivation $H \Rightarrow_{(2)} H' \Rightarrow_{(2)} H''$ be*



The initial assigning function is

$$\begin{aligned} assign(L_1) &= v_{L_1}, \\ assign(x) &= 1, \\ \forall e \notin E_H \; assign(e) &= 0. \end{aligned}$$

The first application of $\dot{\varphi}_2^m$ for the first derivation step changes assign to

$$\begin{aligned} \dot{\varphi}^m_{(L_1,R_2)}(assign)(L_2) &= v_{L_1} - 1, \\ \dot{\varphi}^m_{(L_1,R_2)}(assign)(n_1) &= 1, \\ \dot{\varphi}^m_{(L_1,R_2)}(assign)(x) &= 1, \\ \forall e \notin E_{H'} \; \dot{\varphi}^m_{(L_1,R_2)}(assign)(e) &= 0. \end{aligned}$$

The second application of $\dot{\varphi}_2^m$ for the second derivation step changes assign furthermore to

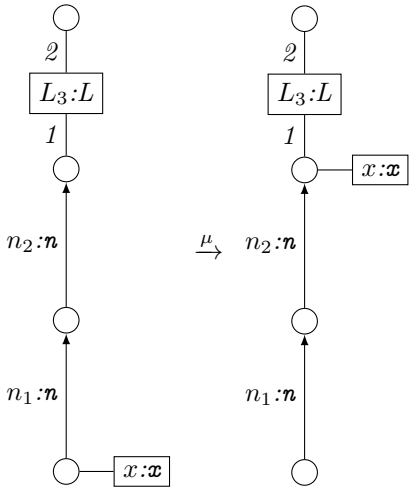
$$\begin{aligned} \dot{\varphi}^m_{(L_1,R_2),(L_2,R_2)}(assign)(L_3) &= v_{L_1} - 2, \\ \dot{\varphi}^m_{(L_1,R_2),(L_2,R_2)}(assign)(n_1) &= 1, \\ \dot{\varphi}^m_{(L_1,R_2),(L_2,R_2)}(assign)(n_2) &= 1, \\ \dot{\varphi}^m_{(L_1,R_2),(L_2,R_2)}(assign)(x) &= 1, \\ \forall e \notin E_{H''} \; \dot{\varphi}^m_{(L_1,R_2),(L_2,R_2)}(assign)(e) &= 0. \end{aligned}$$

Concrete Hypergraph Operation The concrete hypergraph operations μ can only effect terminal hyperedges by design. The operation is done on the concrete part of the hypergraph. Terminal hyperedges are reasonably simple to handle since we require to be able to assign these hyperedges an index i such that the hyperedge satisfies $PROP(i)$. Thus we can use the identity for all not changed hyperedges and assign some index i such that $PROP(e)(e^\bullet)$ hold for all changed hyperedges e.

Definition 4.5 (Transfer Functions for Hypergraph Operations) *Let $\mu \in \mathcal{H}_\Sigma \mapsto \mathcal{H}_\Sigma$ be a concrete hypergraph operation. The transfer function for the concrete hypergraph operation $H \xrightarrow{\mu} H'$ is*

$$\dot{\varphi}^o_{(H,H')}(assign_{Rel})(e) = \begin{cases} assign_{Rel}(e) & \text{if } e \in E_{H'} \cap E_H \\ ir \text{ such } PROP([\![ir]\!]_\nu)(e^\bullet) & \text{if } e \in E_{H'} \setminus E_H \\ \bot & \text{else} \end{cases}$$

Example *We continue our example with the grammar List from Example 3.1 with the relative index set from Example 3.3. The operation μ changes the hyperedge x to another object. Let the concrete hypergraph operation $H \xrightarrow{\mu} H'$ be the following:*



For this example, we take the function assign derived from the transfer functions from the example before:

$$\begin{aligned} assign(L_3) &= v_{L_1} - 2, \\ assign(n_1) &= 1, \\ assign(n_2) &= 1, \\ assign(x) &= 1, \\ \forall e \notin E_H \; assign(e) &= 0. \end{aligned}$$

Then by applying the transfer function for the statement execution $\dot{\varphi}^o$, this yields the same function as we do not delete or add new terminal hyperedges:

$$\begin{aligned} \dot{\varphi}^o_{(H,H')}(assign)(L_3) &= v_{L_1} - 2, \\ \dot{\varphi}^o_{(H,H')}(assign)(n_1) &= 1, \\ \dot{\varphi}^o_{(H,H')}(assign)(n_2) &= 1, \\ \dot{\varphi}^o_{(H,H')}(assign)(x) &= 1, \\ \forall e \notin E_{H'} \; \dot{\varphi}^o_{(H,H')}(assign)(e) &= 0. \end{aligned}$$

Abstraction For an abstraction, we have a sequence of derivations as $H' \Rightarrow_{rs} H$. Similar to materialisation we use for all $(A, R) \in P$ the index abstraction rules $\varphi^a_{(A,R)}$ as transfer functions.

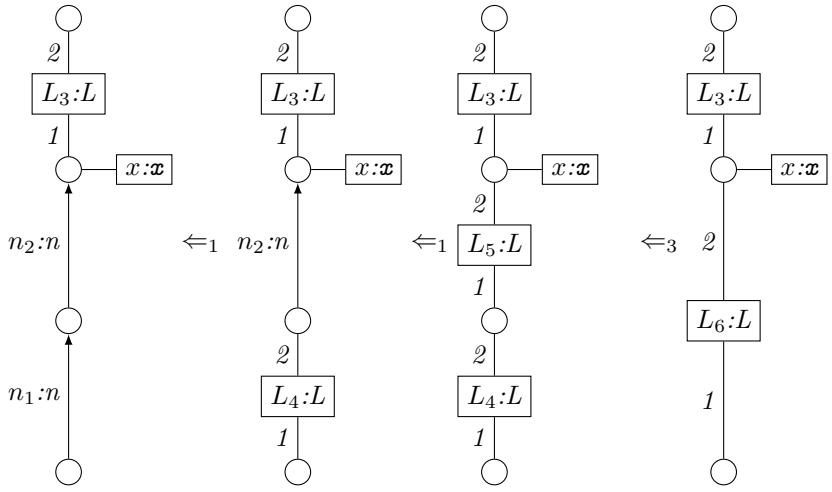
Definition 4.6 (Abstraction Transfer Function)
Let $G \in HRG_\Sigma$ and $\varphi^a_{(A,R)}$ be index abstraction rules for all productions $(A, R) \in P$. For $H \Rightarrow_{r_1} \ldots \Rightarrow_{r_n} H'$ we define the abstraction transfer function $\dot{\varphi}^a_{r_1,\ldots r_n} \in Assigns_{Rel} \mapsto Assigns_{Rel}$:

$$\dot{\varphi}^a_{(\xi,(A,R))}(assign_{Rel})(e) = \begin{cases} assign_{Rel}(e) & \text{if } e \in E_H \setminus E_R \\ \varphi^a_{(A,R)}(assign_{Rel}) & \text{if } e = \xi \\ \bot & \text{else} \end{cases}$$

$$\dot{\varphi}^a_{r_1,r_2\ldots r_n}(assign_{Rel}) = \dot{\varphi}^a_{r_2,\ldots r_n}(\dot{\varphi}^a_{r_1}(assign_{Rel}))$$

Example *For the following example, we use the hyperedge-replacement grammar List from Example 3.1 and use the index abstraction rules from Example 3.5 and Example 3.6. In this example, we abstract the hypergraph in three steps as a symbolic execution does it using the grammar List from Example 3.1 with the backward derivation*

$$H \Leftarrow_{(1)} H' \Leftarrow_{(1)} H'' \Leftarrow_{(3)} H''' :$$



We start with the following assign function:

$$\begin{aligned} assign(L_3) &= v_{L_1} - 2, \\ assign(n_1) &= 1, \\ assign(n_2) &= 1, \\ assign(x) &= 1, \\ \forall e \notin E_H \; assign(e) &= 0. \end{aligned}$$

Applying the first transfer function $\dot{\varphi}^a_1$ will assign the new hyperedge L_4 the same index 1 as the hyperedge n_1. Similarly, in the next step, L_5 is assigned the same

index 1 *as the hyperedge* n_2*, which thus yields the assign function*

$$\begin{aligned}\dot{\varphi}^m_{(L_4,R_1),(L_5,R_1)}(assign)(L_3) &= v_{L_1} - 2,\\ \dot{\varphi}^m_{(L_4,R_1),(L_5,R_1)}(assign)(L_4) &= 1,\\ \dot{\varphi}^m_{(L_4,R_1),(L_5,R_1)}(assign)(L_5) &= 1,\\ \dot{\varphi}^m_{(L_4,R_1),(L_5,R_1)}(assign)(x) &= 1,\\ \forall e \notin E_{H''}\ \dot{\varphi}^m_{(L_4,R_1),(L_5,R_1)}(assign)(e) &= 0.\end{aligned}$$

The last abstraction step combines the hyperedges L_4 *and* L_5 *into a new hyperedge* L_6 *by applying the rule 3.* $\dot{\varphi}^a_3$ *combines then the indices of* L_4 *and* L_5 *into the addition of these indices* $(1 + 1 = 2)$ *and assigns the sum* 2 *to* L_6*:*

$$\begin{aligned}\dot{\varphi}^m_{(L_4,R_1),(L_5,R_1),(L_6,R_3)}(assign)(L_3) &= v_{L_1} - 2,\\ \dot{\varphi}^m_{(L_4,R_1),(L_5,R_1),(L_6,R_3)}(assign)(L_6) &= 2,\\ \dot{\varphi}^m_{(L_4,R_1),(L_5,R_1),(L_6,R_3)}(assign)(x) &= 1,\\ \forall e \notin E_{H'''}\ \dot{\varphi}^m_{(L_4,R_1),(L_5,R_1),(L_6,R_3)}(assign)(e) &= 0.\end{aligned}$$

Combination Finally, we can combine all these steps to get the transfer function for one abstract transition. We can combine the functions by composing all of the before mentioned transfer functions.

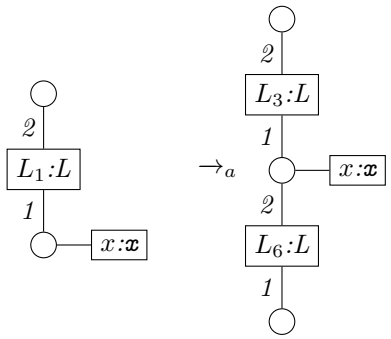
Definition 4.7 (Combined Transfer Function)
Let f *be an abstract transition in an abstract HRG transition system* TS_a *over* $G \in HRG_\Sigma$ *with*

$$\begin{aligned}f &= \big(\langle l, H\rangle, H_m \xrightarrow{\mu} H'_m, \langle l', H'\rangle\big),\\ H &\Rightarrow_{\gamma s} H_m \xrightarrow{\mu} H'_m \Leftarrow_{\alpha s} H'.\end{aligned}$$

The transfer function $\dot{\varphi}_f \in Assigns_{Rel} \mapsto Assigns_{Rel}$ *is*

$$\dot{\varphi}_f = \dot{\varphi}^a_{\alpha s} \circ \dot{\varphi}^o_{(H_m, H'_m)} \circ \dot{\varphi}^m_{\gamma s}. \tag{4.4}$$

Example *Now we combine the last three examples into* $\dot{\varphi}_f$*. We have the transition* $f = \big(\langle l, H\rangle, d, \langle l', H'\rangle\big)$*:*



For the first graph, we use the assign function with

$$\begin{aligned}assign(L_1) &= v_{L_1},\\ \forall e \notin E_H\ assign(e) &= 0.\end{aligned}$$

We apply $\dot{\varphi}_f$ according to this transition step to the function, which then by the last three examples yield the assigning function

$$\begin{aligned} \dot{\varphi}_f(assign)(L_3) &= v_{L_1} - 2, \\ \dot{\varphi}_f(assign)(L_6) &= 2, \\ \dot{\varphi}_f(assign)(x) &= 1, \\ \forall e \notin E_{H'}\ \dot{\varphi}_f(assign)(e) &= 0. \end{aligned}$$

The data flow analysis requires a monotonic transfer function. Thus we also need to prove that $\dot{\varphi}_f$ is indeed monotonic. If the index abstraction rules are monotonic, then $\dot{\varphi}$ will also be monotonic, because monotonicity is complete on composition.

Lemma 4.2
If the index abstraction rules φ_r^a and φ_r^m are monotonic functions, then $\dot{\varphi}$ is also a monotonic function.

Proof See Appendix C.2.

4.1.3 Overall Analysis

The predicate abstraction analysis is the data flow analysis defined on the structures of the last subsections. This data flow analysis will now allow us to approximate an index for every hyperedge in every hypergraph in the abstract transition system.

Definition 4.8 (Predicate Abstraction Analysis)
Let TS_a be the abstract HRG transition system over $G \in HRG_\Sigma$, q the state to analyse and $PROP$ a (G,PROP')-compatible and monotonic directed predicate with relative index set $(I_{Rel}, \sqsubseteq_{Rel})$ and monotonic index abstraction rules $\varphi^a_{(A,R)}$ and $\varphi^m_{(A,R)}$ for all $(A, R) \in P$. We have

$$AP(PROP, c_q) = (Lab, Ex, F, (Assigns_{Rel}, \dot{\sqsubseteq}_{Rel}), \iota, \dot{\varphi})$$

as

- *(Lab, Ex, F) is the flow graph of TS_a that untangles q (Definition 4.1),*
- *$(Assigns_{Rel}, \dot{\sqsubseteq}_{Rel})$ is the assigning domain for $(I_{Rel}, \sqsubseteq_{Rel})$ over the hyperedges $\bigcup\{E_H \mid \langle l, H\rangle \in Stats_a\}$ (Definition 4.2),*
- *the extremal value ι (Definition 4.3),*
- *and the transfer functions $\dot{\varphi}(f) = \dot{\varphi}_f$ (Definition 4.7).*

4.1.4 Widening

We do not enforce our domain to have only stabilising ascending chains. Thus we have to enforce that every ascending chain is transformed into a stabilising ascending chain with other techniques. To archive this, we use a technique

called widening. We introduce a simple widening operator that has a threshold set K. When an element reaches the threshold, the element is set directly to $\top$. If we did not reach the threshold, we use the least upper bound to combine two elements. Whether all ascending chains are transformed into a stabilising ascending chain is depending on the choice of K. Thus, we would need to prove that K yields a valid widening operator. The further results of this thesis do not require to use this widening operator, and any widening operator can be used here.

Definition 4.9 (Widening)
We define $\nabla \in (I_{Rel} \times I_{Rel}) \mapsto I_{Rel}$ *for some* K *to be*

$$i_1 \nabla_K i_2 = \begin{cases} \bigsqcup_{Rel}\{i_1, i_2\} & \text{if } \exists k \in K : i_2 \sqsubseteq_{Rel} k \\ \top & \text{else.} \end{cases} \tag{4.5}$$

The pointwise application of ∇_K *on every hyperedge, which we call* $\dot{\nabla}_K \in (Assigns_{Rel} \times Assigns_{Rel}) \mapsto Assigns_{Rel}$*, is*

$$(a_1 \dot{\nabla}_K a_2)(e) = a_1(e) \nabla_K a_2(e). \tag{4.6}$$

Example *Let the relative index set be* $(\mathbb{N}_{Rel}, \leq_{Rel})$ *from Example 3.3. We use* $K = \{(vars, z) \in \mathbb{N}_{Rel} \mid z = 1\}$ *for the widening operator* ∇_K*. If any element has a* z *value above* 1*, it is widened to* ∞ *by* ∇_K*. We have that*

$$\begin{aligned} v_e + 1 \nabla_K 1 &= v_e + 1, \\ v_e + 1 \nabla_K 2 &= \infty, \\ v_e + 1 \nabla_K v_{e'} + 1 &= v_e + v_{e'} + 1, \\ (v_e + 1 \nabla_K v_{e'} + 1) \nabla_K 2 &= \infty. \end{aligned}$$

4.2 Soundness of Overapproximating Predicates

We now want to prove that this data flow analysis is sound. We understand soundness as a form of language inclusion. If the hypergraph of any concrete state c satisfies the current conditions (i.e. $PROP(i)$), then we can take any concrete path starting in c and follow it on the abstract transition system, and the hypergraphs of the concrete states will also satisfy the current condition assigned by the analysis. We prove the soundness after every step in the abstract transition system first and then combine this to all paths. If every single step is sound in this sense, we can also conclude that every path (without repetition of q) in the transition system is sound in this sense.

To give a meaning to hypergraphs that have indices assigned to their hyperedges, we define a similar notation to $L \cap PROP$. We interpret $L_G(H) \cap PROP(assign)$ a pointwise application of $PROP$ with $assign$ on every hyperedge in H.

Definition 4.10 (Filtering with Index Assignments)
We define the filtering of a hypergraph $H \in \mathcal{H}$ with an $assign \in E \mapsto I$ and a predicate $PROP$ as

$$\begin{aligned} &L_G(H) \cap PROP(assign) \\ &= \{H_T \in L_G(H) \mid \forall e \in E_H \ PROP(assign(e))(H_T(e))\} \end{aligned} \tag{4.7}$$

Let us now consider a transition $\langle l, H_T\rangle \to_c \langle l', H'_T\rangle$ of a concrete hypergraph transition system TS_c. Let a corresponding abstract hypergraph transition in an abstract HRG transition system of TS_c be $\langle l, H\rangle \to_a \langle l', H'\rangle$, i.e. $H_T \in L_G(H)$ and $H'_T \in L_G(H')$. If the concrete hypergraph H_T is in the language $L_G(H) \cap PROP(assign_l)$ where $assign_{\langle l', H'_T\rangle}$ is the solution to the data flow analysis for label $\langle l', H'_T\rangle$, then after the abstract and concrete transition the concrete hypergraph H'_T should be in the language $L_G(H') \cap PROP(\varphi(assign))$. If we prove this, we will then extend this to the abstract path

$$\langle l, H_T\rangle \to_c^{(l,H)\pi(l',H')} \langle l', H'_T\rangle$$

where we again want that $H'_T \in L_G(H') \cap PROP(\varphi(assign_{\langle l', H'_T\rangle}))$.

If after reaching the start state q again (in the label q_f), and the solution $assign_{q_f}$ for label q_f assigns a value for any hyperedge e is strictly smaller then the value of ι, we would have found a loop that would yield a strictly descending chain of indices for the hyperedge e. By the terminating property of $PROP$, we can then conclude, that this state c_q can only be visited finitely often in this transition system since the language will at some time reach the empty language.

Theorem 4.3 (Sound Predicate Abstraction)
Let $PROP$ be a predicate and q a state for $AP(PROP, q)$. Furthermore, let $assign_l = fix(\Phi_{AP(PROP,q)})(l)$ be the values of a fixpoint for the data flow analysis $AP(PROP, q)$. For every variable assignment $\nu \in \mathcal{V}$ we have that if

$$\begin{aligned} &\langle loc, H_T\rangle \to_c^{\langle loc,H\rangle\pi\langle loc',H'\rangle} \langle loc', H'_T\rangle \\ &H_T \in L_G(H) \cap PROP([\![assign_{\langle loc,H\rangle}]\!]_\nu) \end{aligned}$$

where π is a path and does not contain q. Then we have that

$$H'_T \in L_G(H') \cap PROP([\![assign_{\langle loc',H'\rangle}]\!]_\nu). \tag{4.8}$$

We split the proof of this theorem in several proofs. We first prove the soundness of $\dot{\varphi}^m$, then $\dot{\varphi}^o$ and lastly $\dot{\varphi}^a$. In the end, we combine them to φ and conclude that a solution for $\Phi_{AP(PROP,q)}$ has to be sound to be a solution and thus the theorem holds.

4.2.1 Soundness of Materialisation

For the first soundness proof, we show that materialisation is language inclusive (i.e. generates only a superset of hypergraphs) even if the language is filtered to a property $PROP$ and a function $assign$ if we apply φ^m to $assign$ after the forward derivation. If the language is not filtered, this holds by the decomposition of the abstract transition (see Definition 2.20). Since we filter the language to $PROP$ and $assign$, we need to adjust $assign$ to the new hypergraph.

Lemma 4.4
For (G,PROP')-compatible and monotonic directed predicate PROP with a grammar $G \in HRG_\Sigma$, we have that for all $assign \in Assigns_{Rel}$ and all variable assignments $\nu \in \mathcal{V}$ that

$$\begin{aligned} &\forall H \Rightarrow_{\gamma s} H' \; \forall H_T \in L(H') : \\ &H_T \in L_G(H) \cap PROP([\![assign]\!]_\nu) \\ &\Rightarrow H_T \in L_G(H') \cap PROP([\![\dot{\varphi}^m_{\gamma s}(assign)]\!]_\nu) \end{aligned} \tag{4.9}$$

Proof Consider any derivation $H \Rightarrow_{rs} H'$. We first show that theorem for every direct derivation $H \Rightarrow_{(\xi,p)} H'$ and then apply it to derivations with arbitrary length. Let $H \Rightarrow_{(\xi,p)} H'$ be a direct derivation and $p = (A, R)$, then:

$$\begin{aligned} &H \Rightarrow_{(\xi,(A,R))} H' \wedge H_T \in L(H') \wedge \\ &\forall e \in E_H \; PROP([\![assign]\!]_\nu(e))(H_T(e)) \\ \Leftrightarrow &\begin{cases} H \Rightarrow_{(\xi,p)} H' \wedge H_T \in L(H') \wedge \\ \forall e \in E_H \setminus \{\xi\} \; PROP([\![assign]\!]_\nu(e))(H_T(e)) \wedge \\ PROP([\![assign]\!]_\nu(\xi))(H_T(\xi)) \end{cases} && \text{splitting } \xi \\ \Rightarrow &\begin{cases} H \Rightarrow_{(\xi,p)} H' \wedge H_T \in L(H') \wedge \\ \forall e \in E_H \setminus \{\xi\} \; PROP([\![assign]\!]_\nu(e))(H_T(e)) \wedge \\ \exists j \sqsupseteq [\![assign]\!]_\nu(\xi) \; PROP'(j)([\![\varphi^m_p(assign(\xi))]\!]_\nu, R) \wedge \\ \forall e' \in E_R \; PROP([\![\varphi^m_p(assign(\xi))]\!]_\nu(e'))(H_T(e')) \end{cases} && \text{by (3.12)} \\ \Rightarrow &\begin{cases} H \Rightarrow_{(\xi,p)} H' \wedge H_T \in L(H') \wedge \\ \forall e \in E_H \setminus \{\xi\} \; PROP([\![assign]\!]_\nu(e))(H_T(e)) \wedge \\ \forall e' \in E_R \; PROP([\![\dot{\varphi}^m_{(\xi,p)}(assign(\xi))]\!]_\nu(e'))(H_T(e')) \end{cases} && \text{remove } PROP' \\ \Leftrightarrow &\begin{cases} H_T \in L(H') \wedge \\ \forall e \in E_{H'} \; PROP([\![\dot{\varphi}^m_{(\xi,p)}(assign)]\!]_\nu(e))(H_T(e)) \end{cases} && \text{combining sets} \end{aligned}$$

Therefore for one step we have:

$$\begin{aligned} &\{H_T \in L(H') \mid H \Rightarrow_{(\xi,p)} H' \wedge \forall e \in E_H \; PROP([\![assign]\!]_\nu(e))(H_T(e))\} \\ \subseteq &\{H_T \in L(H') \mid \forall e \in E_{H_i} \; PROP([\![\dot{\varphi}^m_{(\xi,p)}(assign)]\!]_\nu(e))(H_T(e))\}. \end{aligned}$$

Inductive application yields overall

$$\begin{aligned} &\{H_T \in L(H') \mid H \Rightarrow_{\gamma s} H' \wedge \forall e \in E_H \; PROP([\![assign]\!]_\nu(e))(H_T(e))\} \\ \subseteq &\{H_T \in L(H') \mid \forall e \in E_{H_i} \; PROP([\![\dot{\varphi}^m_{\gamma s}(assign)]\!]_\nu(e))(H_T(e))\}. \end{aligned}$$

■

4.2.2 Soundness of Hypergraph Operations

Since concrete hypergraph operations can only change terminal hyperedges and $PROP$ with assigns as in the analysis only filters the nonterminal hyperedges, we can also assign concrete indices to changed hyperedges. Either a hyperedge e is not changed, and thus for a concrete hypergraph H_T, $PROP(assign(e))(H_T(e))$ has to hold before and after or it has changed, and we remove the index or assign a new concrete index.

Lemma 4.5
For a (G,PROP')-compatible and monotonic directed predicate PROP and a concrete hypergraph operation $\mu \in \mathcal{H}_\Sigma \mapsto \mathcal{H}_\Sigma$ we have that for all mappings $assign \in Assigns_{Rel}$ and all variable assignments $\nu \in \mathcal{V}$ that if

$$\begin{gathered} H \xrightarrow{\mu} H', \\ \langle loc, H_T \rangle \to_c \langle loc', H'_T \rangle, \\ H_T \in L_G(H) \cap PROP([\![assign]\!]_\nu), \\ H'_T \in L(H') \end{gathered} \tag{4.10}$$

it holds that

$$H'_T \in L_G(H') \cap PROP([\![\dot{\varphi}^o_{(H,H')}(assign)]\!]_\nu). \tag{4.11}$$

Proof We assume the opposite:

$$H'_T \notin L_G(H') \cap PROP([\![\dot{\varphi}^o_{(H,H')}(assign)]\!]_\nu).$$

Since $H'_T \in L(H')$ we then have for some $e \in E_{H'}$ that

$$\neg PROP([\![\dot{\varphi}^o_{(H,H')}(assign)]\!]_\nu)(H'_T(e)).$$

We make a case distinction whether e is part of the graph H or not:

- If $e \in E_H$ then $H_T \notin L_G(H) \cap PROP([\![assign]\!]_\nu)$ which contradicts the assumption.
- If $e \notin E_H$ then by construction in Definition 4.5

  $$PROP([\![\dot{\varphi}^o_{(H,H')}(assign)]\!]_\nu)(e^\bullet),$$

  which contradicts the assumption that

  $$\neg PROP([\![\dot{\varphi}^o_{(H,H')}(assign)]\!]_\nu)(H'_T(e))$$

  because of $label_{H'}(e) \in T$ and therefore $e^\bullet = H'_T(e)$.

Thus for all $e \in E_{H'}$ we have $PROP([\![\dot{\varphi}^o_{(H,H')}(assign)]\!]_\nu)(H'_T(e))$ and therefore

$$H'_T \in L_G(H') \cap PROP([\![\dot{\varphi}^o_{(H,H')}(assign)]\!]_\nu).$$

■

4.2.3 Soundness of Abstraction

The soundness proof for the abstraction step is similar to materialisation. Every abstraction process is a sequence of backwards derivations. Since every backward derivation $H \Leftarrow H'$ yields a language $L(H')$ that is a superset of the language $L(H)$, every concrete hypergraph remains in the language.

Lemma 4.6
For a (G,PROP')-compatible and monotonic directed predicate PROP, we have that for all $assign \in Assigns_{Rel}$ and all variable assignments $\nu \in \mathcal{V}$ that

$$\begin{gathered} \forall H \Leftarrow_{\alpha s} H' \\ L_G(H) \cap PROP([\![assign]\!]_\nu) \subseteq L_G(H') \cap PROP([\![\dot{\varphi}^a_{\alpha s}(assign)]\!]_\nu). \end{gathered} \tag{4.12}$$

PROOF Consider any backwards derivation $H \Leftarrow_{\alpha s} H'$. We first show that theorem for every direct derivation $H \Leftarrow_{(\xi,p)} H'$ and then apply it to derivations with arbitrary length. Let $H \Leftarrow_{(\xi,p)} H'$ be a direct derivation and $p = (A, R)$, then:

$$
\begin{aligned}
&\left\{ \begin{array}{l} H \Leftarrow_{(\xi,p)} H' \wedge H_T \in L(H) \wedge \\ \forall e \in E_H \; PROP([\![assign]\!]_\nu(e))(H_T(e)) \end{array} \right. \\
\Leftrightarrow &\left\{ \begin{array}{l} H \Leftarrow_{(\xi,p)} H' \wedge H_T \in L(H) \wedge \\ \forall e \in E_H \setminus E_R \; PROP([\![assign]\!]_\nu(e))(H_T(e)) \wedge \\ \forall e \in E_R \; PROP([\![assign]\!]_\nu(e))(H_T(e)) \end{array} \right. && \text{splitting } R \\
\Rightarrow &\left\{ \begin{array}{l} H \Leftarrow_{(\xi,p)} H' \wedge H_T \in L(H) \wedge \\ \forall e \in E_H \setminus E_R \; PROP([\![assign]\!]_\nu(e))(H_T(e)) \wedge \\ \exists j \sqsubseteq [\![\varphi_p^a(assign)]\!]_\nu \; PROP'(j)([\![assign]\!]_\nu, R) \wedge \\ \forall e \in E_R \; PROP([\![assign]\!]_\nu(e))(H_T(e)) \end{array} \right. && \text{by (3.14)} \\
\Rightarrow &\left\{ \begin{array}{l} H \Leftarrow_{(\xi,p)} H' \wedge H_T \in L(H) \wedge \\ \forall e \in E_H \setminus E_R \; PROP([\![assign]\!]_\nu(e))(H_T(e)) \wedge \\ PROP([\![\varphi_p^a(assign)]\!]_\nu)(H_T(\xi)) \end{array} \right. && \text{by (3.13)} \\
\Leftrightarrow &\left\{ \begin{array}{l} H \Leftarrow_{(\xi,p)} H' \wedge H_T \in L(H) \wedge \\ \forall e \in E_{H'} \setminus \{\xi\} \; PROP([\![assign]\!]_\nu(e))(H_T(e)) \wedge \\ PROP([\![\varphi_p^a(assign)]\!]_\nu)(H_T(\xi)) \end{array} \right. && \text{change to } E_{H'} \\
\Leftrightarrow &\left\{ \begin{array}{l} H \Leftarrow_{(\xi,p)} H' \wedge H_T \in L(H) \wedge \\ \forall e \in E_{H'} \; PROP([\![\dot\varphi_{(\xi,p)}^a(assign)]\!]_\nu(e))(H_T(e)) \end{array} \right. && \text{combining sets} \\
\Leftrightarrow &\left\{ \begin{array}{l} H_T \in L(H') \wedge \\ \forall e \in E_{H'} \; PROP([\![\dot\varphi_{(\xi,R)}^a(assign)]\!]_\nu(e))(H_T(e)) \end{array} \right. && L(H) \subseteq L(H')
\end{aligned}
$$

Thus one step is proven:

$$L_G(H) \cap PROP([\![assign]\!]_\nu) \subseteq L_G(H') \cap PROP([\![\dot\varphi_{(\xi,p)}^a(assign)]\!]_\nu).$$

By applying this inductively, it yields the lemma

$$L_G(H) \cap PROP([\![assign]\!]_\nu) \subseteq L_G(H') \cap PROP([\![\dot\varphi_{\alpha s}^a(assign)]\!]_\nu). \qquad \blacksquare$$

4.2.4 Soundness of Combination

Now we can combine $\dot\varphi^m$, $\dot\varphi^o$ and $\dot\varphi^a$ to $\dot\varphi$. Then $\dot\varphi$ can be used to overapproximate the predicate $PROP$ for every single abstraction step.

Lemma 4.7
Let TS_a be an abstract HRG transition system for a concrete hypergraph transition system TS_c. We have for every single abstract transition step that if

$$\begin{array}{c} \langle loc, H_T \rangle \rightarrow_c^{\langle loc, H \rangle \langle loc', H' \rangle} \langle loc', H'_T \rangle, \\ H_T \in L_G(H) \cap PROP([\![assign]\!]_\nu) \end{array} \tag{4.13}$$

it holds that

$$H'_T \in L_G(H') \cap PROP([\![\dot\varphi(assign)]\!]_\nu). \tag{4.14}$$

PROOF Every abstract transition in $\rightarrow_a$ is decomposable into a materialisation, a concrete hypergraph operation and an abstraction. We have for the abstract transition $\langle l, H\rangle \xrightarrow{H_m \xrightarrow{\mu} H'_m}_a \langle l', H'\rangle$:

$$H \Rightarrow_{\gamma s} H_m \xrightarrow{\mu} H'_m \Leftarrow_{\alpha s} H'.$$

By combining the three functions $\dot{\varphi}^m$, $\dot{\varphi}^o$ and $\dot{\varphi}^a$:

$$\begin{aligned}
&H_T \in L_G(H) \cap PROP(\llbracket assign \rrbracket_\nu) \\
\Rightarrow &H_T \in L_G(H_m) \cap PROP(\llbracket \dot{\varphi}^m_{\gamma s}(assign) \rrbracket_\nu) && \text{by Lemma 4.4} \\
\Rightarrow &H'_T \in L_G(H'_m) \cap PROP(\llbracket \dot{\varphi}^o_{(H_m, H'_m)}(\dot{\varphi}^m_{\gamma s} assign)) \rrbracket_\nu) && \text{by Lemma 4.5} \\
\Rightarrow &H'_T \in L_G(H') \cap PROP(\llbracket \dot{\varphi}^a_{\alpha s}(\dot{\varphi}^o_{(H_m, H'_m)}(\dot{\varphi}^m_{\gamma s} assign))) \rrbracket_\nu) && \text{by Lemma 4.6} \\
\Rightarrow &H'_T \in L_G(H') \cap PROP(\llbracket \dot{\varphi}(assign) \rrbracket_\nu)
\end{aligned}$$

■

Next, we can finally prove Theorem 4.3 by using Lemma 4.7. By the monotonic data flow analysis, every $assign_l$ has to satisfy the equation system. Together with the abstraction property of the abstract transition system TS_a, we then can conclude that the solution for all paths is sound and therefore also the solutions for the equation system.

PROOF (THEOREM 4.3) We prove this by induction over the path

$$\langle loc, H\rangle \pi \langle loc', H'\rangle.$$

IB: The induction base follows directly from Lemma 4.7. If $\langle loc, H\rangle$ is q, we use $assign_{q_0}$, if $\langle loc', H'\rangle$ is q, we use $assign_{q_f}$.

IH: Assume the theorem holds for a fixed but arbitrary π.

IS: For the induction step we have to prove the theorem for

$$\begin{gathered}
\overbrace{\langle loc, H\rangle}^{a} \rightarrow_a{}^{a\pi a'} \overbrace{\langle loc', H'\rangle}^{a'} \rightarrow_a^{a'a''} \overbrace{\langle loc'', H''\rangle}^{a''} \\
\langle loc, H_T\rangle \rightarrow_c{}^{a\pi a'} \langle loc', H'_T\rangle \rightarrow_c{}^{a',a''} \langle loc'', H''_T\rangle
\end{gathered}$$

Applying the induction hypothesis yields

$$H'_T \in L_G(H') \cap PROP(\llbracket \dot{\varphi}_{a\pi a'}(assign_a) \rrbracket_\nu)$$

where $a' \neq q$. Then by Lemma 4.7, we have

$$H''_T \in L_G(H'') \cap PROP(\llbracket \dot{\varphi}_{a\pi a' a''}(assign_a) \rrbracket_\nu).$$

By inductively applying Lemma 4.7 we thus have for every path π which does not contain q that if

$$\begin{gathered}
\langle loc, H_T\rangle \rightarrow_c{}^{\langle loc, H\rangle \pi \langle loc', H'\rangle} \langle loc', H'_T\rangle \\
H_T \in L_G(H) \cap PROP(\llbracket assign \rrbracket_\nu)
\end{gathered}$$

then

$$H'_T \in L_G(H') \cap PROP([\![\dot{\varphi}_{\langle loc,H\rangle\pi\langle loc',H'\rangle}(assign)]\!]_\nu).$$

$\mathrm{MOP}(a) = \dot{\bigsqcup}_{Rel}\{\dot{\varphi}_\pi(\iota) \mid \pi \in \mathrm{Path}(a)\}$ is the meet over all paths solution for a (see Appendix B.9). The meet over all paths solution guarantees

$$\dot{\varphi}_\pi(\iota) \mathrel{\dot{\sqsubseteq}}_{Rel} \mathrm{MOP}(a) \mathrel{\dot{\sqsubseteq}}_{Rel} \mathrm{fix}(\Phi_{AP(PROP,c_q)})(a).$$

By the monotonic directed and the relative index property of $PROP$, we finally have that for all $\pi \in \mathrm{Path}(l)$

$$\begin{aligned} H'_T \in & L_G(H') \cap PROP([\![\dot{\varphi}_\pi(\iota)]\!]_\nu) \\ \subseteq & L_G(H') \cap PROP([\![\mathrm{MOP}(a)]\!]_\nu) \\ \subseteq & L_G(H') \cap PROP([\![\mathrm{fix}(\Phi_{AP(PROP,q)})(a)]\!]_\nu) \\ = & L_G(H') \cap PROP([\![assign_a]\!]_\nu). \end{aligned}$$

■

Chapter 5

Proving Termination

In the previous chapter, we have learned how to approximate the indices for every hyperedge in an abstract hypergraph transition system. Additional, we can untangle a state q, i.e. we can transform the transition system in such a way that the state q changes to the states q_0 and q_f, where q_0 has only outgoing and q_f has only ingoing transitions. By solving the data flow analysis from Definition 4.8, we can calculate a solution that approximates all indices. We will write $solution_q$ for the solution calculated for a state q. If a hyperedge's index is descending from $solution_{q_0}$ to $solution_{q_f}$, we say that an index in the solution is descending. We can now prove that if an index in the solution is descending, then we can construct an infinite strict descending chain of indices. Since the chain is strictly descending, at some point the index will be terminating. However, concrete states do not have terminating indices, and therefore some state-index combinations will not correspond to a concrete state. Thus this chain can also not correspond to a path in the concrete transition system. It may happen that no index in the solution is descending, although a combination of indices is descending. We could consider two lists that regularly switch their head variables and additionally, decrease in size. The size of the first list j is decreasing to $j-1$ and the size of the second list i is decreasing to $i-1$. The program will terminate, as the lists are decreasing, but as we switch these lists, we can not compare j with $i-1$ and i with $j-1$. We could, however, if we have a suitable combination operator, combine both indices such that we can compare $j+i$ with $i+j-2$, which yields a decreasing index. We will take a look at these combination operators afterwards. Finally, we will show that proving the termination argument described before (which we call partial-termination) of critical abstract states is sufficient to prove the termination of the whole program.

5.1 Decreasing Chains using one Hyperedge

We now prove that if the solution of the data flow analysis $AP(PROP, q)$ yields a descending index (as mentioned before), then every infinite path with this descending index has no corresponding infinite path in the concrete transition system. In $AP(PROP, q)$ we have untangled q so that we can analyse the behaviour after every single iteration and thus can analyse how the index changes

after revisiting the state by comparing q_0 with q_f. It is sufficient if the index of a hyperedge e is descending after one iteration. We require, however, that the hyperedge e is nonterminal. If e would be terminal, in q_0 the hyperedge would not have a relative index with a variable. Without a variable, we cannot assign the hyperedge an arbitrary index, which we require for the proof.

Theorem 5.1 (Proving Termination for a Critical State) *Let TS_a be an abstract HRG transition system of TS_c, $q = \langle loc, H\rangle \in States_a$ a state and PROP a terminating predicate (Definition 3.5). If there exists $e \in E_H$ with $label(e) \in N$ and for the q untangled TS_a we have*

$$fix(\Phi_{AP(PROP,q)})(q_0)(e) \sqsupset_{Rel} fix(\Phi_{AP(PROP,q)})(q_f)(e), \tag{5.1}$$

then there exists no $H^0_T \in \mathcal{H}_T$ with a path $q, \pi_1, q, \pi_2, q, \pi_3 \cdots \in States_a^\omega$ where $\pi_i \in (States_a \setminus \{q\})^*$, such that

$$\langle loc, H^0_T\rangle \rightarrow_c{}^{q,\pi_1,q} \langle loc, H^1_T\rangle \rightarrow_c{}^{q,\pi_2,q} \langle loc, H^2_T\rangle \rightarrow_c{}^{q,\pi_3,q} \dots \tag{5.2}$$

Theorem 5.1 states that if the descending index property holds for the q untangled transition system, then the infinite loops of q do not have corresponding paths in the concrete transition system. We call this the partial termination property of q: visiting q infinitely often has to lead to termination. As an infinite path may not visit q infinitely often, we still have to prove the partial termination for all critical states.

Example *Let MLS be the predicate from Example 3.2 with its relative index $(\mathbb{Z}_{Rel}, \leq_{Rel})$ from Example 3.3 and q a state. We defined MLS as most list size of a list graph. Every graph that is not a list does not have an index, and every graph that is a list does have an index that is higher or equal than its actual index. We assume that for a hyperedge e with $label(e) \in N$ we have that*

$$fix(\Phi_{AP(MLS,q)})(q_0)(e) = v_e,$$
$$fix(\Phi_{AP(MLS,q)})(q_f)(e) = v_e - 1.$$

Let $H^0_T \in \mathcal{H}_T$ with path $q, \pi_1, q, \pi_2, q, \pi_3 \cdots \in States_a^\omega$, with $\pi_i \in (States_a \setminus \{q\})^*$ such that

$$\langle loc, H^0_T\rangle \rightarrow_c{}^{q,\pi_1,q} \langle loc, H^1_T\rangle \rightarrow_c{}^{q,\pi_2,q} \langle loc, H^2_T\rangle \rightarrow_c{}^{q,\pi_3,q} \dots$$

Assume that $MLS(2)(H^1_T(e))$.

1. *Let $\nu_0(v_e) = 2$. We have by Theorem 4.3 that*

   $$MLS(\nu_0(v_e) - 1)(H^1_T(e)).$$

2. *Let $\nu_1(v_e) = \nu_0(v_e) - 1 = 1$. We have again by Theorem 4.3 that*

   $$MLS(\nu_1(v_e) - 1)(H^2_T(e)).$$

   However, that would be evaluated to $MLS(0)(H^2_T(e))$, which contradicts the termination property of MLS.

Thus, there exists no H^0_T with these infinite paths π such that $MLS(2)(H^0_T(e))$. We could repeat this for every index instead of just 2 to prove that there is no such hypergraph and path combination, which concludes that there is not infinite abstract path containing infinitely often q.

In order to prove Theorem 5.1, we will use a sequence of variable assignments $\nu_0, \nu_1, \nu_2, \dots$ where ν_n keeps track of the index after n iterations. These variable assignments ν_{n+1} are defined recursively as the value of the evaluation of the solution of the final label q_f from the data flow analysis with the last variable assignment ν_n. For the first variable assignment ν_0, we use for every hyperedge e in the abstract graph H some index i such that the concrete joint-embedding $H_T^0(e)$ satisfies $PROP(i)$. We then want to prove that the variable assignments are sound. The variable assignments are sound if for the graphs H_T^n and the hyperedges $e \in E_H$ the joint-embedding $H_T^n(e)$ satisfies $PROP(\nu_n(e))$.

We will concretise variable assignments after we reached q again to assign it to the variables for the next iteration. The variable assignment of the next iteration is the assignment that assigns every variable v_e for the hyperedge e the value of the last iteration $\phi(e)$. We say that ι is the function that maps every hyperedge to its variable (we defined ι in Definition 4.3), i.e. the value of the state q_0 and that ϕ is a solution for the state q_f. As we can not compose ϕ with ν directly to $\nu \circ \phi$, we define the notation of concretisation via variable assignment to combine these functions. We define $\nu \circ \phi$ as the concretisation of ν with respect to ϕ. The concrete value of the extremal value ι using $\nu \circ \phi$ is $[\![\iota(e)]\!]_{\nu\circ\phi}$ and equals the value $[\![\phi(e)]\!]_\nu$. As ι assigns every variable to exactly one hyperedge, this defines a unique variable assignment.

Definition 5.1 (Concretisation via Variable Assignment)
Let ι be the mapping from hyperedges to their corresponding variables (see Definition 4.3). For a mapping $\phi \in Assigns_{Rel}$ we write the concretisation of ϕ with ν as $\nu \circ \phi \in \mathcal{V}$:

$$[\![\iota]\!]_{\nu\circ\phi} = [\![\phi]\!]_\nu$$

Lemma 5.2
Let TS_a be the abstract HRG transition system of TS_c, $q = \langle loc, H\rangle \in States_a$, $\pi = q, \pi_1, q, \pi_2, q, \pi_3, q \cdots \in States_a^\omega$ a path where $\pi_i \in (States_a \setminus \{q\})^*$ with

$$\langle loc, H_T^0\rangle \rightarrow_c{}^{q,\pi_1,q} \langle loc, H_T^1\rangle \rightarrow_c{}^{q,\pi_2,q} \langle loc, H_T^2\rangle \rightarrow_c{}^{q,\pi_3,q} \dots \tag{5.3}$$

we have for ν_i defined as

$$\begin{aligned} \nu_0(v_e) &= i \text{ such that } PROP(i)(H_T^0(e)), \\ \nu_{n+1} &= \nu_n \circ \mathit{fix}(\Phi_{AP(PROP,q)})(q_f) \end{aligned} \tag{5.4}$$

that

$$\forall n \in \mathbb{N}\ H_T^n \in L_G(H) \cap PROP([\![\iota]\!]_{\nu_n}). \tag{5.5}$$

Proof ι is the relative value that gives every nonterminal hyperedge e the value $\nu(v_e)$, i.e. if $label(e) \in N$ then $[\![\iota(e)]\!]_\nu = \nu(v_e)$. We write $\mathrm{fix}(\Phi_{AP(PROP,q)})(l)$ as $solution_l$ to shorten the proof. We prove by induction over n that for $\nu_n \in \mathcal{V}$ the Lemma holds:

IB:

$$\begin{aligned} & H_T^0 \in L(H) && \text{by premise} \\ \Rightarrow & H_T^0 \in L(H) \wedge \forall e \in E_H\ PROP([\![\iota]\!]_{\nu_0}(e))(H_T^0(e)) && \text{by construction} \\ \Rightarrow & H_T^0 \in L_G(H) \cap PROP([\![\iota]\!]_{\nu_0}) \end{aligned}$$

IH: For a fixed but arbitrary $n \in \mathbb{N}$ it holds that $H_T^n \in L_G(H) \cap PROP(\llbracket \iota \rrbracket_{\nu_n})$.

IS:

$$
\begin{aligned}
& H_T^n \in L_G(H) \cap PROP(\llbracket \iota \rrbracket_{\nu_n}) && \text{by Hypothesis} \\
\Rightarrow & H_T^n \in L_G(H) \cap PROP(\llbracket solution_{q_0} \rrbracket_{\nu_n}) && \text{as } assign_{q_0} = \iota \\
\Rightarrow & H_T^{n+1} \in L_G(H) \cap PROP(\llbracket solution_{q_f} \rrbracket_{\nu_n}) && \text{by Theorem 4.3} \\
\Rightarrow & H_T^{n+1} \in L_G(H) \cap PROP(\llbracket \iota \rrbracket_{\nu_n \circ solution_{q_f}}) && \text{by Definition 5.1} \\
\Rightarrow & H_T^{n+1} \in L_G(H) \cap PROP(\llbracket \iota \rrbracket_{\nu_{n+1}}) && \text{by Definition of } \nu_{n+1} \quad \blacksquare
\end{aligned}
$$

We remark that ν_m generates a ranking function. If the index is descending after each iteration for q in the hyperedge e, we have that $\nu_0(v_e) \sqsupset \nu_1(v_e) \sqsupset \ldots$, which describes the ranking function $r(\langle loc, H_T^n \rangle) = \nu_n(v_e)$. Usually, ranking functions also require a well-founded domain, that is a domain for which all descending chains are finite. Although our domain is not necessary well-founded, which we would require for a ranking function, we can remove all terminating indices to receive a well-founded domain.

We can now prove the Theorem 5.1. We first prove that ν_n indeed forms a strictly descending chain. Then we can use this to conclude that a path π that visits c_q infinitely often cannot exist, as for some variable assignment ν_n a terminating index $\nu_n(v)$ is reached, which would make $PROP(\nu_n(v))$ unsatisfiable. This way, we prove that c_q is not part of an infinite abstract path corresponding to a concrete path.

Proof (of Theorem 5.1) We write $solution_l$ as a shorthand form for the solution $\text{fix}(\Phi_{AP(PROP,q)})(l)$. We remark that $label(e) \in N$ and thus the extremal value ι maps e to a relative index that is mapped to the value $\nu(v_e)$. We have that:

$$
\begin{aligned}
& solution_{q_0}(e) \sqsupset_{Rel} solution_{q_f}(e) && \text{by Premise} \\
\Rightarrow & \llbracket solution_{q_0}(e) \rrbracket_{\nu} \sqsupset \llbracket solution_{q_f}(e) \rrbracket_{\nu} && \text{by Condition (3.11)} \\
\Rightarrow & \llbracket solution_{q_0}(e) \rrbracket_{\nu_n} \sqsupset \llbracket solution_{q_f}(e) \rrbracket_{\nu_n} && \nu_n \text{ as in Lemma 5.2} \\
\Rightarrow & \llbracket \iota(e) \rrbracket_{\nu_n} \sqsupset \llbracket \iota(e) \rrbracket_{\nu_n \circ solution_{q_f}} && \nu_n \circ solution_{q_f} \text{ as in Definition 5.1} \\
\Rightarrow & \llbracket \iota(e) \rrbracket_{\nu_n} \sqsupset \llbracket \iota(e) \rrbracket_{\nu_{n+1}} && \nu_{n+1} \text{ as in Lemma 5.2}
\end{aligned}
$$

Thus there exists the infinitely strict descending chain:

$$\llbracket \iota(e) \rrbracket_{\nu_0} \sqsupset \llbracket \iota(e) \rrbracket_{\nu_1} \sqsupset \llbracket \iota(e) \rrbracket_{\nu_2} \sqsupset \ldots$$

$PROP$ is a terminating predicate, thus for some $\llbracket \iota(e) \rrbracket_{\nu_m}$ we that

$$L_G(H) \cap PROP(\llbracket \iota(e) \rrbracket_{\nu_m}) = \emptyset$$

where H is the hypergraph of the state q. We have

$$\langle loc, H_T^0 \rangle \rightarrow_c{}^{q,\pi_1,q} \langle loc, H_T^1 \rangle \rightarrow_c{}^{q,\pi_2,q} \ldots \langle loc, H_T^m \rangle \rightarrow_c{}^{q,\pi_{m+1},q} \ldots$$

as in (5.2). By Lemma 5.2 we have

$$H_T^m \in L_G(H) \cap PROP(\llbracket \iota \rrbracket_{\nu_m}).$$

Together we have the contradiction

$$H_T^m \in L_G(H) \cap PROP(\llbracket \iota \rrbracket_{\nu_m}) = \emptyset. \quad \blacksquare$$

5.2 Decreasing Chains using a Set of Hyperedges

In the previous section, we hoped that a hyperedge has a decreasing index. However, if no hyperedge has a decreasing index, that does not mean that the program is not terminating. Especially if the indices are not comparable, it may be beneficial to combine indices in order to make them comparable. Assume we have, for a hyperedge e, the start index $\iota(e)$, the end index $\phi(e)$ and $\iota(e)$ is not comparable to $\phi(e)$. It can help to combine the values of different hyperedges e and e' such that the combination of these is decreasing.

To combine indices, we use a set function Ω that behaves safely. Safe means on the one hand that if the image of this set function $\Omega(\phi(es))$ becomes a terminating index, also at least one hyperedge $e \in es$ has a terminating index $\phi(e)$. On the other hand, safe means we need to have a property that guarantees the possibility to do the same proof as in the last section. We need to be able to turn ϕ into $\nu \circ \phi$ without loss of precision. We can, however, restrict us to the preconditions for the proof: if $\Omega(\iota(es)) \sqsupset \Omega(\phi(es))$ then the concretisation via variable assignment $\nu \circ \phi$ is also descending $[\![\Omega(\phi(es))]\!]_\nu \sqsupseteq [\![\Omega(\iota(es))]\!]_{\nu\circ\phi}$. If the variable assignment is descending, we can again use a similar proof to create descending chains. We can then use the first property to conclude that at one index of the chain is terminating and thus also a hyperedge has a terminating index. In most cases, the concretisation via variable assignments should not change the index. The idea is that we preserve the index as best as possible by Ω and ideally, that means that the index stays the same.

Definition 5.2 (Safe Set Function)
Let the complete lattice $(I, \sqsubseteq)$ be an index set and $(I_{Rel}, \sqsubseteq_{Rel})$ its relative index set. We denote the pointwise application of a function γ on sets as $\dot{\gamma}$. We call $\Omega \in 2^{I_{Rel}} \mapsto I_{Rel}$ a safe set function on I_{Rel} if and only if for all non-empty, finite sets $irs \subseteq I_{Rel}$

$$\begin{aligned} &\forall \nu \in \mathcal{V} : \\ &[\![\Omega(irs)]\!]_\nu \text{ is terminating} \Rightarrow \exists ir \in irs\ [\![ir]\!]_\nu \text{ is terminating} \end{aligned} \tag{5.6}$$

(5.7)

and for all non-empty, finite sets $es \subseteq E$

$$\begin{aligned} &\forall \phi \in Assigns_{Rel}\ \forall \nu \in \mathcal{V} : \\ &\Omega(i(es)) \sqsupset_{Rel} \Omega(\dot{\phi}(es)) \Rightarrow [\![\Omega(\dot{\phi}(es))]\!]_\nu \sqsupseteq [\![\Omega(i(es))]\!]_{\nu\circ\phi} \end{aligned} \tag{5.8}$$

Example *Let $\Sigma \in 2^{\mathbb{Z}_{Rel}} \mapsto \mathbb{Z}_{Rel}$ be a mapping with*

$$\Sigma(irs) = \sum_{ir \in irs} ir.$$

Σ is a safe set function (see Lemma 6.9). We have then:

$$\begin{aligned} \Sigma(\{v_1 + 5, v_2 + 3\}) &= v_1 + v_2 + 8 \\ \Sigma(\{v_1 + 5, v_2 + 3, v_1\}) &= \infty \\ \Sigma(\{v_1 + 5, -2\}) &= v_1 + 3 \end{aligned}$$

The following theorem is similar to Theorem 5.1. This time, however, we allow a non-empty set of nonterminal hyperedges for our analysis. With a safe set function, we combine the indices to a new index, for which we then can construct a descending chain for any abstract path which visits the state in question infinitely often. Then this descending chain is required to have a terminating index, and as we already saw, a terminating index allows us to prove a contradiction for the abstract path.

Theorem 5.3 (Proving Termination with a Safe Set Function) *Let TS_a be an abstract HRG transition system of TS_c, $q = \langle loc, H\rangle$ a state, $PROP$ a terminating predicate and Ω a safe set function. If there exists $\emptyset \neq es \subseteq E$ with $\forall e \in eslabel(e) \in N$ such that for the q untangled TS_a we have*

$$\Omega(\dot{fix}(\Phi_{AP(PROP,q)})(q_0)(es)) \sqsupset_{Rel} \Omega(\dot{fix}(\Phi_{AP(PROP,q)})(q_f)(es)) \tag{5.9}$$

then there exists no $H_T^0 \in \mathcal{H}_T$ with a path $q, \pi_1, q, \pi_2, q, \pi_3 \cdots \in States_a^\omega$ where $\pi_i \in (States_a \setminus \{q\})^*$ such that

$$\langle loc, H_T^0\rangle \rightarrow_c{}^{q,\pi_1,q} \langle loc, H_T^1\rangle \rightarrow_c{}^{q,\pi_2,q} \langle loc, H_T^2\rangle \rightarrow_c{}^{q,\pi_3,q} \ldots \tag{5.10}$$

Proof (of Theorem 5.3) Remark that $label(e) \in N$ and thus the extremal value ι maps e to a relative index that is mapped to the value $\nu(v_e)$. Furthermore, we use here $solution_q$ to denote the pointwise application of the solution for a state q and i to denote the pointwise application of the extremal value ι. We have that:

$$\begin{aligned}
&\Omega(solution_{q_0}(es)) \sqsupset_{Rel} \Omega(solution_{q_f}(es)) && \text{by Premise}\\
\Rightarrow &[\![\Omega(solution_{q_0}(es))]\!]_\nu \sqsupset [\![\Omega(solution_{q_f}(es))]\!]_\nu && \text{by relative index (3.11)}\\
\Rightarrow &[\![\Omega(solution_{q_0}(es))]\!]_{\nu_n} \sqsupset [\![\Omega(solution_{q_f}(es))]\!]_{\nu_n} && \nu_n \text{ as in Lemma 5.2}\\
\Rightarrow &[\![\Omega(solution_{q_0}(es))]\!]_{\nu_n} \sqsupset [\![\Omega(\iota(es))]\!]_{\nu_n \circ assign_{q_f}} && \text{by Condition (5.8)}\\
\Rightarrow &[\![\Omega(solution_{q_0}(es))]\!]_{\nu_n} \sqsupset [\![\Omega(\iota(es))]\!]_{\nu_{n+1}} && \nu_{n+1} \text{ as in Lemma 5.2}\\
\Rightarrow &[\![\Omega(i(es))]\!]_{\nu_n} \sqsupset [\![\Omega(i(es))]\!]_{\nu_{n+1}} && \text{by Definition 4.3}
\end{aligned}$$

Thus there exists an infinitely strict descending chain:

$$[\![\Omega(i(es))]\!]_{\nu_1} \sqsupset [\![\Omega(i(es))]\!]_{\nu_2} \sqsupset [\![\Omega(i(es))]\!]_{\nu_3} \sqsupset \ldots$$

$PROP$ is a terminating predicate; thus, there exists $n \in \mathbb{N}$ such that the index $[\![\Omega(\iota(es))]\!]_{\nu_n}$ is terminating. Then by condition (5.6) there exists also a hyperedge e such that $[\![\iota(e)]\!]_{\nu_n}$ is termination, which gives us

$$L_G(H) \cap PROP([\![\iota(e)]\!]_{\nu_m}) = \emptyset$$

where H is the hypergraph of the state q. We have

$$\langle loc, H_T^0\rangle \rightarrow_c{}^{q,\pi_1,q} \langle loc, H_T^1\rangle \rightarrow_c{}^{q,\pi_2,q} \ldots \langle loc, H_T^m\rangle \rightarrow_c{}^{q,\pi_{m+1},q} \ldots$$

as in (5.10). By Lemma 5.2 we have

$$H_T^m \in L_G(H) \cap PROP([\![\iota]\!]_{\nu_m}).$$

Together we have the contradiction

$$H_T^m \in L_G(H) \cap PROP([\![\iota]\!]_{\nu_m}) = \emptyset.$$

■

This technique allows us to prove the partial termination of a state in question q. Partial termination means that every path that visits q infinite often does not have a corresponding concrete path. We do this by calculating a relative index for the state in the state of question after one cycle and comparing it to the start value. We also can combine indices using the safe set functions to compare the combined indices before and after one iteration.

Example *Let $e, e' \in E$ and $(\mathbb{Z}_{Rel}, \leq_{Rel})$ the relative index domain from Example 3.3. We consider the assign function ϕ with $\phi(e) = v_{e'} - 2$ and $\phi(e') = v_e + 1$. We can combine $\iota(e)$ with $\iota(e')$ and $\phi(e)$ with $\phi(e')$ by using addition. This yields*

$$\iota(e) + \iota(e') = v_e + v_{e'} >_{Rel} v_e + v_{e'} - 1 = \phi(e) + \phi(e').$$

Assume $\nu_0(v_e) = 2$ and $\nu_0(v_{e'}) = 1$; then, we form the descending chain:

$$[\![v_e]\!]_{\nu_0} + [\![v_{e'}]\!]_{\nu_0} = 2 + 1 > -1 + 2 = [\![v_{e'} - 2]\!]_{\nu_0} + [\![v_e + 1]\!]_{\nu_0}$$
$$[\![v_e]\!]_{\nu_1} + [\![v_{e'}]\!]_{\nu_1} = -1 + 2 > 0 + 0 = [\![v_{e'} - 2]\!]_{\nu_1} + [\![v_e + 1]\!]_{\nu_1}$$

After two iterations, the sum of these indices yield the terminating index 0, and as $\nu_2(v_e) = 0 = \nu_2(v_{e'})$, we have found a path that leads to the terminating index 0 for at least one hyperedge. Remark, however, that we had the terminating index -1 already in the first iteration, but our combination method was not able to detect this terminating index yet.

5.3 Proving Overall Termination

Up to now, we only considered one abstract state q and then proved that there exists no path that has infinitely many states that are abstracted by q. Now we lift this to the question of whether there exists an infinite path. We can prove this by using the set of critical locations from Definition 2.9. We use critical locations to calculate the set of critical abstract states. Then we only need to prove that for every critical abstract state, no infinite concrete paths exists such that infinitely many states are abstracted to this critical abstract state.

Theorem 5.4 (Overall Termination using Partial Termination) *Let TS_c be a hypergraph transition system, $\alpha \in States_c \mapsto States_a$ a hypergraph abstraction function where $States_a$ is finite, and Crit are critical locations for TS_c. TS_c does not have an infinite path if and only if for all abstract states $\langle l, H\rangle \in States_a$ with $l \in Crit$ there exists no infinite path $\pi \in States_c^\omega$ such that*

$$\forall m \in \mathbb{N}\ \exists m' \in \mathbb{N} : \quad m < m' \wedge \alpha(\pi(m')) = \langle l, H\rangle.$$

Proof First, we prove the easy direction: if TS_c does not have any infinite paths π, then especially not any where $\alpha(\phi(m')) = \langle l, H\rangle$ occurs infinitely often.

Now to the other direction. Assume TS_c does have an infinite path σ. Then we have

$$\forall m \in \mathbb{N}\ \exists m' \in \mathbb{N} : \quad m < m' \wedge \sigma(m') = \langle l, H^T\rangle \wedge l \in Crit$$

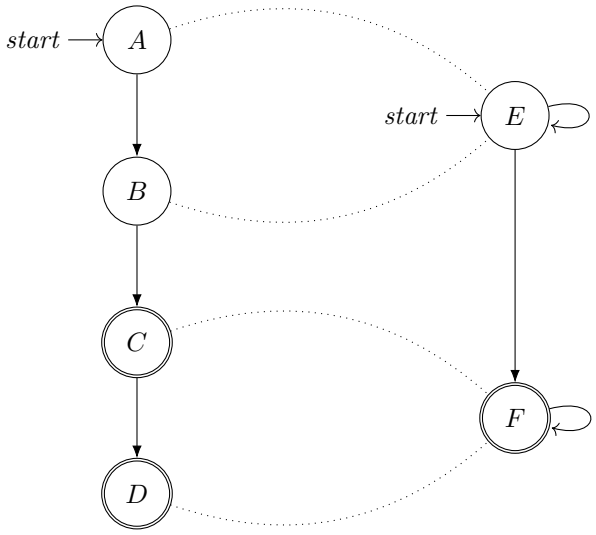
by the set of critical locations *Crit*. As States_a is finite, we have by the pigeonhole principle that there exists at least one abstract state $\langle l, H_a\rangle$ such that

$$\forall m \in \mathbb{N}\ \exists m' \in \mathbb{N} : \quad m < m' \wedge \alpha(\sigma(m')) = \langle l, H_a\rangle \wedge l \in Crit.$$

Then there exists the infinite path σ, where a specific abstract state $\langle l, H_a \rangle$ with $l \in \mathit{Crit}$ occurs infinitely often. ■

Remark that this theorem also holds if States_a is infinite and the set of critical abstract states is finite, as we can then also apply the pigeonhole principle. It does, however, not apply if the set of critical abstract states is infinite. If there are infinitely many critical abstract states, a different abstract state could abstract every critical state in the path. As infinite abstract state spaces are would at least need another abstraction layer to solve, we only consider finite abstract state spaces

Example *Let TS_c be the concrete transition system on the left, and TS_a the abstract transition system of TS_c on the right:*



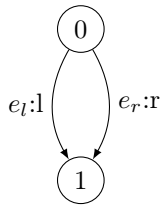
We have $\alpha(A) = \alpha(B) = E$ and $\alpha(C) = \alpha(D) = F$. TS_a is not loop-free; thus, we have infinite paths in TS_a. Every such infinite path π visits either infinitely often E or infinitely often F. Now assume that we have proven that every path in TS_c does not visit infinitely many states c_i with $\alpha(c_i) = E$ or $\alpha(c_i) = F$. Then by Theorem 5.4, TS_c does not have an infinite path.

Chapter 6

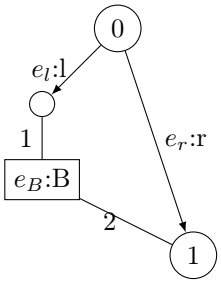
Inequality Predicates

Example 6.1: The grammar Bin-Tree that generates a language of binary trees. The grammar consists of two terminals l and r, one nonterminal B and four production rules.

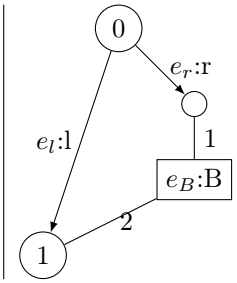
Let Bin-Tree $= (T, N, P)$ with $T = \{l, r\}$, $N = \{B\}$ and P be defined as:

B $\Rightarrow$ 

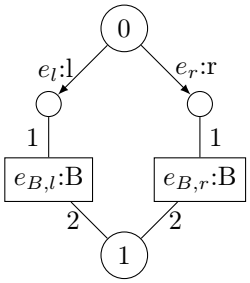
(1)

B $\Rightarrow$ 

(2)



(3)

B $\Rightarrow$ 

(4)

In this chapter, we take a closer look at a particular set of monotonic directed and terminating predicates. These predicates consist, similar to the predicate $MOST\text{-}LIST\text{-}SIZE$ from Example 3.2 we used in most of the previous examples, of inequalities over integers. We require that the predicate $PROP$ is (G,PROP')-compatible (see Definition 3.1). Compatibility is, however, required to be proven manually.

In the following examples, we consider a binary tree data structure, where we can define a more complicated predicate. Binary trees consist of nodes that have at most two children. A child is connected via a left selector or a right selector. The child is itself a node and can again have at most two children. Nodes without any children are called leaves; Nodes with children are called inner nodes. The (unique) node r that does not have any parents, i.e. a node for which r is its child, is called the root. For a node v, we call the subtrees of v the subtrees where v's children are the root[Knu97, p.308].

The hyperedge-replacement grammar Bin-Tree in Example 6.1 generates binary trees. The grammar consists of only one nonterminal symbol B with rank 2. We can interpret hyperedges with the label B as having the root of the tree at the vertex 0 and all leaves of the tree at the vertex 1. We suggest here that trees only have one leaf (i.e. that the leaves are represented as the same null object). If the tree, however, has leaves of different objects, we are unable to abstract this tree using the grammar Bin-Tree from Example 6.1. It is, however, possible to extend this grammar by new productions and another nonterminal symbol to also allow a tree, where one leaf has a different object than the others. For simplicity reasons, we will not consider these here.

This chapter consists of three sections. We first consider the domains of the predicate. We formally introduce the integer domain and the relative integer domain we already examined in the examples previous for $MOST\text{-}LIST\text{-}SIZE$. In the next section, we define the before mentioned set of predicates and see that $MOST\text{-}LIST\text{-}SIZE$ is such a predicate. Finally, we show in the last section how we apply the predicate abstraction analysis to an abstract hypergraph transition system step by step in an example.

6.1 Integers and Relative Integers

Our predicates will be defined on natural numbers. Defining them formally on integers is however simpler, as relative natural numbers could represent negative numbers. Additionally, we need negative and positive infinity to extend the integer to a complete lattice. The complete lattice then is given by the canonical partial order $\leq$. Negative infinity is the smallest element, and positive infinity is the largest element in the domain.

Definition 6.1 (Integer Complete Lattice)
We define the complete lattice over the integers $(Int, \leq)$ *with*

$$Int = \mathbb{Z} \cup \{-\infty, \infty\},$$
$$\forall i \in Int \ -\infty \leq i,$$
$$\forall i \in Int \ i \leq \infty \text{ and}$$
$$\forall i, j \in \mathbb{Z} : \quad i \leq j \Leftrightarrow \exists m \in \mathbb{N} \ i + m = j.$$

Similarly, we define the relative integers. The relative integers differ from the integers that we also allow a set of variables along with the integers. Thus our domain consists of a set of variables and a constant addition. We can map relative integers to concrete integers by adding the variable assignments of every variable and the additional integer together. This domain has similarly to the integers negative infinity as the smallest element and positive infinite as the largest element.

Definition 6.2 (Relative Integer Complete Lattice)
We define the relative integers over Var as

$$Int_{Rel} = (2^{Var} \times \mathbb{Z}) \cup \{-\infty, \infty\}$$

and the complete lattice over the relative integers $(Int_{Rel}, \leq_{Rel})$ *as*

$$\forall ir \in Int_{Rel} \ -\infty \leq_{Rel} ir,$$

$$\forall ir \in Int_{Rel} \ ir \leq_{Rel} \infty,$$

$$\forall (vs, z), (vs', z') \in Int_{Rel} :$$
$$(vs, z) \leq_{Rel} (vs', z') \Leftrightarrow vs \subseteq vs' \wedge z \leq z'.$$

The integer domain and the relative integers are both partial orders and complete lattices. It is not hard to see this statement for the integers. For the relative integers, the property is inherited by $\subseteq$ and $\leq$.

Lemma 6.1
The following are complete lattices:

- $(Int, \leq)$ *with the least upper bound* max
- $(Int_{Rel}, \leq_{Rel})$ *with the least upper bound* $\bigcup^{Rel}$

Proof See Appendix C.3.

To properly use the integers and the relative integers, we also need to define the connection between the integers and the relative integers by introducing the last ingredient for a relative index set as defined in Definition 3.7. We connect the relative integers to the integers by mapping every relative integer and variable assignment combination to an integer. A mapping from relative integers to integers is a form of concretisation, as we transform the abstract variables to a concrete index.

Definition 6.3 (Relative Operator)
We define for a variable assignment $\nu \in \mathcal{V}$ over $I_{\mathcal{V}} = \mathbb{N}$ the operator $[\![\cdot]\!]_\nu \in Int_{Rel} \mapsto Int$ as

$$[\![\infty]\!]_\nu = \infty$$
$$[\![-\infty]\!]_\nu = -\infty$$
$$[\![(vs, z)]\!]_\nu = z + \sum_{v \in vs} \nu(v)$$

Example *We have for $\nu(v) = 5$ that $[\![(\{v\}, -10)]\!]_\nu = -5$ and $[\![(\{v\}, 10)]\!]_\nu = 15$.*

The relative integers are forming a relative index set for the integers using this relative operator. We have a variable element for every variable and an integer element for every integer. Furthermore, if two relative integers $(vs, z), (vs', z')$ are ordered, then their variable sets vs, vs' and the integers z, z' are ordered as well. Thus, the sum of variable assignments $z + \sum_{v \in vs} \nu(v), z' + \sum_{v \in vs'} \nu(v)$ are ordered, and thus the concretisation $[\![(vs, z)]\!]_\nu, [\![(vs', z')]\!]_\nu$ is ordered.

Lemma 6.2 (Relative Index Set)
$(Int_{Rel}, \leq_{Rel})$ with the operator $[\![\cdot]\!]_\nu$ from Definition 6.3 is a relative index set for $(Int, \leq)$.

Proof See Appendix C.4.

Now we have all the requirements that we require for the domain. However, we also introduce an addition operator for both the integer domain and the relative integers to help define the compatible predicates. The addition for the canonical integer domain is ambiguous for the case $(\infty) + (-\infty)$, where we will set the value to ∞ as we want to compute an upper bound. For the relative integers, we use the union for variables and addition for the constants. Thus adding two relative integers is done by taking the adding the variable sets (taking the disjoint union) to a new variable set and adding the constants to a new constant. As we do not allow factors in the relative domain, only the addition of elements with disjoint variable sets have a feasible solution. Thus, if the variable sets are not disjoint, we assign the solution ∞ instead.

Definition 6.4 (Addition)
For $i, j \in Int$ we define

$$i \oplus j = \begin{cases} \infty & \text{if } i = \infty \vee j = \infty \\ -\infty & \text{if } \begin{array}{l}(i = -\infty \wedge j \neq \infty) \vee \\ (i \neq \infty \wedge j = -\infty)\end{array} \\ i + j & \text{else.} \end{cases}$$

For $ir, jr \in Int_{Rel}$ we define

$$ir \oplus_{Rel} jr = \begin{cases} \infty & \text{if } i = \infty \vee j = \infty \\ -\infty & \text{if } \begin{array}{ll}(i = -\infty & \wedge j \neq \infty) \vee \\ (i \neq \infty & \wedge j = -\infty)\end{array} \\ \infty & \text{if } \begin{array}{l} i = (vs_i, z_i) \wedge j = (vs_j, z_j) \wedge \\ vs_i \cap vs_j \neq \emptyset \end{array} \\ (vs_i \cup vs_j, z_i + z_j) & \text{if } \begin{array}{l} i = (vs_i, z_i) \wedge j = (vs_j, z_j) \wedge \\ vs_i \cap vs_j = \emptyset. \end{array} \end{cases}$$

If we add the elements of a set together, we also use the notation Σ for $\oplus$ and Σ^{Rel} for $\oplus_{Rel}$.

These addition operators are connected via the relative index operator. By first applying the variable assignments and applying the addition, the result is always smaller than first applying the addition to the indices and then applying the variable assignments. This Lemma does not hold for equality as the relative addition may result in infinity although none of the elements was infinity, their variable sets were just not disjoint. If, however, the result of the relative addition is not infinity, the Lemma states the equality of both additions.

Lemma 6.3 (Linearity of Relative Integer Addition)
For all variable assignments $\nu \in \mathcal{V}$ and all relative integers $ir, jr \in Int_{Rel}$ we have

$$[\![ir]\!]_\nu \oplus [\![jr]\!]_\nu \leq [\![ir \oplus_{Rel} jr]\!]_\nu.$$

If $ir \oplus_{Rel} jr \neq \infty$ we have in particular the equality

$$[\![ir]\!]_\nu \oplus [\![jr]\!]_\nu = [\![ir \oplus_{Rel} jr]\!]_\nu.$$

Proof See Appendix C.5

Example *Let $\nu(v_i) = i$ then we have:*

- $-2 = -5 + 3 = [\![(\emptyset, -5)]\!]_\nu \oplus [\![(\{v_3\}, 0)]\!]_\nu = [\![(\emptyset, -5) \oplus_{Rel} (\{v_3\}, 0)]\!]_\nu$
- $\infty = [\![\infty]\!]_\nu \oplus [\![-\infty]\!]_\nu = [\![\infty \oplus_{Rel} -\infty]\!]_\nu$
- $2 = [\![(\{v_1\}, 0)]\!]_\nu \oplus [\![(\{v_1\}, 0)]\!]_\nu \leq \infty = [\![(\{v_1\}, 0) \oplus_{Rel} (\{v_1\}, 0)]\!]_\nu$

The additions $\oplus$ and $\oplus_{Rel}$ both form the commutative monoids $(\mathrm{Int}, \oplus, 0)$ and $(\mathrm{Int}_{Rel}, \oplus_{Rel}, (\emptyset, 0)$. A monoid is a structure that is associative and has a neutral

element. We have the associativity as for every $i \oplus j = j \oplus i$ and the neutral element 0 as $i + 0 = 0$. Similar for $\oplus_{Rel}$ we have $ir \oplus_{Rel} jr = jr \oplus_{Rel} ir$ and $j + (\emptyset, 0) = j$. The addition $\oplus$ is commutative since $i \oplus (j \oplus k) = (i \oplus j) \oplus k$ and the addition $\oplus_{Rel}$ is commutative since $ir \oplus_{Rel} (jr \oplus_{Rel} kr) = (ir \oplus_{Rel} jr) \oplus_{Rel} kr$. Neither are, however, groups, as they do not have an inverse. In both cases, ∞ does not have an inverse. We need a transformation of an inequality later, and therefore we prove the transformation that allows us shifting the negated version of a positive integer on the other side. The negated version is not necessarily the same as an inverse. The negated version of ∞ would be $-\infty$, although $-\infty$ is not the inverse of ∞. We can prove this transformation using a case distinction.

Lemma 6.4 (Inequality Negation)
We have, for a positive *integer $i \in \mathbb{N} \cup \{\infty\}$ and integers $j, k \in Int$, that*

$$k \geq i \oplus j \Rightarrow k \oplus (-i) \geq j$$

where $-i$ is the negative version of i.

Proof See Appendix C.6.

Example *We have that:*

- $\infty \geq \infty \oplus 5 \Rightarrow \infty \oplus (-5) \geq \infty$
- $-\infty \geq 1 \oplus 1 \Rightarrow -\infty \oplus (-1) \geq 1$ *as* $-\infty \geq 1 \oplus 1$ *does not hold*
- $3 \geq 1 \oplus 1 \Rightarrow 3 \oplus (-1) \geq 1$

6.2 Inequality Compatible

Using the integer domain, we now define inequality compatible predicates. Inequality compatible predicates are compatible (see Definition 3.1), and we define them as sets of linear and non-factored (i.e. coefficients of 1 except for the constant term) inequalities. A linear and non-factored inequality consists of variables and a constant. We use every inequality $l \geq r$ for one replacement $H[e/R]$. The left-hand side l of the inequality represents the value of the hyperedge e, and the right-hand side represents the combined values of the hypergraph R. The left-hand side of an inequality has to be equal or larger than the right side in order to guarantee the monotonic directed property. Every variable in these inequalities is non-factored, i.e. has the factor 1, which simplifies finding index abstraction rules. The right side is not allowed to be non-positive to guarantee the terminating property. Thus, either the constant is positive, or at least one variable (with a positive variable assignment) exists.

Definition 6.5 (Inequalities)
We call a non-factored, positive and linear inequality a tuple iq over a set of hyperedges E if and only if

$$iq = (E', c) \in 2^E \times \mathbb{N} \text{ with } c \neq 0 \vee E' \neq \emptyset.$$

We call the set of non-factored, positive and linear inequalities INEQ.

We use a set of non-factored, positive and linear inequalities for one direct derivation. We use these rules to define $PROP$ for every hypergraph that is derivable. We only need to be able to calculate indices for our inequalities and every terminyl hyperedge to perform the analysis. By additionally specifying the indices for terminyl hyperedges, we have sufficiently defined $PROP$. Remark, however, that it does not define $PROP$ completely. For non-derivable hypergraph with at least two hyperedges, $PROP$ is undefined by this Definition. We can fix this issue by assigning no index to every hypergraph H that is not derivable by any handle $A^\bullet$ with $A \in \Sigma$, i.e. $PROP(H, i)$ does not hold for any i.

Definition 6.6 (Inequality Compatible)
Let $G \in HRG$ and

$$PROP \in Int \mapsto (\mathcal{H}_T \mapsto \mathbb{B}),$$
$$PROP' \in Int \mapsto ((E_R \mapsto Int) \times \mathcal{H}_\Sigma \mapsto \mathbb{B}).$$

A predicate $PROP$ is (G, $PROP'$)-ineq-compatible if and only if $PROP$ is (G, $PROP'$)-compatible and for all derivations $A^\bullet \Rightarrow_{(\xi,p)} R \Rightarrow^* H \in \mathcal{H}_T$ there exists a set $iqs_p \subseteq INEQ$ such that for all $i \in Int$

$$PROP'(i)(assign, R) \Leftrightarrow \bigwedge_{(E',c)\in iqs_p} i \geq \sum_{e\in E'} assign(e) \oplus c \tag{6.1}$$

and for all terminal symbols $A \in T$ there exists $i \in \mathbb{N} \setminus \{0\}$ such that i is the smallest index with

$$\forall j \geq i \; PROP(j)(A^\bullet). \tag{6.2}$$

Example *We consider the derivation $B^\bullet \Rightarrow_{(2)} R \Rightarrow^* H \in \mathcal{H}_T$ from the tree grammar Bin-Tree in Example 6.1 and want to define a set $iqs_{(2)}$ for this derivation such that it is the predicate for the size i of the tree. Thus the size of $H(B^\bullet)$ is the maximum of its left and its right subtree. We decode this as two inequalities. Let $E_R = \{e_l, e_r, e_B\}$ with $label(e_s) = s$, then we have:*

$$i \geq assign(e_B) \oplus 1 \text{ for the left subtree and}$$
$$i \geq 1 \text{ for the right subtree.}$$

We have the inequality tuples $iqs_{(2)} = \{(\{e_B\}, 1), (\emptyset, 1)\}$

There is not any algorithm known to check for compatibility of these inequality predicates. Both compatibility in general and inequality-compatible may be undecidable. Thus compatibility has to be proven by the user. All other necessary properties are however either decidable or hold by construction. We will now prove that all other conditions presented in chapter 3 hold for inequality-compatible predicates.

The monotonic directed property of an inequality-compatible predicate can be proven by structural induction on every derivation. Every terminal hyperedge is monotonic directed by construction. For nonterminal hyperedges, we have an inequality that enforces the monotonic directed property. We need to prove

that every derivation yields at least one positive, non-infinite index, to show that every derivable hypergraph has a positive, non-infinite index i such that $PROP(i)(H)$.

Lemma 6.5 (Inequality is Monotonic Directed)
Every (G, PROP′)-ineq-compatible predicate PROP is monotonic directed.

Proof See Appendix C.7.

We require our variable assignment always to have a satisfying index. We use variable assignments that map the variables to $\mathbb{N}$. We need to prove that for every derivable hypergraph exists a satisfying index $i \in \mathbb{N}$. The existence, however, directly follows from the proof that every derivable hypergraph has a positive, non-infinite index i such that $PROP(i)(H)$.

Corollary 6.6 (Variable Assignment)
Let be $I_{\mathcal{V}} = \mathbb{N}$ and PROP be an (G,PROP′)-ineq-compatible predicate. The set $\mathcal{V} = Var \mapsto I_{\mathcal{V}}$ with $I_{\mathcal{V}} = \mathbb{N}$ is a set of variable assignments for PROP.

Next, we prove that $PROP$ is terminating. The termination property holds since the constant and all variable assignments are positive. As every non-terminating index is positive, all negative integers are indeed terminating, and thus every strict descending chain will eventually have a negative index. Similarly, we can prove this by structural induction on every derivation.

Lemma 6.7 (Inequality is terminating)
Let PROP be a (G, PROP′)-ineq-compatible predicate. Every non-positive integer is terminating for PROP, and thus PROP is terminating.

Proof See Appendix C.8.

The last essential ingredients for the predicate are the index abstraction rules. We can synthesis these rules by using the relative addition $\oplus_{Rel}$. We can translate the inequality to an relative addition by replacing every $\oplus$ with an $\oplus_{Rel}$ for the abstraction. Since $\oplus_{Rel}$ will yield a higher value by Lemma 6.3, we use the relative addition without any change. The materialisation, however, is more complicated. We estimate the assign function by assuming for every e' that every other assign value $assign(e)$ is minimal. Then we solve for the assign value $assign(e')$ and get an upper bound for $assign(e')$.

Lemma 6.8 (Index Abstraction Rules)
Let PROP be a (G, PROP′)-ineq-compatible predicate. An inequality set iqs_p restricted to a hyperedge e is the set $iqs_p(e) = \{(E', c) \in iqs_p \mid e \in E'\}$.For a derivation $A^\bullet \Rightarrow_{(\xi,p)} R$ and the set $iqs_p \subseteq INEQ$, the functions φ_p^a and φ_p^m with

$$\varphi_p^a(assign_{Rel}) = \bigcup_{(E',c)\in iqs_p}^{Rel} \sum_{e\in E'}^{Rel} assign_{Rel}(e) \oplus_{Rel} c \tag{6.3}$$

$$\varphi_p^m(ir)(e) = \begin{cases} (\emptyset, i) \text{ such that } PROP(i)(e^\bullet) & \text{if } label(e) \in T \\ \bigcup_{(E',c)\in iqs_p(e)}^{Rel} ir \oplus_{Rel} (-|E' \setminus \{e\}| - c) & \text{if } \begin{array}{l} iqs_p(e) \neq \emptyset \\ \wedge label(e) \notin T \end{array} \\ \infty & \text{else} \end{cases} \tag{6.4}$$

are monotone index abstraction rules for PROP.

Proof See Appendix C.9.

Example *We consider the set from the last example* $iqs_p = \{(\{e_B\}, 1), (\emptyset, 1)\}$, *which gives the inequalities*

$$i \geq assign(e_B) \oplus 1 \text{ for the left subtree and}$$
$$i \geq 1 \text{ for the right subtree.}$$

We have then the following index abstraction rules for this set:

$$\varphi^a_{(2)}(assign_{Rel}) = assign_{Rel}(e_l) \oplus_{Rel} 1$$
$$\varphi^m_{(2)}(ir)(e_l) = 1$$
$$\varphi^m_{(2)}(ir)(e_B) = ir \oplus_{Rel} (-1)$$
$$\varphi^m_{(2)}(ir)(e_r) = 1.$$

To also allow the use of hyperedge sets for the termination proof (see section 5.2), we need to define a safe set function for the relative integers. Such a feasible function is the addition. The addition is commutative and associative; thus, the addition of a set is uniquely defined, and since the concretisation of a relative integer is its addition, the addition forms a safe operation. We can then prove by induction on the elements of the set $irs \subseteq \mathrm{Int}_{Rel}$ and $es \subseteq E$ that it is indeed a safe set function.

Lemma 6.9 (Addition is Safe)
The set function $\Sigma^{Rel} \in 2^{Int_{Rel}} \mapsto Int_{Rel}$ *with*

$$\Sigma^{Rel}(\{ir\}) = ir$$
$$\Sigma^{Rel}(irs \cup \{ir\}) = \Sigma^{Rel}(irs) \oplus_{Rel} ir \text{ where } ir \notin irs$$

is a safe set function.

Proof See Appendix C.10.

6.3 A Tree Example

In this section, we take a look at an example for the whole process of the analysis in detail. We will see how an abstract HRG transition system can be analysed using the inequality predicates explained in this chapter. The program in Example 6.2 serves as the base for the abstract transition system. We will not show how to construct the abstract transition system, but other literature gave explanations on how this could be done automatically (see [Jan17, AJMN18, HJKN15] for details). The example program is using a binary tree data structure and tries to find a path in the tree and later save the path in another tree. The path is dirtily saved as a binary tree using the rightmost path for the saved path. This saving method makes the program a bit messy and less readable, but reduced the size of the example, as we do not need to use a list data structure instead.

The Program Code We introduce 6 locations for the program in Example 6.2. These locations represent vital states of the program that are worthwhile to take a closer look at, but we could as well set in many other ways. We annotated the locations by a number beside the program code. We interpret a location as the moment before this statement. The first location is at the first line of the program code and thus represents the moment before the program starts. Afterwards, we enter a while-loop. The second location is right before the branching statement. If `in.l` is successfully checked, the program moves the pointer `in` to the left children and enters the third location. If `in.l` is unsuccessfully checked, the program moves the pointer `in` to the right child and enters the fourth location. Then the respective other child is set as the left child, and the object of the pointer `out` is set as the right children. Both program paths now enter the fifth location. The pointer `out` is set to the new root, and the while-loop continues. If a selector points to `null`, the while-loop terminates, and the program enters the sixth location, where the program terminates. We can also determine a set of critical locations. As every infinite run in this program has to check the while statement after each iteration, it is clear that the set $\{1\}$ is a set of critical locations.

Example 6.2: An example program using the tree data structure. The program iterates through the tree of variable `in` and saves some nodes using `out`.

```
1:      while(in.r != null & in.l != null){
                t:= in;
2:              if(check(in.l)){
                        in := in.l;
3:                      t.l := t.r;
                        t.r := out;
                }else{
                        in := in.r;
4:                      t.r := out;
                }
5:              out := t;
        }
6:      end;
```

From Program Code to Transition System We transform the program in Example 6.2 into an abstract transition system without giving a corresponding concrete transition system for simplicity. If we would, however, draw a corresponding concrete transition system, it would be infinite large as there are infinitely many possible start configurations. We will mark the pointers `in`, `null`, `t` and `out` by hyperedges with only one attached vertex and a label corresponding to their name. We transform the program into the abstract transition system in Figure 6.1 on page 82 by performing a symbolic execution (cf.[HJKN15]). The first location marks the start location. As locations are a unique description of every state, we notate each state as $a_i = \langle i, H \rangle$ where i is the location and H the hypergraph.

We assume that there are two distinct binary trees, one from the root `in` to the leaf `null` and one from the root `out` to the leaf `null` —additionally the

pointer `t` points to the same object as `out`. Then we materialise the tree of `in`. If either one (or both) of the selector points to `null`, we move to the state a_6 and terminate. As there are three possibilities to materialise the nonterminal hyperedge such that one of the children is null, we have three possibilities to take the transition to state a_6. If neither of them is null, we continue to state a_2. This transition also covers moving the pointer `t` to the object of the pointer `in`. We either move to the state a_3 in case **check(`in.l`)** holds or to the state a_4 in case **check(`in.l`)** does not hold. For the execution, we again move the pointer to their corresponding objects. Next, both states join when reaching the state a_5. In both cases, the left and right selectors are adjusted accordingly such that no selector points to the object of `in` and the right selector points to `out`. We then change the pointer `out` to the object of `t`. After abstracting the tree of `out`, we move to the start state a_1 back.

Tree Depth Predicates In the next step, we define a compatible predicate. There are, however, multiple notable properties of a tree that viable to prove termination. We could try a selector-counting approach, where the predicate defines an upper bound for the number of selectors in the tree. This approach would mean, that if two hyperedges share an assigned variable, we are not able to calculate the number of selectors after abstracting the two hyperedges via the production rule (4) anymore. This happens quite often with the tree grammar. We can try to avoid this by instead use a tree-depth approach. We define the predicate MTD that checks whether the index is an upper bound for the depth of the tree. The depth of a tree is the maximum size of a path from the root to the leaf. We can also interpret it recursively: the depth of a tree is the incremented depth of the maximum depth of the children's subtrees. The depth of a tree is also called its maximal level[Knu97, p.308]. If the hypergraph is not a tree at all, the hypergraph does not have any depth, i.e. no index is an upper bound for the depth (or the depth is undefined). If the hypergraph only consists of one terminal hyperedge, we define the depth as 1.

Definition 6.7 (Most Tree Depth)
Let Bin-Tree be the grammar from Example 6.1. We define the predicate $MTD \in Int \times \mathcal{H}_T \mapsto \mathbb{B}$ *as an upper bound for the depth of the tree in the hypergraph.*

- *We have for all terminal symbols* $A \in T$ *and all positive integers* $i \geq 1$ *that* $MTD(i)(A^\bullet)$.
- *We have for all nonterminal symbols* $B \in N$ *with* $H \in L_{Bin\text{-}Tree}(B^\bullet)$ *where* i *is the depth of (the binary tree)* H *that* $\forall j \geq i\ MTD(j)(H)$.
- *For every other combination of* $i \in Int$ *and* $H \in \mathcal{H}_T$*, the predicate* $MTD(i)(H)$ *does not hold.*

Defining MTD' can be done in at least two ways. Either we are independent of the terminal hyperedge, or we incorporate the values of terminal hyperedges in the inequalities. For example, if we would define the rules for the rule (1) of Bin-Tree from Example 6.1, we could either define it as $i \geq 1$, where i is the value of the left side. In this case, we use none of the values of the terminal hyperedges. We could, however, also define it as $i \geq j_l \wedge i \geq j_r$ where j_l is the

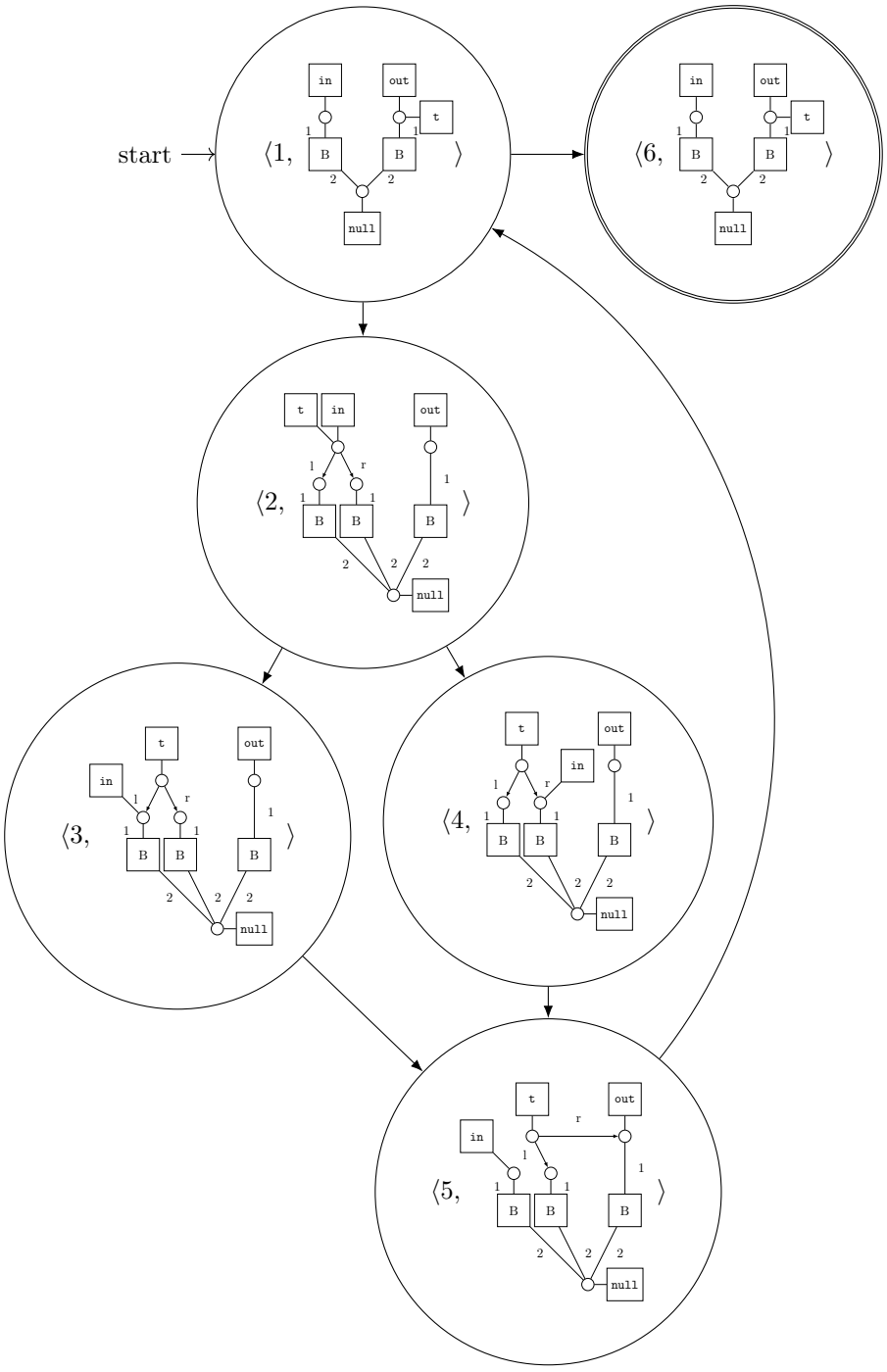


Figure 6.1: An abstract transition system for the program in Example 6.2. Every state has a unique location. The states with location 1, 2, 3, 4 and 5 form a circle.

value of the left selector and j_r is the value of the right selector. As j_l and j_r are fixed to 1 here, both of these definitions are equivalent. We will stick with a definition that is independent of the values of terminal hyperedges, as they tend to be easier to understand.

Every rule of the grammar Bin-Tree from Example 6.1 gets a set of inequalities. The rule (1) yields a tree with the fixed depth 1; therefore, we say that every upper bound that is at least 1 is correct. For the rule (2), the right subtree is empty; therefore, this subtree will not give us a valid depth. The depth of the tree is the depth of the left subtree plus one. The rule (3) is analogous. For the rule (4), we have two nonterminal hyperedges. The depth of the tree is the maximum of the depth of its subtrees plus 1. An upper bound for the depth is larger than the depth of the left subtree plus one and larger than the depth of the right subtree plus one. With this Definition for MTD', we can now prove that MTD is (Bin-Tree, MTD')-ineq-compatible with a case distinction over every rule.

Theorem 6.10 (Most Tree Depth is Inequality Compatible)
The predicate MTD from Definition 6.7 is (Bin-Tree, MTD')-ineq-compatible where Bin-Tree is the grammar from Example 6.1 and MTD' is defined by the inequalities:

- $iqs_{(1)} = \{(\emptyset, 1)\}$ *which corresponds to*

$$i \geq 1$$

- $iqs_{(2)} = \{(\{e_B\}, 1)\}$ *which corresponds to*

$$i \geq assign(e_B) \oplus 1$$

- $iqs_{(3)} = \{((\{e_B\}, 1)\}$ *which corresponds to*

$$i \geq assign(e_B) \oplus 1$$

- $iqs_{(4)} = \{(\{e_{B,l}\}, 1), (\{e_{B,r}\}, 1)\}$ *which corresponds to*

$$i \geq assign(e_{B,l}) \oplus 1 \wedge i \geq assign(e_{B,r}) \oplus 1$$

Proof MTD is represented by inequalities as defined in Definition 6.6. It is left to prove that MTD is (Bin-Tree,MTD')-compatible. We first prove the direction from MTD' to MTD:
Let $A^\bullet \Rightarrow_{(\xi,p)} R \Rightarrow^* H \in \mathcal{H}_T$ and assume that there exists a function $assign \in E_R \mapsto I$ and an integer $i \in$ Int such that $MTD'(i)(assign, R)$ and for all $e \in E_R$ $MTD(assign(e))(H(e))$ holds. We proceed by a case distinction on the productions P of Bin-Tree:

(1) The tree H has a depth of 1; thus, every positive integer is an upper bound and thus $MTD(i)(H)$ for $i \geq 1$.

(2) The tree H either has a depth of 1 or the depth is one large than the left subtree and thus $MTD(i)(H)$ for $i \geq 1 \wedge i \geq assign(e_B) \oplus 1$.

(3) Similar to (2).

(4) The depth of the tree H is one larger than both of its subtrees and thus $MTD(i)(H)$ for $i \geq assign(e_{B,l}) \oplus 1 \wedge i \geq assign(e_{B,r}) \oplus 1$.

Now we prove the other direction. Thus for a derivation $A^\bullet, \Rightarrow_{(\xi,p)} R \Rightarrow^* H \in \mathcal{H}_T$ we now assume that $MTD(i)(H)$. We now prove that there exists an assign function that fulfils the condition

$$PROP'(i)(assign, R) \wedge \forall e \in E_R \; PROP(assign(e))(H(e)). \qquad \blacksquare$$

Again we proceed by a case distinction on the productions P of Bin-Tree:

(1) The tree H has a depth of at least 1, and since i must be positive, we also have that the function $assign$ with $assign(e_l) = 1$, $assign(e_r) = 1$ fulfils the condition.

(2) The tree H has a depth of at most i, and the right subtree has a depth of 1. Assume that the left subtree has a depth of j, then $i \geq j + 1$, which proves that the function $assign$ with $assign(e_l) = 1$, $assign(e_r) = 1$ and $assign(e_B) = j$ fulfils the condition.

(3) Similar to (2).

(4) The tree H has a depth of at most i. Assume that the right subtree has the depth j_r and the left subtree the depth j_l. We have that $i \geq j_r + 1$ and $i \geq j_l + 1$. Thus the function $assign$ with $assign(e_l) = 1$, $assign(e_r) = 1$, $assign(e_{B,l}) = j_l$ and $assign(e_{B,r}) = j_r$ fulfils the condition.

Analysing rhe Critical States We now have to annotate the hypergraphs in the transition system from Figure 6.1. Since $\{1\}$ is a set of critical locations, we analyse all states with the location 1. As only one state has the location 1, we thus only need to analyse this state a_1. The new flow graph duplicates the state to analyse into the new states a_{10} and a_{1f}. Every other state remains the same. To calculate the solution for the data flow analysis $AP(MTD, a_1,)$, we will assign a_{10} the value ι and iterate this over the equations of $AP(MTD, a_1,)$. We will annotate the assign values for every state besides the nonterminal hyperedges in blue bold in the following. We do not annotate the indices for terminal hyperedges, as we will always assign them the index 1. The names e_A of the hyperedges are here composed of their label A and additionally the identifiers *in* and *out* for the two hyperedges labelled B. The start value ι is defined as the following function:

$$\begin{aligned} \iota(e_{in}) &= 1 & \iota(e_{out}) &= 1 \\ \iota(e_t) &= 1 & \iota(e_{null}) &= 1 \\ \iota(e_{B,in}) &= \mathbf{v_i} & \iota(e_{B,out}) &= \mathbf{v_o} \end{aligned}$$

We first analyse the transition from state a_1 to state a_2 as shown in Figure 6.2. We materialise the left nonterminal hyperedge $e_{B,in}$ with the rule (4). In this materialisation, the index $\mathbf{v_i}$ is subtracted by one and then assigned to the two new hyperedges. We do not know the exact depth of the subtrees, but we do know that their maximum is at most $\mathbf{v_i - 1}$. Thus we also know that an upper

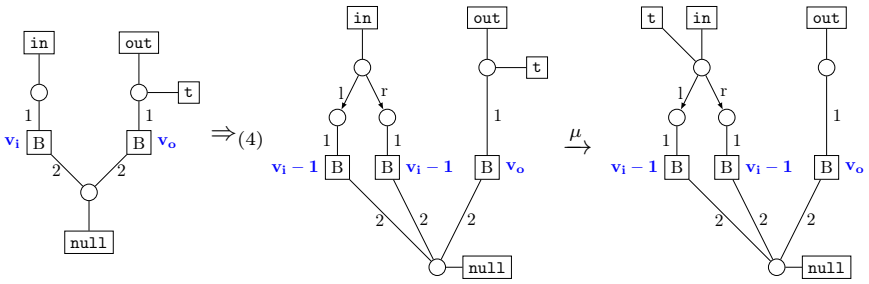


Figure 6.2: The annotated transition from a_1 to a_2

bound for both of them is $\mathbf{v_i - 1}$. In the concrete operation, only terminal hyperedges are moved; thus, we do not change any indices.

We ignore the transition from state a_1 to state a_6, as the state a_6 cannot reach the state a_{1f}. The transitions from a_2 to a_3, from a_2 to a_4, from a_3 to a_5 and from a_4 to a_5 consists only of concrete operations as seen in Figure 6.3. Thus we do not need to adjust any indices of nonterminal hyperedges after the operations. We have, however, a merging in the state a_5 from the two transitions $a_3 \to a_5$ and $a_4 \to a_5$. However, both transitions yield the same assign function. Using our widening operator from Definition 4.9 with $K = \{6\}$ to combine these two elements, we use the least upper bounds for relative integers:

$$\begin{aligned}
\mathbf{v_i - 1} &= \bigcup^{Rel}(\mathbf{v_i - 1}, \mathbf{v_i - 1}) && \text{for the left tree,}\\
\mathbf{v_i - 1} &= \bigcup^{Rel}(\mathbf{v_i - 1}, \mathbf{v_i - 1}) && \text{for the middle tree,}\\
\mathbf{v_o} &= \bigcup^{Rel}(\mathbf{v_o}, \mathbf{v_o}) && \text{for the right tree.}
\end{aligned}$$

In the transition from a_5 to a_1 shown in Figure 6.4, we have a concrete operation and an abstraction. The concrete operation consists only of changes of the terminal hyperedges for which it is trivial to compute an index. The abstraction is a backwards derivation by using the rule (4). By using the index abstraction rule, we compute

$$\mathbf{v_o} + \mathbf{v_i} = \bigcup^{Rel}(\mathbf{v_o}, \mathbf{v_i - 1}).$$

Thus the new index is $\mathbf{v_o} + \mathbf{v_i}$. Since we reached a fixpoint in this computation (again ignoring a_6), we have a solution for the equation system. We are, however, only interested in the assign function ϕ for a_{1f}:

$$\begin{aligned}
\phi(e_{in}) &= 1, & \phi(e_{out}) &= 1,\\
\phi(e_t) &= 1, & \phi(e_{null}) &= 1,\\
\phi(e_{B,in}) &= \mathbf{v_i - 1}, & \phi(e_{B,out}) &= \mathbf{v_i} + \mathbf{v_o}.
\end{aligned}$$

We observe that $\iota(e_{B,in}) = \mathbf{v_i} >_{Rel} \mathbf{v_i - 1} = \phi(e_{B,in})$, which by Theorem 5.1 proves that the program 6.2 terminates, given that the input is as described in the abstract transition system from Figure 6.1.

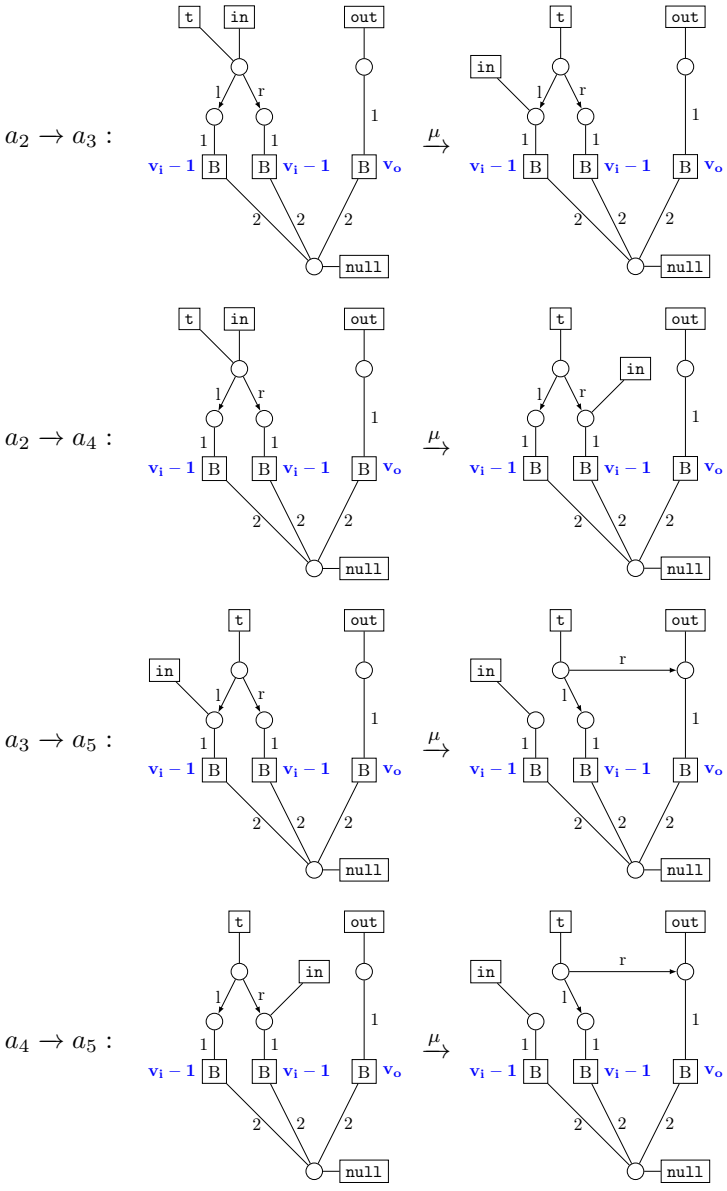


Figure 6.3: The annotated transitions from a_2 to a_3, from a_2 to a_4, from a_3 to a_5 and from a_4 to a_5

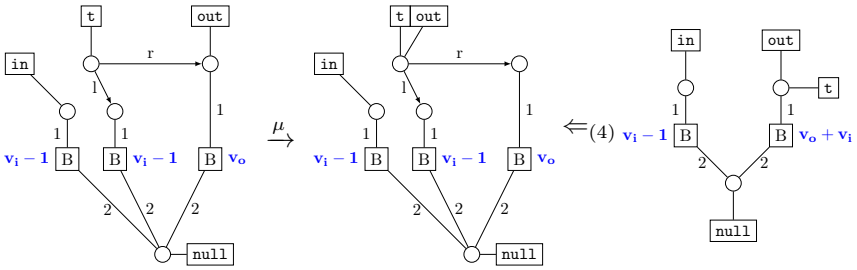


Figure 6.4: The annotated transition from a_5 to a_1

Chapter 7

Conclusion

We wanted to prove termination of pointer programs. We used an invariant-preserving analysis and extended the result to find variants, which we could use to prove the termination. The invariant-preserving analysis computes an abstract transition system. The abstract transition system is too abstract to yield a termination proof. Thus, we annotated the transition system with indices that represent a potential variant property. We have done this by formalising predicates defined on predicates compatible with a graph grammar and well-defined for a monotonic data flow analysis. But even if we could use them on a data flow analysis, the variant property was to concrete. If we would use the indices of these predicate, we would need to expand the transition system again into a potential infinite transition system. Thus we needed to abstract the new refinement again. We have done this by extending the index set with variables to a relative index set. Furthermore, we introduced index abstraction rules, which approximated the relative indices according to the predicate. The relative index set and the index abstraction rules could then be used to construct a data flow analysis. Using the newly gained data flow analysis, we can analyse a set of states of the abstract transition system that correspond to loops for which we prove a partial termination argument. Proving for each of these states the partial termination argument, i.e. that every of their infinite paths is not realisable in the concrete transition system, was sufficient to prove that the whole program terminates for a given precondition. One set of instances are the inequality predicates that are defined on non-factored, positive inequalities over the integer domain.

The main theorems of this thesis are Theorem 4.3 and Theorem 5.1. In Theorem 4.3, we stated the soundness of the index analysis. We said it is sound because every concrete run starting with a hypergraph satisfying the index assignment is abstracted by the annotation of the abstract transition system. Therefore, we extended the precondition to an arbitrary but fixed upper bound for the index and afterwards filtered every other hypergraph accordingly. This gave us a relative upper bound for the hypergraphs of every other state. In Theorem 5.1, we used this index analysis to say that if, after reaching the state again, the upper bound for the index of any nonterminal hyperedge is decreasing, then every run with infinitely many occurrences of this state does not correspond to any concrete run. This theorem allowed us to prove the partial termination argument for one state. The partial termination argument

proved that every infinite run containing the state in question infinitely often does not correspond to an infinite run in the concrete transition system. In Theorem 5.4, we extended it to a termination argument for the whole program. Notable is also that we could generalise Theorem 5.1 to the comparison of a set of hyperedges instead of only one hyperedge in Theorem 5.3 by using a special set function that allowed the safe combination of multiple indices.

Our technique relies on user-defined predicates and user-defined grammars. A bad predicate may not find a termination proof and a non-compatible one may even find termination proofs for non-terminating programs. As the halting problem is not decidable, this technique is not complete. If we cannot find a partial termination argument for any loop, i.e. no index (and no combination of indices) has the decreasing property, we cannot prove the overall termination argument. There are several reasons the analysis did not yield a termination proof. We could lose the decreasing property because of widening. The relative index set and therefore also the least upper bound may also be too abstract. Furthermore, the predicate may not yield a variant. There are at least two natural predicates for trees and in some cases, the tree height can prove the termination, in other the tree size. There may also be cases where none of the two or even none at all may be usable. The last possibility is that the underlying symbolic execution can also fail and is not capable of computing an abstract transition system with the given grammar. The index analysis is especially not capable of proving non-termination.

Nevertheless, we are capable of computing this analysis quasi-automatic. In the general case, this analysis requires the relative index set and the index abstraction rules provided by the user. These would be required to correspond to a monotonically directed, terminating and compatible predicate. However, as we have seen in Chapter 6, if we restrict the predicate to non-factored, linear inequalities using the integers, the predicate is automatically monotonically directed and terminating. Additionally, we already have a predefined relative index set for the integers and index abstraction rules can be computed automatically. Thus, for these instances, only a hyperedge-replacement grammar G, a predicate $PROP'$ and a (G,PROP')-compatible predicate $PROP$ need to be defined and the user needs to prove that $PROP$ is indeed compatible. These predicates probably already cover many notable data structures, as the graph grammars for most data structures only have product rules in which possible sizes change by a constant. Simple programs will likely have a termination argument that uses some sort of notable size information. As we can try to find termination arguments with all predicates that are available, it is like that predefined predicates will be sufficient. However, for more complex algorithms, standard size definitions may not be sufficient.

7.1 Further Work

This work aims to give a basis for proving termination in Attestor[AJK+18]. Attestor is an open-source implementation[1] for Java Bytecode.

Implementation The here presented mechanism is not implemented yet. As there is an open-source implementation for the symbolic execution, there should

[1] https://github.com/moves-rwth/attestor

also be an implementation for this technique to allow ATTESTOR to automatically prove termination on its input. We can also define fitting predefined predicates, either that are compatible with all grammars or also for predefined grammars, such that ATTESTOR automatically can prove termination without any additional user input. As ATTESTOR is not optimised to analyse the decomposition of transitions, we need to find a feasible software design to analyse the decomposition efficiently. Other than that, the implementation will most likely require the implementation of the data flow analysis as shown in Chapter 4. To incorporate the safe set functions as described in Theorem 5.3, we also require an algorithm to efficiently find a combination of indices that generate a decreasing index.

Compatibility Checking It is likely that it is undecidable whether a predicate $PROP$ is (G,PROP')-compatible. In the general case, $PROP$ and $PROP'$ are any decidable predicates. Since they are decidable, there exist programs P and P' which decide them respectively. Thus deciding if the predicate $PROP$ is compatible with G and P' is equivalent to decide if a program P is compatible with G and P'. However, to decide if $PROP$ is a (G,PROP')-ineq-compatible predicate may be more likely, as now $PROP'$ at least is defined as an inequality. By defining $PROP$ as the minimal predicate suited for $PROP'$, it is only left to check whether $PROP'$ behaves compatibly to G. This may or may not be decidable. If there is a way to decide this, however, we do not require any more that the user needs to prove compatibility before inputting a predicate, as this could be done automatically.

Other Index Sets and other Predicates Although non-factored, positive inequalities already allow many predicates, it may be possible to extend the set of possible predicates. It may be possible to allow more complex terms including constant factors, polynomial terms, min/max, etc. Some of these may seem trivial for one direction but is rather difficult for the other as for the other direction we need to compute an upper bound for the inverse function. The inverse function may not exist (for example min and max does not have one) and computing an upper bound is rather complicated. It may also be required to extend the index set to incorporate new terms for the approximation of the inverse function, e.g. constant factors only make sense if we allow a factor for the variables in the relative index sets as well.

Interprocedural Analysis Interprocedural programs can be defined on a single abstract transition system, but it can also be defined on transition systems for each method. The latter one uses contracts between the transition systems to operate. We did not look into how to annotate these contracts to extend the analysis for procedural pointer programs and then also allow recursion to be analysed for termination. This would then extend the functionalities of termination proves with ATTESTOR to even recursive programs.

Appendix A

Set Theory

Sets and Sequences

Sets are a collection of elements. There are a variety of different operations and relations on sets.

Definition A.1 (Set Operators and Relations)
Let S, S' be two sets, we define:

1. *$\emptyset$ is the empty set.*
2. *$S \cup S'$ is the union of S and S'.*
3. *$S \uplus S'$ is the disjoint union of S and S'.*
4. *$S \cap S'$ is the intersection of S and S'.*
5. *$S \setminus S'$ is the difference between S and S'.*
6. *2^S is the powerset of S.*
7. *$S \subseteq S'$ iff S is a subset of S'.*
8. *$s \in S$ iff s is an element of S.*

Some well-known sets are the set of booleans consisting of $true$ and $false$, and the set of natural numbers consisting of all natural numbers including 0.

Definition A.2 (Booleans, Natural Numbers and Integers)
We define the following sets:

- *$\mathbb{B}$ is the set of booleans $\{true, false\}$.*
- *$\mathbb{N}$ is the set of natural numbers.*
- *$\mathbb{Z}$ is the set of integers.*

Example *We consider the set $\mathbb{B}$.*

- $\mathbb{B} \cup \{neither\} = \{true, false, neither\}$

- $\mathbb{B} \cap \{true, neither\} = \{true\}$
- $\mathbb{B} \setminus \{true\} = \{false\}$
- $\mathbb{B} \uplus \mathbb{B} = \{true1, true2, false1, false2\}$
- $2^{\mathbb{B}} = \{\emptyset, \{true\}, \{false\}, \{true, false\}\}$
- $\{true\} \subseteq \mathbb{B}$ *holds.*
- $true \in \mathbb{B}$ *holds.*

A particular subset of the natural numbers are intervals between two natural numbers (or potentially also ∞).

Definition A.3 (Interval)
$[n, n']$ is the interval of natural numbers between n and n':

$$[n, n'] = \{m \in \mathbb{N} \mid n \leq m \leq n'\}$$

Example *Some intervals are:*

- $[1, 3] = \{1, 2, 3\}$
- $[0, \infty] = \mathbb{N}$

Alphabets are sets of symbols. If we assign every symbol a natural number (or a rank), the alphabet is ranked.

Definition A.4 (Ranked Alphabet)
A ranked alphabet is a set of symbols Σ with a function $rank \in \Sigma \mapsto \mathbb{N}$.

Example *Let $\Sigma = \{a, b\}$ and $rank \in \Sigma \mapsto \mathbb{N}$ with $rank(a) = 1$ and $rank(b) = 2$. Σ is a ranked alphabet with $rank$.*

Functions can be composite into a new function if their domain and codomain match. Functions can also be applied to a set by applying the function pointwise on every element of the set.

Definition A.5 (Composite functions)
For two functions $f \in X \mapsto Y$ and $f' \in Y \mapsto Z$, the composite function $f' \circ f \in X \mapsto Z$ is

$$f' \circ f(x) = f'(f(x)).$$

Definition A.6 (Pointwise Application)
For a function $f \in X \mapsto Y$ and a set $S \subseteq X$, the pointwise application of f on S is $\dot{f}(S) = \{f(x) \in Y \mid x \in S\}$.

Example *Let $\neg \in \mathbb{B} \mapsto \mathbb{B}$ with $\neg(true) = false$ and $\neg(false) = true$.*

- $\neg$ *composites with itself to $\neg \circ \neg = id$ where id is the identity.*

- $\neg(\{true\}) = \{false\}$ *is a pointwise application of* $\neg$.

Besides of functions and sets, we also have sequences. A sequence is a series of elements. A sequence can be of any length, can be finite or even infinite. Every element in a sequence has a position addressed by a natural number. A sequence can also be transformed into a set.

Definition A.7 (Sequences)
Let S be an arbitrary but fixed set.

1. *We call $\sigma = s_1 \dots s_m$ a finite sequence with length $|\sigma| = m$.*
2. *S^m is the set of sequences with length m with elements of a set S.*
3. *S^* is the set of all finite sequences with elements of a set S.*
4. *We call $\sigma = s_1 \dots$ an infinite sequence.*
5. *S^ω is the set of all infinite sequences with elements of a set S.*
6. *For a sequence σ, we call $\sigma_m = s_m, s_{m+1}, \dots$ the suffix of σ.*
7. *For a sequence $\sigma \in S^* \cup S^\omega$, the mth element of σ is $\sigma(m) = s_m$.*
8. *For a sequence σ, we write $[\sigma]$ to denote the set of elements in σ.*

Example *Let $\sigma = true, false, true$ be a sequence.*

- *The length of σ is $|\sigma| = 3$.*
- *$\sigma \in S^3$ holds.*
- *$\sigma \in S^*$ holds.*
- *σ is a finite sequence and not an infinite sequence.*
- *The second position of σ is $\sigma(2) = false$.*
- *The set of σ is $[\sigma] = \{true, false\}$.*

Domain Theory

Domain Theory is a part of Set Theory. Here we are interested in the sets of relations and functions. A vital domain is the partially ordered sets. A partially ordered set is reflexive, antisymmetric and transitive.

Definition A.8 (Partial Ordered Set)
Let S be a set and sqsubseteq $\in S \times S$ a relation. $(S, \sqsubseteq)$ is called a partially ordered set if and only if $(S, \sqsubseteq)$ is:

- *reflexive:*

$$\forall s \in S : \quad s \sqsubseteq s$$

- *antisymmetric:*

$$\forall s, s' \in S : \quad s \sqsubseteq s' \wedge s' \sqsubseteq s \Rightarrow s = s'$$

- *transitive:*

$$\forall s, s', s'' \in S : \quad s \sqsubseteq s' \wedge s' \sqsubseteq s'' \Rightarrow s \sqsubseteq s''$$

Example *We consider some notable partial ordered sets:*

- $(\mathbb{N}, \leq)$ *is a partially ordered set.*
- $(\mathbb{B}, \Rightarrow)$ *is a partially ordered set.*
- *For any set S, $(2^S, \subseteq)$ is a partially ordered set.*
- *For any set S, $(S, =)$ is a partially ordered set.*

Partially ordered sets are reflexive. If we remove the reflexiveness of a partially ordered set, we get the strict version of this partially ordered set.

Definition A.9 (Strict Partial Ordered Set)
Let $(S, \sqsubseteq)$ be a partially ordered set. $(S, \sqsubset)$ is the strict version of $(S, \sqsubseteq)$ if and only if

$$\forall s, s' \in S : \quad s \sqsubset s' \Leftrightarrow s \sqsubseteq s' \wedge s \neq s'$$

Example *$(\mathbb{N}, <)$ is the strict version of $(\mathbb{N}, \leq)$.*

A sequence of elements in a partially ordered set where every two neighbouring elements are ordered is a chain. If they are all ordered in one direction, the chain is ascending. If, however, they are all ordered in the opposite direction, the chain is descending. If at some point the elements are not getting larger (or the chain is finite), the chain stabilises.

Definition A.10 (Ascending Chain)
Let $(S, \sqsubseteq)$ be a partially ordered set and σ a sequence.

1. *σ is an ascending chain if and only if $\forall i \in [2, |\sigma|]\ \sigma(i-1) \sqsubseteq \sigma(i)$.*
2. *σ stabilises if and only if $\exists i \in \mathbb{N}\ \forall j \in [i, |\sigma|]\ \sigma(i) = \sigma(j)$.*

Example *Let $\sigma = 1\ 2\ 3\ 3\ 3 \ldots$ be a sequence of $(\mathbb{N}, \leq)$.*

- *σ is ascending: $1 \leq 2 \leq 3 \leq 3 \ldots$.*

- σ *stabilises:* $\forall j > 3\ \sigma(3) = \sigma(j) = 3$.

Definition A.11 (Descending Chain)
Let $(S, \sqsubseteq)$ be a partially ordered set and σ a sequence.

1. *σ is a descending chain if and only if $\forall i \in [2, |\sigma|]\ \sigma(i-1) \sqsupseteq \sigma(i)$.*
2. *σ is a strict descending chain if and only if $\forall i \in [2, |\sigma|]\ \sigma(i-1) \sqsupset \sigma(i)$.*

Example *Let $\sigma = 3\ 2\ 1\ 0$ be a sequence of $(\mathbb{N}, \leq)$.*

- *σ is descending:* $3 \leq 2 \leq 1 \leq 0$.
- *σ is strict:* $3 > 2 > 1 > 0$.

Subsets of partially ordered sets sometimes have upper bounds. A special structure where every subset has an upper bound is a complete lattice. In a complete lattice, every subset has not only at least one upper bound but exactly one least upper bound.

Definition A.12 (Complete Lattice)
Let $(S, \sqsubseteq)$ be a partially ordered set. $(S, \sqsubseteq)$ is a complete lattice if and only if there exists the least upper bound $\bigsqcup \in 2^S \mapsto S$, formally:

- *upper bound:*

$$\forall Z \subseteq S\ \forall z \in Z : \quad z \sqsubseteq \bigsqcup(Z)$$

- *least upper bound:*

$$\forall Z \subseteq S\ \forall u \in S : \quad (\forall z \in Z\ z \sqsubseteq u) \Rightarrow \bigsqcup(Z) \sqsubseteq u$$

Example *Some notable complete lattices are:*

- *For any set S, $(2^S, \subseteq)$ is a complete lattice.*
- *Any finite totally ordered set is a complete lattice.*
- *$(\mathbb{N} \cup \{\infty\}, \leq)$ is a complete lattice.*

These are not complete lattices:

- *$(\mathbb{N}, \leq)$ is not a complete lattice.*
- *$(\{0, 1\}, =)$ is not a complete lattice.*

If a function behaves well on a partial order, i.e. the order remains after applying the function, we call this function monotonic.

Definition A.13 (Monotonic Function)
Let $f \in X \mapsto Y$ be a function over the partially ordered sets $(X, \sqsubseteq)$ and $(Y, \sqsubseteq')$. f is monotonic if and only if

$$\forall x, x' \in X : \quad x \sqsubseteq x' \Rightarrow f(x) \sqsubseteq' f(x')$$

Example *Let $f \in \mathbb{N} \mapsto \mathbb{N}$ over the partially ordered set $(\mathbb{N}, \leq)$ be $f(x) = x + 4$. f is monotonic as for all $x \leq y$ we have $f(x) = x + 4 \leq y + 4 = f(x)$.*

Appendix B

Monotonic Data Flow Analysis

A typical static program analysis technique is the data flow analysis. In a data flow analysis, a system (usually a program) is transformed into a flow graph and then analysed by solving an equation system. The equation system is defined on a complete lattice via transfer functions.

A flow graph is a graph consisting of labels, extremal labels and a flow relation that connects labels.

Definition B.1 (Flow Graph)
Let Lab be a set of labels. (Lab, Ex, F) *is a flow graph where*

- $Ex \in Lab$ *is a set of extremal labels and*
- $F \in Lab \times Lab$ *is a flow.*

The data flow analysis consists of the flow graph, a complete lattice, a value for the extremal labels and a transfer functions. The transfer function maps every flow to a monotonic function. Often the transfer function is defined differently: the transfer functions maps every label to a monotonic function(see [NRNH99, p.67]), but both of these formulations are equivalent as the one can be transformed into another. Such a transformation is shown in DifinitionB.3.

Definition B.2 ((Monotonic) Data Flow Analysis Framework)
Let $\phi \in D \mapsto D$ *be a set of monotonic functions.*$(Lab, Ex, F, (D, \sqsubseteq), \iota, \varphi)$ *is a (monotonic) data flow analysis where*

- (Lab, Ex, F) *is a flow graph,*
- $(D, \sqsubseteq)$ *is a complete lattice,*
- $\iota \in D$ *is the extremal value and*
- $\varphi \in F \mapsto \phi$ *is the transfer function.*

Definition B.3 (Transformation from Flow Based to Label Based) *Let $\phi \in (D \mapsto D)$ be a set of monotonic functions. We have a data flow analysis $DFA = (Lab, Ex, F, (D, \sqsubseteq), \iota, \varphi)$ where $\varphi \in F \mapsto \phi$. The data flow analysis $DFA' = (Lab', Ex, F', (D, \sqsubseteq), \iota, \varphi')$ where $\varphi' \in Lab \mapsto \phi$ is such that every solution to the equation system of DFA' is a solution to the equation system of DFA. We define DFA' by*

- $Lab' = Lab \cup F$,
- $F' = \{(l, f) \mid f = (l, l') \in F\} \cup \{(f, l') \mid f = (l, l') \in F\}$,
- $\varphi'(k) = \begin{cases} k & \text{if } k \in Lab \\ \varphi(f) & \text{if } f \in F. \end{cases}$

A data flow analysis induces an equation system. The solution of this equation system maps every label to a value. The value is ι if the label is extremal. If the label is not extremal, we calculate the value by taking the least upper bound on the images of the transfer functions of its incoming transitions.

Definition B.4 (Equation system of a DFA[NRNH99, p.67]) *Let $DFA = (Lab, Ex, F, (D, \sqsubseteq), \iota, \varphi)$ be a data flow analysis. The equation system defined by DFA where $S \in Lab \mapsto D$ is:*

$$S(l) = \begin{cases} \iota & \text{if } l \in Ex \\ \bigsqcup\{\varphi(f)(S(l')) \mid f = (l', l) \in F\} & else \end{cases}$$

S is called a solution to the equation system.

Similar to an equation system, we can also define a transformation. The transformation works similar to the equation system. We consider a fixpoint of this transformation a solution to the equation system.

Definition B.5 (Transformation of a DFA) *Let $DFA = (Lab, Ex, F, (D, \sqsubseteq), \iota, \varphi)$ be a data flow analysis.*

$$\Phi_{DFA} \in (Lab \mapsto D) \mapsto (Lab \mapsto D)$$

is the transformation for any $S \in Lab \mapsto D$ and $l \in Lab$

$$\Phi_{DFA}(S)(l) = \begin{cases} \iota & \text{if } l \in Ex \\ \bigsqcup\{\varphi(f)(S(l')) \mid f = (l', l) \in F\} & else \end{cases}$$

The fixpoint of a transformation is a value such that applying it to the transformation does not change. As a fixpoint is a solution to the equation system of a data flow analysis, we introduce a notation to describe an arbitrary fixpoint.

Definition B.6 (Fixpoint of Transformation) *Let $DFA = (Lab, Ex, F, (D, \sqsubseteq), \iota, \varphi)$ be a data flow analysis. $fix(\Phi_{DFA})$ is a fixpoint for the function Φ_{DFA}. $lfp(\Phi_{DFA})$ is the least fixpoint for the function Φ_{DFA}.*

A path induces a sequence of transfer functions that composited and mapped to ι yields a solution for this path.

Definition B.7 (Path Solution)
Let $DFA = (Lab, Ex, F, (D, \sqsubseteq), \iota, \varphi)$ be a data flow analysis. The solution of a path π with $\forall i \in [2, |\pi|]\ (\pi(i-1), \pi(i)) = f_{i-1} \in F$ and $\pi(1) \in Ex$ is

$$\varphi(\pi) = \varphi(f_{|\pi|}) \circ \cdots \circ \varphi(f_1)(\iota).$$

The set of paths from an extremal label (which are necessary for the extremal value ι) to a label l are the paths starting in an extremal label and ending in l.

Definition B.8 (Path Set)
Let $DFA = (Lab, Ex, F, (D, \sqsubseteq), \iota, \varphi)$ be a data flow analysis. The set of all pathes to the label $l \in Lab$ is

$$\text{Path}(l) = \{\pi \mid \forall i \in [2, |\pi|]\ (\pi(i-1), \pi(i)) \in F \wedge \pi(1) \in Ex \wedge \pi(|\pi|) = l\}.$$

The least upper bound on all paths is the meet over all paths solution. The meet over all paths solution is not a fixpoint in general but is considered a solution nevertheless.

Definition B.9 (Meet over all Paths Solution[NRNH99, p.76ff.])
Let $DFA = (Lab, Ex, F, (D, \sqsubseteq), \iota, \varphi)$ be a data flow analysis. The meet over all paths solution of a data flow analysis DFA is

$$MOP_{DFA}(l) = \bigsqcup\{\varphi(\pi) \mid \pi \in \text{Path}(l)\}.$$

Both the meet over all paths and a fixpoint solution does always exist. The meet over all paths solution exists as $(D, \sqsubseteq)$ is a complete lattice. We can prove the existence of a (least) fixpoint by using fixpoint theorems.

Theorem B.1 (Existence of Solutions[NRNH99, p.72ff.])
Let $DFA = (Lab, Ex, F, (D, \sqsubseteq), \iota, \varphi)$ be a data flow analysis. There exists at least one fixpoint fix(Φ_{DFA}), and there exists the meet over all paths solution MOP_{DFA}.

The domain $(D, \sqsubseteq)$ is required to have some special properties, that are not always enforceable, to calculate a fixpoint. To avoid this problem, we use widening techniques. We use a widening operator in order to make every ascending chain into a stabilising ascending chain. Thus a fixpoint is reached in finite time by applying a fixpoint iteration.

Definition B.10 (Widening Operator[NRNH99, p.219])
Let $(D, \sqsubseteq)$ be a complete lattice. Let $\nabla \in D \times D \mapsto D$ be an operator and $\bar{\nabla} \in D^* \cup D^\omega \mapsto D$ as

$$\bar{\nabla}(d) = d$$
$$\bar{\nabla}(\pi d) = \nabla(d, \bar{\nabla}(\pi)),$$

∇ is a widening operator if and only if

- *∇ is an upper bound and*
- *for all ascending chains $\pi \in D^* \cup D^\omega$ in $(D, \sqsubseteq)$ the ascending chain $\bar{\nabla}(\pi)$ stabilises.*

The meet over all paths and the fixpoint solution are connected. The meet over all paths solution is always more precise (smaller) then the least fixpoint and the least fixpoint is (obviously) more precise (smaller) than every other fixpoint.

Theorem B.2 (MOP Solution vs Fixpoint Solution[NRNH99, p.78])
Let $DFA = (Lab, Ex, F, (D, \sqsubseteq), \iota, \varphi)$ be a data flow anaylsis. For all $l \in Lab$ and $\pi \in \mathrm{Path}(l)$

$$\varphi(\pi) \sqsubseteq MOP_{DFA}(l) \sqsubseteq \mathit{lfp}(\Phi_{DFA})(l) \sqsubseteq \mathit{fix}(\Phi_{DFA})(l).$$

Appendix C

Additionally Proofs

C.1 Analysis Proofs

Proof C.1 (Lemma 4.1) First, we prove that $(Assigns_{Rel}, \dot{\sqsubseteq}_{Rel})$ is a partially ordered set, i.e. that $\dot{\sqsubseteq}_{Rel} \in Assigns_{Rel} \times Assigns_{Rel}$ is reflexive, asymmetric and transitive over $Assigns_{Rel}$.

- Reflexive:

$$\begin{aligned} & \forall ir \in I_{Rel} : \quad ir \sqsubseteq_{Rel} ir \\ \Rightarrow & \forall a \in Assigns_{Rel} : \quad \forall e \in E \; a(e) \sqsubseteq_{Rel} a(e) \\ \Rightarrow & \forall a \in Assigns_{Rel} : \quad a \dot{\sqsubseteq}_{Rel} a \end{aligned}$$

- Asymmetric:

$$\begin{aligned} & \forall ir, jr \in I_{Rel} : \\ & ir \sqsubseteq_{Rel} jr \wedge jr \sqsubseteq_{Rel} ir \Rightarrow ir = jr \\ \Rightarrow & \begin{cases} \forall a, a' \in Assigns_{Rel} \; \forall e \in E : \\ a(e) \sqsubseteq_{Rel} a'(e) \wedge a'(e) \sqsubseteq_{Rel} a(e) \Rightarrow a(e) = a'(e) \end{cases} \\ \Rightarrow & \begin{cases} \forall a, a' \in Assigns_{Rel} : \\ a \dot{\sqsubseteq}_{Rel} a' \wedge a' \dot{\sqsubseteq}_{Rel} a \Rightarrow a = a' \end{cases} \end{aligned}$$

- Transitive:

$$\begin{aligned} & \forall ir, jr, kr \in I_{Rel} : \\ & ir \sqsubseteq_{Rel} jr \wedge jr \sqsubseteq_{Rel} kr \Rightarrow ir \sqsubseteq_{Rel} kr \\ \Rightarrow & \begin{cases} \forall a, a', a'' \in Assigns_{Rel} \; \forall e \in E : \\ a(e) \sqsubseteq_{Rel} a'(e) \wedge a'(e) \sqsubseteq_{Rel} a''(e) \Rightarrow a(e) \sqsubseteq_{Rel} a''(e) \end{cases} \\ \Rightarrow & \begin{cases} \forall a, a', a'' \in Assigns_{Rel} : \\ a \dot{\sqsubseteq}_{Rel} a' \wedge a' \dot{\sqsubseteq}_{Rel} a'' \Rightarrow a \dot{\sqsubseteq}_{Rel} a'' \end{cases} \end{aligned}$$

Now we also prove that it is a complete lattice. Let $As \subseteq Assigns_{Rel}$ be any subset of $Assigns_{Rel}$. The least upper bound $\dot{\bigsqcup}_{Rel} \in 2^{Assigns_{Rel}} \mapsto Assigns_{Rel}$

of As is

$$(\dot{\bigsqcup}_{Rel} As)(e) = \bigsqcup_{Rel}\{a(e) \mid a \in As\}.$$

We now prove that this is indeed the least upper bound by first proving that it is an upper bound and then proving that it is the least upper bound.

- Upper bound:

$$\begin{aligned}
&\forall J \subseteq I_{Rel}\ \forall jr \in J : \\
&jr \sqsubseteq \bigsqcup_{Rel} J \\
\Rightarrow &\begin{cases} \forall As \subseteq Assigns_{Rel}\ \forall a \in As\ \forall e \in E : \\ a(e) \sqsubseteq_{Rel} \bigsqcup_{Rel}\{a(e) \mid a \in As\} \end{cases} \\
\Rightarrow &\begin{cases} \forall As \subseteq Assigns_{Rel}\ \forall a \in As\ \forall e \in E : \\ a(e) \sqsubseteq_{Rel} (\dot{\bigsqcup}_{Rel} As)(e) \end{cases} \\
\Rightarrow &\begin{cases} \forall As \subseteq Assigns_{Rel}\ \forall a \in As : \\ a \dot{\sqsubseteq}_{Rel} (\dot{\bigsqcup}_{Rel} As) \end{cases}
\end{aligned}$$

- Least upper bound:

$$\begin{aligned}
&\forall J \subseteq I_{Rel}\ \forall ir \in I_{Rel} : \\
&(\forall jr \in J\, jr \sqsubseteq_{Rel} ir) \Rightarrow \bigsqcup_{Rel} J \sqsubseteq_{Rel} ir \\
\Rightarrow &\begin{cases} \forall As \subseteq Assigns_{Rel}\ \forall a \in Assigns_{Rel}\ \forall e \in E : \\ (\forall a' \in As\ a'(e) \sqsubseteq_{Rel} a(e)) \Rightarrow \bigsqcup_{Rel}\{a(e) \mid a \in As\} \sqsubseteq_{Rel} a(e) \end{cases} \\
\Rightarrow &\begin{cases} \forall As \subseteq Assigns_{Rel}\ \forall a \in Assigns_{Rel}\ \forall e \in E : \\ (\forall a' \in As\ a'(e) \sqsubseteq_{Rel} a(e)) \Rightarrow (\dot{\bigsqcup}_{Rel} As)(e) \sqsubseteq_{Rel} a(e) \end{cases} \\
\Rightarrow &\begin{cases} \forall As \subseteq Assigns_{Rel}\ \forall a \in Assigns_{Rel} : \\ (\forall a' \in As\ a' \dot{\sqsubseteq}_{Rel} a) \Rightarrow (\dot{\bigsqcup}_{Rel} As) \dot{\sqsubseteq}_{Rel} a \end{cases}
\end{aligned}$$

■

Proof C.2 (Lemma 4.2) Let $assign\ \dot{\sqsubseteq}_{Rel}\ assign'$ be two ordered elements of $Assigns_{Rel}$.

$\dot{\varphi}^m$ For a direct derivation $H \Rightarrow_r H'$ with $r = (\epsilon, R)$ and $e \in H'$:

$$\begin{aligned}
&\dot{\varphi}^m_r(assign)(e) \begin{cases} = \dot{\varphi}^m_r(assign')(e) & \text{if } e \in E_{H'} \setminus E_R \\ \sqsubseteq_{Rel} \dot{\varphi}^m_r(assign')(e) & \text{if } e \in E_R \end{cases} \quad \text{by monotonicity} \\
\Rightarrow &\dot{\varphi}^m_r(assign)(e) \sqsubseteq_{Rel} \dot{\varphi}^m_r(assign')(e)
\end{aligned}$$

Thus we have also

$$\dot{\varphi}^m_r(assign) \dot{\sqsubseteq}_{Rel} \dot{\varphi}^m_r(assign').$$

By inductive application, we have for derivations of arbitrary length $H \Rightarrow_{rs} H'$ that

$$\dot{\varphi}^m_{rs}(assign) \dot{\sqsubseteq}_{Rel} \dot{\varphi}^m_{rs}(assign').$$

$\dot{\varphi}^o$ Since $\dot{\varphi}^o_{(H,H')}$ is constant, $\dot{\varphi}^o_{(H,H')}$ is also monotonic.

$\dot{\varphi}^a$ For a direct backward derivation $H \Leftarrow_r H'$ with $r = (\epsilon, R)$ and $e \in H'$:

$$\dot{\varphi}^a_{r'}(assign)(e) \begin{cases} = \dot{\varphi}^a_{r'}(assign')(e) & \text{if } e \in E_{H'} \setminus \{\epsilon\} \\ \sqsubseteq_{Rel} \dot{\varphi}^a_{r'}(assign')(e) & \text{if } e = \epsilon \end{cases} \quad \text{by monotonicity}$$
$$\Rightarrow \dot{\varphi}^a_{r'}(assign)(e) \sqsubseteq_{Rel} \dot{\varphi}^a_{r'}(assign')(e)$$

Thus we have also

$$\dot{\varphi}^a_r(assign) \dot{\sqsubseteq}_{Rel} \dot{\varphi}^a_r(assign').$$

By inductive application, we have for backward derivations of arbitrary length $H \Leftarrow_{rs} H'$ that

$$\dot{\varphi}^a_{rs}(assign) \dot{\sqsubseteq}_{Rel} \dot{\varphi}^a_{rs}(assign').$$

$\dot{\varphi}$ Then we have for $\dot{\varphi}$ that

$$\begin{aligned} & assign \dot{\sqsubseteq}_{Rel} assign' \\ \Rightarrow & \dot{\varphi}^m_{rs}(assign) \dot{\sqsubseteq}_{Rel} \dot{\varphi}^m_{rs}(assign') \\ \Rightarrow & \dot{\varphi}^s(\dot{\varphi}^m_{rs}(assign)) \dot{\sqsubseteq}_{Rel} \dot{\varphi}^s(\dot{\varphi}^m_{rs}(assign')) \\ \Rightarrow & \dot{\varphi}^a_{rs'}(\dot{\varphi}^s(\dot{\varphi}^m_{rs}(assign))) \dot{\sqsubseteq}_{Rel} \dot{\varphi}^a_{rs'}(\dot{\varphi}^s(\dot{\varphi}^m_{rs}(assign'))) \\ \Rightarrow & \dot{\varphi}(assign) \dot{\sqsubseteq}_{Rel} \dot{\varphi}(assign') \end{aligned}$$

■

C.2 Inequality Proofs

Proof C.3 (Lemma 6.1) First, we prove that $(\text{Int}, \leq)$ is a partial order:

- Reflexive:

$$-\infty \leq -\infty$$

$$\infty \leq \infty$$

$$\forall i \in \mathbb{Z}\ i + 0 = i \Rightarrow i \leq i$$

- Asymmetric:

$$\begin{aligned} & \forall i \in \text{Int} : \\ & i \leq \infty \wedge \infty \leq i \\ \Rightarrow & \infty \leq i \\ \Rightarrow & \infty = i \end{aligned}$$

$$\begin{aligned}\forall i \in \text{Int} : i \leq -\infty \wedge -\infty \leq i \\ \Rightarrow i \leq -\infty \\ \Rightarrow i = -\infty\end{aligned}$$

$$\begin{aligned}\forall i, j \in \mathbb{Z} : \ & i \leq j \wedge j \leq i \\ & \Rightarrow \exists m \in \mathbb{N}\ i + m = j \wedge \exists m' \in \mathbb{N}\ j + m' = i \\ & \Rightarrow \exists m \in \mathbb{N}\ i + m = j \wedge \exists m' \in \mathbb{N}\ j = i - m' \\ & \Rightarrow \exists m, m' \in \mathbb{N}\ (i + m = j \wedge j = i - m' \wedge i + m = i - m') \\ & \Rightarrow \exists m, m' \in \mathbb{N}\ (i + m = j \wedge j = i - m' \wedge m = -m' = 0) \\ & \Rightarrow i = j\end{aligned}$$

- Transitive:
  Since $\forall i \in \text{Int} : \quad i \leq \infty \wedge (\infty \leq i \Rightarrow i = \infty)$ we have:

$$\begin{aligned}\forall i, j \in \text{Int} : \quad \infty \leq i \wedge i \leq j \Rightarrow \infty = i = j \\ \forall i, j \in \text{Int} : \quad i \leq \infty \wedge \infty \leq j \Rightarrow i \leq \infty = j \\ \forall i, j \in \text{Int} : \quad i \leq j \wedge j \leq \infty \Rightarrow i \leq \infty\end{aligned}$$

  Since $\forall i \in \text{Int} : \quad -\infty \leq i \wedge (i \leq -\infty \Rightarrow i = -\infty)$, we have:

$$\begin{aligned}\forall i, j \in \text{Int} : \quad -\infty \leq i \wedge i \leq j \Rightarrow -\infty \leq j \\ \forall i, j \in \text{Int} : \quad i \leq -\infty \wedge -\infty \leq j \Rightarrow i = -\infty \leq j \\ \forall i, j \in \text{Int} : \quad i \leq j \wedge j \leq -\infty \Rightarrow i = j = -\infty\end{aligned}$$

  For non-infinity integers, we have:

$$\begin{aligned}\forall i, j, k \in \text{Int} : \ & i \leq j \wedge j \leq k \\ & \Rightarrow \exists m, m' \in \mathbb{N}\ (i + m = j \wedge j + m' = k) \\ & \Rightarrow \exists m, m' \in \mathbb{N}\ i + m + m' = k \\ & \Rightarrow i \leq k\end{aligned}$$

Now we prove that $(\text{Int}, \leq)$ is a complete lattice. For this, we show that the least upper bound is

$$\max \in 2^{\text{Int}} \mapsto \text{Int}$$

$$\max(is) = \begin{cases} i & \text{if } i \in is \wedge \forall j \in is\ j \leq i \\ -\infty & \text{if } is = \emptyset \\ \infty & \text{else.} \end{cases}$$

- Upper bound:
  - If $\max(is) = i$ then $\forall j \in is\ j \leq i$ by the condition.

  - If $\max(is) = -\infty$ then $is = \emptyset$ and thus every element is an upper bound.
  - If $\max(is) = \infty$ then $\forall i \in is \subseteq \text{Int}\ i \leq \max(is)$.

- Least Upper Bound:
  - If $\max(is) = i$ then $i \in is$, thus i is the least upper bound.
  - If $\max(is) = -\infty$ then $is = \emptyset$ and thus the least element $-\infty$ is the least upper bound.
  - If $\max(is) = \infty$ then ∞ would be the only upper bound if it is the least upper bound. Assume thus any upper bound k of is and assume that $is \neq \emptyset$ and no element $i \in is$ is an upper bound. We have for all $j \in is$ that $j \leq k$ and thus $j + m = k$. If no m is zero, then also $k - 1$ is an upper bound. By repeating this, at some point, we reach an element $k' \in is$ that is an upper bound. $k' \in is$ contradicts the assumption that there is no such element. Thus ∞ is the only upper bound.

Next, we prove that $(\text{Int}_{Rel}, \leq_{Rel})$ is a partial order:

- Reflexive:
  ∞ and $-\infty$ are similar to $\leq$.

$$\begin{aligned} & \forall (vs, z) \text{Int}_{Rel} : \\ & vs \subseteq vs \wedge z \leq z \\ \Rightarrow\ & (vs, z) \leq_{Rel} (vs, z) \end{aligned}$$

- Asymmetric:
  ∞ and $-\infty$ are similar to $\leq$.

$$\begin{aligned} & \forall (vs, z), (vs', z') \in \text{Int}_{Rel} : \\ & (vs, z) \leq_{Rel} (vs', z') \wedge (vs', z') \leq_{Rel} (vs, z) \\ \Rightarrow\ & vs \subseteq vs' \wedge z \leq z' \wedge vs' \subseteq vs \wedge z' \leq z \\ \Rightarrow\ & vs = vs' \wedge z = z' \\ \Rightarrow\ & (vs, z) = (vs', z') \end{aligned}$$

- Transitive:
  ∞ and $-\infty$ are similar to $\leq$.

$$\begin{aligned} & \forall (vs, z), (vs', z'), (vs'', z'') \in \text{Int}_{Rel} : \\ & (vs, z) \leq_{Rel} (vs', z') \wedge (vs', z') \leq_{Rel} (vs'', z'') \\ \Rightarrow\ & vs \subseteq vs' \wedge z \leq z' \wedge vs' \subseteq vs'' \wedge z' \leq z'' \\ \Rightarrow\ & vs \subseteq vs'' \wedge z \leq z'' \\ \Rightarrow\ & (vs, z) \leq_{Rel} (vs'', z'') \end{aligned}$$

Next, we show that $(\mathrm{Int}_{Rel}, \leq_{Rel})$ is a complete lattice. For this, we define the following where $\bigcup$ is the least upper bound of $\subseteq$ and max is the least upper bound of $\leq$:

$$\forall irs \subseteq 2^{\mathrm{Int}_{Rel}} \ \bigcup(irs) = \bigcup(\{vs \mid (vs, z) \in irs\})$$
$$\forall irs \subseteq 2^{\mathrm{Int}_{Rel}} \ \max(irs) = \max(\{z \mid (vs, z) \in irs\})$$

We show that $(\mathrm{Int}_{Rel}, \leq_{Rel})$ is a complete lattice by showing that the least upper bound is

$$\overset{Rel}{\bigcup} \in 2^{\mathrm{Int}_{Rel}} \mapsto \mathrm{Int}_{Rel}$$
$$\overset{Rel}{\bigcup}(irs) = \begin{cases} \infty & \text{if } \infty \in irs \\ \infty & \text{if } \max(irs) = \infty \\ -\infty & \text{if } irs \subseteq \{-\infty\} \\ (\bigcup(irs), \max(irs)) & \text{else} \end{cases}$$

- Upper bound:
  - If $\overset{Rel}{\bigcup}(irs) = \infty$ then $\forall ir \in irs \subseteq \mathrm{Int}_{Rel} \ ir \leq_{Rel} \overset{Rel}{\bigcup}(irs)$.
  - If $\overset{Rel}{\bigcup}(irs) = -\infty$ then $irs \subseteq \{-\infty\}$ and $-\infty \leq \overset{Rel}{\bigcup}(irs)$.
  - If $\overset{Rel}{\bigcup}(irs) = (\bigcup(irs), \max(irs))$ then $\infty \notin irs$. We have for the least element $-\infty \leq (\bigcup(irs), \max(irs))$. For every other element $(vs, z) \in irs$ we have:

$$vs \subseteq \bigcup(irs) \wedge z \leq \max(irs)$$
$$\Rightarrow (vs, z) \leq_{Rel} (\bigcup(irs), \max(irs))$$ ∎

- Least Upper Bound:
  - If $\overset{Rel}{\bigcup}(irs) = \infty$ then either
    - $\infty \in irs$ thus ∞ is the only upper bound
    - or $\max(irs) = \infty$ thus neither $-\infty$ nor any (vs, z) is an upper bound, thus again ∞ is the only upper bound.
  - If $\overset{Rel}{\bigcup}(irs) = -\infty$ then as $-\infty$ is the least element, every other upper bound has to be at least $-\infty$, and thus $-\infty$ is the least upper bound.
  - If $\overset{Rel}{\bigcup}(irs) = (\bigcup(irs), \max(irs))$ then for all other upper bounds (vs, z) we have $\bigcup(irs) \subseteq vs$ and $\max(irs) \leq z$. Thus $\overset{Rel}{\bigcup}(irs)$ is the least.

Proof C.4 (Lemma 6.2) We prove every condition in Definition 3.7 on after another:

(3.9) Let $v \in \text{Var}$, then we have $\forall \nu \in \mathcal{V}\ [\![(\{v\}, 0)]\!]_\nu = \nu(v)$.

(3.10) Three cases:

- Let $i \in \mathbb{Z}$; then we have $\forall \nu \in \mathcal{V}\ [\![(\emptyset, i)]\!]_\nu = i$.
- We have $\forall \nu \in \mathcal{V}\ [\![\infty]\!]_\nu = \infty$.
- We have $\forall \nu \in \mathcal{V}\ [\![-\infty]\!]_\nu = -\infty$.

(3.11) Three cases:

- For $i <_{Rel} \infty$ we have $\infty \neq [\![i]\!]_\nu < \infty$.
- For $-\infty <_{Rel} i$ we have $-\infty < [\![i]\!]_\nu \neq -\infty$.
- For $(vs, z) <_{Rel} (vs', z')$ we have

$$\begin{aligned}
&(vs, z) <_{Rel} (vs', z') \\
\Rightarrow &(vs \subset vs' \wedge z \leq z') \vee (vs \subseteq vs' \wedge z < z') \\
\Rightarrow &(\sum_{v \in vs} \nu(v) < \sum_{v \in vs'} \nu(v) \wedge z \leq z') \vee (\sum_{v \in vs} \nu(v) \leq \sum_{v \in vs'} \nu(v) \wedge z < z') \\
\Rightarrow &z + \sum_{v \in vs} \nu(v) < z' + \sum_{v \in vs'} \nu(v) \\
\Rightarrow &[\![(vs, z)]\!]_\nu < [\![(vs', z')]\!]_\nu
\end{aligned}$$

■

Proof C.5 (Lemma 6.3) We make a case distinction on i and j:

- If $i = \infty$ then $[\![i \oplus_{Rel} j]\!]_\nu = [\![\infty]\!]_\nu = \infty \geq [\![i]\!]_\nu \oplus [\![j]\!]_\nu$. $j = \infty$ is analogous.
- If $i = -\infty$ and $j \neq \infty$ then $[\![i]\!]_\nu \oplus [\![j]\!]_\nu = -\infty = [\![-\infty]\!]_\nu = [\![i \oplus_{Rel} j]\!]_\nu$. $i \neq \infty$ and $j = -\infty$ are analogous.
- If $i = (vs_i, z_i), j = (vs_j, z_j)$, we have different cases:
  - If $vs_i \cap vs_j \neq \emptyset$ then $[\![i \oplus_{Rel} j]\!]_\nu = [\![\infty]\!]_\nu = \infty \geq [\![i]\!]_\nu \oplus [\![j]\!]_\nu$.
  - If $vs_i \cap vs_j = \emptyset$ then

$$\begin{aligned}
&[\![i]\!]_\nu \oplus [\![j]\!]_\nu \\
=&[\![(vs_i, z_i)]\!]_\nu \oplus [\![(vs_j, z_j)]\!]_\nu && \text{assumption} \\
=&z_i + \sum_{v \in vs_i} \nu(v) + z_j + \sum_{v \in vs_j} \nu(v) \\
=&z_i + z_j + \sum_{v \in vs_i} \nu(v) + \sum_{v \in vs_j} \nu(v) && \text{commutative} \\
=&z_i + z_j + \sum_{v \in vs_i \cup vs_j} \nu(v) && \text{since } vs_i \cap vs_j = \emptyset \\
=&[\![(vs_i \cup vs_j, z_i + z_j)]\!]_\nu && \text{replacing by } [\![\cdot]\!]_\nu \\
=&[\![i \oplus_{Rel} j]\!]_\nu
\end{aligned}$$

■

Proof C.6 (Lemma 6.4) We have multiple cases for $k \geq i \oplus j$. First let $i \in \mathbb{N}$:

- If $j = -\infty$ then $k \geq k \oplus (-i) \geq -\infty = i \oplus j = j$.
- If $k = -\infty, j \neq -\infty$ then $k = -\infty \not\geq i \oplus j$.
- If $k \in \mathbb{Z}, j \in \mathbb{Z}$ then trivially $k \geq i + j \Leftrightarrow k - i \geq j$.
- If $k = \infty$ then $k \oplus (-i) = \infty = k \geq i \oplus j \geq j$.
- If $k \neq \infty, j = \infty$ then $k \not\geq \infty = i \oplus j$. ∎

Not we let $i = \infty$:

- If $k = -\infty$ then $k = -\infty \not\geq \infty = i \oplus j$.
- If $k \in \mathbb{Z}$ then $k \not\geq \infty = i \oplus j$.
- If $k = \infty$ then $k = \infty = i \oplus j$ and $k \oplus (-i) = \infty \geq j$.

Proof C.7 (Lemma 6.5) To prove (3.3), we let $i, j \in \text{Int}$ and $i \geq j$:

- if $A \in T$ then by (6.2) we have if $PROP(i)(A^\bullet)$ then $PROP(j)(A^\bullet)$;
- if $A \in N$ then by (6.1) we have for every $A^\bullet \Rightarrow R \Rightarrow^* H$ that

$$\bigwedge_{(E',c) \in iq_R} i \geq j \geq \sum_{e \in E'} assign(e) + c.$$

  Therefore if $PROP'(j)(assign, R)$ then also $PROP'(i)(assign, R)$. By compatibility of $PROP$, we also have $PROP(j)(H) \Rightarrow PROP(i)(H)$.

We prove condition (3.4) by structural induction on every derivation.

Induction Base: For every terminal A, there exists an index $i \in \mathbb{N}$ such that $PROP(i)(A^\bullet)$ by (6.2).

Induction Hypothesis: For a derivation $A^\bullet \Rightarrow R \Rightarrow^* H \in \mathcal{H}_T$, there exists a function $assign \in E_R \mapsto \mathbb{N}$ such that

$$\forall e \in E_R \; PROP(assign(e))(H(e)).$$

Induction Step: By compatibility, we have $PROP(i)(H)$ if and only if there exists a function $assign \in E_R \mapsto \text{Int}$ such that

$$PROP'(i)(assign, R) \wedge \forall e \in E_R \; PROP(assign(e))(H(e)).$$

By the induction hypothesis, such function $assign \in E_R \mapsto \mathbb{N}$ exists. It is left to prove that $PROP'(i)(assign, R)$ holds for some $i \in \mathbb{N}$. For

$$i = \max_{(E',c) \in iq_R} \big(\sum_{e \in E'} assign(e) + c \big)$$

we have that

$$\bigwedge_{(E',c) \in iq_R} i \geq \sum_{e \in E'} assign(e) + c,$$

and thus $PROP'(i)(assign, R)$ holds. ∎

PROOF C.8 (LEMMA 6.7) We assume $\sigma \in (\mathrm{Int})^\omega$ is a strict descending chain. As then we have $\sigma(m) > \sigma(m+1)$ for all $m \in \mathbb{N}$, we have that after at least $\sigma(m)$ many steps, the integer is non-positive. Thus we now prove that every non-positive integer is terminating.

We prove this by structural induction over all derivations.

Induction Base: For every hyperedge e with terminal symbol $label(e) \in T$ there exists the minimal index $i > 0$ such that $PROP(i)(e^\bullet)$ by (6.2). Therefore $\forall j \leq 0\ \neg PROP(j)(e^\bullet)$.

Induction Hypothesis: For a derivation $A^\bullet \Rightarrow R \Rightarrow^* H \in \mathcal{H}_T$ there exists a function $assign \in E_R \mapsto \mathbb{N}$ such that

$$\forall e \in E_R : PROP(assign(e))(H(e)) \wedge assign(e) > 0.$$

Induction Step: By 6.1 we have $PROP(i)(H)$ if and only if there exists a function $assign \in E_R \mapsto \mathbb{N} \cup \{\infty\}$ such that

$$PROP'(i)(assign, R) \wedge \forall e \in E_R\ PROP(assign(e))(H(e)).$$

By the induction hypothesis, such a function $assign$ exists. It is left to prove that $PROP'(i)(assign, R)$ holds for some $i > 0$. For

$$i = \max_{(E',c) \in iq_R} \sum_{e \in E'_R} assign(e) + c,$$

we have that

$$\bigwedge_{(E',c) \in iq_R} i \geq \sum_{e \in E'_R} assign(e) + c,$$

and thus $PROP'(i)(assign, R)$ holds. If $E' = \emptyset$ then $c > 0$ and therefore $i > 0$. If $c = 0$ then $E'_R \supset \emptyset$ and therefore $\sum_{e \in E'_R} assign(e) > 0$, thus $i > 0$. ∎

PROOF C.9 (LEMMA 6.8) First, we prove that φ^a_r is an index abstraction rule.

$$\varphi^a_r(assign_{Rel}) = \bigcup_{(E',c) \in iqs_R}^{Rel} \sum_{e \in E'}^{Rel} assign_{Rel}(e) \oplus_{Rel} c$$

$$\Rightarrow \bigwedge_{(E',c) \in iqs_R} \varphi^a_r(assign_{Rel}) \geq_{Rel} \sum_{e \in E'}^{Rel} assign_{Rel}(e) \oplus_{Rel} c \qquad \text{by upper bound}$$

$$\Rightarrow \bigwedge_{(E',c) \in iqs_R} [\![\varphi^a_r(assign_{Rel})]\!]_\nu \geq [\![\sum_{e \in E'}^{Rel} assign_{Rel}(e) \oplus_{Rel} c]\!]_\nu \qquad \text{by Lemma 6.2}$$

$$\Rightarrow \bigwedge_{(E',c) \in iqs_R} [\![\varphi^a_r(assign_{Rel})]\!]_\nu \geq \sum_{e \in E'} [\![assign_{Rel}(e)]\!]_\nu \oplus [\![c]\!]_\nu \qquad \text{by Lemma 6.3}$$

$$\Rightarrow \bigwedge_{(E',c) \in iqs_R} [\![\varphi^a_r(assign_{Rel})]\!]_\nu \geq \sum_{e \in E'} [\![assign_{Rel}]\!]_\nu(e) \oplus c \qquad \text{since } [\![c]\!]_\nu = c$$

$$\Rightarrow PROP'([\![\varphi^a_r(assign_{Rel})]\!]_\nu)([\![assign_{Rel}]\!]_\nu, R)$$

Next, we also show that φ_r^m is an index abstraction rule. $PROP'(\infty)(assign, R)$ holds every time, which proves the first part.
For the second part, we assume that there exists $\varphi_r^m(ir)(e')$ such that the statement $PROP([\![\varphi_r^m(ir)]\!]_\nu(e'))(H(e'))$ does **not** hold.
We have by Lemma 3.1 $[\![\varphi_r^m(ir)]\!]_\nu(e') \neq \infty$. We have $e' \notin T$ as otherwise $[\![\varphi_r^m(ir)(e')]\!]_\nu = [\![(\emptyset, i)]\!]_\nu = i$ where $PROP(i)(H(e'))$ holds, which contradicts the assumption that $PROP(i)(H(e'))$ does not hold.
Then by additionally condition (3.4) there exists at least one greater positive integer $j > [\![\varphi_r^m(ir)]\!]_\nu(e')$ with $PROP(j)(H(e'))$. We assume j to be minimal. In the following, we conclude that $j \leq [\![\varphi_r^m(ir)]\!]_\nu(e')$ holds, which would be a contradiction to $j > [\![\varphi_r^m(ir)]\!]_\nu(e')$.

$$
\begin{aligned}
& PROP([\![ir]\!]_\nu)(H) \\
\Leftrightarrow & \exists assign \in E \mapsto \text{Int} \\
& PROP'([\![ir]\!]_\nu)(assign, R) && \text{by compatibility} \\
& \wedge \forall e \in E_R \; PROP(assign(e))(H(e)) \\
\Leftrightarrow & \exists assign \in E \mapsto \text{Int} \\
& \bigwedge_{(E',c) \in iqs_R} [\![ir]\!]_\nu \geq \sum_{e \in E'} assign(e) \oplus c && \text{Definition of } PROP' \\
& \wedge \forall e \in E_R \; PROP(assign(e))(H(e))
\end{aligned}
$$

We have that $c \geq 0$ and $\forall e \in E' \; assign(e) > 0$ because $assign(e)$ cannot be terminating. Thus we can apply Lemma 6.4:

$$
\bigwedge_{(E',c) \in iqs_R, e' \in E'} [\![ir]\!]_\nu \oplus \sum_{e \in E' \setminus \{e'\}} -assign(e) \oplus (-c) \geq assign(e')
$$

Since j is the minimal integer with $PROP(j)(H(e'))$, we have $assign(e') \geq j$ and thus

$$
\bigwedge_{(E',c) \in iqs_R, e' \in E'} [\![ir]\!]_\nu \oplus \sum_{e \in E' \setminus \{e'\}} -assign(e) \oplus (-c) \geq j
$$

Now let $(E', c) \in iq_R$ be arbitrary with $e' \in E'$:

$$
\begin{aligned}
j \leq & [\![ir]\!]_\nu \oplus \sum_{e \in E' \setminus \{e'\}} -assign(e) \oplus (-c) \\
\leq & [\![ir]\!]_\nu \oplus \sum_{e \in E' \setminus \{e'\}} -1 \oplus (-c) \\
= & [\![ir]\!]_\nu \oplus (-|E' \setminus \{e'\}|) \oplus (-c) \\
= & [\![ir \oplus_{Rel} (-|E' \setminus \{e'\}| - c)]\!]_\nu \\
\leq & [\![\varphi_r^m(i)(e')]\!]_\nu
\end{aligned}
$$

If, however, there does not exist any $(E', c) \in iq_R$ with $e' \in E'$ then we have $\varphi_r^m(i)(e') = \infty$. In both cases, we have $j \leq [\![\varphi_r^m(i)(e')]\!]_\nu$ which contradicts the assumption.

Lastly, we prove that φ_r^m and φ_r^a are monotone. We first prove that if $ir \leq_{Rel} jr$ then $ir \oplus_{Rel} kr \leq_{Rel} jr \oplus_{Rel} kr$ for $kr \in \text{Int}_{Rel}$ by a case distinction on ir, jr and kr:

- If $kr = \infty$ then $ir \oplus_{Rel} kr = \infty = jr \oplus_{Rel} kr$.
- If $kr = -\infty$ and $ir, jr \neq \infty$ then $ir \oplus_{Rel} kr = \infty = jr \oplus_{Rel} kr$.
- If $ir = \infty$ then $jr = \infty$ thus trivially $ir \oplus_{Rel} kr = jr \oplus_{Rel} kr$.
- If $ir = -\infty$ and $kr \neq \infty$ then $ir \oplus_{Rel} kr = -\infty \leq_{Rel} jr \oplus_{Rel} kr$.
- If $jr = -\infty$ then $ir = -\infty$ thus trivially $ir \oplus_{Rel} kr = jr \oplus_{Rel} kr$.
- If $jr = \infty$ then $ir \oplus_{Rel} kr \leq_{Rel} \infty = jr \oplus_{Rel} kr$.
- If $ir = (vs_i, z_i)$, $jr = (vs_j, z_j)$ and $kr = (vs_k, z_k)$ and $vs_j \cap vs_k \neq \emptyset$ then $ir \oplus_{Rel} kr \leq_{Rel} \infty = jr \oplus_{Rel} kr$.
- if $ir = (vs_i, z_i)$, $jr = (vs_j, z_j)$ and $kr = (vs_k, z_k)$ and $vs_j \cap vs_k = \emptyset = vs_j \cap vs_k$ then:

$$\begin{aligned} & ir \oplus_{Rel} kr \\ =& (vs_i \cup vs_k, z_i + z_k) \\ \leq_{Rel} & (vs_j \cup vs_k, z_j + z_k) && \text{monotonicty of } \cup, + \\ =& jr \oplus_{Rel} kr \end{aligned}$$

We continue with the monotonicity of φ_r^m. There are three cases for φ_r^m. The case is picked independent of the index; thus, it is sufficient to prove that every case is monotone:

- $label(e) \in T$: as $(\emptyset, i)$ is constant for e, we have $\varphi_r^m(ir)(e) = \varphi_r^m(jr)(e)$.
- $iqs_p(e) \neq \emptyset \wedge label(e) \notin T$: The variable set is not changed; thus, the element with the maximal integer is the least upper bound. Assume that the least upper bound for ir is

$$ir \oplus_{Rel} (-|E_i' \setminus \{e\}| - c_i) = ir \oplus_{Rel} (-c_i)$$

and for jr is

$$jr \oplus_{Rel} (-|E_j' \setminus \{e\}| - c_j) = r \oplus_{Rel} (-c_j)$$

such that $ir \oplus_{Rel} (-c_i) \not\leq_{Rel} jr \oplus_{Rel} (-c_j)$:

$$\begin{aligned} & ir \oplus_{Rel} (-c_i) \not\leq_{Rel} jr \oplus_{Rel} (-c_j) \\ \Rightarrow\quad & (vs_i, z_i) \oplus_{Rel} (-c_i) \not\leq_{Rel} (vs_j, z_j) \oplus_{Rel} (-c_j) && \text{rewrite as tuple} \\ \Rightarrow\quad & (vs_i, z_i - c_i) \not\leq_{Rel} (vs_j, z_j - c_j) && \text{solve } \oplus_{Rel} \\ \Rightarrow\quad & vs_i \not\subseteq vs_j \vee z_i - c_i \not\leq z_j - c_j && \text{solve } \geq_{Rel} \\ \Rightarrow\quad & z_i \leq z_j \wedge z_i - c_i \not\leq z_j - c_j && \text{by } ir \geq_{Rel} jr \\ \Rightarrow\quad & z_i \leq z_j \wedge z_i - c_i > z_j - c_j && \leq \text{ is total} \\ \Rightarrow\quad & -c_i > -c_j && \text{solving} \\ \Rightarrow\quad & ir - c_i > ir - c_j && \text{solving} \end{aligned}$$

Thus $ir - c_i$ is not the least upper bound, which is a contradiction to the assumption.

- *else*: As ∞ is constant for e, we have $\varphi_r^m(ir)(e) = \varphi_r^m(jr)(e)$.

Since the function φ_r^m is monotone for every e, the function is also monotone as a whole.
Next, we prove that φ_r^a is monotone. Assume that we have $ass_i \dot{\leq}_{Rel} ass_j$. For the tuple $ass_i(e)$, we notate the variable set as $ass_i(e)_v$ and the additional integer as $ass_i(e)_z$. First, we prove the monotonicity for every $(E', c) \in iqs_p$ by induction on E'.

IB Let $E = \emptyset$. Then the function consisting only of the constant c is monotone.

IH For a fixed but arbitrary set E', we assume $\sum_{e\in E}^{Rel} assign_{Rel}(e) \oplus_{Rel} c$ is monotone.

IS Let $e' \notin E$:

$$
\begin{aligned}
&\sum_{e\in E\cup\{e'\}}^{Rel} ass_i(e) \oplus_{Rel} c \\
&= ass_i(e') \oplus_{Rel} \sum_{e\in E}^{Rel} ass_i(e) \oplus_{Rel} c \\
&\leq_{Rel} ass_i(e') \oplus_{Rel} \sum_{e\in E}^{Rel} ass_j(e) \oplus_{Rel} c && \text{IH} \\
&\leq_{Rel} ass_j(e') \oplus_{Rel} \sum_{e\in E}^{Rel} ass_j(e) \oplus_{Rel} c && \oplus_{Rel} \text{ monotonicty} \\
&= \sum_{e\in E\cup\{e'\}}^{Rel} ass_j(e) \oplus_{Rel} c
\end{aligned}
$$

Thus, also for the least upper bound of all $(E', c) \in iqs_p$, we have

$$
\bigcup_{(E',c)\in iqs_p}^{Rel} \sum_{e\in E'}^{Rel} ass_i(e) \oplus_{Rel} c \leq_{Rel} \bigcup_{(E',c)\in iqs_p}^{Rel} \sum_{e\in E'}^{Rel} ass_j(e) \oplus_{Rel} c.
$$

■

Proof C.10 (Lemma 6.9) First, we show the condition (5.6). We prove this by induction over irs. Remark that every non-positive index is terminating.

IB Let $irs = \{ir\}$. Then $\Sigma(ir) = ir$, thus $[\![\Sigma(ir)]\!]_\nu = [\![ir]\!]_\nu \leq 0$.

IH For a fixed but arbitrary set $\emptyset \neq irs \subseteq \mathrm{Int}_{Rel}$, we assume $[\![\Sigma(irs)]\!]_\nu \leq 0 \Rightarrow \exists ir \in irs\ [\![ir]\!]_\nu \leq 0$.

IS We prove that the theorem holds for all $irs \cup \{ir\}$ with $ir \notin irs$. We assume that $[\![\Sigma(irs \cup \{ir\})]\!]_\nu = [\![\Sigma(irs) \oplus_{Rel} ir]\!]_\nu \leq 0$.

– If $[\![\Sigma(irs)]\!]_\nu \leq 0$ then by induction hypothesis

$$\exists jr \in irs \; [\![jr]\!]_\nu \leq 0,$$

thus also

$$\exists jr \in irs \cup \{ir\} \; [\![jr]\!]_\nu \leq 0.$$

– If $ir = -\infty$ the condition holds trivially.

– If $\Sigma(irs) = \infty$ or $ir = \infty$ then

$$\infty = [\![\Sigma(irs)]\!]_\nu \oplus [\![ir]\!]_\nu \overset{\text{Lemma 6.3}}{\leq} [\![\Sigma(irs) \oplus_{Rel} ir]\!]_\nu,$$

which contradicts the assumption $[\![\Sigma(irs) \oplus_{Rel} ir]\!]_\nu \leq 0$.

– If $0 < [\![\Sigma(irs)]\!]_\nu \neq \infty$, $-\infty \neq ir \neq \infty$ and $[\![\Sigma(irs \cup \{ir\})]\!]_\nu \leq 0$, we have

$$\begin{gathered}\Sigma(irs) = (vs_\Sigma, z_\Sigma),\\ ir = (vs, z),\\ z_\Sigma + \sum_{v \in vs_\Sigma} \nu(v) > 0,\\ vs_\Sigma \cap vs = \emptyset.\end{gathered}$$

Then we have:

$$\begin{aligned}
&[\![\Sigma(irs \cup \{ir\})]\!]_\nu \leq 0 && \\
\Rightarrow &[\![\Sigma(irs) \oplus_{Rel} ir]\!]_\nu \leq 0 && \text{by } \Sigma \\
\Rightarrow &[\![(vs_\Sigma \cup vs, z_\Sigma + z)]\!]_\nu \leq 0 && \text{by } \oplus_{Rel} \\
\Rightarrow &z_\Sigma + z + \sum_{v \in vs_\Sigma \cup vs} \nu(v) \leq 0 && \text{resolving } [\![\cdot]\!]_\nu \\
\Rightarrow &\big(z_\Sigma + \sum_{v \in vs_\Sigma} \nu(v)\big) + \big(z + \sum_{v \in vs} \nu(v)\big) \leq 0 && \text{commutative of } + \\
\Rightarrow &z + \sum_{v \in vs} \nu(v) \leq 0 && \text{as } z_\Sigma + \sum_{v \in vs_\Sigma} \nu(v) > 0 \\
\Rightarrow &[\![ir]\!]_\nu \leq 0 && \text{reapplying } [\![\cdot]\!]_\nu
\end{aligned}$$

Now we prove that Σ satisfies the condition (5.8). We prove this by induction on the set es.

IB Let $es = \{e\}$. Then

$$\begin{aligned}
&[\![\Sigma(\phi(es))]\!]_\nu \\
=&[\![\phi(e)]\!]_\nu \\
=&\nu^\phi(\iota(e)) \\
=&[\![\iota(e)]\!]_{\nu^\phi} \\
=&[\![\Sigma(\iota(es))]\!]_{\nu^\phi}.
\end{aligned}$$

IH For a fixed but arbitrary set $es \subseteq E$ assume $[\![\Sigma(\phi(es))]\!]_\nu = [\![\Sigma(\iota(es))]\!]_{\nu^\phi}$.

IS We prove that the hypothesis also holds for all $es \cup \{e\}$ with $e \notin es$. We assume now the premise of condition (5.8):

$$\Sigma(\iota(es \cup \{e\})) >_{Rel} \Sigma(\phi(es \cup \{e\})).$$

We make a case distinction:

- If $\Sigma(\phi(es \cup \{e\})) = \infty$ then the assumption

  $$\Sigma(\iota(es \cup \{e\})) >_{Rel} \Sigma(\phi(es \cup \{e\})$$

  cannot hold. Notably, we have this case if
  - $\Sigma(\phi(es)) = \infty$
  - $\phi(e) = \infty$
  - $\phi(e) = (vs, z)$ and $\Sigma(\phi(es)) = (vs_{\Sigma,z_\Sigma})$ with $vs \cap vs_\Sigma \neq \emptyset$.
- If $\phi(e) = -\infty, \Sigma(\phi(es)) \neq \infty$ or $\phi(e) \neq \infty, \Sigma(\phi(es)) = -\infty$ then $\Sigma(\phi(es \cup \{e\})) = -\infty$. Thus there exists $e' \in es \cup \{e\}$ such that $\phi(e') = -\infty$. Therefore $\nu^\phi(\iota(e')) = -\infty$ and thus

  $$[\![\Sigma(\iota(es \cup \{e\}))]\!]_{\nu^\phi} = -\infty = [\![\Sigma(\phi(es \cup \{e\}))]\!]_\nu.$$

- If $\phi(e) = (vs, z)$ and $\Sigma(\phi(es)) = (vs_{\Sigma,z_\Sigma})$ with $vs \cap vs_\Sigma = \emptyset$ then:

$$\begin{aligned}
&[\![\Sigma(\phi(es \cup \{e\}))]\!]_\nu \\
=&[\![\Sigma(\phi(es)) \oplus_{Rel} \phi(e)]\!]_\nu && \text{resolving } \Sigma \\
=&[\![(vs_\Sigma \cup vs, z_\Sigma + z)]\!]_\nu && \text{resolving } \oplus_{Rel} \\
=&z_\Sigma + z + \sum_{v \in vs_\Sigma \cup vs} \nu(v) && \text{resolving } [\![\cdot]\!]_\nu \\
=&\big(z_\Sigma + \sum_{v \in vs_\Sigma} \nu(v)\big) + \big(z + \sum_{v \in vs} \nu(v)\big) && \text{commutative of } + \\
=&[\![\Sigma(\phi(es))]\!]_\nu \oplus [\![\phi(e)]\!]_\nu \\
=&[\![\Sigma(\iota(es))]\!]_{\nu^\phi} \oplus [\![\phi(e)]\!]_\nu && \text{IH} \\
=&[\![\Sigma(\iota(es))]\!]_{\nu^\phi} \oplus [\![\iota(e)]\!]_{\nu^\phi} && \text{see Definition 5.1} \\
=&[\![\Sigma(\iota(es)) \oplus_{Rel} \iota(e)]\!]_{\nu^\phi} && \text{see Lemma 6.3} \\
=&[\![\Sigma(\iota(es \cup \{e\}))]\!]_{\nu^\phi} && \text{combining } \Sigma
\end{aligned}$$

■

Bibliography

[AJK+18] Hannah Arndt, Christina Jansen, Joost-Pieter Katoen, Christoph Matheja, and Thomas Noll. Let this graph be your witness! In Hana Chockler and Georg Weissenbacher, editors, *Computer Aided Verification*, pages 3–11, Cham, 2018. Springer International Publishing.

[AJMN18] Hannah Arndt, Christina Jansen, Christoph Matheja, and Thomas Noll. Graph-based shape analysis beyond context-freeness. In Einar Broch Johnsen and Ina Schaefer, editors, *Software Engineering and Formal Methods*, pages 271–286, Cham, 2018. Springer International Publishing.

[BBC08] James Brotherston, Richard Bornat, and Cristiano Calcagno. Cyclic proofs of program termination in separation logic. *SIGPLAN Not.*, 43(1):101–112, January 2008.

[BDP11] James Brotherston, Dino Distefano, and Rasmus Lerchedahl Petersen. Automated cyclic entailment proofs in separation logic. In *Proceedings of the 23rd International Conference on Automated Deduction*, CADE'11, pages 131–146, Berlin, Heidelberg, 2011. Springer-Verlag.

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. The MIT Press, Cambridge, London, Jan 2008.

[CDG+07] H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. Available on: http://www.grappa.univ-lille3.fr/tata, 2007. release October, 12th 2007.

[CDNQ12] Wei-Ngan Chin, Cristina David, Huu Hai Nguyen, and Shengchao Qin. Automated verification of shape, size and bag properties via user-defined predicates in separation logic. *Science of Computer Programming*, 77(9):1006 – 1036, 2012. The Programming Languages track at the 24th ACM Symposium on Applied Computing (SAC'09).

[Cop04] Jack Copeland. *The Essential Turing: Seminal Writings in Computing, Logic, Philosophy, Artificial Intelligence, and Artificial Life: Plus the Secrets of Enigma*. Oxford University Press, 2004.

[Dav58] Martin D. Davis. *Computability and Unsolvability.* McGraw-Hill Series in Information Processing and Computers. McGraw-Hill, 1958.

[Hab92] Annegret Habel. *Hyperedge Replacement: Grammars and Languages*, volume 643 of *Lecture Notes in Computer Science.* Springer-Verlag, Berlin, Heidelberg, 1992.

[HJKN15] Jonathan Heinen, Christina Jansen, Joost-Pieter Katoen, and Thomas Noll. Juggrnaut: using graph grammars for abstracting unbounded heap structures. *Formal Methods in System Design*, 47(2):159–203, Oct 2015.

[Jan17] Christina Jansen. *Static Analysis of Pointer Programs - Linking Graph Grammars and Separation Logic.* PhD thesis, RWTH Aachen University, Germany, 2017.

[Kle52] Stephen Kleene. *Introduction to Metamathematics.* Van Nostrand, New York, 1952.

[Knu97] Donald Ervin Knuth. *The Art of Computer Programming, Volume I: Fundamental Algorithms, 3rd Edition.* Addison-Wesley, 1997.

[NRNH99] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis.* Springer-Verlag, Berlin, Heidelberg, Jan 1999.

[Plo04] Gordon D. Plotkin. A structural approach to operational semantics. *J. Log. Algebr. Program.*, 60-61:17–139, 2004.

[Rey98] John C. Reynolds. *Theories of Programming Languages.* Cambridge University Press, New York, 1998.