RWTH Aachen University
Chair for Software Modeling and Verification

Bachelor Thesis

Graph-Based Symbolic Execution for Pointer Programs with Data

Hanna Franzen

4th July 2016

First Supervisor: Thomas Noll
Second Supervisor: Joost-Pieter Katoen

Advisor: Christoph Matheja

Eidesstattliche Versicherung

______________________________ ______________________________

Name, Vorname Matrikelnummer (freiwillige Angabe)

Ich versichere hiermit an Eides Statt, dass ich die vorliegende Arbeit/Bachelorarbeit/ Masterarbeit* mit dem Titel

____

____

____

selbständig und ohne unzulässige fremde Hilfe erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt. Für den Fall, dass die Arbeit zusätzlich auf einem Datenträger eingereicht wird, erkläre ich, dass die schriftliche und die elektronische Form vollständig übereinstimmen. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

______________________________ ______________________________

Ort, Datum Unterschrift

*Nichtzutreffendes bitte streichen

Belehrung:

§ 156 StGB: Falsche Versicherung an Eides Statt

Wer vor einer zur Abnahme einer Versicherung an Eides Statt zuständigen Behörde eine solche Versicherung falsch abgibt oder unter Berufung auf eine solche Versicherung falsch aussagt, wird mit Freiheitsstrafe bis zu drei Jahren oder mit Geldstrafe bestraft.

§ 161 StGB: Fahrlässiger Falscheid; fahrlässige falsche Versicherung an Eides Statt

(1) Wenn eine der in den §§ 154 bis 156 bezeichneten Handlungen aus Fahrlässigkeit begangen worden ist, so tritt Freiheitsstrafe bis zu einem Jahr oder Geldstrafe ein.

(2) Straflosigkeit tritt ein, wenn der Täter die falsche Angabe rechtzeitig berichtigt. Die Vorschriften des § 158 Abs. 2 und 3 gelten entsprechend.

Die vorstehende Belehrung habe ich zur Kenntnis genommen:

______________________________ ______________________________

Ort, Datum Unterschrift

Contents

1 Introduction **1**

2 Programming Language **4**
2.1 Syntax 5
2.2 Heap Representation 5
2.3 Data-Aware Heap Configurations 7
2.4 Semantics 9

3 Concretisation and Abstraction **22**
3.1 Basic Principles 23
3.2 Data-Aware Concretisation and Abstraction 26
3.3 Correctness 32
3.4 Loss-free Abstraction 34
3.5 Abstraction with Common Conditions 36

4 Conclusion **40**

A Basic Definitions **42**

1 Introduction

In today's world, software is omnipresent. It enables cars to drive, assists in performing surgeries and enables machines to autonomously explore and analyse the surface of other planets. Software can have many flaws that must be prevented during the development. While a malfunction in a beverage dispenser due to software failure might be harmless, it might be fatal and lead to additional costs and serious danger in safety-critical applications such as a rocket launch. To prevent critical errors, software is tested and when it is essential that specific properties are fulfilled, software should be formally verified. In verification it can formally be proven for the software's code that it possesses certain properties of interest.

Model checking is one approach of verification which based on an exhaustive exploration of the state space of a program. Such a state space can be obtained by symbolic execution. During a symbolic execution of a program it is not executed with concrete input values but with abstract values. Whenever the execution of a statement in the program depends on the input value, all possible cases are explored. In contrast to testing, all possible outcomes can be determined.

In a lot of modern programming languages pointers are supported and used frequently in practice. Since pointers are powerful constructs, numerous common software errors are associated with them, for example null pointer dereferencing. Allowing the usage of pointers enables a programming language to express dynamic data structures like linked lists or tree structures.

These structures can be of an arbitrary size that can change during the execution of the program. Therefore the state space of the program in a symbolic execution can be infinite. For such, Heinen et al. suggest in [8] generalising all heap parts that are currently not referenced by the program, i.e. abstracting these parts. If the abstraction mechanism is chosen accordingly, this yields a finite state space.

The ansatz in this thesis is based on the one presented by Heinen et al. in [8]. There for every execution step of a program in a symbolic execution a hypergraph representing the program's current heap is computed. It represents objects in the heap by the graph's vertices while pointers are represented by edges such that they are attached to the objects they refer to.

An illustration of such a heap representation of a singly linked list is visualised as a hypergraph in Figure 1. The list has three nodes representing objects, each referencing the next element in the list if it exists. The first list element is referenced by a pointer named *head*, the last element with a pointer named *tail*.

As an example for abstraction, consider the abstract edge labelled L in the hypergraph in Figure 2a) to represent a singly linked list of arbitrary length. The abstracted hypergraph therefore represents the hypergraphs in Figure 2b) and 2c)

Figure 1: Example of a hypergraph representing a program heap.

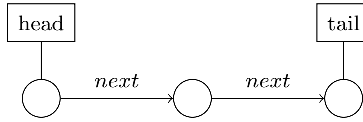
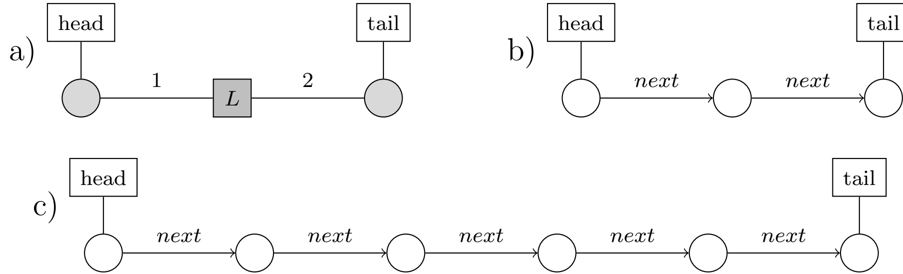


Figure 2: Example of an abstracted hypergraph.



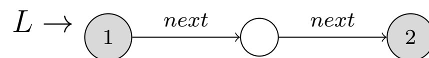
of different length as well as lists of any other possible length.

When during the execution a part of the heap that is abstracted is needed it can be concretised, i.e. the abstracted part or a chunk of it is replaced with concrete heap parts. As the abstraction has to throw away some information for the abstracted parts, like in Example 2 how many list elements there were in the original heap, there need to be rules that determine what possibilities there are to obtain the needed concrete part. Then all different possibilities are further analysed in the symbolic execution.

Heinen et al. define rules for determining these possibilities using hyperedge replacement grammars. These grammars work like context free string grammars except that edges labelled with nonterminal symbols are replaced by hypergraphs instead of replacing nonterminals by strings.

For example, the abstracted hypergraph in Figure 2a) can be concretised into the graph in Figure 2b) if the underlying grammar contains a rule that specifies that L can be replaced by a corresponding list. Such a rule is shown in Figure 3. The grey nodes labelled with 1 and 2 are placeholders for the external nodes of a hyperedge labelled with L. As hyperedges are defined by specifying a sequence of external nodes they are attached to, in Figure 2a) numbers on the edge represent the ordinal number of the attachment. Matching the rule to the abstracted graph yields the graph in Figure 2b).

Figure 3: Example of a hyperedge replacement rule.



As mentioned above, abstracting heaps leads to loss of certain information like the length of a list. Other information of interest that is lost are for example data values stored in the data structure. These are ignored in the symbolic execution of Heinen et al. so far. Therefore this thesis extends the programming language with pointers presented in [8] to handle the data of objects. Correspondingly it introduces an extension of the representation of heaps with the ability to store information about object data by storing sets of valid conditions about the object's data. The storing mechanism is a mapping of the edges of the heap representation to such sets. Information about an object's value is always associated with edges that are attached to the object such that only local conditions are stored.

For example, if all list elements of a list as shown in Figure 2 stored an integer value all these values are lost in the ansatz in [8]. This thesis aims at that exact problem. While keeping all values of the abstracted pointer structure can result in an arbitrarily large value set to store, it is possible to keep some information about the values. This is done by storing sets of information about the edges in the original graph in the abstracted heap parts. With an extension for heap graphs of this form no change is necessary in the hyperedge replacement grammars used for abstraction and concretisation. So in other words, relationships between the data of objects are inferred automatically during the symbolic execution. This opens a variety of possibilities, for example to prove that a list is sorted.

Which set of information about data is worth storing during an abstraction highly depends on the application of the program, hence two different approaches at abstracting heap parts with data are presented. The first approach does not alter any of the information collected during the previous execution while the second approach stores a set of weaker conditions which are implied by all conditions about the original edges.

Related Work. There are some similar approaches to verify pointer programs. Several do not take into account data, like [3] or [9] which only analyse the shape of structures.

A verification mechanism that includes data is presented by Chin et al. in [4]. In this work user-defined predicates for local references enable the capturing of size and bag properties as invariants additionally to the verification of shape properties. In this context, bag properties specify invariants for reachability and size properties specify invariants for data structures, for example the orderedness of a sorted list. As in [3], the approach is based on the use of separation logic.

Another approach by Ferrara et al. in [6] uses heap structures based on directed graphs for the analysis of pointer programs. Its focus is on automatically infering invariants of the heap structures from the information in the program without the help of manual annotations. Each edge has an identifier that contains an

abstracted value state for the values of the edge's source and target node. The approach includes identifiers for abstract edges. On these heap structures it can be determined if the conditions on heap edges can be fulfilled and the current pointer structure can exist in a certain value domain.

An approach of building a framework for verifying programs with pointer structures where the verification depends on the order of the data of objects is presented by Abdulla et al. in [1]. Like [9] it is based on forest automata as introduced by Habermehl et al. in [7]. Such forest automata consist of tree automata that can communicate by referring to each other's roots from their leafs and might be nested. The approach of Abdulla et al. extends forest automata to express relationships between data elements. Therefore it introduces two constraint types for data, local and global ones, which are included as parts of forest automata. Local data constraints express relations between the data of neighbouring nodes while global ones express relationships between data of nodes in other parts of the heap. In contrast to the approach in this thesis as well as the one in [6], data abstraction has to be encoded in the automata and is not inferred automatically. Abstract regular tree model checking as defined by Bouajjani et al. in [2] is adapted and applied to the extended automata (as done in [7] without considering data).

Outline. This thesis first introduces a programming language with pointers and integer values stored in objects in chapter 2. The graph-based heap representation is explained in chapter 2.2 and 2.3. This representation is used in the definition of the language's semantics in chapter 2.4. In chapter 3 the concepts of abstraction and concretisation are explained and a base for an extension to deal with data is defined in chapter 3.2. This base is used to define two abstraction approaches in chapter 3.4 and 3.5. A conclusion is given in chapter 4.

2 Programming Language

In this thesis, a simple programming language with pointers and data values stored in objects is studied. Note that the context-free grammar for the syntax of pointer programs as well as the semantics of the language is based on the ones defined in [8]. They are extended to handle data. After introducing the programming language, a way to represent a program's heap by heap configurations is introduced, which are special hypergraphs. To collect and store information, heap configurations as well as the programming language's semantics are extended to take data into account.

2.1 Syntax

The syntax of the programming language is given by the grammar in Figure 4. A pointer program consists of statements $S \in Stms$ that can be arranged in blocks Blk. It supports pointer expressions Ptr that can reference program objects via selectors $s \in Sel$ denoted as $x.s \in Ptr$. Objects can have data fields referenced by $x.z \in Vls$. Assignments to pointer variables and values are contained in the set of $Stms$, additionally a new object can be created by a function **new**. Pointers that reference no object are valid and denoted with **null**. Conditions Cnd can either be conditions for program variables $PCnd$ or values $VCnd$ or conjunctions and disjunctions of such. Statements contain **if-else** conditions and **while** loops that use such conditions Cnd. The empty statement **noop** is also a valid statement in the language. Only assignments to values of program variables are possible such that there are no global value variables. Note that the syntax for most statements and for blocks and pointers is defined analogous to the syntax defined in [8].

Figure 4: Syntax of Pointer Language

$$
\begin{array}{ll}
S ::= & x := P \mid x.s := P \mid \mathbf{new}(x) \mid x.z := Z \mid \mathbf{while}(C)\ S \mid \\
 & \mathbf{if}\ (C)\ S\ \mathbf{else}\ S \mid \mathbf{noop} \mid B \in Stms \\
B ::= & \{S\ (;S)^*\} \in Blk \\
Z ::= & \zeta \in \mathbb{Z} \mid x.z \mid (Z + Z) \mid (-Z) \mid (Z \cdot Z) \in Vls \\
C_Z ::= & (Z = Z) \mid (Z \neq Z) \mid (Z < Z) \mid (Z > Z) \mid (Z \leq Z) \mid (Z \geq Z) \\
 & \in VCnd \\
C_P ::= & (P = P) \mid (P \neq P) \in PCnd \\
C ::= & C_P \mid C_Z \mid (C \wedge C) \mid (C \vee C) \in Cnd \\
P ::= & \mathbf{null} \mid x \mid x.s \in Ptr
\end{array}
$$

Example 1. A program written in the language defined above is given in Figure 5. It implements a subprocedure of an insertion sort algorithm. The shown algorithm inserts a list element into a doubly linked list that is already sorted in ascending order. It uses an iterator that is set to the next list element until it finds the right spot to insert the new element and uses assignments to selectors of list elements for the insertion. The example is inspired by the implementation of inserting into linked lists in [5] (p. 238).

2.2 Heap Representation

As mentioned before, this thesis is an extension of an existing symbolic execution and abstraction approach from [8]. The most relevant definitions are presented in

Figure 5: Implementation of insertion into sorted doubly linked lists.

```
insert(list, element){

  new(iterator);
  iterator = list.head;

  while(iterator.data ≤ element.data){
    iterator := iterator.next;
  }

  element.next := iterator;
  element.prev := iterator.prev;
  iterator.prev := element;
}
```

this chapter in order to summarise said basis.

Independent of a language in which a program is written, for pointer programs the program's heap can be modelled using a hypergraph.

Definition 1. Let Σ be a finite alphabet with $rk : \Sigma \to \mathbb{N}_0$ assigning a rank $rk(X)$ to each symbol X in Σ. A **hypergraph** H over Σ is defined as a tuple $H = (V, E, att, lab, ext)$ with V being a finite set of vertices, E a finite set of hyperedges, $att : E \to V^*$ a function that maps each hyperedge to a sequence of attached vertices, $lab : E \to \Sigma$ a labelling function and $ext \in V^*$ a sequence of pairwise distinct external vertices. HG_Σ is the set of all hypergraphs over Σ.

For each edge $e \in E$ of rank 2 with $att(e) = uv$, $e.source$ and $e.target$ denote the source and target of the edge ($e.source = u$ and $e.target = v$). For edges e that connect a sequence of vertices of arbitrary length $\neq 2$ the expression $e.target$ is not used in this thesis while $e.source$ is used for the first node in the sequence. If e is clear from the context, $source$ denotes $e.source$ and $target$ denotes $e.target$.

Hypergraphs that represent a pointer program's heap need a few restrictions to model pointer structures.

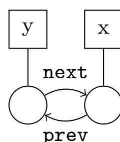
A **heap configuration** is a hypergraph $H = (V, E, att, lab, ext) \in HG_\Sigma$ with such restrictions. In detail, vertices of the hypergraph represent objects of the program, program variables are represented by edges of rank 1 with the labelling lab of the edge being the name of the variable. Selectors are represented by edges of rank 2 with the selector's name as labelling. Therefore Σ is the the union of all program variables x and selectors s of the program. Every variable can only appear once in the graph, it is unique. Every selector can only appear once for each

vertex, i.e. for every vertex v there can only be one edge $e \in E$ with $att(e)(1) = v$ labelled with the selector. The set of all concrete heap configurations is denoted as HC_{Σ}. Additionally we define the set N of nonterminal symbols which will be used in chapter 3. The set of all heap configurations is HC_{Σ_N} with $\Sigma_N := \Sigma \biguplus N$.

A formal definition of heap configurations is given in the appendix on page 42.

Example 2. An example of a heap configuration is given in Figure 6. It shows two vertices that are connected by two edges labelled with selectors. The arrows visualise the order of the nodes in the sequence of vertices attached to the edges. The square boxes contain the labelling of the edges of rank one which represent program variables.

Figure 6: Example of a heap configuration.



2.3 Data-Aware Heap Configurations

Heap configurations as defined above represent only the structure of a heap while ignoring its contents. This chapter enriches heap configurations by information about the contained data such that they can handle values Vls as introduced in the syntax of the programming language.

Definition 2. Let H be a heap configuration. A function α is called an **information function** of H if its signature is $\alpha : E \to \mathcal{P}(C_{data})$, i.e. it maps heap configuration edges to sets of information about the heap's data in C_{data} which is defined as $C_{data} = C_{\alpha} \cup C_{\Sigma}$, $C_{\Sigma} = (Sel \times C_{\alpha})$.

C_{α} contains certain conditions between values, program variables and $e.source$ and $e.target$ for edges e. Let $x, y \in Ptr, e \in E$ and z_1, z_2 possible value fields of objects. Let $z' \in \mathbb{Z}, \sim \in \{=, \neq, <, >, \leq, \geq\}$. C_{α} consists of expressions of the form $(x.z_1 \sim z'), (x.z_1 \sim y.z_2), (e.source.z_1 \sim e.target.z_2)$, $(x = y), (e.source = x), (e.target = y)$.

For information about the data of a concrete heap configuration $H \in HC_{\Sigma}$ only conditions from C_{α} are stored in the information function by the concrete semantics of the pointer language in the following chapter.

For heap configurations $H \in HC_{\Sigma_N}$ there are special cases for edges that are labelled with a nonterminal symbol. For the mapping of such C_{data} contains the set C_{Σ} which consists of pairs of selectors and relative conditions (conditions

containing *e.source*, *e.target*) from C_α. This mapping is used for abstraction in chapter 3.

Additionally, a function $\alpha \in A$ is assumed to be symmetric for selector edges. If $e, e' \in E$ with $lab(e), lab(e') \in Sel$ and $att(e) = vu$ and $att(e') = uv$ for $v, u \in V$ then $\alpha(e') = \{rev[c, e'] \mid c \in \alpha(e) \land c \in VCnd\}$. The *rev* function reverses a value condition such that it corresponds to the edge e'. A formal definition of the function *rev* is given in the appendix on page 43.

The set of all functions $\alpha : E \to \mathcal{P}(C_{data})$ is denoted by A.

Example 3. An example for the symmetry of selector edges for functions in A is the following. If e is mapped to a condition of the form $(e.source.z < e.target.z)$ and $att(e) = vu$ then there exists a corresponding condition $rev[(e.source.z < e.target.z)] = (e'.source.z > e'.target.z)$ in the mapping of the edge e' with $att(e') = uv$.

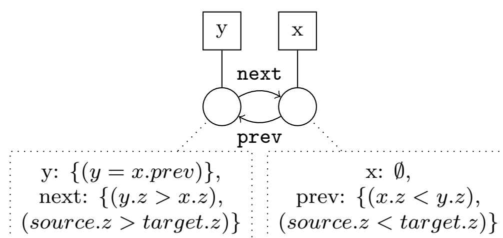
Note that α stores all available information about data including initial assignments. A heap configuration $H \in HC_{\Sigma_N}$ does not include data yet and the definition of a function in $\alpha \in A$ is incomplete if it does not refer to a heap configuration. To include data, data-aware heap configurations are introduced.

Definition 3. A **data-aware heap configuration** (DHC) consists of a heap configuration H and a suitable information function $\alpha \in A$.

Formally, $DHC_{\Sigma_N} := (HC_{\Sigma_N} \times A)$. Usually this is denoted as $H^\alpha \in DHC_{\Sigma_N}$ instead of (H, α).

Example 4. Figure 7 shows a small example of a DHC H^α. Note that the symmetry for selector edges is satisfied. For a compact visualisation the dotted boxes show the mapping in α for all outgoing edges of the connected vertex. As the edge is clear in this context, expressions *next.source* and *next.target* (*prev.source* and *prev.target*) are denoted as *source* and *target*. This format is used for examples throughout this thesis.

Figure 7: Example of a DHC.



2.4 Semantics

In this chapter a graph-based semantics for the previously introduced syntax of the programming language is provided. Like the syntax it is based on the semantics of the language defined in [8] and extended for value expressions. It is therefore based on data-aware heap configurations.

A table of all commonly used variables in this chapter can be found in the appendix on page 42.

When talking about the semantics of the programming language and its values a way to modify the mapping of functions $\alpha \in A$ has to be introduced. Assume a program's heap is represented by a data-aware heap configuration $H^\alpha \in DHC_{\Sigma_N}$. The expression $\alpha[e \mapsto \{\cdot\}]$ denotes a function in A that is identical to α apart from the mapping of the edge $e \in E_H$ with $lab(e) \notin N$ which is mapped to the given set $\{\cdot\} \subseteq C_\alpha$ instead.

For simplicity it is assumed that every function that changes α retains the symmetry property for selector edges implicitly. Thus if the mapping α of an edge e with $att(e) = vu$ to a set of conditions is changed by a function it is assumed to be changed for an edge $e' \in E$ with $att(e') = uv$ conform to the principles described in chapter 2.3 in order to retain the symmetry, even if the edge e did not exist before.

In general the following pointer manipulations can occur on heap configurations H. For $x \in Ptr, v \in V_H$, $x \hookrightarrow_H v$ denotes that x points to the object in the program that is represented by the vertex v in H, i.e the edge labelled with x in H is attached to v. If there is no such edge in H this is denoted by $x \hookrightarrow_H$ **null**. For $x.s$ with $s \in Sel, u \in V_H$, $u \overset{s}{\hookrightarrow}_H v$ denotes that the edge e labelled with s with source u has the target vertex v, i.e. $att(e) = uv$. Again, if there is no such edge this is denoted by $u \overset{s}{\hookrightarrow}_H$ **null**. $H[+v]$ denotes the extension of V_H by a new vertex v. The semantics of the components of the pointer language is denoted by $[\![\cdot]\!]_{H^\alpha}$ for a given DHC H^α representing the current program's heap and is defined in this chapter.

The *semantics of pointer expressions* in this thesis is defined analogously to the semantics of pointer expressions in [8] as a mapping $[\![\cdot]\!]_{H^\alpha} : Ptr \to V \cup \{\mathbf{null}, \bot\}$ for a heap configuration $H^\alpha = ((V, E, att, lab, ext), \alpha) \in DHC_{\Sigma_N}$.

The semantics of **null** is **null** and the semantics of program variables x is either the vertex to which the edge labelled with x in H is attached or **null** if there is no such edge. For pointers $x.s$ with $s \in Sel$, in case x is not **null** the semantics evaluates to the vertex that is the target of an edge labelled s from the vertex that x refers to or **null** if there is no such edge. If x is **null** the semantics for $x.s$ is $\bot$, which corresponds to a run-time error. A formal definition is provided below.

Definition 4. Let $v \in V$, $x, x.s \in Ptr$ with $s \in Sel$.

The **semantics of pointer expressions** is defined as $[\![\cdot]\!]_{H^\alpha} : Ptr \to V \cup \{\mathbf{null}, \bot\}$.

$$[\![\mathbf{null}]\!]_{H^\alpha} := \mathbf{null}$$

$$[\![x]\!]_{H^\alpha} := \begin{cases} v & , x \hookrightarrow_H v \\ \mathbf{null} & , x \hookrightarrow_H \mathbf{null} \end{cases}$$

$$[\![x.s]\!]_{H^\alpha} := \begin{cases} v & , [\![x]\!]_{H^\alpha} \neq \mathbf{null} \wedge [\![x]\!]_{H^\alpha} \overset{s}{\hookrightarrow}_H v \\ \mathbf{null} & , [\![x]\!]_{H^\alpha} \neq \mathbf{null} \wedge [\![x]\!]_{H^\alpha} \overset{s}{\hookrightarrow}_H \mathbf{null} \\ \bot & , [\![x]\!]_{H^\alpha} = \mathbf{null} \end{cases}$$

The *semantics of value expressions* Vls has the signature $[\![\cdot]\!]_{H^\alpha} : Vls \to \mathbb{Z} \cup \bot$. Intuitively, it assigns an integer value to value expressions in the pointer language.

The semantics for integer values ζ is defined as ζ. Values of objects referenced by $x.z$ are defined as the value $z' \in \mathbb{Z}$ that is stored in α as a condition $(x.z = z')$ in the mapping of the edge labelled with x. Note that edges labelled with program variables are unique in the heap configuration so that $[\![x.z]\!]_{H^\alpha}$ is defined unmistakable for these cases. If there is no such condition in the mapping of the edge, the program variable x is actually a selector s of a variable y or if the variable x is non existent, the semantics of $x.z$ is an invalid expression and therefore evaluates to $\bot$. For the arithmetic operator $+$, the semantics of $(z_1 + z_2)$ for the values z_1 and z_2 is defined as the sum of the semantics of z_1 and z_2 or $\bot$ if the semantics of z_1 or z_2 is invalid and therefore not an integer value. The same applies for the semantics of $(z_1 \cdot z_2)$ with the change that $\cdot$ is the operator for the multiplication of integer values. The semantics of $-z_1$ is either the negative value of the integer value that is the semantics of z_1 or $\bot$ if the semantics of z_1 is $\bot$. A formal definition follows.

Definition 5. Let $x, y \in Ptr, z', \zeta \in \mathbb{Z}, s \in Sel, x.z, z_1, z_2 \in Vls$. $+, -, \cdot$ are defined as the usual arithmetic operators on $\mathbb{Z}$.

The **semantics of value expressions** is defined as $[\![\cdot]\!]_{H^\alpha} : Vls \to \mathbb{Z} \cup \{\bot\}$.

$$[\![\zeta]\!]_{H^\alpha} := \zeta$$

$$[\![x.z]\!]_{H^\alpha} := \begin{cases} \bot & , [\![x]\!]_{H^\alpha} = \mathbf{null} \vee [\![x]\!]_{H^\alpha} = \bot \vee x = y.s \vee \\ & (\exists e \in E : lab(e) = x \wedge \neg\exists (x.z = \zeta) \in \alpha(e)) \\ z' & , \exists e \in E : lab(e) = x \wedge (x.z = z') \in \alpha(e) \end{cases}$$

$$[\![(z_1 + z_2)]\!]_{H^\alpha} := \begin{cases} \bot & , [\![z_1]\!]_{H^\alpha} = \bot \vee [\![z_2]\!]_{H^\alpha} = \bot \\ z' & , [\![z_1]\!]_{H^\alpha} + [\![z_2]\!]_{H^\alpha} = z' \end{cases}$$

$$[\![(-z_1)]\!]_{H^\alpha} := \begin{cases} \bot & , [\![z_1]\!]_{H^\alpha} = \bot \\ z' & , -[\![z_1]\!]_{H^\alpha} = z' \end{cases}$$

$$[\![(z_1 \cdot z_2)]\!]_{H^\alpha} := \begin{cases} \bot & , [\![z_1]\!]_{H^\alpha} = \bot \vee [\![z_2]\!]_{H^\alpha} = \bot \\ z' & , [\![z_1]\!]_{H^\alpha} \cdot [\![z_2]\!]_{H^\alpha} = z' \end{cases}$$

For a heap configuration $H^\alpha \in HC_{\Sigma_N}$ the signature of the *semantics of conditions* is $[\![\cdot]\!]_{H^\alpha} : Cnd \to \mathbb{B} \cup \{\bot\}$, where $\mathbb{B} = \{\text{true}, \text{false}\}$.

It considers separated cases for pointer conditions $PCnd$ and value conditions $VCnd$. The semantics for pointer conditions $(x \sim_P y)$ with $\sim_P \in \{=, \neq\}$ evaluates to true if the relation $\sim_P$ is true for the semantics of the variables x and y. The semantics is false if the other relation is true for the semantics of the variables, respectively. E.g. if a condition $(x = y)$ is analysed and $(x \neq y)$ is true, the semantics of the condition is false. If the semantics of one of the variables is invalid, the whole pointer condition evaluates to $\bot$.

The semantics of value conditions is defined similarly. For a condition $(z_1 \sim z_2)$ it evaluates to true if the relation $\sim$ is true for the semantics of the values z_1 and z_2. It evaluates to false if the semantics of the negation of the condition is true. If one of the values in the condition is invalid which can only happen if it is the value of an object, the whole condition evaluates to $\bot$.

For conditions Cnd that are pointer conditions $PCnd$ or value conditions $VCnd$, the semantics is defined as described above. For all other conditions in Cnd the semantics evaluate to true if they are the conjunction of two conditions whose semantics are true or the disjunction of two conditions for which one the semantics evaluates to true. The semantics of the conjunction of two conditions evaluates to false if one of the semantics of the conditions evaluates to false and the semantics of the disjunction of two conditions evaluates to false if the semantics of both of the conditions evaluates to false. The semantics of conditions $(c_1 \wedge c_2)$ or $(c_1 \vee c_2)$ with conditions c_1 and c_2 are invalid and therefore evaluate to $\bot$ if the semantics of c_1 or c_2 evaluates to $\bot$. A formal definition of the semantics of conditions is given in the following.

Definition 6. Let $c \in Cnd, c_Z \in VCnd, c_P \in PCnd, x, y \in Ptr, x.z, y.z, z_1, z_2 \in Vls, c_1, c_2 \in Cnd, \sim_P \in \{=, \neq\}, \sim \in \{=, \neq <, \leq, >, \geq\}$.

The **semantics of conditions** is defined as $[\![\cdot]\!]_{H^\alpha} : Cnd \to \mathbb{B} \cup \{\bot\}$.

$$
[\![c_P]\!]_{H^\alpha} := \begin{cases} \bot & , c_P = (x \sim_P y) \wedge ([\![x]\!]_{H^\alpha} = \bot \vee [\![y]\!]_{H^\alpha} = \bot) \\ \text{true} & , (c_P = (x = y) \wedge [\![x]\!]_{H^\alpha} = [\![y]\!]_{H^\alpha}) \vee \\ & \ (c_P = (x \neq y) \wedge [\![x]\!]_{H^\alpha} \neq [\![y]\!]_{H^\alpha}) \\ \text{false} & , (c_P = (x = y) \wedge [\![x]\!]_{H^\alpha} \neq [\![y]\!]_{H^\alpha}) \vee \\ & \ (c_P = (x \neq y) \wedge [\![x]\!]_{H^\alpha} = [\![y]\!]_{H^\alpha}) \end{cases}
$$

$$
[\![c_Z]\!]_{H^\alpha} := \begin{cases} \bot & , (c_Z = (z_1 \sim y.z) \vee c_Z = (x.z \sim z_2)) \wedge \\ & \ ([\![x.z]\!]_{H^\alpha} = \bot \vee [\![y.z]\!]_{H^\alpha} = \bot) \\ \text{true} & , c_Z = (z_1 \sim z_2) \wedge [\![z_1]\!]_{H^\alpha} \sim [\![z_2]\!]_{H^\alpha} \\ \text{false} & , \text{otherwise.} \end{cases}
$$

$$
[\![c]\!]_{H^\alpha} := \begin{cases} [\![c]\!]_{H^\alpha} & , c \in PCnd \cup VCnd \\ \bot & , (c = (c_1 \wedge c_2) \vee c = (c_1 \vee c_2)) \wedge \\ & \ ([\![c_1]\!]_{H^\alpha} = \bot \vee [\![c_2]\!]_{H^\alpha} = \bot) \\ \text{true} & , (c = (c_1 \wedge c_2) \wedge ([\![c_1]\!]_{H^\alpha} = \text{true} \wedge [\![c_2]\!]_{H^\alpha} = \text{true})) \vee \\ & \ (c = (c_1 \vee c_2) \wedge ([\![c_1]\!]_{H^\alpha} = \text{true} \vee [\![c_2]\!]_{H^\alpha} = \text{true})) \\ \text{false} & , (c = (c_1 \wedge c_2) \wedge ([\![c_1]\!]_{H^\alpha} = \text{false} \vee [\![c_2]\!]_{H^\alpha} = \text{false})) \vee \\ & \ (c = (c_1 \vee c_2) \wedge ([\![c_1]\!]_{H^\alpha} = \text{false} \wedge [\![c_2]\!]_{H^\alpha} = \text{false})) \end{cases}
$$

A formal definition of *neg* is given in the appendix on page 42.

Definition 7. The *semantics of statements* for $H^\alpha \in DHC_\Sigma$ is defined similarly to [8] as a transition system $(Cnf, \triangleright)$ with program configurations $Cnf = (Stms \cup \{\varepsilon\}) \times DHC_\Sigma$, where ε is the empty statement, and a transition relation $\triangleright \subseteq Cnf \times Cnf$.

Let $x \in Ptr, s \in Sel, x.z, z' \in Vls$ with $z' \in \mathbb{Z}$, $H^\alpha \in DHC_{\Sigma_N}$ the current DHC, $S, S_1, S_1', S_2 \in Stms, C \in Cnd$. The transition relation $\triangleright$ is determined by the rules in Figure 8 and will be further explained in the following. For simplicity a function *learn* is used for changes in the information function whose definition is provided later in the chapter.

While the operations done on the heap configuration are analogous to [8], this thesis also takes the data of program objects into account by extending the heap configuration by an information function as introduced in the previous chapter. Additionally there is a rule for the assignment to value fields of variables.

The first rule in the semantics is for the assignment to a variable x. The new value P must be valid in H as a premise. As a derivation the semantics

Figure 8: Semantics of statements.

$$\frac{[\![P]\!]_{H^\alpha} \neq \bot}{\langle x := P, H^\alpha \rangle \rhd \langle \varepsilon, H[x \hookrightarrow [\![P]\!]_{H^\alpha}]^{learn[\alpha, x, \{(x=P)\}]} \rangle}[\text{passign}]$$

$$\frac{[\![P]\!]_{H^\alpha} \neq \bot}{\langle x.s := P, H^\alpha \rangle \rhd \langle \varepsilon, H[[\![x]\!] \stackrel{s}{\hookrightarrow} [\![P]\!]_{H^\alpha}]^{learn[\alpha, x.s, \emptyset]} \rangle}[\text{sassign}]$$

$$\frac{}{\langle x.z := z', H^\alpha \rangle \rhd \langle \varepsilon, H^{learn[\alpha, x.z, \{(x.z=z')\}]} \rangle}[\text{zassign}]$$

$$\frac{}{\langle \mathbf{noop}, H^\alpha \rangle \rhd \langle \varepsilon, H^\alpha \rangle}[\text{noop}]$$

$$\frac{}{\langle \mathbf{new}(x), H^\alpha \rangle \rhd \langle \varepsilon, H^\alpha[+v][x \hookrightarrow_H v] \rangle}[\text{pnew}]$$

$$\frac{}{\langle \{S\}, H^\alpha \rangle \rhd \langle S, H^\alpha \rangle}[\text{blk}]$$

$$\frac{\langle S_1, H^\alpha \rangle \rhd \langle S_1', H'^{\alpha'} \rangle}{\langle S_1; S_2, H^\alpha \rangle \rhd \langle S_1'; S_2, H'^{\alpha'} \rangle}[\text{seq}]$$

$$\frac{[\![C]\!]_{H^\alpha} = \text{true}}{\langle \mathbf{while}(C)S, H^\alpha \rangle \rhd \langle \{S, \mathbf{while}(C)S\}, H^{learn[\alpha, \mathbf{null}, cz[C]]} \rangle}[\text{twhile}]$$

$$\frac{[\![C]\!]_{H^\alpha} = \text{false}}{\langle \mathbf{while}(C)S, H^\alpha \rangle \rhd \langle \varepsilon, H^{learn[\alpha, \mathbf{null}, cz[C]]} \rangle}[\text{fwhile}]$$

$$\frac{[\![C]\!]_{H^\alpha} = \text{true}}{\langle \mathbf{if}(C)S_1\mathbf{else}S_2, H^\alpha \rangle \rhd \langle S_1, H^{learn[\alpha, \mathbf{null}, cz[C]]} \rangle}[\text{tif}]$$

$$\frac{[\![C]\!]_{H^\alpha} = \text{false}}{\langle \mathbf{if}(C)S_1\mathbf{else}S_2, H^\alpha \rangle \rhd \langle S_2, H^{learn[\alpha, \mathbf{null}, cz[C]]} \rangle}[\text{fif}]$$

of the assignment of P to x with given DHC H^α yields an empty statement as the given one was executed and a DHC $H'^{\alpha'}$ in which x points to its new value in H'. The function α is thereby changed to a function α' such that for every former appearance of the variable name x is deleted from α, the new assignment is stored as a pointer condition and any relative information derivable with the newly introduced information is stored. All of this is done by the function *learn* which takes an input of the form $(A \times (Ptr \cup Vls) \times \mathcal{P}(C_\alpha))$. The first argument α

is the information function that is to be modified. If every former appearance of a pointer or value of an object is to be deleted it is passed in the second argument, in this case x. Thus for every mapping to a set of conditions in α every condition in which x appears is removed. Afterwards the function inserts all conditions in the condition set that is the third argument, in this case the new assignment represented as a pointer condition, into the remaining information function. It inserts the new condition to the edge that represents the variable x and derives further information from the current mapping. This further information includes side effects and redundant information like a condition $(e.source = x)$ to each outgoing edge e of the vertex that represents the object that x points to. This redundant information can be used to easily derive other relative conditions that are later used in abstraction. The *learn* function is formally defined later in this chapter.

Example 5. Consider a data-aware heap configuration H^α during the symbolic execution of a program written in the introduced programming language. If an edge e in E that represents a selector is mapped to the set of conditions $\{(e.source = x), (e.target = y), (x.z < y.z)\}$ in α, a new condition $(e.source.z < e.target.z)$ can be learned and added to the former set during a call of the *learn* function. So if x is assigned to another object the condition $(x.z < y.z)$ is deleted from the mapping of e but the information $(e.source.z < e.target.z)$ can remain.

The semantics of the assignment to a selector $x.s$ is handled by changing the target of the edge e that represents $x.s$ in the heap to the object (vertex) v the newly assigned pointer refers to, i.e. $att(e)(2)$ is set to v. Similarly to the assignment to program variables, all appearances of the selector $x.s$ are deleted in the information function by the *learn* function. Note that $x.s$ only explicitly occurs in the information function if there was an assignment of the form $y = x.s$ to a program variable y in the foregone execution. Again, any relative conditions that are derivable from the currently present conditions in the mapping are stored. In contrast to the assignment to program variables no pointer condition is inserted.

Example 6. During the execution of an assignment to $x.s$, the call of the *learn* function can for example add conditions of the form $(e.target = y)$ to the outgoing edge e of the vertex v that x refers to. For this e must be labelled with the selector s and there must be an edge e' with $lab(e') = y$ and $att(e') = att(e)(2)$, i.e. the variable y must reference the target vertex of e.

In the assignment to values of program variables $x.z$ there is no change in the heap itself as heap configurations do not handle data. Instead, the value is written to the information function. As seen in the assignments to program variables all former appearances of $x.z$ are deleted from the stored information in α by using

it as the second argument of the *learn* function. The new assignment is given to the *learn* function, in this case as a value equation condition, to be inserted into the mapping of the edge with labelling x, i.e. the edge that represents x in H.

Any changes in α are handled by a *learn* function. As mentioned above, *learn* first removes every appearance of a given pointer or value of an object from the given information function, secondly inserts a set of given constraints and subsequently derives additional information. It is important to notice that any changes in the information function are applied after the change in the heap as implied by the order in the inference rules.

Definition 8. The function *learn* is defined as
$learn : (A \times (Ptr \cup Vls) \times \mathcal{P}(C_\alpha)) \rightarrow A.$
$learn[\alpha, m, C] := closure[insert[delete[\alpha, m], C]].$

The *delete* function removes outdated information, the *insert* function inserts the conditions in the given set and the *closure* function derives additional relative information. The functions are formally defined in the following.

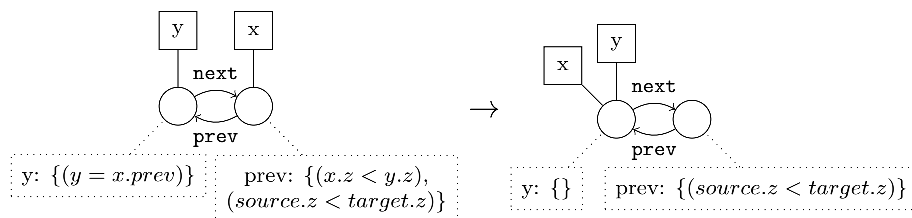
During deletion and insertion, side effects must be considered that are common when working with pointers. In detail, for every assignment to a program variable's value all values of pointers referencing the same object must be changed. If a variable x is assigned the value of another object, all outgoing edges of the vertex representing the object must remove their outdated information by using *delete* and receive the new information using the conditions introduced by *insert*. Similarly, changing the assignment of selectors must be considered in all affected edges.

First, whenever a pointer or value is changed in the heap there potentially is information about them in α that might be violated by the change. So on every assignment outdated information is removed by the function *delete*. It has to cover the mentioned side effects that occur when manipulating pointers and the data of objects. Changing information regarding an object that is referenced by several pointers must be changed in all of said pointers. *delete* removes any occurrence of the given name $m \in (Ptr \cup Vls)$ along with any resulting outdated information including side effects and outdated *target* and *source* references. Note that *delete* removes any non-**null** pointer including selectors s.

In detail, delete identifies four basic cases and as such defines what information is deleted depending on if the given variable m that is to be deleted is a program variable, a selector, a value of an object, or an integer value or **null**. Both can not be deleted from the given information function α, therefore α is returned unchanged. This is used in the last four rules in Figure 8 in which **null** is given to the *learn* function as the second argument as no deletion is intended.

In case the given argument is a program variable x, *delete* returns a function $\beta \in A$ that is equivalent to α apart from the removal of all occurrences of conditions that use x. In particular, all conditions that explicitly contain x or selectors or values of x are removed.

Example 7. Consider the following illustration of a DHC in which only the mapping of the edges representing the program variable y and the pointer $x.prev$ in the information function are shown and only the deletion of explicit occurrences of x is shown during the execution of a statement $x := y$. The function *delete* is called with $[\alpha, x]$ as arguments with α being the current information function.

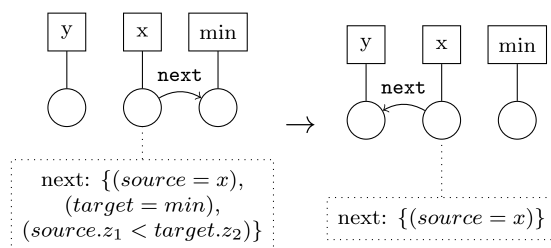


The conditions $(y = x.prev)$ and $(x.z < y.z)$ are removed as they contain x. Note that while the condition $(x.z < y.z)$ is removed, the equivalent relative condition $(prev.source.z < prev.target.z)$ in the mapping of the selector *prev* is kept.

Similar to the example above, the following examples in this chapter usually show only fragments of DHCs that help to illustrate a certain aspect. This mainly applies to the selective showing of parts of the information function to stress the changes happening during a call of the *learn* function.

If the second argument given to *delete* is the selector s of a pointer variable x the function returns γ. This function has the same mapping as α apart from deletions of explicit occurrences of $x.s$ similar to β. Additionally, it removes conditions of the form $(e.target = y)$ with $y \in Ptr$ and $(e.source.z_1 \sim e.target.z_2)$ for edges e that represent selectors s and whose source represents the object that x currently points to.

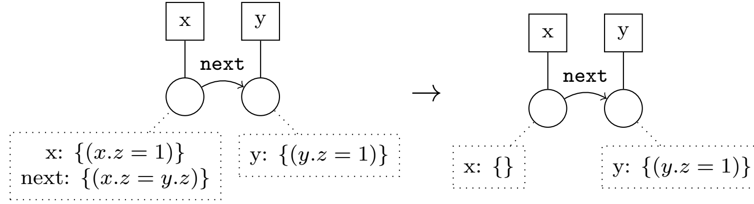
Example 8. Consider the following change in the information function of a selector *next* during the execution of a statement $x.next = y$. The call of *delete* gives the arguments $[\alpha, x.next]$ with α being the current information function.



The conditions $(next.target = min)$, $(next.source.z_1 < next.target.z_2)$ are deleted as they are outdated after changing the target of the edge. As only the target is changed, only occurrences of $x.next$ are removed while the condition $(next.source = x)$ is kept as x is not altered.

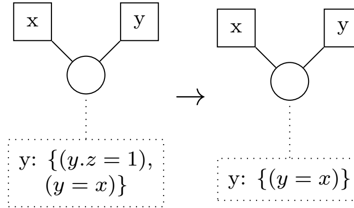
For a second argument m representing the values z of a pointer variable x, *delete* returns a function $\delta \in A$ that again is like α apart from the removal of any condition in the mapping that explicitly contains $x.z$.

Example 9. Such a change can be observed during the following example where the execution of a statement $x.z = 0$ is illustrated by a change of a DHC H^α in which the call of *delete* is executed with the arguments $[\alpha, x.z]$.



Side effects are also modelled by removing all occurrences of the value z of pointers that point to the same object as x in δ as shown in the following example.

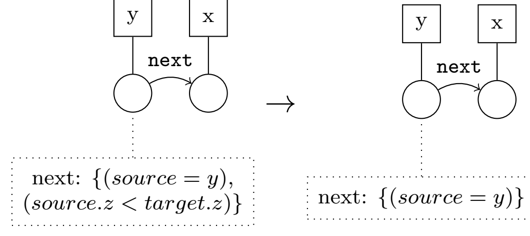
Example 10. The deletion of value assignments due to side effects in the call of $delete[\alpha, x.z]$ is illustrated during the execution of a statement $x.z = 0$.



As the pointer variables x and y in this example point to the same object, a change in the value z of x must be transferred to the value z of y. Therefore the condition $(y.z = 1)$ is deleted from the mapping of y in α.

Lastly, δ does not contain conditions of the form $(e.source.z \sim e.target.z)$ if the edge e that is mapped to them has the vertex as a target or source that represents the object referenced by the pointer variable x.

Example 11. For example consider the following change in a DHC during the execution of a statement $y.z := \zeta$ with the call $delete[\alpha, y.z]$ and information function α.



As the source of the edge labelled with *next* is attached to an edge labelled with x, the condition $(next.source.z < next.target.z)$ is removed.

A formal definition of *delete* is given in the following.

Definition 9. Let $m \in (Ptr \cup Vls), y \in Ptr$ with $y \neq \mathbf{null}$, $s \in Sel$, $\zeta \in \mathbb{Z}$. Moreover, let $p \in Ptr \cup \{source, target\}$, $p.z_1, p.z_2 \in Vls$. $E_\Sigma = \{e \in E \mid lab(e) \notin N\}$.

The function *delete* is defined as $delete : (A \times (Ptr \cup Vls)) \rightarrow A$.

$$delete[\alpha, m] := \begin{cases} \beta & , m = x \wedge x \in Ptr \\ \gamma & , m = x.s \wedge x.s \in Ptr \\ \delta & , m = x.z \wedge x.z \in Vls \\ \alpha & , (m = \zeta) \vee (m = \mathbf{null} \in Ptr) \end{cases}$$

where β, γ, δ are defined as described above as

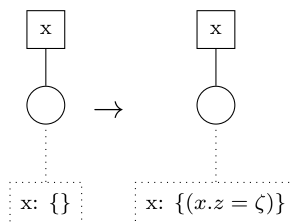
$$\begin{aligned} \beta := \alpha[e \mapsto \alpha(e) \backslash (\{ \quad & c \in \alpha(e) \mid c = (m \sim y) \vee \\ & c = (y \sim m) \vee c = (y \sim m.s) \vee \\ & c = (m.z_1 \sim y.z_2) \vee \\ & c = (y.z_2 \sim m.z_1)\}) \mid e \in E_\Sigma] \end{aligned}$$

$$\begin{aligned} \gamma := \alpha[e \mapsto \alpha(e) \backslash (\{ \quad & c \in \alpha(e) \mid c = (y \sim m) \\ \cup \{ \quad & (e.target = y), (e.source.z_1 \sim e.target.z_2) \mid \\ & lab(e) = s \wedge \exists e' \in E : \\ & lab(e') = x \wedge att(e)(1) = att(e')\}) \mid e \in E_\Sigma] \end{aligned}$$

$$\begin{aligned} \delta := \alpha[e \mapsto \alpha(e) \backslash (\{ \quad & c \in \alpha(e) \mid c = (m \sim y.z_1) \\ & \vee c = (y.z_1 \sim m)\} \\ \cup \{ \quad & (y.z = z') \mid lab(e) = y \wedge \\ & \exists e' \in E : lab(e') = x \wedge \\ & att(e) = att(e')\} \\ \cup \{ \quad & (e.source.z_1 \sim e.target.z_2) \mid \\ & \exists e' \in E : lab(e') = x \wedge \\ & (att(e)(1) = att(e') \vee \\ & att(e)(2) = att(e'))\}) \mid e \in E_\Sigma]. \end{aligned}$$

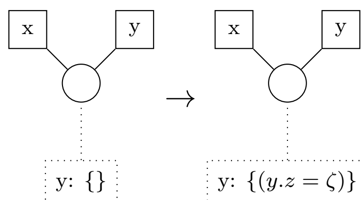
Initially α does not contain any information as it is filled by filtering information from the program. Every time a new piece of information can be obtained during the execution it is stored by the *learn* function. The used *insert* function simply stores the given condition set to α after any outdated information was removed by *delete* and similarly to *delete* takes into account side effects. The insertion of conditions has to consider where to store which piece of information. Valid inputs are sets C of conditions from the set C_α. For every condition $c \in C$ is differentiated if it is a value or pointer condition. If c is a value condition it is additionally checked if the condition is true or false. This is needed when filtering information out of conditions in **while** and **if** statements later on. If the condition is false in the current DHC the negated condition is inserted instead of c as α should only contain valid information. Value conditions are inserted depending on whether they contain information about one or two values of pointers. So if c is of the form $(x.z = \zeta)$ it only contains information about one pointer variable x. In this case c is inserted into the mapping of the edge that represents the program variable x in H like shown in the following.

Example 12. The following illustration shows the insertion of a value condition $(x.z = \zeta)$ into the mapping of the edge representing x.



It is taken into account that any pointer referencing the same object as x should have the same information, i.e. side effects are included.

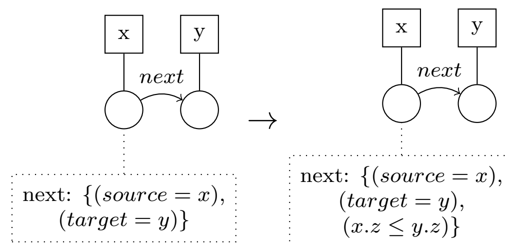
Example 13. Consider the following illustration of a DHC H^α during the call of $insert[\alpha, \{(x.z = \zeta)\}]$.



As x and y reference the same object, a condition $(y.z = \zeta)$ is inserted into the mapping of y.

If c is of the form $(x.z_1 \sim y.z_2)$, $x, y \in Ptr$, it is added to the mapping of any edge e with $att(e) = vu$ where x points to the object represented by v and y to the object represented by u, i.e. to any edge representing a selector of x that points to y. In case there is no such selector the condition c cannot be stored in this ansatz.

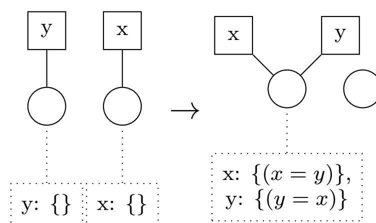
Example 14. The following illustration shows the impact of a call of $insert[\alpha, \{(x.z \leq y.z)\}]$ on the mapping of an edge that is attached to the objects that x and y reference.



The given condition is inserted into the mapping of the edge labelled *next*.

Conditions for relations between pointers are also contained in C_α. If c is such a condition of the form $(x = y)$ it is inserted into the mapping of the edge representing x and with swapped arguments into the edge representing y.

Example 15. During the execution of the statement $x := y$, the call of $insert[\alpha, \{(x = y)\}]$ introduces the following changes in the information function of the following DHC.



A formal definition of *insert* is given in the following.

Definition 10. Let $z' \in \mathbb{Z}$, $x, y \in Ptr$. Let $E_\Sigma = \{e \in E \mid lab(e) \notin N\}$ and for every $p \in Ptr \cup \{source, target\}$, $p.z, p.z_1, p.z_2 \in Vls$.

The function *insert* is defined as $insert : (A \times \mathcal{P}(C_\alpha)) \to A$ with

$$
\begin{aligned}
&insert[\alpha', C] := \\
&\alpha'[e \mapsto \alpha'(e) \cup \{ \quad interpret[c, H^{\alpha}] \mid (c \in C \wedge c \in VCnd) \wedge \\
&\qquad ((lab(e) = x \wedge c = (x.z \sim z')) \vee \\
&\qquad (att(e) = vu \wedge c = (x.z_1 \sim y.z_2) \wedge \\
&\qquad \exists e', e'' \in E : att(e') = v \wedge lab(e') = x \wedge \\
&\qquad att(e'') = u \wedge lab(e'') = y))\} \\
&\quad \cup \{ \quad (y.z \sim z') \mid (x.z \sim z') \in C \wedge lab(e) = y \wedge \\
&\qquad \exists e' \in E : lab(e') = x \wedge att(e) = att(e')\} \\
&\quad \cup \{ \quad c \mid c \in C \wedge c \in PCnd \wedge \\
&\qquad lab(e) = x \wedge c = (x = y)\} \\
&\quad \cup \{ \quad (y = x) \mid (x = y) \in C \wedge lab(e) = y \wedge \\
&\qquad \exists e' \in E : lab(e') = x \wedge att(e) = att(e')\} \mid e \in E_{\Sigma}].
\end{aligned}
$$

When the *insert* function receives a value condition $c_Z \in VCnd$ it determines its validity in H^{α} and negates c_Z if it is false. This is done by the function $interpret : (VCnd \times H) \to VCnd$. Let $c_Z \in VCnd$,

$$
interpret[c_Z, H^{\alpha}] := \begin{cases} c_Z & , [\![c_Z]\!]_{H^{\alpha}} = \text{true} \\ neg[c_Z] & , [\![c_Z]\!]_{H^{\alpha}} = \text{false}. \end{cases}
$$

A formal definition of the negation function *neg* can be found in the appendix on page 42.

Note that as described above, the symmetry of A for conditions is retained during changes in functions A. This is important in the *insert* function as implicitly additional conditions may be inserted.

The final *closure* function used in *learn* derives conditions about the actual objects that are the source and target of the edge from conditions about pointers and values of pointers in the information function. First, it derives conditions of the form $(e.source = x)$ and $(e.target = y)$, $x, y \in Ptr$, for edges e whose sources or targets are currently referenced by program variables x or y, respectively. After that *closure* looks for all conditions of the form $(x.z \sim y.z)$ with $\sim \in \{\neq, =, <, >, \leq, \geq\}$ that are stored for selector edges and derives them into conditions of the form $(source.z \sim target.z)$ if conditions $(source = x)$ and $(target = y)$ are stored for the edge. Such conditions can be kept even if the variables x and y are assigned other objects. As the deriving of such conditions potentially uses information obtained in the actions described before, this is done last in the *learn* function. The formal definition of *closure* is given below.

Definition 11. Let H^{α} be the given DHC, $x, y \in Ptr, \sim \in \{\neq, =, <, >, \leq, \geq\}$, $E_{\Sigma} = \{e \in E \mid lab(e) \notin N\}$. For every $p \in Ptr \cup \{source, target\}$, let $p.z_1, p.z_2 \in Vls$.

The function *closure* is defined as $closure : A \to A$.

$$\begin{aligned} closure[\alpha] := \beta[e \mapsto \beta(e) \cup \{ \quad & (e.source.z_1 \sim e.target.z_2) \mid \\ & \{(x.z_1 \sim y.z_2), (e.source = x), \\ & (e.target = y)\} \subseteq \beta(e)\} \mid e \in E_\Sigma] \end{aligned}$$

$$\begin{aligned} \beta := \alpha[e \mapsto \alpha(e) \cup \{ \quad & (e.source = x) \mid rk(e) = 2 \wedge \\ & \exists e' \in E : lab(e') = x \wedge \\ & att(e)(1) = att(e')\} \\ \cup \{ \quad & (e.target = y) \mid rk(e) = 2 \wedge \\ & \exists e' \in E : lab(e') = y \wedge \\ & att(e)(2) = att(e')\} \mid e \in E_\Sigma]. \end{aligned}$$

The rules for **while** and **if** statements change the heap just as their corresponding inference rules in the semantics of the pointer language defined in [8] which are the commonly used semantics of such statements. Additionally their guards are exploited to update the information function. The function *cz* obtains the set of all value conditions contained in a arbitrary condition $c \in Cnd$. The obtained set can be inserted and processed in the information function using *learn*. As mentioned before, *learn* handles such conditions by checking their validity in the current heap before inserting them. The function $cz[c]$ is defined inductively over the conditions of the programming language.

Definition 12. Let $c_Z \in VCnds, c_P \in PCnd, c_1, c_2 \in Cnd$.
$cz : Cnd \to \mathcal{P}(VCnd)$

$$\begin{aligned} & cz[c_Z] := \{c_Z\}, \\ & cz[c_P] := \emptyset, \\ & cz[c_1 \vee c_2] := cz[c_1] \cup cz[c_2], \\ & cz[c_1 \wedge c_2] := cz[c_1] \cup cz[c_2]. \end{aligned}$$

Note that in **while** and **if** statements **null** is given to *learn* as the second input argument. As *delete* performs no action on the information function in this case the execution of **while** and **if** statements does not cause the removal of any information.

3 Concretisation and Abstraction

In general the state space of heaps obtained by symbolic execution of programs may be infinite. For practical purposes it is important to have a finite state space

which can be obtained by grouping heap configurations and generalising them. Certain parts of heap configurations can be abstracted while they are not in direct use, i.e. while no program variable is referencing them. Whenever the program reaches a point in which it needs (parts of) the abstracted heap a concretisation is done in which concrete heap elements are restored.

The abstraction and concretisation defined in [8] is briefly described in this chapter. It cannot be applied to a data-aware heap configuration H^{α} without loosing information stored in α. Two approaches at abstracting and concretising a data-aware heap representation will be examined, both retain at least a subset of the original information.

3.1 Basic Principles

To formalise abstraction and concretisation, hyperedge replacement grammars (HRGs) are used. HRGs define production rules for replacing nonterminal symbols with heap configurations. The general concept of abstracting a heap configuration can be understood as a backward rule application of the production rules of an HRG to a heap configuration yielding a smaller heap configuration which contains nonterminal symbols.

Correspondingly concretisation can be understood as a forward rule application of an HRG. By using concretisation abstracted parts of a heap configuration are transformed into concrete heap parts.

A nonterminal symbol in an abstracted heap configuration can be replaced by using the rules of an HRG. The replacements might result in a fully concrete heap configuration or one that again contains nonterminal symbols. Therefore an abstracted heap configuration might be capable of representing infinitely many heap configurations depending on the used HRG.

The basic concepts required to understand HRG-based concretisation and abstraction are briefly described in the following. This representation follows the definitions in [8].

Definition 13. Hyperedge Replacement Grammar. A hyperedge replacement grammar G is a finite set of production rules $X \to H$ with $X \in N$, $H \in HG_{\Sigma_N}$, $\mid ext_H \mid = rk(X)$. HRG_{Σ_N} denotes the set of all HRGs over Σ_N.

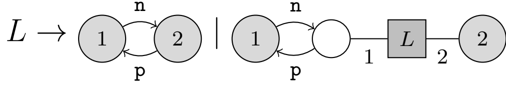
Example 16. An example of an HRG for doubly-linked lists (DLLs) is shown in Figure 9. The abbreviation 'n' stands for the selector name 'next', 'p' for the selector name 'prev'. It shows a simple grammar in which a nonterminal edge labelled L represents DLLs. The L edge can be replaced by either a doubly-linked list of length of at least one. The figure also shows a starting heap configuration which is the smallest abstraction of a heap representing an arbitrary DLL one

can create using the shown grammar. The first and last list element cannot be abstracted as they are referenced by program variables *head* and *tail*.

Figure 9: Hyperedge replacement grammar for doubly linked lists.

Smallest abstraction of a DLL:

HRG for DLLs:



The production rules of an HRG can be used to replace nonterminal edges with hypergraphs using hyperedge replacement. A hyperedge replacement takes place when a nonterminal edge with labelling X in the original graph H is replaced by a graph K. This can only happen if there is a rule $X \to K$ in the used HRG.

Definition 14. Hyperedge replacement. Let $H, K \in HG_{\Sigma_N}$, $e \in E_H$ be a nonterminal edge with $rk(e) = \mid ext_K \mid$. W.l.o.g. let $V_H \cap V_K = E_H \cap E_K = \emptyset$ which can be established by renaming vertices and edges of H or K. The replacement of e by K is denoted as $H[K \backslash e]$ and defined as $H' \in HG_{\Sigma_N}$ with

$$\begin{aligned} V_{H'} &= V_H \cup (V_K \backslash [ext_K]) & E_{H'} &= E_H \backslash \{e\}) \cup E_K \\ lab_{H'} &= (lab_H \restriction (E_H \backslash \{e\})) \cup lab_K & ext_{H'} &= ext_H \\ att_{H'} &= att_H \restriction (E_H \backslash \{e\}) \cup (mod \circ att_K) \end{aligned}$$

where $mod = id_{V_{H'}} \cup \{ \quad ext_K(1) \mapsto att_H(e)(1), \dots, \; ext_K(rk(e)) \mapsto att_H(e)(rk(e))\}$

and $\cdot \restriction E$ restricts $\cdot$ to the domain $E \subseteq dom(\cdot)$.

An application of a hyperedge replacement according to a production rule is called a derivation step. An HRG derivation is a sequence of derivation steps.

Definition 15. For $G \in HRG_{\Sigma_N}$, $H, H' \in HG_{\Sigma_N}$, $p = X \to K \in G$ and $e \in E_H$ with $lab_H(e) = X$ a **derivation** of H' from H by p is possible iff $H' \cong H[K \backslash e]$.

$H' \cong H[K \backslash e]$ denotes that H' is isomorphic to $H[K \backslash e]$. A derivation step is denoted by $H \Rightarrow_{e,p} H'$ or $H \Rightarrow H'$ if G is clear from the context. $\Rightarrow^*$ denotes the reflexive and transitive closure of $\Rightarrow$.

The language of a hypergraph H under an HRG G is the set $\mathcal{L}_G(H)$ of hypergraphs that can be derived from H by successively replacing hyperedges according to the production rules of G that do not contain nonterminal symbols.

Definition 16. Formally, the **language** of a heap configuration under an HRG is defined as $\mathcal{L}_G(H) = \{K \in HG_\Sigma \mid H \Rightarrow^* K\}$.

The language that can be derived from a nonterminal symbol X is denoted as $\mathcal{L}(X^\bullet)$. In this thesis only so-called **data structure grammars** (DSGs) are consistent, i.e. grammars $G \in \mathrm{HRG}_{\Sigma_N}$ for which holds $\forall X \in N : \mathcal{L}(X^\bullet) \subseteq HC_{Sel_\Sigma}$. The set of all DSGs over Σ_N is denoted as DSG_{Σ_N}.

DSGs ensure that all concrete heap configurations that are derivable from nonterminal edges via an HRG do not contain pointer edges. Therefore they can express that only such parts can be derived that are not referenced by the program. This property is needed as abstraction is only allowed if the part that is to be abstracted is not currently referenced by the program.

A heap configuration is called admissible if pointer manipulation only effects concrete parts of the heap configuration, i.e. there exists no vertex with terminal outgoing edges representing program variables that is in the sequence of an edge with a nonterminal symbol as labelling. Formally this is defined as follows.

Definition 17. Let $G \in HRG, H \in HC_{\Sigma_N}$. Consider a production rule $p = X \to K$ in G in which there are outgoing terminal edges from the ith external vertex derived from an edge $e \in E_H$ with $lab(e) = X$. The pair $(e, i) \in E \times \mathbb{N}$ is called a violation point if there exists an edge $e' \in E_H$ with $lab(e') \in Var_\Sigma \wedge att(e')(1) = att(e)(i)$. If there is no such violation point, H is called **admissible**. The set of all admissible heap configurations is denoted as AHC_{Σ_N}.

As stated above concretisation is the forward application of rules of an HRG. With the foregone definitions it is possible to formally define this concept.

Definition 18. Concretisation. Consider a given DSG $G \in DSG_{\Sigma_N}$. Concretisation is the forward application of the rules of G to a heap configuration, i.e. a heap configuration $H' \in HC_{\Sigma_N}$ is called a concretisation of $H \in HC_{\Sigma_N}$ iff $H \Rightarrow^* H'$.

Abstraction can be seen as the backward application of production rules. Therefore a way to define specified parts of a given heap configuration are needed to define what parts of the heap are to be replaced by a nonterminal edge. These parts are formally expressed by using embeddings.

Definition 19. Let I, $H \in HG_\Sigma$. An **embedding** *emb* of I in H is a pair (emb_V,emb_E) with the following properties:

$emb_V : V_I \to V_H$
$emb_E : E_I \to E_H$
$\forall v \in V_I \backslash [ext_I]\big(emb_V(v) \notin [ext_H]\big)$
$\forall v \in V_I, \forall v' \in V_I \backslash [ext_I]\big((v \neq v') \to (emb_V(v) \neq emb_V(v'))\big)$
$\forall e, e' \in E_I\big((e \neq e') \to (emb_E(e) \neq emb_E(e'))\big)$
$\forall e \in E_I\big(lab_I(e) = lab_H(emb_E(e))\big)$
$\forall e \in E_I\big(emb_V(att_I(e)) = att_H(emb_E(e))\big)$
$\forall e \in E_H\big((e \notin emb_E(E_I)) \to ([att_H(e)] \cap emb_V(V_I) \neq \emptyset)\big)$

For a sequence of vertices seq, $[seq]$ is the set of all vertices that appear in seq. $Emb(I, H)$ denotes the set of all embeddings of I in H.

Definition 20. Abstraction. Abstraction is the reverse application of production rules of an HRG $G \in HRG$ to a heap configuration $H' \in HC_{\Sigma_N}$. So a subgraph I in H' with $Emb(I, H') \neq \emptyset$ that is isomorphic to a rule graph on the righthand side of a production rule in G is replaced in H' by the nonterminal edge on the lefthand side of that rule. Formally, $H \in HC_{\Sigma_N}$ is called an abstraction of H' if $H \Rightarrow^* H'$.

3.2 Data-Aware Concretisation and Abstraction

So far, abstraction does not take data into account. To construct data-aware abstraction there is need for concretisation and abstraction techniques that can handle data as well as a definition for embeddings of DHCs that can be used to determine what rules of an HRG are applicable to what parts of a DHC. This thesis shows two different approaches at including data in abstraction which define different requirements for embeddings and abstraction. In this chapter a basic definition for embeddings, concretisation and abstraction with data is given.

As an example of why this is necessary consider the abstraction of a data-aware heap configuration $H^\alpha \in DHC_{\Sigma_N}$ which is defined exactly as the abstraction of a heap configuration K. Let $K = H$. Using an HRG G to abstract K, H^α yields the exact same heap configurations but α is not taken into account. Assume α is simply transferred to such an abstracted graph. It might contain outdated information about vertices and edges that do not exist any more. Therefore information about abstracted parts of a data-aware heap configuration must be transformed into a storable format in the abstraction steps. The two different approaches in this thesis keep different sets of information. They share that outdated information are deleted from the information function and that gathered information is stored in the new nonterminal that is introduced during the abstraction step. During concretisation steps the information is restored to the new concrete elements. The formalisation of all necessary components with data is given in the following.

Embeddings for DHCs are restricted to take present data into account by ensuring that only such DHCs I^β are in H^α whose information function β is the same as α for the edges in I. They also include the property that the conjunction of the mappings of all selector edges with the same labelling and all information stored on nonterminal edges for the same selector in β must imply the conjunction of the conditions of some common set. This way it is ensured that the embedded subgraph can be abstracted with the method defined later in this chapter. A placeholder is introduced such that further restrictions can be inserted.

Definition 21. A **data-aware embedding** of I^β in H^α with $I^\beta, H^\alpha \in DHC_{\Sigma_N}$ is defined as an embedding $(emb_V, emb_E) \in Emb(I, H)$ such that

$$\begin{array}{ll}
\forall v \in V_I: & \beta(v) = \alpha(emb_V(v)), \\
\forall e \in E_I: & \beta(e) = \alpha(emb_E(e)), \\
\forall s \in Sel: & \exists C_s \subseteq C_\alpha : \forall e \in E_I: \\
 & (lab_I(e) = s \rightarrow (\bigwedge(c_\beta \in \beta(e)) \rightarrow \bigwedge(c_s \in C_s)))\wedge \\
 & (lab_I(e) \in N \rightarrow (\bigwedge(c_\beta \in \{c_e \mid (s, c_e) \in \beta(e)\}) \rightarrow \bigwedge(c_s \in C_s))), \\
\text{and } \phi, &
\end{array}$$

where ϕ is an additional restriction to be defined by each abstraction approach in the following.

The set of all embeddings I^β in H^α is denoted analogously to the one for I in H as $Emb(I^\beta, H^\alpha)$. Note that $Emb(I, H) \subseteq Emb(I^\beta, H^\alpha)$.

Concretisation is the forward application of production rules of an HRG to a heap configuration. For DHCs the concretisation of the heap configuration works as described in chapter 3.1. During a concretisation step (derivation step) of a DHC the information about the nonterminal edge that is to be removed must be restored to the part of the graph that it is removed with.

Again, for concretisation only the set of **admissible** DHCs which is denoted by $ADHC_{\Sigma_N}$ is considered, where a DHC H^α is admissible if H is admissible.

For the changes in the information function during concretisation the following property is needed.

Definition 22. For information functions $\alpha, \alpha' \in A$, α' is **more concrete** than α denoted as $\alpha' \sqsubseteq \alpha$ if the following property holds.

$$\begin{array}{ll}
\alpha' \sqsubseteq \alpha :\Leftrightarrow & dom(\alpha) = dom(\alpha') \wedge \forall e \in dom(\alpha): \\
 & (lab(e) = s \rightarrow (\bigwedge_{c \in \alpha'(e)} c \rightarrow \bigwedge_{c \in \alpha(e)} c))
\end{array}$$

For simplicity, this property is also defined for sets $C_1, C_2 \subseteq C_\alpha$.

$$C_1 \sqsubseteq C_2 :\Leftrightarrow \quad \bigwedge_{c \in C_1} c \rightarrow \bigwedge_{c \in C_2} c$$

For each possible concretisation $H'^{\alpha'}$ of a DHC H^{α} the information function α' contains new information for newly introduced edges, the information for the parts of the heap that are simply transferred from H stay the same as in α. All nonterminal edges in the new part of the graph are mapped to the same set of tuples that the replaced nonterminal edge e_N was mapped to in α. All newly introduced terminal edges are mapped to a set of conditions $c \in C_\alpha$ whose conjunction is implied by the conjunction of all conditions that were paired with the edge's labelling in $\alpha(e_N)$. Formally, this is defined as follows.

Definition 23. Let the DSG G be used for concretising a nonterminal edge e_N with labelling X in a DHC H^{α}. A **derivation step** on H^{α} (using G) is denoted as $H^{\alpha} \Rrightarrow H'^{\alpha'}$ and defined as

$$\begin{aligned} H^{\alpha} \Rrightarrow H'^{\alpha'} := \quad & \exists e_N \in E_H \wedge lab_H(e_N) \rightarrow K \in G \wedge H' \cong H[K \backslash e_N] \wedge \\ & \alpha' = \alpha[K^{\gamma} \backslash e_N] \end{aligned}$$

where $\alpha[K^{\gamma} \backslash e_N]$ is defined as follows. Let $e \in E_{H'}$.

$$\alpha[K^{\gamma} \backslash e_N](e) \mapsto \begin{cases} \alpha(e) & , e \in dom(\alpha) \backslash e_N \\ C \subseteq C_\alpha & , e \in E_K \wedge lab(e) = s \in Sel \wedge \\ & \bigwedge_{c \in C} c \rightarrow \bigwedge_{c_e \in \{c | (s,c) \in \alpha(e_N)\}} c_e \\ \alpha(e_N) & , e \in E_K \wedge lab(e) \in N. \end{cases}$$

Using the above definitions, concretisation functions can be defined for DHCs. A concretisation of a DHC H^{α} is a derivation H'^{β} of H^{α} or may have a more concrete information function, i.e. it can also be a DHC $H'^{\alpha'}$ where α' is more concrete than β.

Definition 24. A **concretisation function** for data-aware heap configurations is defined as $con : DHC_{\Sigma_N} \rightarrow 2^{ADHC_{\Sigma_N}}$ with $con(H^{\alpha}) := \{H'^{\alpha'} \mid H^{\alpha} \Rrightarrow^{*} H'^{\beta} \wedge \alpha' \sqsubseteq \beta\}$.

Definition 25. Formally, the **language** of a DHC $H^{\alpha} \in DHC_{\Sigma_N}$ under an HRG G is defined as $\mathcal{L}_G(H^{\alpha}) := \{H'^{\alpha'} \mid H'^{\alpha'} \in con_G(H^{\alpha}) \cap DHC_{\Sigma}\}$ and can be denoted as $\mathcal{L}(H^{\alpha})$ if G is clear from the context. I.e. the language of a DHC is defined as all derivable concrete DHCs.

The language of a set of DHCs $\mathcal{H} \subseteq DHC_{\Sigma_N}$ is defined as the union of the languages of the set members. Formally, $\mathcal{L}_G(\mathcal{H}) = \bigcup_{H^{\alpha} \in \mathcal{H}} \mathcal{L}_G(H^{\alpha})$. Again, if G is clear from the context it can be left out in the notation.

For heap configurations it holds $\mathcal{L}(H) = \cup_{H' \in con(H)} \mathcal{L}(H')$ ([8]), so the language of a heap configuration contains exactly the languages of all of its concretisations. This property is presented by DHCs.

Lemma 1. *For the concretisation function con and a data-aware heap configurations H^α it holds $\mathcal{L}(H^\alpha) = \cup_{H'^{\alpha'} \in con(H^\alpha)} \mathcal{L}(H'^{\alpha'})$.*

Proof. Let $G \in HRG$, $H^\alpha \in DHC_{\Sigma_N}$.

$$\begin{aligned}\mathcal{L}(H^\alpha) &= \{H'^{\alpha'} \mid H'^{\alpha'} \in con_G(H^\alpha) \cap DHC_\Sigma\} \\ &= \{H'^{\alpha'} \mid H^\alpha \Rrightarrow^* H'^\gamma \wedge \alpha' \sqsubseteq \gamma \wedge H'^{\alpha'} \in DHC_\Sigma\} \\ &= \{H'^{\alpha'} \mid H'^{\alpha'} \in \quad (\{H^\alpha\} \cup \{H^\gamma \mid \gamma \sqsubseteq \alpha\} \cup \textstyle\bigcup_{H^\alpha \Rrightarrow^+ I^\beta} \mathcal{L}(I^\beta)) \\ &\qquad \cap DHC_\Sigma\}.\end{aligned}$$

Then from $(H^\alpha \Rrightarrow^+ I^\beta \Rrightarrow^* H'^{\alpha'} \vee H'^{\alpha'} = H^\alpha) \wedge H'^{\alpha'} \in DHC_\Sigma$ it follows that $H'^{\alpha'} \in \mathcal{L}(H^\alpha)$.

As a consequence $\cup_{H'^{\alpha'} \in con(H^\alpha)} \mathcal{L}(H'^{\alpha'}) = \mathcal{L}(H^\alpha)$. □

As explained in chapter 3.1, abstraction can be seen as a part I of a given heap configuration K with $Emb(I, K) \neq \emptyset$ being replaced by a nonterminal edge corresponding to a rule in an HRG G in which a righthand rule graph is isomorphic to I. This abstraction is the backwards application of production rules. For DHCs, data-awareness during abstraction must be introduced.

To retain a set of recoverable information during abstraction, nonterminal edges in the heap are mapped to sets of tuples of selectors and conditions by the information function. This kind of mapping is defined in A as $C_\Sigma \subseteq C_{data}$ but it is not used for concrete parts of heap configurations. Note that SC_{new} only contains tuples of selectors and conditions. Although program variables are represented by edges they are not considered as parts of the heap that are currently referenced by the program cannot be abstracted. Therefore nonterminal edges do not need to contain any information about variable edges.

Definition 26. Let $H^\alpha, H'^{\alpha'} \in DHC_{\Sigma_N}$ with $H^\alpha \Rrightarrow H'^{\alpha'}$ using an HRG G. The **backward application** of a production rule in G is denoted as $H^\alpha \Lleftarrow H'^{\alpha'}$ and defined as

$$\begin{aligned}H^\alpha \Lleftarrow H'^{\alpha'} := \quad & \exists emb \in Emb(I^\beta, H^\alpha), \exists e_N \in E_{H'} : \\ & lab(e) \to I \in G \wedge H'[I \backslash e_N] \cong H \wedge \alpha' = \alpha[e_N \backslash I^\beta]\end{aligned}$$

where $\alpha[e_N \backslash I^\beta]$ is defined for $e \in E_{H'}$ as

$$\alpha[e_N\backslash I^{\beta}](e) := \begin{cases} \alpha(e) & , e \in dom(\alpha)\backslash dom(\beta) \\ SC_{new} \subseteq C_{\Sigma} & , e = e_N \end{cases}$$

The set SC_{new} depends on the used approach of abstracting with data and will be defined differently for each approach.

With the help of backward derivation, abstraction functions can be defined for DHCs.

Definition 27. Let G be an HRG. **Abstraction functions** for data-aware heap configurations are defined as $abs : DHC_{\Sigma_N} \to 2^{DHC_{\Sigma_N}}$. The abstraction of a $H^{\alpha} \in DHC_{\Sigma_N}$ is $abs(H^{\alpha}) := \{H'^{\alpha'} \mid H^{\alpha} \ {}^{+}\!\Lleftarrow H'^{\alpha'}\}$.

The program step in which an abstracted node is to be referenced can only happen after concretising the corresponding part of the heap.

As the abstraction is supposed to compute a generalisation of a DHC it should hold that the language of the abstraction of a DHC includes the language of the original DHC and might even be larger.

Lemma 2. *For $H^{\alpha} \in DHC_{\Sigma_N}$ with $\alpha \sqsubseteq (\alpha[e_N\backslash I^{\beta}])[I^{\beta}\backslash e_N]$ for I^{β} such that $Emb(I^{\beta}, H^{\alpha}) \neq \emptyset$ it holds that $\mathcal{L}(H^{\alpha}) \subseteq \mathcal{L}(abs(H^{\alpha}))$.*

Proof. Let $G \in DSG$ and $H^{\alpha}, H'^{\alpha'}, I'^{\beta'}, I^{\delta}, K^{\gamma} \in DHC_{\Sigma_N}$.

By the definition of $\mathcal{L}(abs(H^{\alpha}))$ it suffices to show $H^{\alpha} \in con(abs(H^{\alpha}))$.

$$\begin{aligned} con(abs(H^{\alpha})) &= con(\{K^{\gamma} \mid H^{\alpha} \ {}^{+}\!\Lleftarrow K^{\gamma}\}) \\ &= \{I'^{\beta'} \mid H'^{\alpha'} \Rrightarrow^{*} I'^{\delta} \wedge \beta' \sqsubseteq \delta \wedge H'^{\alpha'} \in \{K^{\gamma} \mid H^{\alpha} \ {}^{+}\!\Lleftarrow K^{\gamma}\}\} \\ &= \{I'^{\beta'} \mid H'^{\alpha'} \Rrightarrow^{*} I'^{\delta} \wedge \beta' \sqsubseteq \delta \wedge H^{\alpha} \ {}^{+}\!\Lleftarrow H'^{\alpha'}\} \end{aligned}$$

Hypothesis:
$H^{\alpha} \in \{I'^{\beta'} \mid H'^{\alpha'} \Rrightarrow I'^{\delta} \wedge \beta' \sqsubseteq \delta \wedge H^{\alpha} \Lleftarrow H'^{\alpha'}\}$.

Remember the definition of $H^{\alpha} \Lleftarrow H'^{\alpha'}$:

$$\begin{aligned} H^{\alpha} \Lleftarrow H'^{\alpha'} :\Leftrightarrow \quad & \exists emb \in Emb(I^{\beta}, H^{\alpha}), \exists e_N \in E_{H'} : \\ & lab(e) \to I \in G \wedge H'[I\backslash e_N] \cong H \wedge \alpha' = \alpha[e_N\backslash I^{\beta}] \end{aligned}$$

So $con(abs(H^{\alpha})) = con(H'^{\alpha[e_N\backslash I^{\beta}]})$.
As $\mathcal{L}(H) \subseteq \mathcal{L}(H')$ with the above definition of H' is shown in [8] (Lemma 1),
$H^{\alpha} \in \{I'^{\beta'} \mid H'^{\alpha[e_N\backslash I^{\beta}]} \Rrightarrow I'^{\delta} \wedge \beta' \sqsubseteq \delta\}$ if $\alpha \sqsubseteq (\alpha[e_N\backslash I^{\beta}])[I^{\beta}\backslash e_N]$
which is a prerequisite of the Lemma.

So $H^{\alpha} \in \{I'^{\beta'} \mid H'^{\alpha'} \Rrightarrow I'^{\delta} \wedge \beta' \sqsubseteq \delta \wedge H^{\alpha} \Lleftarrow H'^{\alpha'}\}$ holds which implies
$H^{\alpha} \in con(abs(H^{\alpha}))$. □

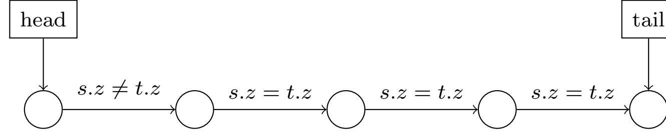
The prerequisite of Lemma 2 is investigated in the different approaches.

Example 17. This is an example of abstracting and concretising a DHC H^α which represents a singly linked list (SLL). Consider the HRG for SLLs in Figure 10. The labelling 'n' of the edges in the HRG is an abbreviation for the selector '*next*'.

Figure 10: HRG for singly linked lists.

Figure 11 visualises H^α. The labelling *next* of the edges is omitted, instead the labelling of the edges represents the information stored in α for each edge. For simplicity it is assumed that for each selector edge only one condition is stored in α. The letter s stands for *source*, the letter t for *target*, z is the name of the data variable of list objects. The pointer variables *head* and *tail* explicitly reference two vertices in the heap such that they cannot be abstracted.

Figure 11: Visualisation of H^α.



Let I^β in H^α with $Emb(I^\beta, H^\alpha) \neq \emptyset$ be the subgraph of H^α that contains the third and fourth vertex from the left as well as the edge attached in between. Assume there is an abstraction function that abstracts DHCs using the HRG in Figure 10. As $Emb(I^\beta, H^\alpha) \neq \emptyset$ that function can be applied to H^α to replace I^β with a nonterminal L. Assume that for every currently possible abstraction step the information function maps the newly introduced edge with labelling L to a set $SC := \{(next, (s.z = t.z))\}$. The abstracted heap $H'^{\alpha'}$ is the following. The information stored for the resulting nonterminal edge is visualised in the dotted box above it.

Now assume the program needs the selector *next* of the object represented by the second vertex which is currently abstracted in the heap configuration. A concretisation step is necessary. The nonterminal edge e_L in $H'^{\alpha'}$ is replaced with the second rule graph in the used HRG. The information function must store for every newly introduced terminal edge e a set whose conjunction is implied by the conjunction of all conditions in $\alpha(e_L)$ that is paired with $lab(e)$. The resulting heap is given in Figure 13.

Figure 12: Visualisation of abstraction of H^{α}.

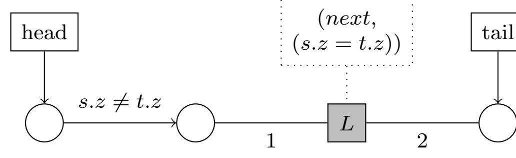
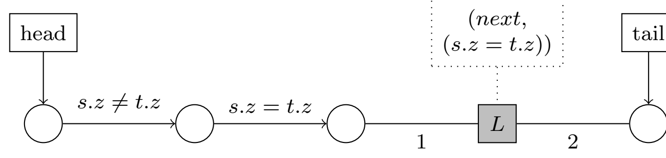


Figure 13: Visualisation of concretisation of abstraction of H^{α}.



3.3 Correctness

It remains to be shown that the proposed abstraction is sound. For this it must first be shown that operations as performed by the semantics of the programming language and concretisation steps are commutable.

Lemma 3. *For $H^{\alpha} \in ADHC_{\Sigma_N}$ and manipulation $\xi \in \{x \hookrightarrow \textbf{null}, x \hookrightarrow v, x \overset{s}{\hookrightarrow} \textbf{null}, x \overset{s}{\hookrightarrow} u, +v, \alpha[e \mapsto C]\}$ it holds that $\{H'^{\alpha'}[\xi] \mid H'^{\alpha'} \in \mathcal{L}(H^{\alpha})\} = \mathcal{L}(H^{\alpha}[\xi])$.*

Proof. The property $\{H'[\mu] \mid H' \in \mathcal{L}(H)\} = \mathcal{L}(H[\mu])$ with $\mu \in \{x \hookrightarrow \textbf{null}, x \hookrightarrow v, v \overset{s}{\hookrightarrow} \textbf{null}, v \overset{s}{\hookrightarrow} u, +v\}$ has already been shown in [8] (Lemma 3).
It remains to be shown that $\{H^{\alpha'[e \mapsto C]} \mid H'^{\alpha'} \in \mathcal{L}(H^{\alpha})\} = \mathcal{L}(H^{\alpha[e \mapsto C]})$.
Let $G \in DSG, H^{\alpha}, H'^{\alpha'} I^{\beta} \in DHC_{\Sigma_N}$ with $Emb(I^{\beta}, H'^{\alpha'}) \neq \emptyset$ and $e_N \in E_H$ a nonterminal edge. Assume there exists a production rule in G such that $H'^{\alpha'} = H[I^{\beta} \backslash e_N]^{\alpha[I^{\beta} \backslash e_N]}$.
By Lemma 1 $\{H^{\alpha'[e \mapsto C]} \mid H'^{\alpha'} \in \mathcal{L}(H^{\alpha})\} = \mathcal{L}(H^{\alpha[e \mapsto C]})$ holds if
$(\alpha[e \mapsto C])[I^{\beta} \backslash e_N] = (\alpha[I^{\beta} \backslash e_N])[e \mapsto C]$ for $e \in E_H$.

Let $d \in E_{H'}$.
By definition,

$$\alpha[I^{\beta} \backslash e_N](d) = \begin{cases} \alpha(d) & , d \in dom(\alpha) \backslash e_N \\ C \subseteq C_{\alpha} & , d \in E_I \wedge lab(d) = s \in Sel \wedge \\ & \bigwedge_{c \in C} c \rightarrow \bigwedge_{c_e \in \{c \mid (s,c) \in \alpha(e_N)\}} c_e \\ \alpha(e_N) & , d \in E_I \wedge lab(d) \in N. \end{cases}$$

It follows that

$$(\alpha[I^\beta \backslash e_N])[e \mapsto C](d) = \begin{cases} \alpha[e \mapsto C](d) & , d \in dom(\alpha)\backslash e_N \\ C \subseteq C_\alpha & , d \in E_I \wedge lab(d) = s \in Sel \wedge \\ & \bigwedge_{c \in C} c \rightarrow \bigwedge_{c_e \in \{c \mid (s,c) \in \alpha[e \mapsto C](e_N)\}} c_e \\ \alpha[e \mapsto C](e_N) & , d \in E_I \wedge lab(d) \in N \end{cases}$$

$$= (\alpha[e \mapsto C])[I^\beta \backslash e_N](d)$$

So changing conditions of terminal edges is commutable with concretisation. □

In the following the result of [8] concerning soundness is adapted to work for data-aware heap configurations.

Definition 28. Let the abstract semantics be defined by an abstract transition relation $\blacktriangleright \subseteq (Stms \times ADHC_{\Sigma_N}) \times ((Stms \cup \{\varepsilon\}) \times ADHC_{\Sigma_N})$ with given abstraction function $abs : DHC_{\Sigma_N} \rightarrow DHC_{\Sigma_N}$, concretisation function $con : DHC_{\Sigma_N} \rightarrow 2^{ADHC_{\Sigma_N}}$ and given concrete semantics $\triangleright$ as

$$\frac{\langle S, H^\alpha \rangle \triangleright \langle S', I'^{\beta'} \rangle \quad H'^{\alpha'} \in cons(abs(I'^{\beta'}))}{\langle S, H^\alpha \rangle \blacktriangleright \langle S', H'^{\alpha'} \rangle}.$$

Theorem 1 (Soundness)**.** *Let $H^\alpha \in DHC_{\Sigma_N}$. For the abstraction function abs with $\mathcal{L}(H^\alpha) \subseteq \mathcal{L}(abs(H^\alpha))$ and for the concretisation function con with $\mathcal{L}(H^\alpha) = \cup_{H'^{\alpha'} \in con(H^\alpha)} \mathcal{L}(H'^{\alpha'})$, $\blacktriangleright$ is an over-approximation of the transition relation $\triangleright$.*

Proof. It must be shown that for $\langle S, K^\gamma \rangle \triangleright \langle S', K'^{\gamma'} \rangle$with $S \in Stms$, $S' \in Stms \cup \{\varepsilon\}$, $K^\gamma, K'^{\gamma'} \in DHC_\Sigma$ there exists $H^\alpha \in con(abs(K^\gamma))$, $H'^{\alpha'} \in ADHC_{\Sigma_N}$ such that $\langle S, H^\alpha \rangle \blacktriangleright \langle S', H'^{\alpha'} \rangle$and $K'^{\gamma'} \in \mathcal{L}(H'^{\alpha'})$.

By Definition 25 and Lemma 2, $\mathcal{L}(K^\gamma) = \{K^\gamma\} \subseteq \mathcal{L}(abs(K^\gamma))$.
By Lemma 1, there exists $H^\alpha \in con(abs(K^\gamma))$ such that $K^\gamma \in \mathcal{L}(H^\alpha)$.
As $H^\alpha \in con(abs(K^\gamma))$, H^α is admissible.
Hence, there exists $I^\beta \in DHC_{\Sigma_N}$ such that $\langle S, H^\alpha \rangle \triangleright \langle S', I^\beta \rangle$.
By Lemma 3, $K'^{\gamma'} \in \mathcal{L}(I^\beta) \subseteq \mathcal{L}(abs(I^\beta))$.
Moreover there exists $H'^{\alpha'} \in con(abs(I^\beta))$ with $K'^{\gamma'} \in \mathcal{L}(H'^{\alpha'})$.
Then, by Definition 28, $\langle S, H^\alpha \rangle \blacktriangleright \langle S', H'^{\alpha'} \rangle$. □

Under the premises of Theorem 1, the presented notion of data-aware abstraction is sound. In the following chapters different instantiations of such are studied.

3.4 Loss-free Abstraction

In order to retain all information stored in the information function of a data-aware heap configuration $H^\alpha \in DHC_{\Sigma_N}$ during the abstraction additional requirements to the ones described in chapter 3.2 have to be met. Only subgraphs can be abstracted in which each selector edge is equipped with the same set of conditions, i.e. only subgraphs I^β in H^α can be abstracted in which for all edges $e_u \in E_I$ whose source is $u \in V_I$ there is an edge $e_v \in E_I$ whose source is $v \in V_I$ with $\alpha(e_u) = \alpha(e_v)$ and visa versa. Analogously, for all edges e_u whose target is u there is an edge e_v whose target is v with $\alpha(e_u) = \alpha(e_v)$ and vice versa. So to abstract a DHC without information loss a way to guarantee the characteristics mentioned above must be introduced.

To find subgraphs in a DHC abstractable with a given HRG the definition of embeddings for data-aware heap abstractions DHC_{Σ_N} is extended such that they consider the above restrictions. Embeddings for DHCs in this chapter are extended such that they match the definition given in chapter 3.2 with the property ϕ. ϕ ensures that all selector edges labelled with the same selector are mapped to the same set of conditions and that all nonterminal edges store the same set of conditions for the selectors it contains. So if the mapping of a nonterminal edge contains a pair (s, c) it must contain pairs for all and only the conditions that appear in the mappings of edges with labelling s. If a selector does not appear in the mapping of a nonterminal edge it was not abstracted yet and does not need to be considered. Formally, the additional restriction for embeddings is given by

$$\begin{aligned}\phi := \forall e, e' \in E_I : \quad & ((lab_I(e) \in Sel \wedge lab_I(e) = lab_I(e')) \\ & \rightarrow (\beta(e) = \beta(e'))) \wedge \\ & (lab_I(e) \in N \rightarrow \forall (s, c) \in \beta(e) : \\ & (lab_I(e') = s \rightarrow (c \in \beta(e') \wedge \\ & \forall c' \in \beta(e') : (s, c') \in \beta(e)))).\end{aligned}$$

$Emb(I^\beta, H^\alpha)$ only contains embeddings for which ϕ holds. The righthand side rule graphs of production rules of the HRG used for abstraction and concretisation can only be matched to isomorphic subgraphs I^β in H^α such that $Emb(I^\beta, H^\alpha) \neq \emptyset$. As I^β is embedded in H^α it is guaranteed that every condition in the information function of an edge $e \in E_I$ appears in all mappings of edges in E_I with the same labelling $s \in Sel$.

During abstraction the used abstraction function abs needs a set SC_{new} of tuples in C_Σ for every abstraction step to map the newly introduced nonterminal edge to. For this method of abstraction with no information loss a DHC H^α is abstracted to an DHC K^γ with I^β being the subgraph of H^α that is to be replaced by a new nonterminal edge $e_N \in K^\gamma$. The set SC_{new} is defined such that it contains

all conditions on selector edges in I paired with the selector and all pairs mapped to nonterminals by β. Formally,

$$\begin{aligned} SC_{new} := \quad & \{(s,c) \mid s \in Sel \wedge \forall e \in E_I : (lab(e)_I = s \rightarrow c \in \beta(e)) \\ & \wedge((lab_I(e) \in N \wedge (s, c' \in C_\alpha) \in \beta(e)) \rightarrow (s,c) \in \beta(e))\}. \end{aligned}$$

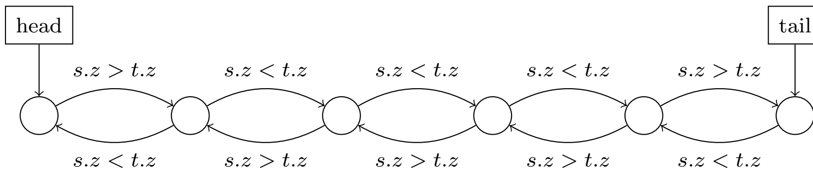
The above definition of SC_{new} allows the abstraction of parts of concrete heap configurations as well as the abstraction of subgraphs that contain terminal as well as nonterminal edges without the loss of any data. Therefore in this approach Lemma 2 holds as $\alpha = (\alpha[e_N \backslash I^\beta])[I^\beta \backslash e_N]$.

Note that information can only be fully preserved using this definition of SC_{new} because of the additional requirements for embeddings. Consider the definition of embeddings as presented in chapter 3.1. Abstraction in which only conditions that are in $\beta(e)$ for all edges $e \in E_I$ with the same selector as labelling are kept implies keeping the intersection of all sets of conditions stored for such edges which results in data loss.

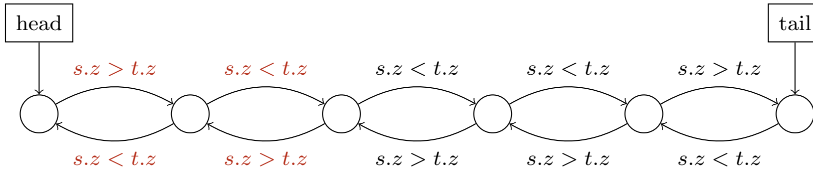
In the following an example is given for abstraction without data loss for a DHC representing a DLL.

Example 18. A data-aware heap configuration H^α of a DLL is abstracted without loss of data using the HRG for DLLs defined in Figure 9 (see page 24). Remember that ϕ holds for all $emb \in Emb(I^\beta, H^\alpha)$ for DHCs I^β and H^α.

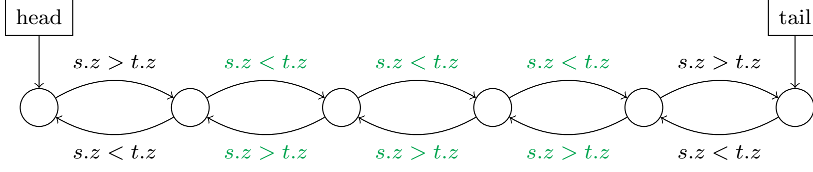
The following shows an example of the abstraction of the data-aware heap configuration H^α of a DLL. The labels of the selectors 'next' and 'prev' are not shown in the pictures. The value of an object is referenced in the mapping of an edge e by $v.z$ with $v \in \{e.source, e.target\}$. The labelling of edges $e \in E_H$ represent the data in α with s standing for $e.source$ and t for $e.target$. Assume *head* and *tail* to be edges representing program references. Given heap:



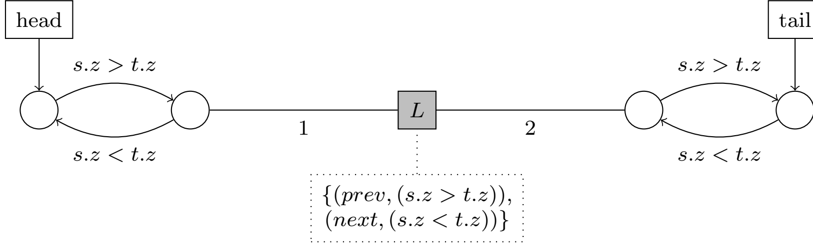
The DHC is searched for a subgraph which has an embedding in H^α that is replaceable by a nonterminal edge.



Continuing, only one such subgraph can be found as only for the edges attached to the two middle nodes are mapped to the same set of conditions in α:



The nodes and all their adjacent edges are abstracted which results in:



3.5 Abstraction with Common Conditions

While the abstraction as described in the chapter above does not result in data loss it might not be possible to find many opportunities to abstract DHCs as the requirements for abstractable parts may not be matched often in practice. Therefore another approach is presented in this chapter which uses weaker requirements that might be met more often and that still ensure that a certain subset of information is preserved during abstraction.

The idea of taking data into account in this approach is to only replace a subgraph of the given DHC with a nonterminal edge during abstraction if all edges that represent a certain selector imply a common set of conditions. During abstraction the information function stores tuples of the selector and every condition in this common set in the mapping of the new nonterminal edge. When concreting parts of the heap the recoverable information might be less specific than before but is not lost entirely.

Example 19. Assume there is an embedding in $Emb(I^\beta, H^\alpha)$ for DHCs I^β and H^α and there are only two edges e, e' in I^β labelled with the selector *next*. If $\beta(e) = \{(e.source < e.target)\}$ and $\beta(e') = \{(e'.source \leq e'.target)\}$ a common set of conditions implied by both $\beta(e)$ and $\beta(e')$ is $\{(e.source \leq e.target)\}$. Assume I^β can be abstracted by being replaced by a nonterminal edge e_{new} using a given HRG. The information function is altered such that the edge e_{new} maps to the set $\{(next, (e.source \leq e.target))\}$.

In this approach data-aware embeddings of a DHC I^β in a DHC H^α must have a property such that for all selector edges e in I the conjunction of all conditions in $\beta(e)$ implies the conjunction of all conditions in a common set of such. Analogously the conjunction of that same common condition set must also be implied by the conjunction of conditions paired with the selector respectively in the mapping of all nonterminal edges in I. As this is already defined in the basic definition in chapter 3.2, there is no need for further restrictions to data-aware embeddings.

When abstracting a DHC H^α using an HRG G to replace a DHC I^β in H^α with $Emb(I^\beta, H^\alpha) \neq \emptyset$ with a nonterminal edge e_{new} in each abstraction step a set $C_s \subseteq C_\alpha$ is determined for each selector $s \in Sel$. All conditions in the set C_s must be implied by the conjunction of the conditions in the mapping of all edges in I labelled with s. Additionally the conjunction all conditions in C_s must also be implied by the conjunction of all conditions that are paired with the selector name s in the mapping of a nonterminal edge in I. From the definition of embeddings follows that such a set exists in I^β for each selector.

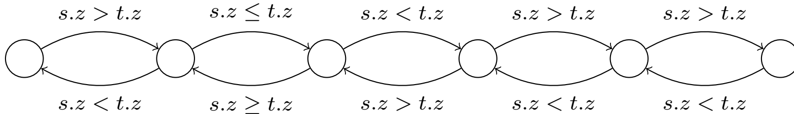
An abstraction function *abs* as defined in chapter 3.2 is used for abstracting a DHC H^α with common conditions. For each abstraction step the set SC_{new} is formally defined as follows. Let I^β have an embedding *emb* in the current DHC H^α and be the part that is to be replaced with a nonterminal edge e_N. Formally,

$$
\begin{aligned}
SC_{new} := \quad & \{(s,c) \mid s \in Sel \wedge \forall e \in E_I : \\
& (lab(e) = s \rightarrow \beta(e) \sqsubseteq \{c\}) \wedge \\
& (lab(e) \in N \rightarrow \{c_e \mid (s, c_e) \in \beta(e)\} \sqsubseteq \{c\})\}.
\end{aligned}
$$

So in each step, the new nonterminal edge e_N is mapped to a set of tuples in which for all selectors s all conditions in a set C_s as described above are contained. If nonterminal edges are abstracted the common set of conditions they store for a selector s might be generalised even more in order to include the current set C_s.

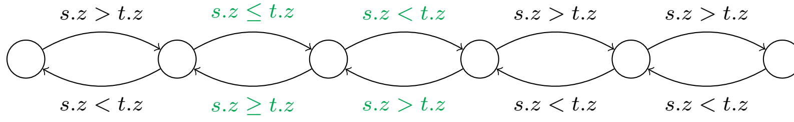
Note that even though there is some data loss parts of a DHC that are not referenced by the program might still not be abstractable due to the fact that no common set of conditions can be found. This can depend on the choice of common conditions. The following example illustrates how much of an impact on abstraction the choice of common conditions can be.

Example 20. A data-aware heap configuration H^α of a DLL is to be abstracted by using commonly implied sets of information. Let the following heap be a non-referenced fragment of H^α of the same form as in Example 18.

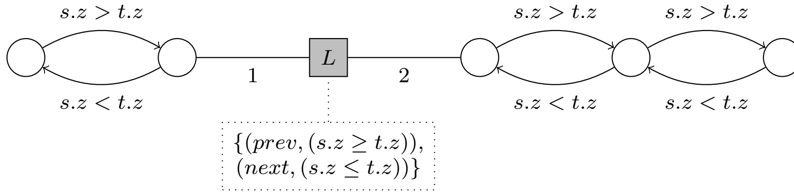


The HRG for DLLs defined in Figure 9 is used for abstraction and the heap is searched for abstractable embeddings starting from the lefthand side. In this example the largest coherent part with an embedding in the current graph that matches a righthand rule graph of the HRG is chosen to be abstracted. To illustrate that abstraction as defined in this chapter looks different depending on the choice of common conditions two such choices are considered.

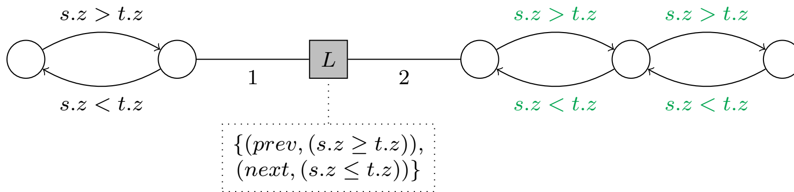
Assume that in the first order $(a < b) \wedge (a \leq b)$ implies $(a \leq b)$ and analogously $(a > b) \wedge (a \geq b)$ implies $(a \geq b)$. Using this method the heap is searched for an abstractable subgraph that has an embedding in H^{α}. The first found match for a righthand side rule graph of the HRG consists of the third node and all of its attached edges.



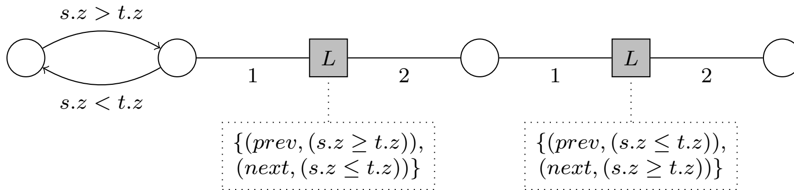
The resulting abstraction looks as follows.



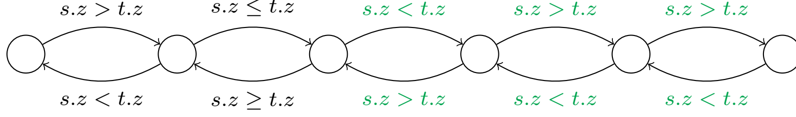
The search for abstractable parts of the graph resumes finding the followingly illustrated match.



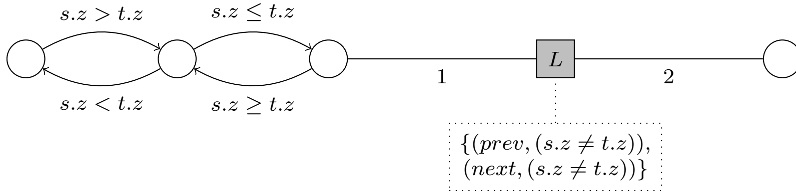
The resulting abstraction looks as follows.



Now assume that in the second choice of common conditions $(a < b) \wedge (a > b)$ implies $(a \neq b)$. The resulting abstraction of the shown graph fragment is quite different from the example using the first method.



The resulting abstraction looks as follows.



For Lemma 2 to hold in this approach it remains to be shown that for $H^{\alpha} \in DHC_{\Sigma_N}$ it holds that $\alpha \sqsubseteq (\alpha[e_N \backslash I^{\beta}])[I^{\beta} \backslash e_N]$ for I^{β} such that $Emb(I^{\beta}, H^{\alpha}) \neq \emptyset$. Let $G \in HRG$, $H'^{\alpha'} \in DHC_{\Sigma_N}$ with $H^{\alpha} \Leftarrow H'^{\alpha'}$, $d \in E_I$. Let $e_N \in E_{H'}$ such that $H'[I \backslash e_N] \cong H$.

By definition,

$$(\alpha[e_N \backslash I^{\beta}])[I^{\beta} \backslash e_N](d) = \begin{cases} \alpha[e_N \backslash I^{\beta}](d) & , d \in dom(\alpha[e_N \backslash I^{\beta}]) \backslash e_N \\ C \subseteq C_{\alpha} & , d \in E_I \wedge lab(d) = s \in Sel \wedge \\ & \bigwedge_{c \in C} c \rightarrow \bigwedge_{c_e \in \{c | (s,c) \in \alpha[e_N \backslash I^{\beta}](e_N)\}} c_e \\ \alpha[e_N \backslash I^{\beta}](e_N) & , d \in E_I \wedge lab(d) \in N. \end{cases}$$

Using the definition of $\alpha[e_N \backslash I^{\beta}](d)$, which is

$$\alpha[e_N \backslash I^{\beta}](d) = \begin{cases} \alpha(d) & , d \in dom(\alpha) \backslash dom(\beta) \\ SC_{new} \subseteq C_{\Sigma} & , d = e_N \end{cases}$$

results in

$$(\alpha[e_N \backslash I^{\beta}])[I^{\beta} \backslash e_N](d) = \begin{cases} \alpha(d) & , d \in E_H \backslash E_I \\ C \subseteq C_{\alpha} & , d \in E_I \wedge lab(d) = s \in Sel \wedge \\ & \bigwedge_{c \in C} c \rightarrow \bigwedge_{c_e \in \{c | (s,c) \in SC_{new}\}} c_e \\ SC_{new} & , d \in E_I \wedge lab(d) \in N. \end{cases}$$

where the first case holds as $\alpha \sqsubseteq \alpha$ and the second and third case hold per definition of SC_{new}.

4 Conclusion

In this thesis the ansatz by Heinen et al. in [8] was extended to include data. For this, a programming language was introduced that includes pointer and a handling for data fields of objects. Heap configurations were paired with newly introduced information functions to form data-aware heap configurations. Therefore the heap configurations were not changed as all data handling is managed by the functions. The introduced information functions map pointer edges of a heap configuration to sets of information about them. For concrete heap configurations these information are relations between program variables, sources and targets of the edges, values thereof and integer values. This information is automatically inferred during the symbolic execution of a program.

Thereafter a base for an automatic handling of data during the abstraction of heap configuration parts was introduced. It features restrictions for embeddings and extensions for data handling for concretisation and abstraction mechanisms. The soundness of them could be proven analogously to the proof in [8] by adding only one restriction for information functions.

Two sound approaches for abstraction were presented using this base. The first one only allows abstraction such that the exact information of the concrete parts of the heap is kept by allowing only the abstraction of heap parts for which the exact same information is stored for edges with the same labelling. This might be important for certain applications. However, the state space of the symbolic execution of a program might remain infinite as the approach might not allow sufficient abstraction. To potentially overcome this problem, during abstraction the second approach only stores sets of information that are commonly implied by all information about edges with the same labelling. The information in abstracted heap parts might be weaker than the one in the original heap, albeit it can be recovered during concretisation as the possibilities of stricter information in the information function is considered. Therefore this approach has no loss of information. As how much of a heap can be abstracted and what information is kept depends on the choice of common conditions the state space of the symbolic execution might still be infinite. Nevertheless this approach is flexible such that common conditions can be chosen differently depending on the intended application.

The next step to take is to apply the different abstraction approaches with data to the symbolic execution of actual programs. The data handling enriches the information achieved during these programs by for example information about the sortedness of lists. Therefore sorting algorithms like selection sort, of which an implementation in the introduced language is given in the appendix on page 44, are candidates for applying the abstraction approaches.

References

[1] Abdulla, P.A., Holík, L., Jonsson, B., Lengál, O., Trinh, C.Q., Vojnar, T.: Verification of heap manipulating programs with ordered data by extended forest automata. In: Automated Technology for Verification and Analysis - 11th International Symposium, ATVA 2013, Hanoi, Vietnam, October 15-18, 2013. Proceedings. (2013) 224–239

[2] Bouajjani, A., Habermehl, P., Rogalewicz, A., Vojnar, T.: Abstract regular tree model checking. Electr. Notes Theor. Comput. Sci. **149**(1) (2006) 37–48

[3] Calcagno, C., Distefano, D., O'Hearn, P.W., Yang, H.: Compositional shape analysis by means of bi-abduction. In: Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009. (2009) 289–300

[4] Chin, W., David, C., Nguyen, H.H., Qin, S.: Automated verification of shape, size and bag properties via user-defined predicates in separation logic. Sci. Comput. Program. **77**(9) (2012) 1006–1036

[5] Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms. 3rd edn. MIT press (2009)

[6] Ferrara, P., Müller, P., Novácek, M.: Automatic inference of heap properties exploiting value domains. In: Verification, Model Checking, and Abstract Interpretation - 16th International Conference, VMCAI 2015, Mumbai, India, January 12-14, 2015. Proceedings. (2015) 393–411

[7] Habermehl, P., Holík, L., Rogalewicz, A., Simácek, J., Vojnar, T.: Forest automata for verification of heap manipulation. In: Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings. (2011) 424–440

[8] Heinen, J., Jansen, C., Katoen, J.P., Noll, T.: Juggrnaut: Using graph grammars for abstracting unbound heap structures. In: Formal Methods in System Design. Volume 47., Springer US (2015) 159–203

[9] Holík, L., Lengál, O., Rogalewicz, A., Simácek, J., Vojnar, T.: Fully automated shape analysis based on forest automata. In: Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings. (2013) 740–755

A Basic Definitions

Commonly used variables and names.

Variable	Of Type	Description
c_Z	$\in VCnd$	A value condition
c_P	$\in PCnd$	A pointer condition
C, C_1, C_2	$\subseteq Cnd$	Subsets of conditions
c	$\in C_\alpha$	Condition that can be stored in an information function
x, y	$\in Ptr$	Common pointers variable names
v, u	$\in V$	Common vertex names
$x.s$	$\in Ptr$	Selector s of x
$x.z$	$\in Vls$	Value z of x
$\sim$	$\in Ops$ or $\in \mathcal{P}(Ops)$	with Ops:
Ops	$= \{\neq, =, <, >, \leq, \geq\}$	Any operator

Definition 29. From [8]. A heap configuration is a hypergraph $H = (V, E, att, lab, ext) \in HG_{\Sigma_N}$ with

$\Sigma_N = \Sigma \uplus N$, $\Sigma = Var_\Sigma \uplus Sel_\Sigma$,
$rk(Var_\Sigma) = 1$, $rk(Sel_\Sigma) = 2$,
$\forall x \in Var_\Sigma, \mid \{e \in E \mid lab(e) = x\} \mid \leq 1$,
$\forall v \in V, \forall s \in Sel_\Sigma, \mid \{e \in E \mid lab(e) = s, att(e)(1) = v\} \mid \leq 1$,
$ext = \varepsilon$, ε denotes the empty sequence.

Definition 30. $neg : VCnd \to VCnd$. Let $c_Z \in VCnd, z_1, z_2 \in Vls$.

$$neg[c_Z] := \begin{cases} z_1 \neq z_2 & , c_Z = (z_1 = z_2) \\ z_1 = z_2 & , c_Z = (z_1 \neq z_2) \\ z_1 > z_2 & , c_Z = (z_1 \leq z_2) \\ z_1 \geq z_2 & , c_Z = (z_1 < z_2) \\ z_1 < z_2 & , c_Z = (z_1 \geq z_2) \\ z_1 \leq z_2 & , c_Z = (z_1 > z_2) \end{cases}$$

Definition 31. Let $c \in C_\alpha, z_1, z_2 \in Vls$, $\sim \in Ops$, $e' \in E$ with $Ops = \{\neq, =, <, >, \leq, \geq\}$.

$$rev : (VCnd \cup \{(e.source.z_1 \sim e.target.z_2)\} \times E)$$
$$\rightarrow VCnd \cup \{(e.source.z_1 \sim e.target.z_2)\}.$$

$$rev[c, e'] := \begin{cases} e'.source.z_2 \ revr[\sim] \ e'.target.z_1 & , c = (e.source.z_1 \sim e.target.z_2) \\ z_2 \ revr[\sim] \ z_1 & , c = (z_1 \sim z_2) \end{cases}$$

with $revr : Ops \rightarrow Ops$

$$revr[\sim] = \begin{cases} = & , \sim \text{ is } = \\ \neq & , \sim \text{ is } \neq \\ \geq & , \sim \text{ is } \leq \\ > & , \sim \text{ is } < \\ \leq & , \sim \text{ is } \geq \\ < & , \sim \text{ is } > \end{cases}$$

The following code is a naïve implementation of the selection sort algorithm using the programming language defined in chapter 2.
It uses the following variables:

possorted:	Marks position until which the list is sorted
min:	Marks element with currently assumed local minimal value
possearch:	Iterator
tmp:	Temporal pointer

```
selection_sort(list){

 new(possorted);
 new(min);
 new(possearch);

 possorted := list.head;
 min := list.head;
 possearch := list.head;

 while(possorted ≠ list.tail){

   while(possearch ≠ list.tail)
     possearch := possearch.next;
     if(possearch.data < min.data){
       min := possearch;
     }
   }

   new(tmp);
   tmp.next := min.next;
   tmp.prev := min.prev;
   min.next := possorted.next;
   min.prev := possorted.prev;
   possorted.next := tmp.next;
   possorted.prev := tmp.prev;

   possorted := possorted.next;
   possearch := possorted;
   min := possorted;
 }
}
```