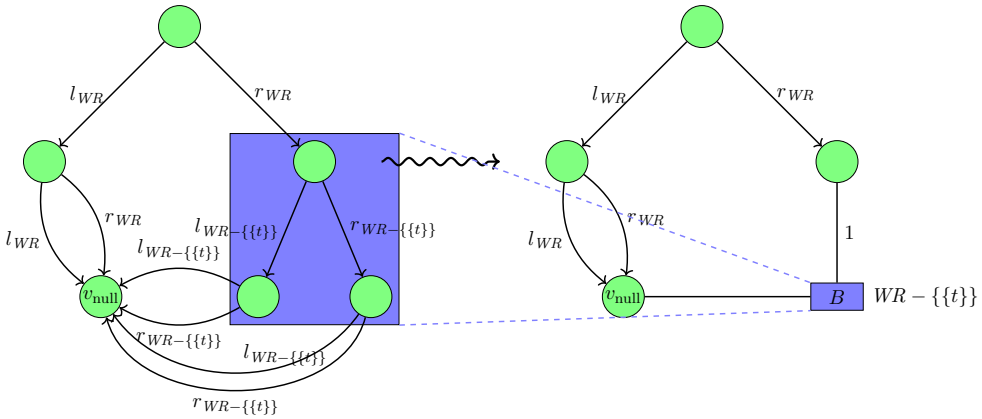
Rheinische-Westfälische Technische Hochschule Aachen
Lehrstuhl für Informatik 2
Software Modeling and Verification
Prof. Dr. Ir. Joost-Pieter Katoen

August 20, 2015

Bachelor Thesis

Thread-Modular Analysis of Heap-Manipulating Programs



Author:	Christoph Welzel
First Reviewer:	apl. Prof. Dr. Thomas Noll
Second Reviewer:	Prof. Dr. Ir. Joost Pieter Katoen
Advisor:	Christina Jansen

Abstract

This paper presents a graph-based modelling of the structure of concurrent heap manipulating programs. It introduces a programming language that allows parallel execution in the means of fork and join. The semantics of the programming language is presented in terms of hypergraphs transformations. The main results are the abstraction of the heap structure by hyperedge replacement grammars which is proven to be a correct overapproximation and that the presented analysis can prove data race free executions of programs.

Selbstständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit eigenständig angefertigt habe und keine anderen Quellen oder Hilfsmittel als die angegebenen benutzt habe. Alle wörtlichen oder sinngemäßen Zitate sind als solche kenntlich gemacht.

(Christoph Welzel)

Aachen, den 15. August 2015

Contents

1 Introduction

Nowadays, software relies on the use of pointers. In order to make use of unbounded data structures these are modelled within the dynamically allocateable part of the memory, the *heap*. As previous research has shown it is a viable approach to analyse pointer programs by representing the structure of the heap as *hypergraph* [7, 10, 11]. This analysis focuses on different problems in the use of pointer in software, most notably the dereferencation of null pointers and memory leaks [7, 6]. Furthermore in order to obtain a finite representation of the unbounded data structures *hyperedge replacement grammars* are used for abstraction. Those hyperedge replacement grammars allow to represent parts of the heap that can grow unboundedly in a finite manner but also to concretise abstracted parts on demand. This modelling technique is already used as basis for computational analysis by the tool Juggrnaut [6]. By now these techniques operate mainly on sequential execution of programs. Another possible approach on analysing pointer programs which relies on formulating structural properties of the heap in logical formulae is *separation logic* [14]. Because in separation logic the heap is separated in different parts which are reasoned about individually this approach has proven well suited for analysis of concurrent programs [3, 4, 15]. This work is a contribution in widening the analysis technique of a graph-based representation to parallel executing programs. By approaching parallelism new challenges arise, most importantly *data races*. Data races are situations where multiple processes access data at the same time and the result of these actions depend on each other. If data races occur all possible interleavings of the executions have to be examined which causes the possible states that have to be analysed to grow uncontrollably large. This is often referred to as *state space explosion* e.g. in [2, pp. 77-80]. In order to avoid state space explosion it is possible to prove the absence of data races which resolves the dependency of the actions and allows to reduce the number of states that have to be examined. In separation logic this is approached by incorporating permission accounting into the formulae that describe the heap states. This permission accounting is a way to distinguish read and write access and can be used to prove data race freedom. As suggested e.g. in [10] and [11] this approach can be examined for graph-based representation of the heap. This is the main contribution of this paper: incorporating permissions into the hypergraphs that are used to model the state of the heap and introducing a mechanism of parallelism by adding fork and join statements to the examined programming language. Where fork starts new processes and join synchronises other processes by waiting for their termination. This is explored in [1] with some formal effort. The goal of the presented analysis is among other things to provide a more intuitive approach by graph-based representation of the heap. For the fork and join statemant appropiate semantics in the sense of graph transformations are defined. Central results of this work are the proof that the used abstraction via hyperedge replacement grammars support the incorporation of permissions and that the analysed states ensure data race freedom. For this result it is generally assumed that the behaviour of forked processes is determined by given contracts, similar to those that are automatically generated in [11].

2 Preliminaries

For a set S, S^* are all finite sequences of elements from S. For such a sequence $s \in S^*$ the number of elements is denoted by $|s|$ and the set of elements of the sequence is denoted by $\wr s \wr$. Furthermore $s(i)$ refers to the i-th element of the sequence s. For a tupel $A = (B, C, D, ...)$ for the single components $B_A, C_A, D_A, ...$ is used as long as A is clear from the context. Additionally, a function restricted to a subset A of its domain is denoted by $f \restriction A$. Functions defined for a set V are lifted to the sequence V^* and the power set $\mathbb{P}(V)$ by elementwise application. For a finite set A the function $enum_A : |A| \to A$ returns an enumeration of the elements of A. And finally, for a function f $f[e \mapsto v]$describes a function that mirrors f except for e where it maps to v, i.e. for an $e \in dom(f)$ and $v \in img(f)$

$$f[e \mapsto v](x) = \begin{cases} v & \text{if } x = e \\ f(x) & \text{otherwise} \end{cases}$$

3 Programming Language

In the following a very basic programming language is discussed, which allows simple *heap* manipulating operations. It is similar to the programming language introduced in [6]. For the manipulation of the heap it is possible to create objects by allocation. All these objects contain a set of *selectors* which refer to other objects or a distinguishable empty state, **null**. Especially there is just one set of selectors (Sel) and all objects are universally typed in the sense that the selectors of all objects are Sel. Additionally every program has a set of variables (Var) which refer to objects within the heap. This heap is implicitly *garbage collected*, i.e. objects that can not be reached from one of the variables are deleted. Initially all variables and selectors of newly allocated objects refer to the **null** value. One of the main features of the presented programming language is the possibility of parallel execution. Therefore the *fork* and the *join* statement are introduced. On the one hand the fork statement allows to spawn a new process and on the other hand the join statement synchronises a process by blocking until the process that is joined terminates. Every process operates on variables which are exclusively accessible to itself. But the objects of the heap can be shared between different processes therefore at the forking of a new process it is provided with a set of parameters which refer to objects of the heap. These parameters are the initial values for some variables of the newly forked process. Because a program might start multiple processes there is a set of *process identifier* (Var_{process}) which are used to identify forked processes. These process identifier are also unique for every executing process (as variables are) and the join statement is called with one of theses identifiers to determine which process the program synchronises with by waiting for its termination. Forked processes execute defined programs concurrently and every process must only be joined once. Let $Proc$ denote a finite set of programs that can be forked.

3.1 Syntax

The syntax of the examined programming language can be given by a context-free grammar starting in $\langle S \rangle$ as follows where $x, x_1, \ldots, x_n \in Var, s \in Sel, t, t_1, t_2 \in Var_{\text{process}}$ and $m \in Proc$ is another program:

$$\langle P \rangle \quad ::= \textbf{null} \mid x \mid x.s$$

$$\langle C \rangle \quad ::= \langle P \rangle == \langle P \rangle \mid \langle P \rangle \; != \langle P \rangle \mid \langle C \rangle \; \&\& \; \langle C \rangle \mid \langle C \rangle \parallel \langle C \rangle$$

$$\begin{aligned} \langle S \rangle \quad ::= {} & x{=}\langle P \rangle \mid x.s = \langle P \rangle \mid \textbf{new}(x) \mid \textbf{while}(\langle C \rangle)\ \textbf{do}\ \langle S \rangle\ \textbf{done} \\ \mid {} & \textbf{if}\ (\langle C \rangle)\ \textbf{then}\ \langle S \rangle\ \textbf{else}\ \langle S \rangle\ \textbf{fi} \mid \textbf{skip} \mid \langle S \rangle;\langle S \rangle \\ \mid {} & t{=}\textbf{fork}(m(x_1, \ldots, x_n)) \mid \text{join}(t) \mid t_1 = t_2 \end{aligned}$$

3.2 Example

In the following a small program is examined which traverses a binary tree. Therefore there is one variable called *curr* which is assumed to initially start on a node of the binary

tree. Every node has two selectors *left*, *right* identifying the left and right subtree. The program executes by forking another process that executes a different program *traverse* on every node initially provided with the right subtree(the program traverse is assumed to not change the subtree it operates on) and itself moving down the left subtree. If there is no left subtree the most recently forked process is synchronised with. At last the *left* selector of *curr* is set to the right subtree of *curr* and the *right* selector is set to the designated **null** value:

Listing 1: An example program

```
1  while(curr.left != null) do
2          tmp = curr.right;
3          t = fork(traverse(tmp));
4          curr = curr.left;
5  done;
6  tmp = curr.right;
7  t = fork(traverse(tmp));
8  tmp = null;
9  join(t);
10 curr.left = curr.right;
11 curr.right = null;
```

Thus for this program the *context*(the sets Sel, Var_{process}, Var, $Proc$) is as follows:

- $Proc = \{traverse\}$
- $Sel = \{left, right\}$
- $Var = \{curr, tmp\}$
- $Var_{\text{process}} = \{t\}$

Furthermore consider the graphical representation of the execution of this program that can be seen in Figure 1. Some interesting observations can be made:

1. Because of the structure of the binary tree the newly forked process can only access the subtrees starting from the parameter object (marked in green).

2. After starting a new process the previously started process cannot be referred to anymore since the process identifier is assigned to another process. Thus there is no information about any process operating on the yellow marked areas of the binary tree.

3. The last process is joined again, thus it can be assured that it has terminated after the join statement. This means especially that no process operates on the subtree which is moved from the right hand side to the left hand side (as indicated by the absence of a yellow mark).

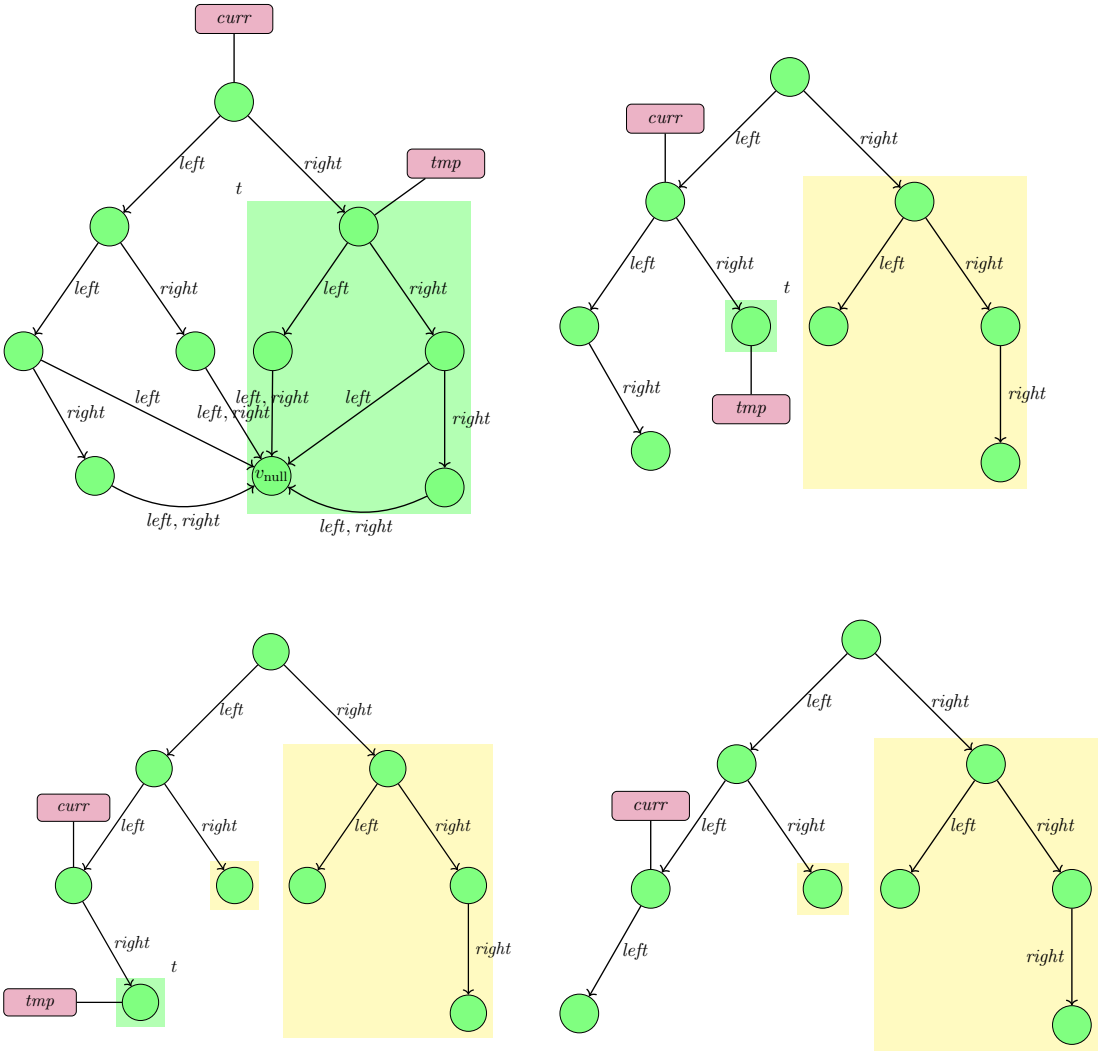


Figure 1: A graphical representation of the execution of the example program for one possible heap state, note that aside from the first step selectors which refer to the **null** value are omitted.

The in the following presented analysis of programs relies heavily on representing the current state as graph. But in order to approach parallel execution of multiple processes which share heap objects and their selectors this representation is furtherly enriched. In order to account which parts of the heap are accessible by multiple processes it is locally logged for the selectors to which processes these are visible.

4 Permission Model

As stated above it is locally accounted for which process a selector is accessible. The main goal of the analysis is to state that the execution of a program is *data race free*. Where data race refers to the situation that two processes access a shared value at the same time and at least one of the processes alters the value. This implies a dependency of both actions with each other. For a proper analysis all interleaving have to be examined (see [2, pp. 77-80] for details). *Data race freedom* describes the absence of data races which can be guaranteed as long the following two conditions are met[1]:

- If two or more parallel processes access a value it must not be altered.
- If one process alters a value it must have at this time exclusive access to it.

The accounting is realised in form of *permissions*. Permissions are used to model access restrictions in order to ensure (1) that write access is exclusive and (2) that if other processes might access the same value at the same time both are restricted to read access. Permissions were already introduced for other modelling techniques for heaps like *separation logic* [3]. There different models for permissions are examined like *fractional permissions* or *counting permissions*. All these models share the property that there is one distinguishable permission representing exclusive access on a value. From this access ticket for one value multiple other access tickets on this value can be derived which then causes all these accesses to become read only. Furthermore derived access tickets can be returned until there exists only one access tickets which then again grants permission to alter the given value. For fractional permission it is possible that once derived tickets can be split indefinitly on demand to generate as many access tickets as needed. On the other hand for counting permissions there is one distinguishable permission per value from which further access tickets are derived. Every derived access ticket is accounted for and can be returned to this distinguishable permission. In the following course an approach is chosen which is closer connected with [8] and combines both presented concepts of *fractional permissions* and *counting permissions* and formalises ideas from [12].

Fundamentally two different *permissions* are distinguished:

- *WR* - denotes that this value can be altered.
- *RD* - denotes a shared access, which restricts to reading only.

These permissions represent access rights where *WR* is the distinguishable value representing exclusive access. From both permissions it is possible to derive further access tickets which are locally accounted and deleted if these tickets are returnded. Therefore the different forked processes are uniquely identified and this identification is used to account derived access tickets. Additionally it is possible to transfer *WR* tickets completely. For the identification of forked processes the programming language relies on

[1] this can be easily derived from the explanation of safe concurrency in [3]

values from Var_{process}. As it can be seen from the grammar of the programming language it is possible that multiple process identifiers from Var_{process} can refer to the same actual process as it can be easily seen by the following example:

```
1 t1 = fork(traverse(tmp));
2 t2 = t1;
3 join(t2)
```

Here t_1 and t_2 refer to the same forked process. The join statement takes any of these possible process identifier to synchronise with the corresponding process. In order to account permissions given to processes *tokens* are introduced. Tokens are sets of Var_{process} which refer to the same process. Thus, for every token T always holds that $T \subseteq Var_{\text{process}}$ and the set of all possible tokens is simply the powerset of the process identifiers ($\mathbb{P}(Var_{\text{process}})$). For every shared value the permission that it held initially is kept as well as a set of tokens that represent all derived tickets to the process that can access this value. Recall that variables and process identifiers are considered process exclusive and only heap objects and selectors can be shared. Furthermore the presented analysis focuses on structural analysis of the heap and therefore only consideres selectors as data of heap objects.

In order to formalises these ideas *permission expressions (PEs)* are introduced in the following. These consist of a so called *BasePerm* which represents the *permission* that was originally granted and a set of tokens for derived permissions.

For an example consider Figure 2, which mirrors the previous example from Figure 1 but added permissions to the selectors. Imagine all selectors are initially equipped with a WR permission. For the part of the heap that the newly forked process may access the token $\{t\}$ is added to the permissions. As already observed in the second iteration of the example from Figure 1 the information if a process still operates on a certain part of the heap is lost. Especially it is impossible to retrieve any information of the status of the once forked process because the synchronisation mechanism (join statement) demands an identifier referring to the process which no longer exists. In this case the derived access tickets are considered permanently lost. In order to model this for permissions two additional *BasePerm* besides WR, RD are introduced: WR^* and RD^* which indicate that the initially granted permission can never be fully returned since the information about any parallel executing processes is lost. The following definition formalises these observations:

Definition 4.1 (Permission Expression)**.** *A* permission expression *is a term of the form*

$$BasePerm - PermSet$$

where $BasePerm \in \{WR, WR^*, RD, RD^*\}$. *And* $PermSet \subseteq \mathbb{P}(Var_{process})$.

For convenience an empty *PermSet* and a *PermSet* after RD^* or WR^* as *BasePerm* might be omitted if it is clear from the context that this information is not needed. Sometimes for a *BasePerm* ρ and a *PermSet* Φ and $T \in \mathbb{P}(Var_{\text{process}})$ $\rho - \Phi - T$ is

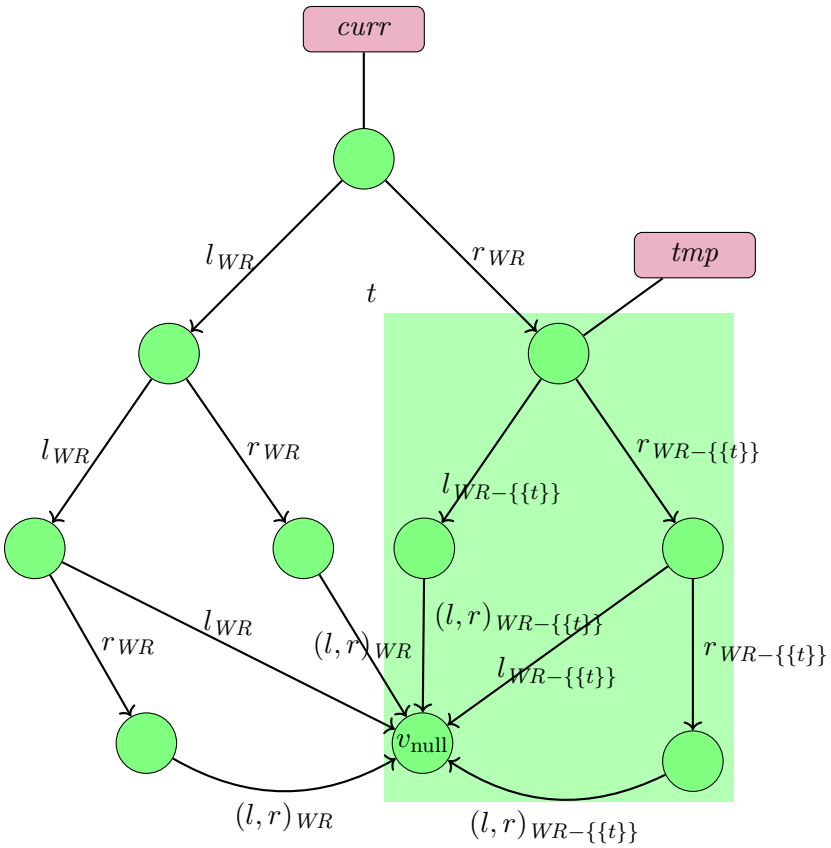


Figure 2: State of the first iteration of the heap shown in Figure 1 with additional permissions on every selector, empty *PermSets* are omitted, *left* and *right* are shortened to l and r respectively. For two selectors that are represented by one edge the permission for those is written only once.

written for $\rho - \Phi \cup \{T\}$. In addition a *PE* with an empty *PermSet* is called *simple*. The set of all *PEs* over Var_{process} is denoted by $PES_{Var_{\text{process}}}$. Note that for a finite set Var_{process} $PES_{Var_{\text{process}}}$ is finite as well.

For now it was always considered that the process only reads the part of the heap it has access to. But as it is introduced later on there also will be a way to specify for a process that it might alter certain selectors. In order to model that this part of the heap must not be accessed anymore because the access ticket is completely submitted to the forked process. This is achieved by substituting this part of the heap that might be altered by a placeholder. This placeholder "hides" parts of the heap so it can not be accessed. Upon joining this process again the placeholder is additionally used to identifiy where the part of the heap to which the access ticket is transferred to is returned. For the program from Listing 1 consider that the forked process might demand write access on the right subtree of its parameter. Then the substitution of this subtree by a placeholder can be seen in Figure 3. Note that the placeholder is connected to the node which can still be accessed from the original heap but is also the node that the parameter of *traverse* pointed to and to the **null** node.

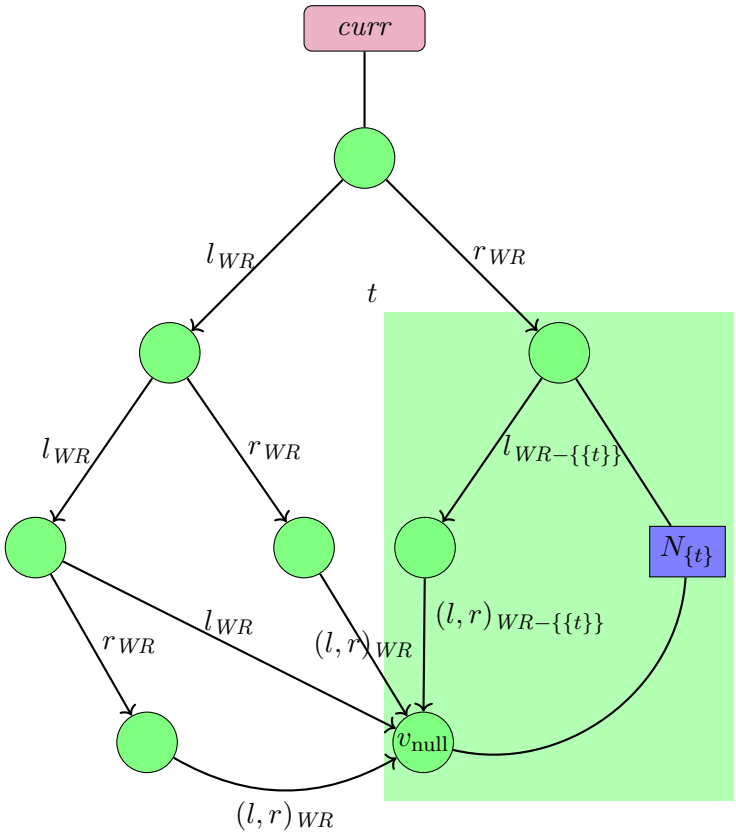


Figure 3: State of the first iteration of the heap shown in figure 1 with permissions as well as partially substitution with a placeholder.

5 Hypergraphs as Heap Representation

In order to formalise the intuitive graph representation from the examples above *hypergraphs (HGs)* are introduced in the following. This follows the general approach presented in [13, 10, 7, 6]. Intuitively, an *HG* is the same as a normal graph, it consists of a set of nodes and edges, but an edge in an *HG* is called hyperedge and can connect arbitrarily many nodes instead of only two. Furthermore the used *HGs* will be labeled, thus there is a function *lab* which assigns a label to every hyperedge. The possible labels come from a set Σ. Then an *HG* can be formally defined as follows:

Definition 5.1 (Hypergraph)**.** *Let Σ be a set of labels and let V be a finite set of nodes, E a finite set of edges. $con : E \to V^*$ maps every edge to the sequence of connected nodes, $lab : E \to \Sigma$ labels every edge. Further let $ext \in V^*$ denote a possibly empty sequence (ε) of nodes and $perm : E \to PES_{Var_{process}}$. An* HG *$H$ is a tupel*

$$H = (V, E, con, lab, ext, perm)$$

The set of all hyperedges with the labels Σ and process identifier Var_{process} is denoted by $HG_{\Sigma}^{Var_{\text{process}}}$. Two *HGs* are called *isomorphic* if they are equivalent modulo renaming of edges and nodes. Isomorphic *HGs* are not distinguished in the following.

In order to represent states of the heap the objects in the heap will be represented as nodes. Variables are represented by hyperedges that are only connected to the node that represents the value of the variable. Such a hyperedge that is connected to only one node is called *unary*. Selectors are modelled as binary hyperedges between nodes, which are interpreted as directed from the first node of the connection sequence to the second one. Furthermore parts that are given away to other processes are substituted by hyperedges. For the hyperedges that represent parts of the heap for which access tickets are transferred a set of labels is introduced as

$$\mathbb{T} = \{N_T \mid T \in \mathbb{P}(Var_{\text{process}})\}$$

States of the heap can now be represented as $H \in HG_{\Sigma}^{Var_{\text{process}}}$ where $\Sigma = Var \uplus Sel \uplus \mathbb{T}$, but of course not all those *HG* H represent valid heaps because it is e.g. possible that every object might have multiple outgoing edges labeled with the same selector. Thus some restrictions are enforced in order to represent valid states of a heap which are called *heap configurations (HCs)*, namely:

- There is exactly one distinguishable node v_{null} representing the **null** value. v_{null} has no outgoing selectors.
- There is exactly one unary hyperedge for every variable.
- Edges labeled by a selector are binary hyperedges.
- For every node there is at most one outgoing edge per selector.

Note that every object is universally typed. Thus, all objects have initially an outgoing edge for every selector. But of course some of these edges can be "hidden" within a hyperedge representing parts of the heap for which *WR* permissions are transferred to another process. As it is possible that the reference to the process that operates exclusively on the selectors is lost, these selectors can permanently be removed from the heap state. This leads to the following formal definition for *HCs*:

Definition 5.2 (Heap Configuration)**.** *A* hypergraph $H = (V, E, con, lab, ext, perm) \in HG_{\Sigma}^{Var_{process}}$ *is called a* heap configuration *if*

- $\Sigma = Var \uplus Sel \uplus \mathbb{T}$
- $v_{null} \in V$ *and* $\nexists e \in E.lab_H(e) \in Sel \wedge con_H(e)(1) = v_{null}$
- $|con_H(e)| = 1$ *for all* $e \in E$ *with* $lab_H(e) \in Var$
- $|con_H(e)| = 2$ *for all* $e \in E$ *with* $lab_H(e) \in Sel$
- $\forall v \in Var. \exists e \in E. lab_H(e) = v$
- $\forall v \in Var, e, e' \in E. v = lab_H(e) = lab_H(e') \rightarrow e = e'$
- $\forall s \in Sel, e, e' \in E. con_H(e)(1) = con_H(e')(1) \wedge s = lab_H(e) = lab_H(e') \rightarrow e = e'$

The set of all *HC* over Σ and Var_{process} is denoted by $HC_{\Sigma}^{Var_{\text{process}}}$.

Furthermore the following notion is defined for $\rho \in PES_{Var_{\text{process}}}$ which collects all edges of the HG with permission ρ:

$$E_{\rho}^{H} = \{e \in E_H \mid perm_H(e) = \rho\}$$

6 Concrete Semantics

Because *HGs* are used to represent the current state of the heap the semantics of the programming language are modelled in terms of graph transformations. Therefore the different statements of a program are connected with changes of the shape of the hypergraph representing the state of the heap. In the following the necessary graph transformations which are based on presented transformations of [6] for modelling the semantics are presented. This is done in two separate parts where firstly structural changes are discussed and secondly transformations that regard the *PEs* of the graph.

6.1 Graph Transformations

The structural graph transformations are presented in the following:

- $H[+v]$ adds a new fresh node v to H, which is of universally type, thus there are edges attached to for all *selectors*
- $H[\backslash E']$ removes the set of edges E' from H
- $H[x \hookrightarrow v]$ connects one x labeled edge with v, where $x \in Var$
- $H[u \overset{s}{\hookrightarrow} v]$ connects one s labeled edge from the node u to the node v
- $H[+_\rho n \rightrightarrows v_1 \cdots v_m]$ adds a fresh n labeled edge with *permission* ρ connected to $v_1 \cdots v_m$

Formally these transformations are defined as follows, where v is a new node and $e, e_1, \ldots$ are new edges:

$H[+v]$	$=$	$(V_H \uplus \{v\}, E_H \uplus \{e_1, \ldots, e_{\lvert Sel\rvert}\},$ $con_H \cup \{e_1 \mapsto vv_{\text{null}}, \ldots, e_{\lvert Sel\rvert} \mapsto vv_{\text{null}}\},$ $lab_H \cup \{e_1 \mapsto enum_{Sel}(1), \ldots, e_{\lvert Sel\rvert} \mapsto enum_{Sel}(\lvert Sel\rvert)\},$ $ext_H, perm_H \cup \{e_1 \mapsto WR, \ldots, e_{\lvert Sel\rvert} \mapsto WR\})$	
$H[\backslash E']$	$=$	$(V_H, E_H \setminus E', con_H \upharpoonright (E \setminus E'), lab_H \upharpoonright (E \setminus E'),$ $ext_H, perm_H \upharpoonright (E \setminus E'))$	
$H[x \hookrightarrow u]$	$=$	$(V_H, E_H,$ $con_H[e \mapsto u],$ $lab_H, ext_H, perm_H)$	for one e with $lab_H(e) = x$, if such an e exists
$H[u \overset{s}{\hookrightarrow} u']$	$=$	$(V_H, E_H,$ $con_H[e \mapsto uu'],$ $lab_H, ext_H, perm_H)$	if there is e with $lab_H(e) = s$ and $con_H(e)(1) = u$ and $u, u' \in V_H$
$H[+_\rho n \rightrightarrows v_1 \cdots v_n]$	$=$	$(V_H, E_H \uplus \{e\}, con_H \cup \{e \mapsto v_1 \cdots v_n\},$ $lab_H \cup \{e \mapsto n\}, ext_H, perm_H \cup \{e \mapsto \rho\})$	

The few more transformation which focus on various aspects of the *permissions* are also introduced below:

- $H[\downarrow t]$ denotes that $t \in Var_{\text{process}}$ does not longer identify a certain process (for example by reassigning t)
- $H[\leftarrow T]$ denotes that the *permissions* for $T \in \mathbb{P}(Var_{\text{process}})$ are returned
- $H[t = t']$ is used to describe that for $t, t' \in Var_{\text{process}}$ t now identifies the same process as t'
- $H[E - T]$ denotes that for the edges in E the token T is added to the *PermSet* of the *permissions*

The following definitions formalise these intuitions:

Definition 6.1 (Dropped Thread Variable)**.** *For an* HC H, $t \in Var_{process}$ *let* $E_{remove} \subseteq E_H$ *be all edges* e *such that* $lab_H(e) = N_{\{t\}}$*. Then* $H[\downarrow t]$ *denotes the HC with:*

$$H[\downarrow t] = (V_H, \underbrace{E_H \setminus E_{remove}}_{=:E'}, con_H \restriction E', lab' \restriction E', perm' \restriction E')$$

where lab' *mirrors* lab_H *except of all edges* e *such that* $lab_H = N_T \in \mathbb{T}$ *where holds* $lab'(e) = N_{(T \setminus \{t\})}$*. And for all edges* e *with* $perm_H(e) = \rho_e - \Phi_e$ $perm'(e)$ *is defined as follows:*

$$perm'(e) = \begin{cases} WR^* - \Phi_e \setminus \{\{t\}\} & \text{if } \rho_e \in \{WR, WR^*\} \text{ and } \{t\} \in \Phi_e \\ RD^* - \Phi_e \setminus \{\{t\}\} & \text{if } \rho_e \in \{RD, RD^*\} \text{ and } \{t\} \in \Phi_e \\ \rho_e - \{T \setminus \{t\} \mid T \in \Phi_e\} & \text{otherwise} \end{cases}$$

Note that all edges labeled with $N_{\{t\}}$ are removed. This is because these edges are the placeholder for the part of the heap that the process t referred to might alter. Thus, in order to ensure data race freedom this part of the heap has to be exclusively accessible by this process. Because the process which operates on this part of the heap can never be rejoined (the token is a singleton set only containing t, thus only t refers to this process and is reassigned) all these nodes and edges must never be accessed. To achieve this the placeholder is removed which removes this part of the heap permanently. Furthermore all tokens containing t are updated in order to indicate that t no longer identifies the same process as the other process identifier of this token. For those *PermSets* that contain the token $\{t\}$ the *BasePerm* is "starred" in order to indicate that, since there is no further reference to the process the corresponding access token can never be returned. And this implies that the *BasePerm* can never be fully recovered.

Definition 6.2 (Returned Token)**.** *For* $H \in HC_\Sigma^{Var_{process}}$, $T \in \mathbb{P}(Var_{process})$

$$H[\leftarrow T] = (V_H, E_H, con_H, lab_H, ext_H, perm')$$

with $perm'(e) = \rho - \Phi \setminus \{T\}$ *where* $perm_H(e) = \rho - \Phi$*.*

Note that simply the token T is removed from all the *PermSets*. The return of parts of the heap that the process identified by T (or rather all $t \in T$) had exclusive access on is dealt with separately.

Definition 6.3 (Thread Variable Assignment)**.** *For $H \in HC_\Sigma^{Var_{process}}$ and $t, t' \in Var_{process}$ let $\Phi_{t'} = \{T \in \Phi \mid t' \in T\}$ denote the set of all tokens in a PermSet Φ which contain the process identifier t'. Then*

$$H[t = t'] = (V_{H[\downarrow t]}, E_{H[\downarrow t]}, con_{H[\downarrow t]}, lab', ext_{H[\downarrow t]}, perm')$$

where lab' mirrors $lab_{H[\downarrow t]}$ except for all $e \in E_{H[\downarrow t]}$ with $lab_{H[\downarrow t]}(e) = N_T$ and $t' \in T$ where $lab'(e) = N_{T \cup \{t\}}$. And $perm'(e) = \rho - (\Phi \setminus \Phi_{t'} \cup \{T \cup \{t\} \mid T \in \Phi_{t'}\})$ if $perm_{H[\downarrow t]}(e) = \rho - \Phi$.

Note that for the assignment the reassigned identifier is previously dropped since its original value will be lost after the assignment. And the set of tokens for all *PermSets* that do not contain the process identifier t' are simply preserved. To thoses tokens that contain t' as process identifier t is added since t identifies from now on the same process as t'.

Definition 6.4 (Add Token to Edge)**.** *For $H \in HC_\Sigma^{Var_{process}}$, $T \in \mathbb{P}(Var_{process})$ and $E \subseteq E_H$*

$$H[E - T] = (V_H, E_H, con_H, lab_H, ext_H, perm')$$

where

$$perm'(e) = \begin{cases} perm_H(e) - T & \text{if } e \in E \\ perm_H(e) & \text{otherwise} \end{cases}$$

The token T is simply added to all *PermSets* of edges in E.

6.2 Semantics

In the following the semantics of programs are defined by graph transformations. Therefore some relations over nodes in *HGs* are defined as follows:

- $v \overset{s}{\hookrightarrow}_\rho v'$ states that the node v is connected to the node v' by an edge labeled with $s \in Sel$ and the permission ρ
- $x \hookrightarrow_\rho v$ denotes that there is an edge labeled with $x \in Var$ and the permission ρ which is connected to the node v

For both relations the permission might be omitted to indicate that one permission ρ exists such that the relation holds.

At first the semantics of pointer expressions are examined. This is, evaluating a pointer expression under an *HC* $H \in HC_\Sigma$ ($[\![\cdot]\!]_H$). Intuitively, this means that variables

are identified with the node the edge labeled with the variable is attached to, the **null** value is identified with v_{null} and dereferencing a selector of the object corresponds to following the outgoing edge from a node labeled by this selector. There are two possible errors: firstly, dereferencing anything from the **null** value and secondly, accessing a selector which is not there. The second case indicates that a selector is accessed although another process demands exclusive access. Because of the universal type of every node the only possible way a selector is absent is because it is hidden behind a placeholder for another process or removed because the reference to the corresponding process is lost. Such an error is indicated by returning an error symbol $\perp$. Formally this leads to the following:

$$
\begin{aligned}
[\![\mathbf{null}]\!]_H &= v_{\text{null}} \\
[\![x]\!]_H &= v \qquad \text{with } x \hookrightarrow v \\
[\![x.s]\!] &= \begin{cases} v & \text{if } [\![x]\!]_H \overset{s}{\hookrightarrow} v \\ \perp & \text{otherwise} \end{cases}
\end{aligned}
$$

Note that for an *HC* these semantics are well defined since every node has maximal one outgoing edge for every selector and for every variable exists exactly one edge labeled with it.

The semantics of conditions are evaluated very intuitively, but where $\perp$ propagates strictly, i.e. if there is one expression yielding $\perp$ the whole evaluation becomes $\perp$. Therefore for pointer expressions p_1, p_2, and conditions c_1, c_2:

$$
[\![p_1 == p_2]\!]_H = \begin{cases} \perp & \text{if } [\![p_1]\!]_H = \perp \text{ or } [\![p_2]\!]_H = \perp \\ \mathit{true} & \text{if } [\![p_1]\!]_H = [\![p_2]\!]_H \\ \mathit{false} & \text{if } [\![p_1]\!]_H \neq [\![p_2]\!]_H \end{cases}
$$

$$
[\![p_1 \mathrel{!}= p_2]\!]_H = \begin{cases} \perp & \text{if } [\![p_1]\!]_H = \perp \text{ or } [\![p_2]\!]_H = \perp \\ \mathit{false} & \text{if } [\![p_1]\!]_H = [\![p_2]\!]_H \\ \mathit{true} & \text{if } [\![p_1]\!]_H \neq [\![p_2]\!]_H \end{cases}
$$

$$
[\![c_1 \mathbin{\&\&} c_2]\!]_H = \begin{cases} \perp & \text{if } [\![c_1]\!]_H = \perp \text{ or } [\![c_2]\!]_H = \perp \\ [\![c_1]\!]_H \wedge [\![c_2]\!]_H & \text{otherwise} \end{cases}
$$

$$
[\![c_1 \parallel c_2]\!]_H = \begin{cases} \perp & \text{if } [\![c_1]\!]_H = \perp \text{ or } [\![c_2]\!]_H = \perp \\ [\![c_1]\!]_H \vee [\![c_2]\!]_H & \text{otherwise} \end{cases}
$$

This gives the basis to define the semantics for the given programming language using a *transition system*:

Definition 6.5 (Semantics of the Programming Language)**.** *The* semantics *of the individual statements are modelled by a* transition relation *of the form* $\rhd \subseteq (\mathbb{S} \cup \{\varepsilon\} \times HC_\Sigma)^2$ *where* $\mathbb{S}$ *denotes the language derived by the grammar in section 3.1.*

The semantics for the *assignment, allocation, loop, conditional and skip statement* can be given by the following rules, where $v \hookrightarrow_\rho -$ abbreviates $\exists v'.v \hookrightarrow_\rho v'$ and $v \overset{s}{\hookrightarrow}_\rho -$ is expanded to $\exists v'.v \overset{s}{\hookrightarrow}_\rho v'$:

$$\frac{[\![P]\!]_H \neq \bot \qquad [\![x]\!]_H \hookrightarrow_{WR} -}{(x = P, H) \rhd (\varepsilon, H[[\![x]\!]_H \hookrightarrow [\![P]\!]_H])}$$

$$\frac{[\![P]\!]_H \neq \bot \qquad [\![x]\!]_H \overset{s}{\hookrightarrow}_{WR} -}{(x.s = P, H) \rhd (\varepsilon, H[[\![x]\!]_H \overset{s}{\hookrightarrow} [\![P]\!]_H])}$$

$$\frac{[\![C]\!]_H = \text{true}}{(\text{while}(C)\ \text{do}\ S\ \text{done}, H) \rhd (S; \text{while}(C)\ \text{do}\ S\ \text{done}, H)}$$

$$\frac{[\![C]\!]_H = \text{false}}{(\text{while}(C)\ \text{do}\ S\ \text{done}, H) \rhd (\varepsilon, H)}$$

$$\frac{[\![C]\!]_H = \text{true}}{(\text{if}(C)\ \text{then}\ S_1\ \text{else}\ S_2\ \text{fi}, H) \rhd (S_1, H)}$$

$$\frac{[\![C]\!]_H = \text{false}}{(\text{if}(C)\ \text{then}\ S_1\ \text{else}\ S_2\ \text{fi}, H) \rhd (S_2, H)}$$

$$\frac{[\![x]\!]_H \hookrightarrow_{WR} -}{(\text{new}(x), H) \rhd (\varepsilon, H[+v][x \hookrightarrow v])}$$

$$\frac{}{(\text{skip}, H) \rhd (\varepsilon, H)}$$

$$\frac{(S_1, H) \rhd (S_1', H')}{(S_1; S_2, H) \rhd (S_1'; S_2, H')}$$

$$\frac{t, t' \in Var_{\text{process}}}{(t = t', H) \rhd (\varepsilon, H[t = t'])}$$

$$\frac{}{(\varepsilon; S, H) \rhd (S, H)}$$

Note that for operations that change an edge a WR permission on that edge is expected. Furthermore every statement that does not have a fitting transition rule yields explicitly $\bot$. Since fork and join are more complex they are examined in more detail below.

6.3 Fork

As described previously fork creates a process and provides it with parameters.This is similar to procedure calls modelled in [11]. In the following two types of parameters are distinguished:

- *Formal parameters*, which belong to the newly created process
- *Actual parameters*, which belong to the process executing the fork statement

In the call of a fork statement the actual parameters are provided to the newly created process and are identified with formal parameters. Thus the new process initially only knows the nodes identified by the actual parameters and can now access other nodes via the different selectors. The subgraph that can be accessed in this way is called *reachable*. The concept of *reachability* will be later on introduced formally but partially relies on the abstraction that is introduced in Section 7.4 and is therefore here only given as the set $reach(v_1, \ldots, v_n)$ which denotes all edges that are considered to be reachable from the nodes $v_1, \ldots, v_n$.[2]

Reconsider the example from Listing 1 and Figure 1 on page 6 where it was observed that *forking* a new process and assigning it to a process identifier potentially causes to lose the possibility to refer to the process the process identifier identified before. In order to model this loss of information the semantics rely on a corresponding graph transformation ($H[\downarrow t]$). Another aspect of the fork statement is that as demonstrated in Figure 3 it is possible that a write access ticket is completely transferred to the *forked* process. As mentioned in Section 1 the behaviour of processes and especially which parts of the heap the process might alter is defined in a *contract*. These contracts consist of a *precondition*, the set of edges which the process might alter and a *postcondition*. The precondition is the representation of the context from which the process is forked from. The postcondition is the representation of the *heap* that resulted from the execution of the process started from an initial heap state. This initial heap state is obtained by a *transformation* of the precondition which is examined later on. Firstly, as mentioned in Section 3 it is expected that every process has its own variables as well as process identifier. W. l. o. g. the heap state for every forked process can be described as an HC over the set Var'_{process} and $\Sigma' = Sel \uplus \mathbb{T}' \uplus Var'$ with renaming of variables and process identifier as well as limiting the used variables and process identifier to subsets of Var' and Var'_{process}. Especially it holds that $Var \cap Var' = \emptyset$ and $Var_{\text{process}} \cap Var'_{\text{process}} = \emptyset$. Secondly, the precondition is examined as a representation which is isomorph to the reachable subgraph. Therefore it is assumed that the precondition is a HG from $HG_{\Sigma}^{Var_{\text{process}}}$. On the other hand the postcondition is the result of the computation of the forked process and therefore represented as HG from $HG_{\Sigma'}^{Var'_{\text{process}}}$. And this computation of the forked process ensures that only edges from the alternable set are altered. These coherences are illustrated graphical in Figure 4. For every program the

[2] actually the set $reach(v_1, \ldots, v_n)$ is a superset of the actual reachable edges, but because it covers at least all actual reachable edges the soundness of the model for the semantics is ensured

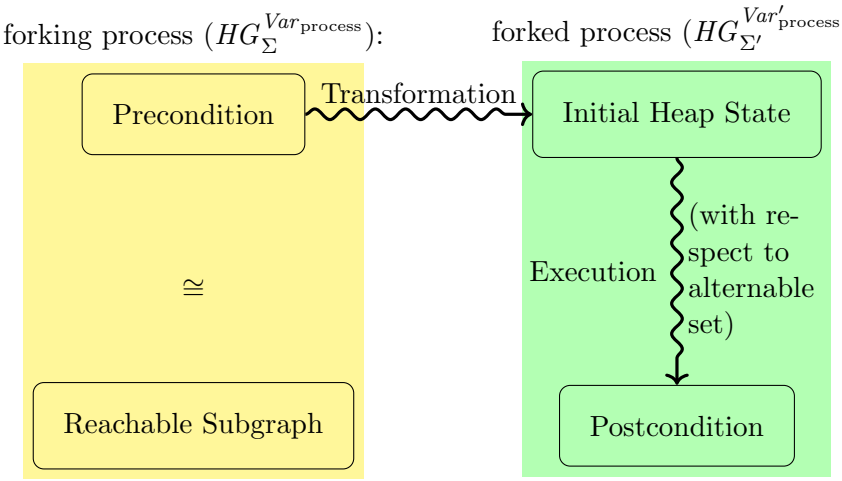


Figure 4: Graphical representation of the coherences between precondition, initial heap state, reachable subgraph and postcondition

set of contracts is formally described as a function

$$Cont\colon Proc \to \mathbb{P}(HG_{\Sigma}^{Var_{\text{process}}} \times \mathbb{P}(E) \times HG_{\Sigma'}^{Var'_{\text{process}}})$$

where for every program m and every contract $C = (P_C, E_C, Q_C) \in Cont(m)$ holds that $E \subseteq E_{WR}^{P_C}$, thus the set of alterable edges (E_C) is a subset of the edges with write access of the precondition.

The necessary graph transformations associated with the fork statement are examined in the following. Let H be an HG and $C = (P_C, E, Q_C) \in Cont(m)$. Let $ap_1, \ldots, ap_n$ be the nodes that are identified by the actual parameter of the fork statement. This means that for the statement

```
1  t = fork(m(x_1, ..., x_n));
```

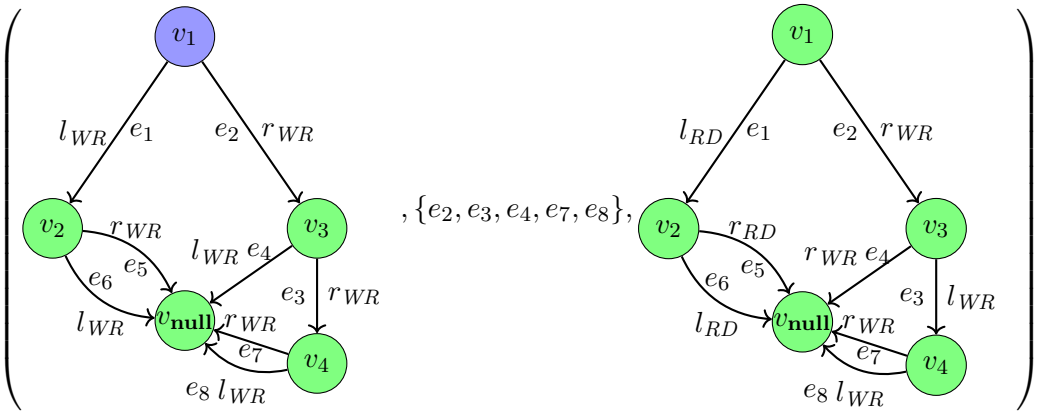
with the current heap representation H it holds that $ap_i = [\![x_i]\!]_H$ for all $1 \leq i \leq n$. Thus the set of reachable edges from the actual parameter is defined as $reach(ap_1, \ldots, ap_n)$. The actual precondition for the newly forked process is the current *heap* representation reduced to the subgraph which is induced by the reachable part. This subgraph is defined as follows:

Definition 6.6 (Edge Induced Subgraph)**.** *Let H be a* HG *and $E' \subseteq E_H$ a set of edges. Then the* subgraph induced by E' *is defined as*

$$H \restriction E' = (\bigcup_{e \in E'} \{\!|con_H(e)|\!\}, E', lab \restriction E', con \restriction E', ext \restriction E', perm \restriction E')$$

To find the fitting contract C for the fork statement it is necessary that the reachable subgraph forms the precondition of C and the actual parameters agree with the formal parameters. Let $fp_1, \ldots, fp_n$ denote the nodes of the formal parameters within the precondition of C. Then the reachable subgraph is structurally identical to the precondition and the actual parameters agree with the formal parameters. Let R denote the reachable subgraph of a fork statement and P denote the fitting precondition. Then R and P are isomorphic and especially this isomorphism matches formal and actual parameter. Let $R \ {}_{ap}\!\cong_{fp} P$ denote the existence of such an isomorphism.

Once again reconsider the example in Figure 2 and let

$C =$ 

be a possible contract such that $C \in \mathit{Cont}(\mathit{traverse})$ and v_1 is the only formal parameter. Here l and r abbreviate *left* and *right* respectively. Note that C fulfills that the alterable set is a subset of the edges with WR permission of the precondition. Secondly, in the postcondition only edges have been altered which were part of the alterable set (namely e_3, e_4). Thirdly, forking by C leads to the replacement seen in Figure 3 because the induced subgraph of $\{e_2, e_3, e_4, e_7, e_8\}$ is replaced. As it can be seen in Figure 3 the replacement is connected to the node that is identified with v_1 and v_{null}. This is because v_1 represents an object which is part of the main process as well as the newly forked one. Such nodes are called *border* nodes and are defined as those nodes that are connected to edges of the alterable set and to edges that are not part of the alterable set. Additionally, the placeholder in Figure 3 is also connected to v_{null}, which is generally assumed for all placeholder, thus the heap representation of every process can refer to the same distinguishable **null** value. The formal definition of the set of border nodes is closely connected to the definition of reachability and is therefore likewise postponed to Section 7.5. For now the set of border nodes for a contract C and a heap representation H is denoted by $\mathit{border}_H(C)$[3]. These border nodes represent the connection points between the heap representation of the main process and the part of the heap of the forked process that can be altered. For joining the process later on it is crucial to identify these nodes in the postcondition. This is achieved by using the mechanism of *external nodes* as introduced for *hyperedge replacement* in Section 6.4. With these considerations

[3]just like the reachable set this set is potentially a superset of the actual border nodes

in mind the transformation of the heap representation that models the fork statement can be described. Let therefore

- $t = \textbf{fork}(m(x_1, \ldots, x_n))$;
- with H as the current heap representation,
- $ap_i = [\![x_i]\!]_H$ for all $1 \leq i \leq n$,
- $(P_C, E_C, Q_C) \in \mathit{Cont}(m)$,
- $P_C \ {}_{ap}{\cong}_{fp} \ (H \upharpoonright \mathit{reach}(ap_1, \ldots, ap_n))$,
- $t \in \mathit{Var}_{\text{process}}$,
- $b_C(H)$ denote the set of border nodes,
- $\mathit{enum}_{b_C(H)} = \mathit{enum}_{b_C(H)}(1) \ldots \mathit{enum}_{b_C(H)}(|b_C(H)|)$ is an arbitrary sequence of the border nodes

W. l. o. g. it is assumed that P_C shares nodes and edges with H which can easily achieved by renaming the edges and nodes of P_C according to the existing isomorphism. Then the graph transformation can be given as follows:

$$H' = \overbrace{\underbrace{\underbrace{H[\downarrow t]}_{1}[\backslash E_C]}_{2}[+_{WR} N_{\{t\}} \rightrightarrows \mathit{enum}_{b_C(H)}][(E_P \setminus E_C) - \{t\}]}^{3}}^{4}$$

where the different steps state the following:

(1) the process identifier used for the newly forked process is freed

(2) the edges that the forked process demands write access on are removed from the heap representation

(3) the formerly removed edges are replaced with an hyperedge

(4) to all edges that can be read by the newly forked process the appropiate token is added

Formally this process is therefore described by the following transition rule:

$$\frac{(P_C, E_C, Q_C) \in \mathit{Cont}(m) \qquad P_C \ {}_{ap}{\cong}_{fp} \ (H \upharpoonright \mathit{reach}([\![p_1]\!]_H, \ldots, [\![p_n]\!]_H))}{(t = \text{fork}(m(p_1, \ldots, p_n)), H) \rhd (\varepsilon, H')}$$

For the *join* statement some formalisms of *hyperedge replacement grammars*, namely *hyperedge replacement* is used and therefore the definition is postponed and hyperedge replacement is introduced before.

6.4 Hyperedge Replacement

In the following the concept of *hyperedge replacement (HR)* is presented. *HR* is a graph transformations which replaces single hyperedges by more complex graphs [16, p. 104]. As the fork statement replaces parts of the heap by a single hyperedge in order to "hide" this subgraph, rendering it unavailable for the main process. Indeed the "hidden" subgraph still exists within the heap representation of the forked process and can especially be returned by a *join* statement. Therefore a way to re-integrate this subgraph is discussed in the following. A possible way to present rules for substituting hyperedges by hypergraphs are *production rules* which have the form $p\colon X \to H$. Here X is a label and H a hypergraph and this production rule p can be applied to a hypergraph by removing a X-labeled edge and replacing it by the hypergraph H. For better understanding the production rule in Figure 5 is considered in the following. On the right hand side of this production rule the nodes marked by 1 and 2 are *external nodes* of the hypergraph and their annotation is their position in the sequence *ext*. For the hyperedges that are labeled with B the numbers denote the position of the nodes in the sequence of connected nodes. In order to replace for instance such a B-labeled hyperedge in an *HG* the

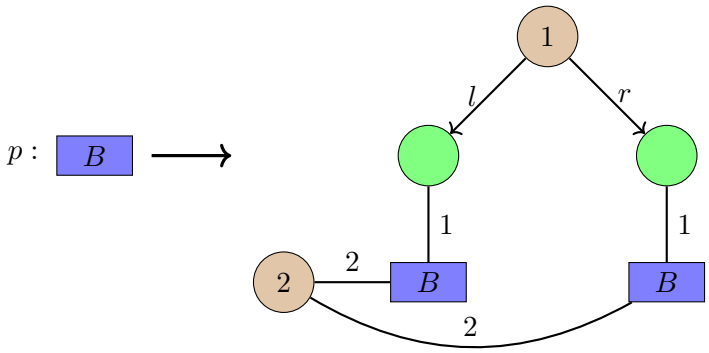


Figure 5: A possible production (permissions are omitted)

external nodes of the right hand side of the production rule are identified with the nodes connected to the hyperedge. A simple example (see Figure 6) illustrates such a replacement. Note that the external nodes and the connected nodes that are identified had the same position within the respective sequences. Such a replacement of an hyperedge is called a *derivation* and is formally described as follows:

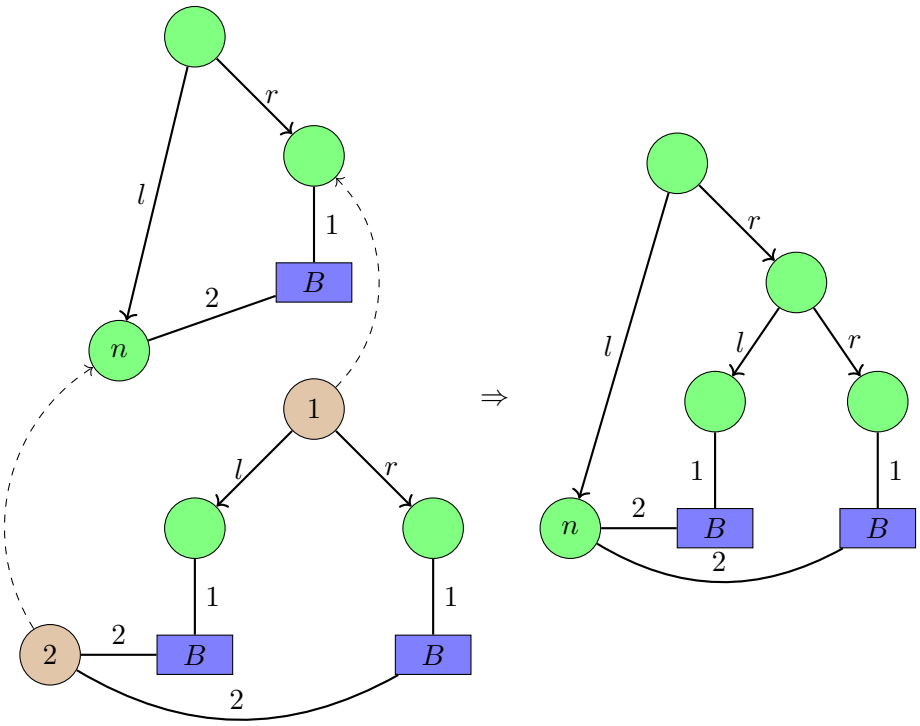


Figure 6: Illustration of an hyperedge replacement

Definition 6.7 (Hyperedge Replacement)**.** *Let p be a production rule of the form $p\colon X \to K$, $H \in HG_\Sigma$ a* HG *with an edge $e \in E_H$ with $lab_H(e) = X$ and $|ext_K| = |con_H(e)|$. Then $H' = H[e/K]$ is an* HG *with:*

- $V_{H'} = V_H \cup (V_K \setminus \wr ext_K \wr)$
- $E_{H'} = (E_H \setminus \{e\}) \cup E_K$
- $con_{H'} = con_H \upharpoonright (E_H \setminus \{e\}) \cup ((id_K[ext_K(1) \mapsto con_H(e)(1)] \ldots [ext_K(|ext_K|) \mapsto con_H(e)(|ext_K|)]) \circ con_K)$
- $lab_{H'} = lab_H \upharpoonright (E_H \setminus \{e\}) \cup lab_K$
- $ext_{H'} = ext_H$
- $perm_{H'} = perm_H \upharpoonright (E_H \setminus \{e\}) \cup perm_K$

An *HG* H' is derived from an *HG* H by a production rule $p\colon X \to K$ (denoted by $H \overset{p}{\Rightarrow} H'$) if H' is isomorphic to $H[e/K]$ for any edge e labeled with X. Furthermore $\Rightarrow^*$ is the transitive closure of this relation.

6.5 Join

Having introduced the mechanism of hyperedge replacement the postponed semantics of the join statement can now be addressed. Therefore it is distinguished between the *joining* process which is the one actually executing the join statement and the *joined* process which is the one that terminated and now returns its permissions. Returning this permissions can be done in two separate steps: Firstly, the permissions for the shared edges, i.e. those edges which *BasePerm* is either *RD* or RD^* for the joined process, are returned. Since these edges are still present in the heap representation of the joining process returning those permissions is a matter of mutating the *PE* of the heap representation of the joining process. Secondly, the edges which were allowed to be altered are returned, i.e. edges which *BasePerm* is either *WR* or WR^*. This is done by replacing the hyperedge in the joining process which represents this part of the heap by its corresponding *HG* of the joined process. This imposes a further restriction to the contracts which have to ensure that the external nodes of the postcondition of the forked process are the same as the border nodes of the placeholder in the heap representation of the forking process. To do so, the sequence of external nodes in the precondition is used to encode which nodes are border nodes and the isomorphism ${}_{ap}{\cong}_{fp}$ is restricted to map the border nodes in the heap representation of the forking process to the external nodes in the precondition. Following it is formally described how the initial heap state for the forked process is obtained from the contract $C = (P_C, E_C, Q_C)$:[4] Let $ap_1, \ldots, ap_n$ denote the nodes that mark the actual parameters and $fp_1, \ldots, fp_1$ denote the names

[4] which refers to the *transformation* in Figure 4

of the formal paramters they are matched to (especially $fp_i \in \Sigma'$ for all $1 \leq i \leq n$). Furthermore let for all $s \in Sel$ $s' \in Sel'$ denote the corresponding selector used to describe the heap states of forked processes, then $I \in HG_{\Sigma'}^{Var'_{\text{process}}}$ is the initial heap state for the forked process with:

- $V_I = \{v^I_{v'} \mid v' \in V_{P_C}\}$
- $E_I = \{e^I_{e'} \mid e' \in E_{P_C}\} \cup \{e_{fp_i} \mid 1 \leq i \leq n\} \cup \{e_v \mid v \in (Var' \setminus \{fp_i \mid 1 \leq i \leq n\})\}$
- $con_I = \{v' \mapsto v^I_{v'} \mid v' \in V_{P_C}\} \circ con_{P_C} \cup \{e_{fp_i} \mapsto ap_i \mid 1 \leq i \leq n\} \cup \{e_v \mapsto v_{\text{null}} \mid v \in (Var' \setminus \{fp_i \mid 1 \leq i \leq n\})\}$
- $lab_I = \{e_{fp_i} \mapsto fp_i \mid 1 \leq i \leq n\} \cup \{e \mapsto s' \mid lab_{P_C}(e) = s\} \cup \{e_v \mapsto v \mid v \in (Var' \setminus \{fp_i \mid 1 \leq i \leq n\})\}$
- $ext_I = \{v^I_{v'} \mapsto v' \mid v' \in V_{P_C}\}(enum_{border_{P_C}}(1) \ldots enum_{border_{P_C}}(|border_{P_C}(C)|))$
- $perm_I = \{e \mapsto WR \mid e \in E_C\} \cup \{e \mapsto RD \mid e \in E_{P_C} \setminus E_C\} \cup \{e_v \mapsto WR \mid v \in (Var' \setminus \{fp_i \mid 1 \leq i \leq n\})$

Note that actually only the selectors are renamed and variables introduced for the formal parameters that point to the nodes the actual parameters pointed to in the fork statement, also all other variables of Var' are introduced and initially point to v_{null}. Furthermore the external nodes of the initial heap state are used to mark the border nodes in order to ensure that the "*WR*-part" of the postcondition agrees on the shared objects when it is joined. At last, all permissions in the initial heap state are simple and grant those access tickets that the contract demands. From this initial heap state the execution of the forked program starts.

For both steps of returning permissions for the join statement it is important to determine if all permissions are returned which were derived when the process was forked. This is not the case for all edges which *PEs* are not *simple* or which *BasePerm* is "starred" (thus WR^* or RD^*). These permissions are unrecoverably lost and this propagates strictly to the permissions of the heap representation of the joining process.

For returning the read permissions it is avoided to match the edges from the heap representation of the joining and joined process since it imposes further difficulties later on. Therefore the minimal permission is returned: for all edges of the joining process from which permissions were derived to the joined process the *BasePerm* is "starred" if there is *any* edge in the postcondition for which the permission cannot be completely returned.

Returning the write permissions is done by simply applying a production rule to the hyperedge of the joining process that represents the borrowed part. Let now $C = (P_C, E_C, Q_C) \in Cont(m)$ be the contract by which the joined process was forked (note that this contract has to be the same as for the fork statement that created the process which is re-joined now, in order to save the contract for matching it from fork to join statement it can be e. g. attached to the placeholder) and T_t the set of all aliases of

the process identifier t that is used in the join statement. Further the following few definitions are used for convenience:

$$Q'_1 = Q_C[\downarrow enum_{Var'_{\text{thread}}}(1)] \ldots [\downarrow Var'_{\text{thread}}(|Var'_{\text{thread}}|)]$$

Note that for Q'_1 all PE are *simple* and every not yet returned permission caused the *BasePerm* to be "starred".

$$Q'_2 = Q'_1[\backslash (E^{Q'_1}_{RD} \cup E^{Q'_1}_{RD^*})]$$

Q'_2 represents the state where the heap representation of the joined process is reduced to the edges of the alterable set. Now two transition rules are used to model the semantics of the join statement where the first one describes the case with lost read permissions and the second one describes the case where all read permissions can be completely returned:

$$\frac{E^{Q'_1}_{RD*} \neq \emptyset \qquad H[\downarrow T_t] \xrightarrow{N_{T_t} \to Q'_2} H'}{(\text{join}(t), H) \rhd (\varepsilon, H')}$$

Note that $H[\downarrow T_t]$ is used which successively drops the process identifier in the token T_t

$$\text{“}H[\downarrow T_t] = H[\downarrow enum_{T_t}(1)] \ldots [\downarrow enum_{T_t}(|T_t|)]\text{”}$$

but preserves N_{T_t} in order to replace it by the given production rule.

$$\frac{E^{H'_1}_{RD*} = \emptyset \qquad H[\leftarrow T_t] \xrightarrow{N_{T_t} \to Q'_2} H'}{(\text{join}(t), H) \rhd (\varepsilon, H')}$$

This concludes the modelling of the semantics of the presented programming language and in the following it is dealt with abstraction techniques to approach potentially unbounded structures.

7 Concretisation & Abstraction

Making use of dynamic allocation and deallocation can lead to heap structures of unbounded size. In order to present these structures in a finite manner *HR* can be used in order to concretise and abstract heap states by applying production rules for concretisation and by applying production rules backwards for abstraction. Therefore introduce a set of production rules in order to abstract parts of the heap representation. In order to illustrate the following steps the data structure of binary trees is examined along the definitions[5]. Reconsider the structure in the example from Figure 1 and furthermore the *HR* from Figure 6. When the node marked by n is interpreted as v_{null} and the production rule from Figure 5 is applied repetitively arbitrary large binary trees can be constructed. But it is already noteable that repetitive application of this production rule always leads to further B-labeled edges. The following formal introduction of the concept of abstraction follows mostly [6, 10, 13].

In order to distinguish the hyperedges that are used for the abstract representation of a subgraph from the hyperedges that represent placeholders, selectors or variables a new set of so called *nonterminals* N is introduced. For the above considered example of binary trees this set would be $N = \{B\}$. In the following $\Sigma_N = \Sigma \uplus N$ denotes the set of labels for *HGs* that are partially concrete and partially abstract. Furthermore these *nonterminals* are expected to connect always the same amount of nodes. This amount is called the rank of the nonterminal. Therefore a function $rk\colon N \to \mathbb{N}$ is given and for every *HG* H it holds that if $lab_H(e) \in N$ then $|con_H(e)| = rk(lab_H(e))$, the rank of B for example is 2. The set of all these partially abstract *HGs* is denoted by $HG_{\Sigma_N}^{Var_{\text{process}}}$ and the set of *HGs* that furthermore satisfy the restrictions of *HCs* is denoted by $HC_{\Sigma_N}^{Var_{\text{process}}}$. Note that for now the permissions were neglected but are actually incorporated as follows: a production rule $p\colon X \to H$ with $X \in N$ is annotated by a permission ρ (p_ρ) such that for $p_\rho\colon X \to H$ holds that $img(perm_H) = \{\rho\}$ (thus all edges in H hold permission ρ). Furthermore this production rule must only be applied to an X-labeled edge if this edge holds the permission ρ.

7.1 Hyperedge Replacement Grammar

To apply abstraction and concretisation a set of production rules is used. Such a set of production rules is called a *hyperedge replacement grammar (HRG)*. Formally defined as follows:

Definition 7.1 (Hyperedge Replacement Grammar)**.** *A* hyperedge replacement grammar *over an alphabet* Σ_N *and* $Var_{process}$ *is a finite set of production rules of the form* $p_\rho\colon X \to G$ *where* $X \in N$, $\rho \in PES_{Var_{process}}$, $G \in HG_{\Sigma_N}$ *and* $rk(X) = |ext_G|$.

$HRG_{\Sigma_N}^{Var_{\text{process}}}$ denotes the set of all *HRGs* over Σ_N and Var_{process}. Especially, in this paper the concepts of structural abstraction and permission accounting are independently

[5] this example reassembles the *hyperedge replacement grammar* from [6, p. 21]

examined, this means in order to separate those concepts the following definition is used which states that every production rule is available for every permission:

Definition 7.2 (Fully Permissive Grammar)**.** $G \in HRG_{\Sigma_N}^{Var_{process}}$ *is called* fully permissive *if the following holds: Every production rule* $p\colon X \to H$ *in* G *exists for every* $\rho \in PES_{Var_{process}}$.

The set of all *fully permissive HRGs (fpHRGs)* is denoted by $fpHRG_{\Sigma_N}^{Var_{\text{process}}}$.

In the following some definitions regarding *fpHRGs* are introduced. These intuitions and their respective definitions are examined in their structural concepts only since the incorporation of permissions is straightforward:

- For $G \in fpHRG_{\Sigma_N}^{Var_{\text{process}}}$ $G^X = \{X' \to H \in G \mid X' = X\}$ denotes the set of all production rules for a nonterminal X.
- For a production rule $p\colon X \to H$ $lhs(p)$ is used for X and respectively $rhs(p)$ for the right hand side H. *rhs* and *lhs* are lifted to sets of *production rules* by elementwise application.
- The *handle* of a nonterminal is introduced which is the *HG* that contains one edge labeled by the nonterminal and nodes connected to this nonterminal. Formally:

  Definition 7.3 (Handle)**.** *For a* fpHRG $G \in fpHRG_{\Sigma_N}^{Var_{process}}$ *and a nonterminal* $X \in N$ *the* handle *of* X *is the* HG $X^\bullet \in HG_{\Sigma_N}$ *with:*
  - $V_{X^\bullet} = \{v_1, \ldots, v_{rk(X)}\}$
  - $E_{X^\bullet} = \{e\}$
  - $con_{X^\bullet} = \{e \mapsto v_1 \cdots v_{rk(X)}\}$
  - $lab_{X^\bullet} = \{e \mapsto X\}$
  - $ext_{X^\bullet} = \epsilon$
  - $perm_{X^\bullet} = \{e \mapsto \rho\}$

- For hyperedges labeled with nonterminals the position on which a node is attached on is called a *tentacle*. A tentacle is just a label and the position a node is attached to an edge of this label.

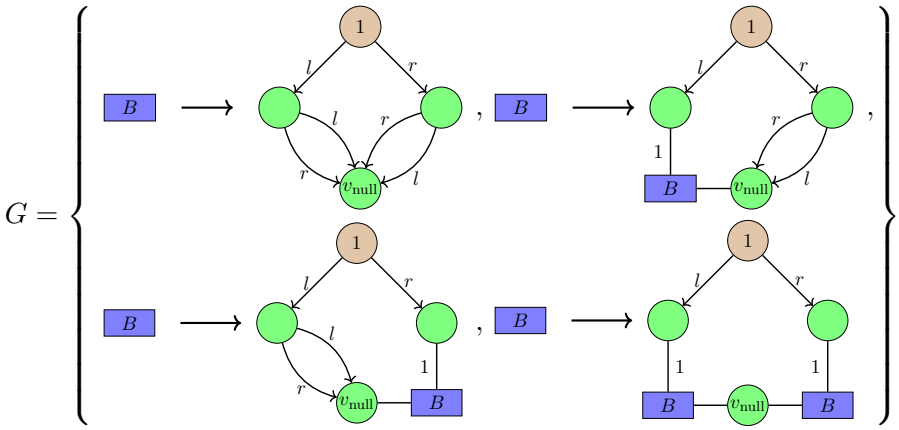
  Definition 7.4 (Tentacle)**.** *A* tentacle *is a pair* (X, i) *where* $X \in N$ *and* $i \in \{1, \ldots rk(X)\}$.

- For hyperedges labeled with nonterminals the notion of tentacles can be extended in order to describe how two nodes are connected by this hyperedge. Such a connection is called a *bridge* and reassembles two tentacles of the same nonterminal.

Definition 7.5 (Bridge)**.** *A* bridge *is a triple* (i, X, j) *where* $X \in N$ *and* $1 \leq i, j \leq rk(X)$.

Let $br(S)$ denote the set of all bridges for the set of ranked labels S.

Consider the following possible *fpHRG* G which describes the abstraction of binary trees:



Note that these production rules are present for every permission, since it is a fully permissive *HRG* and that from $H \in HG_{\Sigma_N}^{Var_{\text{process}}}$ it can be possible to derive multiple *HGs* since there might be more than one production rule for any nonterminal. Consider therefore the *HG* of Figure 6 and the *fpHRG* G from above. It is easily to see that multiple different *HGs* can be derived. The set of all these deriveable *HGs* which no longer contain a nonterminal (like the terminal words of a string grammar) is called the language of H and defined as follows:

Definition 7.6 (Language of an HG)**.** *For* $G \in fpHRG_{\Sigma_N}^{Var_{process}}$ *and* $H \in HG_{\Sigma_N}$

$$L_G(H) = \{K \in HG_\Sigma \mid H \Rightarrow^* K\}$$

is the language *from the* HG H *(under the* fpHRG G*).*

7.2 Properties

In the following some properties of *HRGs* are discussed. The presented properties as well as the presented results for *fpHRGs* follow from [13]. Actually the design of the definitions are built around those in [13] to focus on the compatibility of adding permissions to the already established results. At first it is discussed which selectors a nonterminal actually abstracts. By considering the example in Figure 6 it can easily be seen, that at the different connected nodes of the B-labeled edge different selectors are abstracted. At the node on the tentacle $(B, 1)$ both selectors l and r are "hidden" in the abstraction of the hyperedge whereas the node on tentacle $(B, 2)$ identifies v_{null} which must not have any outgoing selectors. This illustrates that different selectors can be abstracted at

different tentacles. *Typedness* is introduced to ensure that application production rules reveals the same selectors for every tentacle. The function *type* maps from tentacles to the sets of the selectors this tentacle abstracts. In the *fpHRG* from page 31 it holds that $type(B, 1) = \{l, r\}$. The formal definition is as follows:

Definition 7.7 (Typedness)**.** *For $G \in fpHRG_{\Sigma_N}^{Var_{process}}$ a nonterminal $X \in N$ is called typed if for all tentacles (X, i) there is a set $type(X, i) \subseteq Sel$ such that for all $H \in L_G(X^\bullet)$ where $v_i^\bullet$ is the node of the tentacle (X, i) in $X^\bullet$ it holds:*

$$type(X, i) = lab_H(out_H(v_i^\bullet))$$

Note that $out_H(v) = \{e \in E_H \mid con_H(1) = v \wedge lab_H(e) \in Sel\}$ is the set of all outgoing selectors from the object represented by v. Since different tentacles might have different types but the definition of *HC* enforces that every node is universally typed it is possible that the nodes of a handle violate this typing. Therefore the notion of handles is expanded to ensure that at least the initial nodes of the handle are universally typed:

Definition 7.8 (Typed Handle)**.** *For a typed nonterminal $X \in N$ let $Type = \bigcup_{1 \leq i \leq rk(X)} type(X, i)$ denote the set of all selectors of the nonterminal. The typed handle of X (denoted as X°) is defined as follows:*

- $V_{X^\circ} = \{v_1, \ldots, v_{rk(X)}, v_{null}\}$
- $E_{X^\circ} = \{e\} \uplus \{e_{s,i} \mid 1 \leq i \leq rk(X), s \in Type \setminus type(X, i)\}$
- $con_{X^\circ} = \{e \mapsto v_1 \cdots v_{rk(X)}\} \uplus \{e_{s,i} \mapsto v_i v_{null} \mid e_{s,i} \in E_{X^\circ}\}$
- $lab_{X^\circ} = \{e \mapsto X\} \uplus \{e_{s,i} \mapsto s \mid e_{s,i} \in E_{X^\circ}\}$
- $ext_{X^\circ} = \varepsilon$
- $perm_{X^\circ} = \{e \mapsto WR \mid e \in E_{X^\circ}\}$

Secondly, *productivity* is examined which simply states that at least one terminal *HG* can be derived from this nonterminal. Formally stated as:

Definition 7.9 (Productivity)**.** *For $G \in HRG_{\Sigma_N}^{Var_{process}}$ a nonterminal $X \in N$ is called* productive *if*

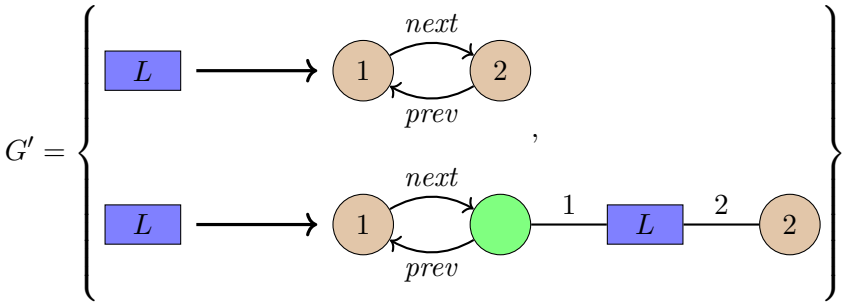
$$L_G(X^\bullet) \neq \emptyset$$

G is called productive *if all nonterminals in G are* productive.

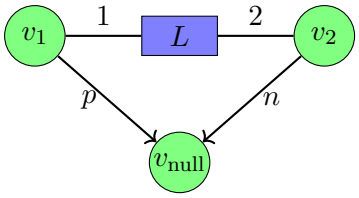
Following *increasingess* is introduced. The idea is that every right hand side of a production rule is strictly "bigger" than the left hand side. Where "bigger" refers to the number of edges, but also terminal *HGs* are considered "bigger". This ensures that applying succesively production rules in a backward fashion terminates, since every applied production rules reduces the size of the *HG*.

Definition 7.10 (Increasingess). *For* $G \in fpHRG_{\Sigma_N}^{Var_{process}}$ *a production rule* $p\colon X \to H \in G$ *is called* increasing *if* $H \in HG_{\Sigma}^{Var_{process}}$ *or* $H \in HG_{\Sigma_N}^{Var_{process}} \wedge |E_H| > 1$. *G is called* increasing *if all* $p \in G$ *are increasing.*

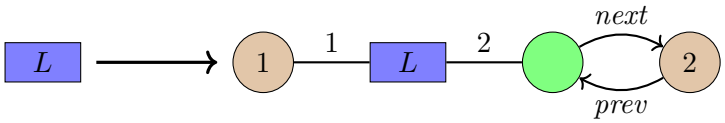
Fourthly, *local concretisability* means that for every tentacle there are production rules that reveal abstracted selectors but preserve the language of the graph. In order to illustrate the problem another *fpHRG* is introduced[6]:



This *fpHRG* describes doublely linked list of arbitrary length. Firstly, it holds that $type(L, 1) = \{next\}$ and $type(L, 2) = \{prev\}$. But, consider the typed handle of L, where n, p abbreviate *next*, *prev* respectively:



In order to reveal the abstracted selectors at the tentacle $(L, 2)$ only the first production rule can be applied. But this reduces the language to the singleton set of the right hand side of the first production rule. Repeatedly applying the second production rule does not reveal the abstracted selectors. Thus this grammar is not locally concretisable. Adding the production rule



establishes local concretisability because it is now possible to reveal abstracted selectors at tentacle $(L, 2)$ but the generated language (doubly linked lists of arbitrary length) is preserved.

[6] this example is featured in various contributions to this approach [7, 10] because of its illustrating quality

The formal definition of local concretisability requires some additional defintions, such as: $(X, i) \rightarrow_p (Y, j)$ denotes that for a production rule $p\colon X \rightarrow H$ there is an edge $e \in E_H$ such that $lab_H(e) = Y$ and $ext_H(i) = con_H(e)(j)$. This means that after applying the production rule p the tentacle (X, i) morphs (for the connected node) to a tentacle (Y, j). Furthermore let $\overline{G^X}$ denote the set of production rules in a *fpHRG* G for all nonterminals except of X, i.e. $\overline{G^X} = G \setminus G^X$. Then local concretisability can be defined as follows:

Definition 7.11 (Local Concretisability)**.** $G \in fpHRG_{\Sigma_N}^{Var_{process}}$ *is called* local concretisable *if for all nonterminals* X *there are sub-grammars* $G_1, \ldots, G_{rk(X)} \subseteq G$ *such that* $\forall i \in \{1, \ldots, rk(X)\}.L_{G_i^X \cup \overline{G^X}}(X^\bullet) = L_G(X^\bullet)$ *and* $\forall i \in \{1, \ldots, rk(X)\}.\forall p \in G_i^X.(X, i) \rightarrow_p (a, 1)$ *for all selectors* a *in type*(X, i).

The following definition gathers now all *fpHRG* for which the language of the typed handle contains only valid *HC*. This ensures that not external nodes of the right hand sides are properly typed and follow the properties of *HCs*.

Definition 7.12 (Data Structure Grammar)**.** *A* HRG $G \in fpHRG_{\Sigma_N}^{Var_{process}}$ *is called a* data structure grammar (DSG) *if it is typed and for all* nonterminals $X \in N$ *the language of its typed handle* X° *only contains valid* heap configurations *(*$L_G(X^\circ) \subseteq HC_\Sigma^{Var_{process}}$*) and for all inner nodes of right hand sides in* G^X *it holds that there is one outgoing edge for every selector in* $\bigcup_{1 \leq i \leq rk(X)} type(X, i)$.

Note that it is assumed that only the selectors exist which are gathered in the different types of the tentacles of the nonterminal. If more selectors exist they can be simply attached to every node and point to v_{null} to ensure that the *HGs* of the language are still valid *HCs*. Again let $DSG_{\Sigma_N}^{Var_{\text{process}}}$ denote the set of all *DSGs* over Σ_N and Var_{process}. Additionally, it is generally assumend that right hand sides of production rules do not contain any edges labeled with variables or placeholders (thus, $rhs(G) \subseteq HG_{\Sigma_N \setminus (Var \cup \mathbb{T})}^{Var_{\text{process}}}$). And therefore all inner nodes are properly typed which means that they have edges for all selectors that exist within the context of the nonterminal (i.e. the union of the types of all tentacles). In the following *Heap Abstraction Grammars* are defined which are *DSGs* that additionally provide the properties introduced above:

Definition 7.13 (Heap Abstraction Grammar)**.** $G \in DSG_{\Sigma_N}^{Var_{process}}$ *is called a* Heap Abstraction Grammar (HAG) *over* Σ_N *if* G *provides the following properties:*

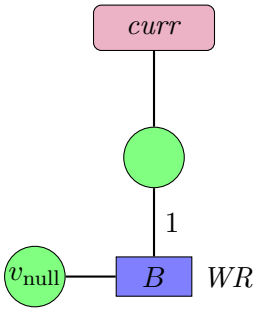
1. Productivity
2. Increasingness
3. Local Concretisability

This is no real restriction since the following theorem can be derived from the results of [13]:

Theorem 7.1. *For every* DSG *a* HAG *can be constructed which is equivalent regarding the language of* HGs*.*

7.3 Concretisation

The basic idea of abstraction is to represent multiple heaps in one *HG*. Therefore *HAGs* are used. Recall the initial programming example in Listing 1 where one possible execution is represented in Figure 1. Analysing executions on binary trees of arbitrary size is realised by representing these as one single B-labeled hyperedge as follows:



where the *fpHRG* G from page 31 (which is actually a *HAG*) is used to derive the concrete binary trees which are represented by the nonterminal B. By applying the different production rules of G all possible binary trees can be derived. Especially since all possible binary trees need to be analysed it is necessary that *every* production rule is used to concretise the nonterminal. This leads to four possible representations, namely the four right hand sides of the production rules in G (as it can be seen in Figure 7). For the three *HCs* that still contain nonterminals further concretisation steps have to be applied. But since the given programming language (see section 3.1) allows only dereferences of depth one (like $x.s$) it is possible to execute all statements (besides fork and join) on these partially abstract *HCs* just like they are executed on fully concrete *HCs*. These states of partially abstract *HCs* are called *admissible* and ensure that all actually from statements referenceable selectors are present in the *HC*. For the formal definition another kind of tentacle is examined, the *reduction tentacle*. Reduction tentacles are tentacles for which in every concretisation there is no outgoing edge for the identified node. This can be formally defined as follows:

Definition 7.14 (Reduction Tentacle)**.** *Let for the tentacle* (X, i) $v_i^\bullet \in V_{X\bullet}$ *be the node such that* $con_{X\bullet}(e)(i) = v_i^\bullet$ *for the only edge* $e \in E_{X\bullet}$ *with* $lab_{X\bullet}(e) = X$ *is called a* reduction tentacle *if*

$$\forall H \in L(X^\bullet).out(v_i^\bullet) = \emptyset$$

For typed nonterminals another characterisation of reduction tentacles is that the type of the tentacle (X, i) is empty ($type(X, i) = \emptyset$). As seen in [13] reduction tentacles can be determined by a syntactical analysis of the used production rules. Admissibility can

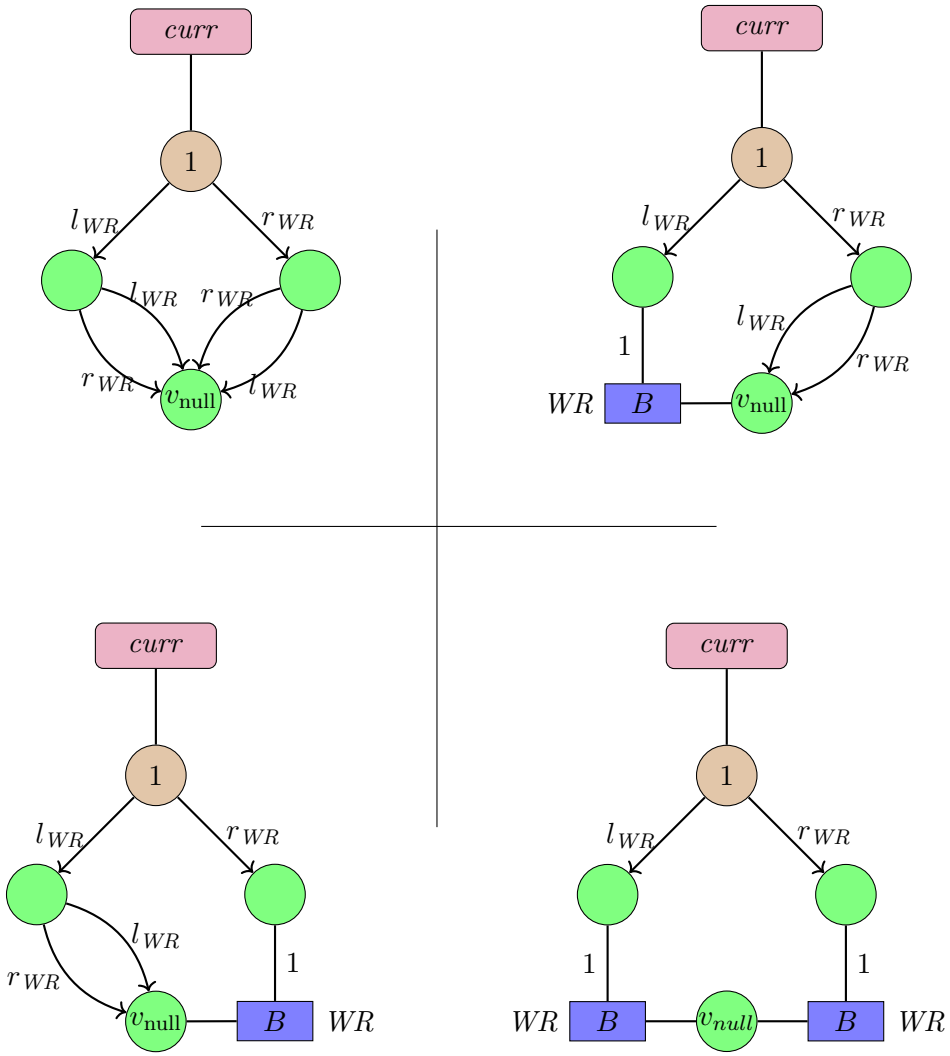


Figure 7: First step of concretisation for abstraction of binary trees, note that the permission of the nonterminal propagates to the inserted right hand side

be defined in terms of *violation points* where violation points are tentacles that hide possibly accessible selectors. Formally:

Definition 7.15 (Violation Point)**.** *For $H \in HC_{\Sigma}^{Var_{process}}$ the tupel (e, i) with $e \in E_H$, $lab_H(e) \in N$ and $1 \leq i \leq rk(lab_H(e))$ is called a violation point, if e hides accessible selectors (i.e. there exists $e' \in E_H$ such that $lab_H(e') \in Var$ and $con_H(e')(1) = con_H(e)(i)$ and $(lab_H(e), i)$ is no reduction tentacle.*

A *HC* H is called *admissible* if there is no violation point in H, the set of admissible *HCs* is denoted by $AHC_{\Sigma_N}^{Var_{\text{process}}}$. Reconsider the abstracted representation of binary trees on page 35 which is not admissible since the tentacle $(B, 1)$ hides the selectors l and r but those are actually referenceable by $curr.l$ or $curr.r$ respectively (when b is the edge of the heap such that $lab(b) = B$ then $(b, 1)$ is the violation point). On the other hand the heap representations in Figure 7 are admissible since the violation point is resolved by applying concretisation. In the following resolving violation points and therefore reestablishing admissibility is formalised for a *HAGs* G in the "re-admissibility" function

$$rea_G \colon HC_{\Sigma_N}^{Var_{\text{process}}} \to \mathbb{P}(AHC_{\Sigma_N}^{Var_{\text{process}}})$$

which maps possible abstracted *HCs* to the set of admissible *HCs* that arise by resolving violation points by applying production rules. But in fact not *every* production rule have to be applied, recall therefore the *fpHRG* G' of doubly linked list from page 33. Imagine now a program that traverses doubly linked lists of arbitrary length from the last element to the first. In order to resolve the violation point caused by the tentacle $(L, 2)$ the second production rule does not remove the violation point. But the production rule which was added to establish local concretisability can be used to resolve the violation point. Additionally, it can easily be seen that in order to establish admissibility for *HCs* different nonterminals have to be concretised (simply imagine a node that is the head of a doubly linked list as well as the root of a binary tree and is therefore connected to a nonterminal B and a nonterminal L). Since the parts that results from concretisation of nonterminals are only connected via the nodes connected to the nonterminals and apart from that are independent it can be seen that the order in which *HRs* are applied is insignificant to the resulting *HG*. This property is called *confluence* and defined as follows:

Definition 7.16 (Confluence)**.** *For $H \in HG_{\Sigma_N}^{Var_{process}}$ with $e_1, e_2 \in E_H$, $e_1 \neq e_2$ and $lab_H(e_1), lab_H(e_2) \in N$ it holds for two production rules $p_1 : lab_H(e_1) \to H_1$, $p_2 : lab_H(e_2) \to H_2$ that*

$$H[e_1/H_1][e_2/H_2] = H[e_2/H_2][e_1/H_1]$$

As stated in [16, p. 105] *HRs* always are confluent. And from this follows the following lemma (for details see [6]):

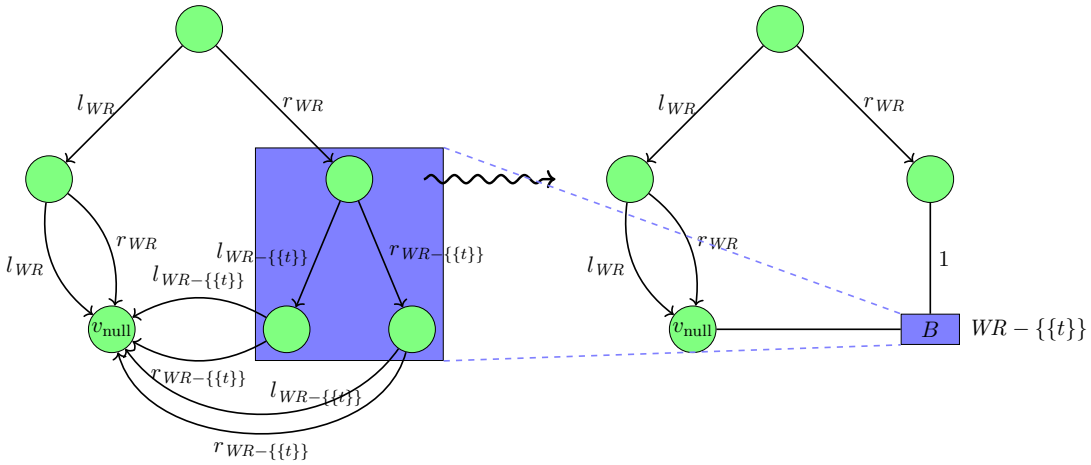


Figure 8: Heap representation of a binary tree with possible abstraction step

Lemma 7.1. *For* $G \in HAG_{\Sigma_N}^{Var_{process}}$, $H \in HC_{\Sigma_N}^{Var_{process}}$, $e \in E_H$ *and* $lab_H(e) \in N$ *holds that:*

$$L_G(H) = \bigcup_{lab_H(e) \to K} L_G(H[e/K])$$

This lemma as well as the definition of local concretisability ensures that applying rea_G still yields all possible *HCs* that can be concretised from the *HC* rea_G is applied to.

7.4 Abstraction

After having introduced how to obtain more concrete heap representations from an abstracted representation in the following it is discussed how concrete heap representations are abstracted into more general representations. The idea of such an abstraction is to apply production rules in a backward fashion. Thus identifying the right hand side of a production rule, removing it and replacing it by the according nonterminal. Note that this approach corresponds to the way write access is handed over to forked processes where also subgraphs are replaced by hyperedges. For a comprehensible presentation of this concept recall once again the grammar G from page 31 and also consider the heap representation and abstraction steps in Figure 8. In this example the right hand side of a production rule is identified in the heap representation (blue mark) and replaced by the nonterminal on the left hand side (B). Note that all permissions on the right hand side of a production rule are the same which prevents further abstraction steps although besides permission the structure fits the right hand side of a production rule, namely the third one. But since the permissions differ there is no right hand side which fits this graph. For a formal approach on identifying the right hand side of a production rule *embeddings* are introduced below. Embeddings are functions that identify subgraphs of

HGs and are formally defined as follows:

Definition 7.17 (Embedding)**.** *For* $K, H \in HRG_{\Sigma_N}^{Var_{process}}$ *an* embedding $emb = (m_V, m_E)$ *of* K *in* H *is a pair of functions with* $m_V : V_K \to V_H$ *and* $m_E : E_K \to E_H$ *that preserve the following properties:*

$lab_K(e) = lab_H(m_E(e))$	*for all* $e \in E_K$
$perm_K(e) = perm_H(m_E(e))$	*for all* $e \in E_K$
$m_V(con_K(e)) = con_H(m_E(e))$	*for all* $e \in E_K$
$⁅con_H(e)⁆ \cap m_V(V_K \setminus ⁅ext_K⁆) = \emptyset$	*for all edges* $e \in E_H \setminus m_E(E_K)$
$m_E(e) \neq m_E(e')$	*for all* $e, e' \in E_K$ *with* $e \neq e'$
$m_V(v) \neq m_V(v')$	*for all* $v \in V_K, v' \in V_K \setminus ⁅ext_K⁆$ *with* $v \neq v'$

The requirements for the functions of the embedding are as indicated distinguisheable in three topics:

1. Preservation, namely of labeling and permissions
2. Structure, namely that identified edges agree on which nodes they connect and inner nodes are not connected to any other edge in H than those that can be also found in K (since the inner nodes will be replaced by the nonterminal it would be unclear where those edges point to afterwards)
3. Injectivity, both the embedding function of edges and nodes are injective with one exception: it is possible to identify multiple external nodes of K with the same node in H, this causes the nonterminal to be connected with multiple tentacles to the same node

With these definitions applying production rules backwards is defined as:

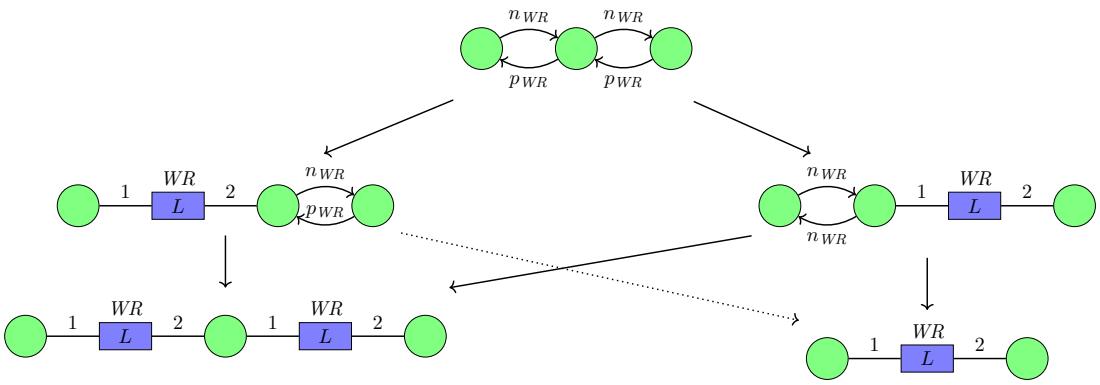
Definition 7.18 (Hyperedge introduction)**.** *For an* HAG G, $(p_\rho \colon X \to M) \in G$, $H \in HC_{\Sigma}^{Var_{process}}$ *and an embedding* (m_V, m_E) *of* K *in* H $H[M/e] \in HG_{\Sigma_N}^{Var_{process}}$ *is defined as follows:*

- $V_{H[M/e]} = V_H \setminus m_V(V_M \setminus ⁅ext_M⁆)$
- $E_{H[M/e]} = (E_H \setminus m_E(E_M)) \uplus \{e\}$
- $con_{H[M/e]} = con_H \upharpoonright (E_H \setminus m_E(E_M)) \cup \{e \mapsto m_V(ext_M)\}$
- $lab_{H[M/e]} = lab_H \upharpoonright (E_H \setminus m_E(E_M)) \cup \{e \mapsto X\}$
- $ext_{H[M/e]} = ext_H$
- $perm_{H[M/e]} = perm_H \upharpoonright (E_H \setminus m_E(E_M)) \cup \{e \mapsto \rho\}$

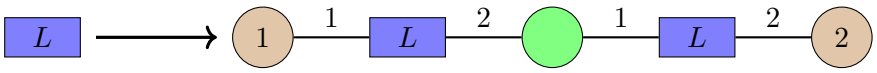
With these definitions an abstraction function

$$abs'_G : HG_{\Sigma_N}^{Var_{\text{process}}} \to \mathbb{P}(HG_{\Sigma_N}^{Var_{\text{process}}})$$

for an *HAG* G is defined which maps an *HG* $H \in HG_{\Sigma_N}^{Var_{\text{process}}}$ to the set of those *HGs* that result by introducing successively hyperedges until there is no further embedding for any right hand side of any production rule in G. Note firstly that this way of abstracting terminates because G is increasing and thus every abstraction step results in a smaller *HG* and secondly that abstraction might possibly lead to a set of *HGs* instead of a single one as the following example illustrates. Consider therefore the *fpHRG* G' from page 33 and the following example of abstraction steps where every arrow indicate the introduction of an hyperedge.



Note that the dotted arrow is only valid if the third production rule (on page 33) which established local concretisability is added to G'. Nevertheless there are two *HGs* that can not be anymore abstracted which are obtained by introducing hyperedges repeatedly. As stated for *HR* it holds that two (and therefore by an inductive argument arbitrary many) production rules are applied and there is exactly one resulting *HG*, but for backward application there are possible multiple resulting *HGs* and it depends on which abstraction is carried out first. Consider now adding additionally the production rule



to G'. This causes the above considered abstraction to result in a singleton set because the left hand leaf of this "abstraction tree" can be further abstracted to the right hand leaf. The property that abstraction results in a single possible *HG* is called *backward confluence* and a *HAG* is called backward confluent if every abstraction for an arbitrary *HG* results in a single *HG*. For a backward confluent *HAGs* G the abstraction function abs_G: $HG_{\Sigma_N}^{Var_{\text{process}}} \to HG_{\Sigma_N}^{Var_{\text{process}}}$ is defined as the mapping from every *HG* H to the single element in $abs'_G(H)$. It is unclear if it is possible to establish backward confluence for *HAGs*, but it is decideable if an *HAG* is backward confluent (for details see [10]).

7.5 Abstract Semantics

As already mentioned above and also explored in [6] it is possible to model the semantics of statements (except fork and join) on partially abstracted but admissible *HCs* because the admissibility ensures that all referenceable objects and selectors are actually currently available in the heap representation. But it is possible that the execution of such a statement invalidates the admissibility because other selectors can become referenceable because a variable might get assigned a new value. This inadmissibility has to be resolved in order to continue the analysis of further statements. In order to resolve inadmissibilities the heap representation is at first completely abstracted and subsequently as far as necessary concretised to reestablish admissibility. The first abstraction step is used in order to minimise the heap representation because rea_G stops as soon as admissibility is established. With these intuitions the abstract semantics (denoted by $\blacktriangleright$) of pointer operations (which are all statements except of fork, join and assignment of process identifiers) can be given as follows:

$$\frac{(S,H) \rhd (S',I) \qquad H' \in rea_G(abs_G(I))}{(S,H) \blacktriangleright (S',H')}$$

For the semantics of the assignment of process identifiers the concrete semantics also transfer to partially abstracted *HCs* since the definitions given for the modifications of permissions in Section 6.1 are applicable for nonterminals as well, thus no changes are required.

For join and fork on the other hand some modifications in semantics have to be introduced. At first the postponed definitions of *reachability* and *border* nodes are addressed and afterwards the semantics of fork and join are applied on partially abstracted *HCs*.

The set of selectors that are reachable from an initial node are those selectors that can be reached via moving along other selectors. For example consider the *HC* described in Figure 9. The set of reachable edges from v_1 is $\{e_1, e_2, e_4, e_5\}$ since from v_1 e_1 is directly

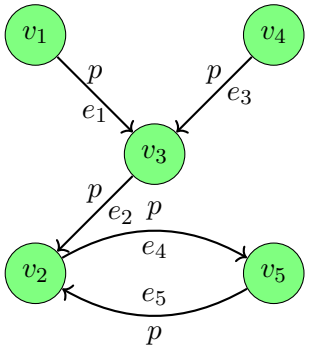


Figure 9: simple *HC* to illustrate reachability

reachable and after moving along e_1 the edge e_2 is reachable and so forth. On the other hand e_3 is not reachable from v_1 since the selector points in the "wrong" direction. In

order to formalise these movements along the edges in $H \in HG_{\Sigma_N}^{Var_{\text{process}}}$ the notion of a path π is introduced as

$$\pi \in (\mathbb{N} \times E_H \times \mathbb{N})^*$$

where it holds for $\pi(k) = (i, e, j)$ and $\pi(k+1) = (i', e', j')$ that $con_H(e)(j) = con_H(e')(i')$ for all $1 \leq k \leq (|\pi| - 1)$. This describes undirected paths, i.e.

$$(1, e_1, 2)(2, e_3, 1)$$

is a valid path although e_3 is actually not reachable from v_1. Thus, not all paths agree with the intuition of reachability. The intuition of reachability is easy for selectors which are interpreted as directed edges. For nonterminals reachability is understood as the possibility to concretise this nonterminal in a way such that there is a reachable path along selectors between the connected nodes. Formally defined is reachability by the use of the term bridge as follows:

Definition 7.19 (Reachability)**.** *A bridge $(i, X, j) \in br(N \cup Sel)$ is called* reachable *for $G \in HAG_{\Sigma_N}^{Var_{process}}$ if $i = 1, j = 2, X \in Sel$ or if $X \in N$ and there is $H \in L_G(X^\bullet)$ such that there is a reachable path π from $v_i^\bullet$ to $v_j^\bullet$.*

Note that it is assumed that the set *Sel* is ranked by the function that maps all selectors to 2 to ensure that $br(N \cup Sel)$ is defined. A path π is called reachable if every used bridge is reachable. Let $Path(u, v)$ denote all paths that start in u and end in v, formally:

$$Path(u, v) = \left\{ \pi \mid \begin{array}{l} \pi \text{ is a path} \\ \wedge\, (i, e, j) = \pi(1) \wedge con_H(e)(i) = u \\ \wedge\, (i', e', j') = \pi(|\pi|) \wedge con_H(e')(j') = v \end{array} \right\}$$

Furthermore it is possible to compute all reachable bridges over nonterminals in a *HAG* over syntactical analysis of the *HAG* as the following lemma states:

Lemma 7.2. *For $G \in HAG_{\Sigma_N}^{Var_{process}}$ the set $RB(G)$ denotes all* reachable bridges *in G. $RB(G)$ can be computed by syntactical analysis of G.*

Proof. Let

$$RB_0 = \{(1, s, 2) \mid s \in Sel\}$$

denote the set of all bridges over selectors in the "right" direction. Let furthermore $NT(\pi)$ denote all used bridges in the path π in an *HG H*

$$NT(\pi) = \{(i, lab_H(e), j) \mid (i, e, j) \in \langle \pi \rangle\}$$

then

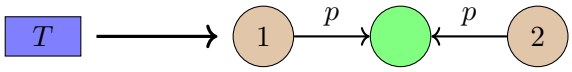
$$RB_{n+1} = \{(i, X, j) \mid \exists H \in rhs(G^X). \exists \pi \in Path(ext_H(i), ext_H(j)). NT(\pi) \subseteq RB_n\}$$

denote the set of all reachable bridges that make use of previously computed reachable bridges which ensure reachable paths. For example, RB_1 is the set of bridges over selectors and those nonterminals for which a right hand side of a production rule exists where a path along selectors connects $ext_H(i)$ and $ext_H(j)$. It is easy to see that this iteration is monotone and since the set of all bridges for a finite set of nonterminals N and selectors Sel is finite as well it terminates after finitly many steps. By induction over n, the depth of applied concretisation steps for nonterminals until a reachable path is found it follows that the fixpoint of this iteration is indeed $RB(G)$. □

Combining this result with a breadth-first search that computes stepwise all possible paths of finite length and with the set $RB(G)$ it can be tested if these paths are reachable. It follows that for a finite graph all reachable paths can be computed. Let $CPath_H(v)$ denote all reachable paths in H starting in the node v:

Corollary 7.1. *For $H \in HG_{\Sigma_N}^{Var_{process}}$, $G \in HAG_{\Sigma_N}^{Var_{process}}$ and $v \in V_H$ the set $CPath_H(v)$ can be computed by syntactical analysis of G and a structural analysis of H.*

Another point has to be taken into account before the set of reachable edges can be formally defined. It is illustrated by the following production rule:



It is evident that the bridges $(1, T, 2), (2, T, 1)$ are both not reachable. But also both tentacles $(T, 1), (T, 2)$ are not reduction tentacles since both actually abstract selectors. If any reachable path from a starting node u to a node v exists such that v is connected to a T-labeled edge e, then e has to be included in the set of reachable edges, since it abstracts edges which can actually be reached. With this in mind the set of reachable edges from a node v in a HG H denoted by $reach^H_{\text{abs}}(v)$ can be formally defined in two steps. First all edges of reachable paths are gathered in

$$P = \{e \mid (i, e, j) \in \wr\pi\wr, \pi \in CPath_H(v)\}$$

and secondly all nonterminals where parts of the concretisating graphs can be reached in

$$D = \left\{ e \mid \begin{array}{l} lab_H(e) \in N, 1 \leq i \leq rk(lab_H(e)).\exists e' \in P.con_H(e)(i) \in \wr con_H(e')\wr \\ \wedge\, (lab_H(e), i) \text{ is no reduction tentacle} \end{array} \right\}$$

Then the set of reachable edges is defined as:

$$reach^H_{\text{abs}}(v) = P \cup D$$

and let

$$reach^H_{\text{abs}}(v_1, \ldots, v_n) = \bigcup_{1 \leq i \leq n} reach^H_{\text{abs}}(v_i)$$

denote the reachable nodes from multiple initial nodes. This concludes computing the reachable edges in a partially abstract *HG*. As mentioned before the computation of reachable edges for fully concrete *HGs* relies on the reachability in the abstracted case and is intuitively achieved by abstracting the concrete *HG*, computating the set of reachable edges, concretising with the same sequence of production rules that is used for the abstraction to rebuild the initial *HG*. The set of reachable edges in the concrete *HG* are all edges that arise from reachable edge in the abstraction. Therefore *production sequences* are introduced as sequence of tupel of production rules and edges. A production sequence denotes the way a heap is transformed via production rules. This means it is possible to revert abstraction steps by saving tupels of the production rules and the introduced hyperedges that arise from backward application of that production rule and applying this sequence in reversed order to the abstracted *HG* yields again the original concrete *HG*. Let π be a finite production sequence then π^{-1} denotes the reversed production sequence. Furthermore let $\pi \restriction E$ denote the production sequence that contains all production rules in π that are applied to edges in E or edges that are concretised from E (in possibly multiple applications of production rules). Let furthermore $H \overset{\pi}{\Rightarrow} Q$ denote the application of a sequence of production rules π. Especially it holds for the production sequence π which is used to obtain $abs_G(H)$ that $abs_G(H) \overset{\pi^{-1}}{\Longrightarrow} H$. Additionally, let H be a partially abstracted *HG* then let E_π for an production sequence π denote the set of all edges that arise from concretisation steps applied to edges of E and edges that are originally obtained from edges in E.

$$E_\pi = \{e \mid (H \restriction E) \overset{\pi \restriction E}{\Longrightarrow} H', e \in E_{H'}\}$$

With these notions reachability for concrete hypergraphs can be defined in context of a *HAG* G as follows:

$$reach_H(v_1, \ldots, v_n) = (reach_{\text{abs}}^{abs_G(H)}(v_1, \ldots, v_n))_{\pi^{-1}}$$

where the abstraction preserves $v_1, \ldots, v_n$ and π denotes the corresponding production sequence of the abstraction.

For the contracts of the forked programs there is one difference to the concrete semantics which is that a partially abstract precondition concretises to various possible *HCs* (similar to the contracts presented in [11]). Just like concretisation of nonterminals demands that all production rules that establish admissibility are analysed (since the information from which of these concrete *HCs* it is abstracted from is lost) there are also multiple postconditions that might apply after the execution of a program depending on the actual structures the partially abstract precondition concretises to. Formally, this leads to a set of contracts for a program m where one precondition leads to multiple postconditions:

$$Cont_{\text{abs}}(m) \subseteq \mathbb{P}((HG_{\Sigma_N}^{Var_{\text{process}}} \times \mathbb{P}(E) \times \mathbb{P}(HG_{\Sigma'_N}^{Var'_{\text{process}}})))$$

where for every $C = (P_C, E_C, \mathbb{Q}_C) \in Cont_{\text{abs}}(m)$ holds that $E_C \subseteq E_{WR}^{P_C}$. Furthermore there is for every program m and precondition P maximal one contract $C \in Cont_{\text{abs}}(m)$.

Because if there are two contracts $C, C' \in Cont_{\text{abs}}(m)$ with $P_C \ {}_{ap}\!\cong_{fp} P \ {}_{ap}\!\cong_{fp} P_{C'}$ then those contracts can be united to a contract $C'' = (P, E_C \cup E_{C'}, \mathbb{Q}')$ such that $\mathbb{Q}'$ contains all possible results from executions of m starting from the initial heap state (which is obtained from the precondition and the set of alternable edges the same way it is described in Section 6.5 with straightforward ajustments for nonterminals).

As already introduced the intuition for border nodes is that these nodes are part of the subgraph for which the access tickets are transferred to the newly forked process but are also still part and accessible (with limitation to some selectors) to the forking process. For the formal definition let H be the current heap representation, $C = (P_C, E_C, \mathbb{Q}_C)$ be a contract for the forked process, R be the reachable subgraph from the actual parameter of the fork statement and it holds that $R \ {}_{ap}\!\cong_{fp} P_C$. Then the set of border nodes can be defined as follows:

$$border^H_{\text{abs}}(C) = \left(\bigcup_{e \in E_C} \wr con_H(e) \wr \right) \bigcap \left(\bigcup_{e \in E_H \setminus E_C} \wr con_H(e) \wr \right)$$

which leads to the following transformation of the heap representation of the forking process:

$$H' = H[\downarrow t][\backslash E_C][+_{WR} N_{\{t\}} \rightrightarrows enum_{b_C(H)}][(E_{P_C} \setminus E_C) - \{t\}]$$

where t denotes the process identifier that identifies the newly forked process initially. With these definitions the abstract semantics for the fork statement can be given as follows where $R = (H \restriction reach_{\text{abs}}([\![x_1]\!]_H, \ldots, [\![x_n]\!]_H))$

$$\frac{(P_C, E_C, \mathbb{Q}_C) \in Cont_{\text{abs}}(m) \qquad P_C \ {}_{ap}\!\cong_{fp} R \qquad H'' \in rea_G(abs_G(H'))}{(t = \text{fork}(m(x_1, \ldots, x_n)), H) \blacktriangleright (\varepsilon, H'')}$$

Lastly, the join statement is dealt with almost the same as in the concrete semantics. Recall therefore the two transformations steps for one precondition Q from page 28:

$$Q'_1 = Q_C[\downarrow enum_{Var'_{\text{thread}}}(1)] \ldots [\downarrow Var'_{\text{thread}}(|Var'_{\text{thread}}|)]$$

and

$$Q'_2 = Q'_1[\backslash (E^{Q'_1}_{RD} \cup E^{Q'_1}_{RD^*})]$$

then there are again two transition rules for the join statement where one deals with fully returned read tickets and the other one with lost read tickets. Especially *every* postcondition $Q \in \mathbb{Q}_{\mathbb{C}}$ is examined (since every postcondition describes a valid execution of the program from the initial heap state).

$$\frac{Q \in \mathbb{Q}_C \qquad E^{Q'_1}_{RD^*} \neq \emptyset \qquad H[\downarrow T_t] \xrightarrow{N_{T_t} \to Q'_2} H' \qquad H'' \in rea_G(abs_G(H'))}{(\text{join}(t), H) \blacktriangleright (\varepsilon, H'')}$$

$$\frac{Q \in \mathbb{Q}_C \qquad E^{Q'_1}_{RD^*} = \emptyset \qquad H[\leftarrow T_t] \xrightarrow{N_{T_t} \to Q'_2} H' \qquad H'' \in rea_G(abs_G(H'))}{(\text{join}(t), H) \blacktriangleright (\varepsilon, H'')}$$

where $C = (P_C, E_C, \mathbb{Q}_C)$ is the contract which the process is forked by before (which can be obtained by attaching it to the placeholder when the fork statement is executed). Additionally it can be explained in the context of abstraction why it is avoided for the concrete semantics as well as for the abstract semantics to match the shared edges from the postcondition to the ones in the heap representation of the joining process. Because of the applied abstraction it is possible that nonterminals that are identified at the join statement get concretised and abstracted differently through the execution of both processes. To avoid accounting from which nonterminals different edges arose (especially since it had to be done for every forked process individually) it is just dealt with by handing over one read ticket for all shared edges and returning the minimum of this read ticket (nothing if any edge cannot return its ticket completely or everything if every edge can guarantee to return the whole ticket). Also this fits the approach to deal with abstraction and permissions as orthogonal concepts (like it is approached by introducing fully permissive grammars on page 30). Note that again one abstraction and one concretisation step is applied to the resulting heap representation. This is done because after the transformation of permissions as well as inserting the "*WR* part" into the heap representation it is possible that the right hand side of production rule can be found in the resulting heap representation which were not present before. Thus, to obtain the most general heap representation the abstraction step is executed and following to obtain the minimal admissible *HCs* one concretisation step is applied.

7.6 Correctness

The main result justifying the taken approach on the presented abstraction is to show that $\blacktriangleright$ is an over-approximation of the transition relation $\rhd$. Therefore it is assumed that abstraction and concretisation relies on a backward confluent *HAG* G and abstract and concrete contracts are connected in the following way: For every $C = (P_C, E_C, Q_C) \in Cont(m)$ exists $C' = (P_{C'}, E_{C'}, \mathbb{Q}_{C'}) \in Cont_{\text{abs}}(m)$ such that

(i) $P_C \in L_G(P_{C'})$, which implies there is a production sequence π such that $P_{C'} \overset{\pi}{\Rightarrow} P_C$

(ii) $P_{C'} \restriction E_{C'} \overset{\pi \restriction E_{C'}}{\Longrightarrow} P_C \restriction E_C$

(iii) there is $Q' \in \mathbb{Q}_{C'}$ with $Q_C \in L_G(Q')$

This intuitively means that (i) preconditions of concrete contracts arise from concretisation of preconditions of abstract contracts, where (ii) every edge in the alternable set of the concrete contract arise from concretisation of edges in the alternable set of the abstract contract and (iii) the postcondition of the concrete contract can be found by concretising one of the postconditions of the abstract contract. Note further that the order of π does not matter due to the confluence property of *HR*, but a previous *HR* can expose the hyperedges which later production rules are applied to. This might cause dependencies between the production rules. A subsequence χ of π denoted as $\chi \prec \pi$ is a sequence of production rules such that these production rules appear also in π.

These assumptions are essential requirements for the presented analysis and abstraction approach of this paper. Because the actual analysis is actually designed for abstracted representation, it is expected that the set of abstracted contracts is computed first and the concrete contracts are simply generated by concretisation of abstracted contracts. This justifies that even for the concrete semantics a corresponding abstracted contract exists for every concrete contract. This is necessary because the border nodes for the concrete semantics are computed by abstracting and computing the border nodes for the abstracted case and concretise by the sequence of production rules used for the abstraction. This is necessary to ensure that the abstract semantics overapproximate the concrete semantics. This approach agrees with the intuition of the border nodes in the concrete semantics because the computation of border nodes in the abstracted *HC* is a superset of the actual border nodes. Recall therefore the intutition of border nodes for the concrete semantics as all nodes that are connected to an edge for which *WR* permission is transferred and an edge that is still present within the heap representation of the forking process. Let H be this heap representation and $C = (P_C, E_C, Q_C)$ the contract by which the fork statement is executed then the intuition translates to the following set:

$$IB_H(C) = \{v \mid \exists e \in E_C . \exists e' \in E_H \setminus E_C . v \in \wr con_H(e) \wr \cap \wr con_H(e') \wr\}$$

Then the following lemma states that this computation ensures that the border nodes are a superset of the nodes that are intuitively understood as border nodes:

Lemma 7.3 (Border Lemma)**.** *For $C \in Cont(m)$ and $C' \in Cont_{abs}(m)$ such that for C and C' $(i), (ii), (iii)$ holds that $IB_H(C) \subseteq border_H(C')$*

Firstly one additional property for *HR*, namely *context-freeness*, is presented in the following because it motivates some of the used results for the presented proofs. But in order to avoid some formal machinery context-freeness is only presented informally, for formal details as well as the proof that *HR* actually is context-free see [16, pp. 111-115]: The deriveration of nonterminals is context-free in the sense that it is independent from the rest of the *HG*. This means, first applying production rules to nonterminals and glueing the resulting hypergraphs together yields the same result as glueing the hypergraphs together and applying the same production rules afterwards. In the following the proof for the *Border Lemma* is presented which actually makes use of context-freeness property:

Border Lemma. Let $v \in IB_H(C)$ be arbitrarily chosen. It follows that $v \in border_H(C)$ by the following argument: Since $v \in IB_H(C)$ it follows that there is $e \in E_C$ and $e' \in E_H \setminus E_C$ such that $v \in \wr con_H(e) \wr \cap \wr con_H(e') \wr$. Let furthermore C' denote the abstract contract to C for which $(i), (ii), (iii)$ holds. By the context-freeness property of *HR* and because (ii) holds follows that there are $e_{\text{abs}} \in E_{C'}$ and $e'_{\text{abs}} \in E_{abs_G(H)} \setminus E_{C'}$ such that e arises from concretisation of e_{abs} and e' from concretisation of e'_{abs}. Therefore it follows immediatly that $v \in \wr con_{abs_G(H)}(e_{\text{abs}}) \wr \cap \wr con_{abs_G(H)}(e'_{\text{abs}}) \wr$ and hence $v \in border_{C'}(H)$. □

This proves that it is viable to rely on abstraction in order to determine the border nodes because it overapproximates the definition of border nodes for the concrete case.

In order to proof the overapproximation of $\blacktriangleright$ it is actually shown that if $(S, H) \rhd (S', H')$ that there is K such that $(S, abs_G(H)) \blacktriangleright (S', K)$ and $H' \in L_G(K)$. Furthermore the proof is only presented for the fork and join statement and the assignment of process identifier. For all other cases the overapproximation can be shown straigthforwardly by adapting the proof of overapproximation presented in [6]. Especially noteworthy in this context is that the presented abs_G and rea_G functions of this paper with Theorem 7.1 and that every backwards application of a production rule can be undone by forwards application of the same production rule satisfy the requirements for the concretisation and abstraction functions demanded by the *Correctness Theorem* in [6, p. 19].

For the following proofs of overapproximation for fork and join some general arguments are presented in front to reduce the formal complexity of the actual argumentation:

(I) Because permissions propagate strictly through production rules it follows for a production sequence π and two *HGs* H, Q with $H \overset{\pi}{\Rightarrow} Q$ and a permission ρ that

$$(E^H_\rho)_\pi = E^Q_\rho$$

because every nonterminal with permission ρ can only concretise to edges with permission ρ and also edges with permission ρ can only be abstracted into nonterminals with permission ρ.

(II) Since production rules are generally assumed to not abstract placeholder and variables it can be assured that nodes that are identified by variables are preserved by abstractions and introduced placeholder are preserved through concretisation.

Lemma 7.4 (Overapproximation of Fork)**.** *For a backward confluent* $G \in HAG^{Var_{process}}_{\Sigma_N}$, $H, H' \in HC^{Var_{process}}_{\Sigma}$ *and* $(t = \textbf{\textit{fork}}(m(x_1, \ldots, x_n)), H) \rhd (\varepsilon, H')$ *it holds that there is* $I \in HC^{Var_{process}}_{\Sigma_N}$ *with* $(t = \textbf{\textit{fork}}(m(x_1, \ldots, x_n)), abs_G(H)) \blacktriangleright (\varepsilon, I)$ *and* $H' \in L_G(I)$.

Proof. Let C be the contracted by which $(t = \textbf{fork}(m(x_1, \ldots, x_n)), H) \rhd (\varepsilon, H')$ is computed. Then by the definition of $\rhd$ it follows that $H' = H[\downarrow t][\backslash E_C][+_{WR} N_{\{t\}} \rightrightarrows enum_{b_C(H)}][(E_P \setminus E_C) - \{t\}]$. Let C' be the abstract contract such that C and C' satisfy $(i), (ii), (iii)$. With this it is shown that $H' \in L_G(abs_G(H)[\downarrow t][\backslash E_{C'}][+_{WR} N_{\{t\}} \rightrightarrows enum_{b_{C'}(abs_G(H))}][(E_{P_{C'}} \setminus E_{C'}) - \{t\}])$ by the following argument: Obviously there is a production sequence π such that $abs_G(H) \overset{\pi}{\Rightarrow} H$. Furthermore the permissions in H and in $abs_G(H)$ are altered both by $[\downarrow t]$ the same way, also because placeholders cannot be abstracted (by argument (II)) and because the permission propagate through concretisation (by argument (I)) it follows that there is a production sequence π' that mirrors π but adapts the permissions of production rules according to the edges these production rules are applied to such that $abs_G(H)[\downarrow t] \overset{\pi'}{\Rightarrow} H[\downarrow t]$. Secondly, it can be assured by the definition of $reach_H$ and because $[\![x_1]\!]_H, \ldots, [\![x_n]\!]_H$ are preserved through abstraction (by argument (II)) that for $R \coloneqq reach^{abs_G(H)}_{\text{abs}}([\![x_1]\!]_{abs_G(H)}, \ldots, [\![x_n]\!]_{abs_G(H)})$ it follows that

$\underbrace{abs_G(H) \upharpoonright R}_{P_{C'}} \overset{\pi'\upharpoonright R}{\Longrightarrow} \underbrace{H \upharpoonright reach_H([\![x_1]\!]_H, \ldots, [\![x_n]\!]_H)}_{P_C}$. Because of the context-freeness of HR and property (ii) for C and C' it follows that removing $E_{C'}$ in $abs_G(H)[\downarrow t]$ removes E_C in $H[\downarrow t]$. This implies that there is a production sequence π'' which is the same as π' but reduced to those production rules for which the edges are actually present after removing $E_{C'}$ in $abs_G(H)$ such that $abs_G(H)[\downarrow t][\backslash E_{C'}] \overset{\pi''}{\Longrightarrow} H[\downarrow t][\backslash E_C]$. Since the border nodes in both cases are determined the same way and (II) holds it follows immediatly that $abs_G(H)[\downarrow t][\backslash E_{C'}][+_{WR} N_{\{t\}} \rightrightarrows enum_{b_{C'}(abs_G(H))}] \overset{\pi''}{\Longrightarrow} H[\downarrow t][\backslash E_C][+_{WR} N_{\{t\}} \rightrightarrows enum_{b_C(H)}]$. Finally, because of (I) it follows that there is π''' such that $abs_G(H)[\downarrow t][\backslash E_{C'}][+_{WR} N_{\{t\}} \rightrightarrows enum_{b_{C'}(abs_G(H))}][(E_{P_{C'}} \setminus E_{C'}) - \{t\}] \overset{\pi'''}{\Longrightarrow} H[\downarrow t][\backslash E_C][+_{WR} N_{\{t\}} \rightrightarrows enum_{b_C(H)}][(E_P \setminus E_C) - \{t\}]$ where π''' mirrors π'' but adapts the permissions of the production rules according to the permissions of the edges they are applied to. This concludes that $H' \in L_G(abs_G(H)[\downarrow t][\backslash E_{C'}][+_{WR} N_{\{t\}} \rightrightarrows enum_{b_{C'}(abs_G(H))}][(E_{P_{C'}} \setminus E_{C'}) - \{t\}])$ and because rea_G preserves the language of the HG it is applied to it follows that there is $I \in HC^{Var_{\text{process}}}_{\Sigma_N}$ such that $(t = \textbf{fork}(m(x_1, \ldots, x_n)), abs_G(H)) \blacktriangleright (\varepsilon, I)$ with $H' \in L_G(I)$. □

And secondly the join statement is examined in detail as follows:

Lemma 7.5 (Overapproximation of Join)**.** *For a backward confluent $G \in HAG^{Var_{process}}_{\Sigma_N}$, $H, H' \in HC^{Var_{process}}_{\Sigma}$ with $(\textbf{\textit{join}}(t), H) \rhd (\varepsilon, H')$ it holds that there is $I \in HC^{Var_{process}}_{\Sigma_N}$ such that $(\textbf{\textit{join}}(t), abs_G(H)) \blacktriangleright (\varepsilon, I)$ and $H' \in L_G(I)$.*

Proof. Let T_t denote the token of identifiers that identify the process t and $C = (P_C, E_C, Q_C) \in Cont(m)$ the contract by which the process identified by all $t' \in T_t$ is joined. Let further more denote $C' = (P_{C'}, E_{C'}, \mathbb{Q}_{C'}) \in Cont_{\text{abs}}(m)$ such that C and C' satisfy $(i), (ii), (iii)$. Therefore there is (at least one) $Q_{C'} \in \mathbb{Q}_{C'}$ such that $Q_C \in L_G(Q_{C'})$. This implies there is a production sequence π such that $Q_{C'} \overset{\pi}{\Rightarrow} Q_C$. Furthermore it is clear that there is a production sequence λ with $abs_G(H) \overset{\lambda}{\Rightarrow} H$. Additionally it is $Q_1 = Q_C[\downarrow enum_{Var'_{\text{process}}}(1)] \ldots [\downarrow enum_{Var'_{\text{process}}}(|\,Var'_{\text{process}}|)]$ and $Q_2 = Q_1[\backslash(E^{Q_1}_{RD} \cup E^{Q_1}_{RD^*})]$, and accordingly $Q^{\text{abs}}_1 = Q_{C'}[\downarrow enum_{Var'_{\text{process}}}(1)] \ldots [\downarrow enum_{Var'_{\text{process}}}(|\,Var'_{\text{process}}|)]$ and $Q^{\text{abs}}_2 = Q^{\text{abs}}_1[\backslash(E^{Q^{\text{abs}}_2}_{RD} \cup E^{Q^{\text{abs}}_2}_{RD^*})]$. By argument (I) and because the successively dropped process identifier alternate the permissions in Q_C and $Q_{C'}$ the same way (an analogous case is examined in the proof of the overapproximation for the fork statement above) it can be savely assumed that $Q^{\text{abs}}_1 \overset{\pi'}{\Rightarrow} Q_1$ where π' mirrors π but adapts the permissions of the production rules to fit the permissions of the edges they are adapted to, and furthermore it follows that $E^{Q_1}_{RD^*} = \emptyset$ if and only if $E^{Q^{\text{abs}}_1}_{RD^*} = \emptyset$, since every e with RD^* in Q_1 has to arise from an edge with RD^* permission in Q^{abs}_1 and every edge in Q^{abs}_1 with an RD^* permission concretises to edges with RD^* permissions. This implies that the abstract semantics as well as the concrete semantics agree upon which of both production rules is used in both cases. Additionally because the "write-part" and the

"read-part" of the postcondition can be distinguished by their *BasePerm* and of argument (I) there is a production sequence π'' which is the same as π' restricted to the edges with *WR*, *WR** permissions ($\pi'' = \pi' \restriction E^{Q_1^{\text{abs}}}_{WR} \cup E^{Q_1^{\text{abs}}}_{WR^*}$) such that $Q_2^{\text{abs}} \overset{\pi''}{\Longrightarrow} Q_2$. Let now in the following $\Delta \in \{\downarrow T_t, \leftarrow T_t\}$ denote the graph transformation which is applied to the heap representation to return the "read-part" of the postcondition. It is already established that $abs_G(H) \overset{\lambda}{\Rightarrow} H$. Because both transformations change the permissions in $abs_G(H)$ and H the same way, this implies that there is λ' which mirrors λ aside the permissions which are adapted to fit the edges the production rules are applied to such that $abs_G(H)[\Delta] \overset{\lambda'}{\Rightarrow} H[\Delta]$. Let in the following $K \in HC^{Var_{\text{process}}}_{\Sigma_N}$ denote the *HC* such that $abs_G[\Delta] \xrightarrow{N_{T_t} \to Q_2^{\text{abs}}} K$ holds. And H' is by the definition of the production rules for the concrete semantics (see page 28) the *HC* such that $H[\Delta] \xrightarrow{N_{T_t} \to Q_2} H'$. It follows from the context-freeness and independence of π'' and λ' (i.e. no production rule in π'' is needed to reveal edges which production rules of λ are applied to and vice versa) that $K \overset{\lambda'}{\Rightarrow} \underbrace{K'}_{\text{intermediate state}} \overset{\pi''}{\Longrightarrow} H'$, where the intermediate step K' is fully concrete in the part around the inserted Q_2^{abs} which is then concretised by π''. Therefore $K \overset{\lambda'\pi''}{\Longrightarrow} H'$ which implies $H' \in L_G(K)$. Finally, because application of abstraction and concretisation yields at least the language of the *HC* they are applied to it follows that there is $I \in rea_G(abs_G(K))$ such that $H' \in L_G(I)$. □

At last the overapproximation of the assignment of process identifier is given.

Lemma 7.6 (Overapproximation of Assignment of Process Identifier)**.** *For a backward confluent* $G \in HAG^{Var_{process}}_{\Sigma_N}$*,* $H, H' \in HC^{Var_{process}}_{\Sigma}$ *with* $(t = t', H) \rhd (\varepsilon, H')$ *it holds that there is* $I \in HC^{Var_{process}}_{\Sigma_N}$ *such that* $(t = t', abs_G(H)) \blacktriangleright (\varepsilon, I)$ *and* $H' \in L_G(I)$*.*

Proof. This follows immediatly from argument (I) and because $[t = t']$ operates on the permissions of the concrete *HC* as well as the corresponding abstraction the same way. □

This concludes the proof that $\blacktriangleright$ is an overapproximation of $\rhd$. ■

8 Data Race Freedom

The main result of incorporating permissions into the heap representation is to avoid data races. This section proves the absence of dataraces for valid executions, i.e. executions that does not yield a $\perp$ symbol. To improve the readability some considerations are given up front:

1. Edges are only altered if the permission of this edge is WR as it can be easily seen from the transition rules on page 19.

2. Selectors can only be read if the corresponding edge is present in the representation of the heap, which follows likewise from the transition rules on page 19

3. Let $Tok(H) = \{T \mid \text{T is a token and either } \exists e \in E_H.perm_H(e) = \rho - \Phi \wedge T \in \Phi \text{ or } \exists e' \in E_H.lab_H(e') = N_T\}$ denote the set of all tokens in H. The set of tokens is called *consistent* if they are all disjoint and all process identifier in a token refer to the same process.

4. *BasePerms* that are once starred can never be "un-starred" again which satisfies to drop accounting for permissions with starred *BasePerms* (as it can be seen in the property **Derived Tickets** only not starred permissions are examined).

8.1 Proof Obligation

It is shown by induction of the transition relation that for every HC H of the transition system described by $\rhd$ holds:

Consistency of Tokens that $Tok(H)$ is consistent

Uniqueness of Access that there is only one process for which the *BasePerm* of an edge in the heap representation is WR or WR^*, i. e. write access is always transferred completely

Derived Tickets that all derived tickets for read access for edges with a *BasePerm* that is either WR or RD are properly accounted in the *PermSet*, where properly accounted means that for every derived ticket the token of the process is given in the *PermSet*

For this induction only the fork and join statement as well as the allocation and assignment of process identifier is examined, because all other statements cannot alter any permission or placeholder.

Let as induction hypothesis for all following proofs H be a heap representation which satisfies **Consistency of Tokens** and **Uniqueness of Access** and **Derived Tickets**.

Allocation. For $(\text{new}(x), H) \rhd (\varepsilon, \underbrace{H[+v][x \hookrightarrow v]}_{H'})$ it holds that there are new edges for used selector attached to the node v. All these edges cannot accessed by any other

process other than the current one. Therefore H' satisfies **Uniqueness of Access**. Furthermore H' satisfies **Consistency of Tokens** by induction hypothesis and because no new tokens are added. Since for the new selectors no access tickets are yet derived the empty *PermSet* is a proper accounting, yielding **Derived Tickets**. □

Assignment of Process Identifier. For $(t = t', H) \rhd (\varepsilon, H[t = t'])$ it holds that $H[t = t']$ satisfies **Consistency of Tokens** because $[t = t']$ implicitly contains $[\downarrow t]$ which ensures, that t is removed from all tokens and afterwards it is added to the token that refers to the process t' refers to (by induction hypothesis there is one unique token $T_{t'}$ to which t is added).

Additionally, **Uniqueness of Access** follows immediatly from the induction hypothesis since it is only possible that *BasePerms* are starred but no access tickets are somehow transferred between processes.

And finally, **Derived Tickets** holds because analogous to the argumentation of **Consistency of Tokens** the *BasePerm* of all permission for which no proper accounting can be guaranteed, i.e. those for which the access ticket was represented by $\{t\}$, are starred (since this derived ticket can never be regained). For the others the proper accounting follows inherently from the induction hypothesis and because the tokens referring to the different processes are consistent (**Consistency of Tokens**). □

Fork. For $(t = fork(m(x_1, \ldots, x_n)), H) \rhd (\varepsilon, \overbrace{H[\downarrow t][\backslash E_C][+_{WR} N_{\{t\}} \rightrightarrows enum_{b_C(H)}][(E_{P_C} \setminus E_C) - \{t\}]}^{H'})$ let $C = (P_C, E_C, Q_C) \in Cont(m)$ be the contract by which the new process is forked.

Consistency of Tokens is ensured for H' because again $[\downarrow t]$ ensures that t is removed from all used tokens and therefore $\{t\}$ can be added safely by the introduction of the placeholder $N_{\{t\}}$ and $[(E_{P_C} \setminus E_C) - \{t\}]$. Note further that $\{t\}$ is a consistent token since t is the only process identifier referring to the newly forked process. Futhermore by the definition of the initial heap state it satisfies **Consistency of Tokens** as well since all its permissions are simple at the beginning which implies an empty set of tokens.

Uniqueness of Access can be assured because by the definition of how the initial heap state is obtained it follows that only for edges from E_C the permission is *WR* (for all others the permission is *RD*). Since $E_C \subseteq E^{P_C}_{WR}$ by the premise for contracts and because H satisfies **Uniqueness of Access** the transformation $[\backslash E_C]$ ensures **Uniqueness of Access** for H' because the edges for which the initial heap state has *WR* access are removed for H'. Furthermore this argument also yields **Uniqueness of Access** for the initial heap state of the forked process as well.

It can also be seen that H' satisfies **Derived Tickets** since for all $e \in E_{H[\downarrow t]}$ with a *BasePerm* that is either *WR* or *RD* **Consistency of Tokens** ensures a proper accounting of the derived access tickets. And all reachable edges are either removed $[\backslash E_C]$ or the token $\{t\}$ is added which accounts the the newly derived access ticket for all reachable edges that are not removed. Concludingly, H' satisfies **Derived Tickets**. Also for the initial heap state all *PermSets* are empty and because there is not yet any derived ticket for the initial heap state it also satisfies **Derived Tickets**. □

Join. For $(\text{join}(t), H) \rhd (\varepsilon, H')$ let $C = (P_C, E_C, Q_C)$ denote the contract by which the joined process was forked intially. Furthermore it holds that $Q_1 = Q_C[\downarrow \mathit{enum}_{\mathit{Var}'_{\text{thread}}}(1)] \ldots [\downarrow \mathit{Var}'_{\text{thread}}(|\mathit{Var}'_{\text{thread}}|)]$ and $Q_2 = Q'_1[\setminus(E^{Q'_1}_{RD} \cup E^{Q'_1}_{RD^*})]$. It is shown for the fork statement that the initial heap state of forked processes satisfies **Consistency of Tokens**, **Uniqueness of Access** and **Derived Tickets**. It is stated that for contracts Q_C is obtained by a valid execution from the initial heap state. This valid execution implies preservation of **Consistency of Tokens**, **Uniqueness of Access** and **Derived Tickets**, thus Q_C satisfies these properties. Furthermore after having dropped all process identifier ($[\downarrow t']$ for all $t' \in \mathit{Var}'_{\text{process}}$) it can be assured that $\mathit{Tok}(Q_1) = \emptyset$ which is therefore consistent. Additionally since from Q_1 to Q_2 only edges are removed it follows that $\mathit{Tok}(Q_2) = \emptyset$ which is also consistent. Also **Uniqueness of Access** is preserved for Q_1 and Q_2 because all *BasePerms* can only alter from *WR* to *WR** or from *RD* to *RD** for every $[\downarrow t]$ transformation. Therefore because Q_C satisfies **Uniqueness of Access** so do Q_1 and especially Q_2. Also, because those edges that preserve an unstarred *BasePerm* through the dropping of all process identifier must have had an empty *PermSet* before and therefore the permission did not change from Q_C. Therefore Q_1 inherently satisfies **Derived Tickets** and because Q_2 is a subgraph of Q_1 it does too.

Distinguish two cases in the following: firstly $E^{Q_1}_{RD^*} \neq \emptyset$, which implies that $H[\downarrow T_t] \xRightarrow{N_{T_t} \to Q_2} H'$. Then H' satisfies **Consistency of Tokens** because the token T_t is removed from the token set (by successively dropping $t' \in T_t$ and replacing the placeholder N_{T_t} by Q_2) and the corresponding process terminated. Furthermore, integrating Q_2 does not add any token (recall that $\mathit{Tok}(Q_2) = \emptyset$).

Uniqueness of Access can be assured because Q_C satisifes **Uniqueness of Access** and thus, integrating those edges with *BasePerm WR* or *WR** into H hands over the write ticket to H. The write ticket was unique before and since the process that hands the write ticket over terminated it is unique afterwards.

For **Derived Tickets** it is noteworthy that it cannot be guaranteed that all derived access tickets are completely returned since there is at least one edge $e \in E^{Q_1}_{RD^*}$ (thus this edge can potentially be accessed concurrently by a process and this process can never be joined since the refernce to it is lost). Because the initial heap state is only provided with *WR* or *RD* permission this implies a loss of information regarding the accounting in the forked process. This loss of information propagates strictly by dropping the token that identifies the derived access ticket. This ensures a proper accounting since those permissions from which this access ticket was derived are starred (more precisely their *BasePerm*). Furthermore for the "WR-part" **Derived Tickets** is shown above which concludes that H' satisfies **Derived Tickets**.

Secondly, examine the case that $E^{Q_1}_{RD^*} = \emptyset$ such that $H[\downarrow T_t] \xRightarrow{N_{T_t} \to Q_2} H'$: it follows analogous to the first case that H' satisfies **Consistency of Tokens** and **Uniqueness of Access**. For the **Derived Tickets** it is important that the whole access ticket is returned because since Q_C satisfies **Derived Tickets** the empty *PermSet* for the permissions with *RD* as *BasePerm* guarantees that in the execution every granted access

ticket is completely recollected (in other words there is no process directly or indirectly[7] forked from the joined process that can access these edges). This means the granted access ticket (represented as *RD* permissions in the initial heap state) are completely returned and it follows, because the "WR-part" satisfies **Derived Tickets** that H' satisfies **Derived Tickets**. □

Concludingly, this induction yields data race freedom by the following argument: Because proper accounting for all edges with a permission that has a *BasePerm* of *WR* or *RD* is guaranteed (**Derived Tickets**) it follows that any derived access ticket from this permission restricts to read access (by the definition of the transition rules). A permission of *WR* guarantees exclusive access because it meets the condition for proper accounting of **Derived Tickets**, thus its empty *PermSet* ensures that no other process an access this value concurrently. ■

[7] since in the given setup process can only be joined by the process that forked them, there is a distinct predecessor relationship between processes. Indirectly forked refers to any process which is in the transitive closure of the predecessor relationship to the examined process

9 Conclusion

For the future it can be generally assumed that complexity of software increases further. Especially the paradigm of parallel programming becomes more and more important (for example in the relatively new programming language *Go* which offers "explicit support for concurrent programming." [5]). In this work a permission model was added to the already established analysing technique of representing pointer structures as hypergraphs. This allows analysing programs with parallel execution and therefore a programming language was introduced which supports simple pointer manipulation as well as parallel execution by fork and join statements. The semantics were appropiatly defined in terms of hypergraph transformations. As long as valid contracts for the different programs that can be forked are provided the data race freedom for the analysed states is proven. But the defined permission model is actually meant as basis for a larger framework which allows computation of those contracts in order to avoid defining those by hand which is error-prone. Thus the presented analysis is meant for providing a basis that allows further work with the framework (as presented in Section 9.1). Other contributions regard the applied abstraction by hyperedge replacement grammars, namely computation of reachability and especially the compatibility of hyperedge replacement with the permission model as shown in the proof of overapproximation of the transition relation.

9.1 Future Work

After having shown that analysis based on contracts yields data race freedom and is therefore a viable approach on parallel execution of processes it is of special interest how to obtain such contracts. Therefore it is open for future research to compute these contracts automatically. Due to the similarity of contracts in [11] and the presented work the same approach of *fixpoint iteration* to compute contracts can be explored for the presented setup. Additionally integrating the presented permission model into the already existent tool Juggrnaut is of interest in order to collect experimental data of the efficency of this approach for actual problems. For permission accounting in separation logic there are approaches to abstract from actual permission models [3, 4] which opens permission accounting to be explored with various possible permission models. Working on such an abstraction for heap representation with hypergraphs opens this approach to choosing fitting permission models for different situations. Also discussed in [3] is another concept of sharing resources between processes: *conditional critical regions*. These are parts of the program that are connected with a shared resource. This shared resource can be aquired (which grants exclusive access), operated on and finally released. This describes a mechanism of programming for parallel execution known as *monitor* [9] which is e.g. part of the programming language Ada95 [17, p. 163ff]. A possible approach of integrating this concept for the presented permission model is to introduce a "ghost process" which is joined by aquiring and forked by release of the resource. For separation logic *invariants* are used to describe conditional critical resources [3], which can possibly applied to hypergraph representation by representing the resource as single nonterminal and ensuring that this nonterminal describes all possible states of the resource. Also

connected to this is considering forked processes as heap objects themselves. Because this is closer to the actual implementation of fork and join in programming languages e.g. Java [1]. This would additionally allow to provide processes as parameter for procedure calls of further fork statements. Moreover [1] allows for forked process to be joined by various processes in order to obtain a part of the permissions of the postcondition, which could possibly adapted for the representation of hypergraphs as well.

Regarding abstraction and abstract contracts it is possible that these contracts are far too rigorous for their alternable sets. Consider therefore a process that might change the right subtree of a provided binary tree but leaves the left subtree as is. Abstraction in the context of a *HRG* as presented on page 31 might cause indistinguishability between the subtrees and causes the process to demand write access on both. A possible approach on this might be to add to contracts the information that applying some concretisation steps might yield considerably finer demands.

References

[1] Afshin Amighi et al. “Permission-Based Separation Logic for Multithreaded Java Programs”. In: *CoRR* abs/1411.0851 (2014).

[2] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008.

[3] Richard Bornat et al. “Permission Accounting in Separation Logic”. In: *Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’05. ACM, 2005, pp. 259–270.

[4] Cristiano Calcagno, Peter W. O’Hearn, and Hongseok Yang. “Local Action and Abstract Separation Logic”. In: *Proceedings of the 22Nd Annual IEEE Symposium on Logic in Computer Science*. LICS ’07. IEEE Computer Society, 2007, pp. 366–378.

[5] Google. *Golang Specification*. [Online; accessed 11-August-2015]. 2015. URL: `https://golang.org/ref/spec#Introduction`.

[6] Jonathan Heinen et al. “Juggrnaut: Using Graph Grammars for Abstracting Unbounded Heap Structures”. In: *Formal Methods in System Design* (2015). In press.

[7] Jonathan Heinen et al. “Verifying Pointer Programs Using Graph Grammars”. In: *Science of Computer Programming* 97, Part 1 (2015). Special Issue on New Ideas and Emerging Results in Understanding Software, pp. 157 –162.

[8] Stefan Heule et al. “Abstract Read Permissions: Fractional Permissions without the Fractions”. In: *Verification, Model Checking, and Abstract Interpretation*. Ed. by Roberto Giacobazzi, Josh Berdine, and Isabella Mastroeni. Vol. 7737. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, pp. 315–334.

[9] C. A. R. Hoare. “Monitors: An Operating System Structuring Concept”. In: *Commun. ACM* 17.10 (1974), pp. 549–557.

[10] Christina Jansen, Florian Göbe, and Thomas Noll. “Generating Inductive Predicates for Symbolic Execution of Pointer-Manipulating Programs”. In: *Graph Transformation*. Ed. by Holger Giese and Barbara König. Vol. 8571. Lecture Notes in Computer Science. Springer International Publishing, 2014, pp. 65–80.

[11] Christina Jansen and Thomas Noll. “Generating Abstract Graph-Based Procedure Summaries for Pointer Programs”. In: *Graph Transformation*. Ed. by Holger Giese and Barbara König. Vol. 8571. Lecture Notes in Computer Science. Springer International Publishing, 2014, pp. 49–64.

[12] Christina Jansen and Thomas Noll. “Thread Modular Analysis”. Draft.

[13] Christina Jansen et al. “A Local Greibach Normal Form for Hyperedge Replacement Grammars”. In: *Language and Automata Theory and Applications*. Ed. by Adrian-Horia Dediu, Shunsuke Inenaga, and Carlos Martín-Vide. Vol. 6638. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2011, pp. 323–335.

[14] Peter W. O'Hearn. "A Primer on Separation Logic (and Automatic Program Verification and Analysis)". In: *Software Safety and Security - Tools for Analysis and Verification*. 2012, pp. 286–318.

[15] Peter W. O'Hearn. "Resources, Concurrency, and Local Reasoning". In: *Theor. Comput. Sci.* 375.1-3 (2007), pp. 271–307.

[16] Grzegorz Rozenberg, ed. *Handbook of Graph Grammars and Computing by Graph Transformation: Volume I. Foundations*. World Scientific Publishing Co., Inc., 1997, pp. 95–156.

[17] S. Tucker Taft and Robert A. Duff, eds. *Ada 95 Reference Manual, Language and Standard Libraries, International Standard ISO/IEC 8652: 1995(E)*. Lecture Notes in Computer Science. Springer.