Generating Inductive Predicates for Symbolic Execution of Pointer-Manipulating Programs

Christina Jansen and Florian Göbe and Thomas Noll
The publications of the Department of Computer Science of RWTH Aachen University are in general accessible through the World Wide Web.

http://aib.informatik.rwth-aachen.de/
Generating Inductive Predicates for Symbolic Execution of Pointer-Manipulating Programs

Christina Jansen
Software Modeling and Verification Group
RWTH Aachen University, Germany
http://moves.rwth-aachen.de/

Florian Gobe
Embedded Software Laboratory
RWTH Aachen University, Germany
http://embedded.rwth-aachen.de/

Thomas Noll
Software Modeling and Verification Group
RWTH Aachen University, Germany
http://moves.rwth-aachen.de/

Abstract—Separation Logic (SL) is an extension of Hoare Logic that supports reasoning about pointer-manipulating programs. It employs inductively-defined predicates for specifying the (dynamic) data structures maintained at runtime, such as lists or trees. To support symbolic execution, SL introduces abstraction rules that yield symbolic representations of concrete heap states. Whenever concrete program instructions are to be applied, so-called unrolling rules are used to locally concretize the symbolic heap. To enable automatic generation of a complete set of unrolling rules, however, predicate definitions have to meet certain requirements, which are currently only implicitly specified and manually established.

To tackle this problem, we exploit the relationship between SL and hyperedge replacement grammars (HRGs). The latter represent (abstracted) heaps by hypergraphs containing placeholders whose interpretation is specified by grammar rules. Earlier work has shown that the correctness of heap abstraction using HRGs requires certain structural properties, such as increasingness, which can automatically be established. We show that it is exactly the Separation Logic counterparts of those properties that enable a direct generation of abstraction and unrolling rules from predicate definitions for symbolic execution.

Technically, this result is achieved by first providing formalisations of the structural properties in SL. We then extend an existing translation between SL and HRGs such that it covers all HRGs describing data structures, and show that it preserves these structural properties.

Index Terms—Heap Abstraction, Separation Logic, Hyperedge Replacement Grammars, Predicate Generation

I. INTRODUCTION

As software often heavily relies on pointers, understanding this concept is indispensable to understanding software. Prominent examples of pointer usage are dynamic data structures such as doubly-linked lists, nested lists, trees, and so forth. While pointers are one of the most common sources of bugs in software [1], those errors are often difficult to trace. Thus automated analysis techniques are of great assistance. However, the problem is highly non-trivial as the presence of dynamic data structures generally gives rise to an unbounded state space of the program to be analysed. A common approach is to apply abstraction techniques to obtain a finite representation.

In this paper we investigate the relationship between two such techniques. The first, introduced in Sect. III, is based on Separation Logic (SL) [2], an extension of Hoare Logic that supports local reasoning in disjoint parts of the heap. It employs inductively defined predicates for specifying the (dynamic) data structures maintained at runtime, such as lists or trees. To support symbolic execution, SL introduces abstraction rules that yield symbolic representations of concrete heap states. Whenever concrete program instructions are to be applied, so-called unrolling rules expose fields of abstracted heap parts on demand to enable the application of the respective operational rule. This yields a (local) concretisation of the symbolic heap [3]. More details are provided in Sect. IV.

Both abstraction and unrolling rules are dependent on the predicates. This, however, raises the question of the appropriateness of predicate definitions to support the (automated) construction of such rules. Current literature largely ignores this problem; it addresses specific settings such as lists or trees but does not provide general conditions under which a complete set of abstraction and unrolling rules can be generated systematically. Rather, such rules have to be developed manually.

As it turns out, similar conditions are well-studied for the second approach, which employs a graph-based representation of heaps supporting both concrete and abstract parts. The latter is realised by deploying hyperedges, i.e., edges that connect an arbitrary number of nodes, and that are labelled with nonterminal symbols. They are interpreted as placeholders for data structures that are specified by hyperedge replacement grammars (HRGs), i.e., sets of context-free production rules. As will be shown in Sect. V, abstraction and concretisation are implemented by applying production rules in backward and forward direction, respectively. To ensure the correctness and practicability of this procedure, the grammars are automatically transformed such that they exhibit certain structural properties, which are defined in Sect. VI.

The motivation of our work is to exploit these results on structural properties and the corresponding HRG transformations to support automated predicate generation for symbolic execution using the SL framework. These predicates should exhibit all properties necessary to soundly specify abstraction and unrolling as reverse operations. To this aim, we first provide logical formalisations of the structural properties and argue why they enable the direct applicability of predicate definitions for symbolic execution. More concretely, we show that they allow to automatically compile proper abstraction and concretisation rules (such as list collapsing and unrolling operations) from the predicate definitions. We then relate the
SL properties to those introduced for HRGs in Sect. V. In Sect. VII, we then extend a semantics-preserving translation between (a fragment of) SL and HRGs as established earlier in [4] such that it covers all HRGs describing data structures, and show that it preserves the structural properties.

Altogether, these achievements provide the formal basis of an integrated framework for heap abstraction that combines the best of both worlds:

1) data structures are specified using the intuitive formalism of HRGs;
2) the structural properties to ensure correctness of abstraction are automatically established;
3) SL predicate definitions are generated by the property-preserving mapping; and
4) symbolic execution is performed using abstraction and unrolling rules derived from the predicate definitions.

The last step can be assisted by a number of tools that support SL reasoning, such as Heap-Hop [5], jStar [6], Predator [7], Smallfoot [8], SpaceInvader [9], Thor [10], and Xvisa [11]. The paper concludes with some final remarks in Sect. VIII.

Related Work: Without giving further details we mention important alternative approaches to heap verification: shape analysis using three-valued logic [12], regular tree automata [13], and pattern-based graph abstraction [14].

The research that is closest to ours is described in [4], [5]. It establishes the correspondence between a restricted subset of HRGs and a fragment of SL, and shows that SL is actually more expressive than HRGs. Our work extends this approach by considering heap nodes with more than two selector fields. Moreover, it strengthens the correspondence result by providing characterisations of the structural properties mentioned before for both formalisms, and by showing that they are preserved by the translations in both directions. The work of Dodds and Plump is inspired by [16], which gives a semantics to graph grammars my mapping them to SL formulae. However, these grammars only allow to define tree data structures, and the translation is only one-way, which does not allow to derive any correspondence results.

In a more general setting, Monadic Second-Order Logic was used to characterise the graph translations defined by such grammars [17], [18] (although less with the goal of program verification). In particular, [19] states that for an HRG and a monadic second-order formula it is decidable whether the language generated by the grammar contains graphs described by the formula.

II. PRELIMINARIES

The following notations will be used throughout the paper. Given a set $S$, $S^*$ is the set of all finite sequences (strings) over $S$, including the empty sequence $\varepsilon$. For $s \in S^*$, the length of $s$ is denoted by $|s|$, the set of all its elements by $\{s\}$, and by $s(i)$ ($i \in [1,|s|]$) we refer to the $i$-th element of $s$. Given a tuple $t = (A, B, C, \ldots)$, we write $A_t, B_t, \ldots$ etc. for the components if their names are understood from the context. We represent a function $f : A \rightarrow B$ by its set of mappings $\{x \mapsto y \mid \exists x \in A, y \in B : f(x) = y\}$ and write $f_B$ to denote the empty function, i.e. $f_B = \emptyset$. Function $f : A \rightarrow B$ is lifted to sets $f : 2^A \rightarrow 2^B$ and to sequences $f : A^* \rightarrow B^*$ by pointwise application. The $\sqcup$ operator denotes the disjoint union of two functions.

III. SEPARATION LOGIC

SL is designed to reason about heaps. It supports the specification of inductively-defined predicates that allow to describe heap shapes. Thus it is particularly interesting for symbolic execution of pointer-manipulating programs, which is studied in e.g. [3]. In this paper our main focus lies on the translation from HRGs to SL predicates and the associated structural properties. Therefore we keep the presentation of SL short and refer to [2] for a detailed introduction.

In SL a heap is understood as a set of locations connected via references. We assume the set of heap locations $\text{Loc} := \mathbb{N}$. To express that a location contains a null reference, we introduce the set $\text{Elem} := \text{Loc} \cup \{\text{null}\}$. A heap is a (partial) mapping $h : \text{Loc} \rightarrow \text{Elem}$. The set of all heaps is denoted by $\text{He}$. SL features logical variables, denoted by $\text{Var}$, that range over heap locations. Thus an SL formula is interpreted over a (heap, interpretation)-pair.

**Definition 3.1 (Interpretation):** An interpretation is a (partial) mapping $i : \text{Var} \rightarrow \text{Elem}$. We denote the set of all interpretations by $\text{Int}$.

We consider the heap to contain objects. Each object has a number of references to other objects, represented by a finite set $\Sigma$ of selectors. In the heap representation we assume these references to reside in successive locations respecting a canonical order and thus reserve $|\Sigma|$ successive locations for each object. We use the notation $cn(s), s, s' \in \Sigma$ to denote the ordinal of selector $s$, where $0 \leq cn(s) < |\Sigma|$, i.e., under interpretation $i$ the location of selector $s$ of an object $x$ is $i(x) + cn(s)$. To simplify notation, we assume w.l.o.g. that for a heap containing $n$ objects, locations $1, 2, \ldots, n$ $|\Sigma|$ are used, i.e. we do not distinguish between heaps that differ only in their location numbering. An interpretation $i \in \text{Int}$ is safe if $img(i) \subseteq \{1,|\Sigma|+1,|\Sigma|+2,\ldots\} \cup \{\text{null}\}$. In the following we assume all interpretations to be safe.

We consider a restricted set of SL formulae where negation $\neg$, true and conjunction $\wedge$ in subformulae speaking about the heap is disallowed. This restriction is necessary and due to the fact that HRGs are less expressive than SL [15].

**Definition 3.2 (Syntax of SL):** Let $\text{Pred}$ be a set of predicate names. The syntax of SL is given by:

$$E ::= x \mid \text{null}$$

$$P ::= x = y \mid P \wedge P \quad \text{pure formulae}$$

$$F ::= \text{emp} \mid x.s \rightarrow E \mid F * F \mid [F] \quad \text{heap formulae}$$

$$\exists x : F \mid \sigma(E_1, \ldots, E_n)$$

$$S ::= F \mid S \vee S \mid P \wedge S \quad \text{SL formulae}$$

where $x, y \in \text{Var}$, $s \in \Sigma$ and $\sigma \in \text{Pred}$. A heap formula of the form $x.s \rightarrow E$ is called a pointer assertion. SLF denotes the set of all SL formulae.
To simplify notation, we introduce the following abbreviations. Given \( \phi \in \text{SLF} \), the set \( \text{Var}(\phi) (\text{FV}(\phi)) \) collects all (free) variables of \( \phi \). If \( \text{FV}(\phi) = \emptyset \), then \( \phi \) is called closed. Atomic(\( \phi \)) denotes the set of all atomic subformulas of \( \phi \), that is, those of the form \( x = y \), \( \operatorname{cmp} \), \( x.s \mapsto E \) and \( \sigma(E,...,E) \). The predicate calls in \( \phi \) are given by \( \text{pred}(\phi) = \{ \sigma(x_1, \ldots, x_n) \in \text{Atomic}(\phi) \mid \sigma \in \text{Pred}, x_1, \ldots, x_n \in \text{Var} \cup \{\emptyset\} \} \). If \( \text{pred}(\phi) = \emptyset \), then \( \phi \) is called primitive.

Predicate definitions are collected in environments.

**Definition 3.3 (Environment):** A predicate definition for \( \sigma \in \text{Pred} \) is of the form \( \sigma(x_1, \ldots, x_n) := \sigma_1 \lor \ldots \lor \sigma_m \) where \( m, n \in \mathbb{N}, x_1, \ldots, x_n \in \text{Var} \) are pairwise distinct, \( \sigma_1, \ldots, \sigma_m \) are heap formulas, \( \text{null} \) does not occur in \( \sigma_1 \lor \ldots \lor \sigma_m \), and \( \text{FV}(\sigma_j) = \{x_1, \ldots, x_n\} \) for each \( j \in [1, m] \). We call \( \sigma_1 \lor \ldots \lor \sigma_m \) the body of \( \sigma \). A set of definitions with distinct predicates is called an environment. The set of all environments is denoted by \( \text{Env} \).

**Example 3.4:** A predicate named \( \text{dll} \) for specifying doubly-linked lists can be defined as follows: \( \text{dll}(x_1, x_2) := \sigma_1 \lor \sigma_2 \) with two disjunctions \( \sigma_1 := x_1 \leadsto x_2 \lor x_2 \leadsto x_1 \) and \( \sigma_2 := \exists r : x_1.n \mapsto r \land r.p \mapsto x_1 \land \text{dll}(r, x_2) \). The corresponding environment is the singleton set \( \{\text{dll}(x_1, x_2) := \sigma_1 \lor \sigma_2\} \).

The (weak) restriction that \( \text{null} \) is not allowed to appear in the predicate body can again be traced back to HRGs, where a similar restriction will be introduced for the sake of simplification. Note that while \( \text{null} \) is disallowed in environments, its usage in \( \text{SLF} \) formulas is admitted. The requirement that each disjunct has to refer to all parameter variables ensures that the HRG resulting from the translation (cf. Sect. VII) is ranked. This assumption does not impose a semantic restriction as each SL formula can be translated into an equivalent one by introducing new predicates fulfilling this property. The construction is similar to establishing an equivalent, ranked HRG as shown in [20].

For \( \Gamma \in \text{Env} \), \( \text{Pred}_\Gamma \) is the set of all \( \sigma \in \text{Pred} \) with a definition in \( \Gamma \). Note that we assume predicate definitions to be in disjunctive normal form. As every SL formula can be translated into this normal form, this is no true restriction [21].

Next we define the predicate interpretation \( \eta_\Gamma \) based on an environment \( \Gamma \in \text{Env} \). Intuitively, for a predicate \( \sigma \) defined in \( \Gamma \), \( \eta_\Gamma(\sigma) \) contains all pairs of locations representing the arguments to \( \sigma \), and heaps that fulfill \( \sigma(x_1, \ldots, x_n) \).

**Definition 3.5 (Predicate Interpretation):** The predicate interpretation \( \eta_\Gamma \) of an environment \( \Gamma \in \text{Env} \) is the least fixpoint of the function \( f_\Gamma : (\text{Pred} \rightarrow 2^\text{loc} \times \text{Hep}) \rightarrow (\text{Pred} \rightarrow 2^\text{loc} \times \text{Hep}) \) w.r.t. \( \subseteq \) where \( f_\Gamma(\eta)(\sigma) \) is given by

\[
\{ (l, h) \mid h, \{ x_1 \mapsto l(1), \ldots, x_n \mapsto l(n) \}, \eta \models \sigma_1 \lor \ldots \lor \sigma_m \}
\]

for \( \sigma(x_1, \ldots, x_n) := \sigma_1 \lor \ldots \lor \sigma_m \in \Gamma \).

Here, the satisfaction relation \( \models \) is determined as follows.

**Definition 3.6 (Semantics of SL):** Given \( \Gamma \in \text{Env} \), the relation \( \models \) is inductively defined by

\[
\begin{align*}
\begin{array}{ll}
\text{h, i, } \eta_\Gamma \models \text{emp} & \iff \text{dom}(h) = \emptyset \\
\text{h, i, } \eta_\Gamma \models x.s \mapsto \text{null} & \iff \text{dom}(h) = \{i(x) + cn(s)\}, \text{h}(i(x) + cn(s)) = \text{null} \\
\text{h, i, } \eta_\Gamma \models x.s \mapsto y & \iff \text{dom}(h) = \{i(x) + cn(s)\}, \text{h}(i(x) + cn(s)) = i(y) \\
\text{ex. } h_1, h_2 : h_1[ \| h_2 = h, h_1, i, \eta_\Gamma \models \phi_1 \land \phi_2 & \iff \text{ex. } v \in \text{Loc} : h_1[ i(x) \mapsto v], \eta_\Gamma \models \phi \\
\text{ex. } v \in \text{Loc} : h_1[ i(x) \mapsto v], \eta_\Gamma \models \phi & \iff \text{ex. } v \in \text{Loc} : h_1[ i(x) \mapsto v], \eta_\Gamma \models \phi \\
\text{h, i, } \eta_\Gamma \models x = y & \iff \eta(x) = \eta(y) \\
\end{array}
\end{align*}
\]

We call two SL formulae equivalent \( (\equiv) \) if for any fixed predicate interpretation they are fulfilled by the same set of heap/interpretation pairs.

IV. PREDICATE PROPERTIES FOR SYMBOLIC EXECUTION

As seen in the previous section, SL provides a formalism for analysing pointer-manipulated programs. Due to dynamic allocation and deallocation, the state space of such programs is potentially infinite and so is the necessary number of formulae describing those states. An approach to resolve this issue is abstraction.

A. Abstraction and Unrolling

Abstraction operations in SL are typically specified by abstraction rules, which define transformations on SL formulae.

**Example 4.1:** For example, consider the environment from Ex. 3.4 defining a doubly-linked list structure. The abstraction rule \( \text{abs}_1 := \phi \land x_1.n \mapsto x_2 \land x_2.p \mapsto x_1 \iff \phi \land \text{dll}(x_1, x_2) \) defines the transformation of \( \phi \land x_1.n \mapsto x_2 \land x_2.p \mapsto x_1 \), where \( \phi \) is some arbitrary SL formula, such that the subformula describing a list of length \( 1 \) is abstracted to a list of arbitrary length.

For the purpose of automated symbolic execution of programs, usually abstraction rules are either pre-defined or can be given by the user, as e.g. tools like jStar allow [6]. To ensure soundness, these rules must result from valid implications.

**Example 4.2:** Consider again the environment and abstraction rule given in Ex. 3.4. Rule \( \text{abs}_1 \) is sound as clearly \( \phi \land x_1.n \mapsto x_2 \land x_2.p \mapsto x_1 \) implies \( \phi \land \text{dll}(x_1, x_2) \) whereas \( \phi \land \text{dll}(x_1, x_2) \iff \phi \land x_1.n \mapsto x_2 \land x_2.p \mapsto x_1 \) is no over-approximation and thus not a sound abstraction rule.

The application of abstraction rules yields heap representations that are partially concrete and partially abstract. For example, the heap containing a doubly-linked list of length two represented by the SL formula \( \exists x_1, x_2, x_3 : x_1.n \mapsto x_2 \land x_2.n \mapsto x_3 \land x_3.p \mapsto x_2 \land x_2.p \mapsto x_1 \) can be abstracted to \( \exists x_1, x_3 : x_1.n \mapsto x_2 \land x_2.p \mapsto x_1 \land \text{dll}(x_2, x_3) \) using the abstraction rule from above.

To generate the abstract state space of a pointer-manipulating program, symbolic execution is performed. The
key idea behind symbolic execution using SL is that whenever fields hidden in an abstracted part are to be accessed during execution, they are exposed first. This can e.g. be achieved by performing frame inference using a theorem prover. Thus whenever in a state $\phi$, field $s$ of object $x$ is accessed, the prover is requested to provide all formulae of the form $\phi' \ast x.s \rightarrow y$ which entail $\phi$. This step is referred to as unrolling. Note that for correctness it is crucial that a complete set of such formulae is considered. Execution is then resumed and performed on this set, resulting in an over-approximated state space.

Instead of specifying abstraction rules manually and relying on the theorem prover’s frame inference procedure during the unrolling step, abstraction and unrolling rules can directly be generated from the predicate definitions. The basic idea is to construct an abstraction rule for each disjunct of the predicate body stating that this disjunct can be abstracted to a call of the predicate. Unrolling is performed by replacing a predicate call by each disjunct of its predicate definition. For instance, concretisation of the formula $3\forall r_1, r_2 : dll(r_1, r_2)$ at $r_1$ yields the following set of formulae (which is a safe over-approximation of the original formula): $\{3\forall r_1, r_2 : r_1.n \rightarrow r_2 \ast r_2.p \rightarrow r_1, 3\forall r_1, r_2 : r : r_1.n \rightarrow r \ast r.p \rightarrow r_1 \ast dll(r, r_2)\}$, where the two formulae result from unrolling of $dll(r_1, r_2)$ by the first and second disjunct of its predicate body, respectively. To ensure the generated rules are sound, predicate definitions have to exhibit certain structural properties, which we will define in the following section. In further sections we will show that these properties impose no real restrictions and how they can be established. The main advantage of deriving abstraction and unrolling rules from such predicate definitions is that proving soundness of those rules and relying on frame inference is not necessary any more. Besides this, we will introduce further properties improving the automated symbolic execution approach that are easily ensured by automated rule generation.

B. Structural Properties

In the following we introduce four structural properties on SL predicates that guarantee soundness and practicability of heap abstractions by formulating syntactical restrictions on the predicate definitions contained in an environment. These properties do not entail any restriction on the data structures defined by the predicates since, as we will see later, predicate definitions can automatically be transformed into ones satisfying these requirements. This transformation procedure employs the relation between SL and HRG$s$, which will be investigated in Sect. VII. Moreover we introduce a fifth property, called confluence under abstraction, that is useful because it provides unique abstraction normal forms (but is not necessary for ensuring soundness of heap abstraction). The property differs from the others in that it cannot be formulated as a structural property. Moreover while there exists a decision procedure, we are not aware of any algorithm that generally establishes confluence.

In Sect. VI, we give an equivalent definition for each of these five properties on the graph grammar side and show that the translation described in Sect. VII preserves them.

We start by considering the two structural properties, productivity and typedness, that are not immediately necessary for the approach to work correctly but will pave the way for the more intriguing and mandatory properties, increasingness and local concretisability, that will be discussed later on.

1) Productivity: A predicate is productive if it can iteratively be unrolled into a formula without predicate calls, i.e., one describing a fully concrete heap. Consider an environment with non-productive predicates. Then there exist SL formulae describing abstract heaps, although no concrete heap can satisfy these. This follows directly from the fixpoint semantics of predicate calls. Thus they represent program states a concrete program execution cannot end up in. Instead of detecting and discarding such heap configurations when they appear, we restrict environments to productive predicates in advance.

Cast as a syntactic property, a predicate is productive if one of its disjuncts contains no predicate calls (i.e., describes a concrete heap) or if each called predicate is already known to be productive.

Definition 4.3 (Productivity): The set of productive predicates is the least subset of Pred such that for each element there exists a disjunct $\sigma$, in its predicate body such that either $\sigma$ is primitive or all predicates occurring in $pred(\sigma)$ are productive. An environment is productive if each of its defined predicates is productive.

2) Typedness: Applying abstraction and unrolling rules in an alternating fashion may yield formulae that do not represent valid heaps. To illustrate this, consider the environment for doubly-linked lists from Ex. 3.4. Now assume we want to modify this grammar such that it generates partly singly- and partly doubly-linked lists. Thus we modify the environment containing the dll-predicate as follows: \(\{list(x_1, x_2) : dll(x_1, x_2) \land x_1.n \rightarrow x_2 \land \exists r : x_1.n \rightarrow r \ast list(r, x_2)\}\).

Given an SL formula $\exists r_1, r_2, r_3 : r_1.n \rightarrow r_2 \ast r_2.p \rightarrow r_1 \ast list(r_2, r_3)$, one could abstract the subformula $r_1.n \rightarrow r_2$ by means of the second disjunct of list, which yields $\exists r_1, r_2, r_3 : list(r_1, r_2) \ast r_2.p \rightarrow r_1 \ast list(r_2, r_3)$. To ensure that exposing the pointers around $r_1$ in a subsequent concretisation step is sound, we have to unroll with all possible disjuncts of predicate list. One possible unrolling is the following: $\exists r_1, r_2, r_3 : list(r_1, r_2) \ast r_2.p \rightarrow r_1 \ast list(r_2, r_3) \implies \exists r_1, r_2, r_3 : r_1.n \rightarrow r_2 \ast r_2.p \rightarrow r_1 \ast list(r_2, r_3)$. This formula is unsatisfiable, as the subformula $r_2.p \rightarrow r_1$ appearing twice is separated by means of the $\ast$-operator. While in this case it is easy to see that no heap could satisfy the formula, it could also happen that this fact is hidden through variable equality or similar.

Typedness ensures that such unsatisfiable formulae cannot arise due to abstraction and unrolling. Intuitively we fix the type, i.e., the set of pointers, for each (predicate, parameter)-pair \((pred, x)\) requiring that for each disjunction in the definition of this predicate exactly this set of pointers is derivable at variable $x$. E.g. the predicate list of the beforementioned example is not typed, as $type(x_1) = \emptyset$ and $type(x_2) = \{p\}$ of the disjuncts contradicts each other.
Note that for any reasonable definition of typedness of SL formulae it is necessary to assume that the interpretation of parameters \(x_1, \ldots, x_n\) is pairwise distinct. This enables us to isolate the selectors derived at \(x_i\) (\(i \in [1, n]\)).

**Definition 4.4 (Typedness):** Let \(\Gamma \in Env\). A predicate definition \(\sigma(x_1, \ldots, x_n) := \sigma_1 \lor \ldots \lor \sigma_m \in \Gamma\) is typed if for each \(x_j\) (\(j \in [1, n]\)) there exists a set \(\text{type}(\sigma_j) \in \Sigma\) such that, for all \(h \in \text{He}\) and \(i \in \text{Int}\) with \(i(x_j) \neq i(x_k)\) (\(j \neq k\)) and \(h, i, \eta_i \models \sigma(x_1, \ldots, x_n), \text{type}(\sigma_j) = \{s \in \Sigma \mid i(x_j) + cn(s) \in \text{dom}(h)\}\). An environment is typed if each of its definitions is.

3) **Increasingness:** When applying abstraction rules, e.g., those automatically generated from predicate definitions, one has to make sure that abstraction actually terminates, that is, no infinite abstraction sequences are possible. The latter can occur for an environment of the form \(\{\sigma(x_1, \ldots, x_n) := \sigma(x_1, \ldots, x_n)\}\). When repeatedly abstracting with \(\sigma\)’s only disjoint, abstraction (and thus symbolic execution) does not terminate. Again, this behaviour is ruled out beforehand by a syntactic restriction on environments, called increasingness. It requires the disjuncts of each predicate definition to either contain a pointer assertion or to be “bigger” (in terms of occurrences of variables and predicate calls) than the left-hand side of the predicate definition.

**Definition 4.5 (Increasingness):** A predicate definition \(\sigma(x_1, \ldots, x_n) := \sigma_1 \lor \ldots \lor \sigma_m\) is increasing if for all \(\sigma_j\) (\(j \in [1, m]\)) it holds that \(\sigma_j\) contains a pointer assertion or \#pred(\(\sigma_j\)) + \#Var(\(\sigma_j\)) > \(n + 1\), where \#pred(\(\sigma_j\)) is the number of predicate calls in \(\sigma_j\) and \#Var(\(\sigma_j\)) the number of predicate variables occurring in \(\sigma_j\). An environment is increasing if all of its predicate definitions are increasing.

4) **Local Concretisability:** One of the key ideas behind symbolic execution of pointer-manipulating programs is that each program statement only affects a local portion rather than the whole heap. Thus if a program statement operates on an abstract heap part, we can expose pointers locally such that the statement can be executed as if the complete heap was concrete. This necessitates that abstraction can always and everywhere be reversed by unrolling using finitely many rule applications. This is not ensured by every environment, as illustrated by the following example.

**Example 4.6:** Consider again the environment providing a predicate definition for doubly-linked lists from Ex. 3.4 and the SL formula \(\phi := \exists r : \text{head} \cdot n \rightarrow r \forall r \cdot \text{pred} \rightarrow \text{head} \cdot \text{dll}(r, \text{tail})\) where \(\text{head}\) and \(\text{tail}\) are program variables. Assume a program that traverses the list from its tail (the last element of the list). Then the pointer of this element is abstracted or, put differently, “hidden” in the predicate call \(\text{dll}(r, \text{tail})\). Unrolling yields the two rules \(\text{unroll} \phi := \exists r : \text{head} \cdot n \rightarrow r \forall r \cdot \text{pred} \rightarrow \text{head} \cdot \text{dll}(r, \text{tail})\), and \(\text{dll}(r, \text{tail}) \rightarrow \exists r_1, r_2 : \text{head} \cdot n \rightarrow r_1 \forall r_1, r_2 \cdot \text{head} \cdot r_1, n \rightarrow r_2 \forall r_2 \cdot \text{dll}(r_2, \text{tail})\), where the second formula is locally concrete at the head of the list instead of the tail. Successive unrolling does not resolve this. On the other hand, ignoring the second disjunct of the definition of \(\text{dll}\) guarantees termination of unrolling but is unsound as it may under-approximate the state space of the program.

Therefore we formulate a property which guarantees that for each (predicate, parameter)-pair \((\text{pred}, x_i)\) there exists an environment unrolling a \(\text{pred}\)-predicate call at variable \(x_i\), that is equivalent to the original one, i.e. whose predicate interpretation is equivalent.

**Definition 4.7 (Local Concretisability):** Let \(\Gamma \in Env\) be typed. A predicate definition \(\sigma(x_1, \ldots, x_n) := \sigma_1 \lor \ldots \lor \sigma_m \in \Gamma\) is locally concretisable if, for all \(j \in [1, n]\), there exists a \(\sigma'(x_1, \ldots, x_n) := \bigvee \Phi_j \subseteq \{\sigma_1, \ldots, \sigma_m\}\) such that

1) \(\forall h \in \text{He}, i \in \text{Int}, \eta_i \in \text{Pl} : h, i, \eta_i \models \sigma(x_1, \ldots, x_n) \iff h, i, \eta_i \models \sigma'(x_1, \ldots, x_n)\)

2) \(\forall \phi \in \Phi_j, s \in \text{type}(\sigma_j) : \phi, s \rightarrow y \in \text{Atomic}(\phi)\) for some \(y \in \text{Var} \cup \{\text{null}\}\).

An environment is locally concretisable if all of its predicate definitions are.

For an environment satisfying this property we can directly extract the rules for unrolling a predicate call in a way that guarantees an over-approximation of the program’s state space.

**Example 4.8:** For instance, consider the environment \(\Gamma\) with one predicate \(\text{dll}\) defined by three disjuncts \(\sigma_1 := x_1, n \rightarrow x_2 \star x_2, p \rightarrow x_1, a_2 := \exists r_1 : x_1, n \rightarrow r_1 \forall r_1, p \rightarrow x_2 \star \text{dll}(r_1, x_2)\) and \(\sigma_3 := \exists r_1 : \text{dll}(r_1, x_1) \forall r_1, n \rightarrow x_2 \star x_2, p \rightarrow r_1\). Here we can unroll \(\text{dll}(x_1, x_2)\) at parameter \(x_1\) by replacing with both \(\sigma_1\) and \(\sigma_2\) as described earlier in this section, while unrolling at \(x_2\) employs the disjuncts \(\sigma_1\) and \(\sigma_3\).

5) **Confluence Under Abstraction:** When applying abstraction rules, it can happen that different application orders yield different formulae.

**Example 4.9:** Consider again the locally concretisable environment from Ex. 4.8 and the SL formula \(\exists r : \text{head} \cdot n \rightarrow r \forall r \cdot \text{pred} \rightarrow \text{head} \cdot \text{dll}(r, \text{tail})\). Then abstraction by choosing the first disjunct \(\sigma_1 := x_1, n \rightarrow x_2 \star x_2, p \rightarrow x_1\) results in the formula \(\exists r : \text{dll}(\text{head}, r) \land \text{dll}(\text{tail}, r)\). Another possibility for abstraction is to select the second disjunct \(\sigma_3 := \exists r_1 : x_1, n \rightarrow r_1 \forall r_1, p \rightarrow x_2 \star \text{dll}(r_1, x_2)\), resulting in \(\text{dll}(\text{head}, r)\). Obviously, the resulting formulae are inequivalent and cannot be abstracted further.

If such deviations cannot appear, abstraction always yields unique normal forms. These unique normal forms are highly desired as they imply that under the assumption that the environment is increasing and describes the data structures arising in the program under consideration (except for finitely many local deviations) – the resulting abstract state space is finite. Therefore we introduce confluence under abstraction, which is guaranteed by an environment if the order in which abstractions are applied does not matter, i.e., they all finally yield the same abstract formula.

**Definition 4.10 (Confluence under Abstraction):** \(\Gamma \in Env\) is confluent under abstraction if, for all closed \(\phi \in \text{SLF}\) and abstractions \(\phi \xrightarrow{r_1} \phi_1, \phi \xrightarrow{r_2} \phi_2\), there exist \(\phi_1 \xrightarrow{r} \phi'_1\) and \(\phi_2 \xrightarrow{r} \phi'_2\) such that \(\phi'_1 \equiv \phi'_2\).
V. HYPEREDGE REPLACEMENT GRAMMARS

In the heap abstraction approach based on graph grammars [22], (abstract) heaps are represented as hypergraphs. Hypergraphs are graphs with edges as proper objects, i.e. they can connect arbitrarily many nodes.

Definition 5.1 (Hypergraph): Let \( \Sigma \) be a finite alphabet with ranking function \( \text{rk} : \Sigma \to \mathbb{N} \). A labelled hypergraph (HG) over \( \Sigma \) is a tuple \( H = (V, E, att, lab, ext) \) where \( V \) is a set of or nodes and \( E \) a set of hyperedges, \( att : E \to \mathbb{V}^* \) maps each hyperedge to a sequence of attached nodes, \( lab : E \to \Sigma \) is a hyperedge-labelling function, and \( ext \in \mathbb{V}^* \) a (possibly empty) sequence of pairwise distinct external nodes. For \( e \in E \), we require \( |att(e)| = \text{rk}(lab(e)) \) and let \( \text{rk}(e) = \text{rk}(lab(e)) \). The set of all hypergraphs over \( \Sigma \) is denoted by \( \text{HG} \).

Two HGs are isomorphic if they are identical modulo renaming of nodes and hyperedges. We will not distinguish between isomorphic HGs.

Using the alphabet of selectors introduced in Sect. III, \( \Sigma \), we model (concrete) heaps as HGs over \( \Sigma \) without external nodes. (The latter will be required later for defining the replacement of hyperedges.) Objects are represented by nodes, and selectors by edges of rank two connecting the corresponding object(s), where selector edges are understood as pointers from the first attached object to the second one. We introduce a node \( v_{\text{null}} \) representing null, which is unique to every hypergraph. To represent abstract parts of the heap we use nonterminal edges, which carry labels from an additional set of nonterminals (NTs) \( \mathbb{N} \) (and we let \( \Sigma_N = \Sigma \cup \mathbb{N} \)).

Example 5.2: A typical implementation of a doubly-linked list consists of a sequence of list elements connected by next (\( _n \)) and previous (\( _p \)) pointers. Fig. 1 depicts a hypergraph representation of such a list. The three circles are nodes representing objects on the heap. A shaded circle indicates an external node; its ordinal (i.e., position in \( ext \)) is given by the label. The \( L \)-labelled box represents an NT edge of rank two indicating an abstracted doubly-linked list between the first and second node attached to this edge. Later we will see how these abstract structures are defined. The connections between NT edges and nodes are labelled with their ordinal number. For the sake of readability, selectors (\( _n \) and \( _p \)) are depicted as directed edges.

Note that not every HG over \( \Sigma \) represents a valid heap: e.g., it is necessary that, for every selector \( s \in \Sigma \), every object has at most one outgoing \( s \)-edge, and that external nodes are absent. HGs that satisfy this requirement (cf. Def. A.1) are called heap configurations (HC), and are collected in the set \( \text{HCG} \). Thus \( \text{HCG} \) represents all concrete HCs, i.e., those without nonterminal edges. HCs with external nodes are called extended and are represented by the sets \( \text{HCEG} \) and \( \text{HCE} \) in the abstract and concrete case, respectively.

An NT edge of an HC acts as a placeholder for a heap part of a particular shape. We use hyperedge replacement grammars to describe its possible structure.

Definition 5.3 (Hyperedge Replacement Grammar): A hyperedge replacement grammar (HRG) over an alphabet \( \Sigma \) is a function \( G : N \to \text{HCG} \) with \( |ext_H| = \text{rk}(X) \) for each \( H \in G(X), X \in N \). We call \( X \to H \) a production rule. The set of all HRGs over \( \Sigma \) is denoted by \( \text{HRG} \).

Example 5.4: Fig. 2 specifies an HRG for doubly-linked lists. It employs one NT \( L \) of rank two and two production rules. The right one recursively adds one list element, whereas the left one terminates a derivation.

HRG derivations are defined through hyperedge replacement, i.e. the substitution of an NT edge by the right-hand side of a corresponding production rule. A sample replacement is illustrated in Fig. 3, where the second rule of the grammar in Fig. 2 is applied to the NT edge \( L \) in the upper graph.

Fig. 1. (Extended) heap as hypergraph

Fig. 2. A grammar for doubly-linked lists

Fig. 3. Replacement of \( L \)-edge

Fig. 4. Resulting hypergraph

To this aim, the external nodes of the rule's right-hand side are mapped to the nodes of the upper graph as indicated by the dashed arrows, and the \( L \)-edge is replaced by the rule's right-hand side. The result of this replacement is given in Fig. 4.

Definition 5.5 (HRG Derivation): Let \( G \in \text{HRG} \). \( \mathbb{H}, \mathbb{H}' \in \text{HCG} \), \( K \in G(X) \), \( p = X \to K \), and \( e \in E_H \) with \( \text{lab}(e) = X \). \( H \) derives \( H' \) by \( p \) if \( H' \) results from \( H \) by replacing \( e \) with \( K \). \( H \Rightarrow H' \) if \( H \Rightarrow_{e,p} H' \) for some \( e \in E_H \), \( K \in G(X) \), \( p = X \to K \), and let \( \Rightarrow \) denote the reflexive-transitive closure of this relation.

When establishing a connection between HRGs and SL, external nodes correspond to the parameters of a predicate definition. Thus after applying a predicate definition parameters may or may not reference the same location on the heap. E.g. consider a formula of the form \( \sigma(x_1, x_2) \), which can be fulfilled by a heap with either interpretation \( i(x_1) = i(x_2) \) or \( i'(x_1) \neq i'(x_2) \). This is true for external nodes of a hypergraph as well, as they indicate those nodes that might collapse when replacing an edge with this hypergraph. The context of a hypergraph \( H \) collects all those graphs that result from collapsing arbitrary external nodes of \( H \) (akin to the different interpretations \( i \) and \( i' \)), and is denoted by context\((H)\). The formalization of this concept can be found in Def. A.2.
The definition of HRGs does not include a particular starting graph. Instead, it is introduced as a parameter of the generated language. If this parameter contains external nodes, they are eliminated by considering its context.

**Definition 5.6 (Language of an HRG):** Let $G \in \text{HRG}_{\Sigma N}$ and $H \in \text{HG}_{\Sigma N}$. $L_G(H) = \{K \in \text{HG}_{\Sigma N} \mid \text{context}(H) \Rightarrow K\}$ is the language generated from $H$.

The language of an NT is defined as the language of its handle, a hypergraph consisting of a single hyperedge $e$ attached to $rk(e)$ many distinct nodes only.

**Definition 5.7 (Handle):** Given $X \in N$ with $rk(X) = n$, an $X$-handle is the hypergraph $X^* = (\{v_1, \ldots, v_n\}, \{e\}, \{e \mapsto v_1 \ldots v_n\}, \{e \mapsto X\}, e) \in \text{HC}_{\Sigma N}$.

Moreover we define the $X$-handle over external nodes.

$X^*_{ext} = (\{v_1, \ldots, v_n\}, \{e\}, \{e \mapsto v_1 \ldots v_n\}, \{e \mapsto X\}, v_1 \ldots v_n) \in \text{HC}_{\Sigma N}$.

Thus $L(X^*)$ is the language induced by NT $X$. For $H \in \text{HC}_{\Sigma N}$, $L(H)$ denotes the set of derivable concrete HGs. Note that it is not guaranteed that $L(H) \subseteq \text{HC}_{\Sigma N}$, i.e., $L(H)$ can contain invalid heaps. If this is excluded, $G$ is called a data structure grammar. This property is decidable [23].

**Definition 5.8 (Data Structure Grammar):** $G \in \text{HRG}_{\Sigma N}$ is called a data structure grammar (DSG) if for all $X \in N$, $L(X^*) \subseteq \text{HC}_{\Sigma N}$. We denote the set of all data structure grammars over $\Sigma N$ by $\text{DSG}_{\Sigma N}$.

**VI. HEAP ABSTRACTION GRAMMAR PROPERTIES**

In the heap abstraction approach based on DSGs, program states are described by HCs, which may contain edges labelled with nonterminals. These nonterminal edges act as a placeholder for an abstracted heap part. The structures the placeholders represent are given by DSGs. The state space of the program under consideration is obtained by executing its statements on these configurations. Similarly to symbolic execution using SL, whenever statements have to be executed which are operating on fields that are hidden in abstracted parts of the heap, these fields are exposed beforehand. This operation is referred to as concretisation. In the following we will see that both abstraction and concretisation rules are directly given by the DSG.

**A. Abstraction and Concretisation**

For DSGs the forward application of a production rule yields a more concrete IIC, and certain abstract parts (viz., the NT that is replaced) are concretised. In case more than one rule is applicable, concretisation yields several heaps. For an example concretisation see Fig. 5, where the occurrence of $L$ is either replaced by an empty list (at the bottom, conforming to the first rule), or a list that is extended by one element (at the top, conforming to the second rule). Thus applying concretisation to the heap depicted on the left yields two successor heaps, one for each possible concretisation.

The resulting transition system representing the program behaviour thus over-approximates the transition system in which all heaps are concrete.

The reverse application, i.e., the replacement of an embedding of a rule's right-hand side by its left-hand side, yields an abstraction (denoted by $\Rightarrow$ and $\Rightarrow^*$ for an abstraction sequence, respectively) of a heap. As (forward) rule application is monotonically decreasing with respect to language inclusion, abstraction of a heap configuration yields an over-approximation of the current set of concrete heap configurations.

Note that employing abstraction and concretisation as reverse operations is similar to the idea of generating abstraction and unrolling rules from predicate definitions as described in Sect. IV.

**B. Structural Properties**

Once we use abstraction and concretisation on the state space of a heap-manipulating program, we have to make sure that arbitrary compositions of these operations yield an (abstract) state space that is a safe over-approximation of the original one. We guarantee this by formulating four additional properties for DSGs. DSGs exhibiting these properties are suitable for heap abstraction, and are thus called heap abstraction grammars (HAGs). Notice that the properties that characterise HAGs correspond to the properties given in Sect. IV, which were introduced for automated generation of abstraction and unrolling rules from SL predicate definitions. It is known that the HAG properties do not restrict the expressiveness of the formalism, as every DSG can automatically be transformed into one satisfying these requirements [20]. Additionally we define the notion of backward confluence in connection with HRGs. Similarly to the SL definition it guarantees unique normal forms under abstraction. Again, backward confluence is beneficial, but not needed for the correctness of the heap abstraction approach. While there exists a decision procedure, we are not aware of any completion algorithm that establishes backward-confluence for DSGs.

1) **Productivity:** An NT is productive if its induced language is non-empty. In a DSG with non-productive NTs, there exist abstract heap configurations that cannot be reached by a concrete program run. We can therefore safely remove non-productive NTs and corresponding production rules of a DSG without changing the language [20]. The corresponding procedure is based on the following definition.

**Definition 6.1 (Productivity):** The set of productive NTs is the least set such that for each element $X$ there exists a production rule $X \rightarrow H$ such that either $H \in \text{HC}_{\Sigma N}$ or all contexts of $Z \in N$ such that $\exists e \in E_H : \text{lab}(e) = Z$ are productive. $G \in \text{DSG}_{\Sigma N}$ is productive if all of its NTs are productive.
2) Typedness: Alternating abstraction and concretisation may yield graphs that do not depict a heap, i.e., are not HCs. To illustrate this, consider again the DSG for doubly-linked lists from Fig. 2. Assume again we want to modify this grammar such that it generates partly singly- and partly doubly-linked lists. Thus we extend the DSG with the two rules given in Fig. 6.

Fig. 7 depicts an abstraction-concretisation sequence employing this grammar and leading to a graph which is not a HC (as there exists a node with two p-successors). Typedness ensures that such a case cannot occur. Intuitively we fix the type, i.e., the set of pointers, for each pair \((X, j)\) \((X \in N, j \in [1, rk(X)])\) requiring that for each production rule \(X \rightarrow H\) exactly this set of pointers is derivable at external node \(ext_H(j)\).

Fig. 7. Abstraction and concretisation yielding an invalid heap configuration

Definition 6.2 (Typedness): An NT \(X \in N\) with \(rk(X) = n\) is typed if, for all \(j \in [1, n]\), there exists a set \(type(X, j) \subseteq \Sigma\) such that, for every \(H \in L(X)\) with \(V_{X^*} = \{v_1, \ldots, v_n\}\), \(type(X, j) = \{lab(e) | \exists e \in E_H : att(e)(1) = v_j\}\). A DSG \(G\) is typed if each of its NTs is typed.

3) Increasingness: Consider a production rule of the form \(X \rightarrow X^*\). When repeatedly applying this rule in a backward fashion, the abstraction (and thus the heap analysis) does not terminate. Again this behaviour is ruled out beforehand by a syntactic restriction on DSGs, called increasingness. It requires the right-hand side of each production rule to either contain a terminal edge or to be “bigger” (in terms of nodes and edges) than the handle of the left-hand side NT.

Definition 6.3 (Increasingness): Given \(G \in DSG_{\Sigma^*}\), \(X \in N\) is increasing if for all \(H \in G(X)\) it holds that either \(H\) consists of at least one terminal edge, i.e., \(lab(E_H) \cap \Sigma \neq \emptyset\), or \(|V_H| + |E_H| > rk(X) + 1\). A DSG \(G\) is increasing if all \(X \in N\) are increasing.

Increasing DSGs guarantee termination of abstraction, as applying rules in backward direction strictly reduces the size of the heap representation. Increasingness is implied when establishing local Greibach Normal Form [20], a normal form guaranteeing the following Hag property.

4) Local Concretisability: To be able to concretise locally before program statements access fields hidden in abstracted heap parts, it must be guaranteed that abstraction can always and everywhere be reversed by finitely many concretisation steps. This is not ensured by DSGs, as illustrated by the following example.

Consider the DSG \(G\) for doubly-linked lists given in Fig. 2 and the HC in Fig. 5 (left). Assume a program that traverses the doubly-linked list from its tail (the rightmost node of the list). Then the p-pointer of this element is abstracted or, put differently, “hidden” in the NT edge \(L\). As seen before, concretisation yields two heap configurations, cf. Fig. 5 (right) where the upper one is locally concrete at the beginning of the list instead of the end. Similarly to the situation in SL demonstrated in Ex. 4.6, successive concretisations do not resolve this issue, while ignoring the second rule is unsound due to under-approximation of the state space.

Therefore we formulate a property which guarantees that for each pair \((X, j)\) \((X \in N, j \in [1, rk(X)])\), there exists a subgrammar of \(G\) concretising an \(X\)-labelled edge at the \(j\)-th attached node while preserving the language of \(G\).

Definition 6.4 (Local Concretisability): A typed \(G \in DSG_{\Sigma^*}\) is locally concretisable if for all \(X \in N\) there exist grammars \(G_{(X, 1)}, \ldots, G_{(X, rk(X))}\) such that for all \(j \in [1, rk(X)]\), \(dom(G_{(X, j)}) = \{X\}\), \(G_{(X, j)} \subseteq G\), and

1) \(L_{G_{(X, j)}}(v) = L_{G}(v^*)\), and
2) \(\forall a \in \text{type}(X, j), H \in G_{(X, j)}(X) : \exists e \in E_H : lab(e) = a \land att_H(e)(1) = ext_H(j)\).

The DSG \(G\) from Fig. 2 is locally concretisable at \((X, 1)\) (pick \(G_{(X, 1)} = G\)) but not locally concretisable at \((X, 2)\). Extending \(G\) by the rule given in Fig. 8 establishes local concretisability.

Fig. 8. Ensuring local concretisability

Thus now we can pick a subgrammar for concretisation depending on where to concretise. That is, we employ all rules in \(G_{(X, j)}\) for concretisation at the \(j\)-th attached node of an \(X\)-labelled edge. Local concretisability can be established for every DSG by transforming it into local Greibach Normal Form [20]. This normal form is akin to Greibach Normal Form for string grammars, since it guarantees for every external node \(v\) the existence of an equivalent set of production rules with (outgoing) terminal edges at \(v\).

5) Backward Confluence: Being a context-free mechanism, the hyperedge replacement relation is clearly confluent, i.e., the order in which NT edges of a HG are replaced does not matter. However, confluence of heap abstraction is also a desired property as it implies that – under the assumptions that the DSG is increasing and describes the data structures arising in the program under consideration (except for finitely many local deviations) – the resulting abstract state space is finite. Therefore we introduce backward confluence for DSGs.

Definition 6.5 (Backward Confluence): \(G \in DSG_{\Sigma^*}\) is backward confluent if for each \(H \in HC_{\Sigma^*}\) and abstractions \(H \xrightarrow{\Delta} H_1\) and \(H \xrightarrow{\Delta} H_2\) there exists \(K \in HC_{\Sigma^*}\) such that \(H_1 \xrightarrow{\Delta} * K\) and \(H_2 \xrightarrow{\Delta} * K\).

To our knowledge there exists no algorithm that transforms a DSG into an equivalent backward confluent DSG. However, it is decidable whether an HRG is backward confluent ([24],
as a consequence of [25]). Notice that we actually consider local confluence. However, for increasing HIRGs abstraction is terminating, and thus Newman’s Lemma also implies global confluence.

VII. TRANSLATING DATA STRUCTURE GRAMMARS TO ENVIRONMENTS AND VICE VERSA

The motivation behind this work is to utilise the results on HAG properties for automatic generation of inductively defined predicates suitable for symbolic execution based on SL. To this aim we extend an existing translation between HIRGs and SL [4] such that it covers all DSGs. We conclude by showing that this translation preserves the properties determined to be necessary for heap abstraction.

For the applications of the translation procedure considered here, we are only interested in translating DSGs to environments and back. Thus we disregard program variables referencing heap objects, as they are not allowed to appear on right-hand sides of production rules or predicate definitions. The translation itself can easily be adapted to handle those variables; for more information we refer to [15], [21].

A. The Translation Mappings

We assume the translation functions graph[,] : SLF → HCEΣN, hr[g][,] : Env → HIRGSN, env[,] : DSGS → Env, form[,] : HCEΣN → SLF. Details can be found in App. A and [21]. In App. A-D we define a criterion for environments which ensures that hr[g][,] yields a DSG.

We have seen that the (internal) nodes of a hypergraph are always preserved under hyperedge replacement, i.e., can never be merged with another node, while external nodes of right-hand sides “disappear” due to hyperedge replacement. Thus the internal nodes of a hypergraph represent the heap locations that correspond to existing objects on the heap, i.e., those in its domain. By contrast, an SL variable can be interpreted as a reference to a location “outside” of the domain of the heap. For example, consider the SL formula φ := ∃x, y : x.n → y, which is fulfilled if there exist vx, vy ∈ Loc such that dom(h) = {vx + cn(n)} with h(vy + cn(s)) = vy. As this does not impose any restriction on vy, we could reference a location outside h. To bridge this gap we consider interpretations only, that fulfil the following assumption:

Assumption 7.1: Let φ ∈ SLF. For all interpretations i ∈ Int satisfying φ and all r1, r2 ∈ Var(φ) \ FV(φ) with r1 ≠ r2, it holds that i(r1) ≠ i(r2) unless the equality is explicitly stated by an equality assertion r1 = r2 in φ.

Note that this property can be checked for a given interpretation and is crucial for the validity of the correctness theorems of the translation. Under this precondition we now give an example application of env[,].

Example 7.2: Consider again the DSG G for doubly-linked lists given in Ex. 5.4. When translating a DSG to an environment, first for each NT X ∈ N of rank n, a predicate σX with parameters x1, . . . , xn is issued (we use → to indicate the translation step):

\[env[G] \quad N = \{L\} \quad \text{and} \quad rk(L) = 2 \twoheadrightarrow \sigma_L(x_1, x_2)\]

In the next step, each right-hand side of a production rule of NT X is converted into an SL formula, and the predicate body of σX results from disjuncting these formulae.

\[env[G] \quad G(X) = \{R_1, R_2\} \twoheadrightarrow \text{disjuncts} \quad \sigma_1, \sigma_2 \quad \text{and} \quad \sigma_L(x_1, x_2) := \sigma_1 \lor \sigma_2\]

For the translation of the right-hand sides, first each node is identified by a unique variable: external node j by xj, internal nodes v1, . . . , vm by rj, . . . , rm and vnull by null. The parameters of the predicate x1, . . . , xn become free variables in the disjuncts, whereas r1, . . . , rm will be existentially bound.

\[env[G] \quad \text{for } R_1: \text{no internal nodes} \sim \text{no quantifiers in } \sigma_1\]

\[env[G] \quad \text{for } R_2: \text{one internal node identified by } r_1 \rightarrow \sigma_2 := \exists r_1 \left[ \ldots \right]\]

Then each edge is translated one-by-one using the node identifiers as variables in the corresponding heap formula. Terminal edges become pointer assertions.

\[env[G] \quad \text{for } R_1: \text{n-edge} \rightarrow x_1.n \rightarrow x_2 \quad \text{and} \quad x_2.p \rightarrow x_1\]

\[env[G] \quad \text{for } R_2: \text{p-edge} \rightarrow x_1.n \rightarrow x_2 \quad \text{and} \quad x_1.p \rightarrow x_1\]

NT edges are translated into predicate calls.

\[env[G] \quad \text{for } R_2: \text{L-edge} \rightarrow \sigma_L(r_1, x_2)\]

The translation is finalised by glueing together the edge translations using *.

\[env[G] \quad \text{for } R_1: \text{r-edge} \rightarrow \sigma_1 := x_1.n \rightarrow x_2 \quad \text{and} \quad x_2.p \rightarrow x_1\]

\[env[G] \quad \text{for } R_2: \text{r-edge} \rightarrow \sigma_2 := \exists r_1 : x_1.n \rightarrow r_1 \quad \text{and} \quad r_1.p \rightarrow x_1 \quad \text{and} \quad \sigma_L(r_1, x_2)\]

Thus G is translated into env[G] = \{σL(x_1, x_2) := σ_1 \lor σ_2\}.

The translation of environments to HRGs, hr[g][,], shares the same correspondences and thus proceeds analogously, glueing graphs instead of assertions. Table I provides an overview of the correspondences between DSGs and SL.

<table>
<thead>
<tr>
<th>DSG</th>
<th>SL</th>
</tr>
</thead>
<tbody>
<tr>
<td>HC H ∈ HCEΣN</td>
<td>closed formula φ ∈ SLF</td>
</tr>
<tr>
<td>extended HC H ∈ HCEΣN</td>
<td>formula φ ∈ SLF</td>
</tr>
<tr>
<td>nonterminal X ∈ N</td>
<td>predicate σ ∈ Pred</td>
</tr>
<tr>
<td>terminal edge (lab(e) ∈ Σ)</td>
<td>pointer assertion x.s → y</td>
</tr>
<tr>
<td>nonterm. edge (lab(e) ∈ N)</td>
<td>pred call σ(x_1, . . . , x_n)</td>
</tr>
<tr>
<td>X-nodes G(X) ∈ N</td>
<td>pred def. σ(x_1, . . . , x_n) = ...</td>
</tr>
<tr>
<td>DSG G</td>
<td>environment Γ</td>
</tr>
</tbody>
</table>

**TABLE I**

OVERVIEW: DSG VS. SL

To relate the concept of heap configurations and SL heaps for the correctness criterion, we introduce a heap mapping α : H → HCΣ (cf. Def. A.14) that generates a graph representation of a heap in a straightforward manner. The following correctness theorem states that given an environment Γ, a heap fulfills the formula σ(x_1, . . . , x_n) iff a graph representing this heap is contained in the language of the HKG that Γ translates to. The starting graph for the derivation is exactly the translation of the predicate call, i.e. the handle of the associated nonterminal. A similar intuition applies to the reverse direction.

**Theorem 7.3 (Correctness of hr[g][,])**: Let Γ ∈ Env. Then for all σ(x_1, . . . , x_n) = φ ∈ Γ:

\[(l, h) ∈ ησ(σ) \iff \alpha(h) ∈ Lhr[g][\{graph[σ(x_1, . . . , x_n)]\}]

\]
Theorem 7.4 (Correctness of $\text{env}[] : \text{DSG}_{2N} \rightarrow \text{Env}$): Let $G \in \text{DSG}_{2N}$. Then for all $X \in N$:

$$H \in L(X_{ext}^n) \iff \exists i \in \int : (l, \alpha^{-1}(H)) \in \eta_{\text{env}}[C]$$

where $l = l(x_1) \ldots l(x_{\#(X)})$.

The proofs of the theorems can be found in App. A-C. Furthermore as a consequence of the theorems above it holds that given $\Gamma \in \text{Env}, G \in \text{DSG}_{2N}$:

1) for all closed $\phi \in \text{SLF}$:

$$h, i_\phi, \eta_\Gamma \models \phi \iff \alpha(h) \in L_{hrg}[\Gamma](\text{graph}[\phi])$$

2) for all $H \in HC_{2N}$:

$$H' \in L_G(H) \iff \alpha^{-1}(H'), i_\phi, \eta_{\text{env}}[G] \models \text{form}[H]$$

B. Property Preservation under Translation

In the preceding section we have motivated the need for additional properties of DSGs, entailing the notion of HAGs. For each of those we gave a (synonymic) characterisation for environments in Sect. IV. A DSG or environment featuring all of them is suitable for symbolic execution of pointer-manipulating programs. Thus they should be preserved under translation in both directions. The following theorem states that this is indeed the case.

Theorem 7.5 (Preservation of Properties under Translation):

- For a productive, typed, increasing and locally concretisable environment $\Gamma$ confluent under abstraction it holds that the DSG $G := hrg[\Gamma]$ is productive, typed, increasing, locally concretisable and backward-confluent.
- For a productive, typed, increasing, locally concretisable and backward-confluent DSG $G$ it holds that the environment $\Gamma := \text{env}[G]$ is productive, typed, increasing, locally concretisable and confluent under abstraction.

The proof of this theorem (divided into several lemmas) is given in App. B. For DSGs there exist transformations (productivity, typedness, increasingness, local concretisability) or decision procedures (backward confluence) to handle those properties. Thus DSGs have the potential to serve as the basis for designing suitable environments for symbolic execution of pointer-manipulating programs.

VIII. CONCLUSION

In this paper, we established a formal connection between two different approaches for analysing heap-manipulating programs, namely DSGs and inductively defined predicates in SL. The former provide an intuitive way to define abstraction on heaps and allow to perform local concretisation automatically, thus avoiding the necessity to define abstract versions of pointer-manipulating operations by hand. To ensure the correctness and practicability of this approach, several structural properties are established, viz., productivity, typedness, increasingness, local concretisability, and backward confluence.

We extended a previous translation result [15] to the translation of DSGs into SL and back. Moreover we defined SL counterparts of the aforementioned properties and proved that they are preserved by the translation in both directions. This enables the automated generation of SL predicates that by themselves specify a sound set of abstraction and unrolling rules for symbolic execution. These theoretical results pave the way for an integrated approach to heap analysis where the intuitive description of data structures using DSGs on the one side and the expressivity and tool support of SL on the other side are combined.

Current research concentrates on algorithms for automated DSG learning on-the-fly. That is, abstraction rules are automatically derived whenever the size of the heap representation reaches a certain threshold [26], [27]. Transferring these learning procedures to SL either directly or indirectly via DSG translation would be worthwhile to investigate. Furthermore we are planning to extend the DSG approach to concurrent programs. In particular, we would like to study whether a permission-based SL approach, such as one of those presented in [28], [29], could provide the basis for this extension.

REFERENCES


We start by giving the definitions of (extended) heap configurations and contexts, as informally introduced in Sect. V. 

**Definition I (Heap Configuration):** A heap configuration (HC) is a hypergraph $H = (V, E, att, lab, ext)$ in $H_G\Sigma_N$ with
- $\Sigma_N = \Sigma \cup N$, $rk(\Sigma) = \{2\}$ and $ext = \varepsilon$,
- for every $v \in V$ and $s \in \Sigma$, $\{e \in E \mid lab(e) = s, att(e)(1) = v\} \leq 1$,
- $\exists e \in E : lab(e) \in \Sigma \wedge att(e)(1) = \nu_0$ and
- for every $v \in V$ there exist $e \in E$ and $i \in \{1, rk(e)\}$ such that $att(e)(i) = v$.

If $lab(E) \cap N = \emptyset$, then $H$ is **concrete**, otherwise **abstract**. The set of all (concrete) HCs is denoted by $H_G\Sigma_N (HC_S)$. If an HC satisfies all conditions above except for $ext = \varepsilon$ and if additionally $\nu_0 \not\in V_H$, it is an **extended HC**. Accordingly the set of all (concrete) extended HCs is denoted by $H_G\Sigma_N (HC_E\Sigma_N)$.

**Definition A.2 (Context):** Let $H \in H_G\Sigma_N$. We define the **context of $H$ as context** $H := \{H' \mid V_{H'} = V_H$ and for arbitrarily many $\nu_1, \nu_6 \in [ext_H], \nu_7 = \nu_6, E_{H'} = E_H, att_{H'} = att_H, lab_{H'} = lab_H, ext_{H'} = \}$. 

A. **SL to HRG**

We begin with translating an SL formula to a hypergraph. In an SL formula, each heap location is described by a dedicated variable, while in heap configurations locations are represented by nodes. Thus the translation process has to identify each node by the variables referring to the location it represents. We formalise this identification by tagging nodes with these variables.

**Definition A.3 (Tagged Heap Configuration):** A tagging of $H \in HC\Sigma_N$ is a mapping $t : V_H \mapsto 2^{V_0 \cup \{\nu_0\}}$. The pair $(H, t)$ is called a tagged heap configuration. The set of all tagged HCs over $\Sigma$ is denoted by $HC\Sigma_N$. If we expect a hypergraph $H$ instead of a tagged one $(H, t)$, we omit the tagging $t$.

In the graphical representation, the tagging of a hypergraph is depicted by a set of variables associated to a node.

A predicate call appearing in the SL formula will be translated to a (nonterminal) edge, which is labelled with a symbol indicating the predicate. We denote the nonterminal for predicate $\sigma$ by $X_\sigma$. Thus $N := \{X_\sigma \mid \sigma \in \text{Pred}\}$.

To simplify the translation functions later on, we introduce two operators, unification and join. The first transforms a tagged hypergraph such that nodes with non-disjoint taggings are merged.

**Definition A.4 (Unification Operator $\therefore$):** Let $H = (V_H, E_H, att_H, lab_H, \varepsilon) \in HC\Sigma_N$ and $t$ a tagging of $H$. The unification $\{\therefore, H, t\}$ of $(H, t)$ is defined as $\therefore := (V_H, E_H, att, lab, \varepsilon) \in HC\Sigma_N$ with $V := \{V_H\}, att(e)(j) := f(V_H, att_H(e)(j))$ and $t'(v) := \bigcup_{v \in V} \{v \mapsto v' \mid v' \in V\}$ $t(v')$ where $f : V_H \rightarrow V$ such that $\forall v_1, v_2 \in V_H : t(v_1) \cap t(v_2) = \emptyset$.

The join operator takes two hypergraphs as inputs and joins them into one.

**Definition A.5 (Join Operator $\bowtie$):** Let $(H_1, t_1), (H_2, t_2)$ be two tagged hypergraphs with $ext_{H_1} = ext_{H_2} = \varepsilon$. W.l.o.g.
we assume that $H_1, H_2$ have disjoint sets of nodes and edges.

The *join* is defined by

\[
(H_1, t_1) \bowtie (H_2, t_2) := \downarrow ((V_{H_1} \cup V_{H_2}, E_{H_1} \cup E_{H_2}, \text{att}_{H_1} \cup \text{att}_{H_2}, \text{lab}_{H_1} \cup \text{lab}_{H_2}, \varepsilon), t_1 \uplus t_2)
\]

The join of sets of tagged hypergraphs is defined in the usual way.

The translation of an SL formula is defined in an inductive way, i.e., for atomic formulae a graph pendant is given, and the translation of a composite SL formula is realised by combining the graphs resulting from the latter using the join-operator. Finally, an expose-function marks the nodes corresponding to the free variables in the SL formula as external.

**Definition A.6** (*tgraph[.]*: SLF to HC): The translation function $tgraph[.] : \text{SLF} \mapsto \mathcal{H}^{\mathcal{E}}_{\Sigma_N}$ is defined by the following base cases

- $tgraph[\text{emp}] := \{(H_{\text{emp}}, t_{\text{emp}})\} = \{(\emptyset, \emptyset, \emptyset, \emptyset, \varepsilon, t_{\text{emp}})\}$
- $tgraph[x \rightarrow y] := \downarrow \{(H_s, t)\}$ where
  
  \[
  (H_s, t) = (\{x\}, \{y\}, t)
  \]

- $tgraph[x \rightarrow \text{null}] := \downarrow \{(H_{\text{null}}, t)\}$ where
  
  \[
  (H_{\text{null}}, t) = (\{x\}, \emptyset, t)
  \]

- $tgraph[\sigma(x_1, \ldots, x_n)] := \downarrow \{(H_{\sigma}, t)\}$ where
  
  \[
  (H_{\sigma}, t) = (\{x_1\}, \ldots, \{x_n\}, \emptyset, \{x_1\}, \ldots, \{x_n\}, t)
  \]

and composite cases

- $tgraph[\phi \lor \psi] := tgraph[\phi] \cup tgraph[\psi]
- $tgraph[\phi \land \psi] := tgraph[\phi] \otimes tgraph[\psi]
- $tgraph[\exists x : \phi] := \{(H, t) \mid (H, t') \in tgraph[\phi], t = \{v \mapsto t'(v) \setminus \{x\} \mid v \in \text{dom}(t')\}\}
- $tgraph[\forall x : \phi] := \{\emptyset (H, t) \mid (H, t') \in tgraph[\phi]\}$ with $\forall v \in V_H$.

\[
t(v) := \begin{cases} t'(v) \cup \{x, y\} & \text{if } x \in t'(v) \lor y \in t'(v) \\ t'(v) & \text{otherwise} \end{cases}
\]

The translation of an SL formula, $tgraph[.]$, yields a set of tagged HCs. The tagging contains the information about the mapping of the formulas free variables and determines $v_{\text{null}}$.

**Example A.7:** Consider the SL formula $\phi = x \rightarrow \text{null} \lor (\exists z : \sigma(x, z) \lor x \rightarrow \text{null})$. The translation $tgraph[\phi]$ is given by

The tagging is attached to the respective node. When we consider environments later on, the predicate bodies are not allowed to contain any free variables except for those appearing in the predicate call. As free variables (predicate parameters) correspond to external nodes in the HRG approach, we define a function expose that converts a tagged HC into an HCE by marking tagged nodes as external.

**Definition A.8** (*expose* Function): Given a tagged heap configuration $(H, t)$, we define

\[
\text{expose}((H, t), x_1 \ldots x_n) := ((V_H, E_H, \text{att}_H, \text{lab}_H, v_1 \ldots v_n), t) \quad \text{if} \quad t(v_j) = \{x_j\}, j \in [1, n] \text{ where } x_1, \ldots, x_n \in \text{Var}.
\]

We can now define the translation of SL formulae to extended HCEs used in the translation of environments to HRGs, where free variables (predicate parameters) in the SL formula are exposed, i.e. made external, in the resulting HCE.

**Definition A.9** (*graph[.]*: SLF to HCE): The translation function $graph[.] : \text{SLF} \mapsto \mathcal{H}^{\mathcal{E}}_{\Sigma_N}$ is defined by

\[
\text{graph}[\phi] := \{\text{expose}((H, t), x_1 \ldots x_n) \mid (H, t) \in tgraph[\phi] \land \{x_1, \ldots, x_n\} = \text{FV}(\phi)\}
\]

The HRG results from translating the body of predicate $\sigma$ into a set of HCEs which then provide the rules for nonterminal $X_\sigma$.

**Definition A.10** (*hrg[.]*: Environment to HRG): The translation function $hrg[.] : \text{Env} \mapsto \mathcal{H}^{\mathcal{E}}_{\Sigma_N}$ is defined by

\[
hrg[\Gamma] := \bigcup_{\sigma(x_1, \ldots, x_n) = \psi \in \Gamma} \{X_\sigma \mapsto H \mid (H, t) \in \text{graph}[\psi]\}.
\]

Note that as all graphs resulting from $\text{graph}[\psi]$ are unified and for predicate definitions we assumed that the $x_j$ are pairwise distinct, there exists a unique $v_j$ with $t(v_j) = \{x_j\}$. Furthermore according to the definition of predicate definitions, $\psi$ contains no free variables except for $x_1, \ldots, x_n$, thus the tags of all non-exposed nodes are empty and therefore intuitively no information about the variables is lost by omitting the tagging in the resulting production rules.

**B. DSG to SL**

During the translation of formulae to heap configurations, we used a graph tagging to associate graph nodes to the
formula’s variable identifiers. Now when generating a formula from a heap configuration, we need to tag each node with an identifier to be able to refer to it in the formula later on. To this aim we define its default tagging.

**Definition A.11 (Tagging):** Let \( H \in \text{HCE}_{\Sigma_N} \). The tagging \( t \) of \( H \) is a mapping \( t : V_H \mapsto 2^{\text{Var}(\{\text{null}\})} \) with

\[
t(v) = \begin{cases} 
\{\text{null}\} & \text{if } v = v_{\text{null}} \\
\{x_j\} & \text{if } v = \text{ext}_H(j) \\
\{r\} & \text{otherwise with } r \text{ a "fresh" variable}
\end{cases}
\]

We denote the set of tagged HCEs/HCs by \( \text{HCE}_{\Sigma_N}^T/\text{HC}_{\Sigma_N}^T \) and \( \text{HCE}_{\Sigma_N}^T/\text{HC}_{\Sigma_N}^T \), respectively.

The translation of HCs is defined in an inductive way, i.e., each edge in the graph is translated to a (sub) formula. These formulae are then combined using separating conjunction \( * \).

**Definition A.12 (HCE to SLF form):** The translation function \( \text{form}[: \text{HCE}_{\Sigma_N} \mapsto \text{SLF} \) translates an edge \( c \) contained in HG \( H \) with default tagging \( t \) by

\[
\text{form}[c, t] = \begin{cases} 
x.s \mapsto y & \text{if } s = \text{lab}(e) \in \Sigma, \\
x.t(v), y.t(v') & \text{if } X = \text{lab}(e) \in N, \\
\sigma_X(y_1, \ldots, y_n) & \text{if } X = \text{lab}(e) \in N
\end{cases}
\]

and the hypergraph by

\[
\text{form}[[H, t]] = \exists r_1 \ldots \exists r_m : \bigcup_{e \in E_H} \text{form}[e, t]
\]

where

\[
\{r_1, \ldots, r_m\} = \text{Var}(\bigcup_{e \in E_H} \text{form}[e, t]) \setminus \{x_1, \ldots, x_{rk(H)}\}
\]

is the set of variables tagging internal nodes of \( H \) and occurring in the resulting formula, and \( \bigcup_{e \in E_H} \text{form}[e, t] \) is defined as

\[
\begin{align*}
\text{emp} & \quad \text{if } E_H = \emptyset \\
\text{form}[e_1, t] * \ldots * \text{form}[e_n, t] & \quad \text{if } E_H = \{e_1, \ldots, e_n\}
\end{align*}
\]

The environment then contains a predicate definition for each nonterminal in the DSG with the disjunction of the translated right-hand sides as predicate bodies.

**Definition A.13 (env):** DSG to Environment: Let \( G \in \text{DSG}_{\Sigma_N} \). The translation \( \text{env}[: \text{DSG}_{\Sigma_N} \mapsto \text{Env} \) is defined as

\[
\text{env}[G] = \bigcup_{X \in \text{dom}(G)} \{\sigma_X(x_1, \ldots, x_{rk(X)}) = \bigvee_{H \in G(X)} \text{form}[[H, t]]
\]

where \( t \) is the tagging of \( H \).

An example of a DSG-to-environment translation can be found in Ex. 7.2.

**C. Correctness of the Translation**

The heap mapping \( \alpha \) establishes a connection between heaps \( h \in \text{He} \) and concrete hypergraphs representing heaps.

**Definition A.14 (Heap Mapping \( \alpha \)):** The mapping \( \alpha : \text{He} \mapsto \text{HC}_{\Sigma} \) is defined by \( \alpha(h) = \{V, E, \text{att}, \text{lab}, \text{ext}\} \) with

\[
V = \{\text{dom}(h) \cup \text{Nil}\} \\
\text{Nil} = \begin{cases} 
\{v_{\text{null}}\} & \text{if } \exists l \in \text{dom}(h) : h(l) = \text{null} \\
\emptyset & \text{otherwise}
\end{cases}
\]

\[
E = \{e_{v, s} : v \in V, \ s \in \Sigma, \ h(v + \text{cn}(s)) \in \text{dom}(h)\}
\]

\[
\text{lab}(e_{v, s}) = v[h(v + \text{cn}(s))]
\]

\[
\text{ext} = \varepsilon
\]

where \([,] \) rounds down to a location contained in \( \{1, |\Sigma| + 1, 2 \cdot |\Sigma| + 1, \ldots\} \).

Interpretations may contain mappings \( x \mapsto n \) that do not influence the satisfiability of a formula \( \phi \) whenever \( x \notin \text{Var}(\phi) \). Thus from now on when speaking about interpretations, we consider the "least" interpretations only, i.e. we restrict \( \text{dom}(z) \) to the variables \( \text{Var}(\phi) \) that occur in \( \phi \). Considering those interpretations only, \( \alpha \) is injective and therefore its inverse is properly defined. We defined \( \alpha \) in such a way that it establishes a connection between the functional and the graphical representation of a heap based on \( h \in \text{He} \). While this is sufficient when considering the relation between satisfying instances of a predicate call and heap configurations contained in the language of a nonterminal handle, we need to establish a connection between interpretations and hypergraphs in the case where the relation between free variables and external nodes is not implicitly given by resorting to the language (and thus eliminating all external nodes). This connection is again defined via a tagging of the graph nodes.

**Definition A.15 (Extended Heap Mapping \( \alpha_t \)):** The mapping \( \alpha_t : \text{He} \times \text{Int} \mapsto \text{HC}_{\Sigma_N}^T \) is defined by

\[
\alpha_t(h, t) = (\alpha(h), t)
\]

with \( t(v) = \{x \in \text{Var} | i(x)v = v\} \).

To establish the correspondence between SL and HRGs, our aim is to show that the set of heap-interpretation pairs satisfying a formula \( \phi \) corresponds to the hypergraphs contained in the language of the translation of \( \Phi \) into a HG and the other way around. The inductive proof structure, however, requires us at some places to prove a stronger statement, namely additionally considering the external nodes of the hypergraphs whose contexts are contained in the language. As in these contexts the information about the external nodes is disregarded, we introduce an additional tagging that remembers for each node in the context if and which external nodes had been collapsed into it.

**Definition A.16 (External Tagging \( t_{\text{ext}} \)):** Let \( H_{\text{ext}} \in \text{HCE}_{\Sigma_N}^T \), \( H \in L(H_{\text{ext}}) \). The tagging \( t_{\text{ext}} \) of \( H \) is defined as \( t_{\text{ext}} : V_H \mapsto 2^{\{x_1, \ldots, x_k\}} \) where \( k = |\text{ext}_H| \) with \( t_{\text{ext}}(v) := \)
\{x_j \mid \exists j, H'_{ext} \in \text{context}(H_{ext}) : H'_{ext} \rightarrow H \land \text{ext}_{H_{ext}}(j) = v\}

a) Graph and Formula Correspondence: To prove Theorems 7.3 and 7.4, we first show the correctness of the graph and formula translation.

To simplify notation in the subsequent proofs, we introduce the following notation for hyperedge replacement. We denote the replacement \( H \rightarrow_{e,i} H' \) for \( G \rightarrow H_{RS}, H \rightarrow H_{RS} \) in \( H_{RS} \), \( p = X \rightarrow K, K \in G(X) \) and \( e \in E_H \) with \( \text{lab}(e) = X \) by \( H \rightarrow_{[e]} H' \).

**Proposition A.17 (graph[\cdot] \rightarrow SL Formula to HCE):** Let \( \phi \in \text{SLF} \). We assume that for the predicate interpretation \( \eta \) it holds that \( V(l, h) \in \eta(\sigma) \iff \alpha(h) \in L_{env}[\gamma](X_{ext}), i(x_i) := l(i) \), with \( \gamma \) an environment, \( \sigma \in \text{Pred}_\phi \) and \( X_{ext} \in N_{env}[\gamma] \).

Then it holds for \( h \in H_{e,i} \), \( i \in \text{Int} \): \( h, i, \eta \models \phi \iff \alpha(h, i) = (H, t), H \in L_{env}[\gamma](\text{graph}[\phi]), \) and \( t(v) \neq \emptyset \iff v \in \text{ext}_{\text{graph}[\phi]} \).

**Proof:** The proof will proceed by structural induction and in its process we make use of the following direct implication of Prop. A.17 when considering composed formulae of the form \( \phi_1 \land \phi_2, \phi \lor \phi_2, \phi \land \phi_2 \) or \( \phi \land x = y \land \delta \land \exists x : \phi \). This is necessary as the expose function, which generalizes the external nodes, is applied only after the translation of the complete formula.

Direct implication of Prop. A.17: Let \( FV(\phi) = \{x_1, \ldots, x_n\} \). Assuming \( i(x_j) \) to be pairwise distinct (\( j \in [1, n] \)), it holds for \( h \in H_{e,i} \), \( i \in \text{Int} \):

\[ h, i, \eta \models \phi \iff \alpha(h, i) = (H, t), H \in L_{env}[\gamma](\text{graph}[\phi]), \text{ and } t(v) \neq \emptyset \iff v \in \text{ext}_{\text{graph}[\phi]} \]

To simplify notation later on, we split the claim into

\[ \ast h \in H_{e,i} \in \text{Int} : h, i, \eta \models \phi \iff \alpha(h, i) = (H, t), H \in L_{env}[\gamma](\text{graph}[\phi]) \]

and

\[ \ast \ast t(v) \neq \emptyset \iff v \in \text{ext}_{\text{graph}[\phi]} \]

We prove the proposition by structural induction on the SL formula \( \phi \).

\( \phi = \text{emp} \). emp is only fulfilled by the empty heap, i.e. \( \text{dom}(h) = \emptyset \), thus \( \alpha(h, t) = (H, t) \) with \( H = (\emptyset, 0, att_0, lab_0, e) \) and \( t = t_0 \).

Furthermore \( hrg[\phi] = \{\{\langle 0, 0, att_0, lab_0, e \rangle, t_0\}\} \).

Thus the claim holds directly.

\( \phi = x.s \rightarrow y \). From the semantics of SL and the restriction that variables refer to defined heap locations only, we know that for every heap satisfying \( \phi \) it holds that \( \text{dom}(h) = \langle i(x) + cn(s), i(y) \rangle \) and thus \( i = 0 \rightarrow [l] \rightarrow y \rightarrow l', l, l' \in \text{Loc} \) such that \( h + cn(s) = l' \). Thus the set of all \( (h, t) \) where \( h, i, \eta \models \phi \) (isomorphic hypergraphs are considered equal) is given by

\[ F := \bigcup_{h \in H_{e,i} \in Int \text{ with } h, i, \eta \models \phi} \{H_1(t_1), (H_2, t_2)\} \]

with

\[ H_1 = (\{i(x), i(y)\}, \{i(x), i(y)\}, i(x), l, l' \rightarrow l, l' \rightarrow l'), (H, t), H \in L_{env}[\gamma](\text{graph}[\phi]) \]

\[ H_2 = (\{i(x), i(x)\}, \{i(x), i(x)\}, i(x), l, l' \rightarrow l, (H', t'), H' \in L_{env}[\gamma](\text{graph}[\phi]) \]

and tagging \( t_1 = t_2 = l \rightarrow x, l' \rightarrow y \). Furthermore we know that \( \text{graph}[x.s \rightarrow y] = \{H, t\} \) with \( \phi = \phi_1 \lor \phi_2 \). According to the SL semantics, \( \phi \) is fulfilled by heap \( h \), interpretation \( i \) and predicate interpretation \( \eta \), if either \( h, i, \eta \models \phi_1 \) or \( h, i, \eta \models \phi_2 \).
\( \phi. \) By IH we get that the claim already holds for 
\[ \text{graph}[\phi[x]], \quad j = 1, 2. \]
The translation of \( \phi \) is defined as 
\[ \text{graph}[\phi] = \text{expose}(\text{graph}[[\alpha]] \cup \text{graph}[\phi[x]], x_1 \ldots x_n) = \text{graph}[\phi[x]] \cup \text{graph}[\phi[x]] \]
where \( x_1 \ldots x_n \) \( = \text{FV}(\phi). \)
Then either \( \alpha_i(h, i) \in L_{\text{env}[\text{graph}[\phi]]} \) or \( \alpha_i(h, i) \in L_{\text{env}[\text{graph}[\phi]]}. \)
Thus it follows directly that \( \alpha_i(h, i) \in L_{\text{env}[\text{graph}[\phi]] \cup \text{graph}[\phi[x]]}. \)

\section*{Proposition A.18 (H to SLF Formula Translation \( \text{form}[\cdot] \))}

Let \( H \in \Pi^{\Pi_2}. \) We assume that for DSG \( G \) it holds that for every \( X \subseteq N, H' \subseteq L_G(X'_\alpha), \) then exists \( \exists h \in \text{int} \\
\langle l, \alpha(H') \rangle \in \text{env}[\alpha(G)] \) such that \( i = i(x_1) \ldots i(x_{i(G)}). \)

Then for all \( H' \subseteq L_G \) it holds that:
\[ \alpha^{-1}(H') \subseteq \text{form}(H) \]

Proof: We prove the proposition by induction over the number \( n \) of edges in the hypergraph \( H_n. \)

First consider \( HC H_0 \) with an empty set of edges, i.e., \( E_{HC_{\Pi_2}} = \emptyset. \) As every node in a HCs has to be attached to at least one edge, \( V_{HC_{\Pi_2}} = \emptyset \) (where \( H_0 \) is the empty graph). Thus we know that for arbitrary \( G, L_G(H_0) = \{ \emptyset \}. \) Furthermore \( \alpha^{-1}(H_0) = h \) with \( \text{dom}(h) = \emptyset. \) The translation procedure directly provides us with \( \text{form}(H_0, t_0) \in \text{omp} \), which is only fulfilled by the empty heap. Thus the claim holds.

Consider \( HC H_1 \) with \( E_{HC_{\Pi_2}} = \{ e_1 \} \) and let \( t \) be the default tagging of \( H_1. \)

\section*{Case 1: lab(e_1) \in \Sigma.}
Then either \( V_{HC_{\Pi_2}} = \{ \emptyset \} \) or \( V_{HC_{\Pi_2}} = \{ e_1, v_{\text{null}} \} \) due to the HC properties. In both cases as \( H_1 \) is concrete \( L(H_1) = \{ \}. \)

Assume \( V_{HC_{\Pi_2}} = \{ e_1 \}, \) then \( \text{att}(e_1) = v_1) e_1 \) and \( \alpha^{-1}(H_1) = h \) with \( \text{dom}(h) = \{ 1 + c n(lab(e_1)) \}, \) \( h(1 + c n(lab(e_1))) = 1 + \text{cn}(lab(e_1)) \), and \( \text{dom}(h) = \emptyset. \) From the translation we get:
\[ \text{form}(H_1, t_1) = \exists r_1 : r_1 \cdot \text{lab}(e_1) \mapsto r_1, \]

\section*{SL semantics fulfilled if there exists \( v \in \text{Loc} \) s.t. \( \text{h}_v[r_1 \mapsto v_1, \text{np} = r_1 \cdot \text{lab}(e_1) \mapsto r_1 \) and holds for \( v = 1 \) (and therefore \( r_1 \cdot \text{lab}(e_1) = 1 + \text{cn}(lab(e_1)) \)) only. Thus the claim holds.

Assume \( V_{HC_{\Pi_2}} = \{ v_1, v_{\text{null}} \} \) and \( \text{att}(e_1) = v_1) e_1. \) Then \( \alpha^{-1}(H_1) = h \) with \( \text{dom}(h) = \{ 1 + \text{cn}(lab(e_1)) \}, \) \( h(1 + \text{cn}(lab(e_1))) = \emptyset. \) Again from the translation we get:
\[ \text{form}(H_1, t_1) = \exists r_1 : r_1 \cdot \text{lab}(e_1) \mapsto \emptyset \]

Thus the claim holds.

\section*{Case 2: lab(e_1) \in N.}
Then \( V_{HC_{\Pi_2}} = \{ v_1, \ldots, v_n \} \) with \( \text{att}(e_1) = v_1 \ldots v_n. \) From the translation we get:
\[ \text{form}(H_1, t_1) = \exists r_1, \ldots, r_n : \text{lab}(e_1)[r_1, \ldots, r_n] \]

Now either \( H_1 \subseteq \text{lab}(e_1)^* \), i.e., the \( v_i \) are pairwise distinct. Then the claim directly holds.

Or there exist \( v_j, v_k \in V_{HC_{\Pi_2}}. \) Then we know that \( L(H_1) = \{ H_1|e_1| K \in L(\text{lab}(e_1)) \} \) and moreover directly by the definition of hyperform replacement \( \text{ext}_K(k) = \text{ext}_K(k) \) after replacement of \( e_1 \). Furthermore we know that for \( \text{form}(H_1, t_1), \) \( \text{lab}(e_1)[r_1, \ldots, r_n] \) holds as \( x_1 = x_k. \) Thus for all \( H' \subseteq L(H_1) \) there exists an \( H \in L(\text{lab}(e_1)^*) \) with \( \alpha^{-1}(H') = (h', \text{np}) \in \text{form}(H_1, t_1) \) iff \( \alpha^{-1}(H) = (h, \text{np}) \in \text{form}(\text{lab}(e_1)^*, \text{t}). \)

For \( E_{HC_{\Pi_2}} = \{ e_1, e_2 \} \) analogous.

Now assume \( HC_{H_{n+1}} \) with \( E_{HC_{H_{n+1}}} = \{ e_1, \ldots, e_{n+1} \} \) and \( V_{HC_{H_{n+1}}} = \{ v_1, \ldots, v_n \}. \) By construction of the default tagging for \( H_{n+1} \) we know that each node has a unique tag. By translation then \( \text{form}(H_{n+1}, t_1) = \exists r_1, \ldots, r_m : \text{form}(e_1, t) \ast \ldots \ast \text{form}(e_{n+1}, t). \)

\section*{Case 1: \( \text{att}(e_{n+1}) \in \{ \text{att}(e_1) \} \subseteq \{ \text{att}(e_1) \} \subseteq \{ \text{att}(e_1) \} \subseteq \{ \text{att}(e_1) \} \).

This means that \( e_{n+1} \) together with its \( \text{att}(e_1) \) forms a separate connected component and \( \text{var form}(\text{en}_{n+1}, t) \subseteq \text{var form}(\text{en}_{n+1}, t) \subseteq \{ 0, j \in [1, n] \).

Together with the IH there trivially exist exactly two disjoint heap parts that satisfy \( \exists r_1, \ldots, r_m : \text{form}(e_1, t) \ast \ldots \ast \text{form}(e_{n+1}, t) \) and \( \exists r_1, \ldots, r_m : \text{form}(e_{n+1}, t) \), respectively. Thus the claim holds.

\section*{Case 2: \( j \in [1, n] : \{ \text{att}(e_{n+1}) \} \cap \{ \text{att}(e_1) \} = \{ \text{null} \} \),

and for all other \( j \) the sets of attached nodes are disjoint. As \( \text{att}(e_1) \) is unique, there exists at most one node tagged with \{null\}. The corresponding identifier for this node in the resulting SL formula is null. According to the definition of a heap h, null cannot be an element of dom(h), as it can only be pointed to but never point to any location in the heap. Analogously to case 1, there exist exactly two disjoint heap parts satisfying the two parts of the translated formula. Thus the claim holds.

\section*{Case 3: \( j \in [1, n] : \{ \text{att}(e_{n+1}) \} \cap \{ \text{att}(e_j) \} = \{ v_{\text{null}} \},

and we can partition \( H_{n+1} \) into the subgraphs \( H' \) \( \langle \{ \text{att}(e_{n+1})\} \rangle, \{ e_1 \} \), \( \text{att}(e_{n+1}) \), \( \text{lab}(e_{n+1}) \), \( e \rangle \)
and \( H''' = \langle V_{HC_{H_{n+1}}} \setminus \{ \text{att}(e_{n+1}) \} \setminus \{ v_1 \}, E_{HC_{H_{n+1}}} \setminus \{ e_{n+1} \}, \text{att}_{e_{n+1}} \setminus \text{lab}_{e_{n+1}} \setminus \text{g}_{e_{n+1}} \setminus e \rangle \), where the claim already holds for \( H' \) and \( H''' \). Moreover \( \alpha^{-1}(H') = h', \alpha^{-1}(H''') = h'' \) with \( \text{dom}(h') \cap \text{dom}(h'') = \emptyset \) (as \( E_{HC_{H''}} \cap E_{HC_{H'''}} = \emptyset \)) and \( \alpha^{-1}(H_{n+1}) = h \) with \( h = h' \ast h''). This case can be generalised to cut containing multiple nodes.
accordingly. Then directly from the semantics of SL and the associativity of HRG replacement the claim follows.

To prove that the translation from HRG to environments is correct, we need to consider the translation of hypergraphs containing external nodes. On the logics side, external nodes correspond to free variables occurring in a formula. Thus later on the level of production rules/predicate definitions, external nodes of the right-hand sides of production rules correspond to the parameters of the predicate definition.

Proposition A.19 (form[·]: HCE to SLF): Let \( H \) ∈ HCE\_\Sigma\_\_\Sigma. We assume that for DSG \( G \) it holds that, for every \( X \in N \), \( H' \in L_G(X^*_{ext}) \iff \exists i \in \text{Int} : (l, o(H')) \in \eta_{env}[G] \) for \( l = i(x_1) \ldots i(x_{rk(X)}) \). Then for all \( H' \in HC\_G \) it holds that

\[
\exists i \in \text{Int} : o^{-1}(H'), i, \eta_{env}[G] = \text{form}[H].
\]

Proof: If \( H \) has no external nodes, \( \text{ext}_H = \emptyset \), then the claim follows directly from Prop. A.18. Now we assume that \( H \) has external nodes \( v_1, \ldots, v_n \), i.e., \( \text{ext}_H = v_1 \ldots v_n \). From the tagging used during the translation procedure we know that each \( v_j \), \( j \in [1, n] \) is tagged with \( \{ x_j \} \) and moreover after translation we know that there exist free variables \( x_1, \ldots, x_n \) in \( \text{form}[H] \). With Prop. A.18 the claim already holds for \( H_{ext} := (V_H, E_H, att_H, lab_H, e) \) and the empty interpretation \( \iota_g \). Again the translation procedure gives us that \( \text{form}[H] \) and \( \text{form}[H_{ext}] \) correspond except for the free variables \( x_1, \ldots, x_n \) in \( \text{form}[H] \), which are existentially bound in \( \text{form}[H_{ext}] \) (and renamed to some \( r_k \)). Now assume that the free variables \( x_1, \ldots, x_n \) in \( \text{form}[H] \) correspond to the existentially bound variables \( r_1, \ldots, r_n \) in \( \text{form}[H_{ext}] \). Then with the implication of SL that all bound variables refer to distinct locations unless the opposite is explicitly stated by an equality assertion together with the fact that they can only be referenced in the scope of \( \text{form}[H] \), we know that \( x_j \neq r_k, j \in [1, n] \) for all \( r_k \) from the set of bound variables in \( \text{form}[H] \) (as by definition of the tagging it cannot happen that \( \text{form}[H] \) contains subformula \( x_j = r_k \)). Then we know that for each \( H' \in L_G(H) \) there exists an interpretation \( \iota \) (namely the one resulting from \( o^{-1}(H'), t_{ext} \) as \( \text{dom}(t_{ext}) = \{ v_1, \ldots, v_n \} \) with \( t_{ext}(v_j) = \{ x_j \}, j \in [1, n] \)) fulfilling the if-part of the claim. Moreover with the assumption of SL that all free variables have to refer to a location in the heap, there cannot exist further interpretations \( \iota \) fulfilling the claim, providing the only-if-part of the claim. Thus the claim holds.

b) HRG and Environment Correspondence: We will prove the correspondence of HRGs and environments with an inductive proof of correspondence between recursive functions whose least fixpoints are the generated graph language and the predicate interpretation, respectively.

We start by giving the recursive functions and argue why they actually compute the correct predicate interpretation and language.

In the case of SL the predicate interpretation provides the semantics of predicate calls. It is defined as the least fixpoint of the function as given in Def. 3.5.

Notice that this definition of the \( k \)-th predicate interpretation implicitly considers each predicate call in \( \Phi_1 \lor \ldots \lor \Phi_m \) under the predicate interpretation of step \( k - 1 \). We recall that a predicate call corresponds to a nonterminal edge. Thus we define a derivation step \( \phi \) that replaces each nonterminal edge in a hypergraph by all eligible right-hand sides.

Definition A.20 (Complete Derivation Step \( \phi \)): We define a complete derivation step of a HG \( H \) via the function \( \phi \): \( HG_{\Sigma_{\bar{G}}} \rightarrow HG_{\Sigma_{\bar{G}}} \) where \( \phi \) \( (H) := \{ H' \in HG_{\Sigma_{\bar{G}}} \mid H' = H[e_1 \# K_1] \ldots [e_k \# K_k] \} \) with \( \{ e_1, \ldots, e_k \} = E_H \{ c \mid \text{lab}(c) \in \Sigma \} \) for arbitrary \( K_i \) with \( K_i \in P(\text{lab}(e_i)) \).

With this definition we can now specify the recursive function \( f_G \) for HRG, which collects all terminal hypergraphs derivable from \( X^*_{ext}, X \in N \) in \( k \) complete derivation steps.

Definition A.21 (HRG Language: Inductive): For an HRG \( G \) we define the function \( f_G : \{ N \rightarrow 2^{HG_{\Sigma_{\bar{G}}}} \} \rightarrow \{ N \rightarrow 2^{HG_{\Sigma_{\bar{G}}}} \} \) where \( f_G(P_k)(X) := \{ H' \in HG_{\Sigma_{\bar{G}}} \mid \exists R \in P(X) : H' \in \downarrow P_k(\text{R}) \} := \{ f_{k+1}(X) \in N \} \).

First we introduce an auxiliary lemma that helps us with the proof of the correspondence of both HRG language definitions, proven subsequently.

Lemma A.22: Let \( H \in HG_{\Sigma_{\bar{G}}} \) and \( G \in HG_{\Sigma_{\bar{G}}} \). Then \( L_G(H) = L_G(\text{context}(H)) = \text{context}(L(H)) \).

Proof: This directly follows from the associativity of HRG replacement and the fact that \( \text{context}(\text{context}(H)) = \text{context}(H) \).

Lemma A.23: The least fixpoint of \( f_G \), \( \text{fix}_G(f_G) \), contains all graphs \( H \in HG_{\Sigma_{\bar{G}}} \) that can be derived from \( X^*_{ext} \), \( \forall X \in N \), i.e., \( X^*_{ext} \Rightarrow* H \). Thus \( L_G(X^*_{ext}) = \text{context}(\text{fix}_G(f_G))(X) \).

Proof: This lemma follows directly from Lemma A.22.

With these prerequisites we can now prove the two main propositions, which build the basis of the proof of Theorems 7.3 and 7.4.

Proposition A.24 (Correspondence of \( f_T \) and \( f_{hrg[\bar{G}]} \)): Let \( \Gamma \in \text{Env} \). Then for all \( \sigma \in \text{Pred}_T \) the following holds:

\[
(l, h) \in f_T(\eta_0(\sigma)) \iff \eta(h) \in L(f_{hrg[\bar{G}]}(P_k)(\text{graph}[\sigma(x_1, \ldots, x_n)]))
\]

Proof: (*) From \( \text{graph}[\sigma(x_1, \ldots, x_n)] \) we know that \( \text{graph}[\sigma(x_1, \ldots, x_n)] \) yields the HG \( X_{\sigma_{ext}} \). Moreover from the translation \( \text{hrg}[\Gamma] \) we directly know that for each predicate definition \( \sigma(x_1, \ldots, x_n) := \Phi_1 \lor \ldots \lor \Phi_m \in \Gamma \) there exists a nonterminal \( X \in N_{hrg[\bar{G}]} \) of rank \( rk(X) = n \) with production rules \( P(X_n) = \{ H \mid \exists (H, t) \in \text{graph}[\Phi_1 \lor \ldots \lor \Phi_m] \} = \{ H \mid \exists (H, t) \in \text{graph}[\Phi_1 \lor \ldots \lor \text{graph}[\Phi_m] \} \} \).

We prove the proposition by induction of the number \( k \) of fixpoint iterations.

- If \( k = 0 \), then \( f_T(\eta_0(\sigma)) = \{ (l, h) \mid h, i_\Phi[x_1 \rightarrow l(1), \ldots, x_n \rightarrow l(n)], \eta_0 = \Phi_1 \lor \ldots \lor \Phi_m \} \), and thus

\[
(l, h) \in f_T(\eta_0(\sigma)) \iff h, i_\Phi[x_1 \rightarrow l(1), \ldots, x_n \rightarrow l(n)], \eta_0 = \Phi_1 \lor \ldots \lor \Phi_m.
\]

With Prop. A.17 we know that \( \forall \Phi : h, i_\Phi = \Phi \iff \eta(h) \in L_{hrg[\bar{G}]}(\text{graph}[\Phi]) \) if \( \Gamma \) and \( \text{hrg}[\Gamma] \) fulfill the
assumption of the proposition. Thus as \(\eta_0\) and \(P_0\) trivially fulfill the assumption of Prop. A.17:

\[
(l, h) \in f_1(\eta_0)(\sigma) \iff \\
\alpha(h) \in L_{\text{hrsg}[\Gamma]}(\text{graph}[\Phi_1 \lor \ldots \lor \Phi_m])
\]

Moreover \(L_{P_0}(\text{graph}[\Phi_1 \lor \ldots \lor \Phi_m]) = \{H \in \text{HG}_S \mid H \in \text{context}\{\text{graph}[\Phi_1 \lor \ldots \lor \text{graph}[\Phi_m]\}\}\}.\) According to the definition of the translation, \(P(\alpha) = \text{graph}[\Phi_1 \lor \ldots \lor \text{graph}[\Phi_m]\}. \) Thus \(L_{P_0}(\text{graph}[\Phi_1 \lor \ldots \lor \Phi_m]) = L_{\text{hrsg}[\Gamma]}(\{R \in \text{HG}_S \mid \exists R \in P(X_S)\})\). As this set contains terminal graphs only and \(P_0\) is the empty function,

\[
(l, h) \in f_1(\eta_0)(\sigma) \iff \\
\alpha(h) \in L_{\text{hrsg}[\Gamma]}(\{H \in \text{HG}_S \mid \exists \Gamma \in P(\alpha) : H \in \downarrow_{\text{hrsg}[\Gamma]}(R)\})
\]

which we can rephrase directly into

\[
(l, h) \in f_1(\eta_0)(\sigma) \iff \alpha(h) \in L_{\text{hrsg}[\Gamma]}(f(P_0)(X_S))
\]

• \(k \to k + 1\). We have to show that

\[
(l, h) \in f_{k+1}(\eta_0)(\sigma) \iff \alpha(h) \in L_{\text{hrsg}[\Gamma]}(f_{k+1}(\eta_0)(\sigma))
\]

By applying the definition of \(f_k\) we get

\[
(l, h) \in f_{k+1}(\eta_0)(\sigma) \iff (l, h) \in f_k(\eta_0)(\sigma)
\]

where \(f_k(\eta_0)(\sigma) = \{(l' : h', t) : h' \in \{h \in \text{graph}[\Phi_1 \lor \ldots \lor \Phi_m] : \exists (l') \in \text{graph}[\Phi_1 \lor \ldots \lor \Phi_m] \} \}\). By ind. hyp. we know that the assumption of Prop. A.17 is fulfilled for \(f_k(\eta_0)\) and \(f_{k+1}(\eta_0)(\sigma)\) and thus

\[
\alpha(h) \in L_{\text{hrsg}[\Gamma]}(f_{k+1}(\eta_0)(\sigma))
\]

Then by the definition of the translation of \(\lor\) we know

\[
\alpha(h) \in L_{\text{hrsg}[\Gamma]}(\text{graph}[\Phi_1 \lor \ldots \lor \Phi_m])
\]

As, for all \(X \in \text{dom}(f_{\text{hrsg}[\Gamma]}), f_{k+1}(\eta_0)(\sigma)\) contains only terminal graphs, we know that

\[
L_{f_{k+1}(\eta_0)}(\text{graph}[\Phi_1 \lor \ldots \lor \text{graph}[\Phi_m]]) = \\
L_{\text{hrsg}[\Gamma]}(\{H \mid H \in \downarrow_{\text{hrsg}[\Gamma]}(\text{graph}[\Phi_1 \lor \ldots \lor \text{graph}[\Phi_m]])\})
\]

As \(P(X_S) = \{H \mid H \in \text{graph}[\Phi_1 \lor \ldots \lor \text{graph}[\Phi_m]\}\} \) we rephrase using the definition of \(f_1\) and \(f_{\text{hrsg}[\Gamma]}\) into

\[
(l, h) \in f_{k+1}(\eta_0)(\sigma) \iff \\
\alpha(h) \in L_{\text{hrsg}[\Gamma]}(f_{k+1}(\eta_0)(\sigma)(X_{\text{ext}}))
\]

Then directly with (*)

\[
(l, h) \in f_{k+1}(\eta_0)(\sigma) \iff \\
\alpha(h) \in L_{\text{hrsg}[\Gamma]}(f_{k+1}(\eta_0)(\sigma)(X_{\text{ext}}))
\]

We have to show that for all \(H \in \text{HG}_S, X \in \text{env}\) \(f_{\text{hrsg}[\Gamma]}(\eta_0)(\sigma)\)

\[
H \in L(f_{\text{hrsg}[\Gamma]}(\eta_0)(\sigma)\}
\]

**Theorem A.25:** Let \(\Gamma \in \text{Env}\) Then for all \(\sigma(x_1, \ldots, x_n) := \Phi_1 \lor \ldots \lor \Phi_m \in \Gamma\) the following holds:

\[
h, i, \eta \vDash \sigma(x_1, \ldots, x_n) \iff \\
\alpha(h) \in L_{\text{hrsg}[\Gamma]}(\text{graph}[\sigma(x_1, \ldots, x_n)]).
\]

**Proof:** From \(\text{graph}[]\) we know that \(\text{graph}[\sigma(x_1, \ldots, x_n)]\) yields the HG \(X_{\text{ext}}\). Moreover from the semantics of SL we know that \(h, i, \eta \vDash \sigma(x_1, \ldots, x_n) \iff \{l := i(x_1), \ldots, i(x_n)\} \in \eta\). As the least fixpoint of \(f_1\) yields \(\eta\) and additionally \(L_{\text{hrsg}[\Gamma]}(X_{\text{ext}}) = \text{context}(\{\bigcup_{X \in \text{env}} L_P^{\text{HRG}[\Gamma]}(\text{graph}[\sigma(x_1, \ldots, x_n)])\})\), the claim follows directly from Lemma A.22 and Prop. A.17.

**Proposition A.26** (Correspondence of \(f_2\) and \(f_{\text{env}[\Gamma]}\)) Let \(G \in \text{DSG}_{\text{ext}}\) with nonterminals \(N\). Then \(H \in L(f_2(P_0)(X)) \iff \alpha_1^{-1}(H, t_{\text{ext}}) = \{h, i, l := i(x_1), \ldots, i(x_n)\} \land (l, h) \in f_{\text{env}[\Gamma]}(\eta_0)(\sigma X), \forall X \in \text{N} \land r_k(X) = n\).

**Proof:** (*) From the translation function \(f_{\text{env}[\Gamma]}\) we directly know that for HRG \(G\) for each \(P(X) := \{R_1, \ldots, R_m\}, X \in \text{N} \land r_k(X) = n\) there exists a predicate definition \(\sigma_X(x_1, \ldots, x_n) := \text{form}[R_1] \lor \ldots \lor \text{form}[R_m]\) in \(\text{env}[\Gamma]\).

Proof by induction over the steps \(k\) of the fixpoint iteration.

• \(k = 0\). Then \(f_0(P_0)(X) = \{H' \in \text{HG}_S \mid \exists \Gamma \in P(X) : H' \in \downarrow_{\text{hrsg}[\Gamma]}(\{R \in \text{HG}_S \mid \exists \Gamma \in P(X) : H \in L_{\text{hrsg}[\Gamma]}(R)\})\}. \) Thus \(H \in L(f_2(P_0)(X)) \iff H \in L_{\text{hrsg}[\Gamma]}(R_i) \lor \ldots \lor L_{\text{hrsg}[\Gamma]}(R_m)\). W.l.o.g. assume that \(H \in L_{\text{hrsg}[\Gamma]}(R_i)\).

From Prop. A.18 we know that

\[
H \in L_{\text{hrsg}[\Gamma]}(R_i) \iff \alpha_1^{-1}(H, t_{\text{ext}}) \vDash \text{form}[R_i]\)

and thus

\[
H \in L_{\text{hrsg}[\Gamma]}(R_i) \iff \alpha_1^{-1}(H, t_{\text{ext}}) \vDash \text{form}[R_i]\}
\]

From the semantics of SL we get

\[
H \in L_{\text{hrsg}[\Gamma]}(R_i) \iff \alpha_1^{-1}(H, t_{\text{ext}}) \vDash \text{form}[R_i] \lor \ldots \lor \text{form}[R_m].
\]

Furthermore for each \(R' \in \{R_1', \ldots, R_m'\}\) we know that \(|\text{ext}_{R'}| = n \land r_k(X) = n\) by construction of \(t_{\text{ext}}\) we know then that \(\text{dom}(i) = \{x_1, \ldots, x_n\}\). Then by definition of \(f_{\text{env}[\Gamma]}\) and (*) it follows that

\[
H \in L_{\text{hrsg}[\Gamma]}(R'_i) \iff \alpha_1^{-1}(H, t_{\text{ext}}) \vDash \text{form}[R'_i] \lor \ldots \lor \text{form}[R_m].
\]

We have to show that for all \(H \in \text{HG}_S, X \in \text{N}:

\[
H \in L(f_{\text{hrsg}[\Gamma]}(\eta_0)(\sigma X)) \iff \alpha_1^{-1}(H, t_{\text{ext}}) \vDash \text{form}[R'_i] \lor \ldots \lor \text{form}[R_m].
\]
for \( l = i(x_1)i, \ldots, i(x_n)i \). For \( f^k_G(P_0) \) and \( f^k_{env[G]} \) we already know by ind. hyp. that the proposition holds. With Prop. A.18 we know that for all \( H, H' \in HCS_\Sigma \):

\[
H' \in L_G(H)
\]

\[
\iff \alpha^{-1}(H'), \eta_{env[G]} = form[H].
\]

Thus for each \( R_j \in P(X_j), j \in [1, m] \) it holds that

\[
H' \in L^k_{f^k_G(P_0)}(R_j)
\]

\[
\iff \alpha^{-1}_h(H', t_{ext}), f^k_{env[G]}(\eta_0) = form[R_j]
\]

and therefore

\[
H' \in L^k_{f^k_G(P_0)}(R_1) \cup \ldots \cup L^k_{f^k_G(P_0)}(R_m)
\]

\[
\iff \alpha^{-1}_h(H', t_{ext}), f^k_{env[G]}(\eta_0) = form[R_1] \lor \ldots \lor form[R_m]
\]

As \( \psi^1_{f^k_G(P_0)}(R_j) \) is by definition the set of all terminal graphs that are derivable from \( R_j \) using the production rules from \( G \) in \( k \) complete derivation steps and with SL semantics of \( \mathcal{V} \) we know that

\[
H' \in L_G(\psi^1_{f^k_G(P_0)}(R_1)) \cup \ldots \cup L_G(\psi^1_{f^k_G(P_0)}(R_m))
\]

\[
\iff \alpha^{-1}_h(H', t_{ext}), f^k_{env[G]}(\eta_0) = form[R_1] \lor \ldots \lor form[R_m]
\]

We rephrase into

\[
H' \in L_G(\{H'' : H'' \in H_G \mid H'' \in \psi^1_{f^k_G(P_0)}(R_1)\})
\]

\[
\cup \ldots \cup
\]

\[
L_G(\{H'' : H'' \in H_G \mid H'' \in \psi^1_{f^k_G(P_0)}(R_m)\})
\]

\[
\iff \alpha^{-1}_h(H', t_{ext}) = h(i, l = i(x_1)i, \ldots, i(x_n)i),
\]

\[
(l, h) \in \{(l', h', h'_{i_0}|x_1 \rightarrow l(1), \ldots, x_n \rightarrow l(n))
\]

\[
f^k_{env[G]}(\eta_0) = form[R_1] \lor \ldots \lor form[R_m]
\]

From the assumption (*) we know that \( \{R_1, \ldots, R_m\} = P(X, \mathcal{I}) \); therefore

\[
H' \in L_G(\{H' : H' \in H_G \mid \exists R \in P(X): H' \in \psi^1_{f^k_G(P_0)}(R)\})
\]

\[
\iff \alpha^{-1}_h(H', t_{ext}) = h(i, l = i(x_1)i, \ldots, i(x_n)i),
\]

\[
(l, h) \in \{(l', h', h'_{i_0}|x_1 \rightarrow l(1), \ldots, x_n \rightarrow l(n))
\]

\[
f^k_{env[G]}(\eta_0) = form[R_1] \lor \ldots \lor form[R_m]
\]

Then directly by definition of \( f_G \) and \( f_{env[G]} \)

\[
H' \in L_G(\{f^k_{env[G]}(\eta_0)(\sigma x) : \sigma \in \Sigma \})
\]

\[
\iff \alpha^{-1}_h(H', t_{ext}) = h(i, l, h) \in f^k_{env[G]}(\eta_0)(\sigma x)
\]

for \( l := i(x_1)i, \ldots, i(x_n)i \).

\[
\Box
\]

**Theorem A.27 (Correctness of \( env[\cdot] \): HRG to Environment):** Let \( G \in DSG_{\Sigma, N} \). Then for all \( X \in N \) the following holds:

\[
H \in L(X^*_G) \iff \exists \alpha^{-1}(H) = (h, i_o) \wedge h, i, \eta_{env[G]} = form[X^*_G].
\]

\[
\Box
\]

**Proof:** We know that \( form[X^*_G] = \sigma_X(x_1, \ldots, x_n) \) for \( r_k(X) = n \) and thus by semantics of SL we know that \( \sigma_X(x_1, \ldots, x_n) \) is fulfilled by \( (h, i, \eta) \iff (h, l) \in \eta(\sigma_X) \) where \( l := i(x_1)i, \ldots, i(x_n)i \). Furthermore from [15] we know that \( \eta \) is defined as the least fixpoint of the function \( f_{env[G]} \). Moreover \( L_G \) is defined as \( L_G(\{X' \in \text{context} \mid \{\sum_{\eta \in \mathcal{N}} Lf_G(X') \}) \). Then the claim follows directly from Lemma A.22 and Prop. A.26.

**D. DSG Criterion for Environments**

While every DSG translates to an environment, the opposite does not necessarily hold for the translation from environments to HRGs. Consider for example the environment \( \Gamma \) containing predicate definitions \( \sigma_1(x_1, x_2) := x_1 \rightarrow x_2 \land x_2 \rightarrow y \) and \( \sigma_2(x_1) := x_1 \rightarrow x_2 \). The resulting grammar satisfies all requirements of an HRG, but does allow to derive a hypergraph containing a node with two outgoing \( s \)-selectors. Thus it depicts no heap configuration.

From the semantics point of view this is unproblematic, as any SL formula containing a predicate call to \( \sigma_2(x_1) \) is unsatisfiable on the one hand and any hypergraph \( H \) derivable from a hypergraph containing an \( X_{\sigma_2} \)-labelled edge violates the ICC requirements. However, the data structure environment property guarantees that translation always yields a DSG. It utilises the notion of unrolling a predicate call.

**Definition A.28 (Unroll):** Let \( x(x_1, \ldots, x_n) \) be a predicate call. The \( k \)-th unrolling of \( \psi^k_G(x(x_1, \ldots, x_n)) \) is defined as

\[
\psi^k_G(x(x_1, \ldots, x_n)) = \sigma(x_1, \ldots, x_n)
\]

\[
\psi^k_G(x(x_1, \ldots, x_n)) \text{ results by replacing each predicate call of } \psi^k_G(x(x_1, \ldots, x_n)) \text{ by the corresponding predicate body (where parameters are mapped with the arguments).}
\]

Intuitively an environment is a data structure environment, if for each of its predicate definitions \( \sigma(x_1, \ldots, x_n) := \phi \) arbitrarily many (say \( k \)) unrollings of a call to \( \sigma \) cannot yield a formula that defines more than one \( s \)-selector for each \( x \in Var(\psi^k_G(x(x_1, \ldots, x_n))) \).

**Definition A.29 (Data Structure Environment):** We call a predicate definition \( \sigma(x_1, \ldots, x_n) := \sigma_1 \lor \ldots \lor \sigma_m \) a data structure predicate, if for all \( k \in N \) the following holds:

for any disjunct \( \phi \) in the disjunctive normal form of \( \psi^k_G(x(x_1, \ldots, x_n)) \exists x, y_1, y_2 \in Var, s \in \Sigma : x, s \rightarrow y_1 \in \phi, \phi, x, s \rightarrow y_2 \in \phi \) and \( y_1 \neq y_2 \). An environment is a data structure environment, if every predicate it contains is a data structure predicate.

It can be checked algorithmically whether an environment is a data structure environment [21].

**Theorem A.30:**

1. Let \( \Gamma \) be a data structure environment. Then \( hrg(\Gamma) \) is a DSG.
2. Let \( G \) be a DSG, then \( env[G] \) is a data structure environment.

**Proof:** (Sketch)

1. Let \( \Gamma \) be a data structure environment. Assume \( hrg(\Gamma) \) is not an DSGs. Thus one of the following HRG properties
or HC properties for a graph \( H \in L_{\text{hr}}[\Gamma]\) for some \( X \in N \) is violated.

- each nonterminal has a fixed rank: this follows directly by Def. 3.3 (each disjunct in a predicate definition refers to all parameters)
- all external nodes are pairwise distinct: again directly by Def. 3.3 (all parameter names are pairwise distinct)
- \( \Sigma_N = \Sigma \cup \Sigma^N \) directly by definition of \( \text{graph}[\Gamma] \)
- \( \text{rk}(\Sigma) = \{\} \) again directly by definition of \( \text{graph}[\Gamma] \) (terminal edges only arise from pointer assertions)
- no node with more than one \( s \)-selector, \( \forall s \in \Sigma \); proof by contradiction, assume there exists a disjunct in the unrolling that defines the \( s \)-selector for variable \( x_j \) twice, show that this leads to a derivable in \( \text{hr}g[\Gamma] \) where the node tagged with \( x_j \) holds two outgoing \( s \)-edges
- \( v_{\text{null}} \) has no selectors: directly by Def. 3.3 (null not allowed in predicate body)
- every node has at least one edge attached: directly by definition of heap formula/predicate bodies and \( \text{graph}[\Gamma] \)
- \( \text{ext} = e \) by the observation that hyperedge replacement cannot generate new external nodes

2. Analogous.

For details on this proof we refer to [21].

**APPENDIX B**

**PROPERTY PRESERVATION UNDER TRANSLATION**

A. Productivity

**Lemma B.1:**

1. For a productive environment \( \Gamma \) it holds that the DSG
   \[ G := \text{hr}g[\Gamma] \]
   is productive.
2. For a productive DSG \( G \) it holds that the environment
   \[ \Gamma := \text{env}[G] \]
   is productive.

**Proof:** (1) We prove the first statement by induction over the number \( n \) of productive predicate definitions in \( \Gamma \).

\( n = 1 \). Assume \( \Gamma := \{ \Phi(\cdot) \alpha_1 \ldots \alpha_m \} \). As we know that \( \Phi \) is productive, we know that there exists one \( \alpha_j, j \in [1, m] \) which is primitive, i.e. contains no predicate calls. Thus \( \text{graph}[\alpha_j] \) does not contain any nonterminal edges. Then directly by construction of \( \text{hr}g[\Gamma] \) we know that \( X_\sigma \) is productive.

Let \( \Gamma \) contain \( n + 1 \) predicate definitions \( \alpha_1, \ldots, \alpha_{n+1} \). Consider an arbitrary \( \phi \in \{ \phi_1, \ldots, \phi_{n+1} \} \). Then from \( \phi := \alpha_1 \ldots \alpha_m \alpha_k \), \( k \in [1, m] \) with the at least number of predicate calls different from \( \phi \). As \( \phi \) is productive, we know that \( \sigma_k \) has calls to at most \( n \) different productive predicates different from \( \phi \). Let us assume these predicates to be \( \phi_1, \ldots, \phi_n \). We know that \( \phi_1, \ldots, \phi_n \) can only be productive if one of the disjuncts in their predicate bodies contains at most \( n - 1 \) calls to productive predicates different from \( \phi \) (as the productivity of \( \phi \) depends on the productivity of each of these predicates). Then by induction we know that each \( X_{\phi_j}, j \in [1, n] \) is productive. As \( \text{graph}[\alpha_1] \) constructs hypergraphs where nonterminal edges labelled with \( X_{\phi_j}, j \in [1, n] \) only arise due to predicate calls \( \phi_j \) occurring in \( \sigma_k \), again by construction of \( \text{hr}g[\Gamma] \) we directly know that \( X_\sigma \) is productive.

As an environment is productive if each single predicate definition is and the only nonterminals \( G = \text{hr}g[\Gamma] \) contains are \( \{ X_\phi \mid \phi \in \text{Pred}_G \} \) the claim follows directly.

(2) Analogous.

B. Increasingness

**Lemma B.2:**

1. For an increasing environment \( \Gamma \) it holds that the DSG
   \[ G := \text{hr}g[\Gamma] \]
   is increasing.
2. For an increasing DSG \( G \) it holds that the environment
   \[ \Gamma := \text{env}[G] \]
   is increasing.

**Proof:** (1) We first prove for a single increasing predicate definition \( \sigma(x_1, \ldots, x_n) := \sigma_1 \vee \ldots \vee \sigma_m \), that the resulting nonterminal \( X_\sigma \) is increasing.

For \( \sigma \) to be increasing each \( \sigma_i, i \in [1, m] \) contains either a pointer assertion or the number of predicate calls and variables appearing in \( \sigma_i \) is greater than the number of parameters of \( \sigma_i + 1 \). In the first case \( \text{graph}[\sigma_i] \) directly translates the pointer assertion into a terminal edge. By definition of \( \text{hr}g[\Gamma] \) we know that each \( \sigma_i \) is translated into a production rule

\[ X_\sigma \to \text{graph}[\sigma_i]. \]

Therefore in this case \( X_\sigma \to \text{graph}[\sigma_i] \) is increasing. In the second case \( \text{graph}[\sigma_i] \) generates a new node tagged with \( x \), for each variable \( x \) and a nonterminal edge for each predicate call appearing in \( \sigma_i \). Thus \( \mid V_{\text{graph}[\sigma_i]} \mid + \mid E_{\text{graph}[\sigma_i]} \mid > n + 1 \). As by definition of \( \text{hr}g[\Gamma] \) we know that \( \text{rk}(X_\sigma) = n, X_\sigma \to \text{graph}[\sigma_i] \) is increasing for the second case, too.

As an environment is increasing if each single definition is and \( G \) contains exactly those nonterminals \( X_\sigma \) where \( \sigma \in \text{Pred}_G \), it follows directly that all nonterminals in \( G \) are increasing as thus so is \( G \).

(2) Analogous.

C. Typedness

**Lemma B.3:**

1. For a typed environment \( \Gamma \) it holds that the DSG
   \[ G := \text{hr}g[\Gamma] \]
   is typed.
2. For a typed DSG \( G \) it holds that the environment
   \[ \Gamma := \text{env}[G] \]
   is typed.

**Proof:** (1) We first prove for a single typed predicate definition \( \sigma := \sigma_1 \vee \ldots \vee \sigma_m \) that the resulting nonterminal \( X_\sigma \) is typed.

Assume that \( X_\sigma \) is not typed. Then there exists an \( v \in V_{X_\sigma} \). HCs \( H_1, H_2 \in L(X_\sigma) \) and \( e \in E_{H_1} \) with 

\[ \text{att}_{H_1}(e){(1)} = v, \text{lab}_{H_1}(e) = s \]

such that \( \text{be} \in E_{H_2} \) and 

\[ \text{att}_{H_2}(e'){(1)} = v \text{ and } \text{lab}(e') = s. \]

As \( H_1, H_2 \in L(X_\sigma) \) we know by the translation correctness (Theorem 7.3) that there exist \( h,h' \in \text{He} \) and \( i,i' \in \text{Int} \) satisfying \( \sigma(x_1, \ldots, x_n) \) such that \( \alpha_h(i) = \langle h_1, i_1 \rangle \) and \( \alpha(h', i') = \langle h_2, i_2 \rangle \). But as we know that \( i(x_1) \neq \ldots \neq i(x_n) \), then from the definition of \( \alpha_h \) it follows that

\[ i(x_j) + cn(s) \in \text{dom}(h) \text{ and } i(x_j) + cn(s) \notin \text{dom}(h'). \]

which is a contradiction to the typedness of \( \sigma(x_1, \ldots, x_n) \).
As an environment is typed if every single predicate definition is and the only nonterminals $G$ contains correspond to a predicate defined in $\Gamma$, it follows directly that all nonterminals in $G$ are typed in such a way so is $G$.

(2) Analogous.

D. Local Concretisability

Lemma B.4:

1) For a typed and locally concretisable environment $\Gamma$ it holds that the DSG $G := hrg[\Gamma]$ is locally concretisable.

2) For a typed and locally concretisable DSG $G$ it holds that the environment $\Gamma := env[G]$ is locally concretisable.

Proof: (1) We first prove for a single typed and locally concretisable predicate $\sigma = \sigma_1 \lor \ldots \lor \sigma_m$, that the resulting nonterminal $X_\sigma$ is locally concretisable.

For $\sigma(x_1, \ldots, x_n)$ we know that for each $1 \leq j \leq n$ there exists a $\Phi_j \subseteq \{\sigma_1, \ldots, \sigma_m\}$ fulfilling the two properties given in Def. 4.7. Let $\Gamma'$ be the resulting environment that results from $\Gamma$ by exchanging all occurrences of $\sigma$ into $\sigma'$. We know that during translation of the predicate definition of $\sigma$ a nonterminal $X_\sigma$ of rank $m$ with production rules $X_\sigma \rightarrow graph[\sigma_1] \lor \ldots \lor graph[\sigma_m]$ will be generated. We define the subgrammar $G(X_{\sigma', \sigma})$ such that it contains the production rules $X_{\sigma'} \rightarrow graph[\phi], \phi \in \Phi_j, i.e. G(X_{\sigma', \sigma}) := G \setminus G(X_{\sigma})$ where $G := hrg[\Gamma']$. For this subgrammar we show that it fulfills the two properties given in Def. 6.4.

1) As $\Gamma$ is locally concretisable we know that \{h, i\} $\in H \times int \mid h, i, \eta \models \sigma(x_1, \ldots, x_n)\} = \{(h, i) \in H \times int \mid h, i, \eta \models \sigma'(x_1, \ldots, x_n)\}.

With Theorem 7.3 we then know that for all those $(h, i)$-pairs corresponding HCs $\alpha(h)$ are contained in $L(hrg[\Gamma'])(graph[\sigma_1, \ldots, x_n])$. As by definition of the translation $\text{graph}[\sigma_1, \ldots, x_n] = X_{\sigma', \sigma}$, it follows that $L(hrg[\Gamma'])(X_{\sigma'}) = L(hrg[\Gamma'])(X_{\sigma'}) = L(G(X_{\sigma', \sigma}) \cup (G \setminus (G(X_{\sigma'}))))(X_{\sigma'})$.

2) From the local concretisability of $\sigma$ we know that $\forall s \in \Sigma: s \in type(\sigma, j) \iff \{x_j, s \mapsto y\} \in \text{Atomic}(\phi)$ for $\phi \in \Phi_j$ and $y \in Var \cup \{null\}$. As $\text{graph}[\phi]$ translates pointer tagged nodes into an HCN with two nodes $v_1, v_2$ tagged $x_j$ and $y$, and an s-labelled edge from $v_1$ to $v_2$ and merges all $x_j$-tagged nodes, the production rule $p : X_\sigma \rightarrow \text{graph}[\phi], \phi \in \Phi_j$ derives an outgoing s-labelled edge at external node $x_j$. i.e. $(3e \in E_{\text{graph}[\phi]} \cdot la\text{b}_{\text{graph}[\phi]}(\phi) \cdot \{s \land att_{\text{graph}[\phi]}(\phi)\}(j) = e \land ext_{\text{graph}[\phi]}(\phi)$. As the translation procedure generates edges labelled with pointers only from pointer assertions, $ext_{\text{graph}[\phi]}(\phi)$ defined exactly those selectors in $type(\sigma, j), \forall \phi \in \Phi_j$. Moreover by the definition of hyperedge replacement we then know that $type(X_{\sigma', j}) = type(\sigma, j)$. Thus the claim holds.

As a typed environment is locally concretisable if every single predicate definition is and the only nonterminals $G$ contains are corresponding to a $\sigma$ in $\Gamma$ it follows that all nonterminals in $G$ are locally concretisable an thus so is $G$.

(2) Analogous.

E. Confluence

Lemma B.5:

1) For an environment $\Gamma$ confluent under abstraction it holds that the DSG $G := hrg[\Gamma]$ is backward confluent.

2) For a backward confluent DSG $G$ it holds that the environment $\Gamma := env[G]$ is confluent under abstraction.

Proof: (1) Let the environment $\Gamma$ be confluent under abstraction. Assume that $hrg[\Gamma]$ is not backward confluent. Then there exist HCs $H_1, H_2, H_3$ such that $H_1 \Rightarrow hrg[\Gamma] H_2$ and $H_2 \Rightarrow hrg[\Gamma] H_1$, and $H_1 \Rightarrow hrg[\Gamma] H_3$ with $H_1 \Rightarrow hrg[\Gamma] H_1$, $H_2 \Rightarrow hrg[\Gamma] H_2$ and $H_1, H_2$ isomorphic. From the definition of the translation we know that each predicate definition $\sigma(x_1, \ldots, x_n) := \sigma_1 \lor \ldots \lor \sigma_m$ in $\Gamma$ is translated into a nonterminal $X_{\sigma'}$ of rank $m$ with production rules $hrg[\Gamma'](X_{\sigma'}) = \{\text{graph}[\sigma_1], \ldots, \text{graph}[\sigma_m]\}$. Now let us assume that $H \Rightarrow hrg[\Gamma'] H_1$ abstracts $H'$ of $H$ using production rule $X_{\sigma'} \rightarrow \text{graph}[\sigma_j]$ for some $j \in [1, m]$. From the formula-to-graph translation Prop. A.17 we know that $H$ corresponds to SL formula $\text{graph}[\phi] = H$ and by the definition of the translation procedure we then know that there exists a subformula $\phi'$ of $\phi$ which corresponds to $H'$ (namely $\text{graph}[\phi'] = H'$). As every production rule $X_{\sigma'} \rightarrow \text{graph}[\sigma_j]$ in $hrg[\Gamma']$ results from a disjunct $\sigma_j$ in an SL predicate definition in $\Gamma$, there exists an abstraction $\phi \Rightarrow hrg[\Gamma'] \phi_1$ using disjunct $\phi_j$ (and accordingly for $H \Rightarrow hrg[\Gamma'] H_2$ and $\phi \Rightarrow hrg[\Gamma'] \phi_2$). But then as $\Gamma$ is confluent under abstraction there exist SL formulae $\phi_1, \phi_2$ such that $\phi_j \Rightarrow hrg[\Gamma'] \phi_1, \phi_2 \Rightarrow hrg[\Gamma'] \phi_2$ and $\phi_1$ and $\phi_2$ equal. Assume that the first abstraction step in this sequence utilises disjunct $\phi_j$ of predicate $\sigma'$. We know that for each disjunct of $\sigma'$ there exists a production rule $X_{\sigma'} \rightarrow \text{graph}[\sigma_j']$ and $H_1$ is the corresponding graph to $\phi_1$ (i.e. contains the subgraph corresponding to the subformula $\sigma_j'$ in $\phi_j$) and again accordingly for $\phi_2$ and $H_2$). Then by associativity of hyperedge replacement the same holds for every further abstraction step in the sequence. Therefore there exists $H_1 \Rightarrow hrg[\Gamma'] H_1'$ and $H_2 \Rightarrow hrg[\Gamma'] H_2'$ as $\phi_1$ and $\phi_2$ are equal, i.e. are fulfilled by exactly the same heap-interpretation-pairs for predicate interpretation $\eta$ and $H_1' = \text{graph}[\phi_1], H_2' = \text{graph}[\phi_2]$, $H_1'$ and $H_2'$ must be isomorphic. This is a contradiction to the assumption that $hrg[\Gamma']$ is not backward confluent.

(2) Analogous.
### Aachener Informatik-Berichte

This list contains all technical reports published during the past three years. A complete list of reports dating back to 1987 is available from:

http://aib.informatik.rwth-aachen.de/

To obtain copies please consult the above URL or send your request to:

Informatik-Bibliothek, RWTH Aachen, Ahornstr. 55, 52056 Aachen,
Email: biblio@informatik.rwth-aachen.de

<table>
<thead>
<tr>
<th>Year</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>2011-01</td>
<td>Fachgruppe Informatik: Jahresbericht 2011</td>
</tr>
<tr>
<td>2011-02</td>
<td>Marc Brockschmidt, Carsten Otto, Jürgen Giesl: Modular Termination Proofs of Recursive Java Bytecode Programs by Term Rewriting</td>
</tr>
<tr>
<td>2011-03</td>
<td>Lars Noschinski, Fabian Emmes, Jürgen Giesl: A Dependency Pair Framework for Innermost Complexity Analysis of Term Rewrite Systems</td>
</tr>
<tr>
<td>2011-04</td>
<td>Christina Jansen, Jonathan Heinen, Joost-Pieter Katoen, Thomas Noll: A Local Greibach Normal Form for Hyperedge Replacement Grammars</td>
</tr>
<tr>
<td>2011-06</td>
<td>Johannes Lotz, Klaus Leppkes, and Uwe Naumann: dco/c++ - Derivative Code by Overloading in C++</td>
</tr>
<tr>
<td>2011-07</td>
<td>Shahar Maoz, Jan Oliver Ringert, Bernhard Rumpe: An Operational Semantics for Activity Diagrams using SMV</td>
</tr>
<tr>
<td>2011-08</td>
<td>Thomas Ströder, Fabian Emmes, Peter Schneider-Kamp, Jürgen Giesl, Carsten Fuhs: A Linear Operational Semantics for Termination and Complexity Analysis of ISO Prolog</td>
</tr>
<tr>
<td>2011-09</td>
<td>Markus Beckers, Johannes Lotz, Viktor Mosenkis, Uwe Naumann (Editors): Fifth SIAM Workshop on Combinatorial Scientific Computing</td>
</tr>
<tr>
<td>2011-10</td>
<td>Markus Beckers, Viktor Mosenkis, Michael Maier, Uwe Naumann: Adjoint Subgradient Calculation for McCormick Relaxations</td>
</tr>
<tr>
<td>2011-12</td>
<td>Ingo Felscher, Wolfgang Thomas: On Compositional Failure Detection in Structured Transition Systems</td>
</tr>
<tr>
<td>2011-13</td>
<td>Michael Förster, Uwe Naumann, Jean Utke: Toward Adjoint OpenMP Games via Safety Games</td>
</tr>
<tr>
<td>2011-14</td>
<td>Daniel Nieder, Roman Rabinovich, Martin Zimmermann: Solving Muller Games via Safety Games</td>
</tr>
<tr>
<td>2011-15</td>
<td>Niloofer Safran, Uwe Naumann: Toward Adjoint OpenFOAM</td>
</tr>
<tr>
<td>2011-16</td>
<td>Carsten Fuhs: SAT Encodings: From Constraint-Based Termination Analysis to Circuit Synthesis</td>
</tr>
<tr>
<td>2011-17</td>
<td>Kamal Barakat: Introducing Timers to pi-Calculus</td>
</tr>
<tr>
<td>2011-24</td>
<td>Callum Corbett, Uwe Naumann, Alexander Mitsos: Demonstration of a Branch-and-Bound Algorithm for Global Optimization using McCormick Relaxations</td>
</tr>
</tbody>
</table>
2011-25 Callum Corbett, Michael Maier, Markus Beckers, Uwe Naumann, Amin Ghoheity, Alexander Mitsos: Compiler-Generated Subgradient Code for McCormick Relaxations
2011-26 Honglei Fu: The Complexity of Deciding a Behavioural Pseudometric on Probabilistic Automata
2012-01 Fachgruppe Informatik: Annual Report 2012
2012-02 Thomas Heer: Controlling Development Processes
2012-03 Arne Haber, Jan Oliver Ringert, Bernhard Rumpe: MontiArc - Architectural Modeling of Interactive Distributed and Cyber-Physical Systems
2012-04 Marcus Gelderie: Strategy Machines and their Complexity
2012-05 Thomas Ströder, Fabian Emmes, Jürgen Giesl, Peter Schneider-Kamp, and Carsten Fuhs: Automated Complexity Analysis for Prolog by Term Rewriting
2012-06 Marc Brockschmidt, Richard Musiol, Carsten Otto, Jürgen Giesl: Automated Termination Proofs for Java Programs with Cyclic Data
2012-08 Honglei Fu: Computing Game Metrics on Markov Decision Processes
2012-09 Dennis Guck, Tingting Han, Joost-Pieter Katoen, and Martin R. Neuhaüser: Quantitative Timed Analysis of Interactive Markov Chains
2012-10 Uwe Naumann and Johannes Lotz: Algorithmic Differentiation of Numerical Methods: Tangent-Linear and Adjoint Direct Solvers for Systems of Linear Equations
2012-12 Jürgen Giesl, Thomas Ströder, Peter Schneider-Kamp, Fabian Emmes, and Carsten Fuhs: Symbolic Evaluation Graphs and Term Rewriting — A General Methodology for Analyzing Logic Programs
2012-15 Uwe Naumann, Johannes Lotz, Klaus Leppkes, and Markus Towara: Algorithmic Differentiation of Numerical Methods: Tangent-Linear and Adjoint Solvers for Systems of Nonlinear Equations
2012-17 Viet Yen Nguyen: Trustworthy Spacecraft Design Using Formal Methods
2013-02 Michael Reke: Modellbasierte Entwicklung automobiler Steuerungssysteme in Klein- und mittelständischen Unternehmen
2013-03 Markus Towara and Uwe Naumann: A Discrete Adjoint Model for OpenFOAM
2013-05 Andreas Rausch and Marc Sihling: Software & Systems Engineering Essentials 2013
2013-06 Marc Brockschmidt, Byron Cook, and Carsten Fuhs: Better termination proving through cooperation
2013-07 André Stollenwerk: Ein modellbasiertes Sicherheitskonzept für die extrakorporale Lungenunterstützung
2013-08  Sebastian Junges, Ulrich Loup, Florian Corzilius and Erika Ábrahám: On Gröbner Bases in the Context of Satisfiability-Modulo-Theories Solving over the Real Numbers
2013-12  Marc Brockschmidt, Fabian Emmes, Stephan Falke, Carsten Fuhs, and Jürgen Giesl: Alternating Runtime and Size Complexity Analysis of Integer Programs
2013-14  Jörg Brauer: Automatic Abstraction for Bit-Vectors using Decision Procedures
2013-20  Jacob Palczynski: Time-Continuous Behaviour Comparison Based on Abstract Models
2014-02  Daniel Merschen: Integration und Analyse von Artefakten in der modellbasierten Entwicklung eingebetteter Software
2014-03  Uwe Naumann, Klaus Leppkes, and Johannes Lotz: dco/c++ User Guide
2014-04  Namit Chaturvedi: Languages of Infinite Traces and Deterministic Asynchronous Automata
2014-05  Thomas Ströder, Jürgen Giesl, Marc Brockschmidt, Florian Frohn, Carsten Fuhs, Jera Hensel, and Peter Schneider-Kamp: Automated Termination Analysis for Programs with Pointer Arithmetic
2014-06  Esther Horbert, Germán Martin García, Simone Frintrop, and Bastian Leibe: Sequence Level Salient Object Proposals for Generic Object Detection in Video

* These reports are only available as a printed version.
Please contact bibli@informatik.rwth- aachen.de to obtain copies.