RWTH Aachen

Department of Computer Science
Technical Report

Generating Inductive Predicates for Symbolic Execution of Pointer-Manipulating Programs

Christina Jansen and Florian Göbe and Thomas Noll

ISSN 0935–3232 · Aachener Informatik-Berichte · AIB-2014-08

RWTH Aachen · Department of Computer Science · May 2014

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 Göbe
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 concretise 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 Xisa [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], [15]. 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 transductions 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, $S^\star$ is the set of all finite sequences (strings) over S, including the empty sequence ε. For $s \in S^\star$, the length of s is denoted by $|s|$, the set of all its elements by $\wr s \wr$, and by $s(i)$ ($i \in [1, |s|]$) we refer to the i-th element of s. Given a tuple $t = (A, B, C, \dots)$, we write A_t, B_t etc. for the components if their names are understood from the context. We represent a function $f : A \to B$ by its set of mappings $\{x \mapsto y \mid \exists x \in A, y \in B : f(x) = y\}$ and write $f_\emptyset$ to denote the empty function, i.e. $f_\emptyset = \emptyset$. Function $f : A \to B$ is lifted to sets $f : 2^A \to 2^B$ and to sequences $f : A^\star \to B^\star$ by pointwise application. The $\uplus$ 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 $\mathsf{Loc} := \mathbb{N}$. To express that a location contains a null reference, we introduce the set $\mathsf{Elem} := \mathsf{Loc} \cup \{\mathbf{null}\}$. A *heap* is a (partial) mapping $h : \mathsf{Loc} \rightharpoonup \mathsf{Elem}$. The set of all heaps is denoted by He.

SL features logical variables, denoted by *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 $i : Var \rightharpoonup \mathsf{Elem}$. We denote the set of all interpretations by Int.

We consider the heap to contain objects. Each object has a number of references to other objects, represented by a finite set Σ 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 $\mathrm{cn}(s), s \in \Sigma$ to denote the ordinal of selector s, where $0 \leq \mathrm{cn}(s) < |\Sigma|$, i.e., under interpretation i the location of selector s of an object x is $i(x) + \mathrm{cn}(s)$. To simplify notation, we assume w.l.o.g. that for a heap containing n objects, locations $1, \dots, n \cdot |\Sigma|$ are used, i.e. we do not distinguish between heaps that differ only in their location numbering. An interpretation $i \in \mathsf{Int}$ is *safe* if $img(i) \subseteq \{1, |\Sigma|+1, 2|\Sigma|+1, \dots\} \cup \{\mathbf{null}\}$. In the following we assume all interpretations to be safe.

We consider a restricted set of SL formulae where negation $\neg$, $\mathbf{true}$ 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 Pred be a set of predicate names. The syntax of SL is given by:

$$
\begin{array}{lll}
E ::= x \mid \mathbf{null} & & \\
P ::= x = y \mid P \wedge P & & \text{pure formulae} \\
F ::= \mathbf{emp} \mid x.s \mapsto E \mid F * F \mid & & \text{heap formulae} \\
\qquad \exists x : F \mid \sigma(E, ..., E) & & \\
S ::= F \mid S \vee S \mid P \wedge S & & \text{SL formulae}
\end{array}
$$

where $x, y \in Var$, $s \in \Sigma$ and $\sigma \in$ Pred. A heap formula of the form $x.s \mapsto E$ is called a *pointer assertion*. SLF denotes the set of all SL formulae.

To simplify notation, we introduce the following abbreviations. Given $\phi \in \mathsf{SLF}$, the set $Var(\phi)$ ($FV(\phi)$) collects all (free) variables of ϕ. If $FV(\phi) = \emptyset$, then ϕ is called *closed*. $Atomic(\phi)$ denotes the set of all *atomic* subformulae of ϕ, that is, those of the form $x = y$, $\mathbf{emp}$, $x.s \mapsto E$ and $\sigma(E, ..., E)$. The *predicate calls* in ϕ are given by $pred(\phi) := \{\sigma(x_1, \ldots, x_n) \in Atomic(\phi) \mid \sigma \in \mathrm{Pred}, x_1, \ldots, x_n \in Var \cup \{\mathbf{null}\}\}$. If $pred(\phi) = \emptyset$, then ϕ is called *primitive*.

Predicate definitions are collected in environments.

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

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

The (weak) restriction that $\mathbf{null}$ 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 $\mathbf{null}$ is disallowed in environments, its usage in SL formulae 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 $\Gamma \in \mathrm{Env}$, Pred_Γ is the set of all $\sigma \in \mathrm{Pred}$ with a definition in Γ. 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 η_Γ based on an environment $\Gamma \in \mathrm{Env}$. Intuitively, for a predicate σ defined in Γ, $\eta_\Gamma(\sigma)$ contains all pairs of locations representing the arguments to σ, and heaps that fulfil $\sigma(x_1, \ldots, x_n)$.

Definition 3.5 (Predicate Interpretation): The *predicate interpretation* η_Γ of an environment $\Gamma \in \mathrm{Env}$ is the least fixpoint of the function $f_\Gamma : (\mathrm{Pred} \rightharpoonup 2^{\mathsf{Loc}^* \times \mathsf{He}}) \to (\mathrm{Pred} \rightharpoonup 2^{\mathsf{Loc}^* \times \mathsf{He}})$ w.r.t $\subseteq$ where $f_\Gamma(\eta)(\sigma)$ is given by

$$\{(l, h) \mid h, \{x_1 \mapsto l(1), \ldots, x_n \mapsto l(n)\}, \eta \models \sigma_1 \vee \ldots \vee \sigma_m\}$$

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

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

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

$$\begin{array}{lll}
h, i, \eta_\Gamma \models \mathbf{emp} & \Longleftrightarrow & dom(h) = \emptyset \\
h, i, \eta_\Gamma \models x.s \mapsto \mathbf{null} & \Longleftrightarrow & dom(h) = \{i(x) + \mathrm{cn}(s)\}, \\
 & & h(i(x) + \mathrm{cn}(s)) = \mathbf{null} \\
h, i, \eta_\Gamma \models x.s \mapsto y & \Longleftrightarrow & dom(h) = \{i(x) + \mathrm{cn}(s)\}, \\
 & & h(i(x) + \mathrm{cn}(s)) = i(y) \\
h, i, \eta_\Gamma \models \phi_1 * \phi_2 & \Longleftrightarrow & \text{ex. } h_1, h_2 \colon h_1 \uplus h_2 = h, \\
 & & h_1, i, \eta_\Gamma \models \phi_1,\ h_2, i, \eta_\Gamma \models \phi_2 \\
h, i, \eta_\Gamma \models \exists x : \phi & \Longleftrightarrow & \text{ex. } v \in \mathsf{Loc} : \\
 & & h, i[x \mapsto v], \eta_\Gamma \models \phi \\
h, i, \eta_\Gamma \models \sigma(x_1, \ldots, x_n) & \Longleftrightarrow & ((i(x_1), \ldots, i(x_n)), h) \in \eta_\Gamma(\sigma) \\
h, i, \eta_\Gamma \models \phi \vee \psi & \Longleftrightarrow & h, i, \eta_\Gamma \models \phi \text{ or } h, i, \eta_\Gamma \models \psi \\
h, i, \eta_\Gamma \models x = y \wedge \phi & \Longleftrightarrow & i(x) = i(y) \text{ and } h, i, \eta_\Gamma \models \phi.
\end{array}$$

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 $abs_1 := \phi * x_1.n \mapsto x_2 * x_2.p \mapsto x_1 \overset{abs}{\Longrightarrow} \phi * dll(x_1, x_2)$ defines the transformation of $\phi * x_1.n \mapsto x_2 * x_2.p \mapsto x_1$, where ϕ 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 abs_1 is sound as clearly $\phi * x_1.n \mapsto x_2 * x_2.p \mapsto x_1$ implies $\phi * dll(x_1, x_2)$ whereas $\phi * dll(x_1, x_2) \overset{abs}{\Longrightarrow} \phi * x_1.n \mapsto x_2 * x_2.p \mapsto x_1$ 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 $\exists x_1, x_2, x_3 : x_1.n \mapsto x_2 * x_2.n \mapsto x_3 * x_2.p \mapsto x_1 * x_3.p \mapsto x_2$ can be abstracted to $\exists x_1, x_2, x_3 : x_1.n \mapsto x_2 * x_2.p \mapsto x_1 * dll(x_2, x_3)$ 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 parts 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 ϕ, field s of object x is accessed, the prover is requested to provide all formulae of the form $\phi' * x.s \mapsto y$ which entail ϕ. 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 $\exists r_1, r_2 : \mathit{dll}(r_1, r_2)$ at r_1 yields the following set of formulae (which is a safe over-approximation of the original formula): $\{\exists r_1, r_2 : r_1.n \mapsto r_2 * r_2.p \mapsto r_1, \exists r_1, r_2, r : r_1.n \mapsto r * r.p \mapsto r_1 * \mathit{dll}(r, r_2)\}$, where the two formulae result from unrolling of $\mathit{dll}(r_1, r_2)$ 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 requirements 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 HRGs, 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 σ_i in its predicate body such that either σ_i is primitive or all predicates occurring in $\mathit{pred}(\sigma_i)$ 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: $\{\mathit{list}(x_1, x_2) := \mathit{dll}(x_1, x_2) \vee x_1.n \mapsto x_2 \vee \exists r : x_1.n \mapsto r * \mathit{list}(r, x_2)\}$.

Given an SL formula $\exists r_1, r_2, r_3 : r_1.n \mapsto r_2 * r_2.p \mapsto r_1 * \mathit{list}(r_2, r_3)$, one could abstract the subformula $r_1.n \mapsto r_2$ by means of the second disjunct of list, which yields $\exists r_1, r_2, r_3 : \mathit{list}(r_1, r_2) * r_2.p \mapsto r_1 * \mathit{list}(r_2, r_3)$. 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: $\exists r_1, r_2, r_3 : \mathit{list}(r_1, r_2) * r_2.p \mapsto r_1 * \mathit{list}(r_2, r_3) \overset{unroll}{\Longrightarrow} \exists r_1, r_2, r_3 : r_1.n \mapsto r_2 * r_2.p \mapsto r_1 * r_2.p \mapsto r_1 * \mathit{list}(r_2, r_3)$. This formula is unsatisfiable, as the subformula $r_2.p \mapsto r_1$ appearing twice is separated by means of the $*$-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_i) requiring that for each disjunction in the definition of this predicate exactly this set of pointers is derivable at variable x_i. E.g. the predicate list of the beforementioned example is not typed, as $\mathit{type}(x_2) = \emptyset$ and $\mathit{type}(x_2) = \{p\}$ 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_j ($j \in [1, n]$).

Definition 4.4 (Typedness): Let $\Gamma \in \mathrm{Env}$. A predicate definition $\sigma(x_1, \ldots, x_n) := \sigma_1 \vee \ldots \vee \sigma_m \in \Gamma$ is *typed* if for each x_j ($j \in [1, n]$) there exists a set $type(\sigma, j) \subseteq \Sigma$ such that, for all $h \in \mathsf{He}$ and $i \in \mathsf{Int}$ with $i(x_j) \neq i(x_k)$ ($j \neq k$) and $h, i, \eta_\Gamma \models \sigma(x_1, \ldots, x_n)$, $type(\sigma, j) = \{s \in \Sigma \mid i(x_j) + \mathrm{cn}(s) \in dom(h)\}$. 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 e.g. occur for an environment of the form $\{\sigma(x_1, \ldots, x_n) := \sigma(x_1, \ldots, x_n)\}$. When repeatedly abstracting with σ's only disjunct, 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 $\sigma(x_1, \ldots, x_n) := \sigma_1 \vee \ldots \vee \sigma_m$ is *increasing* if for all σ_j ($j \in [1, m]$) it holds that σ_i contains a pointer assertion or $\#pred(\sigma_j) + \#Var(\sigma_j) > n + 1$, where $\#pred(\sigma_j)$ is the number of predicate calls and $\#Var(\sigma_j)$ the number of variables occurring in σ_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 abstracted 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 $\phi := \exists r : head.n \mapsto r * r.p \mapsto head * dll(r, tail)$ where $head$ and $tail$ are program variables. Assume a program that traverses the list from its tail (the last element of the list). Then the p-pointer of this element is abstracted or, put differently, "hidden" in the predicate call $dll(r, tail)$. Unrolling yields the two formulae $unroll(\phi) := \{\exists r : head.n \mapsto r * r.p \mapsto head * r.n \mapsto tail * tail.p \mapsto r,\ \exists r_1, r_2 : head.n \mapsto r_1 * r_1.p \mapsto head * r_1.n \mapsto r_2 * r_2.p \mapsto r_1 * dll(r_2, tail)\}$, 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 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 $(pred, x_i)$ there exists an environment unrolling a $pred$-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 $\Gamma \in \mathrm{Env}$ be typed. A predicate definition $\sigma(x_1, \ldots, x_n) := \sigma_1 \vee \ldots \vee \sigma_m \in \Gamma$ is *locally concretisable* if, for all $j \in [1, n]$, there exists a $\sigma'(x_1, \ldots, x_n) := \bigvee \Phi_j$ ($\Phi_j \subseteq \{\sigma_1, \ldots, \sigma_m\}$) such that

1) $\forall h \in \mathsf{He}, i \in \mathsf{Int}, \eta_\Gamma \in \mathsf{PI}$: $h, i, \eta_\Gamma \models \sigma(x_1, ..., x_n) \iff h, i, \eta_{\Gamma'} \models \sigma'(x_1, ..., x_n)$ where Γ' results from Γ by renaming σ to σ'.
2) $\forall \phi \in \Phi_j, s \in type(\sigma, j)$: $x_j.s \mapsto y \in Atomic(\phi)$ for some $y \in Var \cup \{\mathbf{null}\}$.

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 Γ with one predicate dll defined by three disjuncts $\sigma_1 := x_1.n \mapsto x_2 * x_2.p \mapsto x_1$, $\sigma_2 := \exists r_1 : x_1.n \mapsto r_1 * r_1.p \mapsto x_1 * dll(r_1, x_2)$, and $\sigma_3 := \exists r_1 : dll(x_1, r_1) * r_1.n \mapsto x_2 * x_2.p \mapsto r_1$. Here we can unroll $dll(x_1, x_2)$ at parameter x_1 by replacing with both σ_1 and σ_2 as described earlier in this section, while unrolling at x_2 employs the disjuncts σ_1 and σ_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 $\exists r : head.n \mapsto r * r.p \mapsto head * dll(r, tail)$. Then abstraction by choosing the first disjunct $\sigma_1 = x_1.n \mapsto x_2 * x_2.p \mapsto x_1$ results in the formula $\exists r : dll(head, r) * dll(r, tail)$. Another possibility for abstraction is to select the second disjunct $\sigma_2 = \exists r_1 : x_1.n \mapsto r_1 * r_1.p \mapsto x_1 * dll(r_1, x_2)$, resulting in $dll(head, tail)$. 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): $\Gamma \in \mathrm{Env}$ is *confluent under abstraction* if, for all closed $\phi \in \mathsf{SLF}$ and abstractions $\phi \overset{abs}{\Longrightarrow}_\Gamma \phi_1$, $\phi \overset{abs}{\Longrightarrow}_\Gamma \phi_2$, there exist $\phi_1 \overset{abs}{\Longrightarrow}^*_\Gamma \phi'_1$ and $\phi_2 \overset{abs}{\Longrightarrow}^*_\Gamma \phi'_2$ 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 Σ be a finite alphabet with ranking function $rk : \Sigma \to \mathbb{N}$. A *(labelled) hypergraph (HG)* over Σ is a tuple $H = (V, E, att, lab, ext)$ where V is a set of or nodes and E a set of hyperedges, $att : E \to V^\star$ maps each hyperedge to a sequence of attached nodes, $lab : E \to \Sigma$ is a hyperedge-labelling function, and $ext \in V^\star$ a (possibly empty) sequence of pairwise distinct external nodes. For $e \in E$, we require $|att(e)| = rk(lab(e))$ and let $rk(e) = rk(lab(e))$. The set of all hypergraphs over Σ is denoted by HG_Σ.

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, Σ, we model (concrete) heaps as HGs over Σ 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 $v_{\mathbf{null}}$ 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)* N (and we let $\Sigma_N = \Sigma \uplus N$).

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 a 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.

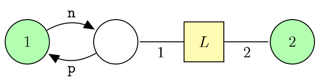


Fig. 1. (Extended) heap as hypergraph

Note that not every HG over Σ 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 HC_{Σ_N}. Thus HC_Σ represents all concrete HCs, i.e., those without nonterminal edges. HCs with external nodes are called *extended* and are represented by the sets HCE_{Σ_N} and HCE_Σ 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 Σ_N is a function $G : N \to 2^{\mathrm{HCE}_{\Sigma_N}}$ with $|ext_H| = rk(X)$ for each $H \in G(X), X \in N$. We call $X \to H$ a *production rule*. The set of all HRGs over Σ_N is denoted by HRG_{Σ_N}.

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.

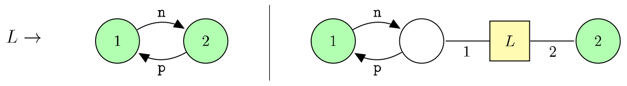


Fig. 2. A grammar for doubly-linked lists

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.

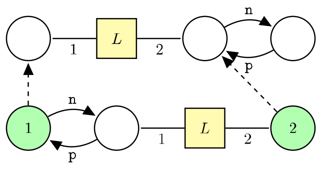


Fig. 3. Replacement of L-edge

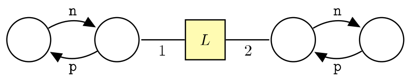


Fig. 4. Resulting hypergraph

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 $G \in \mathrm{HRG}_{\Sigma_N}$, $H, H' \in \mathrm{HG}_{\Sigma_N}$, $K \in G(X)$, $p = X \to K$, and $e \in E_H$ with $lab(e) = X$. H *derives* H' *by* p if H' results from H by replacing e with K. $H \Rightarrow_{e,p} H'$ refers to this derivation. Let $H \Rightarrow_G H'$ if $H \Rightarrow_{e,p} H'$ for some $e \in E_H$, $K \in G(X)$, $p = X \to K$, and let $\Rightarrow_G^\star$ 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 $G \in \mathrm{HRG}_{\Sigma_N}$ and $H \in \mathrm{HG}_{\Sigma_N}$. $L_G(H) = \{K \in \mathrm{HG}_\Sigma \mid \mathit{context}(H) \Rightarrow^\star K\}$ 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

$$X^\bullet = (\{v_1, \ldots, v_n\}, \{e\}, \{e \mapsto v_1 \ldots v_n\}, \{e \mapsto X\}, \varepsilon) \\ \in \mathrm{HC}_{\Sigma_N}.$$

Moreover we define the X*-handle over external nodes*,

$$X^\bullet_{ext} = (\{v_1, \ldots, v_n\}, \{e\}, \{e \mapsto v_1 \ldots v_n\}, \{e \mapsto X\}, \\ v_1 \ldots v_n) \in \mathrm{HCE}_{\Sigma_N}.$$

Thus $L(X^\bullet)$ is the language induced by NT X. For $H \in \mathrm{HC}_{\Sigma_N}$, $L(H)$ denotes the set of derivable concrete HGs. Note that it is not guaranteed that $L(H) \subseteq \mathrm{HC}_\Sigma$, 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): $G \in \mathrm{HRG}_{\Sigma_N}$ is called a *data structure grammar (DSG)* if for all $X \in N$, $L(X^\bullet) \subseteq \mathrm{HC}_\Sigma$. We denote the set of all data structure grammars over Σ_N by DSG_{Σ_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 HC as 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.

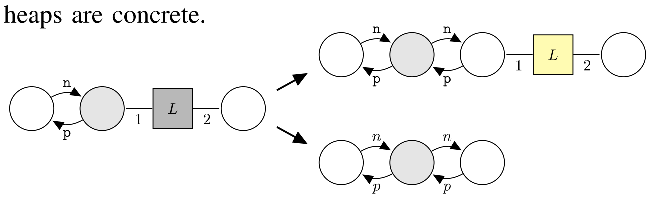


Fig. 5. Concretisation at the shaded node using both L-rules

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 $\overset{abs}{\Longrightarrow}$ and $\overset{abs}{\Longrightarrow}^*$ 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 expressivity 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 $X \to H$ such that either $H \in \mathrm{HC}_\Sigma$ or all elements of $\{Z \in N \mid \exists e \in E_H : lab(e) = Z\}$ are productive. $G \in \mathrm{DSG}_{\Sigma_N}$ is *productive* if all of its NTs are productive.

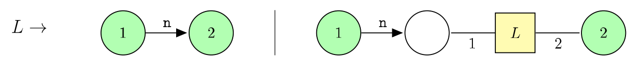


Fig. 6. Untyped DSG extension for singly-linked lists

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 $X \to H$ exactly this set of pointers is derivable at external node $ext_H(j)$.

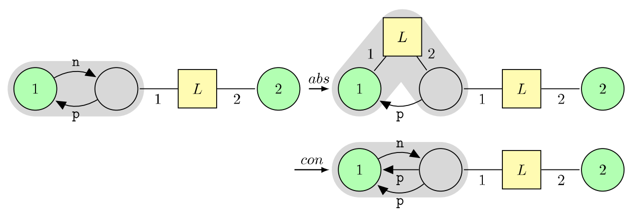


Fig. 7. Abstraction and concretisation yielding an invalid heap configuration

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 $H \in L(X^\bullet)$ with $V_{X^\bullet} = \{v_1, \dots, v_n\}$, $type(X, j) = \{lab(e) \mid \exists e \in E_H : att(e)(1) = v_j\}$. A DSG G is *typed* if each of its NTs is typed.

3) Increasingness: Consider a production rule of the form $X \to X^\bullet_{ext}$. 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 $G \in \mathrm{DSG}_{\Sigma_N}$, $X \in N$ 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 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 $G \in \mathrm{DSG}_{\Sigma_N}$ is *locally concretisable* if for all $X \in N$ there exist grammars $G_{(X,1)}, \cdots, G_{(X,rk(X))}$ such that for all $j \in [1, rk(X)]$, $dom(G_{(X,j)}) = \{X\}$, $G_{(X,j)} \subseteq G(X)$, and

1) $L_{G_{(X,j)} \cup (G \setminus G(X))}(X^\bullet) = L_G(X^\bullet)$, and
2) $\forall a \in type(X, j)$, $H \in G_{(X,j)}(X) : \exists e \in E_H : lab(e) = a \wedge att_H(e)(1) = ext_H(j)$.

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.

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.

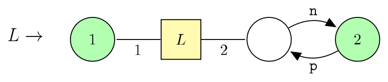


Fig. 8. Ensuring local concretisability

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): $G \in \mathrm{DSG}_{\Sigma_N}$ is *backward confluent* if for each $H \in \mathrm{HC}_{\Sigma_N}$ and abstractions $H \overset{abs}{\Longrightarrow} H_1$ and $H \overset{abs}{\Longrightarrow} H_2$ there exists $K \in \mathrm{HC}_{\Sigma_N}$ such that $H_1 \overset{abs}{\Longrightarrow}{}^* K$ and $H_2 \overset{abs}{\Longrightarrow}{}^* K$.

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 HRGs 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 HRGs 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 $graph[\![.]\!] : \mathsf{SLF} \to \mathrm{HCE}_{\Sigma_N}$, $hrg[\![.]\!] : \mathrm{Env} \to \mathrm{HRG}_{\Sigma_N}$, $env[\![.]\!] : \mathrm{DSG}_{\Sigma_N} \to \mathrm{Env}$, $form[\![.]\!] : \mathrm{HCE}_{\Sigma_N} \to \mathsf{SLF}$. Details can be found in App. A and [21]. In App. A-D we define a criterion for environments which ensures that $hrg[\![.]\!]$ 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 $\phi := \exists x, y : x.n \mapsto y$, which is fulfilled if there exist $v_x, v_y \in \mathsf{Loc}$ such that $dom(h) = \{v_x + \mathrm{cn}(n)\}$ with $h(v_x + \mathrm{cn}(s)) = v_y$. As this does not impose any restriction on v_y, y could reference a location outside h. To bridge this gap we consider interpretations only, that fulfil the following assumption.

Assumption 7.1: Let $\phi \in \mathsf{SLF}$. For all interpretations $i \in \mathsf{Int}$ satisfying ϕ and all $r_1, r_2 \in Var(\phi) \setminus FV(\phi)$ with $r_1 \neq r_2$, it holds that $i(r_1) \neq i(r_2)$ unless the equality is explicitly stated by an equality assertion $r_1 = r_2$ 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 $X \in N$ of rank n, a predicate σ_X with parameters $x_1, \ldots, x_n$ is issued (we use $\rightsquigarrow$ to indicate the translation step):
$env[\![G]\!] \quad N = \{L\}$ and $rk(L) = 2 \rightsquigarrow \sigma_L(x_1, x_2)$
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 σ_X results from disjuncting these formulae.
$env[\![G]\!] \quad G(X) = \{R_1, R_2\} \rightsquigarrow$ disjuncts σ_1, σ_2 and thus $\sigma_L(x_1, x_2) := \sigma_1 \vee \sigma_2$
For the translation of the right-hand sides, first each node is identified by a unique variable: external node j by x_j, internal nodes $v_1, \ldots, v_m$ by $r_1, \ldots, r_m$ and $v_{\mathbf{null}}$ by $\mathbf{null}$. The parameters of the predicate $x_1, \ldots, x_n$ become free variables in the disjuncts, whereas $r_1, \ldots, r_m$ will be existentially bound.
$env[\![G]\!] \quad$ for R_1: no internal nodes $\rightsquigarrow$ no quantifiers in σ_1
$env[\![G]\!] \quad$ for R_2: one internal node identified by $r_1 \rightsquigarrow \sigma_2 := \exists r_1 : \ldots$
Then each edge is translated one-by-one using the node identifiers as variables in the corresponding heap formula. Terminal edges become pointer assertions.
$env[\![G]\!] \quad$ for R_1: n- and p-edge $\rightsquigarrow x_1.n \mapsto x_2$ and $x_2.p \mapsto x_1$
$env[\![G]\!] \quad$ for R_2: n- and p-edge $\rightsquigarrow x_1.n \mapsto r_1$ and $r_1.p \mapsto x_1$
NT edges are translated into predicate calls.
$env[\![G]\!] \quad$ for R_2: L-edge $\rightsquigarrow \sigma_L(r_1, x_2)$
The translation is finalised by glueing together the edge translations using $*$.
$env[\![G]\!] \quad$ for R_1: $R_1 \rightsquigarrow \sigma_1 := x_1.n \mapsto x_2 * x_2.p \mapsto x_1$
$env[\![G]\!] \quad$ for R_2: $R_2 \rightsquigarrow \sigma_2 := \exists r_1 : x_1.n \mapsto r_1 * r_1.p \mapsto x_1 * \sigma_L(r_1, x_2)$
Thus G is translated into $env[\![G]\!] = \{\sigma_L(x_1, x_2) := \sigma_1 \vee \sigma_2\}$.

The translation of environments to HRGs, $hrg[\![.]\!]$, 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.

DSG	**SL**
HC $H \in \mathrm{HC}_{\Sigma_N}$	closed formula $\phi \in \mathsf{SLF}$
extended HC $H \in \mathrm{HCE}_{\Sigma_N}$	formula $\phi \in \mathsf{SLF}$
nonterminal $X \in N$	predicate $\sigma \in \mathrm{Pred}$
terminal edge e ($lab(e) \in \Sigma$)	pointer assertion $x.s \mapsto y$
nonterm. edge e ($lab(e) \in N$)	pred. call $\sigma(x_1, \ldots, x_n)$
X-rules $G(X)$ ($X \in N$)	pred. def. $\sigma(\mathtt{x}_1, \ldots, \mathtt{x}_n) = \ldots$
DSG G	environment Γ

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 $\alpha : \mathsf{He} \mapsto \mathrm{HC}_\Sigma$ (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 fulfils the formula $\sigma(x_1, \ldots, x_n)$ iff a graph representing this heap is contained in the language of the HRG 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 $hrg[\![.]\!] : \mathrm{Env} \to \mathrm{HRG}_{\Sigma_N}$): Let $\Gamma \in \mathrm{Env}$. Then for all $\sigma(x_1, \ldots, x_n) := \phi \in \Gamma$:

$$(l, h) \in \eta_\Gamma(\sigma) \iff \alpha(h) \in L_{hrg[\![\Gamma]\!]}(graph[\![\sigma(x_1, \ldots, x_n)]\!])$$

Theorem 7.4 (Correctness of $env[\![.]\!]$: $\mathrm{DSG}_{\Sigma_N} \to \mathrm{Env}$): Let $G \in \mathrm{DSG}_{\Sigma_N}$. Then for all $X \in N$:

$$H \in L(X^{\bullet}_{ext}) \iff \exists i \in \mathsf{Int} : (l, \alpha^{-1}(H)) \in \eta_{env[\![G]\!]}$$

where $l = i(x_1) \dots i(x_{rk(X)})$.

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

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

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

2) for all $H \in \mathrm{HC}_{\Sigma_N}$:

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

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 an productive, typed, increasing and locally concretisable environment Γ 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 $\Gamma := env[\![G]\!]$ 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

[1] P. Fradet, R. Caugne, and D. L. Métayer, "Static detection of pointer errors: An axiomatisation and a checking algorithm," in *European Symp. on Programming*, ser. LNCS, vol. 1058. Springer, 1996, pp. 125–140.

[2] J. C. Reynolds, "Separation logic: A logic for shared mutable data structures," in *Proc. 17th IEEE Symp. on Logic in Computer Science*. IEEE, 2002, pp. 55–74.

[3] J. Berdine, C. Calcagno, and P. W. O'Hearn, "Symbolic execution with separation logic," in *3rd Asian Symp. on Programming Languages and Systems*, ser. LNCS, vol. 3780. Springer, 2005, pp. 52–68.

[4] M. Dodds and D. Plump, "From hyperedge replacement to separation logic and back," in *Proc. Doctoral Symp. at ICGT*, vol. 16, 2008.

[5] J. Villard, E. Lozes, and C. Calcagno, "Tracking heaps that hop with heap-hop," in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. LNCS, vol. 6015. Springer, 2010, pp. 275–279.

[6] D. Distefano and M. J. Parkinson J, "jStar: Towards practical verification for Java," *ACM Sigplan Notices*, vol. 43, no. 10, pp. 213–226, 2008.

[7] K. Dudka, P. Peringer, and T. Vojnar, "Predator: A practical tool for checking manipulation of dynamic data structures using logic," in *Computer Aided Verification*, ser. LNCS, vol. 6806. Springer, 2011, pp. 372–378.

[8] J. Berdine, C. Calcagno, and P. W. O'Hearn, "Smallfoot: Modular automatic assertion checking with separation logic," in *Formal Methods for Components and Objects*, ser. LNCS, vol. 4111. Springer, 2006, pp. 115–137.

[9] H. Yang, O. Lee, J. Berdine, C. Calcagno, B. Cook, D. Distefano, and P. O'Hearn, "Scalable shape analysis for systems code," in *Computer Aided Verification*, ser. LNCS, vol. 5123. Springer, 2008, pp. 385–398.

[10] S. Magill, M.-H. Tsai, P. Lee, and Y.-K. Tsay, "Thor: A tool for reasoning about shape and arithmetic," in *Computer Aided Verification*, ser. LNCS, vol. 5123. Springer, 2008, pp. 428–432.

[11] B.-Y. E. Chang and X. Rival, "Relational inductive shape analysis," in *Proc. 35th POPL*. ACM, 2008, pp. 247–260.

[12] S. Sagiv, T. W. Reps, and R. Wilhelm, "Parametric shape analysis via 3-valued logic," *ACM Trans. Program. Lang. Syst.*, vol. 24, no. 3, pp. 217–298, 2002.

[13] A. Bouajjani, P. Habermehl, A. Rogalewicz, and T. Vojnar, "Abstract regular tree model checking of complex dynamic data structures," in *Static Anal. Symp.*, ser. LNCS, vol. 4134. Springer, 2006, pp. 52–70.

[14] A. Rensink and E. Zambon, "Pattern-based graph abstraction," in *ICGT*, 2012, pp. 66–80.

[15] M. Dodds, "Graph transformation and pointer structures," Ph.D. dissertation, The University of York, Department of Computer Science, 2008.

[16] O. Lee, H. Yang, and K. Yi, "Automatic verification of pointer programs using grammar-based shape analysis," in *ESOP 2005*, ser. LNCS, vol. 3444. Springer Berlin Heidelberg, 2005, pp. 124–140. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-31987-0_10

[17] B. Courcelle, "The expression of graph properties and graph transformations in monadic second-order logic," in *Handbook of Graph Grammars and Computing by Graph Transformations, Vol. 1: Foundations*. World Scientific, 1997, pp. 313–400.

[18] B. Courcelle and J. Engelfriet, "A logical characterization of the sets of hypergraphs defined by hyperedge replacement grammars," *Math. Syst. Th.*, vol. 28, no. 6, pp. 515–552, 1995.

[19] B. Courcelle, "The monadic second-order logic of graphs I: Recognizable sets of finite graphs," *Information and Computation*, vol. 85, no. 1, pp. 12–75, 1990.

[20] C. Jansen, J. Heinen, J.-P. Katoen, and T. Noll, "A local Greibach normal form for hyperedge replacement grammars," in *5th Int. Conf. on Language and Automata Theory and Applications*, ser. LNCS, vol. 6638. Springer, 2011, pp. 323–335.

[21] F. Göbe, "Transformation von Separation-Logic-Prädikaten durch Hyperkantenersetzungsgrammatiken," Master's thesis, RWTH Aachen University, 2012.

[22] J. Heinen, T. Noll, and S. Rieger, "Juggrnaut: Graph grammar abstraction for unbounded heap structures," in *Proc. 3rd Int. Workshop on Harnessing Theories for Tool Support in Software*, ser. ENTCS, vol. 266. Elsevier, 2010, pp. 93–107.

[23] C. Jansen, J. Heinen, J.-P. Katoen, and T. Noll, "A local Greibach normal form for hyperedge replacement grammars," RWTH Aachen University, Germany, Tech. Rep. AIB 2011-15, January 2011.

[24] J. Nellen, "Konfluenzanalyse und Vervollständigung von Graphersetzungssystemen," Master's thesis, RWTH Aachen University, 2010.

[25] D. Plump, "Checking graph-transformation systems for confluence," *ECEASST*, vol. 26, 2010.

[26] E. Jeltsch and H.-J. Kreowski, "Grammatical inference based on hyperedge replacement," in *Graph-Grammars and Their Application to Computer Science*, 1990, pp. 461–474.

[27] M. Bals, C. Jansen, and T. Noll, "Incremental construction of Greibach normal form for context-free grammars," in *Proc. 2013 Int. Symp. on Theoretical Aspects of Software Engineering*. IEEE, 2013, pp. 165–168.

[28] R. Bornat, C. Calcagno, P. O'Hearn, and M. Parkinson, "Permission accounting in separation logic," in *POPL '05*. ACM, 2005, pp. 259–270.

[29] C. Haack and C. Hurlin, "Separation logic contracts for a Java-like language with fork/join," in *Algebraic Methodology and Software Technology*. Springer, 2008, pp. 199–215.

Appendix A
Translation

In the following section we define translation functions $hrg[\![.]\!]$ and $env[\![.]\!]$ for respectively deriving an HRG from an environment and an environment from a DSG. The correctness of the translation is shown in Sect. A-C. To conclude the section we introduce a criterion for environments guaranteeing that the HRG resulting from the translation satisfies the DSG property.

To give some intuition about the correspondence of environments and DSGs and to simplify the understanding of the translation functions later on, Table II provides a refined overview of the related concepts as given in Table I.

DSG (Syntax & Semantics)	SL (Syntax)	SL (Semantics)
HC $H \in \mathrm{HC}_{\Sigma_N}$	closed formula	(heap, empty int.) $(h, i_\emptyset)$
extended HC $H \in \mathrm{HCE}_{\Sigma_N}$	formula with free variables	(heap, int.) (h, i)
(internal) node v	bound variable r	location in $dom(h)$
(external) node $ext(j)$	free variable x_j	-
nonterminal X	predicate σ	-
terminal edge e ($lab(e) \in \Sigma$)	pointer assertion $x.s \mapsto y$	-
nonterminal edge e ($lab(e) \in N$)	predicate call $\sigma(x_1, \ldots, x_n)$	-
(nonterminal, ordinal) (X, j)	parameter (of predicate call) $\mathtt{x}_j$	-
X-rules, $X \in N$, $G(X)$	predicate def. $\sigma(\mathtt{x}_1, \ldots, \mathtt{x}_n) = \psi$	-
DSG G	environment Γ	-

TABLE II
Overview: DSG vs. environment

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

Definition A.1 (Heap Configuration): A *heap configuration (HC)* is a hypergraph $H = (V, E, att, lab, ext) \in \mathrm{HG}_{\Sigma_N}$ with

- $\Sigma_N = \Sigma \uplus N$, $rk(\Sigma) = \{2\}$ and $ext = \varepsilon$,
- for every $v \in V$ and $s \in \Sigma$, $|\{e \in E \mid lab(e) = s, att(e)(1) = v\}| \leq 1$,
- $\nexists e \in E : lab(e) \in \Sigma \wedge att(e)(1) = v_{\mathbf{null}}$ 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 HC_{Σ_N} (HC_Σ). If an HC satisfies all conditions above except for $ext = \varepsilon$ and if additionally $v_{\mathbf{null}} \notin V_H$, it is an *extended* HC. Accordingly the set of all (concrete) extended HCs is denoted by HCE_{Σ_N} (HCE_Σ).

Definition A.2 (Context): Let $H \in HG_{\Sigma_N}$. We define the *context* of H as $context(H) := \{H' \mid V_{H'} = V_H$ and for arbitrarily many $v_j, v_k \in \wr ext_H \wr . v_j = v_k, E_{H'} = E_H, att_{H'} = att_H, lab_{H'} = lab_H, ext_{H'} = \varepsilon\}$.

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 $H \in \mathrm{HC}_\Sigma$ is a mapping $t : V_H \mapsto 2^{Var \cup \{\mathbf{null}\}}$. The pair (H, t) is called a *tagged heap configuration*. The set of all tagged HCs over Σ is denoted by HC_Σ^T. 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 σ by X_σ. Thus $N := \{X_\sigma \mid \sigma \in \mathrm{Pred}\}$.

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 $\Downarrow$): Let $H = (V_H, E_H, att_H, lab_H, \varepsilon) \in \mathrm{HC}_\Sigma$ and t a tagging of H. The *unification* $(\Downarrow H, t')$ of (H, t) is defined as $\Downarrow H := (V, E_H, att, lab_H, \varepsilon)$ with $V := f(V_H)$, $att(e)(j) := f(att_H(e)(j))$ and $t'(v_f) := \bigcup_{v \in \{v' \in V_H \mid f(v') = v_f\}} t(v)$ where $f : V_H \to V$ such that $\forall v_1, v_2 \in V_H : t(v_1) \cap t(v_2) \neq \emptyset \iff f(v_1) = f(v_2)$.

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

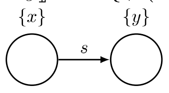
$$\begin{aligned}(H_1,t_1) \bowtie (H_2,t_2) \;:=\; & \Downarrow((V_{H_1} \uplus V_{H_2}, E_{H_1} \uplus E_{H_2},\\ & att_{H_1} \uplus att_{H_2}, lab_{H_1} \uplus lab_{H_2}, \varepsilon),\\ & t_1 \uplus t_2)\end{aligned}$$

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

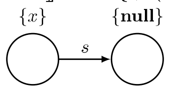
The translation of an SL formula is defined in an inductive way, i.e., for atomic formulae a graph pendant is given, and the translation of a composite SL formula is realised by combining the graphs resulting from the latter using the $join$-operator. Finally, an $expose$-function marks the nodes corresponding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

Definition A.6 ($tgraph[\![.]\!]$: SLF to HC): The translation function $tgraph[\![.]\!]$: $\mathsf{SLF} \mapsto 2^{\mathrm{HC}^{T}_{\Sigma_N}}$ is defined by the following base cases

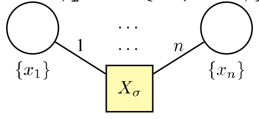
- $tgraph[\![\mathbf{emp}]\!] := \{(H_{\text{emp}}, i_\emptyset)\}$
  $= \{((\emptyset, \emptyset, att_\emptyset,\ lab_\emptyset, \varepsilon), t_\emptyset)\}$
- $tgraph[\![x.s \mapsto y]\!] \;:=\; \{\Downarrow(H_s, t)\}$ where

$(H_s, t) =$ 

- $tgraph[\![x.s \mapsto \mathbf{null}]\!] \;:=\; \{\Downarrow(H_{\mathbf{null}}, t)\}$ where

$(H_{\mathbf{null}}, t) =$ 

- $tgraph[\![\sigma(x_1, \dots, x_n)]\!] \;:=\; \{\Downarrow(H_\sigma, t)\}$ where

$(H_\sigma, t) =$ 

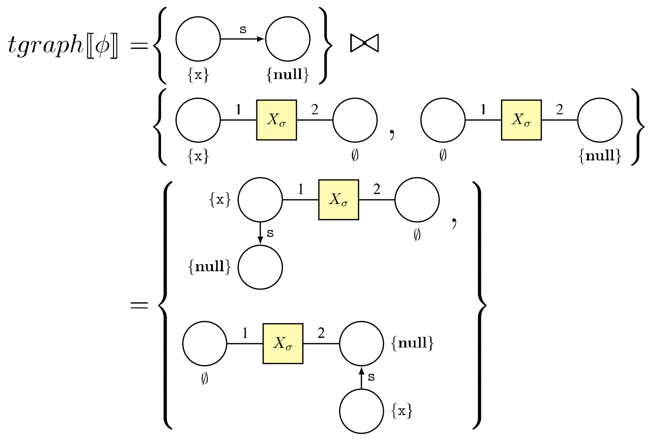
and composite cases

- $tgraph[\![\phi \vee \psi]\!] \;:=\; tgraph[\![\phi]\!] \cup tgraph[\![\psi]\!]$
- $tgraph[\![\phi * \psi]\!] \;:=\; tgraph[\![\phi]\!] \bowtie tgraph[\![\psi]\!]$
- $tgraph[\![\exists x : \phi]\!] \;:=\; \{(H,t) \mid (H,t') \in tgraph[\![\phi]\!],\ t = \{v \mapsto t'(v) \setminus \{x\} \mid v \in dom(t')\}\}$
- $tgraph[\![x = y \wedge \phi]\!] \;:=\; \{\Downarrow(H,t) \mid (H,t') \in tgraph[\![\phi]\!]\}$ with $\forall v \in V_H$,

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

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 $v_{\mathbf{null}}$.

Example A.7: Consider the SL formula $\phi = \mathtt{x}.s \mapsto \mathbf{null} * (\exists \mathtt{z} :\ \sigma(\mathtt{x}, \mathtt{z}) \vee \sigma(\mathtt{z}, \mathbf{null}))$. The translation $tgraph[\![\phi]\!]$ is given by



where 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 $\mathrm{expose}((H,t), x_1 \dots x_n) := ((V_H, E_H, att_H, lab_H, v_1 \dots v_n), t)$ if $t(v_j) = \{x_j\}$, $j \in [1,n]$ where $x_1, \dots, x_n \in \mathit{Var}$.

We can now define the translation of SL formulae to extended HCs 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 $graph[\![.]\!]$: $\mathsf{SLF} \mapsto 2^{\mathrm{HCE}^{T}_{\Sigma_N}}$ is defined by

$$\begin{aligned} graph[\![\phi]\!] := \{&\mathrm{expose}((H,t), x_1 \dots x_n) \mid \\ &(H,t) \in tgraph[\![\phi]\!] \wedge \{x_1, \dots, x_n\} = FV(\phi)\}.\end{aligned}$$

The HRG results from translating the body of predicate σ into a set of HCEs which then provide the rules for nonterminal X_σ.

Definition A.10 ($hrg[\![.]\!]$: Environment to HRG): The translation function $hrg[\![.]\!]$: $\mathrm{Env} \mapsto \mathrm{HRG}_{\Sigma_N}$ is defined by

$$hrg[\![\Gamma]\!] := \bigcup_{\sigma(\mathtt{x}_1, \dots, \mathtt{x}_n) = \psi \in \Gamma} \{X_\sigma \mapsto H \mid (H,t) \in graph[\![\psi]\!]\}.$$

Note that as all graphs resulting from $graph[\![\psi]\!]$ 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, ψ contains no free variables except for $x_1, \dots, x_n$, 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 $H \in \mathrm{HCE}_{\Sigma_N}$. The *tagging* t of H is a mapping $t: V_H \mapsto 2^{Var \cup \{\mathbf{null}\}}$ with

$$t(v) = \begin{cases} \{\mathbf{null}\} & \text{if } v = v_{\mathbf{null}} \\ \{\mathrm{x}_j\} & \text{if } v = ext_H(j) \\ \{r\} & \text{otherwise with } r \text{ a "fresh" variable} \end{cases}$$

We denote the set of tagged HCEs/HCs by HCE^T_Σ/HC^T_Σ and $\mathrm{HCE}^T_{\Sigma_N}$/$\mathrm{HC}^T_{\Sigma_N}$, 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 $form[\![.]\!] : \mathrm{HCE}^T_{\Sigma_N} \mapsto \mathsf{SLF}$ translates an edge e contained in HG H with default tagging t by

$$form[\![e,t]\!] = \begin{cases} x.s \mapsto y & \text{if } s := lab(e) \in \Sigma, \\ & \quad att(e) = vv', \\ & \quad x \in t(v), y \in t(v') \\ \sigma_X(y_1, \ \ldots, \ y_n) & \text{if } X := lab(e) \in N \\ & \quad att(e) = v_1 \ldots v_n, \\ & \quad y_i \in t(v_i) \end{cases}$$

and the hypergraph by

$$form[\![(H,t)]\!] \ = \ \exists r_1 \ldots \exists r_m : \bigodot_{e \in E_H} form[\![e,t]\!]$$

where

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

is the set of variables tagging internal nodes of H and occurring in the resulting formula, and $\bigodot_{e \in E_H} form[\![e,t]\!]$ is defined as

$$\begin{cases} \mathbf{emp} & \text{if } E_H = \emptyset \\ form[\![e_1,t]\!] * \ldots * form[\![e_n,t]\!] & \text{if } E_H = \{e_1, \ldots, e_n\} \end{cases}$$

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 $G \in \mathrm{DSG}_{\Sigma_N}$. The translation $env[\![.]\!] : \mathrm{DSG}_\Sigma \mapsto \mathrm{Env}$ is defined as

$$env[\![G]\!] = \bigcup_{X \in dom(G)} \{\sigma_X(\mathrm{x}_1, \ldots, \mathrm{x}_{rk(X)}) = \bigvee_{H \in G(X)} form[\![(H,t)]\!]\}$$

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 α establishes a connection between heaps $h \in \mathsf{He}$ and concrete hypergraphs representing heaps.

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

$$\begin{aligned} V &= \lfloor dom(h) \rfloor \cup Nil \\ Nil &= \begin{cases} \{v_{\mathbf{null}}\} & \text{if } \exists l \in dom(h) : h(l) = \mathbf{null} \\ \emptyset & \text{otherwise} \end{cases} \\ E &= \{e_{v,s} \mid v \in V, \ s \in \Sigma, \ h(v + \mathrm{cn}(s)) \in dom(h)\} \\ att(e_{v,s}) &= v \lfloor h(v + \mathrm{cn}(s)) \rfloor \\ lab(e_{v,s}) &= s \\ ext &= \varepsilon \end{aligned}$$

where $\lfloor . \rfloor$ 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 ϕ whenever $x \notin Var(\phi)$. Thus from now on when speaking about interpretations, we consider the "least" interpretations only, i.e. we restrict $dom(i)$ to the variables $Var(\phi)$ that occur in ϕ. Considering those interpretations only, α is bijective and therefore its inverse is properly defined. We defined α in such a way that it establishes a connection between the functional and the graphical representation of a heap based on $h \in \mathsf{He}$ only. 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 α_t): The mapping $\alpha_t : \mathsf{He} \times \mathsf{Int} \mapsto \mathrm{HCE}^T_{\Sigma_N}$ is defined by

$$\alpha_t(h,i) = (\alpha(h), t)$$

with

$$t(v) = \{x \in Var \mid i(x)i = v\}.$$

To establish the correspondence between SL and HRGs, our aim is to show that the set of heap-interpretation pairs satisfying a formula ϕ corresponds to the hypergraphs contained in the language of the translation of Φ 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_{ext}): Let $H_{ext} \in \mathrm{HCE}_{\Sigma_N}$, $H \in L(H_{ext})$. The tagging t_{ext} of H is defined as $t_{ext} : V_H \mapsto 2^{\{x_1, \ldots, x_k\}}$ where $k = |ext_H|$ with $t_{ext}(v) :=$

$\{x_j \mid \exists j, H'_{ext} \in context(H_{ext}) : H'_{ext} \to^* H \wedge ext_{H_{ext}}(j) = v\}$.

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 $H \Rightarrow_{e,p} H'$ for $G \in \mathrm{HRG}_{\Sigma_N}$, $H, H' \in HG_{\Sigma_N}$, $p = X \to K, K \in G(X)$ and $e \in E_H$ with $lab(e) = X$ by $H[e/K]$.

Proposition A.17 (graph$[\![.]\!]$: SL Formula to HCE): Let $\phi \in$ SLF. We assume that for the predicate interpretation η_Γ it holds that $\forall (l,h) \in \eta_\Gamma(\sigma) \iff \alpha(h) \in L_{env[\![\Gamma]\!]}(X_{\sigma}{}^{\bullet}_{ext}), i(x_i)i := l(i)$, with Γ an environment, $\sigma \in \mathrm{Pred}_\Gamma$ and $X_\sigma \in N_{env[\![\Gamma]\!]}$. Then it holds for $h \in \mathsf{He}, i \in \mathsf{Int}$: $h, i, \eta_\Gamma \models \phi \iff \alpha_t(h,i) = (H,t), H \in L_{env[\![\Gamma]\!]}(graph[\![\phi]\!])$, and $t(v) \neq \emptyset \iff v \in \wr ext_{graph[\![\Phi]\!]} \int$.

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 $\phi_1 * \phi_2, \phi_1 \vee \phi_2, x = y \wedge \phi, \exists x : \phi$. This is necessary as the expose function, which generates 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 $i(x_j)i$ to be pairwise distinct ($j \in [1,n]$), it holds for $h \in \mathsf{He}, i \in \mathsf{Int}$ that $h, i, \eta_\Gamma \models \phi \iff \alpha_t(h,i) = (H,t), H \in L_{env[\![\Gamma]\!]}(tgraph[\![\phi]\!])$ and $t(v) \neq \emptyset \iff v \in \wr ext_{\mathrm{expose}(tgraph[\![\Phi]\!])} \int$.

To simplify notation later on, we split the claim into

- (*) $h \in \mathsf{He}, i \in \mathsf{Int}$: $h, i, \eta_\Gamma \models \phi \iff \alpha_t(h,i) = (H,t), H \in L_{env[\![\Gamma]\!]}(graph[\![\phi]\!])$
- (**) $t(v) \neq \emptyset \iff v \in \wr ext_{graph[\![\Phi]\!]} \int$

We prove the proposition by structural induction on the SL formula ϕ.

$\phi = \mathbf{emp}$. $\mathbf{emp}$ is only fulfilled by the empty heap, i.e. $dom(h) = \emptyset$, thus $\alpha_t(h,i) = (H,t)$ with $H = (\emptyset, \emptyset, att_\emptyset, lab_\emptyset, \varepsilon)$ and $t = t_\emptyset$. Furthermore $hrg[\![\phi]\!] = \{((\emptyset, \emptyset, att_\emptyset, lab_\emptyset, \varepsilon), t_\emptyset)\}$. Thus the claim holds directly.

$\phi = x.s \mapsto y$. From the semantics of SL and the restriction that variables refer to defined heap locations only, we know that for every heap h satisfying ϕ it holds that $dom(h) = \{i(x)i + \mathrm{cn}(s), i(y)i\}$ and thus $i = i_\emptyset[x \to l][y \to l'], l, l' \in \mathsf{Loc}$ such that $h(l + \mathrm{cn}(s)) = l'$. Thus the set of all $\alpha_t(h,i)$ where $h, i, \eta_\Gamma \models \Phi$ (isomorphic hypergraphs are considered equal) is given by

$$F := \bigcup_{h \in \mathsf{He}, i \in \mathsf{Int} \text{ with } h,i,\eta_\Gamma \models \phi} \{\alpha_t(h,i)\} = \{(H_1,t_1),(H_2,t_2)\}$$

with

$$\begin{aligned} H_1 = (&\{i(x)i, i(y)i\}, \{e_{i(x)i,s}\}, \\ &\{e_{i(x)i,s} \mapsto i(x)ii(y)i\}, \{e_{i(x)i,s} \mapsto s\}, \varepsilon), \\ H_2 = (&\{i(x)i\}, \{e_{i(x)i,s}\}, \\ &\{e_{i(x)i,s} \mapsto i(x)ii(x)i\}, \{e_{i(x)i,s} \mapsto s\}, \varepsilon) \end{aligned}$$

and tagging $t_1 = t_2 = \{l \to x, l' \to y\}$. Furthermore we know that $tgraph[\![x.s \mapsto y]\!] := \{\Downarrow (H_s, t')\} = \{(H_s, t')\}$ with $H_s = (\{v_1, v_2\}, \{e_1\}, \{e_1 \mapsto v_1 v_2\}, \{e_1 \mapsto s\}, \varepsilon)$ and $t' = \{v_1 \mapsto \{x\}, v_2 \mapsto \{y\}\}$. Thus $graph[\![x.s \mapsto y]\!] = H'_s$ with $H'_s = (V_{H_s}, E_{H_s}, att_{H_s}, lab_{H_s}, v_1 v_2)$ and thus (**). Then by definition of the language of a HG it follows directly that $F = L_{env[\![\Gamma]\!]}(H'_s)$ and thus (*) holds.

$\phi = x.s \mapsto \mathbf{null}$. Analogous to the case $\phi = x.s \mapsto y$.

$\phi = \sigma(x_1, \ldots, x_n)$. This follows directly from the assumption in the proposition as the translation of ϕ results in $tgraph[\![\phi]\!] = \{\Downarrow (H_\sigma, t')\}$ with $H_\sigma = (\{v_1, \ldots, v_n\}, \{e_1\}, \{e_1 \mapsto v_1 \ldots v_n\}, \{e_1 \mapsto X_\sigma\}, \varepsilon)$. Thus if the x_j are pairwise distinct, then $\Downarrow (H_\sigma, t') = X_\sigma{}^\bullet$ and the claim directly holds. If some $x_j = x_m$, $j \neq m$ and $j, m \in [1,n]$, then the corresponding nodes are merged during unification. By definition of the translation we know that $t'(v) = \{x_j\}$ and thus $graph[\![\phi]\!] = X^\bullet_{ext}$. Then directly by the assumption for the correspondence of Γ and $env[\![\Gamma]\!]$, (*) holds. Moreover for all taggings t resulting from $\alpha_t(h,i), i := i_\emptyset[x_1 \to l(1), \ldots, x_n \to l(n)], (l,h) \in \eta_\Gamma(\sigma)$ we know that $t(v) = \{x_j \mid v = i(x_j)i\}$. Thus (**) holds.

$\phi = \phi_1 * \phi_2$. This formula is only fulfilled if there exist two disjunct heaps h_1, h_2 and an interpretation i, such that $h_1, i, \eta_\Gamma \models \phi_1$ and $h_2, i, \eta_\Gamma \models \phi_2$. By IH we already know that for all $h \in \mathsf{He}, i \in \mathsf{Int}$ the implication of the propostion is fulfilled for $tgraph[\![\phi_j]\!], j = 1, 2$. We furthermore know from SL semantics that $\forall h, i, \eta_\Gamma \models \Phi$ it holds that $h = h_1 \uplus h_2$, thus $dom(h) = dom(h_1) \uplus dom(h_2)$. By definition of α_t, all taggings resulting from $\alpha_t(h,i)$ with $h, i, \eta_\Gamma \models \Phi_j$ it holds that $t(v) = \{x \in Var(\Phi_j) \mid i(x)i = v\}$. Now according to the translation procedure,

$$\begin{aligned} graph[\![\phi_1 * \phi_2]\!] &:= \mathrm{expose}(tgraph[\![\phi_1]\!] \bowtie tgraph[\![\phi_2]\!], \\ &\qquad x_1 \ldots x_n) \\ &= \mathrm{expose}(\Downarrow ((V_{H_1} \uplus V_{H_2}, E_{H_1} \uplus E_{H_2}, \\ &\qquad att_{H_1} \uplus att_{H_2}, lab_{H_1} \uplus lab_{H_2}, \varepsilon), \\ &\qquad t_1 \uplus t_2), x_1 \ldots x_n) \\ &:= \{(H', t'), (H'', t''), \ldots\} \\ &= F \end{aligned}$$

for every $H_1 \in tgraph[\![\phi_j]\!], H_2 \in tgraph[\![\phi_2]\!]$ and $\{x_1, \ldots, x_n\} = FV(\phi)$. Thus first the two hypergraphs (and their taggings) are merged into one. This yields a hypergraph containing a subgraph H_1 and a subgraph H_2, where H_1 and H_2 are not connected (as node sets and edge sets are disjunct). Furthermore we know that $dom(h) = dom(h_1) \uplus dom(h_2) = V_{H'}$ for $(H', t') \in F$ where during unification for each $j \in [1,n]$ all nodes tagged with $\{x_j\}$, i.e. nodes that correspond to free variables in Φ_1 or Φ_2, have been merged. Thus as the merge of taggings t_1 and t_2 form t_{tmp} and t' is constructed from t_{tmp} by assigning every node the union of the taggings of nodes merged into it, (**) holds. Now as edges are not influenced by the unification operator and only nodes tagged with free variables of ϕ are exposed in $graph[\![\phi]\!]$, (*) follows.

$\phi = \phi_1 \vee \phi_2$. According to the SL semantics, ϕ is fulfilled by heap h, interpretation i and predicate interpretation η_Γ, if either $h, i, \eta_\Gamma \models \phi_1$ or $h, i, \eta_\Gamma \models$

ϕ_2. By IH we get that the claim already holds for $graph[\![\phi_j]\!], j = 1,2$. The translation of ϕ is defined as $graph[\![\phi]\!] = \text{expose}(tgraph[\![\phi_1]\!] \cup tgraph[\![\phi_2]\!], x_1 \dots x_n) = graph[\![\phi_1]\!] \cup graph[\![\phi_2]\!]$ where $\{x_1, \dots, x_n\} = FV(\phi)$. Then either $\alpha_t(h,i) \in L_{env[\![\Gamma]\!]}(graph[\![\Phi_1]\!])$ or $\alpha_t(h,i) \in L_{env[\![\Gamma]\!]}(graph[\![\Phi_2]\!])$. Thus it follows directly that $\alpha_t(h,i) \in L_{env[\![\Gamma]\!]}(graph[\![\Phi_1]\!] \cup graph[\![\Phi_2]\!])$ (*). And trivially (**).

$\phi = x = y \wedge \phi_1$. The SL semantics state that $h, i, \eta_\Gamma \models x = y \wedge \phi_1$ iff $h, i, \eta_\Gamma \models \phi_1$ and $i(x)i = i(y)i$ and from IH we know that $tgraph[\![\phi_1]\!]$ already fulfills the claim. The translation of ϕ adapts the tagging of all $(H,t) \in tgraph[\![\phi_1]\!]$ such that nodes that were tagged with either x or y, are now additionally tagged with both x and y. Thus with $h, i, \eta_\Gamma \models \phi_1 \iff \alpha_t(h,i) \in L_{env[\![\Gamma]\!]}(graph[\![\phi_1]\!])$ and $i(x)i = i(y)i$ we know that $L_{env[\![\Gamma]\!]}(graph[\![\phi]\!])$ yields all graphs from $L_{env[\![\Gamma]\!]}(graph[\![\phi_1]\!])$ where nodes that correspond to variable x or y are merged, thus x and y refer to the same node. As $\alpha_t(h,i)$ creates a node for each location of h and ϕ is only fulfilled if $i(x)i = i(y)i$, the claim follows directly.

$\phi = \exists x : \phi_1$. From SL semantics we know that $h, i, \eta_\Gamma \models \phi$ iff there exists a location l such that $h, i[x \mapsto l], \eta_\Gamma \models \phi_1$. Again from IH we know that the claim holds for ϕ_1. The translation now adapts the tagging from all $(H,t) \in tgraph[\![\phi_1]\!]$ such that x is removed from all tag sets in t. Then we know directly that (**) holds. Moreover we know that $\forall (H', t') \in graph[\![\phi_1]\!] \, \exists (H'', t'') \in graph[\![\phi]\!]$ such that H is isomorphic to H' where all nodes $v \in V_{H'}$ with $t'(v) = \{x\}$ and thus $v \in \lbag ext_{H'} \rbag$, it holds that $t''(v) = \emptyset$ and thus $v \notin \lbag ext_{H''} \rbag$. Then with the property of SL formulae stated in Sect. VII saying that all bound variables implicitly reference different heap location unless states otherwise, the claim follows directly. ■

Proposition A.18 (HC to SLF Formula Translation $form[\![.]\!]$): Let $H \in \mathrm{HC}_{\Sigma_N}$. We assume that for DSG G it holds that for every $X \in N$, $H' \in L_G(X^\bullet_{ext}) \iff \exists i \in \mathsf{Int} : (l, \alpha(H')) \in \eta_{env[\![G]\!]}$ where $l = i(x_1)i \dots i(x_{rk(X)})i$. Then for all $H' \in \mathrm{HC}_\Sigma$ it holds that:

$\alpha^{-1}(H'), i_\emptyset, \eta_G \models form[\![H]\!] \iff H' \in L_G(H).$

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. $E_{H_0} = \emptyset$. As every node in a HCs has to be attached to at least one edge, $V_{H_0} = \emptyset$ (where H_0 is the empty graph). Thus we know that for arbitrary G, $L_G(H_0) = \{H_0\}$. Furthermore $\alpha^{-1}(H_0) = h$ with $dom(h) = \emptyset$. The translation procedure directly provides us with $form[\![H_0, t_\emptyset]\!] = \mathbf{emp}$, which is only fulfilled by the empty heap. Thus the claim holds.

Consider HC H_1 with $E_{H_1} = \{e_1\}$ and let t be the default tagging of H_1.

Case 1: $lab(e_1) \in \Sigma$. Then either $V_{H_1} = \{v_1\}$ or $V_{H_1} = \{v_1, v_{\mathbf{null}}\}$ due to the HC properties. In both cases as H_1 is concrete $L(H_1) = \{H_1\}$.

Assume $V_{H_1} = \{v_1\}$, thus $att(e_1) = v_1 v_1$ and $\alpha^{-1}(H_1) = h$ with $dom(h) = \{1 + \mathrm{cn}(lab(e_1))\}$, $h(1 + \mathrm{cn}(lab(e_1))) = 1 + \mathrm{cn}(lab(e_1))$ and $dom(i) = \emptyset$. From the translation we get $form[\![(H_1, t)]\!] = \exists r_1 : r_1.lab(e_1) \mapsto r_1$, which is by SL semantics fulfilled if there exists $v \in \mathsf{Loc}$ s.t. $h, i_\emptyset[r_1 \mapsto v], \eta_P \models r_1.lab(e_1) \mapsto r_1$ and holds for $v = 1$ (and therefore $r_1.lab(e_1) = 1 + \mathrm{cn}(lab(e_1)))$ only. Thus the claim holds.

Assume $V_{H_1} = \{v_1, v_{\mathbf{null}}\}$ and $att(e_1) = v_1 v_{\mathbf{null}}$. Then $\alpha^{-1}(H_1) = h$ with $dom(h) = \{1 + \mathrm{cn}(lab(e_1))\}$, $h(1 + \mathrm{cn}(lab(e_1))) = \mathbf{null}$. Again from the translation we get $form[\![(H_1, T)]\!] = \exists r_1 : r_1.lab(e_1) \mapsto \mathbf{null}$ which is only fulfilled for $r_1 = 1$ (i.e. if $h(1 + \mathrm{cn}(lab(e_1))) = \mathbf{null}$). Thus the claim holds.

Case 2: $lab(e_1) \in N$. Then $V_{H_1} = \{v_1, \dots, v_n\}$ with $att(e_1) = v_1 \dots v_n$. From the translation we get $form[\![(H_1, t)]\!] = \exists r_1, \dots, r_n : \sigma_{lab(e_1)}(r_1, \dots, r_n)$.

Now either $H_1 \cong lab(e_1)^\bullet$, i.e., the v_j are pairwise distinct. Then the claim directly holds.

Or there exist $v_j, v_k \in V_{H_1}$ s.t. $v_j = v_k$ and thus $t(v_j) = t(v_k)$. Then we know that $L(H_1) = \{H_1[e_1/K] \mid K \in L(lab(e_1)^\bullet_{ext})\}$ and moreover directly by the definition of hyperedge replacement $ext_K(j) = ext_K(k) = v_j$ after replacement of e_1. Furthermore we know that for $form[\![(H_1, t)]\!]$, $i(x_j)i = i(x_k)i$ holds as $x_j = x_k$. Thus for all $H' \in L(H_1)$ there exists an $H \in L(lab(e_1)^\bullet)$ with $\alpha^{-1}(H') = (h', i'), \eta_P \models form[\![(H_1, t)]\!]$ iff $\alpha^{-1}(H) = (h, i), \eta_P \models form[\![(lab(e_1)^\bullet, t)]\!]$ (and the other way around).

For $E_{H_2} = \{e_1, e_2\}$: analogous.

Now assume HC H_{n+1} with $E_{H_{n+1}} = \{e_1, \dots, e_{n+1}\}$ and $V_{H_{n+1}} = \{v_1, \dots, v_m\}$. By construction of the default tagging t for H_{n+1} we know that each node has a unique tag. By translation then $form[\![(H_{n+1}, t)]\!] = \exists r_1, \dots, r_m : form[\![(e_1, t)]\!] * \dots * form[\![(e_{n+1}, t)]\!]$.

Case 1: $\lbag att(e_{n+1}) \rbag \cap \lbag att(e_1) \rbag = \dots = \lbag att(e_{n+1}) \rbag \cap \lbag att(e_n) \rbag = \emptyset$. This means that e_{n+1} together with its attached nodes forms a separate connected component and $Var(form[\![(e_{n+1}, t)]\!]) \cap Var(form[\![(e_j, t)]\!]) = \emptyset$, $j \in [1, n]$. Together with the IH there trivially exist exactly two disjunct heap parts that satisfy $\exists r_1, \dots, r_m : form[\![(e_1, t)]\!] * \dots * form[\![(e_n, t)]\!]$ and $\exists r_1, \dots, r_m : form[\![(e_{n+1}, t)]\!]$, respectively. Thus the claim holds.

Case 2: $\exists j \in [1, n] : \lbag att(e_{n+1}) \rbag \cap \lbag att(e_j) \rbag = \{v_{\mathbf{null}}\}$, and for all other j the sets of attached nodes are disjunct. As $v_{\mathbf{null}}$ is unique, there exists at most one node tagged with $\{\mathbf{null}\}$. The corresponding identifier for this node in the resulting SL formula is $\mathbf{null}$. According to the definition of a heap h, $\mathbf{null}$ cannot be an element of $dom(h)$, as it can only be pointed to but never point to any location in the heap. Then analogously to case 1, there exist exactly two disjunct heap parts satisfying the two parts of the translated formula. Thus the claim holds.

Case 3: $\exists j \in [1, n] : \lbag att(e_{n+1}) \rbag \cap \lbag att(e_j) \rbag = \{v\}$. We can partition H_{n+1} into the subgraphs $H' := (\lbag att(e_{n+1}) \rbag, \{e_{n+1}\}, att_{H_{n+1}}(e_{n+1}), lab_{H_{n+1}}(e_{n+1}), \varepsilon)$ and $H'' := (V_{H_{n+1}} \setminus (\lbag att(e_{n+1}) \rbag \setminus \{v\}), E_{H_{n+1}} \setminus \{e_{n+1}\}, att_{H_{n+1}}|_{E_{H''}}, lab_{H_{n+1}}|_{E_{H''}}, \varepsilon)$, where the claim already holds for H' and H''. Moreover $\alpha^{-1}(H') = h'$, $\alpha^{-1}(H'') = h''$ with $dom(h') \cap dom(h'') = \emptyset$ (as $E_{H'} \cap E_{H''} = \emptyset$) and $\alpha^{-1}(H_{n+1}) = h$ with $h = h' \uplus h''$. This case can be generalised to cuts 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 $H \in \mathrm{HCE}_{\Sigma_N}$. We assume that for DSG G it holds that, for every $X \in N$, $H' \in L_G(X^\bullet_{ext}) \iff \exists i \in \mathsf{Int} : (l, \alpha(H')) \in \eta_{env[\![G]\!]}$ for $l = i(x_1)i \ldots i(x_{rk(X)})i$. Then for all $H' \in \mathrm{HC}_\Sigma$ it holds that

$$\begin{aligned}&H' \in L_G(H) \iff \\ &\exists i \in \mathsf{Int} : \alpha^{-1}(H'), i, \eta_{env[\![G]\!]} \models form[\![H]\!].\end{aligned}$$

Proof: If H has no external nodes, i.e., $ext_H = \varepsilon$, then the claim follows directly from Prop. A.18. Now we assume that H has external nodes $v_1, \ldots, v_n$, i.e., $ext_H = v_1 \ldots v_n$. 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 $form[\![H]\!]$. With Prop. A.18 the claim already holds for $H_{-ext} := (V_H, E_H, att_H, lab_H, \varepsilon)$ and the empty interpretation $i_\emptyset$. Again the translation procedure gives us that $form[\![H]\!]$ and $form[\![H_{-ext}]\!]$ correspond except for the free variables $x_1, \ldots, x_n$ in $form[\![H]\!]$, which are existentially bound in $form[\![H_{-ext}]\!]$ (and renamed to some r_k). Now assume that the free variables $x_1, \ldots, x_n$ in $form[\![H]\!]$ correspond to the existentially bound variables $r_1, \ldots, r_n$ in $form[\![H_{-ext}]\!]$. Then with the implication of SL that all bound variables refer to distinct locations unless the opposite is explicitly states by an equality assertion together with the fact that they can only be referenced in the scope of $form[\![H]\!]$, we know that $x_j \neq r_k$, $j \in [1, n]$ for all r_k from the set of bound variables in $form[\![H]\!]$ (as by definition of the tagging it cannot happen that $form[\![H]\!]$ contains subformula $x_j = r_k$). Then we know that for each $H' \in L_G(H)$ there exists an interpretation i (namely the one resulting from $\alpha_t^{-1}(H', t_{ext}) = (h, i)$ as $dom(t_{ext}) = \{v_1, \ldots, v_n\}$ 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 i 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 $\phi_1 \vee \ldots \vee \phi_m$ 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 $\Downarrow$ that replaces each nonterminal edge in a hypergraph by all eligible right-hand sides.

Definition A.20 (Complete Derivation Step $\Downarrow$): We define a *complete derivation step* of a HG H via the function $\Downarrow_P$: $\mathrm{HG}_{\Sigma_N} \to 2^{\mathrm{HG}_{\Sigma_N}}$, where $\Downarrow_P(H) := \{H' \in \mathrm{HG}_{\Sigma_N} \mid H' = H[e_1\backslash K_1] \ldots [e_k\backslash K_k]$ with $\{e_1, \ldots, e_k\} = E_H \backslash \{e \mid lab(e) \in \Sigma\}$ for arbitrary K_j with $K_j \in P(lab(e_j))\}$.

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

Definition A.21 (HRG Language: Inductive): For an HRG G we define the function $f_G : (N \to 2^{HG_\Sigma}) \to (N \to 2^{HG_\Sigma})$ where $f_G(P_k)(X) := \{H' \in HG_\Sigma \mid \exists R \in P(X) : H' \in \Downarrow_{P_k} (R)\} := P_{k+1}, X \in N$ and $P_0 := P_\emptyset$.

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 $H \in HG_{\Sigma_N}$ and $G \in \mathrm{HRG}_{\Sigma_N}$. Then $L_G(H) = L_G(context(H)) = context(L(H))$

Proof: This directly follows from the associativity of HRG replacement and the fact that $context(context(H)) = context(H)$. ■

Lemma A.23: The least fixpoint of f_G, $lfp(f_G)$, contains all graphs $H \in HG_\Sigma$ that can be derived from $X^\bullet_{ext}, \forall X \in N$, i.e. $X^\bullet_{ext} \Rightarrow^* H$. Thus $L_G(X^\bullet_{ext}) = context(lfp(f_G)(X))$.

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_Γ and $f_{hrg[\![\Gamma]\!]}$): Let $\Gamma \in$ Env. Then for all $\sigma \in Pred_\Gamma$ the following holds:

$$\begin{aligned}&(l, h) \in f_\Gamma(\eta_k)(\sigma) \iff \\ &\alpha(h) \in L(f_{hrg[\![\Gamma]\!]}(P_k)(graph[\![\sigma(x_1, \ldots, x_n)]\!])).\end{aligned}$$

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

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

- $k = 0$. Then $f_\Gamma(\eta_0 = \eta_\emptyset)(\sigma) = \{(l, h) \mid h, i_\emptyset[x_1 \to l(1), \ldots, x_n \to l(n)], \eta_0 \models \Phi_1 \vee \ldots \vee \Phi_m\}$, and thus

  $$\begin{aligned}&(l, h) \in f_\Gamma(\eta_0)(\sigma) \iff \\ &h, i_\emptyset[x_1 \to l(1), \ldots, x_n \to l(n)], \eta_0 \models \Phi_1 \vee \ldots \vee \Phi_m.\end{aligned}$$

  With Prop. A.17 we know that $\forall \Phi$: $h, i, \eta_\Gamma \models \Phi \iff \alpha(h) \in L_{hrg[\![\Gamma]\!]}(graph[\![\Phi]\!])$ if Γ and $hrg[\![\Gamma]\!]$ fulfil the assumption of the proposition. Thus as η_0 and P_0 trivially fulfil the assumption of Prop. A.17:

$$(l,h) \in f_\Gamma(\eta_0)(\sigma) \iff \\ \alpha(h) \in L_{P_0}(graph[\![\Phi_1 \vee \ldots \vee \Phi_m]\!])$$

Moreover $L_{P_0}(graph[\![\Phi_1 \vee \ldots \vee \Phi_m]\!]) = \{H \in \mathrm{HG}_\Sigma \mid H \in context(graph[\![\Phi_1]\!] \cup \ldots \cup graph[\![\Phi_m]\!])\}$. According to the definition of the translation, $P(X_\sigma) = graph[\![\Phi_1]\!] \cup \ldots \cup graph[\![\Phi_m]\!]$. Thus $L_{P_0}(graph[\![\Phi_1 \vee \ldots \vee \Phi_m]\!]) = L_{P_0}(\{R \in \mathrm{HG}_\Sigma \mid \exists R \in P(X_\sigma)\})$. As this set contains terminal graphs only and P_0 is the empty function,

$$(l,h) \in f_\Gamma(\eta_0)(\sigma) \iff \\ \alpha(h) \in L_{hrg[\![\Gamma]\!]}(\{H \in \mathrm{HG}_\Sigma \mid \\ \exists R \in P(X_\sigma) : H \in \Downarrow^1_{P_0}(R)\}),$$

which we can rephrase directly into

$$(l,h) \in f_\Gamma(\eta_0)(\sigma) \iff \alpha(h) \in L_{hrg[\![\Gamma]\!]}(f(P_0)(X_\sigma)).$$

- $k \to k+1$. We have to show that

$$(l,h) \in f_\Gamma^{k+1}(\eta_0) \iff \\ \alpha(h) \in L_{hrg[\![\Gamma]\!]}(f^{k+1}_{hrg[\![\Gamma]\!]}(P_0)(graph[\![\sigma(x_1,\ldots,x_n)]\!]))$$

By applying the definition of f_Γ we get

$$(l,h) \in f_\Gamma^{k+1}(\eta_0) \iff (l,h) \in f_\Gamma(f_\Gamma^k(\eta_0))(\sigma)$$

where $f_\Gamma(f_\Gamma^k(\eta_0))(\sigma) = \{(l',h' \mid h', i_\emptyset[x_1 \to l'(1),\ldots,x_n \to l'(n)], f_\Gamma^k(\eta_0)) \models \Phi_1 \vee \ldots \vee \Phi_m\}$. By ind. hyp. we know that the assumption of Prop. A.17 is fulfilled for $f_\Gamma^k(\eta_0)$ and $f^k_{hrg[\![\Gamma]\!]}(P_0)$ and thus

$$h, i_\emptyset[x_1 \to l(1),\ldots,x_n \to l(n)], f_\Gamma^k(\eta_0) \models \Phi_1 \vee \ldots \vee \Phi_m \\ \iff \alpha(h) \in L_{f^k_{hrg[\![\Gamma]\!]}}(graph[\![\Phi_1 \vee \ldots \vee \Phi_m]\!])$$

Then by definition of the translation of $\vee$ we know

$$h, i_\emptyset[x_1 \to l(1),\ldots,x_n \to l(n)], f_\Gamma^k(\eta_0) \models \Phi_1 \vee \ldots \vee \Phi_m \\ \iff \alpha(h) \in L_{f^k_{hrg[\![\Gamma]\!]}}(graph[\![\Phi_1]\!] \cup \ldots \cup graph[\![\Phi_m]\!])$$

As, for all $X \in dom(f_{hrg[\![\Gamma]\!]})$, $f^k_{hrg[\![\Gamma]\!]}(X)$ by definition contains only terminal graphs, we know that

$$L_{f^k_{hrg[\![\Gamma]\!]}}(graph[\![\Phi_1]\!] \cup \ldots \cup graph[\![\Phi_m]\!]) = \\ L_{hrg[\![\Gamma]\!]}(\{H \mid H \in \Downarrow^1_{f^k_{hrg[\![\Gamma]\!]}}(graph[\![\Phi_1]\!])\} \cup \ldots \cup \\ \{H \mid H \in \Downarrow^1_{f^k_{hrg[\![\Gamma]\!]}}(graph[\![\Phi_m]\!])\}).$$

As $P(X_\sigma) = \{H \mid H \in graph[\![\Phi_1]\!] \cup \ldots \cup graph[\![\Phi_m]\!]\}$ we rephrase using the definition of f_Γ and $f_{hrg[\![\Gamma]\!]}$ into

$$(l,h) \in f_\Gamma^{k+1}(\eta_0)(\sigma) \iff \\ \alpha(h) \in L_{hrg[\![\Gamma]\!]}(f^{k+1}_{hrg[\![\Gamma]\!]}(P_0)(graph[\![\sigma(x_1,\ldots,x_n)]\!]))$$

Then directly with (*)

$$(l,h) \in f_\Gamma^{k+1}(\eta_0)(\sigma) \iff \\ \alpha(h) \in L_{hrg[\![\Gamma]\!]}(f^{k+1}_{hrg[\![\Gamma]\!]}(P_0)(X_{\sigma\overset{\bullet}{e}xt}))$$

■

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

$$h, i, \eta_\Gamma \models \sigma(x_1,\ldots,x_n) \iff \\ \alpha(h) \in L_{hrg[\![\Gamma]\!]}(graph[\![\sigma(x_1,\ldots,x_n)]\!]).$$

Proof: From $graph[\![.]\!]$ we know that $graph[\![\sigma(x_1,\ldots,x_n)]\!]$ yields the HG $X_{\sigma\overset{\bullet}{e}xt}$. Moreover from the semantics of SL we know that $h,i,\eta \models \sigma(x_1,\ldots,x_n) \iff (l := i(x_1)\ldots i(x_n)i, h) \in \eta$. As the least fixpoint of f_Γ yields η_Γ and additionally $L_{hrg[\![\Gamma]\!]}(X_{\sigma\overset{\bullet}{e}xt}) = context(\{\bigcup_{X \in N_{hrg[\![\Gamma]\!]}} lpf(f_{hrg[\![\Gamma]\!]})(X)\})$, the claim follows directly from Lemma A.22 and Prop. A.17. ■

Proposition A.26 (Correspondence of f_G and $f_{env[\![G]\!]}$): Let $G \in \mathrm{DSG}_{\Sigma_N}$ with nonterminals N. Then $H \in L(f_G(P_k)(X)) \iff \alpha_t^{-1}(H, t_{ext}) = (h,i), l := i(x_1)i,\ldots,i(x_n)i$ and $(l,h) \in f_{env[\![G]\!]}(\eta_k)(\sigma_X)$, $\forall X \in N$ with $rk(X) = n$.

Proof: (*) From the translation function $env[\![.]\!]$ we directly know that for HRG G for each $P(X) := \{R_1,\ldots,R_m\}, X \in N$ with $rk(X) = n$ there exists a predicate definition $\sigma_X(x_1,\ldots,x_n) := form[\![R_1]\!] \vee \ldots \vee form[\![R_m]\!]$ in $env[\![G]\!]$.

Proof by induction over the steps k of the fixpoint iteration.

- $k = 0$. Then $f_G(P_0)(X) = \{H' \in \mathrm{HG}_\Sigma \mid \exists R \in P(X) : H' \in \Downarrow_{P_0}(R)\} = \{R \in \mathrm{HG}_\Sigma \mid \exists R \in P(X)\} := \{R'_1,\ldots,R'_p\}$. Thus $H \in L_G(f_G(P_0)(X)) \iff H \in L_G(R'_1) \cup \ldots \cup L_G(R'_p)$. W.l.o.g. assume that $H \in L_G(R'_j), j \in [1,p]$.

  From Prop. A.18 we know that
  $$H \in L_G(R'_j) \\ \iff \alpha_t^{-1}(H, t_{ext}), \eta_0 \models form[\![R'_j]\!]$$
  and thus
  $$H \in L_G(R'_1) \cup \ldots \cup L_G(R'_p) \\ \iff \alpha_t^{-1}(H, t_{ext}), \eta_0 \models form[\![R'_1]\!] \text{ or } \ldots \text{ or } \\ \alpha_t^{-1}(H, t_{ext}), \eta_0 \models form[\![R'_p]\!].$$
  From the semantics of SL we get
  $$H \in L_G(R'_1) \cup \ldots \cup L_G(R'_p) \\ \iff \alpha_t^{-1}(H, t_{ext}), \eta_0 \models form[\![R'_1]\!] \vee \ldots \vee form[\![R'_p]\!].$$
  Furthermore for each $R'_j \in \{R'_1,\ldots,R'_p\}$ we know that $|ext_{R'_j}| = n$ for $rk(X) = n$ and by construction of t_{ext} we know then that $dom(i) = \{x_1,\ldots,x_n\}$. Then by definition of $f_{env[\![G]\!]}$ and (*) it follows that
  $$H \in L_G(R'_1) \cup \ldots \cup L_G(R'_p) \\ \iff \alpha_t^{-1}(H, t_{ext}) = (h,i), (l,h) \in f_{env[\![G]\!]}(\eta_0)(\sigma_X).$$
  for $l := i(x_1)i,\ldots,i(x_n)i$.
- $k = k+1$.

  We have to show that for all $H \in \mathrm{HC}_\Sigma$, $X \in N$:
  $$H \in L(f_G^{k+1}(P_0)(X)) \\ \iff \alpha_t^{-1}(H, t_{ext}) = (h,i), (l,h) \in f^{k+1}_{env[\![G]\!]}(\eta_0)(\sigma_X)$$

for $l = i(x_1)i, \ldots, i(x_n)i$. For $f_G^k(P_0)$ and $f^k_{env[\![G]\!]}$ we already know by ind. hyp. that the proposition holds. With Prop. A.18 we know that for all $H, H' \in HC_\Sigma$:

$$\begin{aligned}&H' \in L_G(H)\\ \iff &\alpha^{-1}(H'), \eta_{env[\![G]\!]} \models form[\![H]\!].\end{aligned}$$

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

$$\begin{aligned}&H' \in L_{f^k_G(P_0)}(R_j)\\ \iff &\alpha_t^{-1}(H', t_{ext}), f^k_{env[\![G]\!]}(\eta_0) \models form[\![R_j]\!]\end{aligned}$$

and therefore

$$\begin{aligned}&H' \in L_{f^k_G(P_0)}(R_1) \cup \ldots \cup L_{f^k_G(P_0)}(R_m)\\ \iff &\alpha_t^{-1}(H', t_{ext}), f^k_{env[\![G]\!]}(\eta_0) \models form[\![R_1]\!]\\ &\text{or } \ldots \text{ or}\\ &\alpha_t^{-1}(H', t_{ext}), f^k_{env[\![G]\!]}(\eta_0) \models form[\![R_m]\!].\end{aligned}$$

As $\Downarrow^1_{f^k_G(P_0)}(R_j)$ 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 $\vee$ we know that

$$\begin{aligned}&H' \in L_G(\Downarrow^1_{f^k_G(P_0)}(R_1)) \cup \ldots \cup L_G(\Downarrow^1_{f^k_G(P_0)}(R_m))\\ \iff &\alpha_t^{-1}(H', t_{ext}), f^k_{env[\![G]\!]}(\eta_0) \models form[\![R_1]\!] \vee \ldots \vee\\ &\qquad\qquad form[\![R_m]\!].\end{aligned}$$

We rephrase into

$$\begin{aligned}&H' \in L_G(\{H'' \in \mathrm{HG}_\Sigma \mid H'' \in \Downarrow^1_{f^k_G(P_0)}(R_1)\})\\ &\cup \ldots \cup\\ &L_G(\{H'' \in \mathrm{HG}_\Sigma \mid H'' \in \Downarrow^1_{f^k_G(P_0)}(R_m)\})\\ \iff &\alpha_t^{-1}(H', t_{ext}) = (h, i), l := i(x_1)i, \ldots, i(x_n)i,\\ &(l, h) \in \{(l', h' \mid h', i_\emptyset[x_1 \to l(1), \ldots, x_n \to l(n)],\\ &f^k_{env[\![G]\!]}(\eta_0) \models form[\![R_1]\!] \vee \ldots \vee form[\![R_m]\!]\}.\end{aligned}$$

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

$$\begin{aligned}&H' \in L_G(\{H' \in \mathrm{HG}_\Sigma \mid\\ &\qquad \exists R \in P(X) : H' \in \Downarrow^1_{f^k_G(P_0)}(R)\})\\ \iff &\alpha_t^{-1}(H', t_ext) = (h, i), l := i(x_1)i, \ldots, i(x_n)i,\\ &(l, h) \in \{(l', h' \mid h', i_\emptyset[x_1 \to l(1), \ldots, x_n \to l(n)],\\ &f^k_{env[\![G]\!]}(\eta_0) \models form[\![R_1]\!] \vee \ldots \vee form[\![R_m]\!]\}.\end{aligned}$$

Then directly by definition of f_G and $f_{env[\![G]\!]}$

$$\begin{aligned}&H' \in L_G(f^{k+1}_G(P_0)(X))\\ \iff &\alpha_t^{-1}(H', t_{ext}) = (h, i), (l, h) \in f^{k+1}_{env[\![G]\!]}(\eta_0)(\sigma_X)\end{aligned}$$

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

■

Theorem A.27 (Correctness of env[\![.]\!]: HRG to Environment): Let $G \in \mathrm{DSG}_{\Sigma_N}$. Then for all $X \in N$ the following holds:

$$\begin{aligned}&H \in L(X^\bullet_{ext}) \iff\\ &\exists i : \alpha^{-1}(H) = (h, i_0) \wedge h, i, \eta_{env[\![G]\!]} \models form[\![X^\bullet_{ext}]\!].\end{aligned}$$

Proof: We know that $form[\![X^\bullet_{ext}]\!] = \sigma_X(x_1, \ldots, x_n)$ for $rk(X) = n$ and thus by semantics of SL we know that $\sigma_X(x_1, \ldots, x_n)$ is fulfilled by (h, i, η) iff $(l, h) \in \eta(\sigma_X)$ where $l := i(x_1) \ldots i(x_n)$. Furthermore from [15] we know that η is defined as the least fixpoint of the function $f_{env[\![G]\!]}$. Moreover L_G is defined as $L_G(X^\bullet_{ext}) = context(\{\bigcup_{X \in N} lfp(f_G)(X)\})$. 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 Γ containing predicate definitions $\sigma_1(x_1, x_2) := x_1.s \mapsto x_2 * x_2.s \mapsto y$ and $\sigma_2(x_1) := \sigma_1(x_1, x_1)$. 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_{σ_2}-labelled edge violates the HC 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 $\sigma(x_1, \ldots, x_n)$ be a predicate call. The k-th unrolling $\overset{unroll}{\Longrightarrow}_k (\sigma(x_1, \ldots, x_n))$ is defined as

- $\overset{unroll}{\Longrightarrow}_0 (\sigma(x_1, \ldots, x_n)) = \sigma(x_1, \ldots, x_n)$
- $\overset{unroll}{\Longrightarrow}_k (\sigma(x_1, \ldots, x_n))$ results by replacing each predicate call of $\overset{unroll}{\Longrightarrow}_{k-1} (\sigma)$ 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 σ cannot yield a formula that defines more than one s-selector for each $x \in Var(\overset{unroll}{\Longrightarrow}_k (\sigma(x_1, \ldots, x_n)))$.

Definition A.29 (Data Structure Environment): We call a predicate definition $\sigma(x_1, \ldots, x_n) := \sigma_1 \vee \ldots \vee \sigma_m$ a *data structure predicate*, if for all $k \in \mathbb{N}$ the following holds: for any disjunct ϕ in the disjunctive normal form of $\overset{unroll}{\Longrightarrow}_k (\sigma(x_1, \ldots, x_n)) \nexists x, y_1, y_2 \in Var, s \in \Sigma : x.s \mapsto y_1 \in \phi$, $x.s \mapsto y_2 \in \phi$ 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 Γ 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 Γ 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 $H \in L_{hrg[\![\Gamma]\!]}((^{\bullet}X))$ 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)
- $\Sigma_N = \Sigma \uplus N$: directly by definition of $graph[\![.]\!]$
- $rk(\Sigma) = \{2\}$: again directly by definition of $graph[\![.]\!]$ (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 derivation in $hrg[\![\Gamma]\!]$ where the node tagged with x_j holds two outgoing s-edges
- $v_{\mathbf{null}}$ 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 $graph[\![.]\!]$
- $ext = \varepsilon$: 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 Γ it holds that the DSG $G := hrg[\![\Gamma]\!]$ is productive.
2) For a productive DSG G it holds that the environment $\Gamma := env[\![G]\!]$ is productive.

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

$n = 1$. Assume $\Gamma := \{\Phi(.))\sigma_1 \vee \ldots \vee \sigma_m\}$. As we know that Φ is productive, we know that there exists one $\sigma_j, j \in [1, m]$ which is primitive, i.e. contains no predicate calls. Thus $graph[\![\sigma_j]\!]$ does not contain any nonterminal edges. Then directly by construction of $hrg[\![\Gamma]\!]$ we know that X_σ is productive.

Let Γ contain $n + 1$ predicate definitions $\phi_1, \ldots, \phi_{n+1}$. Consider an arbitrary $\phi \in \{\phi_1, \ldots, \phi_{n+1}\}$. Then from $\phi := \sigma_1 \vee \ldots \vee \sigma_m$ we pick $\sigma_k, k \in [1, m]$ with the least number of predicate calls different from ϕ. As ϕ is productive, we know that σ_k has calls to at most n different productive predicates all different from ϕ. 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 ϕ (as the productivity of ϕ 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 $graph[\![\sigma_k]\!]$ constructs hypergraphs where nonterminal edges labelled with $X_{\phi_j}, j \in [1, n]$ only arise due to predicate calls ϕ_j occuring in σ_k, again by construction of $hrg[\![\Gamma]\!]$ we directly know that X_ϕ is productive.

As an environment is productive if each single predicate definition is and the only nonterminals $G = hrg[\![\Gamma]\!]$ contains are $\{X_\phi \mid \phi \in \mathrm{Pred}_\Gamma\}$ the claim follows directly.

(2) Analogous. ■

B. Increasingness

Lemma B.2:

1) For an increasing environment Γ it holds that the DSG $G := hrg[\![\Gamma]\!]$ is increasing.
2) For an increasing DSG G it holds that the environment $\Gamma := env[\![G]\!]$ 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_σ is increasing.

For σ 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 σ_i is greater than the number of parameters of σ + 1. In the first case $graph[\![\sigma_i]\!]$ directly translates the pointer assertion into a terminal edge. By definition of $hrg[\![.]\!]$ we know, that each σ_i is translated into a production rule $X_\sigma \to graph[\![\sigma_i]\!]$. Therefore in this case $X_\sigma \to graph[\![\sigma_i]\!]$ is increasing. In the second case $graph[\![\sigma_i]\!]$ generates a new node tagged with x, for each variable x and a nonterminal edge for each predicate call appearing in σ_i. Thus $|V_{graph[\![\sigma_i]\!]}| + |E_{graph[\![\sigma_i]\!]}| > n+1$. As by definition of $hrg[\![.]\!]$ we know that $rk(X_\sigma) = n$, $X_\sigma \to graph[\![\sigma_i]\!]$ is increasing for the second case, too.

As an environment is increasing if each single definition is and G contains exactly those nonterminals X_σ where $\sigma \in Pred_\Gamma$, it follows directly that all nonterminals in G are increasing an thus so is G.

(2) Analogous. ■

C. Typedness

Lemma B.3:

1) For a typed environment Γ it holds that the DSG $G := hrg[\![\Gamma]\!]$ is typed.
2) For a typed DSG G it holds that the environment $\Gamma := env[\![G]\!]$ 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_σ is typed.

Assume that X_σ is not typed. Then there exists an $v \in V_{X_\sigma{}^\bullet}$, HCs $H_1, H_2 \in L(X_\sigma{}^\bullet)$ and $e \in E_{H_1}$ with $att_{H_1}(e)(1) = v$, $lab_{H_1}(e) = s$ such that $\nexists e' \in E_{H_2}$ with $att_{H_2}(e')(1) = v \wedge lab(e') = s$.

As $H_1, H_2 \in L(X_\sigma{}^\bullet)$ we know by the translation correctness (Theorem 7.3) that there exist $h, h' \in \mathsf{He}$ and $i, i' \in \mathsf{Int}$ satisfying $\sigma(x_1, \ldots, x_n)$ such that $\alpha_t(h, i) = (H_1, t1_{ext})$ and $\alpha(h', i') = (H_2, t2_{ext})$. But as we know that $i(x_1) \neq \ldots \neq i(x_n)$, then from the definition of α_t it follows that $i(x_j) + cn(s) \in dom(h)$ and $i(x_j) + cn(s) \notin dom(h')$, 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 Γ, it follows directly that all nonterminals in G are typed an thus so is G.

(2) Analogous.

■

D. *Local Concretisability*

Lemma B.4:

1) For a typed and locally concretisable environment Γ 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 definition $\sigma(x_1, \ldots, x_n) := \sigma_1 \vee \ldots \vee \sigma_m$, that the resulting nonterminal X_σ 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 Γ' be the resulting environment that results from Γ by exchanging all occurences of σ into σ'. We know that during translation of the predicate definition of σ a nonterminal X_σ of rank n with m production rules $X_\sigma \to graph[\![\sigma_1]\!] \mid \ldots \mid graph[\![\sigma_m]\!]$ will be generated. We define the subgrammar $G_{(X_\sigma, j)}$ such that it contains the production rules $X_\sigma \to graph[\![\phi]\!], \phi \in \Phi_j$, i.e. $G_{(X_\sigma,j)} \cup (G \setminus G(X)) = hrg[\![\Gamma']\!]$ where $G := hrg[\![\Gamma]\!]$. For this subgrammar we show that it fulfills the two properties given in Def. 6.4.

1) As Γ is locally concretisable we know that $\{(h, i) \in \mathsf{He} \times \mathsf{Int} \mid h, i, \eta_\Gamma \models \sigma(x_1, \ldots, x_n)\} = \{(h, i) \in \mathsf{He} \times \mathsf{Int} \mid h, i, \eta_{\Gamma'} \models \sigma'(x_1, \ldots, x_n)\}$.
   With Theorem 7.3 we then know that for all those (h, i)-pairs corresponding HCs $\alpha(h)$ are contained in $L_{hrg[\![\Gamma]\!]}(graph[\![\sigma(x_1, \ldots, x_n)]\!])$. As by definition of the translation $graph[\![\sigma(x_1, \ldots, x_n)]\!] = X_\sigma{}^\bullet_{ext}$, it follows that $L_{hrg[\![\Gamma]\!]}(X_\sigma) = L_{hrg[\![\Gamma']\!]}(X_\sigma) = L_{G_{(X_\sigma,j)} \cup (G \setminus G(X_\sigma))}(X_\sigma)$.
2) From the local concretisability of σ we know that $\forall s \in \Sigma$: $s \in type(\sigma, j) \iff \{x_j.s \mapsto y\} \in Atomic(\phi)$ for $\phi \in \Phi_j$ and $y \in Var \cup \{\mathbf{null}\}$. As $graph[\![.]\!]$ translates pointer assertions into a HC 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 $p := X_\sigma \to graph[\![\phi]\!], \phi \in \Phi_j$ derives an outgoing s-labelled edge at external node x_j, i.e. $(\exists e \in E_{graph[\![\Phi]\!]}.lab_{graph[\![\Phi]\!]}(e) = s \wedge att_{graph[\![\Phi]\!]}(e)(1) = ext_{graph[\![\Phi]\!]}(j)$. As the translation procedure generates edges labelle with selectors only from pointer assertions, $ext_{graph[\![\Phi]\!]}(j)$ 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 σ in Γ 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 Γ 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 Γ be confluent under abstraction. Assume that $hrg[\![\Gamma]\!]$ is not backward confluent. Then there exist HCs H, H_1, H_2 such that $H \overset{abs}{\Longrightarrow}_{hrg[\![\Gamma]\!]} H_1$ and $H \overset{abs}{\Longrightarrow}_{hrg[\![\Gamma]\!]} H_2$ and $\nexists H_1', H_2'$ with $H_1 \overset{abs}{\Longrightarrow}{}^*_{hrg[\![\Gamma]\!]} H_1'$, $H_2 \overset{abs}{\Longrightarrow}{}^*_{hrg[\![\Gamma]\!]} H_2'$ and H_1, H_2 isomorphic. From the definition of the translation we know that each predicate definition $\sigma(x_1, \ldots, x_n) := \sigma_1 \vee \ldots \vee \sigma_m$ in Γ is translated into a nonterminal X_σ of rank n with production rules $hrg[\![\Gamma]\!](X_\sigma) = \{graph[\![\sigma_1]\!], \ldots, graph[\![\sigma_m]\!]\}$. Now let us assume that $H \overset{abs}{\Longrightarrow}_{hrg[\![\Gamma]\!]} H_1$ abstracts subgraph H' of H using production rule $X_\sigma \to graph[\![\sigma_j]\!]$ for some $j \in [1, m]$. From the formula-to-graph translation Prop. A.17 we know that H corresponds to SL formula $graph[\![\phi]\!] = H$ and by the definition of the translation procedure we then know that there exists a subformula ϕ' of ϕ wich corresponds to H' (namely $graph[\![\phi']\!] = H$). As every production rule $X_\sigma \to graph[\![\sigma_j]\!]$ in $hrg[\![\Gamma]\!]$ results from a disjunct σ_j in an SL predicate definition in Γ, there exists an abstraction $\phi \overset{abs}{\Longrightarrow}_\Gamma \phi_1$ using disjunct ϕ_j (and accordingly for $H \overset{abs}{\Longrightarrow}_{hrg[\![\Gamma]\!]} H_2$ and $\phi \overset{abs}{\Longrightarrow}{}^*_\Gamma \phi_2$). But then as Γ is confluent under abstraction there exist SL formulae ϕ_1', ϕ_2' such that $\phi_1 \overset{abs}{\Longrightarrow}{}^*_\Gamma \phi_1'$, $\phi_2 \overset{abs}{\Longrightarrow}{}^*_\Gamma \phi_2'$ and ϕ_1' and ϕ_2' equal. Assume that the first abstraction step in this sequence utilises disjunct σ_j' of predicate σ'. We know that for each disjunct of σ' there exists a production rule $X_{\sigma'} \to graph[\![\sigma_j']\!]$ and H_1 is the corresponding graph to ϕ_1 (i.e. contains the subgraph corresponding to the subformula σ_j' in ϕ_1) (and again accordingly for ϕ_2 and H_2). Then by associativity of hyperedge replacement the same holds for every further abstraction step in the sequence. Thus there exists abstractions $H_1 \overset{abs}{\Longrightarrow}{}^*_{hrg[\![\Gamma]\!]} H_1'$ and $H_2 \overset{abs}{\Longrightarrow}{}^*_{hrg[\![\Gamma]\!]} H_2'$. As ϕ_1' and ϕ_2' are equal, i.e. are fulfilled by exactly the same heap-interpretation-pairs for predicate interpretation η_Γ and $H_1' = graph[\![\phi_1']\!]$, $H_2' = graph[\![\phi_2']\!]$, 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

2011-01 *	Fachgruppe Informatik: Jahresbericht 2011
2011-02	Marc Brockschmidt, Carsten Otto, Jürgen Giesl: Modular Termination Proofs of Recursive Java Bytecode Programs by Term Rewriting
2011-03	Lars Noschinski, Fabian Emmes, Jürgen Giesl: A Dependency Pair Framework for Innermost Complexity Analysis of Term Rewrite Systems
2011-04	Christina Jansen, Jonathan Heinen, Joost-Pieter Katoen, Thomas Noll: A Local Greibach Normal Form for Hyperedge Replacement Grammars
2011-06	Johannes Lotz, Klaus Leppkes, and Uwe Naumann: dco/c++ - Derivative Code by Overloading in C++
2011-07	Shahar Maoz, Jan Oliver Ringert, Bernhard Rumpe: An Operational Semantics for Activity Diagrams using SMV
2011-08	Thomas Ströder, Fabian Emmes, Peter Schneider-Kamp, Jürgen Giesl, Carsten Fuhs: A Linear Operational Semantics for Termination and Complexity Analysis of ISO Prolog
2011-09	Markus Beckers, Johannes Lotz, Viktor Mosenkis, Uwe Naumann (Editors): Fifth SIAM Workshop on Combinatorial Scientific Computing
2011-10	Markus Beckers, Viktor Mosenkis, Michael Maier, Uwe Naumann: Adjoint Subgradient Calculation for McCormick Relaxations
2011-11	Nils Jansen, Erika Ábrahám, Jens Katelaan, Ralf Wimmer, Joost-Pieter Katoen, Bernd Becker: Hierarchical Counterexamples for Discrete-Time Markov Chains
2011-12	Ingo Felscher, Wolfgang Thomas: On Compositional Failure Detection in Structured Transition Systems
2011-13	Michael Förster, Uwe Naumann, Jean Utke: Toward Adjoint OpenMP
2011-14	Daniel Neider, Roman Rabinovich, Martin Zimmermann: Solving Muller Games via Safety Games
2011-16	Niloofar Safiran, Uwe Naumann: Toward Adjoint OpenFOAM
2011-17	Carsten Fuhs: SAT Encodings: From Constraint-Based Termination Analysis to Circuit Synthesis
2011-18	Kamal Barakat: Introducing Timers to pi-Calculus
2011-19	Marc Brockschmidt, Thomas Ströder, Carsten Otto, Jürgen Giesl: Automated Detection of Non-Termination and NullPointerExceptions for Java Bytecode
2011-24	Callum Corbett, Uwe Naumann, Alexander Mitsos: Demonstration of a Branch-and-Bound Algorithm for Global Optimization using McCormick Relaxations

2011-25	Callum Corbett, Michael Maier, Markus Beckers, Uwe Naumann, Amin Ghobeity, Alexander Mitsos: Compiler-Generated Subgradient Code for McCormick Relaxations
2011-26	Hongfei 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
2012-07	André Egners, Björn Marschollek, and Ulrike Meyer: Hackers in Your Pocket: A Survey of Smartphone Security Across Platforms
2012-08	Hongfei Fu: Computing Game Metrics on Markov Decision Processes
2012-09	Dennis Guck, Tingting Han, Joost-Pieter Katoen, and Martin R. Neuhäußer: 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
2012-16	Georg Neugebauer and Ulrike Meyer: SMC-MuSe: A Framework for Secure Multi-Party Computation on MultiSets
2012-17	Viet Yen Nguyen: Trustworthy Spacecraft Design Using Formal Methods
2013-01 *	Fachgruppe Informatik: Annual Report 2013
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
2013-04	Max Sagebaum, Nicolas R. Gauger, Uwe Naumann, Johannes Lotz, and Klaus Leppkes: Algorithmic Differentiation of a Complex C++ Code with Underlying Libraries
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
2013-10	Joost-Pieter Katoen, Thomas Noll, Thomas Santen, Dirk Seifert, and Hao Wu: Performance Analysis of Computing Servers using Stochastic Petri Nets and Markov Automata
2013-12	Marc Brockschmidt, Fabian Emmes, Stephan Falke, Carsten Fuhs, and Jürgen Giesl: Alternating Runtime and Size Complexity Analysis of Integer Programs
2013-13	Michael Eggert, Roger Häußling, Martin Henze, Lars Hermerschmidt, René Hummen, Daniel Kerpen, Antonio Navarro Pérez, Bernhard Rumpe, Dirk Thißen, and Klaus Wehrle: SensorCloud: Towards the Interdisciplinary Development of a Trustworthy Platform for Globally Interconnected Sensors and Actuators
2013-14	Jörg Brauer: Automatic Abstraction for Bit-Vectors using Decision Procedures
2013-19	Florian Schmidt, David Orlea, and Klaus Wehrle: Support for error tolerance in the Real-Time Transport Protocol
2013-20	Jacob Palczynski: Time-Continuous Behaviour Comparison Based on Abstract Models
2014-01 *	Fachgruppe Informatik: Annual Report 2014
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 Martín García, Simone Frintrop, and Bastian Leibe: Sequence Level Salient Object Proposals for Generic Object Detection in Video
2014-07	Niloofar Safiran, Johannes Lotz, and Uwe Naumann: Algorithmic Differentiation of Numerical Methods: Second-Order Tangent and Adjoint Solvers for Systems of Parametrized Nonlinear Equations

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