From Separation Logic to Hyperedge Replacement and Back

Mike Dodds

Computer Lab
University of Cambridge, UK

ICGT 2008

Overview

Two approaches to defining classes of structures.

- *Separation logic*: logic for heap properties which enables *local reasoning*.
- *Hyperedge replacement grammars*: context-free graph grammars.

How are they related?

We define a correspondence between formulas in a fragment of separation logic and restricted graph grammars:

- Define a translation g from formulas to grammars, and translation s from grammars to formulas.
- Translations preserve semantics.

This is joint work with Detlef Plump.

Motivation

Formal Properties

- Hyperedge replacement is well understood, while separation logic was developed comparatively recently.
- Separation logic fragment inherits the properties of hyperedge replacement:
  - Decidable membership.
  - Known inexpressible languages.

Practical Application

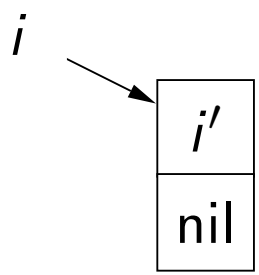
- Hyperedge replacement and separation logic both used for specifying shapes of data structures.
- We want to share shape-checking approaches.

Separation logic (1)

Separation logic formulas define classes of satisfying *heaps*.

- Defined over a set Loc of locations.
- Heap domain: $Elem = Loc \cup \{\text{nil}\}$.
- Heap defined by partial function $h\colon Loc \rightarrow Elem \times Elem$

Represent singleton heap h with $h(i) = (i', \text{nil})$ as:



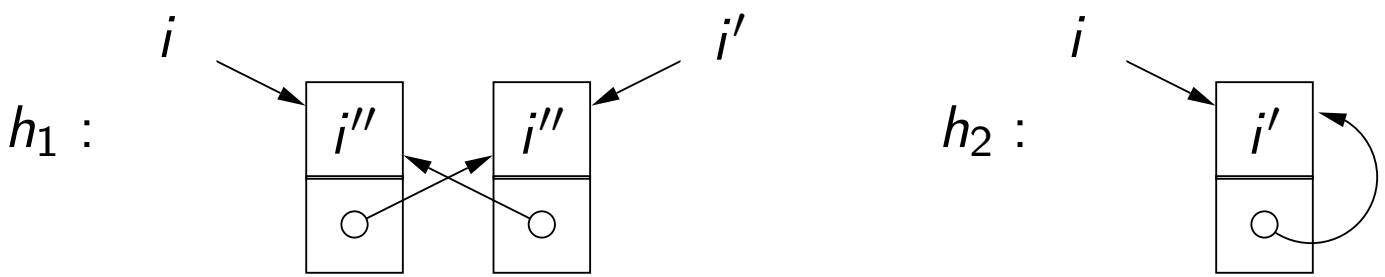
Separation logic (2)

Basic assertions:

- $x_1 \mapsto x_2, x_3$ – Heap consists of a single location containing two elements.
- $P_1 * P_2$ – Heap can be separated into two parts: one satisfies P_1, the other P_2.

Separation divides the heap.

- Assertions must hold in *disjoint subheaps*.
- Example: $\exists xyz.\,(x \mapsto z, y) * (y \mapsto z, x)$ is satisfied by h_1, but not h_2.



Recursion in separation logic

Define predicates using a recursive let: let Γ in P

- Γ predicate definitions, P let body.
- Separation logic without recursion is weak.
  - equivalent to first-order logic.

Formula satisfied by heaps containing circular lists.

$$\begin{aligned}&\text{let } ls(x_1, x_2) = (x_1 \mapsto x_2, x_2) \vee \\ &\qquad\qquad (\exists x_3.\, x_1 \mapsto x_3, x_3 * ls(x_3, x_2)) \\ &\text{in } \exists x.\, ls(x, x)\end{aligned}$$

Separation logic fragment

Correspondence defined over fragment $\mathcal{SL}$.

- Basic separation logic constructs: $x_1 \mapsto x_2, x_3$, $P * Q$.
- Recursion: let Γ in P.
- Some first-order constructs: $\vee$, $\exists$, **false**.

Omit universal quantification ($\forall$), conjunction ($\wedge$), negation ($\neg$) and **true**.

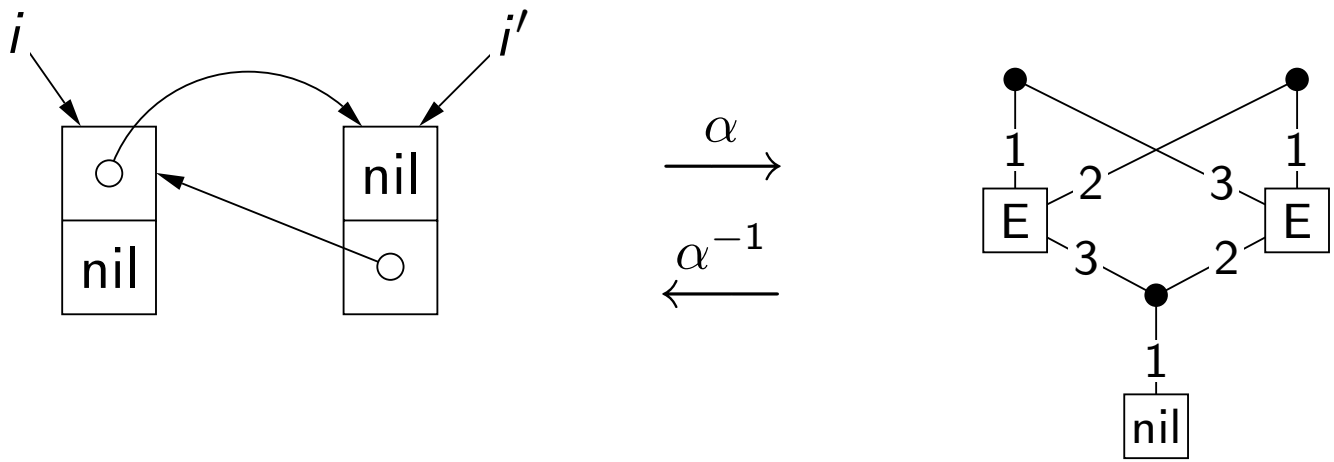
Heaps and heap-graphs

Define a class of *heap graphs*:

- Edge labels E of arity 3, and nil of arity 1.
- Each node is the first attachment point of at most one edge.

α is a bijective mapping from heaps to heap-graphs.

- Locations correspond to E-labelled edges.
- Unique nil mapped to a single nil-labelled edge.



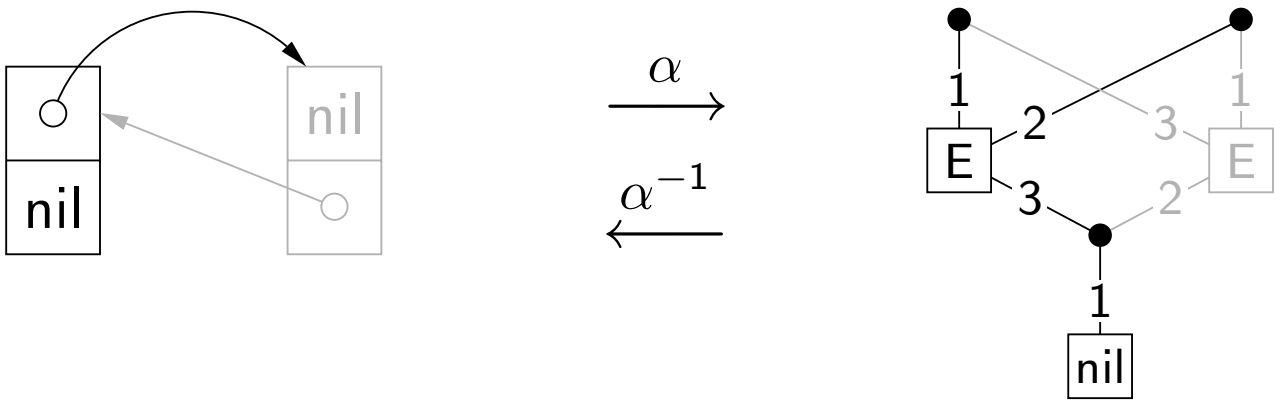
Heaps and heap-graphs

Define a class of *heap graphs*:

- Edge labels E of arity 3, and nil of arity 1.
- Each node is the first attachment point of at most one edge.

α is a bijective mapping from heaps to heap-graphs.

- Locations correspond to E-labelled edges.
- Unique nil mapped to a single nil-labelled edge.



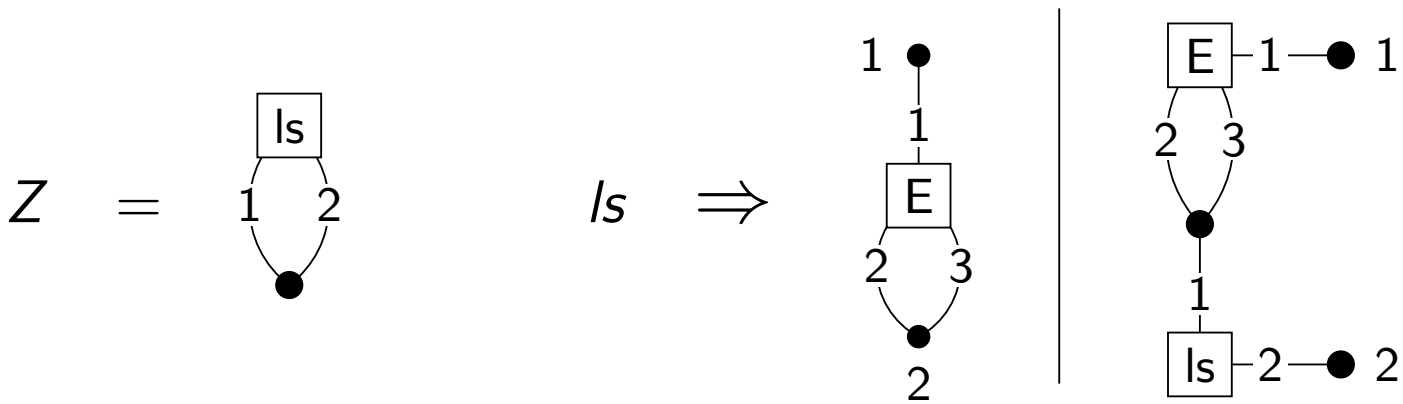
Heap-graph grammars

Hyperedge replacement grammar $H = \langle N, T, Z, P \rangle$.

- Productions P rewrite non-terminal edges.
- Graph in language of H if it can be derived from initial graph Z using productions in P.

Heap-graph grammars are hyperedge replacement grammars producing languages of heap-graphs.

Example: heap-graph grammar producing cyclic lists.



Correspondence

Intuition for the correspondence:

- Recursive definitions correspond to hyperedge replacement productions.
- Separating property of separating conjunction corresponds to the context-free property in hyperedge-replacement.

Mapping functions g and s are *semantics-preserving* with respect to α.

- $\alpha \circ g = g \circ \alpha$.
- $\alpha^{-1} \circ s = s \circ \alpha^{-1}$.

From hyperedge-replacement to separation logic

Grammar $H = \langle N, T, Z, P \rangle$ maps to formula $g[\![H]\!] = \text{let } \Gamma_P \text{ in } F_Z$.

Initial graph Z map to let-free formula F_Z.

- Graphs map to separating conjunctions.
- Terminal edges map to points-to assertions.
- Non-terminal edges map to instances of recursive predicates

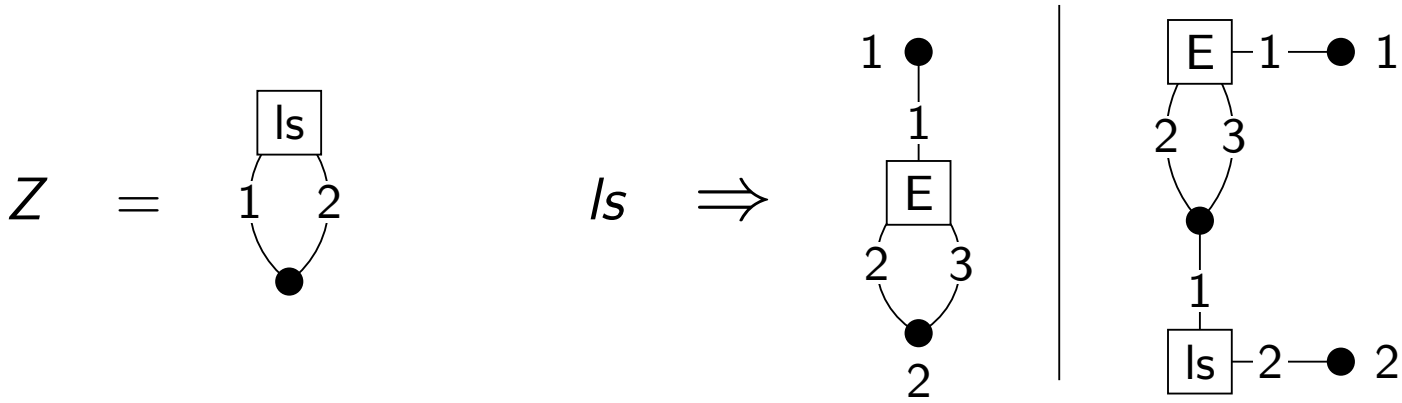
Productions in P map to predicate definitions Γ_P.

- Hyperedge tentacles map to predicate arguments $x_1, \ldots, x_n$.
- Right-hand sides are defined by g as with the initial graph.
- For production $K \Rightarrow G_1 \mid G_2$ formula is:

$$K(x_1, \ldots, x_n) = g[\![G_1]\!] \vee g[\![G_2]\!]$$

Example: Cyclic Linked Lists

Heap-graph grammar defining the set of cyclic singly-linked lists:

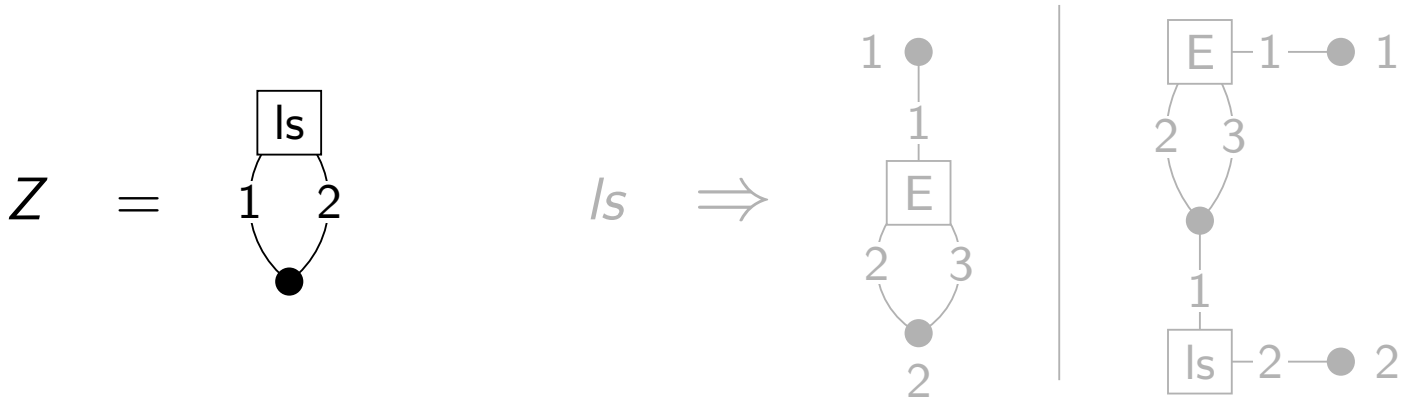


Corresponding separation logic formula:

let $ls(x_1, x_2) = (x_1 \mapsto x_2, x_2) \vee$
$(\exists x_3.\, x_1 \mapsto x_3, x_3 * ls(x_3, x_2))$
in $\exists x.\, ls(x, x)$

Example: Cyclic Linked Lists

Heap-graph grammar defining the set of cyclic singly-linked lists:

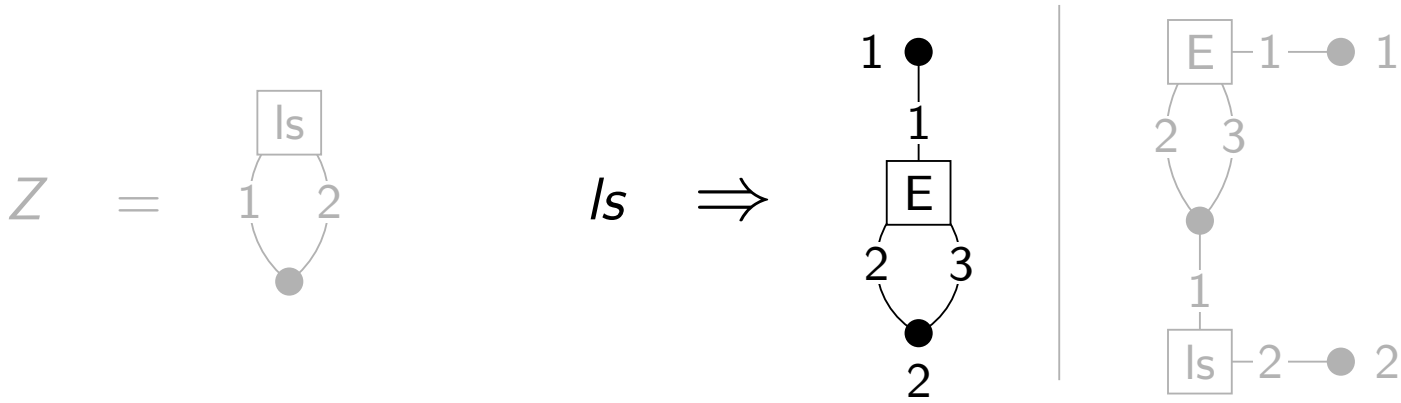


Corresponding separation logic formula:

$$\begin{aligned}&\text{let } ls(x_1, x_2) = (x_1 \mapsto x_2, x_2) \vee \\ &\qquad\qquad (\exists x_3.\, x_1 \mapsto x_3, x_3 * ls(x_3, x_2)) \\ &\text{in } \exists x.\, ls(x, x)\end{aligned}$$

Example: Cyclic Linked Lists

Heap-graph grammar defining the set of cyclic singly-linked lists:

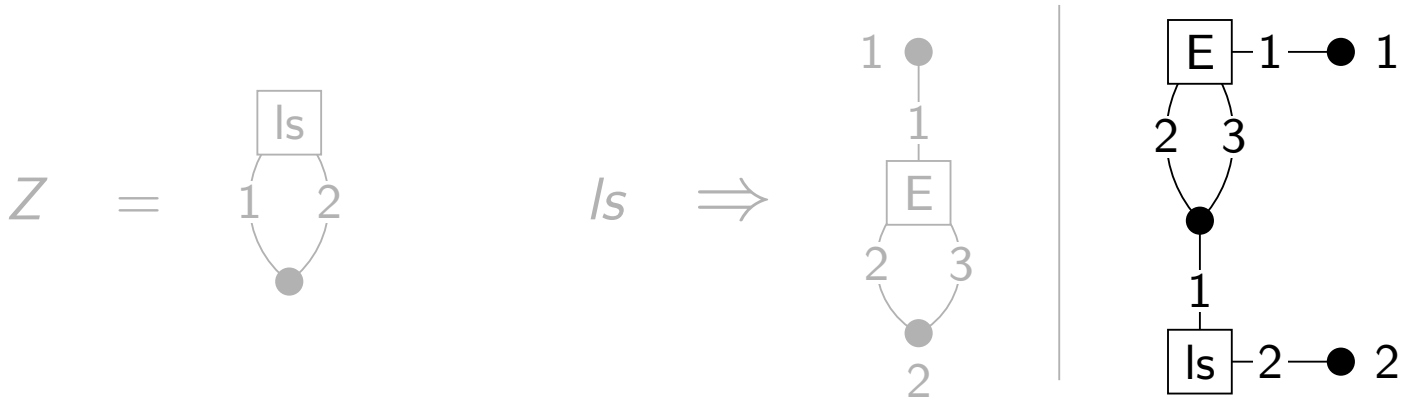


Corresponding separation logic formula:

let $ls(x_1, x_2) = (x_1 \mapsto x_2, x_2) \vee$
$(\exists x_3.\, x_1 \mapsto x_3, x_3 * ls(x_3, x_2))$
in $\exists x.\, ls(x, x)$

Example: Cyclic Linked Lists

Heap-graph grammar defining the set of cyclic singly-linked lists:



Corresponding separation logic formula:

$$\text{let } ls(x_1, x_2) = (x_1 \mapsto x_2, x_2) \vee (\exists x_3.\, x_1 \mapsto x_3, x_3 * ls(x_3, x_2))$$
$$\text{in } \exists x.\, ls(x, x)$$

From separation logic to hyperedge-replacement

Formulas are first *flattened*, removing any nested let-statement.

- Predicates are renamed to remove conflicts.
- Nested lets merged by promoting inner lets.

Formula $S = \text{let } \Gamma \text{ in } F$ maps to grammar $s[\![S]\!] = \langle N, T, Z_F, P_\Gamma \rangle$.

Initial graph Z_F constructed from F.

- Points-to assertions map to terminal E-labelled edges.
- Instances of a predicate K map to K-labelled non-terminal edges.

Productions P_Γ constructed from Γ.

- Definition $K(x_1, \ldots, x_n) = G$ results in production $K \Rightarrow s[\![G]\!]$.

Consequences of correspondence

Inexpressibility results.

- Pumping lemma, linear-growth theorem etc. can be applied to $\mathcal{SL}$.
- Cannot define balanced trees, grids, etc.
- Some operators are inexpressible (see next slide).

Fragment corresponds to *symbolic heaps* used for symbolic execution.

- Omit equality / inequality (but these may be simulatable).
- Otherwise, results for fragment hold for symbolic heaps.

Inexpressible constructs

Conjunction ($\wedge$)

- Corresponds to language intersection.
- Known to be HR-inexpressible (pumping lemma).

Negation ($\neg$)

- With disjunction, can simulate conjunction $\therefore$ inexpressible.

Elementary formula **true**

- Corresponds to a language with unbounded clique size.
- Known that all hyperedge replacement languages of simple graphs must have a bounded clique size.

Conclusion

A correspondence exists between heap-graph grammars and formulas in $\mathcal{SL}$.

- Two translation functions: g from grammars to formulas, and s from formulas to grammars.
- g and s are semantics-preserving with respect to α. That is, $\alpha \circ g = g \circ \alpha$, and $\alpha^{-1} \circ s = s \circ \alpha^{-1}$.
- Consequently, heap-graph grammars are of equivalent expressive power to $\mathcal{SL}$.
- Conjunction ($\wedge$), negation ($\neg$) and **true** cannot be modelled by hyperedge replacement.