Separation Logic
Introduction

Arthur Charguéraud

January 25th, 2016

Separation Logic

Separation Logic:
a set of rules for reasoning about programs that mutate the heap.

Separation Logic with interactive proofs:
a successful approach to the verification of imperative programs.

Where does it come from?

- John Reynolds (2000)
  - Intuitionistic Reasoning about Shared Mutable Data Structure
  - —building on ideas from Burstall (1972).
- John Reynolds, Peter O'Hearn, Hongseok Yang (2001)
  - Local reasoning about programs that alter data structures
- John Reynolds (2002)
  - Separation Logic: A logic for shared mutable data structure.

How successful?

Micro-controller	Klein et al	NICTA	Isabelle
Assembly language	Chlipala et al	MIT	Coq
Operating system	Shao et al	Yale	Coq
C (drivers)	Yang et al	Oxford	Other
C-light	Appel et al	Princeton	Coq
C11 (concurrent)	Vafeiadis et al	MPI and MSR	Paper
Java	Parkinson et al	MSR and Cambridge	Other
Java	Jacobs et al	Leuven	Verifast
Javascript	Gardner et al	Imperial College	Paper
ML	Morisset et al	Harvard	Coq
OCaml	Charguéraud	Inria	Coq
SML	Myreen et al	Cambridge	HOL

Why successful?

- Specifications that one can read: concise and intuitive.
- Proofs that one can understand: step-by-step analysis.
- Modularity that enable scaling up: $>$ 10k loc.

$$\text{Separation Logic} + \text{interactive proofs} = \text{no limits}$$

Purpose of the course

Goal: make *you* an expert in Separation Logic for sequential programs.

You will learn how Separation Logic handles:

- Tree-shaped structures
- Structures with sharing
- Polymorphic functions
- Higher-order functions
- Abstraction
- Modularity
- Interactive proofs

Choice of the logic

We will *define* Separation Logic in terms of a standard higher-order logic.

Our definitions will be given in Coq, because:

1. most formalizations of SL are in Coq;
2. my own work belong to this majority.

Choice of the source language

Separation Logic can be adapted to pretty much any language.

To best present all its aspects, we need a language with:

- a clean and concise syntax,
- null pointers,
- pointer arithmetic,
- types and polymorphism,
- higher-order functions.

We will use a virtual language: OCaml extended with null pointers and pointers arithmetic; we ignore the existence of headers used by the GC.