Modern Separation Logic — Derek Dreyer

Lecture 2 - HeapLang & Weakest Preconditions
June 29, 2026

1 HeapLang Introduction

The Iris framework does not have any fixed language, as it is a framework that helps others build more separation logic systems. To use Iris, one needs to pick a language (or make your own) and define the small-step operational semantics. Once both have been done, Iris can be instantiated to said language.

The rest of the lecture uses the HeapLang language, a simple language for reasoning about heaps. We introduce the syntax and the reduction rules for the language in the rest of this section.

1.1 Syntax

The syntax of HeapLang consists of terms e, values v, and evaluation contexts K, defined below.

$$\begin{aligned}
\text{Values } v ::= {} & n \mid \texttt{fix}\, f(x).e \mid \texttt{true} \mid \texttt{false} \mid () \mid \\
& (v_1, v_2) \mid \texttt{Inj}_1(v) \mid \texttt{Inj}_2(v) \mid \ell \ (\in \text{Loc}) \\
\text{Terms } e ::= {} & v \mid x \mid e_1\ e_2 \mid e_1\ \texttt{op}\ e_2 \mid \pi_1 e \mid \pi_2 e \mid \\
& \texttt{if}\, e\, \texttt{then}\, e_1\, \texttt{else}\, e_2 \mid \texttt{match} \cdots \mid \\
& \texttt{new}\, e \mid !\, e \mid e_1 \leftarrow e_2 \mid
\end{aligned}$$

Overall, the language is quite similar to the Imp language studied during Dexter's lectures, but with new values and terms for manipulating heaps. In particular: $\texttt{new}\ e$ creates a new memory location; $!\, e$ reads from the memory location after evaluating e, $l(e)$; and $e_1 \leftarrow e_2$ updates memory location evaluated at $l(e_1)$ with the value evaluated by e_2. On the value side, we add l, a set of memory locations.

To make things a bit more convenient, the small-step operational semantics of HeapLang is defined

Compiled By:
Ricky Cheng, Kalab Assefa, Ziyang Wang

using evaluation contexts, as follows:

$$\begin{aligned}\text{Eval. Ctx. } K ::= {} & \bullet \mid e\ K \mid K v \mid e\ \texttt{op}\ K \mid K\ \texttt{op}\ v \mid \\ & (e, K) \mid (K, v) \mid \texttt{if}\ K\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \mid \texttt{match}\ K \mid \\ & \texttt{new}\ K \mid\ !\,K \mid e_1 \leftarrow K \mid K \leftarrow v\end{aligned}$$

Note that the evaluation of HeapLang follows a right-to-left evaluation order.

1.2 Operational Semantics

As HeapLang reasons about objects on heaps, we define a heap as $h \in \mathit{Loc} \rightarrow^{\mathit{fin}} \mathit{Val}$, a finite mapping that maps memory locations to values. In general, the base reductions has the form of

$$\frac{(h; e) \rightsquigarrow_b (h'; e')}{(h; K[e]) \rightsquigarrow_b (h'; K[e'])}$$

We then have two different classes of base reductions, depending on if they affect the heap's state or not.

Pure Reductions are reductions that do not affect the heap at all and can be evaluated directly

$$\frac{e \rightsquigarrow_{\text{pure}} e'}{(h; e) \rightsquigarrow_b (h; e')}$$

where the reductions $e \rightsquigarrow_{\text{pure}} e'$ are defined as in Dexter's letures.

Stateful Reductions are reductions that do affect the heap. These are the new terms that we have added for heap manipulation. These rules thread the heap through allocation, dereferencing, and assignment, so we need to add specific conditions for the heap.

$$\frac{l \notin \mathsf{dom}(h)}{(h; \texttt{new}\ v) \rightsquigarrow_b (h[l \mapsto v], l)} \qquad \frac{h(l) = v}{(h; !\ l \rightsquigarrow_b (h; v)} \qquad \frac{l \in \mathsf{dom}(h)}{(h; l \leftarrow v) \rightsquigarrow_b (h[l \mapsto v]; ())}$$

2 Weakest Precondition Assertions

Instead of Hoare triples, Iris typically work with "weakest preconditions" connectives, which has the following form.

$$\texttt{wp}\ e\ \{v.Q(v)\}$$

where v is a binder for any return value that e may have.

The intuition of `wp` is that `wp` has enough ownership of memory to ensure e is safe to execute and, if e returns a value a v, then $Q(v)$ holds of the final heap.

We actually can encode a Hoare triple easily in Iris as well.

$$\{P\}\ e\ \{v.Q(v)\} \approx P \vdash \texttt{wp}\ e\ \{v.Q(v)\}$$

Here, P is any proposition that is at least as strong as `wp`. Also noted is that weakest precondition is similar to that of the weakest liberal precondition in Dexter's lectures.

Benefits of Weakest Precondition Having `wp` fits into the interactive style of Iris proofs.

- We can split between proof goal (a `wp`) and a "massaged" proof context
- We also can avoid Hoare logic rules that redundantly duplicate rules of the assertion logic to manipulate precondition

Furthermore, to support "backwards" reasoning as in Rocq, we adopt a style where the conclusions of `wp` rules have an arbitrary postcondition Q. Doing so allows us to instant the postcondition with whatever we want, making it much easier to apply tactics.

3 Structural Proof Rules for wp

We introduce the following four core structural proof rules for `wp`.

VALUE
$Q(v) \vdash \texttt{wp}\ v\ \{w, Q(w)\}$

PURESTEP
$e \rightsquigarrow_{\text{pure}} e' \star \texttt{wp}\ e'\ \{v.Q(v)\} \vdash \texttt{wp}\ e\ \{v.Q(v)\}$

WAND
$\forall w.Q(w) \mathrel{-\!\!*} Q'(w) \star \texttt{wp}\ e\ \{w.Q(w)\} \vdash \texttt{wp}\ e\ \{w, Q'(w)\}$

BIND
$\texttt{wp}\ e\ \{v.\texttt{wp}\ K[v]\ \{w.Q(w)\}\} \vdash \texttt{wp}\ K[e]\ \{w.Q(w)\}$

Similar to Rocq's goal-directed proof strategies, when we use these rules, we often apply it backwards, working our way up towards something trivial.

- We can apply VALUE rule whenever we have $\texttt{wp}\ v\ \{w.Q(w)\}$. Since we have already evaluated to some value v, we trivially get the goal $Q(v)$.
- WAND is conceptually very similar to to the consequence rule in normal SL and HL without the precondition strengthening. In Iris proof mode, applying this rule (backwards) will give you two subgoals to prove further.
- The BIND rule is essentially the SEQUETIALCOMPOSITION rule in HL and SL, but applied to evaluation contexts. Note that the postcondition on the left side contains nested assertions, something that is absent in Hoare logic. It's important to remember that `wp` is *not a fact, it's a stateful assertion.* Since e must step to some value v, we can make *assertions* within the postcondition about what must happen.
- The PURESTEP looks a bit backwards, but actually it is not. If we have a pure step (no heap change) from e to e', and we have $\texttt{wp}\ e'\ \{v.Q(v)\}$, then we can necessarily conclude that the state prior to the step e', ie $\texttt{wp}\ e\ \{v.Q(v)\}$, remains the same, since pure steps do not change the state. Applying this rule in Iris proof mode conveniently allows for stepping to the next expression.

3.1 Deriving Frame Rule

The frame rule is notably absent from the previous structural proof rules, because it's actually derivable from the given rules!

$$\frac{P \vdash \texttt{wp}\, e\, \{v.\ Q(v)\}}{P * R \vdash \texttt{wp}\, e\, \{v.\ Q(v) * R\}}$$

Derivation.

Context	**Tactic**	**Goal**
$P \vdash \texttt{wp}\, e\, \{v. Q(v)\}$	`intros` P, R	$P * R \vdash \texttt{wp}\, e\, \{v. Q(v) \star R\}$
$P \vdash \texttt{wp}\, e\, \{v. Q(v)\}, P,\ R$	`wand`	$\texttt{wp}\, e\, \{v. Q(v) \star R\}$
$P \vdash \texttt{wp}\, e\, \{v. Q(v)\}, P,\ R$		$\texttt{wp}\, e\, \{v. Q(v)\} \star (\forall v.\ Q(v) \mathrel{-\!\!*} Q(v) \star R)$
	`split`	
		Goal 1: $\texttt{wp}\, e\, \{v. Q(v)\}$
$P \vdash \texttt{wp}\, e\, \{v. Q(v)\}, P$	`done`	$\texttt{wp}\, e\, \{v. Q(v)\}$
		Goal 2: $\forall v.\ Q(v) \mathrel{-\!\!*} Q(v) \star R$
R	`wandIntro.`	$Q(v) \mathrel{-\!\!*} Q(v) \star R$
$R \star Q(v)$	`done`	$Q(v) \star R$

4 Heap Command Rules for wp

New
$\forall l. l \mapsto v \mathrel{-\!\!*} Q(l) \vdash \texttt{wp new } v\ \{w. Q(w)\}$

Store
$l \mapsto u \star (l \mapsto u \mathrel{-\!\!*} Q(())) \vdash \texttt{wp } l \leftarrow v\ \{w. Q(w)\}$

Load
$l \mapsto u \star (l \mapsto u \mathrel{-\!\!*} Q(u)) \vdash \texttt{wp } !l\ \{w. Q(w)\}$