Precise reasoning about resources in an affine separation logic

Aleš Bizjak
(Lars Birkedal, Daniel Gratzer, and Robbert Krebbers)

October 9, 2018

Aarhus University

A primer on separation logic

$$\{P\}\, e\, \{v.Q\} \qquad \underbrace{\Big\{\underbrace{P}_{\textit{assertion}}\Big\}\, e\, \Big\{\underbrace{v.Q}_{\textit{assertion}}\Big\}}_{\textit{specification/Hoare triple}}$$

- An extension of Hoare logic particularly well suited for verification of heap manipulating programs.
- Now extended to richer languages (concurrency, higher-order functions, general references, ...).
- Enables concise and modular specifications via the **separating conjunction** $*$.

Assertions

- Primitive points-to assertion:

$$\ell \hookrightarrow v$$

  describes the singleton heap fragment with location ℓ.

- The assertion $P * Q$ describes those heap fragments h which can be decomposed as $h = h_1 \cdot h_2$ with $h_1 \in P$ and $h_2 \in Q$.

- For example

$$\ell_1 \hookrightarrow 1 * \ell_2 \hookrightarrow 2$$

  describes heap fragments with **two distinct** locations ℓ_1 and ℓ_2.

Specifications

- **Specifications** describe properties of programs.
- The meaning of $\{P\}\, e\, \{v.Q\}$ is approximately
  - if e is run in a state satisfying P
  - **and** it terminates with value v
  - then the end state satisfies $Q(v)$.
- The key structural rule is the **frame rule**:

$$\frac{\{P\}\, e\, \{v.Q\}}{\{P * R\}\, e\, \{v.Q * R\}}$$

- This rule is the key to **small footprint** specifications.

Sample specifications

HOARE-ALLOC

$$\frac{}{\{\mathsf{Emp}\}\ \mathsf{ref}(v)\ \{\ell.\ell \hookrightarrow v\}}$$

HOARE-FREE

$$\frac{}{\{\ell \hookrightarrow v\}\ \mathsf{free}(\ell)\ \{\_.\mathsf{Emp}\}}$$

HOARE-LOAD

$$\frac{}{\{\ell \hookrightarrow v\}\ !\,\ell\ \{u.u = v \wedge \ell \hookrightarrow v\}}$$

HOARE-STORE

$$\frac{}{\{\ell \hookrightarrow v\}\ \ell \leftarrow u\ \{\_.\ell \hookrightarrow u\}}$$

HOARE-SEQ

$$\frac{\{P\}\ e_1\ \{\_.Q\} \qquad \{Q\}\ e_2\ \{v.R\}}{\{P\}\ e_1; e_2\ \{v.R\}}.$$

HOARE-CSQ

$$\frac{\vdash P_1 \Rightarrow P_2 \qquad \{P_2\}\ e\ \{v.Q_2\} \qquad \vdash \forall v, Q_2(v) \Rightarrow Q_1(v)}{\{P_1\}\ e\ \{v.Q_1\}}$$

The assertion logic

The assertion logic has all the usual logical connectives, $\forall, \exists, \wedge, \vee, \ldots$, and additionally:

- separating conjuction $*$
- the Emp assertion (unit for the separating conjunction).

These satisfy (in particular) the following axioms:

$$\frac{}{P * \mathrm{Emp} \dashv\vdash P} \qquad \frac{}{P * Q \dashv\vdash Q * P} \qquad \frac{}{P * (Q * R) \dashv\vdash (P * Q) * R}$$

$$\frac{}{P * (Q \vee R) \dashv\vdash P * Q \vee P * R} \qquad \cdots$$

The weakening axiom

A separation logic is **affine** if for all assertions P, Q we have

$$P * Q \vdash P.$$

One can think of "forgetting" the resource Q, e.g.,

$$\ell_1 \hookrightarrow 1 * \ell_2 \hookrightarrow 2 \vdash \ell_1 \hookrightarrow 1$$

can be seen as forgetting that the heap contains location ℓ_2.

On weakening

- Weakening allows us to "leak resources", i.e., we can prove specifications such as

$$\{\ell \hookrightarrow 1\}\ \mathsf{skip}\ \{\_.\mathrm{Emp}\}$$

by using

$$\ell \hookrightarrow 1 \vdash \mathrm{Emp} * \ell \hookrightarrow 1 \vdash \mathrm{Emp}.$$

- As a consequence the triple

$$\{\mathrm{Emp}\}\, e\, \{\_.\mathrm{Emp}\}$$

cannot guarantee that the program does not leak memory, e.g.,

$$\{\mathrm{Emp}\}\ \mathrm{ref}(3); \mathsf{skip}\ \{\_.\mathrm{Emp}\}$$

- It is sometimes useful to have weakening when we do not want to reason about resources precisely.

Originally two variants of separation logic.

- **Affine** ("intuitionistic") for reasoning about garbage collected languages.
- **Linear** ("classical") for reasoning about languages with explicit memory reclamation.
- In a linear separation logic

$$\{\mathsf{Emp}\}\, e\, \{\_.\mathsf{Emp}\}$$

  typically guarantees that all allocated memory is freed before the program terminates.

- Recent concurrent separation logics for reasoning about fork-style concurrency **are all affine**.
- The main reason is it is unclear how to have general enough **sharing mechanisms** in a linear logic.
- Existing sharing mechanisms can leak resources.

- We show how to ensure memory reclamation in an affine separation logic.
- The solution extends to ensuring that a program which delegates memory reclamation to a background thread does not leak memory.

Two models, and a third

The model $\mathcal{M}_1$: assertion logic

- Assertions are modelled as arbitrary subsets of heap fragments,

$$\llbracket P \rrbracket \in \mathcal{P}(\mathcal{H})$$

- The points-to assertion denotes the singleton set.

$$\llbracket \ell \hookrightarrow v \rrbracket = \{h \mid \mathrm{dom}(h) = \{\ell\} \text{ and } h(\ell) = v\}$$

- The Emp assertion contains only the empty heap fragment.

$$\llbracket \mathsf{Emp} \rrbracket = \{h \mid \mathrm{dom}(h) = \emptyset\}.$$

- Separating conjunction combines the heap fragments.

$$\llbracket P * Q \rrbracket = \{h \mid \exists h_1 \in \llbracket P \rrbracket, h_2 \in \llbracket Q \rrbracket, h = h_1 \cdot h_2\}$$

- This model does not validate weakening since

$$\llbracket \ell \hookrightarrow 1 \rrbracket \not\subseteq \llbracket \mathsf{Emp} \rrbracket.$$

The model $\mathcal{M}_1$: specification logic

The meaning of the specification $\{P\}\, e\, \{v.Q\}$ is approximately

- for any **heap fragment** $h \in [\![P]\!]$ and any **disjoint** heap fragment h'
- running e in the heap $h \cdot h'$ is **safe** *and*
- if the program terminates with value v and heap h_1 then $h_1 = h_1' \cdot h'$ for **some** $h_1' \in [\![Q(v)]\!]$.

Thus the meaning of the specification

$$\{\mathsf{Emp}\}\, e\, \{\_.\mathsf{Emp}\}$$

is that if the program starts in heap h' then

- it does not fault (is safe) and
- **if it terminates the end heap is h'.**

Properties of $\mathcal{M}_1$

This variant of the logic is very good for reasoning about a language with explicit memory management.

But it is sometimes too precise

- for a garbage collected language
- or when we don't care about specifying memory management, but only care about functional correctness
- or when there are other resources (e.g., ghost state).

In a garbage collected language we would like

$$\{\ell \hookrightarrow v\}\ \mathsf{skip}\ \{\_.\mathsf{Emp}\}$$

so we can "forget" ownership of locations.

The model $\mathcal{M}_2$: assertion logic

- Assertions are modelled as **upwards closed subsets of heap fragments**,

$$\llbracket P \rrbracket \in \mathcal{P}^{\uparrow}(\mathcal{H})$$

- Upwards closure is with respect to **extension order**.

$$h_1 \leq h_2 \iff \exists h_f, h_1 \cdot h_f = h_2.$$

- The points-to assertion now denotes the set of all heaps containing that particular location.

$$\llbracket \ell \hookrightarrow v \rrbracket = \{h \mid \mathrm{dom}(h) \supseteq \{\ell\} \text{ and } h(\ell) = v\}$$

- The Emp assertion contains not only the empty heap fragment, but all heap fragments.

$$\llbracket \mathrm{Emp} \rrbracket = \mathcal{H} = \llbracket \mathrm{True} \rrbracket$$

- Separating conjunction as before.

The model $\mathcal{M}_2$: weakening

- This model validates weakening since if $h \in [\![P * Q]\!]$ then there exist
  - $h_1 \in [\![P]\!]$ and
  - $h_2 \in [\![Q]\!]$ such that
  - $h = h_1 \cdot h_2$.

  Hence $h_1 \leq h$ and thus $h \in [\![P]\!]$.
- There is no assertion satisfied only by the empty heap fragment ε.
- Indeed, any heap fragment h' satisfies $\varepsilon \leq h'$.
- Thus if $\varepsilon \in [\![P]\!]$ then $[\![P]\!] = \mathcal{H}$.

The model $\mathcal{M}_2$: specification logic

The meaning of the specification $\{P\}\, e\, \{v.Q\}$ is as before.

- for any **heap fragment** $h \in [\![P]\!]$ and any **disjoint** heap fragment h'
- running e in the heap $h \cdot h'$ is **safe** *and*
- if the program terminates with value v and heap h_1 then $h_1 = h_1' \cdot h'$ for **some** $h_1' \in [\![Q(v)]\!]$.

However now the specification

$$\{\mathrm{Emp}\}\, e\, \{\_.\mathrm{Emp}\}$$

means simply that if e starts in some heap $h \cdot h_f$

- it does not fault (is safe) and
- **if it terminates the end heap is $h' \cdot h_f$ for some h'**

The model $\mathcal{M}_3$: a more precise affine model

- For the simple sequential language there is no need for the extension we are about to describe.
- However it is easiest to understand the extension in this simplified setting.
- The model (and the corresponding logic) we construct is affine.
- And it can still be used to guarantee that memory is correctly managed.
- It achieves this with only a modicum of additional bookkeeping.

The logic of $\mathcal{M}_3$

- The new assertions $\ell \hookrightarrow_\pi v$ and $\mathfrak{e}_\pi$ satisfy:

EMP-SPLIT

$$\mathfrak{e}_{\pi_1} * \mathfrak{e}_{\pi_2} \dashv\vdash \mathfrak{e}_{\pi_1+\pi_2}$$

PT-SPLIT

$$\ell \hookrightarrow_{\pi_1} v * \mathfrak{e}_{\pi_2} \dashv\vdash \ell \hookrightarrow_{\pi_1+\pi_2} v$$

PT-DISJ

$$\ell_1 \hookrightarrow_{\pi_1} v_1 * \ell_2 \hookrightarrow_{\pi_2} v_2 \vdash \ell_1 \neq \ell_2$$

- $\mathfrak{e}_\pi$ can be thought of as “permission” to allocate.
- And the specifications of the basic operations are

HOARE-ALLOC

$$\{\mathfrak{e}_\pi\}\ \mathsf{ref}(v)\ \{\ell.\, \ell \hookrightarrow_\pi v\}$$

HOARE-FREE

$$\{\ell \hookrightarrow_\pi v\}\ \mathsf{free}(\ell)\ \{\mathfrak{e}_\pi\}$$

HOARE-LOAD

$$\{\ell \hookrightarrow_\pi v\}\ !\,\ell\ \{w.\, w = v \land \ell \hookrightarrow_\pi v\}$$

HOARE-STORE

$$\{\ell \hookrightarrow_\pi v\}\ \ell \leftarrow u\ \{\ell \hookrightarrow_\pi u\}$$

- The splitting properties EMP-SPLIT and PT-SPLIT allow us to, e.g., derive

$$\{\mathfrak{e}_\pi\}\ (\mathsf{ref}(v_1), \mathsf{ref}(v_2))\ \left\{(\ell_1, \ell_2).\ell_1 \hookrightarrow_{\frac{\pi}{2}} v_1 * \ell_2 \hookrightarrow_{\frac{\pi}{2}} v_2\right\}$$

- **The fraction is used to keep track of weakening.**
- In this logic the following triple is derivable for any π.

$$\{\ell \hookrightarrow_\pi v\}\ \mathsf{skip}\ \{\_.\mathsf{Emp}\}$$

- However if we forget a points-to like this then **this will be visible in the specification**.

- The following specification is not possible

$$\{\mathfrak{e}_\pi\}\ \mathsf{ref}(3); \mathsf{skip}\ \{\_.\mathfrak{e}_\pi\}.$$

- We can only show

$$\{\mathfrak{e}_\pi\}\ \mathsf{ref}(3); \mathsf{skip}\ \{\_.\mathfrak{e}_{\pi'}\}.$$

for some (in fact all) $\pi' < \pi$.

- However if we free the allocated location then we get back the full $\mathfrak{e}_\pi$.

$$\{\mathfrak{e}_\pi\}\ \mathsf{let}\, \ell = \mathsf{ref}\,(3)\ \mathsf{in}\ \mathsf{free}(\ell)\, \{\_.\mathfrak{e}_\pi\}$$

Is derivable.

Let $\mathfrak{M}$ be the partial commutative monoid with carrier

$$\{\varepsilon\} + (0, 1] \times \mathcal{H}.$$

ε is defined to be the unit, and otherwise the operation is

$$(\pi_1, h_1) \cdot (\pi_2, h_2) = (\pi_1 + \pi_2, h_1 \cdot h_2)$$

if $\pi_1 + \pi_2 \leq 1$ and $h_1 \cdot h_2$ are both defined.

- Assertions are modelled as **upwards closed subsets of $\mathfrak{M}$**,

$$[\![P]\!] \in \mathcal{P}^{\uparrow}(\mathfrak{M})$$

- Upwards closure is again with respect to **extension order**.

$$m_1 \leq m_2 \iff \exists m_f, m_1 \cdot m_f = m_2.$$

The model $\mathcal{M}_3$

- The points-to assertion is modelled as the up-closure of the set

$$[\![\ell \hookrightarrow_\pi v]\!] = \uparrow\{(\pi, h) \mid \mathrm{dom}(h) = \{\ell\} \text{ and } h(\ell) = v\}.$$

- Concretely

$$[\![\ell \hookrightarrow_1 v]\!] = \{(1, h) \mid \mathrm{dom}(h) = \{\ell\} \text{ and } h(\ell) = v\}$$

  is the singleton set.
- For $\pi < 1$ the set $[\![\ell \hookrightarrow_\pi v]\!]$ also contains pairs (π', h) with $\pi' > \pi$, $\mathrm{dom}(h) \supseteq \{\ell\}$ and $h(\ell) = v$.

The model $\mathcal{M}_3$

- The $\mathfrak{e}_\pi$ assertion is modelled as.

$$[\![\mathfrak{e}_\pi]\!] = \uparrow\{(\pi, \varepsilon)\}$$

- Concretely

$$[\![\mathfrak{e}_1]\!] = \{(1, \varepsilon)\}$$

- For $\pi < 1$ the assertion $[\![\mathfrak{e}_\pi]\!]$ also contains all pairs (π', h) with $\pi' > \pi$.

The model $\mathcal{M}_3$

The Emp and separating conjunction are modelled as in $\mathcal{M}_2$.

- The Emp assertion contains all elements of the monoid (this enables weakening).

$$[\![\mathrm{Emp}]\!] = \mathfrak{M} = [\![\mathrm{True}]\!]$$

- Separating conjunction combines the elements as before.

$$[\![P * Q]\!] = \{m \mid \exists m_1 \in [\![P]\!], m_2 \in [\![Q]\!], m = m_1 \cdot m_2\}$$

The specification logic of $\mathcal{M}_3$

The meaning of the specification $\{P\}\, e\, \{v.Q\}$ needs to take into account the fractions. It is approximately:

- for any $(\pi, h) \in [\![P]\!]$ and any **disjoint** element $m \in \mathfrak{M}$. Suppose $(\pi, h) \cdot m = (\pi', h')$.
- running e in the heap h' is **safe** *and*
- if the program terminates with value v and heap h_1 then there exists $m' \in [\![Q(v)]\!]$ and a fraction π_1 such that $m' \cdot m = (\pi_1, h_1)$.

In particular from the specification

$$\{\mathfrak{e}_\pi\}\, e\, \{\_.\mathfrak{e}_\pi\}$$

we can derive that if we run e in heap h then

- e does not fault and
- if it terminates the **end heap is** h.

The crucial ingredient in this proof is the fact that the fraction π in the pre- and post-conditions is the same.

Recap

- In a linear logic we can reason about resources very precisely.
- Admitting weakening we lose this ability. We can no longer guarantee absence of resources.
- However adding a small amount of annotations to keep track of weakening we regain the ability to reason about resources precisely.

Concurrent separation logic

- We are interested in reasoning in a language with concurrency.
- In particular fork $\{\}$ concurrency.
- Now a program is a set of threads running in parallel.
- Threads communicate through shared memory.
- The fork $\{e\}$ construct creates a new thread which executes the program e.

$$\begin{array}{l}\mathsf{let}\ \mathsf{flag} = \mathsf{ref}\,(\mathsf{false})\ \mathsf{in}\\ \mathsf{fork}\ \{\mathsf{flag} \leftarrow \mathsf{true}\}\,;\\ \mathsf{if}\ !\,\mathsf{flag}\ \mathsf{then}\ 0\ \mathsf{else}\ 1\end{array}$$

The result of this program can be either 0 or 1, depending on the scheduler.

Thread local reasoning

- In the logic we want to reason locally.
- In the triple
$$\{P\}\ e\ \{v.Q\}$$
e is a single term.
- This enables modular specifications (we don't need to know how many or what other threads are running).
- Separating conjunction is used to split resources between threads, e.g.,
$$\frac{\{P_1\}\ e_1\ \{\_.\mathrm{Emp}\} \qquad \{P_2\}\ e_2\ \{v.Q_2\}}{\{P_1 * P_2\}\ \mathsf{fork}\ \{e_1\}\,;\, e_2\ \{v.Q_2\}}$$

Sharing

- However simply splitting resources is not enough.
- Sometimes it is necessary to share resources, e.g., in

$$\begin{array}{l} \mathsf{let}\ \mathsf{flag} = \mathsf{ref}(\mathsf{false})\ \mathsf{in} \\ \mathsf{fork}\ \{\mathsf{flag} \leftarrow \mathsf{true}\}\,; \\ \mathsf{if}\ !\,\mathsf{flag}\ \mathsf{then}\ 0\ \mathsf{else}\ 1 \end{array}$$

  the location flag is shared.
- This is achieved with **invariants**.
- These are special assertions $\boxed{P}$ which can be duplicated.
- Their downside is that they can only be used in a restricted way.

Invariants

- Invariants are duplicable

$$\boxed{R} * \boxed{R} \dashv\vdash \boxed{R}$$

- Any assertion can be made into an invariant.

$$\frac{\boxed{R} \vdash \{P\}\, e\, \{w.\, Q\}}{\{P * R\}\, e\, \{w.\, Q\}}$$

- The assertion in the invariant can be accessed in a restricted way.

$$\frac{\mathrm{atomic}(e) \qquad \{P * R\}\, e\, \{w.\, Q * R\}}{\boxed{R} \vdash \{P\}\, e\, \{w.\, Q\}}$$

Example

To specify the program

$$\begin{aligned}&\mathsf{let}\ \mathsf{flag} = \mathsf{ref}\,(\mathsf{false})\ \mathsf{in}\\ &\mathsf{fork}\ \{\mathsf{flag} \leftarrow \mathsf{true}\}\,;\\ &\mathsf{if}\ !\,\mathsf{flag}\ \mathsf{then}\,0\,\mathsf{else}\,1\end{aligned}$$

We would use the invariant

$$\boxed{\mathsf{flag} \hookrightarrow \mathsf{true} \vee \mathsf{flag} \hookrightarrow \mathsf{false}}\,.$$

to give the specification

$$\{\mathsf{Emp}\}\,e\,\{v.v = 0 \vee v = 1\}.$$

Note that the invariant is needed because $\mathsf{flag} \hookrightarrow b$ cannot be split so that both threads can use the location.

The following derivation is valid.

$$\frac{\overline{\boxed{\ell \hookrightarrow v} \vdash \{\mathsf{Emp}\}\ \mathsf{skip}\ \{\_.\mathsf{Emp}\}}}{\{\ell \hookrightarrow v\}\ \mathsf{skip}\ \{\_.\mathsf{Emp}\}}$$

Hence the following triple is derivable

$$\{\ell \hookrightarrow v\}\ \mathsf{skip}\ \{\_.\mathsf{Emp}\}.$$

Observation

None of the existing logics for reasoning about languages with fork can guarantee correct memory management.

Iron

Iron

- Iron is a an extension of the Iris program logic that ensures precise resource management.
- Iris is a state of the art program logic for reasoning about imperative, concurrent, higher-order programs.
- It is an affine logic.
- It supports very general invariants.
- Used for verification of sophisticated fine-grained concurrent algorithms.
- And as a meta-language for studies of type systems, etc.
- But unclear how to manage resources (e.g., memory) precisely.

Key points

In Iron we can reason about resources precisely using the same idea as in $\mathcal{M}_3$.

At the same time **we retain all the reasoning facilities of Iris**, including impredicative and higher-order invariants.

- The key idea is that if we transfer, e.g., a points-to to an invariant we lose a degree of knowledge of it (i.e., lose a fraction of π).
- **Thus if we wish to guarantee there are no memory leaks some thread must be in charge of disposing allocated memory.**

Meaning of specifications in Iron

In Iron the meaning of the specification

$$\{\mathfrak{e}_\pi\}\, e\, \{\_.\mathfrak{e}_\pi\}$$

is approximately: if e starts in heap h then

- it is safe and
- if **all the threads** (that e spawns) **have terminated**, then the resulting heap is h.

Example

The following example program can be given the specification

$$\{\mathfrak{e}_\pi\} \ldots \{\_.\mathfrak{e}_\pi\}$$

$$
\begin{array}{l}
\textsf{let channel} = \textsf{ref}(\textsf{None})\ \textsf{in} \\
\textsf{let rec receive}() = \begin{array}[t]{l}\textsf{match !channel with} \\ \quad \textsf{None} \qquad\quad \Rightarrow \textsf{receive}() \\ \mid \textsf{Some data} \Rightarrow \textsf{free}(\textsf{data}); \textsf{free}(\textsf{channel}) \\ \textsf{end in}\end{array} \\
\textsf{fork}\ \{\textsf{receive}()\}\,; \\
\textsf{let data} = \textsf{ref}(57)\ \textsf{in} \\
\textsf{channel} \leftarrow \textsf{Some data}
\end{array}
$$

A more abstract view

- The Iron logic is expressive.
- And can be used to verify many intricate examples, guaranteeing correct resource management.
- However the fraction accounting can be tedious.
- However for a lot of examples it can be abstracted away in a uniform way.
- The key idea is that if $\mathcal{B}$ is a model of the assertion logic then the set of all functions $[0,1] \to \mathcal{B}$ is also a model of the assertion logic.

- Standard propositional connectives lift pointwise, e.g.,

$$(P \mathrel{\widehat{\Rightarrow}} Q)(\pi) = P(\pi) \Rightarrow Q(\pi).$$

- Separating conjunction also splits the fractions.

$$(P \mathrel{\widehat{*}} Q)(\pi) = \bigvee_{\pi_1 + \pi_2 = \pi} P(\pi_1) * Q(\pi_2).$$

- The points-to connective can be lifted (almost) pointwise.

$$(\ell \mathrel{\widehat{\hookrightarrow}} v)(\pi) = \begin{cases} \text{False} & \text{if } \pi = 0 \\ \ell \hookrightarrow_\pi v & \text{otherwise} \end{cases}$$

- The Emp assertion guarantees that the fraction is 0.

$$\widehat{\text{Emp}}(\pi) = \begin{cases} \text{Emp} & \text{if } \pi = 0 \\ \text{False} & \text{otherwise} \end{cases}$$

The Hoare triples can also be lifted “pointwise”.

$$\{P\}\, e\, \{v.Q\} = \bigwedge_{\pi} \{P(\pi)\}\, e\, \{v.Q(\pi)\}.$$

Theorem

Applying this construction to $\mathcal{M}_3$ we recover all the rules and guarantees of $\mathcal{M}_1$.

Moreover there is a large class of invariants which can be used in the lifted logic.

Thus most of the examples in Iron can be done in the lifted logic, without any fraction accounting.

Formalization in Coq

```
Theorem channel_example :
  {{{ emp }}} prog #() {{{ v, RET v; emp }}}.
Proof.
  iIntros (Φ) "_ HΦ". wp_lam. wp_alloc l as "Hl"; wp_let.
  iMod (fcinv_alloc_strong _ N) as (γ) "[Hγ Halloc]".
  iEval (rewrite -Qp_three_quarter_quarter) in "Hγ"; iDestruct "Hγ" as "[Hγ Hγ']".
  iMod ("Halloc" $! (transfer_inv2 γ l) with "[Hl]") as "[Hγc #?]".
  { iExists NONEV; eauto with iFrame. }
  wp_apply (iron_wp_fork with "[Hγ HΦ]").
  - iIntros "!>". do 2 wp_lam. wp_alloc k as "Hk". wp_let.
    iMod (fcinv_open_strong _ N with "[$] [$]") as "(Hinv & Hγ & Hclose)"; first done.
    iDestruct "Hinv" as (v) ">[Hl [->|Hinv]]".
    + wp_store. iApply "HΦ". iApply "Hclose".
      iLeft. iExists (SOMEV (#k)). auto 10 with iFrame.
    + iDestruct "Hinv" as (k') "(_&?&Hγ')".
      by iDestruct (fcinv_own_valid with "Hγ Hγ'") as %[].
  - iLöb as "IH"; wp_rec. wp_bind (! _)%E.
    iMod (fcinv_open_strong _ N with "[$] [$]") as "(Hinv & Hγ & Hclose)"; first done.
    iDestruct "Hinv" as (v) ">[Hl [->|Hinv]]".
    + wp_load. iMod ("Hclose" with "[Hl]").
      { iLeft. iExists NONEV. eauto 10 with iFrame. }
      iModIntro. wp_match. iApply ("IH" with "[$] [$]").
    + wp_load. iDestruct "Hinv" as (k') "(->&Hk'&Hγ1) /=".
      iMod ("Hclose" with "[Hγc Hγ1 Hγ]") as "_".
      { iRight. iEval (rewrite -{1}Qp_three_quarter_quarter). iFrame. }
      iModIntro. wp_match. wp_apply (iron_wp_free with "[$]"); iIntros "_".
      wp_seq. wp_apply (iron_wp_free with "[$]"); auto.
Qed.
```

Conclusion

- We showed how to reason about resources precisely in presence of the weakening rule.
- The Iron logic is the first which is able to guarantee absence of memory leaks in presence of fork $\{\}$.
- The idea of annotating assertions with fractions can also be applied to other resources.
- For instance it can be used to guarantee correct lock management, i.e., that they are released.
- For details see

  Iron: Managing Obligations in Higher-Order Concurrent Separation Logic

  https://iris-project.org/pdfs/2018-iron.pdf