Purity of an ST Monad

Full Abstraction by Semantically Type Back-Translation

Koen Jacobs (me)
Dominique Devriese
Amin Timany
Lazy Functional State Threads

John Launchbury and Simon L. Peyton Jones
University of Glasgow
Email: {simonpj,jl}@dcs.glasgow.ac.uk Phone: +44-41-330-4500
March 10, 1994

Abstract

Some algorithms make critical internal use of updatable state, even though their external specification is purely functional. Based on earlier work on monads, we present a way of securely encapsulating stateful computations that manipulate multiple, named, mutable objects, in the context of a non-strict, purely-functional language.

The security of the encapsulation is assured by the type system, using parametricity. Intriguingly, this parametricity requires the provision of a (single) constant with a rank-2 polymorphic type.

A shorter version of this paper appears in the Proceedings of the ACM Conference on Programming Languages Design and Implementation (PLDI), Orlando, June 1994.

1 Introduction

Purely functional programming languages allow many algorithms to be expressed very concisely, but there are a few algorithms in which in-place updatable state seems to play a crucial role. For these algorithms, purely-functional languages, which lack updatable state, appear to be inherently inefficient (Ponder, McGeer & Ng [1988]).

Take, for example, algorithms based on the use of incrementally-modified hash tables, where lookups are interleaved with the insertion of new items. Similarly, the union/find algorithm relies for its efficiency on the set representations being simplified each time the structure is examined. Likewise, many graph algorithms require a dynamically changing structure in which sharing is explicit, so that changes are visible non-locally.

There is, furthermore, one absolutely unavoidable use of state in every functional program: input/output. The plain fact of the matter is that the whole purpose of running a program, functional or otherwise, is to make some side effect on the world — an upstate-in-place, if you please. To many programs these I/O effects are rather complex, involving interleaved reads from and writes to the world state.

We use the term "stateful" to describe computations or algorithms in which the programmer really does want to manipulate (updatable) state. What has been lacking until now is a clean way of describing such algorithms in a functional language — especially a non-strict one — without throwing away the main virtues of functional languages: independence of order of evaluation (the Church-Rosser property), referential transparency, non-strict semantics, and so on.

In this paper we describe a way to express stateful algorithms in non-strict, purely-functional languages. The approach is a development of our earlier work on monadic I/O and state encapsulation (Launchbury [1992]; Peyton Jones & Wadler [1993]), but with an important technical innovation: we use parametric polymorphism to achieve safe encapsulation of state. It turns out that this allows mutable objects to be named without losing safety, and also allows input/output to be smoothly integrated with other state manipulation.

The other important feature of this paper is that it describes a complete system, and one that is implemented in the Glasgow Haskell compiler and freely available. The system has the following properties:

- Complete referential transparency is maintained. At first it is not clear what this statement means: how can a stateful computation be said to be referentially transparent? To be more precise, a stateful computation is a state transformer, that is, a function from an initial state to a final state. It is like a "script", detailing the actions to be performed on its input state. Like any other function, it is quite possible to apply a single stateful computation to more than one input state.

So, a state transformer is a pure function. But, because we guarantee that the state is used in a single-threaded way, the final state can be constructed by modifying the input state in-place. This efficient implementation respects the purely-functional semantics.
-- NAIVE Interface of the haskell ST-Monad

ST :: * -> *
Ref :: * -> *

newRef :: a -> ST (Ref a)
readRef :: Ref a -> ST a
writeRef :: Ref a -> a -> ST ()

return :: a -> ST a
(>>=) :: ST a -> (a -> ST b) -> ST b
runST :: ST a -> a

-- PROBLEMS AHEAD

location :: Ref Int
location :: runST (newRef 0)

produceInteger :: () -> Int
produceInteger = runST (do
  n <- readRef location
  writeRef location (n + 1)
  return n)

definitelyTrue :: Bool
definitelyTrue = produceInteger () == produceInteger ()
-- NAIVE Interface of the haskell ST-Monad

ST :: * -> *
Ref :: * -> *

newRef :: a -> ST (Ref a)
readRef :: Ref a -> ST a
writeRef :: Ref a -> a -> ST ()

return :: a -> ST a
(>>=) :: ST a -> (a -> ST b) -> ST b

runST :: ST a -> a

-- PROBLEMS AHEAD

location :: Ref Int
location :: runST (newRef 0)

produceInteger :: () -> Int
produceInteger = runST (do
  n <- readRef location
  writeRef location (n + 1)
  return n)

definitelyTrue :: Bool
definitelyTrue = produceInteger () == produceInteger ()
1 -- NAIVE Interface of the haskell ST-Monad
2
3 ST :: * -> *
4 Ref :: * -> *
5
6 newRef :: a -> ST (Ref a)
7 readRef :: Ref a -> ST a
8 writeRef :: Ref a -> a -> ST ()
9
10 return :: a -> ST a
11 (>>=) :: ST a -> (a -> ST b) -> ST b
12
13 runST :: ST a -> a
14
15 -- PROBLEMS AHEAD
16
17 location :: Ref Int
18 location :: runST (newRef 0)
19
20 produceInteger :: () -> Int
21 produceInteger = runST (do
22      n <- readRef location
23      writeRef location (n + 1)
24      return n)
25
26 definitelyTrue :: Bool
27 definitelyTrue = produceInteger () == produceInteger ()
-- NAIVE Interface of the haskell ST-Monad

ST :: * -> *
Ref :: * -> *

newRef :: a -> ST (Ref a)
readRef :: Ref a -> ST a
writeRef :: Ref a -> a -> ST ()

return :: a -> ST a
(>>=) :: ST a -> (a -> ST b) -> ST b

runST :: ST a -> a

-- PROBLEMS AHEAD

location :: Ref Int
location :: runST (newRef 0)

produceInteger :: () -> Int
produceInteger = runST (do
  n <- readRef location
  writeRef location (n + 1)
  return n)

definitelyTrue :: Bool
definitelyTrue = produceInteger () == produceInteger ()
1 -- NAIVE Interface of the haskell ST-Monad
2
3 ST :: * -> *
4 Ref :: * -> *
5
6 newRef :: a -> ST (Ref a)
7 readRef :: Ref a -> ST a
8 writeRef :: Ref a -> a -> ST ()
9
10 return :: a -> ST a
11 (>>=) :: ST a -> (a -> ST b) -> ST b
12
13 runST :: ST a -> a
14
15 -- PROBLEMS AHEAD
16
17 location :: Ref Int
18 location :: runST (newRef 0)
19
20 produceInteger :: () -> Int
21 produceInteger = runST (do
22
23
24
25
26 definitelyTrue :: Bool
27 definitelyTrue = produceInteger () == produceInteger ()
-- NAIVE Interface of the haskell ST-Monad

1.3
3.4

newRef :: a -> ST (Ref a)
readRef :: Ref a -> ST a
writeRef :: Ref a -> a -> ST ()

return :: a -> ST a
(>>=) :: ST a -> (a -> ST b) -> ST b

runST :: ST a -> a

location :: Ref Int

location :: runST (newRef 0)

produceInteger :: () -> Int

produceInteger = runST (do
    n <- readRef location
    writeRef location (n + 1)
    return n)

definitelyTrue :: Bool
definitelyTrue = produceInteger () == produceInteger ()

-- PROBLEMS AHEAD
```
1 -- NAIVE Interface of the haskell ST-Monad
2
3   ST :: * -> *
4   Ref :: * -> *
5
6   newRef :: a -> ST (Ref a)
7   readRef :: Ref a -> ST a
8   writeRef :: Ref a -> a -> ST ()
9
10  return :: a -> ST a
11  (>>=) :: ST a -> (a -> ST b) -> ST b
12
13  runST :: ST a -> a
14
15  -- PROBLEMS AHEAD
16
17  location :: Ref Int
18  location :: runST (newRef 0)
19
20  produceInteger :: () -> Int
21  produceInteger = runST (do
22      n <- readRef location
23      writeRef location (n + 1)
24      return n)
25
26  definitelyTrue :: Bool
27  definitelyTrue = produceInteger () == produceInteger ()
```
-- NAIVE Interface of the haskell ST-Monad

ST :: * -> *
Ref :: * -> *

newRef :: a -> ST (Ref a)
readRef :: Ref a -> ST a
writeRef :: Ref a -> a -> ST ()

return :: a -> ST a
(>>=) :: ST a -> (a -> ST b) -> ST b

runST :: ST a -> a

-- PROBLEMS AHEAD

location :: Ref Int

location :: runST (newRef 0)

produceInteger :: () -> Int
produceInteger = runST (do
  n <- readRef location
  writeRef location (n + 1)
  return n)

definitelyTrue :: Bool
definitelyTrue = produceInteger () == produceInteger ()
-- NAIVE Interface of the haskell ST-Monad

ST :: * -> *
Ref :: * -> *

newRef :: a -> ST (Ref a)
readRef :: Ref a -> ST a
writeRef :: Ref a -> a -> ST ()

return :: a -> ST a
(>>=) :: ST a -> (a -> ST b) -> ST b

-- PROBLEMS AHEAD

location :: Ref Int
location :: runST (newRef 0)

produceInteger :: () -> Int
produceInteger = runST (do
    n <- readRef location
    writeRef location (n + 1)
    return n)

definitelyTrue :: Bool
definitelyTrue = produceInteger () == produceInteger ()
tics of the state-transformer function, so all the usual techniques for reasoning about functional programs continue to work. Similarly, stateful programs can be exposed to the full range of program transformations applied by a compiler, with no special cases or side conditions.

- The programmer has complete control over where in-place updates are used and where they are not. For example, there is no complex analysis to determine when an array is used in a single-threaded way. Since the visibility of the entire program may be predicated on the use of in-place updates, the programmer must be confident in, and be able to reason about, the outcome.

- Mutable objects can be named. This ability sounds innocuous enough, but once an object can be named its use cannot be controlled as readily. Yet naming is important. For example, it gives us the ability to manipulate multiple mutable objects simultaneously.

- Input/output takes its place as a specialised form of stateful computation. Indeed, the type of I/O-performing computations is an instance of the (more polymorphic) type of stateful computations. Along with I/O comes the ability to call imperative procedures written in other languages.

- It is possible to encapsulate stateful computations so that they appear to the rest of the program as pure (stateless) functions which are guaranteed by the type system to have no interactions whatever with other computations, whether stateful or otherwise (except via the values of arguments and results, of course).

Complete safety is maintained by this encapsulation. A program may contain an arbitrary number of stateful sub-computations, each simultaneously active, without concern that a mutable object from one might be mutated by another.

- Stateful computations can even be performed lazily without losing safety. For example, suppose that stateful depth-first search of a graph returns a list of vertices in depth-first order. If the consumer of this list only evaluates the first few elements of the list, then only enough of the stateful computation is executed to produce those elements.

2 Overview

This section introduces the key ideas of our approach to stateful computation. We begin with the programmer’s eye-view.

2.1 State transformers

A value of type \( ST \ a \) is a computation which transforms a state indexed by type \( a \), and delivers a value of type \( a \). You can think of it as a box, like this:

\[ \text{State in} \rightarrow \text{Result} \]

Notice that this is a purely functional account of state. The “\( ST \)” stands for “a state transformer”, which we take to be synonymous with “a stateful computation”: the computation is seen as transforming one state into another. (Of course, it is our intention that the new state will actually be constructed by modifying the old one in place, a matter to which we return in Section 6.) A state transformer is a first-class value; it can be passed to a function, returned as a result, stored in a data structure, duplicated freely, and so on.

A state transformer can have other inputs besides the state; if so, it will have a functional type. It can also have many results, by returning them in a tuple. For example, a state transformer with two inputs of type \( \text{Int} \) and two results of type \( \text{Int} \) and \( \text{Bool} \) would have the type:

\[ \text{Int} \rightarrow \text{Int} \rightarrow \text{ST} \ a \ (\text{Int},\text{Bool}) \]

Its picture might look like this:

\[ \text{Inputs} \rightarrow \text{Results} \]

The simplest state transformer, \( \text{returnST} \), simply delivers a value without affecting the state at all:

\( \text{returnST} :: a \rightarrow \text{ST} \ a \ a \)

The picture for \( \text{returnST} \) is like this:

\[ \text{State in} \rightarrow \text{State out} \]

2.2 References

What, then, is a “state”? Part of every state is a finite mapping from references to values. (A state may also have other components, as we will see in Section 4.) A reference can be thought of as the name of (or address of)
So all the usual techniques for reasoning about functional programs continue to work...

It is possible to encapsulate stateful computations so that they appear to the rest of the programs as pure (stateless) functions which are guaranteed by the type system to have no interaction whatever with other computations...

2.1 State transformers

A value of type \(SI \to a\) is a computation which transforms a state indexed by type \(S\), and delivers a value of type \(a\). You can think of it as a box, like this:

\[
\text{State in} \quad \text{Result} \quad \text{State out}
\]

Notice that this is a purely functional account of state. The \(SI\) stands for "state transformer", which we take to be synonymous with "a stateful computation": the computation is seen as transforming one state into another. (Of course, it is our intention that the new state will actually be constructed by modifying the old one in place, a matter to which we return in Section 6.) A state transformer is a first-class value: it can be passed to a function, returned as a result, stored in a data structure, duplicated freely, and so on.

A state transformer can have other inputs besides the state; if so, it will have a functional type. It can also have many results, by returning them in a tuple. For example, a state transformer with two inputs of type \(Int\), and two results of type \(Int\) and \(Bool\), would have the type:

\[
Int \to Int \to SI \to (Int,Bool)
\]

Its picture might look like this:

\[
\text{Inputs} \quad \text{Results} \quad \text{State in} \quad \text{State out}
\]

The simplest state transformer, \(\text{returnST}\), simply delivers a value without affecting the state at all:

\[
\text{returnST} :: a \to SI \to a
\]

The picture for \(\text{returnST}\) is like this:

\[
\text{State in} \quad \text{State out}
\]

2.2 References

What, then, is a "state"? Part of every state is a finite mapping from references to values. (A state may also have other components, as we will see in Section 4.) A reference can be thought of as the name of (or address of) 

---

[Note: The text contains some formatting issues such as missing symbols and alignment issues which might affect readability. The content is fragmented and requires restructuring for better coherence.]
-- Interface of the haskell ST-Monad

ST :: 📳 -> * -> *

Ref :: 📳 -> * -> *

newRef :: ∀ a, s. a -> ST s (Ref s a)
readRef :: ∀ a, s. Ref s a -> ST s a
writeRef :: ∀ a, s. Ref s a -> a -> ST s ()

return :: ∀ a, s. a -> ST s a
(>>=) :: ∀ a, b, s. ST s a -> (a -> ST s b) -> ST s b
runST :: ∀ a. (∀ s. ST s a) -> a

location :: RunST (newRef 0)
1  -- Interface of the haskell ST-Monad
2
3  ST :: N -> * -> *
4  Ref :: N -> * -> *
5
6  newRef :: ∀ a, s. a -> ST s (Ref s a)
7  readRef :: ∀ a, s. Ref s a -> ST s a
8  writeRef :: ∀ a, s. Ref s a -> a -> ST s ()
9
10 return :: ∀ a, s. a -> ST s a
11 (>>=) :: ∀ a, b, s. ST s a -> (a -> ST s b) -> ST s b
12
13 runST :: ∀ a. (∀ s. ST s a) -> a
14
15 location :: Ref s Int
16 location :: runST (newRef 0)
-- Interface of the haskell ST-Monad

ST :: 𝓝 -> * -> *
Ref :: 𝓝 -> * -> *

newRef :: ∀ a, s. a -> ST s (Ref s a)
readRef :: ∀ a, s. Ref s a -> ST s a
writeRef :: ∀ a, s. Ref s a -> a -> ST s ()

return :: ∀ a, s. a -> ST s a
(>>=) :: ∀ a, b, s. ST s a -> (a -> ST s b) -> ST s b

runST :: ∀ a. ( ∀ s. ST s a) -> a

location :: Ref s Int
location :: runST (newRef 0)
-- Interface of the haskell ST-Monad

ST :: 𝓝  -> * -> *
Ref  :: 𝓝  -> * -> *

newRef  :: ∀  a, s. a -> ST  s ( Ref  s a)
readRef  ::  ∀  a, s. Ref  s a -> ST  s a
writeRef  :: ∀  a, s. Ref  s a -> a -> ST  s ()

runST  :: ∀  a. ( ∀  s. ST  s a) -> a

location  :: Ref  s Int

location  :: runST (newRef 0 )

location :: runST (newRef 0 )
-- Interface of the haskell ST-Monad

ST :: 𝓝  -> * -> * :: ∀  a, b, s. ST  s a -> (a -> ST  s b) -> ST  s b

runST :: ∀  a. ( ∀  s. ST  s a) -> a

location :: Ref  s Int

location :: runST (newRef 0 )
1 -- Interface of the haskell ST-Monad
2
3 ST :: N -> * -> *
4 Ref :: N -> * -> *
5
6 newRef :: ∀ a, s. a -> ST s (Ref s a)
7 readRef :: ∀ a, s. Ref s a -> ST s a
8 writeRef :: ∀ a, s. Ref s a -> a -> ST s ()
9
10 return :: ∀ a, s. a -> ST s a
11 (>>=) :: ∀ a, b, s. ST s a -> (a -> ST s b) -> ST s b
12
13 runST :: ∀ a. (∀ s. ST s a) -> a
14
15 location :: Ref s Int
16 location :: runST (newRef 0)
So all the usual techniques for reasoning about functional programs continue to work...
“So all the usual techniques for reasoning about functional programs continue to work...
Given two programs, $e_1$ and $e_2$, they are contextually equivalent, $e_1 \approx_{ctx} e_2$

if

$\forall C. C[e_1]$ behaves the same as $C[e_2]$
Any two pure programs, $e_1$ and $e_2$, contextually equivalent in the pure language, should be contextually equivalent in the extended stateful language.
Any two pure programs, $e_1$ and $e_2$, contextually equivalent in the pure language, should be contextually equivalent in the extended stateful language.
Any two pure programs, \( e_1 \) and \( e_2 \), contextually equivalent in the pure language, should be contextually equivalent in the extended stateful language.

"No pure context can distinguish the two"

"No stateful context can distinguish the two"
Any two pure programs, $e_1$ and $e_2$, contextually equivalent in the pure language, should be contextually equivalent in the extended stateful language.

“**Statefulness does not provide us with any more distinguishability**

“**No pure context can distinguish the two**

“**No stateful context can distinguish the two**
Any two pure programs, $e_1$ and $e_2$, contextually equivalent in the pure language, should be contextually equivalent in the extended stateful language.

"Statefulness does not provide us with any more distinguishability" "No pure context can distinguish the two"

"No stateful context can distinguish the two"

Note: full abstraction is 1) preservation of ctx. equiv. and 2) reflection of ctx. equiv.
sort :: (Int -> Int -> Bool) -> List Int -> List Int

bloo :: ST n Bool -> Int
bloo = runST ...

blaa :: Int -> Int
blaa n = runST ...

sort :: (Int -> Int -> Bool) -> List Int -> List Int
sort = ...

foo :: (Int -> Int -> Bool)
faa n m = runST ...
sort :: (Int -> Int -> Bool) -> List Int -> List Int

bloo :: ST n Bool -> Int
bloo = runST ...

blaa :: Int -> Int
blaa n = runST ...

sort :: (Int -> Int -> Bool) -> List Int -> List Int
sort = ...

foo :: (Int -> Int -> Bool)
foo n m = runST ...
```
1  bloo :: ST n Bool -> Int
2  bloo = runST ...
3
4  blaa :: Int -> Int
5  blaa n = runST ...
6
7  sort :: (Int -> Int -> Bool) -> List Int -> List Int
8  sort = ...
9
10 foo :: (Int -> Int -> Bool)
11  faa n m = runST ...
```
Adding ST to Haskell → Adding ST to $STLC_{\mu}$
Adding ST to Haskell → Adding ST to $STLC_\mu$

- No polymorphism -
- Call by Value -
\( \text{STLC}_\mu \quad \lambda^+ \)

\[ \tau ::= 1 \mid Z \mid B \mid \tau \times \tau \mid \tau + \tau \mid X \mid \mu X. \tau \mid \tau \to \tau \]

\[ \Gamma \vdash e : \tau \]
\[ \text{Extension with ST} \] \( \lambda_{\text{ST}} \)

\[ \tau ::= 1 \mid Z \mid B \mid \tau \times \tau \mid \tau + \tau \mid X \mid \mu X. \tau \mid \tau \rightarrow \tau \]

\[ \Gamma \vdash e : \tau \]
\[ STLC_{\mu} \quad \lambda^{\Downarrow} \]

\[ \tau ::= 1 \mid Z \mid B \mid \tau \times \tau \mid \tau + \tau \mid X \mid \mu X.\tau \mid \tau \rightarrow \tau \]

\[ \Gamma \vdash e : \tau \]

---

**Extension with ST** \[ \lambda_{ST}^{\Downarrow} \]

---

<table>
<thead>
<tr>
<th>[ \ell \not\in \text{dom}(h) ]</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ \langle h, \text{ref } v \rangle \leadsto_h \langle h \cup { \ell \mapsto v }, \text{return } \ell \rangle ]</td>
</tr>
</tbody>
</table>

| \[ \langle h \cup \{ \ell \mapsto v \}, \ell \mapsto v' \rangle \leadsto_h \langle h \cup \{ \ell \mapsto v \}, \text{return } () \rangle \] |
$$\text{STLC}_\mu \quad \lambda^\uparrow$$

$$\tau ::= 1 \mid Z \mid B \mid \tau \times \tau \mid \tau + \tau \mid X \mid \mu X.\tau \mid \tau \to \tau$$

$$\Gamma \vdash e : \tau$$

---

**Extension with ST**

$$\lambda_{ST}^\downarrow$$

$$\tau ::= \ldots \mid \text{STRef X } \tau \mid \text{ST X } \tau$$

$$\Xi \mid \Gamma \vdash e : \tau$$

$$\begin{align*}
\ell \not\in \text{dom}(h) & \quad \frac{}{\langle h, \text{ref } v \rangle \leadsto_h \langle h \cup \{\ell \mapsto v\} \rangle, \text{return } \ell} \\
\langle h \cup \{\ell \mapsto v\}, !\ell \rangle & \leadsto_h \langle h \cup \{\ell \mapsto v\} \rangle, \text{return } v \\
\langle h \cup \{\ell \mapsto v'\}, \ell \leftarrow v \rangle & \leadsto_h \langle h \cup \{\ell \mapsto v\} \rangle, \text{return } ()
\end{align*}$$
\[ \text{STLC}_\mu \quad \lambda^\top \]

\[ \tau ::= 1 \mid Z \mid B \mid \tau \times \tau \mid \tau + \tau \mid X \mid \mu X.\tau \mid \tau \Rightarrow \tau \]

\[ \Gamma \vdash e : \tau \]

---

**Extension with ST** \[ \lambda^\top_{ST} \]

\[ \tau ::= \ldots \mid \text{STRef } X \tau \mid \text{ST } X \tau \]

\[ \Xi \mid \Gamma \vdash e : \tau \]

\[ \begin{array}{l}
\Xi, X \mid \Gamma \vdash e : \text{ST } X \tau \quad \Xi \vdash \tau \\
\Xi \mid \Gamma \vdash \text{runST } \{ e \} : \tau \\
\Xi \mid \Gamma \vdash e : \tau \quad \Xi \vdash X \\
\Xi \mid \Gamma \vdash \text{ref } e : \text{ST } X (\text{STRef } X \tau) \\
\Xi \mid \Gamma \vdash e : \text{STRef } X \tau \quad \Xi \mid \Gamma \vdash e' : \tau \\
\Xi \mid \Gamma \vdash e \leftarrow e' : \text{ST } X 1
\end{array} \]
$\mathcal{STLC}_\mu \quad \lambda^\dashv$

\[
\tau ::= 1 \mid Z \mid B \mid \tau \times \tau \mid \tau + \tau \mid X \mid \mu X. \tau \mid \tau \rightarrow \tau
\]

$\Gamma \vdash e : \tau$

---

Extension with ST $\lambda^\dashv_{ST}$

\[
\tau ::= \ldots \mid \text{STRef } X \tau \mid \text{ST } X \tau
\]

$\Xi \mid \Gamma \vdash e : \tau$

---

\begin{align*}
\frac{\ell \notin \text{dom}(h)}{\langle h, \text{ref } v \rangle \sim_h \langle h \cup \{ \ell \mapsto v \}, \text{return } \ell \rangle} & \\
\frac{\langle h \cup \{ \ell \mapsto v \},!\ell \rangle \sim_h \langle h \cup \{ \ell \mapsto v \}, \text{return } v \rangle}{\Xi, X \mid \Gamma \vdash ST X \tau \quad \Xi \vdash \tau} & \\
\frac{\Xi \mid \Gamma \vdash e : \tau \quad \Xi \vdash X}{\Xi \mid \Gamma \vdash \text{runST } \{ e \} : \tau} & \\
\frac{\Xi \mid \Gamma \vdash e \in \text{STRef } X \tau}{\Xi \mid \Gamma \vdash \text{STRef } X (\text{STRef } X \tau)} & \\
\frac{\Xi \mid \Gamma \vdash e : \text{STRef } X \tau \quad \Xi \mid \Gamma \vdash e' : \tau}{\Xi \mid \Gamma \vdash e \leftarrow e' : \text{ST } X 1} & \\
\frac{\Xi \mid \Gamma \vdash e \leftarrow e' : \text{ST } X 1}{\Xi \mid \Gamma \vdash e : \text{ST } X 1}
\end{align*}
\[ \text{STLC}_\mu \quad \lambda^\downarrow \]

\[ \tau ::= 1 \mid Z \mid B \mid \tau \times \tau \mid \tau^+ \tau \mid X \mid \mu X.\tau \mid \tau \to \tau \]

\[ \Gamma \vdash e : \tau \]

---

\textit{Extension with ST} \[ \lambda^\downarrow_{\text{ST}} \]

\[ \tau ::= \ldots \mid \text{STRef} \ X \ \tau \mid \text{ST} \ X \ \tau \]

\[ \exists \mid \Gamma \vdash e : \tau \]

\[ \frac{\ell \not\in \text{dom}(h)}{\langle h, \text{ref} \ v \rangle \rightsquigarrow_h \langle h \cup \{ \ell \mapsto v \}, \text{return} \ \ell \rangle} \]

\[ \langle h \cup \{ \ell \mapsto v \}, !\ell \rangle \rightsquigarrow_h \langle h \cup \{ \ell \mapsto v \}, \text{return} \ v \rangle \]

\[ \langle h \cup \{ \ell \mapsto v' \}, \ell \leftarrow v \rangle \rightsquigarrow_h \langle h \cup \{ \ell \mapsto v \}, \text{return} \ () \rangle \]

\[ \exists, X \mid \Gamma \vdash e : \text{ST} \ X \ \tau \quad \exists \vdash \tau \]

\[ \exists \mid \Gamma \vdash \text{runST} \{ e \} : \tau \]

\[ \exists \mid \Gamma \vdash e : \tau \quad \exists \vdash X \]

\[ \exists \mid \Gamma \vdash \text{ref} \ e : \text{ST} \ X \ (\text{STRef} \ X \ \tau) \]

\[ \exists \mid \Gamma \vdash e : \text{STRef} \ X \ \tau \quad \exists \mid \Gamma \vdash e' : \tau \]

\[ \exists \mid \Gamma \vdash e \leftarrow e' : \text{ST} \ X \ 1 \]

11.6
\[ \Pi : \lambda^r \rightarrow \lambda_{ST}^r \]

If \( \Gamma \vdash e_1 \approx_{ctx} e_2 : \tau \), then \( \mid \Pi \mid \vdash \mid e_1 \mid \approx_{ctx} \mid e_2 \mid : \mid \tau \mid \)
\[ \prod \Xi : \mathcal{L}^+ \to \mathcal{L}_{ST}^+ \]

If \( \Gamma \vdash e_1 \approx_{ctx} e_2 : \tau \), then \( \cdot \mid \prod \Xi \vdash \Xi e_1 \Xi \approx_{ctx} \Xi e_2 \Xi : \Xi \tau \Xi \)

If \( \forall \vdash C : (\Gamma ; \tau) \Rightarrow (\cdot ; 1) \)
\( C[e_1] \downarrow \iff C[e_2] \downarrow \)

then \( \forall \vdash C : (\cdot \mid \Xi \Xi ; \Xi \tau \Xi) \Rightarrow (\cdot \mid \cdot ; 1) \)
\( C[\Xi e_1 \Xi] \downarrow \iff C[\Xi e_2 \Xi] \downarrow \)
\( \Pi \triangleright : \lambda^+ \rightarrow \lambda^+ \text{_{ST}} \)

If \( \Gamma \vdash e_1 \approx_{ctx} e_2 : \tau \), then \( \cdot | \lfloor \Pi \rfloor \vdash \lfloor e_1 \rfloor \approx_{ctx} \lfloor e_2 \rfloor : \lfloor \tau \rfloor \)

If \( \forall \vdash C : (\Pi; \tau) \Rightarrow (\cdot ; 1) \)
\( C[e_1] \downarrow \text{ iff } C[e_2] \downarrow \)

then \( \forall \vdash C : (\cdot | \lfloor \Pi \rfloor ; \lfloor \tau \rfloor) \Rightarrow (\cdot | \cdot ; 1) \)
\( C[\lfloor e_1 \rfloor] \downarrow \text{ iff } C[\lfloor e_2 \rfloor] \downarrow \)

Given \( \vdash C : (\cdot | \lfloor \Pi \rfloor ; \lfloor \tau \rfloor) \Rightarrow (\cdot | \cdot ; 1) \), there exists a context \( \vdash C^+_b : (\Pi; \tau) \Rightarrow (\cdot ; 1) \).
This is a valid emulation, i.e. for all \( \Gamma \vdash e : \tau \) we have:
\( C^+_b[e] \downarrow \text{ iff } C[\lfloor e \rfloor] \downarrow \)
\( \prod_\prod : \lambda^+ \rightarrow \lambda^\text{ST} \)

If \( \Gamma \vdash e_1 \approx_{\text{ctx}} e_2 : \tau \), then \( \cdot \mid \prod \Gamma \vdash \prod e_1 \approx_{\text{ctx}} \prod e_2 : \prod \tau \)

If \( \forall \vdash C : (\Gamma ; \tau) \Rightarrow (\cdot ; 1) \)
\( C[e_1] \downarrow \text{iff} \ C[e_2] \downarrow \)

then \( \forall \vdash C : (\cdot \mid \prod \Gamma ; \prod \tau) \Rightarrow (\cdot \mid \cdot ; 1) \)
\( C[\prod e_1] \downarrow \text{iff} \ C[\prod e_2] \downarrow \)

"Statefulness can be purely emulated"

Given \( \vdash C : (\cdot \mid \prod \Gamma ; \prod \tau) \Rightarrow (\cdot \mid \cdot ; 1) \), there exists a context \( \vdash C^+: (\Gamma ; \tau) \Rightarrow (\cdot ; 1) \).
This is a valid emulation, i.e. for all \( \Gamma \vdash e : \tau \) we have:
\( C^+_b[e] \downarrow \text{iff} \ C[\prod e] \downarrow \)
\[
\| - \| : \lambda^+ \rightarrow \lambda_{ST}^+
\]

If \( \Gamma \vdash e_1 \approx_{ctx} e_2 : \tau \), then \( \cdot | \quad \| \Gamma \| \vdash \| e_1 \| \approx_{ctx} \| e_2 \| : \| \tau \| \)

If \( \forall \vdash C : (\Gamma; \tau) \Rightarrow (\cdot; 1) \)
\[
\begin{align*}
C[e_1] & \Downarrow \iff C[e_2] \Downarrow
\end{align*}
\]

, then

\[
\begin{align*}
\forall + C : (\cdot | \| \Gamma \|; \| \tau \|) & \Rightarrow (\cdot | \cdot; 1) \\
C[\| e_1 \|] & \Downarrow \iff C[\| e_2 \|] \Downarrow
\end{align*}
\]

“Statefulness can be purely emulated”

Given \( \vdash C : (\cdot | \| \Gamma \|; \| \tau \|) \Rightarrow (\cdot | \cdot; 1) \), there exists a context \( \vdash C_b^+ : (\Gamma; \tau) \Rightarrow (\cdot; 1) \). This is a valid emulation, i.e. for all \( \Gamma \vdash e : \tau \) we have:

\[
C_b^+[e] \Downarrow \iff C[\| e \|] \Downarrow
\]
“Statefulness can be purely emulated.”
A stateful computation is like a state transformer, that is, a function from an initial state to a final state. It is like a "script" detailing the actions to be performed on its input state...
Statefulness can be purely emulated

A stateful computation is like a state transformer, that is, a function from an initial state to a final state. It is like a "script" detailing the actions to be performed on its input state...
A stateful computation is like a state transformer, that is, a function from an initial state to a final state. It is like a "script" detailing the actions to be performed on its input state...
ST computations by State Monad?

\[ \langle\langle ST \times B \rangle\rangle \overset{?}{=} H \rightarrow B \times H \]
ST computations by State Monad?

\[ \langle \langle \text{ST} \times B \rangle \rangle \cong H \rightarrow B \times H \]

Given \( \vdash C : (\cdot | [\Gamma];[\tau]) \Rightarrow (\cdot | \cdot;1) \), there exists a context \( \vdash C^b : (\Gamma;\tau) \Rightarrow (\cdot;1) \). This is a valid emulation, i.e. for all \( \Gamma \vdash e : \tau \) we have:

\[ C^b[e] \Downarrow \text{iff } C[[e]] \Downarrow \]
ST computations by State Monad?

\[ \langle \langle \text{ST} \times \text{B} \rangle \rangle \cong \mathbb{H} \rightarrow \text{B} \times \mathbb{H} \]

Given \( \vdash C : (\cdot | [\Gamma]; [\tau]) \Rightarrow (\cdot | \cdot; 1) \), there exists a context \( \vdash C' : (\Gamma; \tau) \Rightarrow (\cdot; 1) \). This is a valid emulation, i.e. for all \( \Gamma \vdash e : \tau \) we have:

\[ C'_b[e] \downarrow \iff C[[e]] \downarrow \]
ST computations by State Monad?

\[ \langle \langle ST \times B \rangle \rangle \overset{?}{\Rightarrow} \mathbb{H} \rightarrow B \times H \]

*Given \( \vdash C : (\cdot | [\Gamma]; [\tau]) \Rightarrow (\cdot | \cdot ; 1) \), there exists a context \( \vdash C^x_b : (\Gamma; \tau) \Rightarrow (\cdot ; 1) \). This is a valid emulation, i.e. for all \( \Gamma \vdash e : \tau \) we have:*

\[ C^x_b[e] \downarrow \text{ iff } C[[e]] \downarrow \]
ST computations by *Untyped* State Monad

\[ \langle ST \times B \rangle = UntypedStore \to UntypedStore \times B \]
ST computations by *Untyped* State Monad

\[ \langle \langle ST \times B \rangle \rangle = \text{UntypedStore} \rightarrow \text{UntypedStore} \times B \]

\[ \mathcal{E} : \text{List Val} \rightarrow \text{Val} \]
\[ \mathcal{E}([v_1; v_2; v_3]) = ((v_3, (v_2, (v_1, ()))), 3) \]
ST computations by *Untyped* State Monad

\[ \langle \langle \text{ST} \times \text{B} \rangle \rangle = \text{UntypedStore} \rightarrow \text{UntypedStore} \times \text{B} \]

\[ \mathcal{E} : \text{List} \ \text{Val} \rightarrow \text{Val} \]
\[ \mathcal{E}([v_1; v_2; v_3]) = ((v_3, (v_2, (v_1, ()))), 3) \]

read \( z \) \( \mathcal{E}(\overline{v}) \rightarrow^{*} (\mathcal{E}(\overline{v}), \overline{v}.z) \) if \( 0 \leq z < |\overline{v}| \)

ref \( v \) \( \mathcal{E}(\overline{v}) \rightarrow^{*} (\mathcal{E}(\overline{v} \leftrightarrow [v]), |\overline{v}|) \)

write \( z \) \( v \) \( \mathcal{E}(\overline{v}) \rightarrow^{*} (\mathcal{E}(\overline{v}[z \mapsto v]), ()) \) if \( 0 \leq z < |\overline{v}| \)
Decomposing the syntactic-typing problem
Decomposing the syntactic-typing problem
Decomposing the *syntactic-typing* problem
Stateful contexts can be emulated by pure, syntactically-typed contexts

Given $\vdash C : (\cdot \mid \llbracket \Gamma \rrbracket ; \llbracket \tau \rrbracket ) \Rightarrow (\cdot \mid \cdot ; 1)$, there exists a context $\vdash C^+_b : (\Gamma ; \tau ) \Rightarrow (\cdot ; 1)$. This is a valid emulation, i.e. for all $\Gamma \vdash e : \tau$ we have:

$$C^+_b[e] \downarrow \text{iff } C[\llbracket e \rrbracket] \downarrow$$
Stateful contexts can be emulated by pure, syntactically-typed contexts

Given $\vdash C : (\cdot | \left[\Gamma\right]; \left[\tau\right]) \Rightarrow (\cdot | \cdot ; 1)$, there exists a context $\vdash C^+_b : (\Gamma; \tau) \Rightarrow (\cdot ; 1)$. This is a valid emulation, i.e. for all $\Gamma \vdash e : \tau$ we have:

$$C^+_b[e] \Downarrow \iff C[\left[e\right]] \Downarrow$$

Stateful contexts can be emulated by pure, semantically-typed contexts

Given $\vdash C : (\cdot | \left[\Gamma\right]; \left[\tau\right]) \Rightarrow (\cdot | \cdot ; 1)$, there exists $\vdash_{int.} C^+_b : (\Gamma; \tau) \Rightarrow (\cdot ; 1)$. This is a valid emulation, i.e. for all $\Gamma \vdash e : \tau$:

$$C^+_b[e] \Downarrow \iff C[\left[e\right]] \Downarrow$$
Stateful contexts can be emulated by pure, syntactically-typed contexts

Given $\vdash C : (\cdot \mid \|\Gamma\|; \|\tau\|) \Rightarrow (\cdot \mid \cdot ; 1)$, there exists a context $\vdash C^+_b : (\Gamma ; \tau) \Rightarrow (\cdot ; 1)$. This is a valid emulation, i.e. for all $\Gamma \vdash e : \tau$ we have:

$$C^+_b[e] \downarrow \text{iff } C[\|e\|] \downarrow$$

Stateful contexts can be emulated by pure, semantically-typed contexts

Given $\vdash C : (\cdot \mid \|\Gamma\|; \|\tau\|) \Rightarrow (\cdot \mid \cdot ; 1)$, there exists $\vdash_{int.} C^+_b : (\Gamma ; \tau) \Rightarrow (\cdot ; 1)$. This is a valid emulation, i.e. for all $\Gamma \vdash e : \tau$:

$$C^+_b[e] \downarrow \text{iff } C[\|e\|] \downarrow$$

Pure, semantically-typed contexts can be emulated by pure, syntactically-typed contexts

Given $\vdash_{int.} C : (\Gamma ; \tau) \Rightarrow (\cdot ; 1)$, there exists $\vdash C^+_b : (\Gamma ; \tau) \Rightarrow (\cdot ; 1)$. This is a valid emulation, i.e. for all $\Gamma \vdash e : \tau$ we have:

$$C^+_b[e] \downarrow \text{iff } C[e] \downarrow$$
"Stateful contexts can be emulated by pure, syntactically-typed contexts

Given $\vdash C : (\cdot | \llbracket \Gamma \rrbracket; \llbracket \tau \rrbracket) \Rightarrow (\cdot | \cdot; 1)$, there exists a context $\vdash C_b^+ : (\Gamma; \tau) \Rightarrow (\cdot; 1)$. This is a valid emulation, i.e. for all $\Gamma \vdash e : \tau$ we have:

$$C_b^+[e] \Downarrow \iff C[\llbracket e \rrbracket] \Downarrow$$

"Stateful contexts can be emulated by pure, semantically-typed contexts

Given $\vdash C : (\cdot | \llbracket \Gamma \rrbracket; \llbracket \tau \rrbracket) \Rightarrow (\cdot | \cdot; 1)$, there exists $\vdash_{int.} C_b^+ : (\Gamma; \tau) \Rightarrow (\cdot; 1)$. This is a valid emulation, i.e. for all $\Gamma \vdash e : \tau$:

$$C_b^+[e] \Downarrow \iff C[\llbracket e \rrbracket] \Downarrow$$

"Pure, semantically-typed contexts can be emulated by pure, syntactically-typed contexts

Given $\vdash_{int.} C : (\Gamma; \tau) \Rightarrow (\cdot; 1)$, there exists $\vdash C_b^+ : (\Gamma; \tau) \Rightarrow (\cdot; 1)$. This is a valid emulation, i.e. for all $\Gamma \vdash e : \tau$ we have:

$$C_b^+[e] \Downarrow \iff C[e] \Downarrow$$
Defining Semantic Typedness

\[ \Gamma \vdash_{int.} e : \tau \]
Defining Semantic Typedness

\[ \Gamma \vdash_{int.} e \leq e : \tau \]
Logical Relations on Values

\[ \mathcal{V}_{int}[\text{1}](v, v') \triangleq v = () \ast v' = () \]

\[ \mathcal{V}_{int}[\text{B}](v, v') \triangleq \exists b \in \{\text{true, false}\}. v = b \ast v' = b \]

\[ \mathcal{V}_{int}[\text{Z}](v, v') \triangleq \exists z \in \mathbb{Z}. v = z \ast v' = z \]

\[ \mathcal{V}_{int}[\tau_1 + \tau_2](v, v') \triangleq \bigvee_{i \in \{1,2\}} \exists w, w'. v = \text{inj}_i w \ast v' = \text{inj}_i w' \ast \mathcal{V}_{int}[\tau_i](w, w') \]

\[ \mathcal{V}_{int}[\tau_1 \times \tau_2](v, v') \triangleq \exists v_1, v'_1, v_2, v'_2. v = (v_1, v_2) \ast v' = (v'_1, v'_2) \ast \mathcal{V}_{int}[\tau_1](v_1, v'_1) \ast \mathcal{V}_{int}[\tau_2](v_2, v'_2) \]

\[ \mathcal{V}_{int}[\tau_1 \rightarrow \tau_2](v, v') \triangleq \Box (\forall w, w'. \mathcal{V}_{int}[\tau_1](w, w') \ast \text{lift } \mathcal{V}_{int}[\tau_2](v, w, w')) \]

\[ \mathcal{V}_{int}[\mu X. \tau](v, v') \triangleq \exists w, w'. v = \text{fold } w \ast v' = \text{fold } w' \ast \mathcal{V}_{int}[\tau[\mu X. \tau/X]](w, w') \]
Logical Relations on Closed Expressions

\[ \text{lift} : (\text{Val} \to \text{Val} \to \text{iProp}) \to (\text{Expr} \to \text{Expr} \to \text{iProp}) \]

\[ \text{lift } \Phi (e, e') = \text{wp } e \left\{ v. \exists v'. e' \rightarrow^* v' \ast \Phi (v, v') \right\} \]

\[ \mathcal{E}_{\text{int.}}[[\tau]] = \text{lift } \mathcal{V}_{\text{int.}}[[\tau]] \]
Logical Relations on Open Expressions

$$\Gamma \vDash_{int.} e \leq e' : \tau \triangleq \forall \vec{v}, \vec{v}'. \vec{V}_{int.}([\Gamma]) (\vec{v}, \vec{v}') \vdash \mathcal{E}_{int.}([[\tau]]) (e[\vec{v}'/\vec{x}], e'[\vec{v}'/\vec{x}'])$$
Lemma 2.1 (Logical Relation Adequacy). If $\cdot \models_{int} e \leq e' : \tau$, then if $e$ halts to a value, so must $e'$.

Theorem 2.2 (Fundamental Theorem Intermediate Language). For any well syntactically typed expression (in $\lambda^+$), say $\Gamma \vdash e : \tau$, we automatically have that $\Gamma \models_{int} e : \tau$. 

Pure, semantically-typed contexts can be emulated by pure, syntactically-typed contexts.

Given \( \vdash_{\text{int.}} C : (\Gamma; \tau) \Rightarrow (\cdot ; 1) \), there exists \( \vdash C_b^\uparrow : (\Gamma; \tau) \Rightarrow (\cdot ; 1) \).

This is a valid emulation, i.e. for all \( \Gamma \vdash e : \tau \) we have:

\[
C_b^\uparrow[e] \Downarrow \iff C[e] \Downarrow
\]
\[ F_{int.} \ C : (x_1 : \tau_1, \ldots, x_n : \tau_n ; \tau) \Rightarrow (\_ ; 1) \]
\[ F_{int.} \; C : (x_1 : \tau_1, \ldots, x_n : \tau_n : \tau) \Rightarrow (\cdot ; 1) \]

\[ U = \mu X. (1 + B + Z + (X + X) + (X \times X) + (X \to X) + \mu Y.X) \]
\( F_{\text{int.}} \ C : (x_1 : \tau_1, \ldots, x_n : \tau_n ; \tau) \Rightarrow (\cdot ; 1) \)

\[
U = \mu X. (1 + B + Z + (X + X) + (X \times X) + (X \to X) + \mu Y. X)
\]
\[
\mu x. (z + (x \times (x + (x + z)) + z)) = n
\]
\[ F_{\text{int.}}(C : (x_1 : \tau_1, \ldots, x_n : \tau_n ; \tau)) \Rightarrow (\cdot ; 1) \]

\[ U = \mu X, (1 + B + Z + (X + X) + (X \times X) + (X \to X) + \mu Y.X) \]
Let $C : (x_1 : \tau_1, \ldots, x_n : \tau_n ; \tau) \Rightarrow (\cdot ; 1)$

$U = \mu X. (1 + B + Z + (X + X) + (X \times X) + (X \rightarrow X) + \mu Y.X)$

**Fully Abstract Compilation via Universal Embedding**

**Fully-Abstract Compilation by Approximate Back-Translation**

**Fully Abstract from Static to Gradual**

**On the Semantic Expressiveness of Recursive Types**

MARCO PATRIGNANI, Stanford University, USA and CISPA Helmholtz Center for Information Security, Germany

ERIC MARK MARTIN, Stanford University, USA

DOMINIQUE DEVRIESE, Vrije Universiteit Brussel, Belgium

Recursive types extend the simply-typed lambda calculus (STLC) with the additional expressive power to enable diverging computation and to encode recursive data-types (e.g., lists). Two formulations of recursive types exist: iso-recursive and equi-recursive. The relative advantages of iso- and equi-recursive are well-studied when it comes to their impact on type-inference. However, the relative semantic expressiveness of the two formulations remains unclear so far.

This paper studies the semantic expressiveness of STLC with iso- and equi-recursive types, proving that these formulations are equally expressive. In fact, we prove that they are both as expressive as STLC with
\( \langle \pi_1 \ e \rangle = \pi_1 (\text{extract}_\otimes \langle e \rangle) \)

\( \text{extract}_\otimes : U \to (U \otimes U) \)

\[ F_{\text{int.}} \ C : (x_1 : \tau_1, \ldots, x_n : \tau_n ; \tau) \Rightarrow (\ \cdot \ ; 1) \]

\[ U = \mu X. (1 + B + Z + (X + X) + (X \times X) + (X \to X) + \mu Y.X) \]

\[ \vdash \langle C \rangle : (x_1 : U, \ldots, x_n : U ; U) \Rightarrow (\ \cdot \ ; U) \]
\[
\langle \pi_1 \mathbf{e} \rangle = \pi_1 (\operatorname{extract}_\otimes \langle \mathbf{e} \rangle)
\]

\[
\operatorname{extract}_\otimes : \mathcal{U} \to (\mathcal{U} \otimes \mathcal{U})
\]

\[
\exists \mathbf{C} : (x_1 : \tau_1, \ldots, x_n : \tau_n ; \tau) \Rightarrow (\mathbf{\cdot} ; 1)
\]

\[
\mathcal{U} = \mu X. (1 + \mathbf{B} + \mathbf{Z} + (X + X) + (X \times X) + (X \to X) + \mu Y.X)
\]

\[
\vdash \langle \mathbf{C} \rangle : (x_1 : \mathcal{U}, \ldots, x_n : \mathcal{U}; \mathcal{U}) \Rightarrow (\mathbf{\cdot} ; \mathcal{U})
\]

\[
\operatorname{project}_\tau : \mathcal{U} \to \tau
\]

\[
\operatorname{embed}_\tau : \tau \to \mathcal{U}
\]
\[ \langle \pi_1 \mathbf{e} \rangle = \pi_1 (\text{extract}_{\otimes} \langle \mathbf{e} \rangle) \]

\[ \text{extract}_{\otimes} : \mathcal{U} \to (\mathcal{U} \otimes \mathcal{U}) \]

\[ \Phi_{\text{int}}. \mathbf{C} : (x_1 : \tau_1, \ldots, x_n : \tau_n ; \tau) \Rightarrow (\cdot ; 1) \]

\[ \mathcal{U} = \mu X. (1 + B + Z + (X \times X) + (X \times X) + (X \to X) + \mu Y.X) \]

\[ \vdash \langle \mathbf{C} \rangle : (x_1 : \mathcal{U}, \ldots, x_n : \mathcal{U}; \mathcal{U}) \Rightarrow (\cdot ; \mathcal{U}) \]

\[ \text{project}_\tau : \mathcal{U} \to \tau \]

\[ \text{embed}_\tau : \tau \to \mathcal{U} \]

\[ \vdash \mathcal{E}_\Phi \langle \mathbf{C} \rangle : (\Gamma ; \tau) \Rightarrow (\cdot ; 1) \]
Pure, semantically-typed contexts can be emulated by pure, syntactically-typed contexts.

Given $\vdash_{int.} C : (\Gamma; \tau) \Rightarrow (\cdot; 1)$, there exists $\vdash C^r_b : (\Gamma; \tau) \Rightarrow (\cdot; 1)$. This is a valid emulation, i.e. for all $\Gamma \vdash e : \tau$ we have:

$C^r_b[e] \Downarrow \text{ iff } C[e] \Downarrow$
Pure, semantically-typed contexts can be emulated by pure, syntactically-typed contexts

Given $\Gamma \vdash C : (\Gamma ; \tau) \Rightarrow (\cdot ; 1)$, there exists $\vdash C'_b : (\Gamma ; \tau) \Rightarrow (\cdot ; 1)$. This is a valid emulation, i.e. for all $\Gamma \vdash e : \tau$ we have:

$C'_b[e] \downarrow$ iff $C[e] \downarrow$

Our proof does not assume a fixed set of ghost resources!
Stateful contexts can be emulated by pure, semantically-typed contexts

Given \( \Gamma \vdash C : (\cdot \mid \Gamma; [\tau]) \Rightarrow (\cdot \mid \cdot; 1) \), there exists \( \Gamma \vdash C_b^\tau : (\Gamma; [\cdot]; [\cdot;1]) \Rightarrow (\cdot; 1) \). This is a valid emulation, i.e. for all \( \Gamma \vdash e : \tau \):

\[
C_b^\tau[e] \downarrow \text{iff } C[e] \downarrow
\]
Stateful contexts can be emulated by pure, semantically-typed contexts

\[ \text{Given } C : \langle \cdot, [\Gamma], [\tau] \rangle \Rightarrow (\cdot, \cdot) \Rightarrow 1, \text{ there exists } \vdash \text{int. } C^b : \langle \Gamma, \tau \rangle \Rightarrow (\cdot) \Rightarrow 1. \]

This is a valid emulation, i.e. for all \( \Gamma \vdash e : \tau \):

\[ C^b[e] \Downarrow \iff C[[e]] \Downarrow \]

\(\mathcal{E} : \text{List Val} \rightarrow \text{Val} \)

\[ \mathcal{E}([v_1; v_2; v_3]) = ((v_3, (v_2, (v_1,()))), 3) \]

- read \( z \mathcal{E}(\vec{v}) \rightarrow^* (\mathcal{E}(\vec{v}), \vec{v}.z) \) if \( 0 \leq z < |\vec{v}| \)
- ref \( v \mathcal{E}(\vec{v}) \rightarrow^* (\mathcal{E}(\vec{v} \leftrightarrow [v]), |\vec{v}|) \)
- write \( z v \mathcal{E}(\vec{v}) \rightarrow^* (\mathcal{E}(\vec{v}[z \mapsto v]), ()) \) if \( 0 \leq z < |\vec{v}| \)
"Stateful contexts can be emulated by pure, semantically-typed contexts"

Given \( \Gamma \vdash C : (\cdot \mid \Gamma; \tau) \Rightarrow (\cdot \mid \cdot; 1) \), there exists \( \vdash_{\text{int.}} C^b : (\Gamma; \tau) \Rightarrow (\cdot; 1) \). This is a valid emulation, i.e. for all \( \Gamma \vdash e : \tau \):

\[
C^b[e] \downarrow \iff C[\Gamma[e]] \downarrow
\]

\( \mathcal{E} : \text{List Val} \rightarrow \text{Val} \)

\( \mathcal{E}([v_1; v_2; v_3]) = ((v_3, (v_2, (v_1, ()))), 3) \)

\[
\lambda x. \lambda f. \lambda h_0. \text{let } (h_1, a) = x h_0 \text{ in } f a h_1 \langle e \rangle \langle e' \rangle
\]

\[
\begin{align*}
\langle \langle !e \rangle \rangle & = \text{read } \langle \langle e \rangle \rangle \\
\langle \langle e \leftarrow e' \rangle \rangle & = \text{write } \langle \langle e \rangle \rangle \langle \langle e' \rangle \rangle \\
\langle \langle \text{ref } e \rangle \rangle & = \text{ref } \langle \langle e \rangle \rangle \\
\langle \langle e \Rightarrow e' \rangle \rangle & = \lambda x. \lambda f. \lambda h_0. \text{let } (h_1, a) = x h_0 \text{ in } f a h_1 \langle e \rangle \langle e' \rangle
\end{align*}
\]
Given \( \vdash C : (\cdot | [\Gamma]; [\tau]) \Rightarrow (\cdot | \cdot; 1) \), there exists \( \equiv_{\text{int}}. C_b^\tau : (\Gamma; \tau) \Rightarrow (\cdot; 1) \). This is a valid emulation, i.e. for all \( \Gamma \vdash e : \tau \):

\[
C_b^\tau[e] \downarrow \iff C[\llbracket e \rrbracket] \downarrow
\]

Given \( \vdash C : (\cdot | [\Gamma]; [\tau]) \Rightarrow (\cdot | \cdot; 1) \),

to prove \( \equiv_{\text{int}}. \llbracket C \rrbracket \leq \llbracket \llbracket C \rrbracket \rrbracket : (\Gamma; \tau) \Rightarrow (\cdot; 1) \)
**Given** \( \vdash C : (\mathbf{\cdot} \mid \llbracket \Gamma \rrbracket ; \llbracket \tau \rrbracket) \Rightarrow (\mathbf{\cdot} \mid \mathbf{\cdot} ; 1) \), there exists \( \vdash int. C^b \mathcal{b} : (\Gamma ; \tau) \Rightarrow (\mathbf{\cdot} ; 1) \).

This is a valid emulation, i.e. for all \( \Gamma \vdash e : \tau \):

\[
C^b[e] \Downarrow \text{ iff } C[\llbracket e \rrbracket] \Downarrow
\]

**Given** \( \vdash C : (\mathbf{\cdot} \mid \llbracket \Gamma \rrbracket ; \llbracket \tau \rrbracket) \Rightarrow (\mathbf{\cdot} \mid \mathbf{\cdot} ; 1) \),

**to prove** \( \vdash int. \llangle C \rrangle \leq \llangle \llangle C \rrangle \rangle : (\Gamma ; \tau) \Rightarrow (\mathbf{\cdot} ; 1) \)

\[\Gamma \vdash int. e \leq e' : \tau\]

\[\exists \mid \Gamma \vdash \chi e \leq e' : \tau\]
Given $\vdash C : (\cdot \mid \llbracket \Gamma \rrbracket; \llbracket \tau \rrbracket) \Rightarrow (\cdot \mid \cdot; 1)$, there exists $\vdash_{\text{int}} C^b : (\Gamma; \tau) \Rightarrow (\cdot; 1)$. This is a valid emulation, i.e. for all $\Gamma \vdash e : \tau$:

$$C^b_b[e] \downarrow \text{ iff } C[\llbracket e \rrbracket] \downarrow$$

---

**Given** $\vdash C : (\cdot \mid \llbracket \Gamma \rrbracket; \llbracket \tau \rrbracket) \Rightarrow (\cdot \mid \cdot; 1)$, 

**to prove** $\vdash_{\text{int}} \langle C \rangle \leq \langle C \rangle : (\Gamma; \tau) \Rightarrow (\cdot; 1)$

$$\Gamma \vdash_{\text{int}} e \leq e' : \tau$$

$$\exists \mid \Gamma \vdash_{\chi} e \leq e' : \tau$$

$$\Gamma \vdash_{\text{int}} e \leq e' : \tau \vdash \cdot \mid \llbracket \Gamma \rrbracket \vdash_{\chi} e \leq e' : \llbracket \tau \rrbracket$$
Given $\vdash C : (\cdot \mid \llbracket \Gamma \rrbracket; \llbracket \tau \rrbracket) \Rightarrow (\cdot \mid \cdot ; 1)$, there exists $\equiv_{int.} C^\oplus_b : (\Gamma; \tau) \Rightarrow (\cdot ; 1)$. This is a valid emulation, i.e. for all $\Gamma \vdash e : \tau$:

$$C^\oplus_b[e] \downarrow \iff C[\llbracket e \rrbracket] \downarrow$$

**Given**

$\vdash C : (\cdot \mid \llbracket \Gamma \rrbracket; \llbracket \tau \rrbracket) \Rightarrow (\cdot \mid \cdot ; 1)$,

**to prove** $\equiv_{int.} \langle C \rangle \leq \langle C \rangle : (\Gamma; \tau) \Rightarrow (\cdot ; 1)$

$\Gamma \equiv_{int.} e \leq e' : \tau$

$\exists \mid \Gamma \equiv_X e \leq e' : \tau$

$\Gamma \equiv_{int.} e \leq e' : \tau \vdash \cdot \mid \llbracket \Gamma \rrbracket \equiv_X e \leq e' : \llbracket \tau \rrbracket$

$\forall \exists \mid \Gamma \vdash e : \tau. \exists \mid \Gamma \equiv_X \langle e \rangle \leq \langle e \rangle : \tau$
Given $\vdash C : (\cdot \mid \llbracket [\Gamma] \rrbracket; \llbracket \tau \rrbracket) \Rightarrow (\cdot \mid \cdot ; 1)$, there exists $\models_{\text{int.}} C^b : (\Gamma; \tau) \Rightarrow (\cdot ; 1)$.

This is a valid emulation, i.e. for all $\Gamma \vdash e : \tau$:

$$
C^b[e] \downarrow \iff C[\llbracket e \rrbracket] \downarrow
$$
Given $\vdash C : (\cdot | \llbracket \Gamma \rrbracket ; \llbracket \tau \rrbracket ) \Rightarrow (\cdot | \cdot ; 1)$, there exists $\equiv_{int.} C^\beta_b : (\Gamma ; \tau) \Rightarrow (\cdot ; 1)$.
This is a valid emulation, i.e. for all $\Gamma \vdash e : \tau$:

$$C^\beta_b[e] \downarrow \iff C[\llbracket e \rrbracket] \downarrow$$

**Theorem 5.2 (Fundamental Theorem).** Given a typed expression in $\lambda^\beta_{ST}$, say $\Xi | \Gamma \vdash e : \tau$, we have the following:

$$\Xi | \Gamma \equiv_R e \leq \langle \langle e \rangle \rangle : \tau$$
Given $\vdash C : (\cdot \mid \llbracket \Gamma \rrbracket; \llbracket \tau \rrbracket) \Rightarrow (\cdot \mid \cdot ; 1)$, there exists $\vdash_{\text{int.}} C_b^\tau : (\Gamma; \tau) \Rightarrow (\cdot ; 1)$. This is a valid emulation, i.e. for all $\Gamma \vdash e : \tau$

$$C_b^\tau[e] \downarrow \iff C[\llbracket e \rrbracket] \downarrow$$

**Theorem 5.2 (Fundamental Theorem).** Given a typed expression in $\lambda_{ST}^\tau$, say $\Xi \mid \Gamma \vdash e : \tau$, we have the following:

$$\Xi \mid \Gamma \vdash_R e \leq \langle \langle e \rangle \rangle : \tau$$

**Lemma 5.3 (Logical Relation Adequacy).** If $\cdot \mid \cdot \vdash_R e \leq e' : \tau$, then if $e$ halts to a value, so must $e'$. 
\[ \text{OwnState}_\gamma(\vec{v}) \vdash \Rightarrow \text{OwnState}_\gamma(\vec{v} \leftrightarrow [v]) \ast |\vec{v}| \mapsto_\gamma v \]

\[ \text{OwnState}_\gamma(\vec{v}) \ast z \mapsto_\gamma v \vdash \vec{v}.z = v \]

\[ \text{OwnState}_\gamma(\vec{v}) \ast z \mapsto_\gamma v \vdash \Rightarrow \text{OwnState}_\gamma(\vec{v}[z \mapsto w]) \ast z \mapsto_\gamma w \]
\[ OwnState_\gamma(\vec{v}) \vdash \Rightarrow OwnState_\gamma(\vec{v} \leftrightarrow [v]) * |\vec{v}| \mapsto_\gamma v \]

\[ OwnState_\gamma(\vec{v}) * z \mapsto_\gamma v \vdash \vec{v}.z = v \]

\[ OwnState_\gamma(\vec{v}) * z \mapsto_\gamma v \vdash \Rightarrow OwnState_\gamma(\vec{v}[z \mapsto w]) * z \mapsto_\gamma w \]

\[ OwnLocs_\gamma(\vec{\ell}) \vdash \Rightarrow OwnLocs_\gamma(\vec{\ell} \leftrightarrow [\ell]) * |\vec{\ell}| \mapsto_\gamma \Box \ell \]

\[ OwnLocs_\gamma(\vec{\ell}) * z \mapsto_\gamma \Box \ell \vdash \vec{\ell}.z = \ell \]

\[
\begin{align*}
    z & \mapsto_\gamma \Box \ell \vdash z \mapsto_\gamma \Box \ell * z \mapsto_\gamma \Box \ell \\
    z & \mapsto_\gamma \Box \ell * z \mapsto_\gamma \Box \ell' \vdash \ell = \ell'
\end{align*}
\]
\[ \text{OwnState}_\gamma (\vec{\nu}) + \implies \text{OwnState}_\gamma (\vec{\nu} \leftrightarrow [v]) \ast |\vec{\nu}| \mapsto_\gamma v \]

\[ \text{OwnState}_\gamma (\vec{\nu}) \ast z \mapsto_\gamma \nu \mapsto \vec{\nu}.z = v \]

\[ \text{OwnState}_\gamma (\vec{\nu}) \ast z \mapsto_\gamma \nu \mapsto \implies \text{OwnState}_\gamma (\vec{\nu}[z \mapsto w]) \ast z \mapsto_\gamma w \]

\[ \text{OwnLocs}_\gamma (\vec{\ell}) + \implies \text{OwnLocs}_\gamma (\vec{\ell} \leftrightarrow [\ell]) \ast |\vec{\ell}| \mapsto \square \ell \]

\[ \text{OwnLocs}_\gamma (\vec{\ell}) \ast z \mapsto \square \ell \mapsto \vec{\ell}.z = \ell \]

\[ z \mapsto \square \ell \mapsto z \mapsto \square \ell \ast z \mapsto \square \ell \]

\[ z \mapsto \square \ell \ast z \mapsto \square \ell' \mapsto \ell = \ell' \]

\[ \text{lift}_R : (\text{Val} \to \text{Val} \to i\text{Prop}) \to (\text{Expr} \to \text{Expr} \to i\text{Prop}) \]

\[ \text{lift}_R \Phi (e, e') = \text{wp} \ e \ \{v. \ \exists v'. \ e' \to^* v' \ast \Phi(v, v')\} \]
\[ V_R[[\exists \vdash \text{STRef} \; X \; \tau]]_{\Delta}(v, v') \triangleq \exists \ell, z. v = \ell \ast v' = z \ast z \leadsto_{\Delta(X).1} \ell \ast \]

\[ \exists w, w'. \ell \mapsto w \ast z \mapsto_{\Delta(X).2} (w') \ast V_R[[\exists \vdash \tau]]_{\Delta}(w, w') \]
\[ \mathcal{V}_R[[\Xi \vdash \text{STRef } X \ \tau]]_{\Delta} (v, v') \triangleq \exists \ell, z. \ v = \ell \ast v' = z \ast z \mapsto_{\Delta(X).1} \ell \ast \]

\[ \exists w, w'. \ell \mapsto w \ast z \mapsto_{\Delta(X).2} (w') \ast \mathcal{V}_R[[\Xi \vdash \tau]]_{\Delta} (w, w') \]

\[ \mathcal{V}_R[[\Xi \vdash \text{ST } X \ \tau]]_{\Delta} (v, v') \triangleq \forall \vec{l}_i, \vec{v}_i. \ |\vec{l}_i| = |\vec{v}_i|. \square \left( \text{OwnLocs}_{\Delta(X).1} (\vec{l}_i) \ast \text{OwnState}_{\Delta(X).2} (\vec{v}_i) \rightarrow \ast \right) \]

\[ \text{wp runST } \{ v \} \left\{ w. \exists w', \vec{l}_f, \vec{v}_f. \ |\vec{l}_f| = |\vec{v}_f|. (v' \ E (\vec{v}_i) \rightarrow \ast (E (\vec{v}_f), w')) \ast \right\} \]

\[ \text{OwnLocs}_{\Delta(X).1} (\vec{l}_f) \ast \text{OwnState}_{\Delta(X).2} (\vec{v}_f) \ast \mathcal{V}_R[[\Xi \vdash \tau]]_{\Delta} (w, w') \right\} \]
\[ \text{lift}_R : (\text{Val} \rightarrow \text{Val} \rightarrow \text{iProp}) \rightarrow (\text{Expr} \rightarrow \text{Expr} \rightarrow \text{iProp}) \]

\[ \text{lift}_R \Phi (e, e') = \text{wp} \ e \left\{ v. \ \exists v'. \ e' \rightarrow^* v' \ * \Phi (v, v') \right\} \]
\[ \text{lift}_\mathcal{R} : (\text{Val} \rightarrow \text{Val} \rightarrow \text{iProp}) \rightarrow (\text{Expr} \rightarrow \text{Expr} \rightarrow \text{iProp}) \]

\[ \text{lift}_\mathcal{R} \Phi (e, e') = \text{wp } e \{ v. \exists v'. e' \rightarrow^* v' \ast \Phi (v, v') \} \]

\[ \Xi \mid \Gamma \vdash_\mathcal{R} e \leq e' : \tau \triangleq \]

\[ \forall \Delta, \vec{v}, \vec{v}' . \tilde{\mathcal{V}}_\mathcal{R} [[\Xi \vdash \Gamma]]_\Delta (\vec{v}, \vec{v}') \vdash \mathcal{E}_\mathcal{R} [[\Xi \vdash \tau]]_\Delta (e[\vec{x}/\vec{v}], e'[\vec{x}/\vec{v}']) \]
https://github.com/scaup/sem_backs_st
A Personal Retrospective using Iris
Small distance between *intuition* and *formalization*
Sometimes *existing abstractions* are not *sufficient*
Sometimes *existing abstractions* are not *sufficient*

\[
\text{wp } e \{ \Phi \}
\]
Sometimes *existing abstractions* are not *sufficient*.

\[ \text{wp } e \{ \Phi \} \]

expressions are of *finite depth*
Sometimes *existing abstractions* are not *sufficient*.

\[ \text{wp } e \{ \Phi \} \]

expressions are of *finite depth*.
Sometimes *existing abstractions* are not *sufficient*

\[
\text{wp } e \{ \Phi \}
\]

expressions are of *finite depth*

\[
e ::= \ldots | \triangleright (e)
\]

\[
\begin{align*}
\triangleright (w) & \rightarrow_h w \quad \text{if } w = (), b, z \\
\triangleright (\text{inj}_1 v) & \rightarrow_h \text{inj}_1 (\triangleright (v)) \\
\triangleright (\text{inj}_2 v) & \rightarrow_h \text{inj}_2 (\triangleright (v)) \\
\triangleright (\lambda x.e) & \rightarrow_h \lambda y.((\lambda x.\triangleright (e)) \triangleright (y)) \\
\triangleright (\text{fold } v) & \rightarrow_h \text{fold } (\triangleright (v))
\end{align*}
\]
To do at some point in the future

- polymorphism
- stronger, more intuitive properties
- formalize $wp$ to take advantage of finite expressions
Questions?
Extras

What is (isn't) the difficulty when adding polymorphism?
Stateful contexts can be emulated by pure, syntactically-typed contexts.

Stateful contexts can be emulated by pure, semantically-typed contexts.

Pure, semantically-typed contexts can be emulated by pure, syntactically-typed contexts.
Extras

Well definedness of back-translation from stateful language into the semantically typed language
Given $\vdash C : (\cdot \mid [\cdot \Gamma \cdot] ; [\cdot \tau \cdot]) \Rightarrow (\cdot \mid \cdot ; 1),$

**to prove** $\vdash_{int.} \llangle C \rrangle \leq \llangle C \rrangle : (\Gamma \tau) \Rightarrow (\cdot ; 1)$
Given \[ \vdash C : (\cdot | \llbracket \Gamma \rrbracket; \llbracket \tau \rrbracket) \Rightarrow (\cdot | \cdot ; 1) , \]

to prove \[ \vdash_{int.} \llbracket C \rrbracket \leq \llbracket C \rrbracket : (\Gamma; \tau) \Rightarrow (\cdot ; 1) \]

\[ \Gamma \vdash_{int.} e \leq e' : \tau \]
\[ \Xi | \Gamma \vdash_{\chi} e \leq e' : \tau \]
Given $\vdash C : (\cdot | \llbracket \Gamma \rrbracket ; \llbracket \tau \rrbracket ) \Rightarrow (\cdot | \cdot ; 1)$,

to prove $\vdash_{\text{int.}} \llbracket C \rrbracket \leq \llbracket C \rrbracket : (\Gamma ; \tau) \Rightarrow (\cdot ; 1)$

\[
\begin{align*}
\Gamma & \vdash_{\text{int.}} e \leq e' : \tau \\
\exists! & \mid \Gamma \vdash_{\chi} e \leq e' : \tau
\end{align*}
\]

\[
\begin{align*}
\Gamma & \vdash_{\text{int.}} e \leq e' : \tau \vdash \cdot \mid \llbracket \Gamma \rrbracket \vdash_{\chi} e \leq e' : \llbracket \tau \rrbracket
\end{align*}
\]
Given \( \vdash C : (\cdot | \llbracket\Gamma\rrbracket; \llbracket\tau\rrbracket) \Rightarrow (\cdot | \cdot; 1) \),

to prove \( \models_{\text{int}} \langle C \rangle \leq \langle C \rangle : (\Gamma; \tau) \Rightarrow (\cdot; 1) \)

\[ \Gamma \models_{\text{int}} e \leq e' : \tau \]
\[ \Xi \models \Gamma \models_{\text{c}} e \leq e' : \tau \]

\[ \Gamma \models_{\text{int}} e \leq e' : \tau \vdash \cdot \quad | \quad \llbracket\Gamma\rrbracket \models_{\text{c}} e \leq e' : \llbracket\tau\rrbracket \]

\[ \forall \Xi \models \Gamma \vdash e : \tau. \quad \Xi \models \Gamma \models_{\text{c}} \langle e \rangle \leq \langle e \rangle : \tau \]
\[ \text{OwnStates}_y(\vec{v}, \vec{v}') \vdash \Rightarrow \text{OwnStates}_y(\vec{v} + [v], \vec{v}' + [v']) \cdot |\vec{v}| \mapsto_y (v, v') \]

\[ \text{OwnStates}_y(\vec{v}, \vec{v}') \cdot z \mapsto_y (v, v') \vdash \vec{v}.z = v \cdot \vec{v}'.z = v' \]

\[ \text{OwnStates}_y(\vec{v}, \vec{v}') \cdot z \mapsto_y (v, v') \vdash \Rightarrow \text{OwnStates}_y(\vec{v}[z \mapsto w], \vec{v}'[z \mapsto w']) \cdot z \mapsto_y (w, w') \]
\[ \mathcal{V}_X [[\Xi \vdash \text{ST} \times \tau]]_\Delta (v, v') \triangleq \forall \vec{v}_i, \vec{v}'_i. \quad \square \left( \text{OwnStates}_{\Delta(X)}(\vec{v}_i, \vec{v}'_i) \ast \text{wp } v \mathcal{E}(\vec{v}_i) \left\{ (\mathcal{E}(\vec{v}_f), w). \exists \vec{v}'_f, w' \right\} \right) \]

\[ \mathcal{V}_X [[\Xi \vdash \text{ST} \times \tau]]_\Delta (v, v') \triangleq \exists z. \quad v = z \ast v' = z \ast \]

\[ \exists w, w'. \quad z \mapsto_{\Delta(X)} (w, w') \ast \mathcal{V}_X [[\Xi \vdash \tau]]_\Delta (w, w') \]

\[ \Xi \mid \Gamma \models_X e \leq e' : \tau \triangleq \forall \Delta, \vec{v}, \vec{v}'. \quad |\vec{v}| = |\vec{v}'| = |\Gamma| \ast \mathcal{V}_X [[\Xi \vdash \Gamma. i]]_\Delta (\vec{v}.i, \vec{v}'.i) \vdash 0 \leq i < |\Gamma| \]

\[ \mathcal{E}_X [[\Xi \vdash \tau]]_\Delta (e[\vec{x}/\vec{v}], e'[\vec{x}/\vec{v}']) \]
Extras

Existing \textit{wp} not sufficient
lift : (Val → Val → iProp) → (Expr → Expr → iProp)

\[ \text{lift } \Phi(e, e') = \text{wp } e \{ v. \exists v'. e' \rightarrow^* v' \rightarrow^* \Phi(v, v') \} \]
lift : (Val → Val → iProp) → (Expr → Expr → iProp)

lift Φ (e, e') = wp e \{ v. \exists v'. e' \rightarrow^* v' \ast Φ(v, v') \}

list \( Z \triangleq \mu X. 1+ (Z \times X) \)

f \triangleq \text{map} (\lambda x. x + 0)
lift : (Val → Val → iProp) → (Expr → Expr → iProp)

lift Φ (e, e') = wp e \{ v. \exists v'. e' →^* v' * Φ(v, v') \}

list Z ≜ \mu X.1+(Z \times X) \quad f ≜ \text{map} (\lambda x.x + 0)

∫ ν \text{int.}[\text{[[list Z→list Z]]} (\lambda x.x, f)
lift : (Val → Val → iProp) → (Expr → Expr → iProp)

lift \Phi (e, e') = \text{wp } e \{ v. \exists v'. e' \rightarrow^* v' \ast \Phi(v, v') \}

\text{list } Z \triangleq \mu X.1+ (Z \times X) \quad f \triangleq \text{map } (\lambda x.x + 0)

\not\models \mathcal{V}_\text{int.}[[[ \text{list } Z \rightarrow \text{list } Z]]] (\lambda x. x, f)

\forall v, v'. \mathcal{V}_\text{int.}[[[ \text{list } Z]]] (v, v') \not\models \mathcal{E}_\text{int.}[[[ \text{list } Z]]] ((\lambda x. x) v, f v')
lift : \((\text{Val} \rightarrow \text{Val} \rightarrow \text{iProp}) \rightarrow (\text{Expr} \rightarrow \text{Expr} \rightarrow \text{iProp})\)

\[
lift \Phi (e, e') = \text{wp } e \{ v \cdot \exists v'. e' \rightarrow^* v' \ast \Phi(v, v') \}\]

\[
\text{list } Z \triangleq \mu X.1+(Z \times X) \quad f \triangleq \text{map } (\lambda x.x + 0)
\]

\[
\not\in \mathcal{V}_{int.}[\text{list } Z \rightarrow \text{list } Z](\lambda x.x, f)
\]

\[
\forall v, v'. \mathcal{V}_{int.}[\text{list } Z](v, v') \not\in \mathcal{E}_{int.}[\text{list } Z](\lambda x.x \mapsto f, v, v')
\]

\[
(\lambda x.x) \ [1, 2, 3, 4] \quad f \ [1, 2, 3, \text{true}]
\]
Extras

Proving that emulations of semantically-typed into syntactically-typed is well behaved
Pure, semantically-typed contexts can be emulated by pure, syntactically-typed contexts

Given \( \vdash_{\text{int}} C : (\Gamma; \tau) \Rightarrow (\cdot; 1) \), there exists \( \vdash C^*_b : (\Gamma; \tau) \Rightarrow (\cdot; 1) \). This is a valid emulation, i.e. for all \( \Gamma \vdash e : \tau \) we have:

\[
C^*_b[e] \Downarrow \iff C[e] \Downarrow
\]
\( \mathcal{V}_? : \text{Val} \rightarrow \text{Val} \rightarrow \text{iProp} \)

\( \mathcal{V}_? (v, v') = (v = \text{injected}_{1}^{\text{Val}}() \ast v' = ()) \)

\( \forall (\exists b \in \{\text{true, false}\}. \ v = \text{injected}_{\text{B}}^{\text{Val}}(b) \ast v' = b) \)

\( \forall (\exists z \in \mathbb{Z}. \ v = \text{injected}_{\text{Z}}^{\text{Val}}(z) \ast v' = z) \)

\( \forall (\exists w, w'. \ \bigvee_{i \in \{1, 2\}} (v = \text{injected}_{+}^{\text{Val}}(\text{inj}_i w) \ast v' = \text{inj}_i w' \ast \mathcal{V}_? (w, w')) \)

\( \forall (\exists v_1, v'_1, v'_2, v'_2. \ v = \text{injected}_{\times}^{\text{Val}}((v'_1, v'_2)) \ast v' = (v'_1, v'_2) \ast \mathcal{V}_? (v_1, v'_1) \ast \mathcal{V}_? (v_2, v'_2)) \)

\( \forall (\exists e. \ v = \text{injected}_{\rightarrow}^{\text{Val}}(\lambda x. e) \ast \sqcap (\forall w, w'. \ \mathcal{V}_? (w, w') \ast \text{lift} \ \mathcal{V}_? (e[w/x], v' w')) \)

\( \forall (\exists w, w'. \ v = \text{injected}_{\mu}^{\text{Val}}(\text{fold } w) \ast v' = \text{fold } w' \ast \mathcal{V}_? (w, w')) \)

\[
\begin{array}{c}
\mathcal{E}_? (e, e') \\
\mathcal{E}_{\text{int.}} [[\tau]] (\text{project}_\tau e, e') \\
\mathcal{E}_{\text{int.}} [[\tau]] (e, e') \\
\mathcal{E}_? (\text{embed}_\tau e, e')
\end{array}
\]
Pure, semantically-typed contexts can be emulated by pure, syntactically-typed contexts.

Given $\vdash_{int} C : (\Gamma; \tau) \Rightarrow (\cdot; 1)$, there exists $\vdash C_b^\tau : (\Gamma; \tau) \Rightarrow (\cdot; 1)$.

This is a valid emulation, i.e. for all $\Gamma \vdash e : \tau$ we have:

$$C_b^\tau[e] \downarrow \iff C[e] \downarrow$$