Purity of an ST Monad

Full Abstraction by Semantically Type Back-Translation

Koen Jacobs (me)

Dominique Devriese

Amin Timany

1994

John & Simon add an ST monad to Haskell

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 update-in-place, if you please. In 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 [1993]; 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 it also allows input/output to be smoothly integrated with other state mainpulation.

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 seman-

```haskell
-- 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 ()
```

```haskell
-- 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 ()
```

```haskell
-- 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 ()
```

```haskell
-- 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 ()
```

```haskell
-- 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 ()
```

```haskell
-- 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 ()
```

```haskell
-- 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 ()
```

```haskell
-- 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 ()
```

```haskell
-- 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 ()
```

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 viability 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.

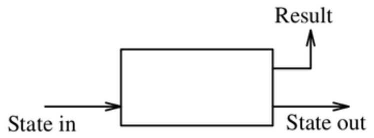
- Stateful computations can even be performed *lazily* without losing safety. For example, suppose that stateful depth-first search of a graph returns a list of vertices in depth-first order. If the consumer of this list only evaluates the first few elements of the list, then only enough of the stateful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

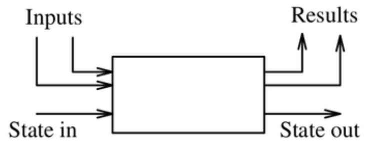


Notice that this is a purely-functional account of state. The "`ST`" stands for "a state transformer", which we take to be synonymous with "a stateful computation": the computation is seen as transforming one state into another. (Of course, it is our intention that the new state will actually be constructed 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 `Int`, and two results of type `Int` and `Bool`, would have the type:

```
Int -> Int -> ST s (Int,Bool)
```

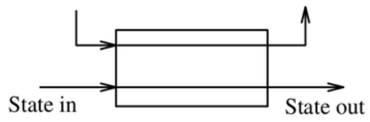
Its picture might look like this:



The simplest state transformer, **`returnST`**, simply delivers a value without affecting the state at all:

```
returnST :: a -> ST s a
```

The picture for `returnST` is like this:



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…*

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 viability 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.

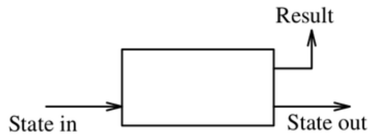
- Stateful computations can even be performed *lazily* without losing safety. For example, suppose that stateful depth-first search of a graph returns a list of vertices in depth-first order. If the consumer of this list only evaluates the first few elements of the list, then only enough of the stateful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

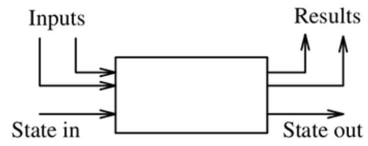


Notice that this is a purely-functional account of state. The "ST" stands for "a state transformer", which we take to be synonymous with "a stateful computation": the computation is seen as transforming one state into another. (Of course, it is our intention that the new state will actually be constructed 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 `Int`, and two results of type `Int` and `Bool`, would have the type:

```
Int -> Int -> ST s (Int,Bool)
```

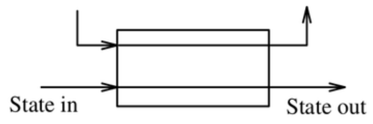
Its picture might look like this:



The simplest state transformer, `returnST`, simply delivers a value without affecting the state at all:

```
returnST :: a -> ST s a
```

The picture for `returnST` is like this:



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)

```haskell
-- 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)
```

```haskell
-- 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)
```

```haskell
-- 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)
```

```haskell
-- 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 ()

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)
```

```haskell
-- 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)
```

*“ So all the usual techniques for **reasoning about** functional **programs** continue to work…*

```haskell
blee :: ST n Int
blee = ...

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

blaa :: Int -> Int
blaa = ...

subprogram :: (Int -> Int) -> Bool ->
          ((String -> String) -> String) ->
    List String
subprogram = ...

foo :: ST n (Int -> ST n Bool)
foo = ...
```

*“ So all the usual techniques for **reasoning about** functional **programs** continue to work…*

```
blee :: ST n Int
blee = ...

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

blaa :: Int -> Int
blaa = ...

subprogram :: (Int -> Int) -> Bool ->
        ((String -> String) -> String) ->
    List String
subprogram = ...

foo :: ST n (Int -> ST n Bool)
foo = ...
```

$$\textit{Given two programs, } e_1 \textit{ and } e_2\textit{, they are contextually equivalent, } \; e_1 \approx_{ctx} e_2$$

$$\textit{if}$$

$$\forall C.\, C[e_1] \textit{ behaves the same as } C[e_2]$$

Any two pure programs, e_1 and e_2,
contextually equivalent in the pure language,
should be contextually equivalent in the extended stateful language.

“ *No **pure** 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.

*“ No **pure** 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.

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

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

“ *No* ***pure*** *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.

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

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

“ *No* ***pure*** *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.

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

Note: full abstraction is 1) preservation of ctx. equiv. and 2) reflection of ctx. equiv.

```
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 ...
```

```
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 ...
```

```haskell
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 ...
```

Adding ST to *Haskell* ⟶ Adding ST to *STLC*$_\mu$

Adding ST to *Haskell* $\longrightarrow$ Adding ST to $STLC_\mu$

- No polymorphism -

- Call by Value -

$STLC_\mu$ $\lambda^{\vdash}$

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

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

STLCμ $\lambda^{\vdash}$

$$\tau ::= 1 \mid Z \mid B \mid \tau{\times}\tau \mid \tau{+}\tau \mid X \mid \mu X.\tau \mid \tau{\rightarrow}\tau$$

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

Extension with ST $\lambda^{\vdash}_{ST}$

*STLC*μ $\lambda^{\vdash}$

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

$$\Gamma \vdash \mathrm{e} : \tau$$

Extension with ST $\lambda^{\vdash}_{\mathrm{ST}}$

$$\boxed{\dfrac{\ell \notin \mathrm{dom}(h)}{\langle h, \mathbf{ref}\ \mathrm{v}\rangle \rightsquigarrow_h \langle h \uplus \{\ell \mapsto \mathrm{v}\}, \mathbf{return}\ \ell\rangle}}$$

$$\boxed{\langle h \uplus \{\ell \mapsto \mathrm{v}\}, !\ell\rangle \rightsquigarrow_h \langle h \uplus \{\ell \mapsto \mathrm{v}\}, \mathbf{return}\ \mathrm{v}\rangle}$$

$$\boxed{\langle h \uplus \{\ell \mapsto \mathrm{v}'\}, \ell \leftarrow \mathrm{v}\rangle \rightsquigarrow_h \langle h \uplus \{\ell \mapsto \mathrm{v}\}, \mathbf{return}\ ()\rangle}$$

STLC$_\mu$ $\lambda^{\vdash}$

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

$$\Gamma \vdash \mathrm{e} : \tau$$

Extension with ST $\lambda^{\vdash}_{\mathrm{ST}}$

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

$$\Xi \mid \Gamma \vdash \mathrm{e} : \tau$$

$$\boxed{\dfrac{\ell \notin \mathrm{dom}(h)}{\langle h, \mathbf{ref}\ \mathrm{v}\rangle \rightsquigarrow_h \langle h \uplus \{\ell \mapsto \mathrm{v}\}, \mathbf{return}\ \ell\rangle}}$$

$$\boxed{\langle h \uplus \{\ell \mapsto \mathrm{v}\}, !\ell\rangle \rightsquigarrow_h \langle h \uplus \{\ell \mapsto \mathrm{v}\}, \mathbf{return}\ \mathrm{v}\rangle}$$

$$\boxed{\langle h \uplus \{\ell \mapsto \mathrm{v}'\}, \ell \leftarrow \mathrm{v}\rangle \rightsquigarrow_h \langle h \uplus \{\ell \mapsto \mathrm{v}\}, \mathbf{return}\ ()\rangle}$$

STLC$_\mu$ $\lambda^{\vdash}$

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

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

Extension with ST $\lambda^{\vdash}_{\mathrm{ST}}$

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

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

$$\boxed{\frac{\ell \notin \mathrm{dom}(h)}{\langle h, \mathbf{ref}\ v\rangle \rightsquigarrow_h \langle h \uplus \{\ell \mapsto v\}, \mathbf{return}\ \ell\rangle}}$$

$$\boxed{\langle h \uplus \{\ell \mapsto v\}, !\ell\rangle \rightsquigarrow_h \langle h \uplus \{\ell \mapsto v\}, \mathbf{return}\ v\rangle}$$

$$\boxed{\langle h \uplus \{\ell \mapsto v'\}, \ell \leftarrow v\rangle \rightsquigarrow_h \langle h \uplus \{\ell \mapsto v\}, \mathbf{return}\ ()\rangle}$$

$$\boxed{\frac{\Xi, X \mid \Gamma \vdash e : \mathrm{ST}\ X\ \tau \qquad \Xi \vdash \tau}{\Xi \mid \Gamma \vdash \mathbf{runST}\ \{e\} : \tau}}$$

$$\boxed{\frac{\Xi \mid \Gamma \vdash e : \tau \qquad \Xi \vdash X}{\Xi \mid \Gamma \vdash \mathbf{ref}\ e : \mathrm{ST}\ X\ (\mathrm{STRef}\ X\ \tau)}}$$

$$\boxed{\frac{\Xi \mid \Gamma \vdash e : \mathrm{STRef}\ X\ \tau \qquad \Xi \mid \Gamma \vdash e' : \tau}{\Xi \mid \Gamma \vdash e \leftarrow e' : \mathrm{ST}\ X\ 1}}$$

STLC$_\mu$ $\lambda^{\vdash}$

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

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

Extension with ST $\lambda^{\vdash}_{ST}$

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

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

$$\boxed{\dfrac{\ell \notin \mathrm{dom}(h)}{\langle h, \mathbf{ref}\ v\rangle \rightsquigarrow_h \langle h \uplus \{\ell \mapsto v\}, \mathbf{return}\ \ell\rangle}}$$

$$\boxed{\langle h \uplus \{\ell \mapsto v\}, !\ell\rangle \rightsquigarrow_h \langle h \uplus \{\ell \mapsto v\}, \mathbf{return}\ v\rangle}$$

$$\boxed{\langle h \uplus \{\ell \mapsto v'\}, \ell \leftarrow v\rangle \rightsquigarrow_h \langle h \uplus \{\ell \mapsto v\}, \mathbf{return}\ ()\rangle}$$

$$\boxed{\dfrac{\Xi, X \mid \Gamma \vdash e : \mathrm{ST}\ X\ \tau \qquad \Xi \vdash \tau}{\Xi \mid \Gamma \vdash \mathbf{runST}\ \{e\} : \tau}}$$

$$\boxed{\dfrac{\Xi \mid \Gamma \vdash e : \tau \qquad \Xi \vdash X}{\Xi \mid \Gamma \vdash \mathbf{ref}\ e : \mathrm{ST}\ X\ (\mathrm{STRef}\ X\ \tau)}}$$

$$\boxed{\dfrac{\Xi \mid \Gamma \vdash e : \mathrm{STRef}\ X\ \tau \qquad \Xi \mid \Gamma \vdash e' : \tau}{\Xi \mid \Gamma \vdash e \leftarrow e' : \mathrm{ST}\ X\ 1}}$$

STLCμ $\lambda^{\vdash}$

$$\tau ::= 1 \mid \mathsf{Z} \mid \mathsf{B} \mid \tau\times\tau \mid \tau+\tau \mid \mathsf{X} \mid \mu\mathsf{X}.\tau \mid \tau\rightarrow\tau$$

$$\Gamma \vdash \mathrm{e} : \tau$$

Extension with ST $\lambda^{\vdash}_{\mathrm{ST}}$

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

$$\Xi \mid \Gamma \vdash \mathrm{e} : \tau$$

$$\boxed{\dfrac{\ell \notin \mathrm{dom}(h)}{\langle h, \mathrm{ref}\ \mathrm{v}\rangle \rightsquigarrow_h \langle h \uplus \{\ell \mapsto \mathrm{v}\}, \mathrm{return}\ \ell\rangle}}$$

$$\boxed{\langle h \uplus \{\ell \mapsto \mathrm{v}\}, !\ell\rangle \rightsquigarrow_h \langle h \uplus \{\ell \mapsto \mathrm{v}\}, \mathrm{return}\ \mathrm{v}\rangle}$$

$$\boxed{\langle h \uplus \{\ell \mapsto \mathrm{v}'\}, \ell \leftarrow \mathrm{v}\rangle \rightsquigarrow_h \langle h \uplus \{\ell \mapsto \mathrm{v}\}, \mathrm{return}\ ()\rangle}$$

$$\boxed{\dfrac{\Xi, \mathrm{X} \mid \Gamma \vdash \mathrm{e} : \mathrm{ST}\ \mathrm{X}\ \tau \qquad \Xi \vdash \tau}{\Xi \mid \Gamma \vdash \mathrm{runST}\ \{\mathrm{e}\} : \tau}}$$

$$\boxed{\dfrac{\Xi \mid \Gamma \vdash \mathrm{e} : \tau \qquad \Xi \vdash \mathrm{X}}{\Xi \mid \Gamma \vdash \mathrm{ref}\ \mathrm{e} : \mathrm{ST}\ \mathrm{X}\ (\mathrm{STRef}\ \mathrm{X}\ \tau)}}$$

$$\boxed{\dfrac{\Xi \mid \Gamma \vdash \mathrm{e} : \mathrm{STRef}\ \mathrm{X}\ \tau \qquad \Xi \mid \Gamma \vdash \mathrm{e}' : \tau}{\Xi \mid \Gamma \vdash \mathrm{e} \leftarrow \mathrm{e}' : \mathrm{ST}\ \mathrm{X}\ 1}}$$

$$\llbracket \_ \rrbracket : \lambda^{\vdash} \longrightarrow \lambda^{\vdash}_{\mathrm{ST}}$$

$$\textit{If}\ \ \Gamma \vdash \mathrm{e}_1 \approx_{ctx} \mathrm{e}_2 : \tau,\ \textit{then}\ \ \cdot \mid \llbracket \Gamma \rrbracket \vdash \llbracket \mathrm{e}_1 \rrbracket \approx_{ctx} \llbracket \mathrm{e}_2 \rrbracket : \llbracket \tau \rrbracket$$

$$[\![\_]\!] : \lambda^{\vdash} \longrightarrow \lambda^{\vdash}_{ST}$$

$$\textit{If } \Gamma \vdash e_1 \approx_{ctx} e_2 : \tau, \textit{ then } \cdot \mid [\![\Gamma]\!] \vdash [\![e_1]\!] \approx_{ctx} [\![e_2]\!] : [\![\tau]\!]$$

$$\textit{If } \boxed{\begin{array}{c} \forall \vdash C : (\Gamma; \tau) \Rightarrow (\cdot\, ; 1) \\ C[e_1] \Downarrow \textit{ iff } C[e_2] \Downarrow \end{array}}, \textit{ then } \boxed{\begin{array}{c} \forall \vdash C : (\cdot \mid [\![\Gamma]\!]; [\![\tau]\!]) \Rightarrow (\cdot \mid \cdot\, ; 1) \\ C[[\![e_1]\!]] \Downarrow \textit{ iff } C[[\![e_2]\!]] \Downarrow \end{array}}$$

$$\llbracket\_\rrbracket : \lambda^{\vdash} \rightarrow \lambda^{\vdash}_{\mathrm{ST}}$$

$$\textit{If } \Gamma \vdash e_1 \approx_{ctx} e_2 : \tau, \textit{ then } \cdot \mid \llbracket\Gamma\rrbracket \vdash \llbracket e_1\rrbracket \approx_{ctx} \llbracket e_2\rrbracket : \llbracket\tau\rrbracket$$

$$\textit{If } \boxed{\begin{array}{c}\forall \vdash C : (\Gamma; \tau) \Rightarrow (\cdot\,; 1) \\ C[e_1] \Downarrow \textit{ iff } C[e_2] \Downarrow\end{array}}, \textit{ then } \boxed{\begin{array}{c}\forall \vdash C : (\cdot \mid \llbracket\Gamma\rrbracket; \llbracket\tau\rrbracket) \Rightarrow (\cdot \mid \cdot\,; 1) \\ C[\llbracket e_1\rrbracket] \Downarrow \textit{ iff } C[\llbracket e_2\rrbracket] \Downarrow\end{array}}$$

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

$$C^{\vdash}_b[e] \Downarrow \textit{ iff } C[\llbracket e\rrbracket] \Downarrow$$

$$\llbracket \_ \rrbracket : \lambda^{\vdash} \to \lambda^{\vdash}_{ST}$$

$$\textit{If } \Gamma \vdash e_1 \approx_{ctx} e_2 : \tau, \textit{ then } \cdot \mid \llbracket \Gamma \rrbracket \vdash \llbracket e_1 \rrbracket \approx_{ctx} \llbracket e_2 \rrbracket : \llbracket \tau \rrbracket$$

$$\textit{If } \boxed{\begin{array}{c} \forall \vdash C : (\Gamma; \tau) \Rightarrow (\cdot\, ; 1) \\ C[e_1] \Downarrow \textit{ iff } C[e_2] \Downarrow \end{array}}\textit{, then } \boxed{\begin{array}{c} \forall \vdash C : (\,\cdot \mid \llbracket \Gamma \rrbracket; \llbracket \tau \rrbracket) \Rightarrow (\,\cdot \mid \cdot\, ; 1) \\ C[\llbracket e_1 \rrbracket] \Downarrow \textit{ iff } C[\llbracket e_2 \rrbracket] \Downarrow \end{array}}$$

“ ***Statefulness*** *can be* ***purely*** *emulated*

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

$$C^{\vdash}_b[e] \Downarrow \textit{ iff } C[\llbracket e \rrbracket] \Downarrow$$

$$\llbracket \_ \rrbracket : \lambda^{\vdash} \to \lambda^{\vdash}_{ST}$$

$$\textit{If } \Gamma \vdash e_1 \approx_{ctx} e_2 : \tau, \textit{ then } \cdot \mid \llbracket \Gamma \rrbracket \vdash \llbracket e_1 \rrbracket \approx_{ctx} \llbracket e_2 \rrbracket : \llbracket \tau \rrbracket$$

$$\textit{If } \boxed{\begin{array}{c}\forall \vdash C : (\Gamma; \tau) \Rightarrow (\cdot\,; 1) \\ C[e_1] \Downarrow \textit{ iff } C[e_2] \Downarrow\end{array}}, \textit{ then } \boxed{\begin{array}{c}\forall \vdash C : (\cdot \mid \llbracket \Gamma \rrbracket; \llbracket \tau \rrbracket) \Rightarrow (\cdot \mid \cdot\,; 1) \\ C[\llbracket e_1 \rrbracket] \Downarrow \textit{ iff } C[\llbracket e_2 \rrbracket] \Downarrow\end{array}}$$

" ***Statefulness*** *can be* ***purely*** *emulated*

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

$$C^{\vdash}_{b}[e] \Downarrow \textit{ iff } C[\llbracket e \rrbracket] \Downarrow$$

“ ***Statefulness*** *can be* ***purely*** *emulated*

“ **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…

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 update-in-place, if you

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 [1993]; 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 it also allows input/output to be smoothly integrated with other state mainpulation.

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

“ **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...*

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

the world state.

We use the term “stateful” to describe computations or algorithms in which the programmer really does want to
) state. What has been lacking
ay of describing such algorithms
e — especially a non-strict one —
the main virtues of functional lan-
f order of evaluation (the Church-
ential transparency, non-strict se-

ibe a way to express stateful al-
purely-functional languages. The
ent of our earlier work on monadic
lation (Launchbury [1993]; Pey-
93]), but with an important tech-
use parametric polymorphism to
ion of state. It turns out that this
to be named without losing safety,
/output to be smoothly integrated
ulation.

eature of this paper is that it de-
em, and one that is implemented
compiler and freely available. The
g properties:

al transparency is maintained. At
what this statement means: how
putation be said to be referentially
e more precise, a stateful compu-
nsformer, that is, a function from
a final state. It is like a “script”,
ns to be performed on its input
ner function, it is quite possible to
eful computation to more than one

rmer is a pure function. But, be-
e that the state is used in a single-
final state can be constructed by

Monads for functional programming

Philip Wadler, University of Glasgow⋆

Department of Computing Science, University of Glasgow, G12 8QQ, Scotland
(wadler@dcs.glasgow.ac.uk)

Abstract. The use of monads to structure functional programs is described. Monads provide a convenient framework for simulating effects found in other languages, such as global state, exception handling, output, or non-determinism. Three case studies are looked at in detail: how monads ease the modification of a simple evaluator; how monads act as the basis of a datatype of arrays subject to in-place update; and how monads can be used to build parsers.

1 Introduction

Shall I be pure or impure?

The functional programming community divides into two camps. *Pure* languages, such as Miranda[0] and Haskell, are lambda calculus pure and simple. *Impure* languages, such as Scheme and Standard ML, augment lambda calculus with a number of possible *effects*, such as assignment, exceptions, or continuations. Pure languages are easier to reason about and may benefit from lazy evaluation, while impure languages offer efficiency benefits and sometimes make possible a more compact mode of expression.

Recent advances in theoretical computing science, notably in the areas of type theory and category theory, have suggested new approaches that may integrate the benefits of the pure and impure schools. These notes describe one, the use of *monads* to integrate impure effects into pure functional languages.

The concept of a monad, which arises from category theory, has been applied by Moggi to structure the denotational semantics of programming languages [13, 14]. The same technique can be applied to structure functional programs [21

“In the state monad, a computation accepts an initial state and returns a value paired with the final 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...

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

the world state.

We use the term "stateful" to describe computations or algorithms in which the programmer really does want to

Monads for functional programming

2.8 Variation two, revisited: State

In the state monad, a computation accepts an initial state and returns a value paired with the final state.

$$
\begin{array}{lll}
\textbf{type}\ M\ a & = & State \to (a, State) \\
\textbf{type}\ State & = & Int \\
unit & :: & a \to M\ a \\
unit\ a & = & \lambda x.\,(a, x) \\
(\star) & :: & M\ a \to (a \to M\ b) \to M\ b \\
m \star k & = & \lambda x.\,\textbf{let}\ (a, y) = m\ x\ \textbf{in} \\
 & & \quad \textbf{let}\ (b, z) = k\ a\ y\ \textbf{in} \\
 & & \quad (b, z) \\
tick & :: & M\,() \\
tick & = & \lambda x.\,((), x + 1)
\end{array}
$$

The call *unit a* returns the computation that accept initial state x and returns value a and final state x; that is, it returns a and leaves the state unchanged. The call $m \star k$ performs computation m in the initial state x, yielding value a and intermediate state y; then performs computation $k\ a$ in state y, yielding value b and final state z. The call *tick* increments the state, and returns the empty value (), whose type is also written ().

In an impure language, an operation like *tick* would be represented by a function of type () → (). The spurious argument () is required to delay the effect until the function is applied, and since the output type is () one may guess that the function's purpose lies in a side effect. In contrast, here *tick* has type M (): no spurious argument is needed, and the appearance of M explicitly indicates

ST computations by State Monad?

$$\langle\!\langle \mathbf{ST\ X\ B} \rangle\!\rangle \stackrel{?}{=} \mathsf{H} \longrightarrow \mathsf{B} \times \mathsf{H}$$

ST computations by State Monad?

$$\langle\!\langle \mathsf{ST}\ \mathsf{X}\ \mathsf{B} \rangle\!\rangle \stackrel{?}{=} \mathsf{H} \longrightarrow \mathsf{B} \times \mathsf{H}$$

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

$$\mathsf{C}_b^{\vdash}[\mathsf{e}] \Downarrow \ \textit{iff}\ \ \mathsf{C}[\llbracket\mathsf{e}\rrbracket] \Downarrow$$

ST computations by State Monad?

$$\langle\!\langle \mathrm{ST\ X\ B} \rangle\!\rangle \stackrel{?}{=} \mathrm{H} \longrightarrow \mathrm{B} \times \mathrm{H}$$

Given $\vdash \mathrm{C} : (\,\cdot \mid \lceil\!\lceil \Gamma \rceil\!\rceil ; \lceil\!\lceil \tau \rceil\!\rceil) \Rightarrow (\,\cdot \mid \cdot\,; 1)$, *there exists a context* $\vdash \mathrm{C}^{\vdash}_{b} : (\Gamma ; \tau) \Rightarrow (\,\cdot\,; 1)$. *This is a valid emulation, i.e. for all* $\Gamma \vdash \mathrm{e} : \tau$ *we have:*

$$\mathrm{C}^{\vdash}_{b}[\mathrm{e}] \Downarrow \ \textit{iff}\ \mathrm{C}[\lceil\!\lceil \mathrm{e} \rceil\!\rceil] \Downarrow$$

ST computations by State Monad?

$$\langle\!\langle \mathrm{ST\ X\ B} \rangle\!\rangle \stackrel{?}{=} \mathrm{H} \longrightarrow \mathrm{B} \times \mathrm{H}$$

Given $\vdash \mathrm{C} : (\,\cdot \mid \lceil\!\lceil \Gamma \rceil\!\rceil ; \lceil\!\lceil \tau \rceil\!\rceil) \Rightarrow (\,\cdot \mid \cdot\,; 1)$, *there exists a context* $\vdash \mathrm{C}_b^{\vdash} : (\Gamma; \tau) \Rightarrow (\,\cdot\,; 1)$. *This is a valid emulation, i.e. for all* $\Gamma \vdash \mathrm{e} : \tau$ *we have:*

$$\mathrm{C}_b^{\vdash}[\mathrm{e}] \Downarrow \ \textit{iff}\ \ \mathrm{C}[\lceil\!\lceil \mathrm{e} \rceil\!\rceil] \Downarrow$$

ST computations by *Untyped* State Monad

$$\langle\!\langle \mathbf{ST\ X\ B} \rangle\!\rangle = \mathit{UntypedStore} \longrightarrow \mathit{UntypedStore} \times \mathrm{B}$$

ST computations by *Untyped* State Monad

$$\langle\!\langle \mathrm{ST}\ \mathrm{X}\ \mathrm{B} \rangle\!\rangle = \mathit{UntypedStore} \longrightarrow \mathit{UntypedStore} \times \mathrm{B}$$

$$\mathcal{E} : \mathit{List}\ \mathsf{Val} \rightarrow \mathsf{Val}$$
$$\mathcal{E}([\mathsf{v}_1;\ \mathsf{v}_2;\ \mathsf{v}_3]) = ((\mathsf{v}_3, (\mathsf{v}_2, (\mathsf{v}_1, ()))), 3)$$

ST computations by *Untyped* State Monad

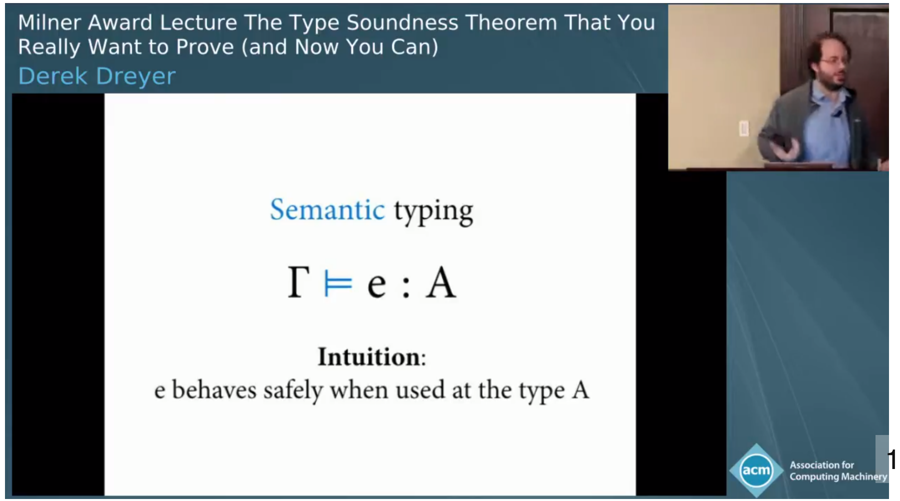
$$\langle\!\langle \mathrm{ST\ X\ B} \rangle\!\rangle = \mathit{UntypedStore} \longrightarrow \mathit{UntypedStore} \times \mathrm{B}$$

$$\mathcal{E} : \mathit{List}\ \mathrm{Val} \longrightarrow \mathrm{Val}$$
$$\mathcal{E}([\mathrm{v}_1;\ \mathrm{v}_2;\ \mathrm{v}_3]) = ((\mathrm{v}_3, (\mathrm{v}_2, (\mathrm{v}_1, ()))), 3)$$

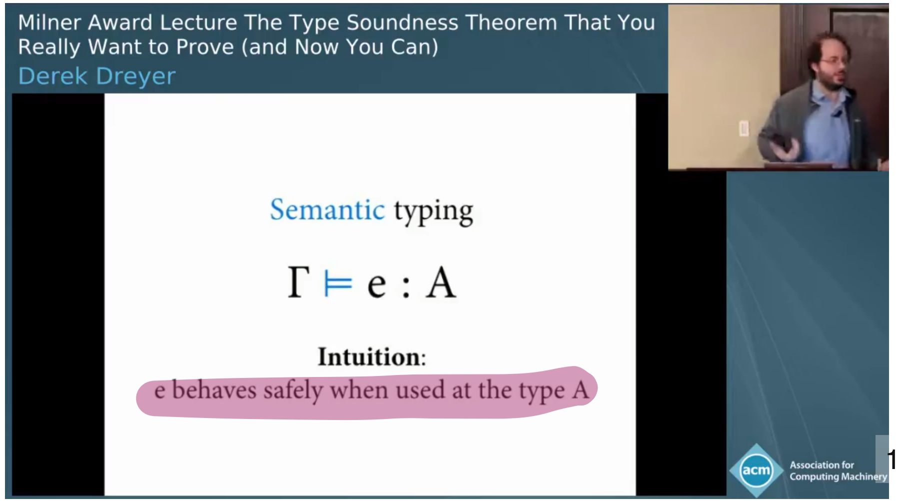
$$\begin{aligned}
\mathrm{read\ z}\ \mathcal{E}(\vec{\mathrm{v}}) &\longrightarrow^* (\mathcal{E}(\vec{\mathrm{v}}), \vec{\mathrm{v}}.\mathrm{z}) && \mathit{if}\ 0 \leq \mathrm{z} < |\vec{\mathrm{v}}| \\
\mathrm{ref\ v}\ \mathcal{E}(\vec{\mathrm{v}}) &\longrightarrow^* (\mathcal{E}(\vec{\mathrm{v}} \mathbin{+\!\!+} [\mathrm{v}]), |\vec{\mathrm{v}}|) && \\
\mathrm{write\ z\ v}\ \mathcal{E}(\vec{\mathrm{v}}) &\longrightarrow^* (\mathcal{E}(\vec{\mathrm{v}}[\mathrm{z} \mapsto \mathrm{v}]), ()) && \mathit{if}\ 0 \leq \mathrm{z} < |\vec{\mathrm{v}}|
\end{aligned}$$

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 $\vdash \mathsf{C} : (\,\cdot \mid \llbracket\Gamma\rrbracket; \llbracket\tau\rrbracket) \Rightarrow (\,\cdot \mid \cdot\,; 1)$*, there exists a context* $\vdash \mathsf{C}^{\vdash}_{b} : (\Gamma; \tau) \Rightarrow (\,\cdot\,; 1)$.
This is a valid emulation, i.e. for all $\Gamma \vdash \mathsf{e} : \tau$ *we have:*

$$\mathsf{C}^{\vdash}_{b}[\mathsf{e}] \Downarrow \;\; \textit{iff} \;\; \mathsf{C}[\llbracket\mathsf{e}\rrbracket] \Downarrow$$

“ ***Stateful*** *contexts can be emulated by* ***pure, syntactically-typed*** *contexts*

Given $\vdash \mathrm{C} : (\ \cdot \mid \lceil\!\lceil \Gamma \rceil\!\rceil ; \lceil\!\lceil \tau \rceil\!\rceil) \Rightarrow (\ \cdot \mid \cdot\ ; 1)$, *there exists a context* $\vdash \mathrm{C}_b^{\vdash} : (\Gamma; \tau) \Rightarrow (\ \cdot\ ; 1)$.
This is a valid emulation, i.e. for all $\Gamma \vdash \mathrm{e} : \tau$ *we have:*

$$\mathrm{C}_b^{\vdash}[\mathrm{e}] \Downarrow \ \textit{iff}\ \mathrm{C}[\lceil\!\lceil \mathrm{e} \rceil\!\rceil] \Downarrow$$

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

Given $\vdash \mathrm{C} : (\cdot \mid \lceil\!\lceil \Gamma \rceil\!\rceil ; \lceil\!\lceil \tau \rceil\!\rceil) \Rightarrow (\cdot \mid \cdot\ ; 1)$, *there exists* $\vDash_{int.} \mathrm{C}_b^{\vDash} : (\Gamma; \tau) \Rightarrow (\cdot\ ; 1)$.
This is a valid emulation, i.e. for all $\Gamma \vdash \mathrm{e} : \tau$*:*

$$\mathrm{C}_b^{\vDash}[\mathrm{e}] \Downarrow \ \textit{iff}\ \mathrm{C}[\lceil\!\lceil \mathrm{e} \rceil\!\rceil] \Downarrow$$

“ ***Stateful** contexts can be emulated by **pure, syntactically-typed** contexts*

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

$$\mathsf{C}^{\vdash}_{b}[\mathsf{e}] \Downarrow \;\; \textit{iff} \;\; \mathsf{C}[\llbracket\mathsf{e}\rrbracket] \Downarrow$$

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

Given $\vdash \mathsf{C} : (\cdot \mid \llbracket\Gamma\rrbracket; \llbracket\tau\rrbracket) \Rightarrow (\cdot \mid \cdot\,; 1)$, *there exists* $\vDash_{int.} \mathsf{C}^{\vDash}_{b} : (\Gamma; \tau) \Rightarrow (\cdot\,; 1)$.
This is a valid emulation, i.e. for all $\Gamma \vdash \mathsf{e} : \tau$*:*

$$\mathsf{C}^{\vDash}_{b}[\mathsf{e}] \Downarrow \;\; \textit{iff} \;\; \mathsf{C}[\llbracket\mathsf{e}\rrbracket] \Downarrow$$

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

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

$$\mathsf{C}^{\vdash}_{b}[\mathsf{e}] \Downarrow \;\; \textit{iff} \;\; \mathsf{C}[\mathsf{e}] \Downarrow$$

“ ***Stateful** contexts can be emulated by **pure, syntactically-typed** contexts*

Given $\vdash \mathsf{C} : (\cdot \mid [\![\Gamma]\!] ; [\![\tau]\!]) \Rightarrow (\cdot \mid \cdot ; 1)$, *there exists a context* $\vdash \mathsf{C}_b^{\vdash} : (\Gamma ; \tau) \Rightarrow (\cdot ; 1)$.
This is a valid emulation, i.e. for all $\Gamma \vdash \mathsf{e} : \tau$ *we have:*

$$\mathsf{C}_b^{\vdash}[\mathsf{e}] \Downarrow \text{ iff } \mathsf{C}[[\![\mathsf{e}]\!]] \Downarrow$$

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

Given $\vdash \mathsf{C} : (\cdot \mid [\![\Gamma]\!] ; [\![\tau]\!]) \Rightarrow (\cdot \mid \cdot ; 1)$, *there exists* $\vDash_{int.} \mathsf{C}_b^{\vDash} : (\Gamma ; \tau) \Rightarrow (\cdot ; 1)$.
This is a valid emulation, i.e. for all $\Gamma \vdash \mathsf{e} : \tau$*:*

$$\mathsf{C}_b^{\vDash}[\mathsf{e}] \Downarrow \text{ iff } \mathsf{C}[[\![\mathsf{e}]\!]] \Downarrow$$

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

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

$$\mathsf{C}_b^{\vdash}[\mathsf{e}] \Downarrow \text{ iff } \mathsf{C}[\mathsf{e}] \Downarrow$$

Defining Semantic Typedness

$$\Gamma \vDash_{int.} \mathrm{e} : \tau$$

Iris

Defining Semantic Typedness

$$\Gamma \vDash_{int.} e \leq e : \tau$$

Logical Relations on Values

$$\mathcal{V}_{int.}[\![1]\!](\mathsf{v},\mathsf{v}') \triangleq \mathsf{v} = () * \mathsf{v}' = ()$$
$$\mathcal{V}_{int.}[\![\mathsf{B}]\!](\mathsf{v},\mathsf{v}') \triangleq \exists b \in \{\mathit{true}, \mathit{false}\}.\ \mathsf{v} = b * \mathsf{v}' = b$$
$$\mathcal{V}_{int.}[\![\mathsf{Z}]\!](\mathsf{v},\mathsf{v}') \triangleq \exists z \in \mathbb{Z}.\ \mathsf{v} = z * \mathsf{v}' = z$$
$$\mathcal{V}_{int.}[\![\tau_1 + \tau_2]\!](\mathsf{v},\mathsf{v}') \triangleq \bigvee_{i \in \{1,2\}} \exists \mathsf{w}, \mathsf{w}'.\ \mathsf{v} = \mathsf{inj}_i\ \mathsf{w} * \mathsf{v}' = \mathsf{inj}_i\ \mathsf{w}' * \mathcal{V}_{int.}[\![\tau_i]\!](\mathsf{w},\mathsf{w}')$$
$$\begin{aligned}\mathcal{V}_{int.}[\![\tau_1 \times \tau_2]\!](\mathsf{v},\mathsf{v}') \triangleq{}& \exists \mathsf{v}_1, \mathsf{v}_1', \mathsf{v}_2, \mathsf{v}_2'.\ \mathsf{v} = (\mathsf{v}_1,\mathsf{v}_2) * \mathsf{v}' = (\mathsf{v}_1',\mathsf{v}_2') \\ & * \mathcal{V}_{int.}[\![\tau_1]\!](\mathsf{v}_1,\mathsf{v}_1') * \mathcal{V}_{int.}[\![\tau_2]\!](\mathsf{v}_2,\mathsf{v}_2')\end{aligned}$$
$$\mathcal{V}_{int.}[\![\tau_1 \to \tau_2]\!](\mathsf{v},\mathsf{v}') \triangleq \Box\left(\forall \mathsf{w}, \mathsf{w}'.\ \mathcal{V}_{int.}[\![\tau_1]\!](\mathsf{w},\mathsf{w}') \twoheadrightarrow \mathsf{lift}\ \mathcal{V}_{int.}[\![\tau_2]\!]\ (\mathsf{v}\ \mathsf{w}, \mathsf{v}'\ \mathsf{w}')\right)$$
$$\mathcal{V}_{int.}[\![\mu X.\tau]\!](\mathsf{v},\mathsf{v}') \triangleq \exists \mathsf{w}, \mathsf{w}'.\ \mathsf{v} = \mathsf{fold}\ \mathsf{w} * \mathsf{v}' = \mathsf{fold}\ \mathsf{w}' * \triangleright \mathcal{V}_{int.}[\![\tau[\mu X.\tau/X]]\!](\mathsf{w},\mathsf{w}')$$

Logical Relations on Closed Expressions

$$\text{lift} : (\text{Val} \longrightarrow \text{Val} \longrightarrow iProp) \longrightarrow (\text{Expr} \longrightarrow \text{Expr} \longrightarrow iProp)$$

$$\text{lift}\ \Phi\ (e, e') = \text{wp}\ e\ \left\{v.\ \exists v'.\ e' \longrightarrow^* v' * \Phi(v, v')\right\}$$

$$\mathcal{E}_{int.}[[\tau]] = \text{lift}\ \mathcal{V}_{int.}[[\tau]]$$

Logical Relations on Open Expressions

$$\Gamma \vDash_{int.} e \leq e' : \tau \triangleq$$
$$\forall \vec{v}, \vec{v}'. \vec{\mathcal{V}}_{int.}[\![\Gamma]\!](\vec{v}, \vec{v}') \vdash \mathcal{E}_{int.}[\![\tau]\!](e[\vec{v}'/\vec{x}], e'[\vec{v}'/\vec{x}'])$$

Adequacy/Fundamental Theorem

LEMMA 2.1 (LOGICAL RELATION ADEQUACY). *If* $\cdot \vDash_{int.} \mathrm{e} \leq \mathrm{e}' : \tau$, *then if* e *halts to a value, so must* e'.

THEOREM 2.2 (FUNDAMENTAL THEOREM INTERMEDIATE LANGUAGE). *For any well* syntactically *typed expression (in* $\lambda^{\vdash}$*), say* $\Gamma \vdash \mathrm{e} : \tau$, *we automatically have that* $\Gamma \vDash_{int.} \mathrm{e} : \tau$.

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

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

$$\mathrm{C}_b^{\vdash}[\mathrm{e}] \Downarrow \;\; \textit{iff} \;\; \mathrm{C}[\mathrm{e}] \Downarrow$$

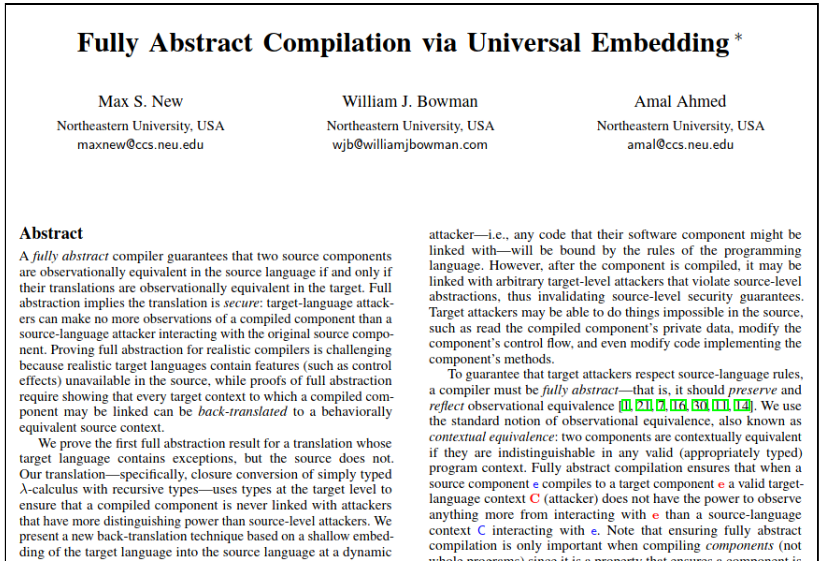
$$\vDash_{int.} \mathsf{C} : (\mathsf{x}_1 : \tau_1, \ldots, \mathsf{x}_n : \tau_n; \tau) \Rightarrow (\,\cdot\,; \mathsf{1})$$

$$\vDash_{int.} \mathsf{C} : (\mathsf{x}_1 : \tau_1, \ldots, \mathsf{x}_n : \tau_n; \tau) \Rightarrow (\,\cdot\,; \mathsf{1})$$

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

$$\vDash_{int.} \mathsf{C} : (\mathsf{x}_1 : \tau_1, \ldots, \mathsf{x}_n : \tau_n; \tau) \Rightarrow (\,\cdot\,; 1)$$

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



Fully Abstract Compilation via Universal Embedding*

Max S. New
Northeastern University, USA
maxnew@ccs.neu.edu

William J. Bowman
Northeastern University, USA
wjb@williamjbowman.com

Amal Ahmed
Northeastern University, USA
amal@ccs.neu.edu

Abstract

A *fully abstract* compiler guarantees that two source components are observationally equivalent in the source language if and only if their translations are observationally equivalent in the target. Full abstraction implies the translation is *secure*: target-language attackers can make no more observations of a compiled component than a source-language attacker interacting with the original source component. Proving full abstraction for realistic compilers is challenging because realistic target languages contain features (such as control effects) unavailable in the source, while proofs of full abstraction require showing that every target context to which a compiled component may be linked can be *back-translated* to a behaviorally equivalent source context.

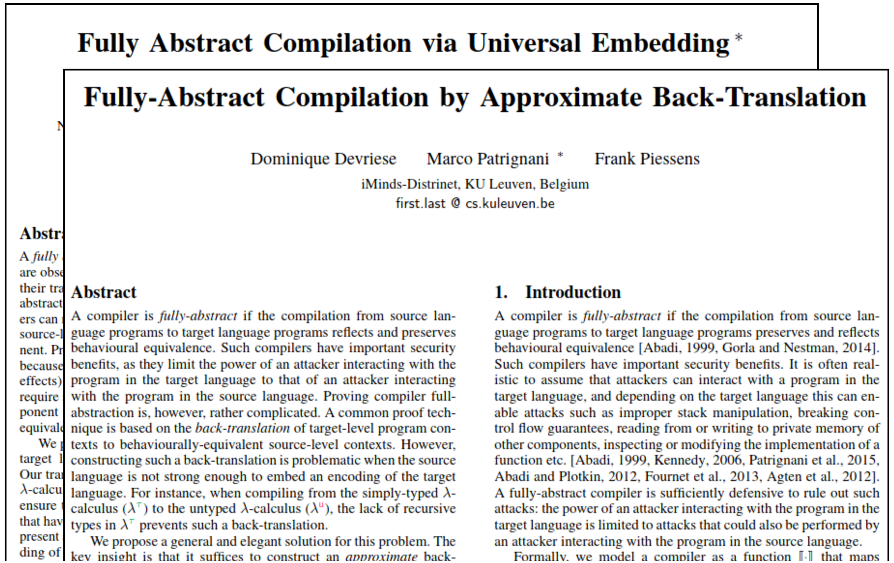
We prove the first full abstraction result for a translation whose target language contains exceptions, but the source does not. Our translation—specifically, closure conversion of simply typed λ-calculus with recursive types—uses types at the target level to ensure that a compiled component is never linked with attackers that have more distinguishing power than source-level attackers. We present a new back-translation technique based on a shallow embedding of the target language into the source language at a dynamic

attacker—i.e., any code that their software component might be linked with—will be bound by the rules of the programming language. However, after the component is compiled, it may be linked with arbitrary target-level attackers that violate source-level abstractions, thus invalidating source-level security guarantees. Target attackers may be able to do things impossible in the source, such as read the compiled component's private data, modify the component's control flow, and even modify code implementing the component's methods.

To guarantee that target attackers respect source-language rules, a compiler must be *fully abstract*—that is, it should *preserve* and *reflect* observational equivalence [1, 21, 7, 16, 30, 11, 14]. We use the standard notion of observational equivalence, also known as *contextual equivalence*: two components are contextually equivalent if they are indistinguishable in any valid (appropriately typed) program context. Fully abstract compilation ensures that when a source component e compiles to a target component e a valid target-language context C (attacker) does not have the power to observe anything more from interacting with e than a source-language context C interacting with e. Note that ensuring fully abstract compilation is only important when compiling *components* (not

$$\vDash_{int.} \mathsf{C} : (\mathsf{x}_1 : \tau_1, \ldots, \mathsf{x}_n : \tau_n; \tau) \Rightarrow (\,\cdot\,; \mathsf{1})$$

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



Fully Abstract Compilation via Universal Embedding*

Fully-Abstract Compilation by Approximate Back-Translation

Dominique Devriese   Marco Patrignani *   Frank Piessens

iMinds-Distrinet, KU Leuven, Belgium

first.last @ cs.kuleuven.be

Abstract

A compiler is *fully-abstract* if the compilation from source language programs to target language programs reflects and preserves behavioural equivalence. Such compilers have important security benefits, as they limit the power of an attacker interacting with the program in the target language to that of an attacker interacting with the program in the source language. Proving compiler full-abstraction is, however, rather complicated. A common proof technique is based on the *back-translation* of target-level program contexts to behaviourally-equivalent source-level contexts. However, constructing such a back-translation is problematic when the source language is not strong enough to embed an encoding of the target language. For instance, when compiling from the simply-typed λ-calculus (λ^{τ}) to the untyped λ-calculus (λ^{u}), the lack of recursive types in λ^{τ} prevents such a back-translation.

We propose a general and elegant solution for this problem. The key insight is that it suffices to construct an *approximate* back-

1. Introduction

A compiler is *fully-abstract* if the compilation from source language programs to target language programs preserves and reflects behavioural equivalence [Abadi, 1999, Gorla and Nestman, 2014]. Such compilers have important security benefits. It is often realistic to assume that attackers can interact with a program in the target language, and depending on the target language this can enable attacks such as improper stack manipulation, breaking control flow guarantees, reading from or writing to private memory of other components, inspecting or modifying the implementation of a function etc. [Abadi, 1999, Kennedy, 2006, Patrignani et al., 2015, Abadi and Plotkin, 2012, Fournet et al., 2013, Agten et al., 2012]. A fully-abstract compiler is sufficiently defensive to rule out such attacks: the power of an attacker interacting with the program in the target language is limited to attacks that could also be performed by an attacker interacting with the program in the source language.

Formally, we model a compiler as a function $[\![\cdot]\!]$ that maps

$$\vDash_{int.} \mathsf{C} : (\mathsf{x}_1 : \tau_1, \ldots, \mathsf{x}_n : \tau_n; \tau) \Rightarrow (\,\cdot\,; 1)$$

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

Fully Abstract Compilation via Universal Embedding *

Fully-Abstract Compilation by Approximate Back-Translation

Fully Abstract from Static to Gradual

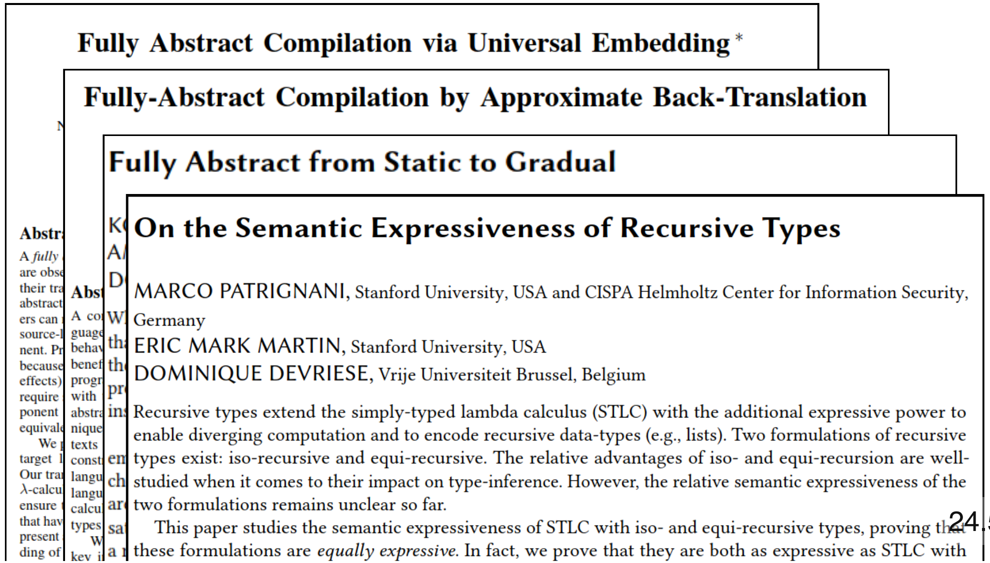
KOEN JACOBS, imec-DistriNet, KU Leuven, Belgium
AMIN TIMANY, Aarhus University, Denmark
DOMINIQUE DEVRIESE, Vrije Universiteit Brussel, Belgium

What is a good gradual language? Siek et al. have previously proposed the refined criteria, a set of formal ideas that characterize a range of guarantees typically expected from a gradual language. While these go a long way, they are mostly focused on syntactic and type safety properties and fail to characterize how richer semantic properties and reasoning principles that hold in the static language, like non-interference or parametricity for instance, should be upheld in the gradualization.

In this paper, we investigate and argue for a new criterion previously hinted at by Devriese et al.: the embedding from the static to the gradual language should be fully abstract. Rather than preserving an arbitrarily chosen interpretation of source language types, this criterion requires that *all* source language equivalences are preserved. We demonstrate that the criterion weeds out erroneous gradualizations that nevertheless satisfy the refined criteria. At the same time, we demonstrate that the criterion is realistic by reporting on a mechanized proof that the property holds for a standard example: GTLC$_\mu$, the natural gradualization of

$$\vDash_{int.} \mathsf{C} : (\mathsf{x}_1 : \tau_1, \ldots, \mathsf{x}_n : \tau_n; \tau) \Rightarrow (\,\cdot\,; 1)$$

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


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-recursion 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

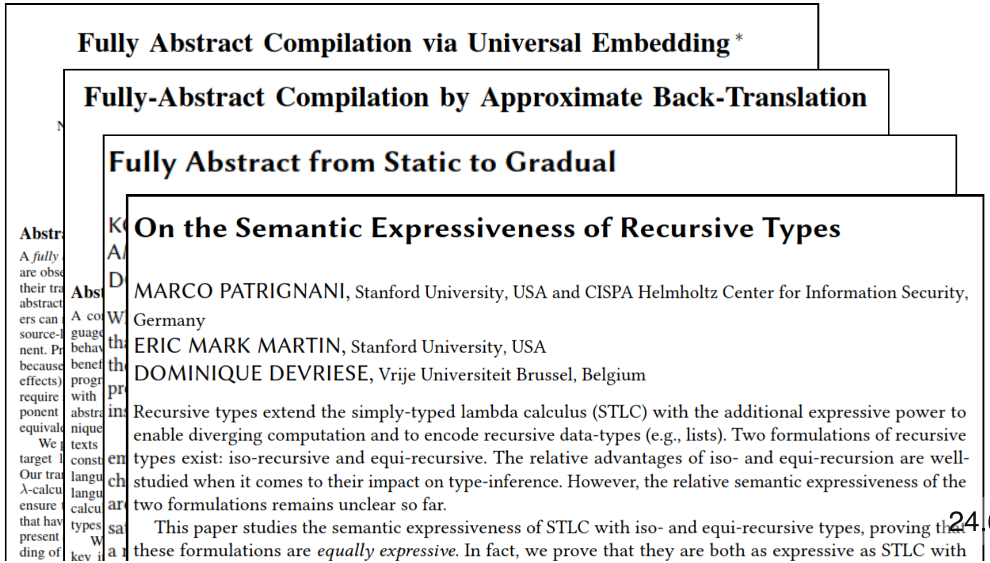
$$\langle\!\langle \pi_1\ e \rangle\!\rangle = \pi_1\ (\mathrm{extract}_{\times}\ \langle\!\langle e \rangle\!\rangle)$$

$$\mathrm{extract}_{\otimes} : \mathcal{U} \to (\mathcal{U} \otimes \mathcal{U})$$

$$\vDash_{int.} \mathrm{C} : (x_1 : \tau_1, \ldots, x_n : \tau_n; \tau) \Rightarrow (\,\cdot\,; 1)$$

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

$$\vdash \langle\!\langle \mathrm{C} \rangle\!\rangle : (x_1 : \mathcal{U}, \ldots, x_n : \mathcal{U}; \mathcal{U}) \Rightarrow (\,\cdot\,; \mathcal{U})$$



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-recursion 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

$$\langle\!\langle \pi_1\ e \rangle\!\rangle = \pi_1\ (\mathrm{extract}_\times\ \langle\!\langle e \rangle\!\rangle)$$

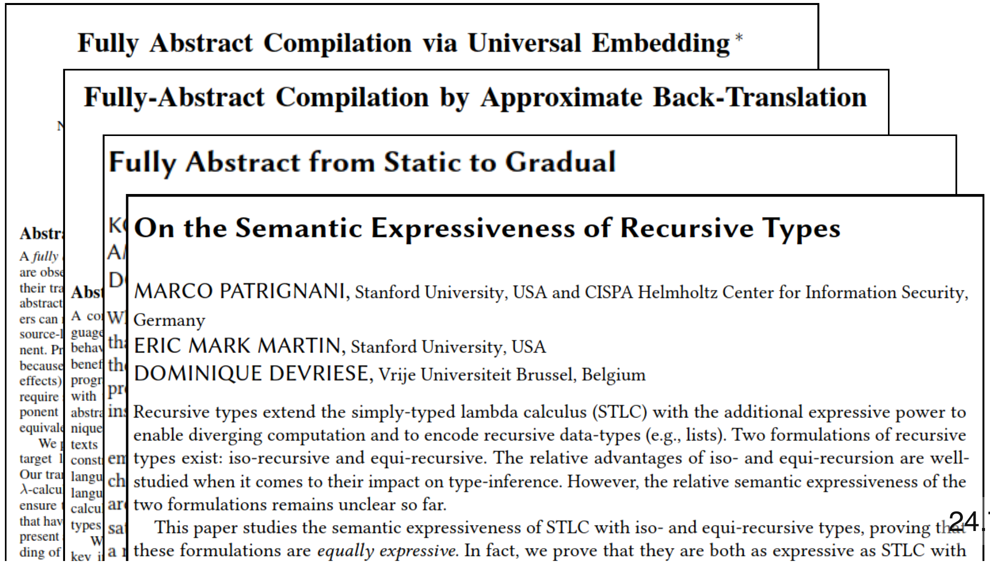
$$\mathrm{extract}_\otimes : \mathcal{U} \to (\mathcal{U} \otimes \mathcal{U})$$

$$\vDash_{int.} C : (x_1 : \tau_1, \ldots, x_n : \tau_n; \tau) \Rightarrow (\,\cdot\,; 1)$$

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

$$\vdash \langle\!\langle C \rangle\!\rangle : (x_1 : \mathcal{U}, \ldots, x_n : \mathcal{U}; \mathcal{U}) \Rightarrow (\,\cdot\,; \mathcal{U})$$

$$\mathrm{project}_\tau : \mathcal{U} \to \tau$$
$$\mathrm{embed}_\tau : \tau \to \mathcal{U}$$



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-recursion 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

$$\langle\langle \pi_1\ e \rangle\rangle = \pi_1\ (\mathrm{extract}_{\times}\ \langle\langle e \rangle\rangle)$$

$$\mathrm{extract}_{\otimes} : \mathcal{U} \to (\mathcal{U} \otimes \mathcal{U})$$

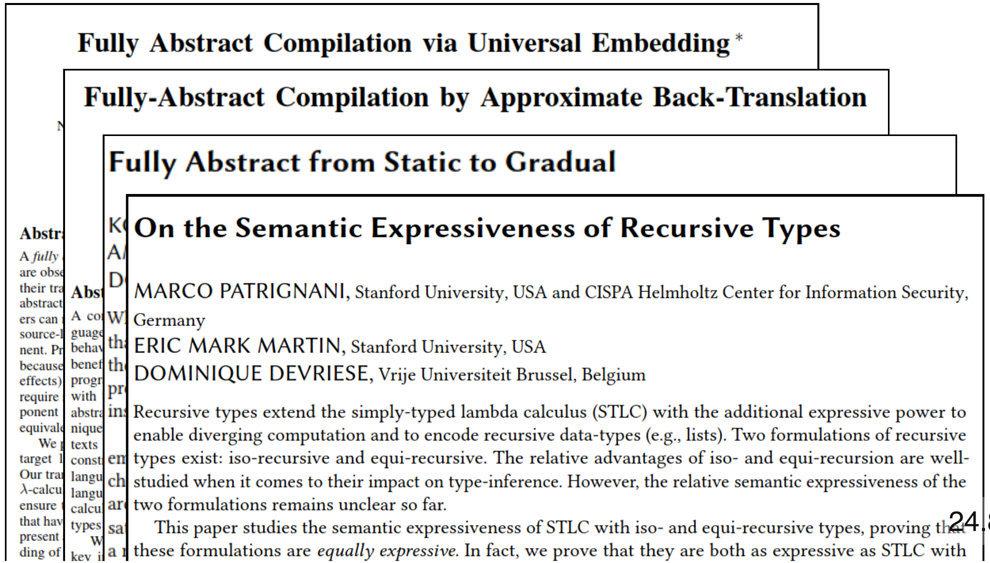
$$\vDash_{int.}\ C : (x_1 : \tau_1, \ldots, x_n : \tau_n; \tau) \Rightarrow (\ \cdot\ ; 1)$$

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

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

$$\mathrm{project}_{\tau} : \mathcal{U} \to \tau$$
$$\mathrm{embed}_{\tau} : \tau \to \mathcal{U}$$

$$\vdash \mathcal{EP}\langle\langle C \rangle\rangle : (\Gamma; \tau) \Rightarrow (\cdot\ ; 1)$$


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-recursion 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

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

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

$$\mathsf{C}^{\vdash}_{b}[\mathsf{e}] \Downarrow \ \textit{iff}\ \mathsf{C}[\mathsf{e}] \Downarrow$$

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

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

$$\mathrm{C}_b^{\vdash}[\mathrm{e}] \Downarrow \;\; \textit{iff} \;\; \mathrm{C}[\mathrm{e}] \Downarrow$$

Our proof does not assume a fixed set of ghost resources!

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

Given $\vdash \mathrm{C} : (\cdot \mid \lceil\!\lceil \Gamma \rceil\!\rceil ; \lceil\!\lceil \tau \rceil\!\rceil) \Rightarrow (\cdot \mid \cdot\, ; 1)$, *there exists* $\vDash_{int.} \mathrm{C}^{\vDash}_{b} : (\Gamma ; \tau) \Rightarrow (\cdot\, ; 1)$.
This is a valid emulation, i.e. for all $\Gamma \vdash \mathrm{e} : \tau$:

$$\mathrm{C}^{\vDash}_{b}[\mathrm{e}] \Downarrow \textit{ iff } \mathrm{C}[\lceil\!\lceil \mathrm{e} \rceil\!\rceil] \Downarrow$$

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

Given $\vdash \mathrm{C} : (\cdot \mid \lceil\!\lceil \Gamma \rceil\!\rceil ; \lceil\!\lceil \tau \rceil\!\rceil) \Rightarrow (\cdot \mid \cdot \,; 1)$, *there exists* $\vDash_{int.} \mathrm{C}^{\vDash}_{b} : (\Gamma ; \tau) \Rightarrow (\cdot \,; 1)$.
This is a valid emulation, i.e. for all $\Gamma \vdash \mathrm{e} : \tau$:

$$\mathrm{C}^{\vDash}_{b}[\mathrm{e}] \Downarrow \ \textit{iff}\ \ \mathrm{C}[\lceil\!\lceil \mathrm{e} \rceil\!\rceil] \Downarrow$$

$$\mathcal{E} : \textit{List}\ \mathsf{Val} \to \mathsf{Val}$$
$$\mathcal{E}([\mathrm{v}_1;\ \mathrm{v}_2;\ \mathrm{v}_3]) = ((\mathrm{v}_3, (\mathrm{v}_2, (\mathrm{v}_1, ()))), 3)$$

$$\begin{array}{rll}
\mathsf{read}\ \mathrm{z}\ \mathcal{E}(\vec{\mathrm{v}}) \longrightarrow^{*} & (\mathcal{E}(\vec{\mathrm{v}}) \qquad\quad , \vec{\mathrm{v}}.\mathrm{z}) & \textit{if}\ 0 \le \mathrm{z} < |\vec{\mathrm{v}}| \\
\mathsf{ref}\ \mathrm{v}\ \mathcal{E}(\vec{\mathrm{v}}) \longrightarrow^{*} & (\mathcal{E}(\vec{\mathrm{v}} +\!\!+ [\mathrm{v}])\ , |\vec{\mathrm{v}}|) & \\
\mathsf{write}\ \mathrm{z}\ \mathrm{v}\ \mathcal{E}(\vec{\mathrm{v}}) \longrightarrow^{*} & (\mathcal{E}(\vec{\mathrm{v}}[\mathrm{z} \mapsto \mathrm{v}]), ()) & \textit{if}\ 0 \le \mathrm{z} < |\vec{\mathrm{v}}|
\end{array}$$

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

Given $\vdash C : (\cdot \mid \llbracket\Gamma\rrbracket; \llbracket\tau\rrbracket) \Rightarrow (\cdot \mid \cdot\,; 1)$, *there exists* $\vDash_{int.} C^{\vDash}_{b} : (\Gamma; \tau) \Rightarrow (\cdot\,; 1)$. *This is a valid emulation, i.e. for all* $\Gamma \vdash e : \tau$*:*

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

$$\mathcal{E} : List\ \mathsf{Val} \rightarrow \mathsf{Val}$$

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

$$\begin{aligned}
\mathsf{read}\ z\ \mathcal{E}(\vec{v}) &\longrightarrow^{*} (\mathcal{E}(\vec{v}) &&, \vec{v}.z) && \text{if } 0 \leq z < |\vec{v}| \\
\mathsf{ref}\ v\ \mathcal{E}(\vec{v}) &\longrightarrow^{*} (\mathcal{E}(\vec{v} \mathbin{+\!\!+} [v]) &&, |\vec{v}|) \\
\mathsf{write}\ z\ v\ \mathcal{E}(\vec{v}) &\longrightarrow^{*} (\mathcal{E}(\vec{v}[z \mapsto v]) &&, ()) && \text{if } 0 \leq z < |\vec{v}|
\end{aligned}$$

$$\begin{aligned}
\langle\!\langle !e \rangle\!\rangle &= \mathsf{read}\ \langle\!\langle e \rangle\!\rangle \\
\langle\!\langle e \leftarrow e' \rangle\!\rangle &= \mathsf{write}\ \langle\!\langle e \rangle\!\rangle\ \langle\!\langle e' \rangle\!\rangle \\
\langle\!\langle \mathbf{ref}\ e \rangle\!\rangle &= \mathsf{ref}\ \langle\!\langle e \rangle\!\rangle \\
\langle\!\langle e \ggg e' \rangle\!\rangle &= (\lambda x.\ \lambda f.\ \lambda h_0.\ \mathsf{let}\ (h_1, a) = x\ h_0\ \mathsf{in}\ f\ a\ h_1)\ \langle\!\langle e \rangle\!\rangle\ \langle\!\langle e' \rangle\!\rangle
\end{aligned}$$

Given $\vdash \mathsf{C} : (\cdot \mid [\![\Gamma]\!]; [\![\tau]\!]) \Rightarrow (\cdot \mid \cdot ; 1)$, *there exists* $\vDash_{int.} \mathsf{C}_b^{\vDash} : (\Gamma; \tau) \Rightarrow (\cdot ; 1)$.
This is a valid emulation, i.e. for all $\Gamma \vdash \mathsf{e} : \tau$:

$$\mathsf{C}_b^{\vDash}[\mathsf{e}] \Downarrow \text{ iff } \mathsf{C}[[\![\mathsf{e}]\!]] \Downarrow$$

Given $\vdash \mathsf{C} : (\cdot \mid [\![\Gamma]\!]; [\![\tau]\!]) \Rightarrow (\cdot \mid \cdot ; 1)$,

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

Given $\vdash \mathsf{C} : (\cdot \mid \llbracket\Gamma\rrbracket; \llbracket\tau\rrbracket) \Rightarrow (\cdot \mid \cdot\,; 1)$, *there exists* $\vDash_{int.} \mathsf{C}^{\vDash}_{b} : (\Gamma; \tau) \Rightarrow (\cdot\,; 1)$.
This is a valid emulation, i.e. for all $\Gamma \vdash \mathsf{e} : \tau$*:*

$$\mathsf{C}^{\vDash}_{b}[\mathsf{e}] \Downarrow \;\textit{iff}\; \mathsf{C}[\llbracket\mathsf{e}\rrbracket] \Downarrow$$

$$\text{Given } \vdash \mathsf{C} : (\cdot \mid \llbracket\Gamma\rrbracket; \llbracket\tau\rrbracket) \Rightarrow (\cdot \mid \cdot\,; 1),$$

$$\text{to prove } \vDash_{int.} \langle\!\langle \mathsf{C} \rangle\!\rangle \leq \langle\!\langle \mathsf{C} \rangle\!\rangle : (\Gamma; \tau) \Rightarrow (\cdot\,; 1)$$

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

$$\Xi \mid \Gamma \vDash_{\chi} \mathsf{e} \leq \mathsf{e}' : \tau$$

Given $\vdash \mathsf{C} : (\cdot \mid [\![\Gamma]\!] ; [\![\tau]\!]) \Rightarrow (\cdot \mid \cdot\, ; 1)$, *there exists* $\vDash_{int.} \mathsf{C}_b^{\vDash} : (\Gamma; \tau) \Rightarrow (\cdot\, ; 1)$. *This is a valid emulation, i.e. for all* $\Gamma \vdash \mathsf{e} : \tau$*:*

$$\mathsf{C}_b^{\vDash}[\mathsf{e}] \Downarrow \ \textit{iff} \ \mathsf{C}[[\![\mathsf{e}]\!]] \Downarrow$$

$$\textit{Given} \ \vdash \mathsf{C} : (\cdot \mid [\![\Gamma]\!] ; [\![\tau]\!]) \Rightarrow (\cdot \mid \cdot\, ; 1),$$

$$\textit{to prove} \ \vDash_{int.} \langle\!\langle \mathsf{C} \rangle\!\rangle \leq \langle\!\langle \mathsf{C} \rangle\!\rangle : (\Gamma; \tau) \Rightarrow (\cdot\, ; 1)$$

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

$$\Xi \mid \Gamma \vDash_{\chi} \mathsf{e} \leq \mathsf{e}' : \tau$$

$$\Gamma \vDash_{int.} \mathsf{e} \leq \mathsf{e}' : \tau \dashv\vdash \cdot \mid [\![\Gamma]\!] \vDash_{\chi} \mathsf{e} \leq \mathsf{e}' : [\![\tau]\!]$$

Given $\vdash \mathsf{C} : (\cdot \mid \llbracket\Gamma\rrbracket; \llbracket\tau\rrbracket) \Rightarrow (\cdot \mid \cdot\,; 1)$, *there exists* $\vDash_{int.} \mathsf{C}^{\vDash}_{b} : (\Gamma; \tau) \Rightarrow (\cdot\,; 1)$.
This is a valid emulation, i.e. for all $\Gamma \vdash \mathsf{e} : \tau$*:*

$$\mathsf{C}^{\vDash}_{b}[\mathsf{e}] \Downarrow \ \textit{iff}\ \mathsf{C}[\llbracket\mathsf{e}\rrbracket] \Downarrow$$

$$\boxed{\begin{array}{l} \textit{Given} \ \vdash \mathsf{C} : (\cdot \mid \llbracket\Gamma\rrbracket; \llbracket\tau\rrbracket) \Rightarrow (\cdot \mid \cdot\,; 1), \\ \quad \textit{to prove} \ \vDash_{int.} \langle\!\langle \mathsf{C} \rangle\!\rangle \leq \langle\!\langle \mathsf{C} \rangle\!\rangle : (\Gamma; \tau) \Rightarrow (\cdot\,; 1) \end{array}}$$

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

$$\Xi \mid \Gamma \vDash_{\chi} \mathsf{e} \leq \mathsf{e}' : \tau$$

$$\Gamma \vDash_{int.} \mathsf{e} \leq \mathsf{e}' : \tau \ \dashv\vdash\ \cdot \mid \llbracket\Gamma\rrbracket \vDash_{\chi} \mathsf{e} \leq \mathsf{e}' : \llbracket\tau\rrbracket$$

$$\forall\, \Xi \mid \Gamma \vdash \mathsf{e} : \tau.\ \ \Xi \mid \Gamma \vDash_{\chi} \langle\!\langle \mathsf{e} \rangle\!\rangle \leq \langle\!\langle \mathsf{e} \rangle\!\rangle : \tau$$

Given $\vdash \mathsf{C} : (\cdot \mid \lceil\!\lceil \Gamma \rceil\!\rceil; \lceil\!\lceil \tau \rceil\!\rceil) \Rightarrow (\cdot \mid \cdot ; 1)$, *there exists* $\vDash_{int.} \mathsf{C}_b^{\vDash} : (\Gamma; \tau) \Rightarrow (\cdot ; 1)$.
This is a valid emulation, i.e. for all $\Gamma \vdash \mathsf{e} : \tau$:

$$\mathsf{C}_b^{\vDash}[\mathsf{e}] \Downarrow \ \textit{iff} \ \mathsf{C}[\lceil\!\lceil \mathsf{e} \rceil\!\rceil] \Downarrow$$

Given $\vdash \mathsf{C} : (\cdot \mid \lceil\!\lceil \Gamma \rceil\!\rceil ; \lceil\!\lceil \tau \rceil\!\rceil) \Rightarrow (\cdot \mid \cdot \,; 1)$*, there exists* $\vDash_{int.} \mathsf{C}^{\vDash}_{b} : (\Gamma ; \tau) \Rightarrow (\cdot \,; 1)$*. This is a valid emulation, i.e. for all* $\Gamma \vdash \mathsf{e} : \tau$*:*

$\Leftarrow$

$$\mathsf{C}^{\vDash}_{b}[\mathsf{e}] \Downarrow \textit{ iff } \mathsf{C}[\lceil\!\lceil \mathsf{e} \rceil\!\rceil] \Downarrow$$

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

$$\Xi \mid \Gamma \vDash_{\mathcal{R}} \mathbf{e} \leq \langle\!\langle \mathbf{e} \rangle\!\rangle : \tau$$

Given $\vdash \mathrm{C} : (\cdot \mid \llbracket\Gamma\rrbracket; \llbracket\tau\rrbracket) \Rightarrow (\cdot \mid \cdot\,; 1)$, *there exists* $\vDash_{int.} \mathrm{C}^{\vDash}_{b} : (\Gamma; \tau) \Rightarrow (\cdot\,; 1)$. *This is a valid emulation, i.e. for all* $\Gamma \vdash \mathrm{e} : \tau$:

$$\mathrm{C}^{\vDash}_{b}[\mathrm{e}] \Downarrow \textit{ iff } \mathrm{C}[\llbracket\mathrm{e}\rrbracket] \Downarrow$$

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

$$\Xi \mid \Gamma \vDash_{\mathcal{R}} \mathrm{e} \leq \langle\!\langle \mathrm{e} \rangle\!\rangle : \tau$$

Lemma 5.3 (Logical Relation Adequacy). *If* $\cdot \mid \cdot \vDash_{\mathcal{R}} \mathrm{e} \leq \mathrm{e}' : \tau$, *then if* e *halts to a value, so must* e'.

$$\mathit{OwnState}_\gamma(\vec{\mathsf{v}}) \vdash \rightrightarrows\mathit{OwnState}_\gamma(\vec{\mathsf{v}} \mathbin{+\!\!+} [\mathsf{v}]) * |\vec{\mathsf{v}}| \mapsto_\gamma \mathsf{v}$$

$$\mathit{OwnState}_\gamma(\vec{\mathsf{v}}) * z \mapsto_\gamma \mathsf{v} \vdash \vec{\mathsf{v}}.z = \mathsf{v}$$

$$\mathit{OwnState}_\gamma(\vec{\mathsf{v}}) * z \mapsto_\gamma \mathsf{v} \vdash \rightrightarrows\mathit{OwnState}_\gamma(\vec{\mathsf{v}}[z \mapsto \mathsf{w}]) * z \mapsto_\gamma \mathsf{w}$$

$$\mathit{OwnState}_\gamma(\vec{\mathsf{v}}) \vdash \Rrightarrow \mathit{OwnState}_\gamma(\vec{\mathsf{v}} \mathbin{+\!\!+} [\mathsf{v}]) * |\vec{\mathsf{v}}| \mapsto_\gamma \mathsf{v}$$

$$\mathit{OwnState}_\gamma(\vec{\mathsf{v}}) * z \mapsto_\gamma \mathsf{v} \vdash \vec{\mathsf{v}}.z = \mathsf{v}$$

$$\mathit{OwnState}_\gamma(\vec{\mathsf{v}}) * z \mapsto_\gamma \mathsf{v} \vdash \Rrightarrow \mathit{OwnState}_\gamma(\vec{\mathsf{v}}[z \mapsto \mathsf{w}]) * z \mapsto_\gamma \mathsf{w}$$

$$\mathit{OwnLocs}_\gamma(\vec{\ell}) \vdash \Rrightarrow \mathit{OwnLocs}_\gamma(\vec{\ell} \mathbin{+\!\!+} [\ell]) * |\vec{\ell}| \mapsto_\gamma^\square \ell$$

$$\mathit{OwnLocs}_\gamma(\vec{\ell}) * z \mapsto_\gamma^\square \ell \vdash \vec{\ell}.z = \ell$$

$$z \mapsto_\gamma^\square \ell \vdash z \mapsto_\gamma^\square \ell * z \mapsto_\gamma^\square \ell$$

$$z \mapsto_\gamma^\square \ell * z \mapsto_\gamma^\square \ell' \vdash \ell = \ell'$$

$$OwnState_\gamma(\vec{v}) \vdash \Rrightarrow OwnState_\gamma(\vec{v} +\!\!+ [v]) * |\vec{v}| \mapsto_\gamma v$$

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

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

$$OwnLocs_\gamma(\vec{\ell}) \vdash \Rrightarrow OwnLocs_\gamma(\vec{\ell} +\!\!+ [\ell]) * |\vec{\ell}| \mapsto_\gamma^\square \ell$$

$$OwnLocs_\gamma(\vec{\ell}) * z \mapsto_\gamma^\square \ell \vdash \vec{\ell}.z = \ell$$

$$z \mapsto_\gamma^\square \ell \vdash z \mapsto_\gamma^\square \ell * z \mapsto_\gamma^\square \ell$$

$$z \mapsto_\gamma^\square \ell * z \mapsto_\gamma^\square \ell' \vdash \ell = \ell'$$

$$\mathrm{lift}_{\mathcal{R}} : (\mathrm{Val} \to \mathrm{Val} \to iProp) \to (\mathrm{Expr} \to \mathrm{Expr} \to iProp)$$

$$\mathrm{lift}_{\mathcal{R}}\ \Phi\ (e, e') = \mathrm{wp}\ e\ \{v.\ \exists v'.\ e' \to^* v' * \Phi(v, v')\}$$

$$\mathcal{V}_{\mathcal{R}}[\![\Xi \vdash \mathsf{STRef}\ \mathsf{X}\ \tau]\!]_{\Delta}(\mathrm{v}, \mathrm{v}') \triangleq \exists \ell, z.\ \mathrm{v} = \ell * \mathrm{v}' = z * z \mapsto^{\square}_{\Delta(\mathsf{X}).1} \ell * \boxed{\exists \mathrm{w}, \mathrm{w}'.\ \ell \mapsto \mathrm{w} * z \mapsto_{\Delta(\mathsf{X}).2} (\mathrm{w}') * \mathcal{V}_{\mathcal{R}}[\![\Xi \vdash \tau]\!]_{\Delta}(\mathrm{w}, \mathrm{w}')}^{\Delta(\mathsf{X}).z}$$

$$\mathcal{V}_{\mathcal{R}}[\![\Xi \vdash \mathrm{STRef}\ \mathrm{X}\ \tau]\!]_{\Delta}(\mathrm{v}, \mathrm{v}') \triangleq \exists \ell, z.\ \mathrm{v} = \ell * \mathrm{v}' = z * z \mapsto^{\square}_{\Delta(\mathrm{X}).1} \ell * \boxed{\exists \mathrm{w}, \mathrm{w}'.\ \ell \mapsto \mathrm{w} * z \mapsto_{\Delta(\mathrm{X}).2} (\mathrm{w}') * \mathcal{V}_{\mathcal{R}}[\![\Xi \vdash \tau]\!]_{\Delta}(\mathrm{w}, \mathrm{w}')}^{\Delta(\mathrm{X}).z}$$

$$\mathcal{V}_{\mathcal{R}}[\![\Xi \vdash \mathrm{ST}\ \mathrm{X}\ \tau]\!]_{\Delta}(\mathrm{v}, \mathrm{v}') \triangleq \forall \vec{\ell}_i, \vec{\mathrm{v}}_i.\ |\vec{\ell}_i| = |\vec{\mathrm{v}}_i|.\ \square\Big(\mathit{OwnLocs}_{\Delta(\mathrm{X}).1}(\vec{\ell}_i) * \mathit{OwnState}_{\Delta(\mathrm{X}).2}(\vec{\mathrm{v}}_i) \mathrel{-\!\!*}$$
$$\mathrm{wp}\ \mathrm{runST}\ \{\mathrm{v}\} \Big\{\mathrm{w}.\ \exists \mathrm{w}', \vec{\ell}_f, \vec{\mathrm{v}}_f.\ |\vec{\ell}_f| = |\vec{\mathrm{v}}_f|.(\mathrm{v}'\ \mathcal{E}(\vec{\mathrm{v}}_i) \rightarrow^{*} (\mathcal{E}(\vec{\mathrm{v}}_f), \mathrm{w}')) *$$
$$\mathit{OwnLocs}_{\Delta(\mathrm{X}).1}(\vec{\ell}_f) * \mathit{OwnState}_{\Delta(\mathrm{X}).2}(\vec{\mathrm{v}}_f) * \mathcal{V}_{\mathcal{R}}[\![\Xi \vdash \tau]\!]_{\Delta}(\mathrm{w}, \mathrm{w}')\Big\}\Big)$$

$$\mathrm{lift}_{\mathcal{R}} : (\mathrm{Val} \longrightarrow \mathrm{Val} \longrightarrow \mathit{iProp}) \longrightarrow (\mathrm{Expr} \longrightarrow \mathrm{Expr} \longrightarrow \mathit{iProp})$$

$$\mathrm{lift}_{\mathcal{R}}\ \Phi\ (\mathrm{e}, \mathrm{e}') = \mathrm{wp}\ \mathrm{e}\ \left\{\mathrm{v}.\ \exists \mathrm{v}'.\ \mathrm{e}' \longrightarrow^{*} \mathrm{v}' * \Phi(\mathrm{v}, \mathrm{v}')\right\}$$

$$\mathrm{lift}_{\mathcal{R}} : (\mathrm{Val} \to \mathrm{Val} \to iProp) \to (\mathrm{Expr} \to \mathrm{Expr} \to iProp)$$

$$\mathrm{lift}_{\mathcal{R}}\ \Phi\ (\mathrm{e}, \mathrm{e}') = \mathrm{wp}\ \mathrm{e}\ \left\{\mathrm{v}.\ \exists \mathrm{v}'.\ \mathrm{e}' \to^{*} \mathrm{v}' * \Phi(\mathrm{v}, \mathrm{v}')\right\}$$

$$\Xi \mid \Gamma \vDash_{\mathcal{R}} \mathrm{e} \leq \mathrm{e}' : \tau \triangleq$$

$$\forall \Delta, \vec{\mathrm{v}}, \vec{\mathrm{v}}', .\ \vec{\mathcal{V}}_{\mathcal{R}}[\![\Xi \vdash \Gamma]\!]_{\Delta}(\vec{\mathrm{v}}, \vec{\mathrm{v}}') \vdash \mathcal{E}_{\mathcal{R}}[\![\Xi \vdash \tau]\!]_{\Delta}(\mathrm{e}[\vec{\mathrm{x}}/\vec{\mathrm{v}}], \mathrm{e}'[\vec{\mathrm{x}}/\vec{\mathrm{v}}'])$$

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*

$$\mathrm{wp}\ \mathrm{e}\ \{\Phi\}$$

Sometimes *existing abstractions* are not *sufficient*

$$\mathrm{wp}\ e\ \{\Phi\}$$

expressions are of *finite depth*

Sometimes *existing abstractions* are not *sufficient*

$$\mathrm{wp}\ e\ \{\Phi\}$$

expressions are of *finite depth*

Sometimes *existing abstractions* are not *sufficient*

$$\mathrm{wp}\ \mathrm{e}\ \{\Phi\}$$

expressions are of *finite depth*

$$\mathrm{e} ::= \dots \mid \mapsto(\mathrm{e})$$

$$\mapsto(\mathrm{w}) \rightarrow_h \mathrm{w} \quad \textit{if}\ \mathrm{w} = (), \mathrm{b}, \mathrm{z}$$

$$\mapsto(\mathrm{inj}_2\ \mathrm{v}) \rightarrow_h \mathrm{inj}_2\ (\mapsto(\mathrm{v}))$$

$$\mapsto(\lambda x.\mathrm{e}) \rightarrow_h \lambda y.((\lambda x.\mapsto(\mathrm{e}))\ \mapsto(y))$$

$$\mapsto(\mathrm{inj}_1\ \mathrm{v}) \rightarrow_h \mathrm{inj}_1\ (\mapsto(\mathrm{v}))$$

$$\mapsto((\mathrm{v}_1, \mathrm{v}_2)) \rightarrow_h (\mapsto(\mathrm{v}_1), \mapsto(\mathrm{v}_2))$$

$$\mapsto(\mathrm{fold}\ \mathrm{v}) \rightarrow_h \mathrm{fold}\ (\mapsto(\mathrm{v}))$$

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 polymorphsim?

“ ***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

$$\textit{Given} \quad \vdash \mathbf{C} : (\cdot \mid [\![\Gamma]\!] ; [\![\tau]\!]) \Rightarrow (\,\cdot \mid \cdot\,; 1),$$

$$\textit{to prove} \quad \vDash_{\mathit{int.}} \langle\!\langle \mathrm{C} \rangle\!\rangle \leq \langle\!\langle \mathrm{C} \rangle\!\rangle : (\Gamma; \tau) \Rightarrow (\cdot\,; 1)$$

$$\textit{Given} \quad \vdash C : (\cdot \mid [\![\Gamma]\!] ; [\![\tau]\!]) \Rightarrow (\cdot \mid \cdot ; 1),$$

$$\textit{to prove} \quad \vDash_{int.} \langle\!\langle C \rangle\!\rangle \leq \langle\!\langle C \rangle\!\rangle : (\Gamma ; \tau) \Rightarrow (\cdot ; 1)$$

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

$$\Xi \mid \Gamma \vDash_{\chi} e \leq e' : \tau$$

$$\textit{Given} \quad \vdash C : (\cdot \mid \llbracket\Gamma\rrbracket ; \llbracket\tau\rrbracket) \Rightarrow (\,\cdot \mid \cdot\, ; 1),$$

$$\textit{to prove} \quad \vDash_{int.} \langle\!\langle C \rangle\!\rangle \leq \langle\!\langle C \rangle\!\rangle : (\Gamma ; \tau) \Rightarrow (\cdot \,; 1)$$

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

$$\Xi \mid \Gamma \vDash_{\chi} e \leq e' : \tau$$

$$\Gamma \vDash_{int.} e \leq e' : \tau \dashv\vdash \cdot \mid \llbracket\Gamma\rrbracket \vDash_{\chi} e \leq e' : \llbracket\tau\rrbracket$$

$$\boxed{\begin{array}{l} \textit{Given} \;\; \vdash C : (\cdot \mid [\![\Gamma]\!] ; [\![\tau]\!]) \Rightarrow (\cdot \mid \cdot ; 1), \\ \quad \textit{to prove} \;\; \vDash_{int.} \langle\!\langle C \rangle\!\rangle \leq \langle\!\langle C \rangle\!\rangle : (\Gamma ; \tau) \Rightarrow (\cdot \, ; 1) \end{array}}$$

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

$$\Xi \mid \Gamma \vDash_{\chi} e \leq e' : \tau$$

$$\Gamma \vDash_{int.} e \leq e' : \tau \dashv\vdash \cdot \mid [\![\Gamma]\!] \vDash_{\chi} e \leq e' : [\![\tau]\!]$$

$$\forall\, \Xi \mid \Gamma \vdash e : \tau . \;\; \Xi \mid \Gamma \vDash_{\chi} \langle\!\langle e \rangle\!\rangle \leq \langle\!\langle e \rangle\!\rangle : \tau$$

$$\mathit{OwnStates}_\gamma(\vec{v}, \vec{v}') \vdash \Rrightarrow \mathit{OwnStates}_\gamma(\vec{v} +\!\!+ [v], \vec{v}' +\!\!+ [v']) * |\vec{v}| \mapsto_\gamma (v, v')$$

$$\mathit{OwnStates}_\gamma(\vec{v}, \vec{v}') * z \mapsto_\gamma (v, v') \vdash \vec{v}.z = v * \vec{v}'.z = v'$$

$$\mathit{OwnStates}_\gamma(\vec{v}, \vec{v}') * z \mapsto_\gamma (v, v') \vdash \Rrightarrow \mathit{OwnStates}_\gamma(\vec{v}[z \mapsto w], \vec{v}'[z \mapsto w']) * z \mapsto_\gamma (w, w')$$

$$\mathcal{V}_{\mathcal{X}}[\![\Xi \vdash \mathbf{ST}\ \mathbf{X}\ \boldsymbol{\tau}]\!]_{\Delta}(\mathrm{v}, \mathrm{v}') \triangleq \forall \vec{\mathrm{v}}_i, \vec{\mathrm{v}}_i'.\ \Box \Big(\mathit{OwnStates}_{\Delta(\mathrm{X})}(\vec{\mathrm{v}}_i, \vec{\mathrm{v}}_i') \twoheadrightarrow \mathrm{wp}\ \mathrm{v}\ \mathcal{E}(\vec{\mathrm{v}}_i) \Big\{ (\mathcal{E}(\vec{\mathrm{v}}_f), \mathrm{w}).\ \exists \vec{\mathrm{v}}_f', \mathrm{w}'.$$

$$(\mathrm{v}'\ \mathcal{E}(\vec{\mathrm{v}}_i')) \to^{*} (\mathcal{E}(\vec{\mathrm{v}}_f'), \mathrm{w}')) * \mathit{OwnStates}_{\Delta(\mathrm{X})}(\vec{\mathrm{v}}_f, \vec{\mathrm{v}}_f') * \mathcal{V}_{\mathcal{X}}[\![\Xi \vdash \boldsymbol{\tau}]\!]_{\Delta}(\mathrm{w}, \mathrm{w}') \Big\} \Big)$$

$$\mathcal{V}_{\mathcal{X}}[\![\Xi \vdash \mathbf{STRef}\ \mathbf{X}\ \boldsymbol{\tau}]\!]_{\Delta}(\mathrm{v}, \mathrm{v}') \triangleq \exists z.\ \mathrm{v} = z * \mathrm{v}' = z * \boxed{\exists \mathrm{w}, \mathrm{w}'.\ z \mapsto_{\Delta(\mathrm{X})} (\mathrm{w}, \mathrm{w}') * \mathcal{V}_{\mathcal{X}}[\![\Xi \vdash \boldsymbol{\tau}]\!]_{\Delta}(\mathrm{w}, \mathrm{w}')}^{\Delta(\mathrm{X}).z}$$

$$\Xi \mid \Gamma \vDash_{\mathcal{X}} \mathrm{e} \le \mathrm{e}' : \boldsymbol{\tau} \triangleq \forall \Delta, \vec{\mathrm{v}}, \vec{\mathrm{v}}'.\ |\vec{\mathrm{v}}| = |\vec{\mathrm{v}}'| = |\Gamma| * \mathop{\Large *}_{0 \le i < |\Gamma|} \mathcal{V}_{\mathcal{X}}[\![\Xi \vdash \Gamma.i]\!]_{\Delta}(\vec{\mathrm{v}}.i, \vec{\mathrm{v}}'.i) \vdash$$

$$\mathcal{E}_{\mathcal{X}}[\![\Xi \vdash \boldsymbol{\tau}]\!]_{\Delta}(\mathrm{e}[\vec{\mathrm{x}}/\vec{\mathrm{v}}], \mathrm{e}'[\vec{\mathrm{x}}/\vec{\mathrm{v}}'])$$

Extras

Existing *wp* not sufficient

$$\mathsf{lift} : (\mathsf{Val} \rightarrow \mathsf{Val} \rightarrow \mathit{iProp}) \rightarrow (\mathsf{Expr} \rightarrow \mathsf{Expr} \rightarrow \mathit{iProp})$$

$$\mathsf{lift}\ \Phi\ (\mathsf{e}, \mathsf{e}') = \mathsf{wp}\ \mathsf{e}\ \left\{\mathsf{v}.\ \exists \mathsf{v}'.\ \mathsf{e}' \rightarrow^{*} \mathsf{v}' * \Phi(\mathsf{v}, \mathsf{v}')\right\}$$

$$\mathsf{lift} : (\mathsf{Val} \to \mathsf{Val} \to iProp) \to (\mathsf{Expr} \to \mathsf{Expr} \to iProp)$$

$$\mathsf{lift}\ \Phi\ (\mathsf{e}, \mathsf{e}') = \mathsf{wp}\ \mathsf{e}\ \left\{\mathsf{v}.\ \exists \mathsf{v}'.\ \mathsf{e}' \longrightarrow^{*} \mathsf{v}' * \Phi(\mathsf{v}, \mathsf{v}')\right\}$$

$$\mathsf{list}\ \mathsf{Z} \triangleq \mu \mathsf{X}.1 + (\mathsf{Z} \times \mathsf{X}) \qquad \mathsf{f} \triangleq \mathsf{map}\ (\lambda \mathsf{x}.\mathsf{x} + 0)$$

$$\mathsf{lift} : (\mathsf{Val} \longrightarrow \mathsf{Val} \longrightarrow iProp) \longrightarrow (\mathsf{Expr} \longrightarrow \mathsf{Expr} \longrightarrow iProp)$$

$$\mathsf{lift}\ \Phi\ (\mathsf{e}, \mathsf{e}') = \mathsf{wp}\ \mathsf{e}\ \left\{\mathsf{v}.\ \exists \mathsf{v}'.\ \mathsf{e}' \longrightarrow^{*} \mathsf{v}' * \Phi(\mathsf{v}, \mathsf{v}')\right\}$$

$$\mathsf{list}\ \mathsf{Z} \triangleq \mu X.1+(\mathsf{Z} \times X) \qquad \mathsf{f} \triangleq \mathsf{map}\ (\lambda x.x + 0)$$

$$\nvdash \mathcal{V}_{int.}[\![\mathsf{list}\ \mathsf{Z} \rightarrow \mathsf{list}\ \mathsf{Z}]\!]\,(\lambda x.x, \mathsf{f})$$

$$\mathsf{lift} : (\mathsf{Val} \longrightarrow \mathsf{Val} \longrightarrow \mathit{iProp}) \longrightarrow (\mathsf{Expr} \longrightarrow \mathsf{Expr} \longrightarrow \mathit{iProp})$$

$$\mathsf{lift}\ \Phi\ (\mathsf{e}, \mathsf{e}') = \mathsf{wp}\ \mathsf{e}\ \left\{\mathsf{v}.\ \exists \mathsf{v}'.\ \mathsf{e}' \longrightarrow^{*} \mathsf{v}' * \Phi(\mathsf{v}, \mathsf{v}')\right\}$$

$$\mathsf{list}\ \mathsf{Z} \triangleq \mu X.1+(\mathsf{Z} \times X) \qquad \mathsf{f} \triangleq \mathsf{map}\ (\lambda x.x + 0)$$

$$\nvdash \mathcal{V}_{int.}[\![\mathsf{list}\ \mathsf{Z} \rightarrow \mathsf{list}\ \mathsf{Z}]\!]\,(\lambda x.x, \mathsf{f})$$

$$\forall \mathsf{v}, \mathsf{v}'.\ \mathcal{V}_{int.}[\![\mathsf{list}\ \mathsf{Z}]\!]\,(\mathsf{v}, \mathsf{v}') \nvdash \mathcal{E}_{int.}[\![\mathsf{list}\ \mathsf{Z}]\!]\,((\lambda x.x)\ \mathsf{v}, \mathsf{f}\ \mathsf{v}')$$

$$\mathsf{lift} : (\mathsf{Val} \to \mathsf{Val} \to iProp) \to (\mathsf{Expr} \to \mathsf{Expr} \to iProp)$$

$$\mathsf{lift}\ \Phi\ (\mathsf{e}, \mathsf{e}') = \mathsf{wp}\ \mathsf{e}\ \left\{\mathsf{v}.\ \exists \mathsf{v}'.\ \mathsf{e}' \to^* \mathsf{v}' * \Phi(\mathsf{v}, \mathsf{v}')\right\}$$

$$\mathsf{list}\ Z \triangleq \mu X.1+(Z \times X) \qquad \mathsf{f} \triangleq \mathsf{map}\ (\lambda x.x + 0)$$

$$\nvdash \mathcal{V}_{int.}[\![\mathsf{list}\ Z \to \mathsf{list}\ Z]\!](\lambda x.x, \mathsf{f})$$

$$\forall \mathsf{v}, \mathsf{v}'.\ \mathcal{V}_{int.}[\![\mathsf{list}\ Z]\!](\mathsf{v}, \mathsf{v}') \nvdash \mathcal{E}_{int.}[\![\mathsf{list}\ Z]\!]((\lambda x.x)\ \mathsf{v}, \mathsf{f}\ \mathsf{v}')$$

$$(\lambda x.x)\ [1, 2, 3, 4] \qquad \mathsf{f}\ [1, 2, 3, \mathsf{true}]$$

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

$$\mathsf{C}_b^{\vdash}[\mathsf{e}] \Downarrow \ \textit{iff}\ \ \mathsf{C}[\mathsf{e}] \Downarrow$$

$\Rightarrow$

$$\mathcal{V}_? : \mathit{Val} \to \mathit{Val} \to \mathit{iProp}$$

$$\begin{aligned}\mathcal{V}_?(\mathrm{v}, \mathrm{v}') = &\left(\mathrm{v} = \mathrm{injected}_1^{\mathrm{Val}}(()) * \mathrm{v}' = ()\right)\\ &\vee\left(\exists b \in \{\mathit{true}, \mathit{false}\}.\ \mathrm{v} = \mathrm{injected}_{\mathrm{B}}^{\mathrm{Val}}(\mathrm{b}) * \mathrm{v}' = \mathrm{b}\right)\\ &\vee\left(\exists z \in \mathbb{Z}.\ \mathrm{v} = \mathrm{injected}_{\mathrm{Z}}^{\mathrm{Val}}(\mathrm{z}) * \mathrm{v}' = \mathrm{z}\right)\\ &\vee\left(\exists \mathrm{w}, \mathrm{w}'.\ \bigvee_{i \in \{1,2\}} (\mathrm{v} = \mathrm{injected}_{+}^{\mathrm{Val}}(\mathrm{inj}_{\mathrm{i}}\ \mathrm{w}) * \mathrm{v}' = \mathrm{inj}_{\mathrm{i}}\ \mathrm{w}' * \triangleright \mathcal{V}_?(\mathrm{w}, \mathrm{w}'))\right)\\ &\vee\left(\exists \mathrm{v}_1, \mathrm{v}_1', \mathrm{v}_2', \mathrm{v}_2'.\ \mathrm{v} = \mathrm{injected}_{\times}^{\mathrm{Val}}((\mathrm{v}_1', \mathrm{v}_2')) * \mathrm{v}' = (\mathrm{v}_1', \mathrm{v}_2') * \triangleright \mathcal{V}_?(\mathrm{v}_1, \mathrm{v}_1') * \triangleright \mathcal{V}_?(\mathrm{v}_2, \mathrm{v}_2')\right)\\ &\vee\left(\exists \mathrm{e}.\ \mathrm{v} = \mathrm{injected}_{\to}^{\mathrm{Val}}(\lambda \mathrm{x}.\mathrm{e}) * \triangleright \square(\forall \mathrm{w}, \mathrm{w}'.\ \mathcal{V}_?(\mathrm{w}, \mathrm{w}') \mathrel{-\!\!*} \mathrm{lift}\ \mathcal{V}_?\ (\mathrm{e}[\mathrm{w}/\mathrm{x}], \mathrm{v}'\ \mathrm{w}'))\right)\\ &\vee\left(\exists \mathrm{w}, \mathrm{w}'.\ \mathrm{v} = \mathrm{injected}_{\mu}^{\mathrm{Val}}(\mathrm{fold}\ \mathrm{w}) * \mathrm{v}' = \mathrm{fold}\ \mathrm{w}' * \triangleright \mathcal{V}_?(\mathrm{w}, \mathrm{w}')\right)\end{aligned}$$

$$\frac{\mathcal{E}_?(\mathrm{e}, \mathrm{e}')}{\mathcal{E}_{\mathit{int.}}[\![\tau]\!](\mathrm{project}_\tau\ \mathrm{e}, \mathrm{e}')} \qquad \frac{\mathcal{E}_{\mathit{int.}}[\![\tau]\!](\mathrm{e}, \mathrm{e}')}{\mathcal{E}_?(\mathrm{embed}_\tau\ \mathrm{e}, \mathrm{e}')}$$

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

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

$$\mathrm{C}^{\vdash}_{b}[\mathrm{e}] \Downarrow \ \mathit{iff}\ \mathrm{C}[\mathrm{e}] \Downarrow$$

$\Leftarrow$

$$\cancel{\dfrac{\mathcal{E}_{¿}(\mathrm{e}, \mathrm{e}')}{\mathcal{E}_{int.}[\![\tau]\!](\mathrm{e}, \mathrm{project}_{\tau}\ \mathrm{e}')}} \qquad \cancel{\dfrac{\mathcal{E}_{int.}[\![\tau]\!](\mathrm{e}, \mathrm{e}')}{\mathcal{E}_{¿}(\mathrm{e}, \mathrm{embed}_{\tau}\mathrm{e}')}}$$

$$\dfrac{\mathcal{E}_{¿}(\mathrm{e}, \mathrm{e}')}{\mathcal{E}_{int.}[\![\tau]\!](\mathrm{assert}_{\tau}\ \mathrm{e}, \mathrm{project}_{\tau}\ \mathrm{e}')} \qquad \dfrac{\mathcal{E}_{int.}[\![\tau]\!](\mathrm{e}, \mathrm{e}')}{\mathcal{E}_{¿}(\mathrm{guard}_{\tau}\ \mathrm{e}, \mathrm{embed}_{\tau}\ \mathrm{e}')}$$

$$\dfrac{\mathcal{E}_{int.}[\![\tau]\!](\mathrm{e}, \mathrm{e}')}{\mathcal{E}_{int.}[\![\tau]\!](\mapstochar\dashrightarrow(\mathrm{e}), \mathrm{guard}_{\tau}\ \mathrm{e}')} \qquad \dfrac{\mathcal{E}_{int.}[\![\tau]\!](\mathrm{e}, \mathrm{e}')}{\mathcal{E}_{int.}[\![\tau]\!](\mapstochar\dashrightarrow(\mathrm{e}), \mathrm{assert}_{\tau}\ \mathrm{e}')}$$

$$\dfrac{\mathcal{E}_{int.}[\![\tau]\!](\mathrm{e}, \mathrm{e}')}{\mathcal{E}_{int.}[\![\tau]\!](\mathrm{e}, \mapstochar\dashrightarrow(\mathrm{e}'))} \qquad \dfrac{\mathcal{E}_{int.}[\![\tau]\!](\mathrm{e}, \mathrm{e}')}{\mathcal{E}_{int.}[\![\tau]\!](\mathrm{e}, \mapstochar\dashrightarrow(\mathrm{e}'))}$$