Programming Languages meets Program Verification 2014

Substructural Typestates

Filipe Militão (CMU & UNL)

Jonathan Aldrich (CMU)

Luís Caires (UNL)

Information and Communication Technologies Institute
Carnegie Mellon | PORTUGAL
AN INTERNATIONAL PARTNERSHIP

FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Motivation

```
File file = new File( “out.txt” );
file.write( “stuff” );
file.close();
file.write( “more stuff” );
```

Note: consider a simplified `File` object, similar to Java’s `FileOutputStream`.

Motivation

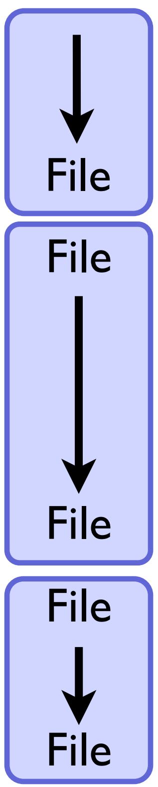
```
File file = new File( “out.txt” );
file.write( “stuff” );
file.close();
file.write( “more stuff” );
```

FAILS with *runtime* exception
(“invalid file descriptor”)

Motivation

```
class File {
    FileDescriptor fd;
    File( string filename ){
     fd = OS.createFile( filename );
    }
    void write( string s ){
     if( fd == null )
      throw Exception("invalid file descriptor");
     fd.write( s );
    }
    void close(){
     fd = null;
    }
}
```

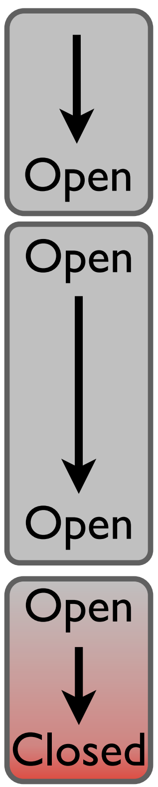
Motivation




```
class File {
    FileDescriptor fd;
    File( string filename ){
     fd = OS.createFile( filename );
    }
    void write( string s ){
     if( fd == null )
      throw Exception("invalid file descriptor");
     fd.write( s );
    }
    void close(){
     fd = null;
    }
}
```

The `File` type abstraction does not precisely express the **changing properties of** `File`'s **internal state** (*`fd`*).

Motivation




```
class File {
    FileDescriptor fd;
    File( string filename ){
     fd = OS.createFile( filename );
    }
    void write( string s ){
     if( fd == null )
      throw Exception("invalid file descriptor");
     fd.write( s );
    }
    void close(){
     fd = null;
    }
}
```

Superfluous if statically ensured to only be used when `File` is open.

Motivation

```
class File {
    FileDescriptor fd;
    File( string filename ){
     fd = OS.createFile( filename );
    }
    void write( string s ){
     if( fd == null )
      throw Exception(“invalid file descriptor”);
     fd.write( s );
    }
    void close(){
     fd = null;
    }
}
```

$\downarrow$ Open

Open $\rightarrow$ Open

Open $\rightarrow$ Closed

Superfluous if statically ensured to only be used when `File` is open.

`Open` and `Close` are **typestates**.

Contributions

1. Reconstruct **typestate** features from standard type-theoretic programming language primitives.

   We focus on the following set of **typestate** features:

   a) state abstraction, hiding an object representation while expressing the type of the state;

   b) state “dimensions”, enabling multiple orthogonal typestates over the same object;

   c) “dynamic state tests”, allowing a case analysis over the abstract state.

2. We show how to idiomatically support both *state-based* (**typestate**) and *transition-based* (**behavioral types**) specifications of abstract state evolution.

Language

- Polymorphic λ-calculus with mutable references (and immutable records, tagged sums, ...).
- Technically, we use a variant of **L³** adapted for usability (by simplifying the handling of capabilities, adding support for sum types, universal/existential type quantification, alternatives, labeled records, ...).

Ahmed, Fluet, and Morrisett. **L³: A linear language with locations**. Fundam. Inform. 2007.

Language

- Mutable state handled as a linear resource:
  - split in *pure* references and *linear* capabilities.
  - use location-dependent types to link both.

Language

- Mutable state handled as a linear resource:
  - split in *pure* references and *linear* capabilities.
  - use location-dependent types to link both.

ref A

Language

- Mutable state handled as a linear resource:
  - split in *pure* references and *linear* capabilities.
  - use location-dependent types to link both.

type of contents of cell

ref A

Language

- Mutable state handled as a linear resource:
  - split in *pure* references and *linear* capabilities.
  - use location-dependent types to link both.

type of contents of cell

ref A $\xrightarrow{\text{becomes}}$ **ref** p **rw** p A

Language

- Mutable state handled as a linear resource:
  - split in *pure* references and *linear* capabilities.
  - use location-dependent types to link both.

type of contents of cell

ref A $\xrightarrow{\text{becomes}}$ **ref** *p* **rw** *p* A

location *p* links **ref** to **r**ead+**w**rite **capability**
that tracks the contents of that cell

Language

- Mutable state handled as a linear resource:
  - split in *pure* references and *linear* capabilities.
  - use location-dependent types to link both.

type of contents of cell

ref A —becomes→ **ref** *p* **rw** *p* A

type of contents of cell *p*
(linear - cannot be duplicated)

location *p* links **ref** to **r**ead+**w**rite **capability**
that tracks the contents of that cell

Language

- Mutable state handled as a linear resource:
  - split in *pure* references and *linear* capabilities.
  - use location-dependent types to link both.

type of contents of cell

can be freely copied (pure)

type of contents of cell p
(linear - cannot be duplicated)

ref A $\xrightarrow{\text{becomes}}$ **ref** p **rw** p A

location p links **ref** to **r**ead+**w**rite **capability**
that tracks the contents of that cell

Language

- Mutable state handled as a linear resource:
  - split in *pure* references and *linear* capabilities.
  - use location-dependent types to link both.

Language

- Mutable state handled as a linear resource:
  - split in *pure* references and *linear* capabilities.
  - use location-dependent types to link both.

$y : \mathbf{ref}\ p$

$z : \mathbf{ref}\ q$ $x : \mathbf{ref}\ p$

Language

- Mutable state handled as a linear resource:
  - split in *pure* references and *linear* capabilities.
  - use location-dependent types to link both.

$\mathsf{y} : \textbf{ref}\ p$ $\quad$ $\textbf{rw}\ q\ \mathsf{B}$

$\mathsf{z} : \textbf{ref}\ q$ $\quad$ $\mathsf{x} : \textbf{ref}\ p$ $\quad$ $\textbf{rw}\ p\ \mathsf{A}$

Language

- Mutable state handled as a linear resource:
  - split in *pure* references and *linear* capabilities.
  - use location-dependent types to link both.

$y : \mathbf{ref}\ p$ $\quad$ $\mathbf{rw}\ q\ B$

$z : \mathbf{ref}\ q$ $\quad$ $x : \mathbf{ref}\ p$ $\quad$ $\mathbf{rw}\ p\ A$

$!x$
("de-reference x")

Language

- Mutable state handled as a linear resource:
  - split in *pure* references and *linear* capabilities.
  - use location-dependent types to link both.

y : **ref** *p*

z : **ref** *q*   x : **ref** *p*

rw *q* B

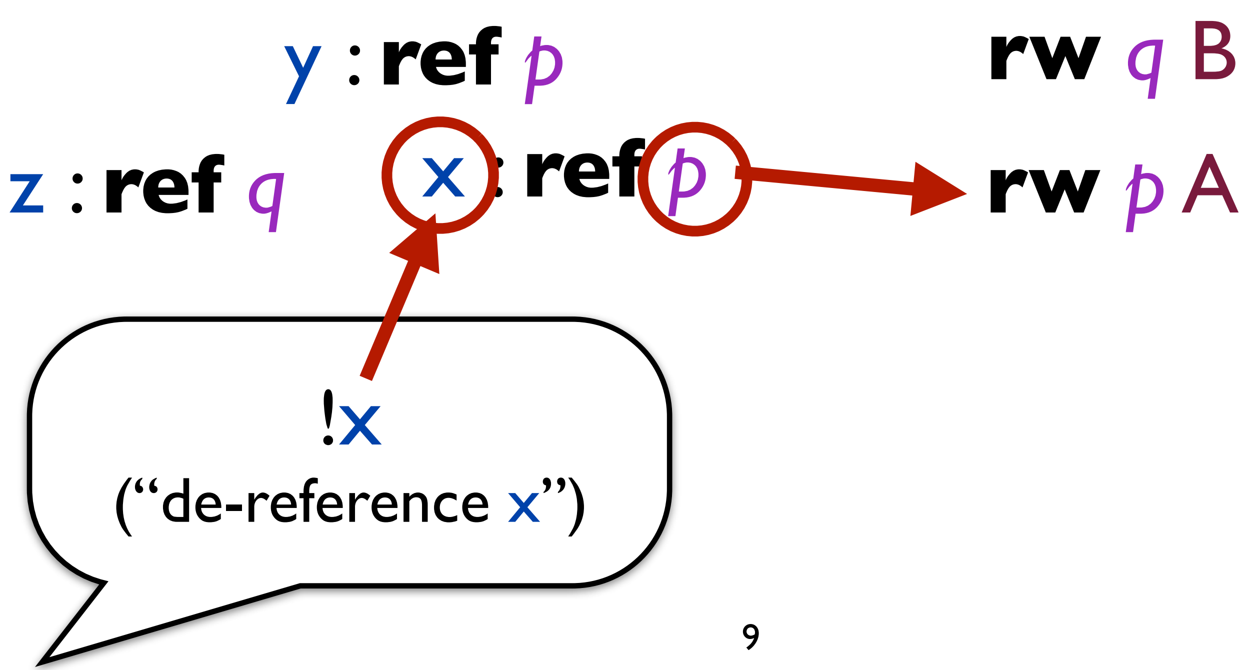
rw *p* A

!x

("de-reference x")

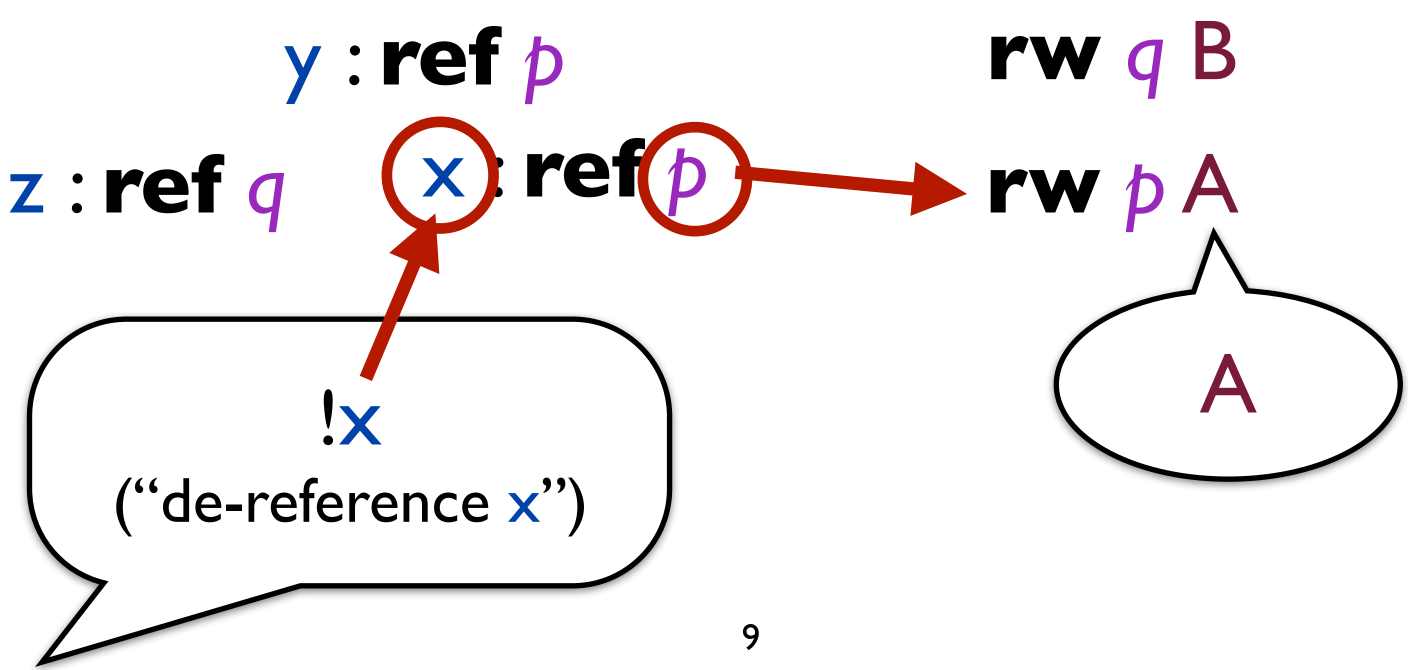
Language

- Mutable state handled as a linear resource:
  - split in *pure* references and *linear* capabilities.
  - use location-dependent types to link both.



Language

- Mutable state handled as a linear resource:
  - split in *pure* references and *linear* capabilities.
  - use location-dependent types to link both.



Language

- Capabilities are (linear) typing artifacts (not values) that are *threaded* and *stacked* implicitly.
- For that, we use a **Type**-and-**Effect** system.
- Typing judgement format:

$$\Gamma; \Delta_0 \vdash e : A \dashv \Delta_1$$

Language

- Capabilities are (linear) typing artifacts (not values) that are *threaded* and *stacked* implicitly.
- For that, we use a **Type**-and-**Effect** system.
- Typing judgement format:

Lexical Typing Environment

$$\Gamma; \Delta_0 \vdash e : A \dashv \Delta_1$$

Language

- Capabilities are (linear) typing artifacts (not values) that are *threaded* and *stacked* implicitly.
- For that, we use a **Type**-and-**Effect** system.
- Typing judgement format:

Lexical Typing Environment

$$\Gamma; \Delta_0 \vdash e : A \dashv \Delta_1$$

Initial Linear Typing Environment

Language

- Capabilities are (linear) typing artifacts (not values) that are *threaded* and *stacked* implicitly.
- For that, we use a **Type**-and-**Effect** system.
- Typing judgement format:

Lexical Typing Environment

$$\Gamma; \Delta_0 \vdash e : A \dashv \Delta_1$$

Initial Linear Typing Environment

Type of Expression

Language

- Capabilities are (linear) typing artifacts (not values) that are *threaded* and *stacked* implicitly.
- For that, we use a **Type**-and-**Effect** system.
- Typing judgement format:

Lexical Typing Environment

Resulting **Effects**

$$\Gamma; \Delta_0 \vdash e : A \dashv \Delta_1$$

Initial Linear Typing Environment

Type of Expression

Language

- Capabilities are (linear) typing artifacts (not values) that are *threaded* and *stacked* implicitly.
- For that, we use a **Type**-and-**Effect** system.
- Typing judgement format:

$$\Gamma; \Delta_0 \vdash e : A \dashv \Delta_1$$

resources are
either consumed

Language

- Capabilities are (linear) typing artifacts (not values) that are *threaded* and *stacked* implicitly.
- For that, we use a **Type**-and-**Effect** system.
- Typing judgement format:

$$\Gamma; \Delta_0 \vdash e : A \dashv \Delta_1$$

Language

- Capabilities are (linear) typing artifacts (not values) that are *threaded* and *stacked* implicitly.
- For that, we use a **Type**-and-**Effect** system.
- Typing judgement format:

$$\Gamma ; \Delta_0 \vdash e : A \dashv \Delta_1$$

or, *threaded* through

Language

- Capabilities are (linear) typing artifacts (not values) that are *threaded* and *stacked* implicitly.
- For that, we use a **Type**-and-**Effect** system.
- Typing judgement format:

$$\Gamma; \Delta_0 \vdash e : A \dashv \Delta_1$$

or, *threaded* through

Language

- Capabilities can be *stacked* and *unstacked* on top of some type, allowing them to accompany that type.

Language

- Capabilities can be *stacked* and *unstacked* on top of some type, allowing them to accompany that type.

(T:CAP-STACK)

$$\frac{\Gamma; \Delta_0 \vdash e : A_0 \dashv \Delta_1, A_1}{\Gamma; \Delta_0 \vdash e : A_0 :: A_1 \dashv \Delta_1}$$

Language

- Capabilities can be *stacked* and *unstacked* on top of some type, allowing them to accompany that type.

(T:CAP-STACK)

$$\frac{\Gamma; \Delta_0 \vdash e : A_0 \dashv \Delta_1, \boxed{A_1}}{\Gamma; \Delta_0 \vdash e : A_0 :: A_1 \dashv \Delta_1}$$

Language

- Capabilities can be *stacked* and *unstacked* on top of some type, allowing them to accompany that type.

(T:Cap-Stack)

$$\frac{\Gamma; \Delta_0 \vdash e : A_0 \dashv \Delta_1, A_1}{\Gamma; \Delta_0 \vdash e : A_0 :: \boxed{A_1} \dashv \Delta_1}$$

Language

- Capabilities can be *stacked* and *unstacked* on top of some type, allowing them to accompany that type.

(T:CAP-STACK)
$$\frac{\Gamma; \Delta_0 \vdash e : A_0 \dashv \Delta_1, A_1}{\Gamma; \Delta_0 \vdash e : A_0 :: \boxed{A_1} \dashv \Delta_1}$$

(T:CAP-UNSTACK)
$$\frac{\Gamma; \Delta_0 \vdash e : A_0 :: A_1 \dashv \Delta_1}{\Gamma; \Delta_0 \vdash e : A_0 \dashv \Delta_1, A_1}$$

Language

- Capabilities can be *stacked* and *unstacked* on top of some type, allowing them to accompany that type.

(T:CAP-STACK)
$$\frac{\Gamma; \Delta_0 \vdash e : A_0 \dashv \Delta_1, A_1}{\Gamma; \Delta_0 \vdash e : A_0 :: \boxed{A_1} \dashv \Delta_1}$$

(T:CAP-UNSTACK)
$$\frac{\Gamma; \Delta_0 \vdash e : A_0 :: \boxed{A_1} \dashv \Delta_1}{\Gamma; \Delta_0 \vdash e : A_0 \dashv \Delta_1, A_1}$$

Language

- Capabilities can be *stacked* and *unstacked* on top of some type, allowing them to accompany that type.

(T:CAP-STACK)

$$\frac{\Gamma; \Delta_0 \vdash e : A_0 \dashv \Delta_1, A_1}{\Gamma; \Delta_0 \vdash e : A_0 :: \boxed{A_1} \dashv \Delta_1}$$

(T:CAP-UNSTACK)

$$\frac{\Gamma; \Delta_0 \vdash e : A_0 :: A_1 \dashv \Delta_1}{\Gamma; \Delta_0 \vdash e : A_0 \dashv \Delta_1, \boxed{A_1}}$$

Language

- Capabilities can be *stacked* and *unstacked* on top of some type, allowing them to accompany that type.

(T:CAP-STACK)
$$\frac{\Gamma;\Delta_0 \vdash e : A_0 \dashv \Delta_1, A_1}{\Gamma;\Delta_0 \vdash e : A_0 :: \boxed{A_1} \dashv \Delta_1}$$

(T:CAP-UNSTACK)
$$\frac{\Gamma;\Delta_0 \vdash e : A_0 :: A_1 \dashv \Delta_1}{\Gamma;\Delta_0 \vdash e : A_0 \dashv \Delta_1, \boxed{A_1}}$$

(T:CAP-ELIM)
$$\frac{\Gamma;\Delta_0, x : A_0, A_1 \vdash e : A_2 \dashv \Delta_1}{\Gamma;\Delta_0, x : A_0 :: A_1 \vdash e : A_2 \dashv \Delta_1}$$

Types

$$
\begin{array}{lcll}
A & ::= & !A & \text{(pure/persistent)} \\
 & | & A \multimap A & \text{(linear function)} \\
 & | & A :: A & \text{(stacking)} \\
 & | & A * A & \text{(separation)} \\
 & | & X & \text{(type variable)} \\
 & | & \forall X.A & \text{(universal type quantification)} \\
 & | & \exists X.A & \text{(existential type quantification)} \\
 & | & [\overline{\mathtt{f} : A}] & \text{(record)} \\
 & | & \forall t.A & \text{(universal location quantification)} \\
 & | & \exists t.A & \text{(existential location quantification)} \\
 & | & \mathbf{ref}\ p & \text{(reference type)} \\
 & | & \mathbf{rec}\ X.A & \text{(recursive type)} \\
 & | & \sum_i \mathtt{l}_i \# A_i & \text{(tagged sum)} \\
 & | & A \oplus A & \text{(alternative)} \\
 & | & \mathbf{rw}\ p\ A & \text{(read-write capability to } p\text{)} \\
 & | & \textsf{none} & \text{(empty capability)}
\end{array}
$$

Syntax

$v \in$ VALUES	$::=$	ρ	(address)
	$\mid$	x	(variable)
	$\mid$	$\mathsf{fun}(x : A).e$	(function)
	$\mid$	$\langle t \rangle\, e$	(universal location)
	$\mid$	$\langle X \rangle\, e$	(universal type)
	$\mid$	$\langle p, v \rangle$	(pack location)
	$\mid$	$\langle A, v \rangle$	(pack type)
	$\mid$	$\{\overline{\mathtt{f} = v}\}$	(record)
	$\mid$	$\mathtt{l}\#v$	(tagged value)
$e \in$ EXPRS.	$::=$	v	(value)
	$\mid$	$v[p]$	(location application)
	$\mid$	$v[A]$	(type application)
	$\mid$	$v.\mathtt{f}$	(field)
	$\mid$	$v\ v$	(application)
	$\mid$	$\mathsf{let}\ x = e\ \mathsf{in}\ e\ \mathsf{end}$	(let)
	$\mid$	$\mathsf{open}\ \langle t, x \rangle = v\ \mathsf{in}\ e\ \mathsf{end}$	(open location)
	$\mid$	$\mathsf{open}\ \langle X, x \rangle = v\ \mathsf{in}\ e\ \mathsf{end}$	(open type)
	$\mid$	$\mathsf{new}\ v$	(cell creation)
	$\mid$	$\mathsf{delete}\ v$	(cell deletion)
	$\mid$	$!v$	(dereference)
	$\mid$	$v := v$	(assign)
	$\mid$	$\mathsf{case}\ v\ \mathsf{of}\ \overline{\mathtt{l}\#x \rightarrow e}\ \mathsf{end}$	(case)

Syntax

$v \in$ VALUES	$::=$	ρ	(address)
	$\mid$	x	(variable)
	$\mid$	$\mathsf{fun}(x : A).e$	(function)
	$\mid$	$\langle t \rangle\, e$	(universal location)
	$\mid$	$\langle X \rangle\, e$	(universal type)
	$\mid$	$\langle p, v \rangle$	(pack location)
	$\mid$	$\langle A, v \rangle$	(pack type)
	$\mid$	$\{\overline{\mathtt{f} = v}\}$	(record)
	$\mid$	$\mathtt{l}\#v$	(tagged value)
$e \in$ EXPRS.	$::=$	v	(value)
	$\mid$	$v[p]$	(location application)
	$\mid$	$v[A]$	(type application)
	$\mid$	$v.\mathtt{f}$	(field)
	$\mid$	$v\ v$	(application)
	$\mid$	$\mathsf{let}\ x = e\ \mathsf{in}\ e\ \mathsf{end}$	(let)
	$\mid$	$\mathsf{open}\ \langle t, x \rangle = v\ \mathsf{in}\ e\ \mathsf{end}$	(open location)
	$\mid$	$\mathsf{open}\ \langle X, x \rangle = v\ \mathsf{in}\ e\ \mathsf{end}$	(open type)
	$\mid$	$\mathsf{new}\ v$	(cell creation)
	$\mid$	$\mathsf{delete}\ v$	(cell deletion)
	$\mid$	$!v$	(dereference)
	$\mid$	$v := v$	(assign)
	$\mid$	$\mathsf{case}\ v\ \mathsf{of}\ \overline{\mathtt{l}\#x \to e}\ \mathsf{end}$	(case)

let-expanded

Syntax

$v \in \textsc{Values}$	$::=$	ρ	(address)
	$\mid$	x	(variable)
	$\mid$	$\mathsf{fun}(x : A).e$	(function)
	$\mid$	$\boxed{\langle t \rangle\, e}$	(universal location)
	$\mid$	$\langle X \rangle\, e$	(universal type)
	$\mid$	$\boxed{\langle p, v \rangle}$	(pack location)
	$\mid$	$\langle A, v \rangle$	(pack type)
	$\mid$	$\{\overline{\mathtt{f} = v}\}$	(record)
	$\mid$	$\mathtt{l}\#v$	(tagged value)
$e \in \textsc{Exprs.}$	$::=$	v	(value)
	$\mid$	$\boxed{v[p]}$	(location application)
	$\mid$	$v[A]$	(type application)
	$\mid$	$v.\mathtt{f}$	(field)
	$\mid$	$v\ v$	(application)
	$\mid$	$\mathsf{let}\ x = e\ \mathsf{in}\ e\ \mathsf{end}$	(let)
	$\mid$	$\boxed{\mathsf{open}\ \langle t, x \rangle = v\ \mathsf{in}\ e\ \mathsf{end}}$	(open location)
	$\mid$	$\mathsf{open}\ \langle X, x \rangle = v\ \mathsf{in}\ e\ \mathsf{end}$	(open type)
	$\mid$	$\mathsf{new}\ v$	(cell creation)
	$\mid$	$\mathsf{delete}\ v$	(cell deletion)
	$\mid$	$!v$	(dereference)
	$\mid$	$v := v$	(assign)
	$\mid$	$\mathsf{case}\ v\ \mathsf{of}\ \overline{\mathtt{l}\#x \rightarrow e}\ \mathsf{end}$	(case)

let-expanded

Pair Example

- Function that creates stateful Pair objects.
- The Pair's components (left and right) are private, not accessible to clients.
- The state of Pair is changed indirectly by calling functions contained in a labeled record (which are technically closures).

```
let newPair = fun( _ : [] ).
 open <pl,l> = new {} in
 open <pr,r> = new {} in
  {
   initL = fun( i : int :: rw pl [] ). l := i,
   initR = fun( i : int :: rw pr [] ). r := i,
   sum = fun( _ : [] :: rw pl int * rw pr int ). !l+!r,
   destroy = fun( _ : [] :: rw pl int * rw pr int ).
                            delete l; delete r
  }
 end
 end
```

```
let newPair = fun( _ : [] ).
 open <pl,l> = new {} in
 open <pr,r> = new {} in
  {
   initL = fun( i : int :: rw pl [] ). l := i,
   initR = fun( i : int :: rw pr [] ). r := i,
   sum = fun( _ : [] :: rw pl int * rw pr int ). !l+!r,
   destroy = fun( _ : [] :: rw pl int * rw pr int ).
                    delete l; delete r
  }
 end
 end
```

```
let newPair = fun( _ : [] ).
 open <pl,l> = new {} in
 open <pr,r> = new {} in
  {
   initL = fun( i : int :: rw pl [] ). l := i,
   initR = fun( i : int :: rw pr [] ). r := i,
   sum = fun( _ : [] :: rw pl int * rw pr int ). !l+!r,
   destroy = fun( _ : [] :: rw pl int * rw pr int ).
                    delete l; delete r
  }
 end
 end
```

(T:NEW)

$$\frac{\Gamma ; \Delta_0 \vdash v : A \dashv \Delta_1}{\Gamma ; \Delta_0 \vdash \mathsf{new}\ v : \exists t.(\mathbf{ref}\ t :: \mathbf{rw}\ t\ A) \dashv \Delta_1}$$

(T:LOC-OPEN)

$$\frac{\Gamma ; \Delta_0 \vdash v : \exists t.A_0 \dashv \Delta_1 \qquad \Gamma, t : \mathbf{loc} ; \Delta_1, x : A_0 \vdash e : A_1 \dashv \Delta_2}{\Gamma ; \Delta_0 \vdash \mathsf{open}\ \langle t, x \rangle = v\ \mathsf{in}\ e\ \mathsf{end} : A_1 \dashv \Delta_2}$$

```
let newPair = fun( _ : [] ).
 open <pl,l> = new {} in
 open <pr,r> = new {} in
  {
   initL = fun( i : int :: rw pl [] ). l := i,
   initR = fun( i : int :: rw pr [] ). r := i,
   sum = fun( _ : [] :: rw pl int * rw pr int ). !l+!r,
   destroy = fun( _ : [] :: rw pl int * rw pr int ).
                        delete l; delete r
  }
 end
 end
```

Γ = pl : loc, l : ref pl
Δ = rw pl []

(T:NEW)

$$\frac{\Gamma;\Delta_0 \vdash v : A \dashv \Delta_1}{\Gamma;\Delta_0 \vdash \mathsf{new}\ v : \exists t.(\mathbf{ref}\ t :: \mathbf{rw}\ t\ A) \dashv \Delta_1}$$

(T:LOC-OPEN)

$$\frac{\Gamma;\Delta_0 \vdash v : \exists t.A_0 \dashv \Delta_1 \qquad \Gamma, t : \mathbf{loc};\Delta_1, x : A_0 \vdash e : A_1 \dashv \Delta_2}{\Gamma;\Delta_0 \vdash \mathsf{open}\ \langle t, x\rangle = v\ \mathsf{in}\ e\ \mathsf{end} : A_1 \dashv \Delta_2}$$

```
let newPair = fun( _ : [] ).
 open <pl,l> = new {} in
 open <pr,r> = new {} in
  {
   initL = fun( i : int :: rw pl [] ). l := i,
   initR = fun( i : int :: rw pr [] ). r := i,
   sum = fun( _ : [] :: rw pl int * rw pr int ). !l+!r,
   destroy = fun( _ : [] :: rw pl int * rw pr int ).
                        delete l; delete r
  }
 end
 end
```

```
Γ = pl : loc, l : ref pl,
    pr : loc, r : ref pr
Δ = rw pl [], rw pr []
```

(T:NEW)

$$\frac{\Gamma;\Delta_0 \vdash v : A \dashv \Delta_1}{\Gamma;\Delta_0 \vdash \mathsf{new}\ v : \exists t.(\mathbf{ref}\ t :: \mathbf{rw}\ t\ A) \dashv \Delta_1}$$

(T:LOC-OPEN)

$$\frac{\Gamma;\Delta_0 \vdash v : \exists t.A_0 \dashv \Delta_1 \quad \Gamma, t : \mathbf{loc};\Delta_1, x : A_0 \vdash e : A_1 \dashv \Delta_2}{\Gamma;\Delta_0 \vdash \mathsf{open}\ \langle t, x\rangle = v\ \mathsf{in}\ e\ \mathsf{end} : A_1 \dashv \Delta_2}$$

```
let newPair = fun( _ : [] ).
 open <pl,l> = new {} in
 open <pr,r> = new {} in
  {
   initL = fun( i : int :: rw pl [] ). l := i,
   initR = fun( i : int :: rw pr [] ). r := i,
   sum = fun( _ : [] :: rw pl int * rw pr int ). !l+!r,
   destroy = fun( _ : [] :: rw pl int * rw pr int ).
                       delete l; delete r
  }
 end
 end
```

```
Γ = pl : loc, l : ref pl,
    pr : loc, r : ref pr
Δ = rw pl [], rw pr []
```

(T:FUNCTION)

$$\frac{\Gamma; \Delta, x : A_0 \vdash e : A_1 \dashv \cdot}{\Gamma; \Delta \vdash \mathsf{fun}(x : A_0).e : A_0 \multimap A_1 \dashv \cdot}$$

(T:RECORD)

$$\frac{\overline{\Gamma; \Delta \vdash v : A \dashv \cdot}}{\Gamma; \Delta \vdash \{\overline{\mathtt{f} = v}\} : [\overline{\mathtt{f} : A}] \dashv \cdot}$$

```
let newPair = fun( _ : [] ).
 open <pl,l> = new {} in
 open <pr,r> = new {} in
  {
   initL = fun( i : int :: rw pl [] ). l := i,
   initR = fun( i : int :: rw pr [] ). r := i,
   sum = fun( _ : [] :: rw pl int * rw pr int ). !l+!r,
   destroy = fun( _ : [] :: rw pl int * rw pr int ).
                    delete l; delete r
  }
 end
 end
```

```
let newPair = fun( _ : [] ).
 open <pl,l> = new {} in
 open <pr,r> = new {} in
  {
   initL = fun( i : int :: rw pl [] ). l := i,
   initR = fun( i : int :: rw pr [] ). r := i,
   sum = fun( _ : [] :: rw pl int * rw pr int ). !l+!r,
   destroy = fun( _ : [] :: rw pl int * rw pr int ).
                        delete l; delete r
  }
 end
 end
```

(T:FUNCTION)

$$\frac{\Gamma; \Delta, x : A_0 \vdash e : A_1 \dashv \cdot}{\Gamma; \Delta \vdash \mathsf{fun}(x : A_0).e : A_0 \multimap A_1 \dashv \cdot}$$

(T:CAP-STACK)

$$\frac{\Gamma; \Delta_0 \vdash e : A_0 \dashv \Delta_1, A_1}{\Gamma; \Delta_0 \vdash e : A_0 :: A_1 \dashv \Delta_1}$$

```
let newPair = fun( _ : [] ).
 open <pl,l> = new {} i
 open <pr,r> = new {} i
  {
   initL = fun( i : int :: rw pl [] ). l := i,
   initR = fun( i : int :: rw pr [] ). r := i,
   sum = fun( _ : [] :: rw pl int * rw pr int ). !l+!r,
   destroy = fun( _ : [] :: rw pl int * rw pr int ).
                    delete l; delete r
  }
 end
 end
```

Δ = rw *pl* []

(T:FUNCTION)

$$\frac{\Gamma; \Delta, x : A_0 \vdash e : A_1 \dashv \cdot}{\Gamma; \Delta \vdash \mathsf{fun}(x : A_0).e : A_0 \multimap A_1 \dashv \cdot}$$

(T:CAP-STACK)

$$\frac{\Gamma; \Delta_0 \vdash e : A_0 \dashv \Delta_1, A_1}{\Gamma; \Delta_0 \vdash e : A_0 :: A_1 \dashv \Delta_1}$$

```
let newPair = fun( _ : [] ).
 open <pl,l> = new {} in
 open <pr,r> = new {} in
  {
   initL = fun( i : int :: rw pl [] ). l := i,
   initR = fun( i : int :: rw pr [] ). r := i,
   sum = fun( _ : [] :: rw pl int * rw pr int ). !l+!r,
   destroy = fun( _ : [] :: rw pl int * rw pr int ).
                     delete l; delete r
  }
 end
 end
```

Δ = rw *pl* int

(T:FUNCTION)

$$\frac{\Gamma; \Delta, x : A_0 \vdash e : A_1 \dashv \cdot}{\Gamma; \Delta \vdash \mathsf{fun}(x : A_0).e : A_0 \multimap A_1 \dashv \cdot}$$

(T:CAP-STACK)

$$\frac{\Gamma; \Delta_0 \vdash e : A_0 \dashv \Delta_1, A_1}{\Gamma; \Delta_0 \vdash e : A_0 :: A_1 \dashv \Delta_1}$$

```
let newPair = fun( _ : [] ).
 open <pl,l> = new {} in
 open <pr,r> = new {} in
  {
   initL = fun( i : int :: rw pl [] ). l := i,
   initR = 
   sum = fu                                                    ,
   destroy = fun( _ : [] :: rw pl int * rw pr int ).
                  delete l; delete r
  }
 end
end
```

```
!( ( int :: rw pl [] ) ⊸ ( [] :: rw pl int ) )
```

(T:FUNCTION)

$$\frac{\Gamma; \Delta, x : A_0 \vdash e : A_1 \dashv \cdot}{\Gamma; \Delta \vdash \mathsf{fun}(x : A_0).e : A_0 \multimap A_1 \dashv \cdot}$$

(T:CAP-STACK)

$$\frac{\Gamma; \Delta_0 \vdash e : A_0 \dashv \Delta_1, A_1}{\Gamma; \Delta_0 \vdash e : A_0 :: A_1 \dashv \Delta_1}$$

(T:PURE)

$$\frac{\Gamma; \cdot \vdash v : A \dashv \cdot}{\Gamma; \cdot \vdash v : !A \dashv \cdot}$$

```
let newPair = fun( _ : [] ).
 open <pl,l> = new {} in
 open <pr,r> = new {} in
  {
   initL = fun( i : int :: rw pl [] ). l := i,
   initR =
   sum = fu                                                  ,
   destroy = fun( _ : [] :: rw pl int * rw pr int ).
                  delete l; delete r
  }
 end
 end
```

```
!( ( int :: rw pl [] ) ⊸ ( [] :: rw pl int ) )
```

(T:FUNCTION)

$$\frac{\Gamma; \Delta, x : A_0 \vdash e : A_1 \dashv \cdot}{\Gamma; \Delta \vdash \mathsf{fun}(x : A_0).e : A_0 \multimap A_1 \dashv \cdot}$$

(T:CAP-STACK)

$$\frac{\Gamma; \Delta_0 \vdash e : A_0 \dashv \Delta_1, A_1}{\Gamma; \Delta_0 \vdash e : A_0 :: A_1 \dashv \Delta_1}$$

```
let newPair = fun( _ : [] ).
 open <pl,l> = new {} in
 open <pr,r> = new {} in
  {
   initL = fun( i : int :: rw pl [] ). l := i,
   initR = fun( i : int :: rw pr [] ). r := i,
   sum = fun( _ : [] :: rw pl int * rw pr int ). !l+!r,
   destroy = fun( _ : [] :: rw pl int * rw pr int ).
                    delete l; delete r
  }
 end
 end
```

```
[
   initL : !( ( int :: rw pl [] ) ⊸ ( [] :: rw pl int ) ),
   initR : !( ( int :: rw pr [] ) ⊸ ( [] :: rw pr int ) ),
     sum : !( ( [] :: rw pl int * rw pr int ) ⊸
                   ( int :: rw pl int * rw pr int ) ),
 destroy : !( ( [] :: rw pl int * rw pr int ) ⊸ [] )
]
```

```
let newPair = fun( _ : [] ).
 open <pl,l> = new {} in
 open <pr,r> = new {} in
  {
   initL = fun( i : int :: rw pl [] ). l := i,
   initR = fun( i : int :: rw pr [] ). r := i,
   sum = fun( _ : [] :: rw pl int * rw pr int ). !l+!r,
   destroy = fun( _ : [] :: rw pl int * rw pr int ).
                         delete l; delete r
  }
 end
 end
```

Δ = rw pl [], rw pr []

```
[
   initL : !( ( int :: rw pl [] ) ⊸ ( [] :: rw pl int ) ),
   initR : !( ( int :: rw pr [] ) ⊸ ( [] :: rw pr int ) ),
     sum : !( ( [] :: rw pl int * rw pr int ) ⊸
                          ( int :: rw pl int * rw pr int ) ),
 destroy : !( ( [] :: rw pl int * rw pr int ) ⊸ [] )
]
```

```
let newPair = fun( _ : [] ).
 open <pl,l> = new {} in
 open <pr,r> = new {} in
  {
   initL = fun( i : int :: rw pl [] ). l := i,
   initR = fun( i : int :: rw pr [] ). r := i,
   sum = fun( _ : [] :: rw pl int * rw pr int ). !l+!r,
   destroy = fun( _ : [] :: rw pl int * rw pr int ).
                        delete l; delete r
  }
 end
 end
```

```
[
   initL : !( ( int :: rw pl [] ) ⊸ ( [] :: rw pl int ) ),
   initR : !( ( int :: rw pr [] ) ⊸ ( [] :: rw pr int ) ),
     sum : !( ( [] :: rw pl int * rw pr int ) ⊸
                         ( int :: rw pl int * rw pr int ) ),
 destroy : !( ( [] :: rw pl int * rw pr int ) ⊸ [] )
] :: ( rw pl [] * rw pr [] )
```

```
let newPair = fun( _ : [] ).
 open <pl,l> = new {} in
 open <pr,r> = new {} in
  <pl, <pr,{
   initL = fun( i : int :: rw pl [] ). l := i,
   initR = fun( i : int :: rw pr [] ). r := i,
   sum = fun( _ : [] :: rw pl int * rw pr int ). !l+!r,
   destroy = fun( _ : [] :: rw pl int * rw pr int ).
                      delete l; delete r
  }> >
 end
 end
```

```
∃ll.∃lr.( [
  initL : !( ( int :: rw ll [] ) ⊸ ( [] :: rw ll int ) ),
  initR : !( ( int :: rw lr [] ) ⊸ ( [] :: rw lr int ) ),
    sum : !( ( [] :: rw ll int * rw lr int ) ⊸
                 (int :: rw ll int * rw lr int ) ),
destroy : !( ( [] :: rw ll int * rw lr int ) ⊸ [] )
] :: ( rw ll [] * rw lr [] ) )
```

```
let newPair = fun( _ : [] ).
 open <pl,l> = new {} in
 open <pr,r> = new {} in
  <pl, <pr,{
   initL = fun( i : int :: rw pl [] ). l := i,
   initR = fun( i : int :: rw pr [] ). r := i,
   sum = fun( _ : [] :: rw pl int * rw pr int ). !l+!r,
   destroy = fun( _ : [] :: rw pl int * rw pr int ).
                           delete l; delete r
  }> >
 end
 end
```

```
∃ll.∃lr.( [
   initL : !( ( int :: rw ll [] ) ⊸ ( [] :: rw ll int ) ),
   initR : !( ( int :: rw lr [] ) ⊸ ( [] :: rw lr int ) ),
     sum : !( ( [] :: rw ll int * rw lr int ) ⊸
                              ( int :: rw ll int * rw lr int ) ),
     roy : !( ( [] :: rw ll int * rw lr int ) ⊸ [] )
   ( rw ll [] * rw lr [] ) )
```

The object's representation is exposed!

```
let newPair = fun( _ : [] ).
 open <pl,l> = new {} in
 open <pr,r> = new {} in
  < rw pl [], < rw pl int, < rw pr [], < rw pr int {
   initL = fun( i : int :: rw pl [] ). l := i,
   initR = fun( i : int :: rw pr [] ). r := i,
   sum = fun( _ : [] :: rw pl int * rw pr int ). !l+!r,
   destroy = fun( _ : [] :: rw pl int * rw pr int ).
                         delete l; delete r
  }> > > >
 end
 end
```

```
∃EL.∃L.∃ER.∃R.( [
      initL : !( int :: EL      ⊸   [] :: L ),
      initR : !( int :: ER      ⊸   [] :: R ),
        sum : !(  [] :: L * R   ⊸ int :: L * R ),
    destroy : !(  [] :: L * R   ⊸   [] )
  ] :: EL * ER )
```

Pair Typestate

```
newPair :
 !( [] ⊸  ∃EL.∃L.∃ER.∃R.([
          initL : !( int :: EL      ⊸   [] :: L ),
          initR : !( int :: ER      ⊸   [] :: R ),
            sum : !(  [] :: L * R   ⊸  int :: L * R ),
        destroy : !(  [] :: L * R   ⊸   [] )
        ] :: EL * ER ) )
```

- Type expresses the changing properties of the object's state, **typestate** (*E*mpty*L*eft, *L*eft, *E*mpty*R*ight and *R*ight).
- Orthogonal typestates, "state dimensions" (*EL*/*L* and *ER*/*R*), correlate to separate internal state that operates independently.

Stack Typestate

- Type of a function (polymorphic in the contents to be stored in the stack) that creates stack objects.
- Each stack has two states: **E**mpty and **N**on**E**mpty.
- Imprecision in the exact state of the stack is typed with **E**$\oplus$**NE** (alternative): we either have the **E** typestate or **NE** the typestate.

Stack Typestate

- Type of a function (polymorphic in the contents to be stored in the stack) that creates stack objects.
- Each stack has two states: **E**mpty and **N**on**E**mpty.
- Imprecision in the exact state of the stack is typed with **E**$\oplus$**NE** (alternative): we either have the **E** typestate or **NE** the typestate.

Typestates do not exist at runtime. How can client code distinguish between different states without breaking the abstraction?

```
newStack :
  ∀T.( [] ⊸
    ∃E.∃NE.[
         push :  T :: E⊕NE ⊸ [] :: NE,
          pop : [] :: NE    ⊸ T :: E⊕NE,
      isEmpty : [] :: E⊕NE ⊸ Empty#([]::E) + NonEmpty#([]::NE),
          del : [] :: E     ⊸ []
    ] :: E )
```

Note: !‘s omitted from the type for brevity.

```
newStack :
  ∀T.( [] ⊸
    ∃E.∃NE.[
        push :  T :: E⊕NE ⊸ [] :: NE,
         pop : [] :: NE    ⊸ T :: E⊕NE,
     isEmpty : [] :: E⊕NE ⊸ Empty#([]::E) + NonEmpty#([]::NE),
         del : [] :: E      ⊸ []
    ] :: E )
```

Note: !'s omitted from the type for brevity.

```
newStack :
  ∀T.( [] ⊸
    ∃E.∃NE.[
        push :  T :: E⊕NE ⊸ [] :: NE,
         pop : [] :: NE    ⊸ T :: E⊕NE,
     isEmpty : [] :: E⊕NE ⊸ Empty#([]::E) + NonEmpty#([]::NE),
         del : [] :: E      ⊸ []
    ] :: E )
```

Clients can use case analysis to determine precisely in which state the stack is at, "dynamic state test", anchoring values to the abstract stack states.

Note: !'s omitted from the type for brevity.

Contributions

1. Reconstruct typestate features from standard type-theoretic programming language primitives.

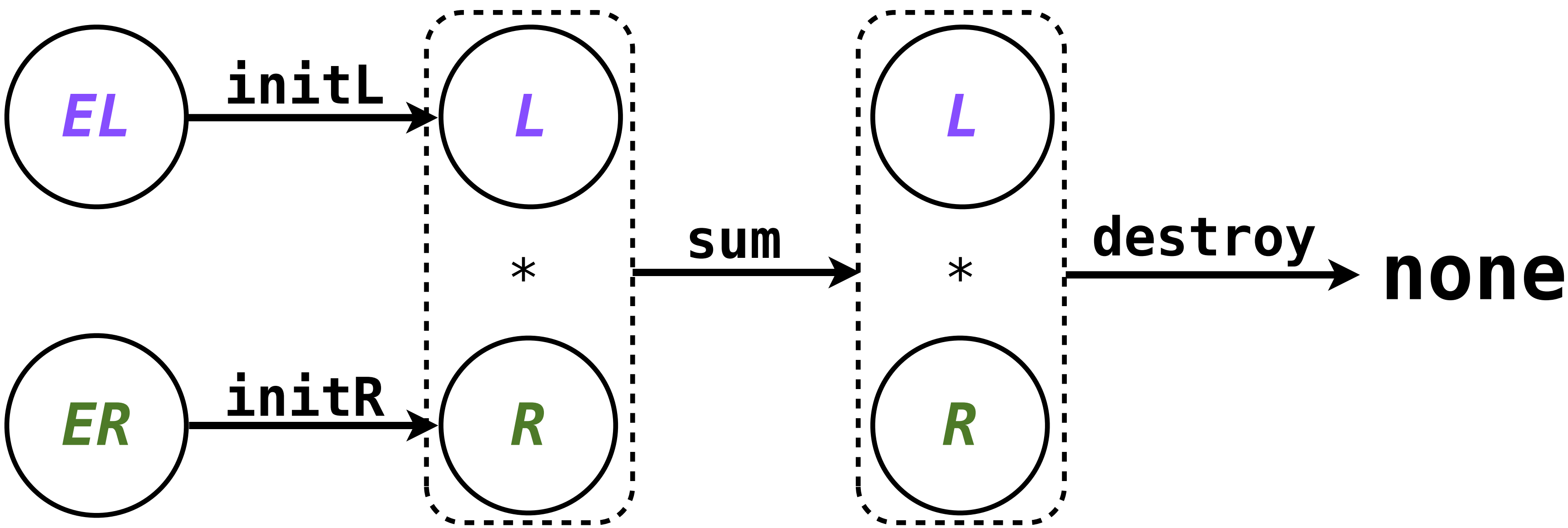
   We focus on the following set of **typestate** features:

   a) state abstraction, hiding an object representation while expressing the type of the state;
   b) state "dimensions", enabling multiple orthogonal typestates over the same object;
   c) "dynamic state tests", allowing a case analysis over the abstract state.

2. We show how to idiomatically support both *state-based* (**typestate**) and *transition-based* (**behavioral types**) specifications of abstract state evolution.

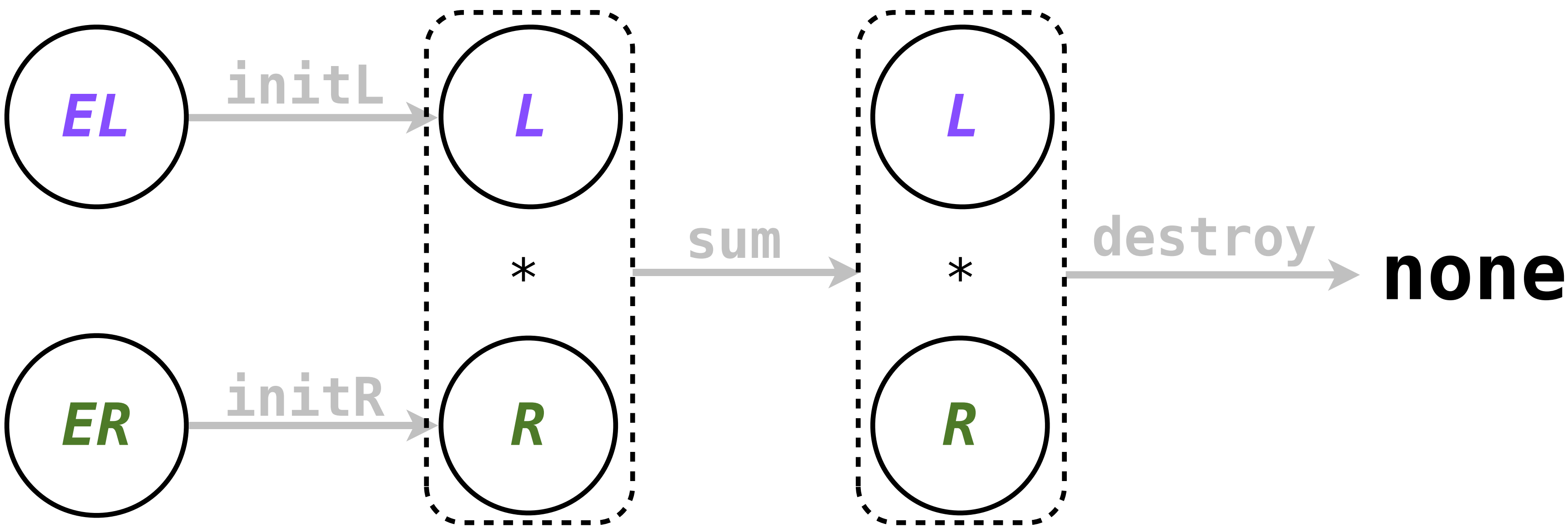
Back to Pair...

The evolution of the abstract state can be specified using a state-machine/automaton/protocol.



Back to Pair...

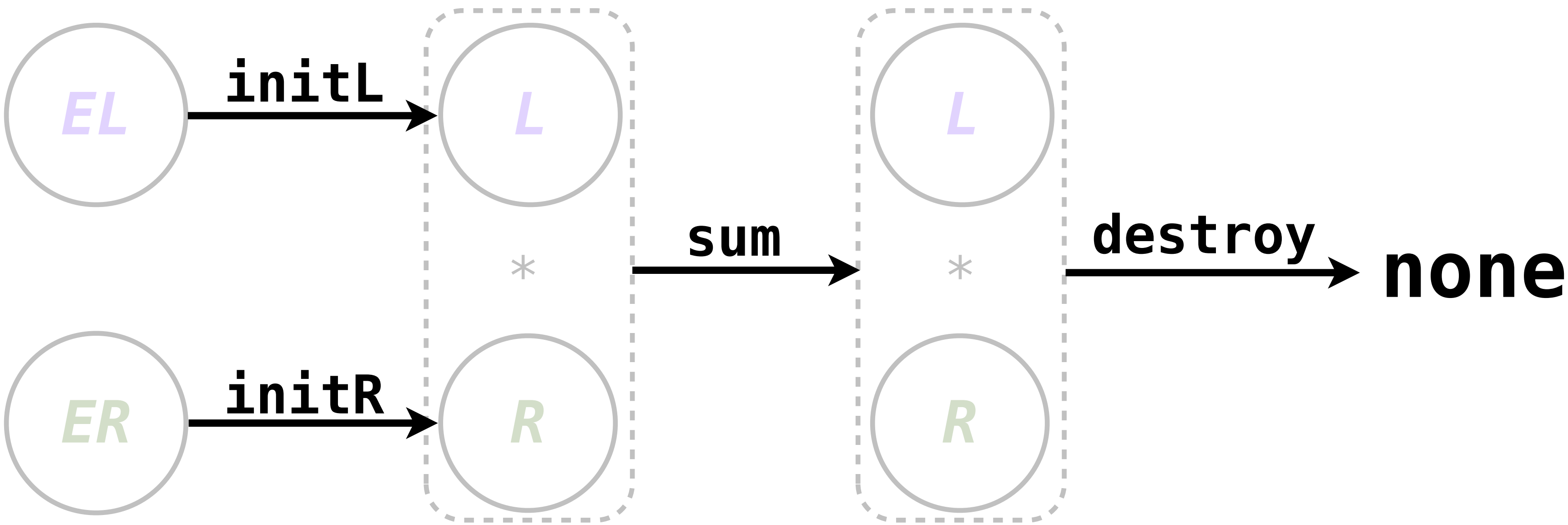
The evolution of the abstract state can be specified using a state-machine/automaton/protocol.



Typestates focus on the *states* that model the abstracted changes of the mutable state.

Back to Pair...

The evolution of the abstract state can be specified using a state-machine/automaton/protocol.



Behavioral Types focus on the *transitions* ("behavior") keeping the states anonymous.

Abstracting and Hiding State

- In our system, the notion of **typestates** is related to state **abstraction**, while the notion of **behavior** is related to **hiding** state.
- With typestates, states are named which can be convenient when there are multiple paths through the protocol.
- With behavioral types, states are implicit which simplifies descriptions of linear usages and makes it easier to provide structural equivalences.

Caires and Seco. **The type discipline of behavioral separation**. POPL 2013.

Abstracting and Hiding State

- We have already seen how to model **typestates** through standard existential abstraction.
- Interestingly, the notion of "**behavior**" can be modeled with what was already shown!
- However, it requires using an idiom to *capture* the typestate inside a function effectively hiding it.

Borrowing and Capturing

- A typestate can be *borrowed* by a function if that function requires the typestate as an argument but the function returns the typestate as a result.

```
initL : !( int :: EL ⊸ [] :: L )
```

Borrowing and Capturing

- Alternatively, a function may depend on state that was *captured* from the enclosing linear environment (similar to a closure, but with state).

(T:FUNCTION)

$$\frac{\Gamma; \Delta, x : A_0 \vdash e : A_1 \dashv \cdot}{\Gamma; \Delta \vdash \mathsf{fun}(x : A_0).e : A_0 \multimap A_1 \dashv \cdot}$$

Borrowing and Capturing

- Alternatively, a function may depend on state that was *captured* from the enclosing linear environment (similar to a closure, but with state).

```
fun( x : int ).(initL x)
```

(T:FUNCTION)

$$\frac{\Gamma; \Delta, x : A_0 \vdash e : A_1 \dashv \cdot}{\Gamma; \Delta \vdash \mathsf{fun}(x : A_0).e : A_0 \multimap A_1 \dashv \cdot}$$

Borrowing and Capturing

- Alternatively, a function may depend on state that was *captured* from the enclosing linear environment (similar to a closure, but with state).

```
Γ = initL : !( int :: EL ⊸ [] :: L )
```

```
fun( x : int ).(initL x)
```

(T:FUNCTION)

$$\frac{\Gamma; \Delta, x : A_0 \vdash e : A_1 \dashv \cdot}{\Gamma; \Delta \vdash \mathsf{fun}(x : A_0).e : A_0 \multimap A_1 \dashv \cdot}$$

Borrowing and Capturing

- Alternatively, a function may depend on state that was *captured* from the enclosing linear environment (similar to a closure, but with state).

```
Γ = initL : !( int :: EL ⊸ [] :: L )
```

```
Δ = EL    fun( x : int ).(initL x)
```

(T:FUNCTION)

$$\frac{\Gamma; \Delta, x : A_0 \vdash e : A_1 \dashv \cdot}{\Gamma; \Delta \vdash \mathsf{fun}(x : A_0).e : A_0 \multimap A_1 \dashv \cdot}$$

Borrowing and Capturing

- Alternatively, a function may depend on state that was *captured* from the enclosing linear environment (similar to a closure, but with state).

```
Γ = initL : !( int :: EL ⊸ [] :: L )
```

```
Δ = EL    fun( x : int ).(initL x)
```

(T:FUNCTION)

$$\frac{\Gamma; \Delta, x : A_0 \vdash e : A_1 \dashv \cdot}{\Gamma; \Delta \vdash \mathsf{fun}(x : A_0).e : A_0 \multimap A_1 \dashv \cdot}$$

(T:CAP-STACK)

$$\frac{\Gamma; \Delta_0 \vdash e : A_0 \dashv \Delta_1, A_1}{\Gamma; \Delta_0 \vdash e : A_0 :: A_1 \dashv \Delta_1}$$

Borrowing and Capturing

- Alternatively, a function may depend on state that was *captured* from the enclosing linear environment (similar to a closure, but with state).

```
Γ = initL : !( int :: EL ⊸ [] :: L )
```

```
Δ = EL   fun( x : int ).(initL x)   Δ = .
```

(T:FUNCTION)

$$\frac{\Gamma; \Delta, x : A_0 \vdash e : A_1 \dashv \cdot}{\Gamma; \Delta \vdash \mathsf{fun}(x : A_0).e : A_0 \multimap A_1 \dashv \cdot}$$

(T:CAP-STACK)

$$\frac{\Gamma; \Delta_0 \vdash e : A_0 \dashv \Delta_1, A_1}{\Gamma; \Delta_0 \vdash e : A_0 :: A_1 \dashv \Delta_1}$$

Borrowing and Capturing

- Alternatively, a function may depend on state that was *captured* from the enclosing linear environment (similar to a closure, but with state).

```
Γ = initL : !( int :: EL ⊸ [] :: L )
```

```
Δ = EL   fun( x : int ).(initL x)   Δ = .
```

```
int ⊸ [] :: L
```

(T:FUNCTION)

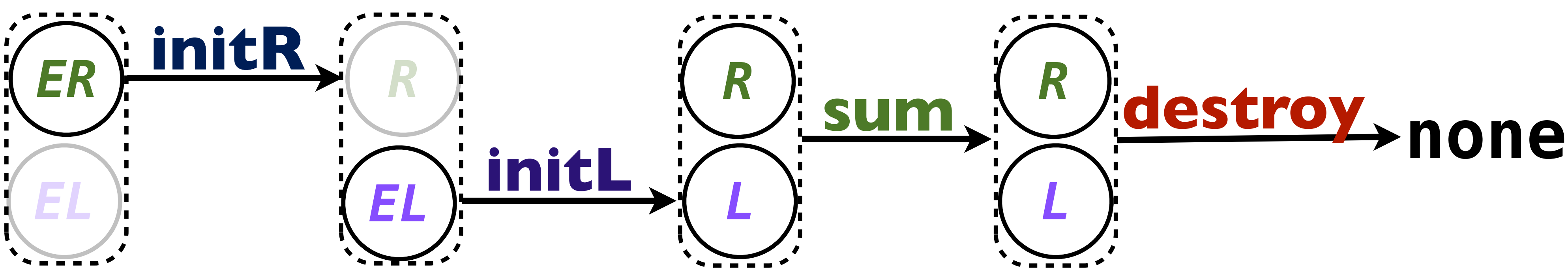
$$\frac{\Gamma; \Delta, x : A_0 \vdash e : A_1 \dashv \cdot}{\Gamma; \Delta \vdash \mathsf{fun}(x : A_0).e : A_0 \multimap A_1 \dashv \cdot}$$

(T:CAP-STACK)

$$\frac{\Gamma; \Delta_0 \vdash e : A_0 \dashv \Delta_1, A_1}{\Gamma; \Delta_0 \vdash e : A_0 :: A_1 \dashv \Delta_1}$$

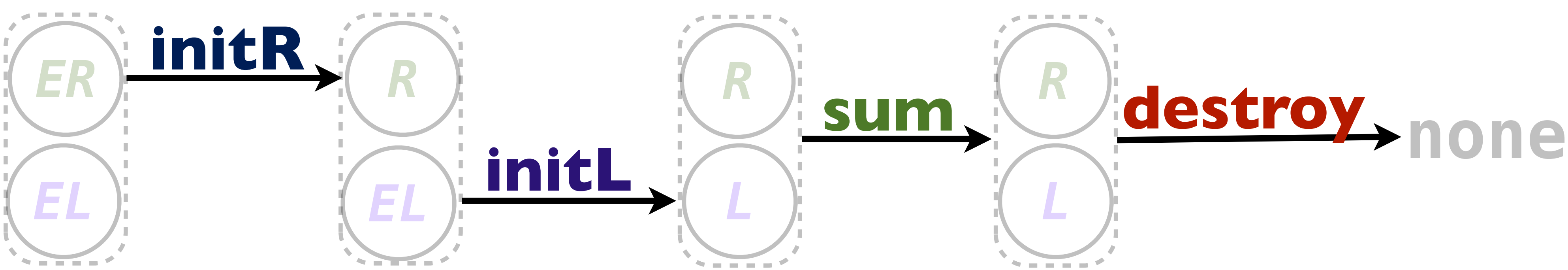
Hiding (Type)state

- Capturing the typestate enables us to hide the typestate needed by the function's *argument*.
- Hiding the typestate from the *result* is not immediately possible. However, we can define a complete sequence of uses ("behavior") that ends in a function that destroys the (type)state.
- One possible linear "behavior" for the Pair is:



Hiding (Type)state

- Capturing the typestate enables us to hide the typestate needed by the function's *argument*.
- Hiding the typestate from the result is not immediately possible. However, we can define a complete sequence of uses ("behavior") that ends in a function that destroys the (type)state.
- One possible linear "behavior" for the Pair is:



```
fun( a : int ).
 {
   initR(a)
 ,
   fun( b : int ).
    {
      initL(b)
    ,
      fun( _ : [] ).
       {
        sum(_)
       ,
        fun( _ : [] ).destroy(_)
       }
    }
 }
```

```
fun( a : int ).
 {
   initR(a)
 ,
   fun( b : int ).
    {
      initL(b)
    ,
      fun( _ : [] ).
       {
        sum(_)
       ,
       fun( _ : [] ).destroy(_)
       }
    }
 }
```

[] ⊸ []

[] :: *L* * *R* ⊸ []

```
fun( a : int ).
 {
   initR(a)
 ,
   fun( b : int ).
    {
      initL(b)
    ,
      fun( _ : [] ).
       {
        sum(_)
       ,
        fun( _ : [] ).destroy(_)
       }
    }
 }
```

[] :: L * R ⊸ int :: L * R

[] ⊸ []

[] :: L * R ⊸ []

```
fun( a : int ).
 {
   initR(a)          int :: ER ⊸ [] :: R
 ,
   fun( b : int ).
    {
      initL(b)          int :: EL ⊸ [] :: L
    ,
      fun( _ : [] ).
       {
        sum(_)          [] :: L * R ⊸ int :: L * R
       ,
[] ⊸ []   fun( _ : [] ).destroy(_)
       }                         [] :: L * R ⊸ []
    }
 }
```

Δ = EL, ER

```
fun( a : int ).
 {
   initR(a)
 ,
   fun( b : int ).
    {
      initL(b)
    ,
      fun( _ : [] ).
       {
        sum(_)
       ,
        fun( _ : [] ).destroy(_)
       }
    }
 }
```

int :: ER ⊸ [] :: R

int :: EL ⊸ [] :: L

[] :: L * R ⊸ int :: L * R

[] ⊸ []

[] :: L * R ⊸ []

$$\overbrace{\mathsf{int} \multimap [\, [\,],\ \overbrace{\mathsf{int} \multimap [\, [\,],\ \overbrace{[\,] \multimap [\, \mathsf{int},\ \underbrace{[\,] \multimap [\,]}_{\texttt{destroy}} \,]}^{\texttt{sum}} \,]}^{\texttt{initL}} \,]}^{\texttt{initR}}$$

$$\overbrace{\text{int} \multimap [\,[\,],\ \overbrace{\text{int} \multimap [\,[\,],\ \overbrace{[\,] \multimap [\,\text{int},\ \underbrace{[\,] \multimap [\,]}_{\texttt{destroy}}\,]}^{\texttt{sum}}\,]}^{\texttt{initL}}\,]}^{\texttt{initR}}$$

Clients never see the underlying **typestates**. They only see the usage requirement ("**behavior**").

Technical Results

Theorem 1 (Progress). *If e_0 is a closed expression (and where Γ and Δ_0 are also closed) such that:*

$$\Gamma; \Delta_0 \vdash e_0 : A \dashv \Delta_1$$

then either:

- *e_0 is a value, or;*
- *if exists H_0 such that $\Gamma; \Delta_0 \vdash H_0$ then $\langle\, H_0 \,\|\, e_0 \,\rangle \mapsto \langle\, H_1 \,\|\, e_1 \,\rangle$.*

Theorem 2 (Preservation). *If e_0 is a closed expression such that:*

$$\Gamma_0; \Delta_0 \vdash e_0 : A \dashv \Delta \qquad \Gamma_0; \Delta_0 \vdash H_0 \qquad \langle\, H_0 \,\|\, e_0 \,\rangle \mapsto \langle\, H_1 \,\|\, e_1 \,\rangle$$

then, for some Δ_1, Γ_1:

$$\Gamma_0, \Gamma_1; \Delta_1 \vdash H_1 \qquad \Gamma_0, \Gamma_1; \Delta_1 \vdash e_1 : A \dashv \Delta$$

Related Work

DeLine and Fähndrich. **Typestates for objects.** ECOOP 2004.

DeLine and Fähndrich. **Enforcing high-level protocols in low-level software**. PLDI 2001.

Bierhoff and Aldrich. **Modular typestate checking of aliased objects**. OOPSLA 2007.

Beckman, Bierhoff, and Aldrich. **Verifying correct usage of atomic blocks and typestate**. OOPSLA 2008.

Sunshine, Naden, Stork, Aldrich, and Tanter. **First-class state change in Plaid**. OOPSLA 2011.

- They support many advanced uses (method dispatch, inheritance, sharing mechanisms, concurrency, etc).
- We focus on reconstructing a smaller set of typestate features from type-theoretic primitives (separation and linear logic). Which enables combining abstracting and hiding state.

Related Work

Ahmed, Fluet, and Morrisett. **L3: A linear language with locations**. Fundam. Inform. 2007.

Walker and Morrisett. **Alias types for recursive data structures**. TIC 2001.

Smith, Walker, and Morrisett. **Alias types**. ESOP 2000.

- We extend their work with usability related changes (implicitly threaded capabilities, alternatives, etc).

Parkinson and Bierman. **Separation logic and abstraction**. POPL 2005.

- *Abstract predicates* can represent a richer domain of abstract state (not limited to a finite number, can be parametric, etc).
- Typestates encode a simpler notion of abstraction, generally targets a more lightweight verification.

Paper includes additional Related Work.

Summary

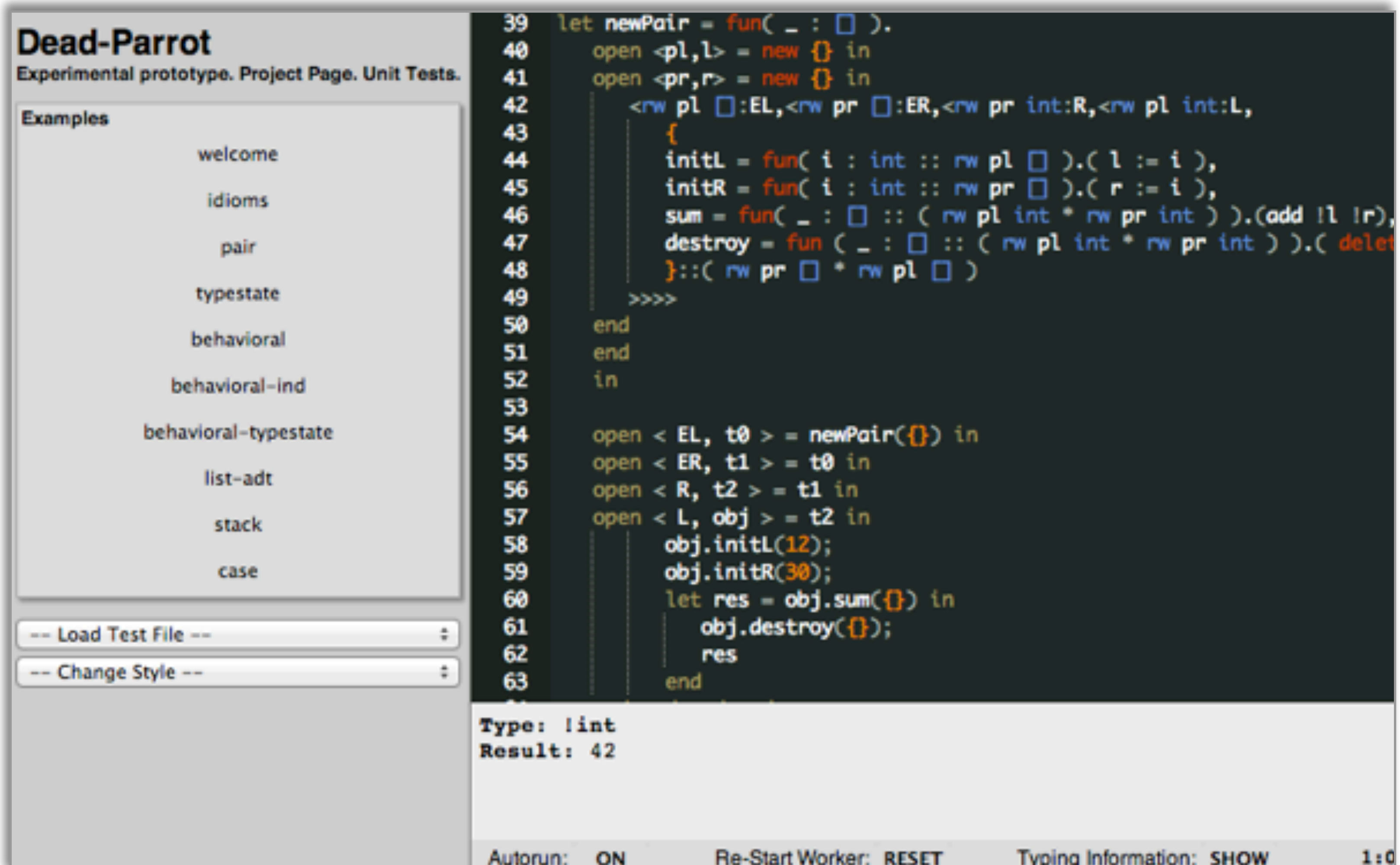
1. Encoding **typestates** using existential types in a substructural type-and-effect system.
2. Support both state-based and transition-based specifications of abstract state evolution.

Experimental Prototype Implementation:
https://code.google.com/p/**dead-parrot**

- *Future Work*:
  Sharing of resources through disconnected variables.

Prototype

JavaScript-based implementation, runs in browser.



Summary

1. Encoding **typestates** using existential types in a substructural type-and-effect system.
2. Support both state-based and transition-based specifications of abstract state evolution.

Experimental Prototype Implementation:
https://code.google.com/p/**dead-parrot**

- *Future Work*:
  Sharing of resources through disconnected variables.