Type Soundness and Race Freedom for Mezzo

Thibaut Balabonski *François Pottier* Jonathan Protzenko

INRIA

FLOPS 2014

Mezzo is a high-level programming language, equipped with:

- algebraic data types;
- first-class functions;
- garbage collection;
- *mutable state;*
- shared-memory *concurrency.*

Its static discipline is based on *permissions*...

```
val r1 = newref ()
(* r1 @ ref () *)
```

```
val r1 = newref ()
(* r1 @ ref () *)
val r2 = r1
(* r1 @ ref () * r2 @ =r1 *)
```

```
val r1 = newref ()
(* r1 @ ref () *)
val r2 = r1
(* r1 @ ref () * r2 @ =r1 *)
(* r1 @ ref () * r2 = r1 *)
```

```
val r1 = newref ()
(* r1 @ ref () *)
val r2 = r1
(* r1 @ ref () * r2 @ =r1 *)
(* r1 @ ref () * r2 = r1 *)
val () = r1 := 0
(* r1 @ ref int * r2 = r1 *)
```

```
val r1 = newref ()
(* r1 @ ref () *)
val r2 = r1
(* r1 @ ref () * r2 @ =r1 *)
(* r1 @ ref () * r2 = r1 *)
val () = r1 := 0
(* r1 @ ref int * r2 = r1 *)
val x2 = !r2 + 1
(* r1 @ ref int * r2 = r1 * x2 @ int *)
```

```
val r1 = newref ()
(* r1 @ ref () *)
val r2 = r1
(* r1 @ ref () * r2 @ =r1 *)
(* r1 @ ref () * r2 = r1 *)
val () = r1 := 0
(* r1 @ ref int * r2 = r1 *)
val x2 = !r2 + 1
(* r1 @ ref int * r2 = r1 * x2 @ int *)
val p = (r1, r2)
(* r1 @ ref int * r2 = r1 * x2 @ int * p @ (=r1, =r2) *)
```

```
val r1 = newref ()
(* r1 @ ref () *)
val r2 = r1
(* r1 @ ref () * r2 @ =r1 *)
(* r1 @ ref () * r2 = r1 *)
val () = r1 := 0
(* r1 @ ref int * r2 = r1 *)
val x2 = !r2 + 1
(* r1 @ ref int * r2 = r1 * x2 @ int *)
val p = (r1, r2)
(* r1 @ ref int * r2 = r1 * x2 @ int * p @ (=r1, =r2) *)
val () = assert p @ (ref int, ref int) (* REJECTED *)
```

```
val r1 = newref ()
(* r1 @ ref () *)
val r2 = r1
(* r1 @ ref () * r2 @ =r1 *)
(* r1 @ ref () * r2 = r1 *)
val () = r1 := 0
(* r1 @ ref int * r2 = r1 *)
val x2 = !r2 + 1
(* r1 @ ref int * r2 = r1 * x2 @ int *)
val p = (r1, r2)
(* r1 @ ref int * r2 = r1 * x2 @ int * p @ (=r1, =r2) *)
val () = assert p @ (ref int, ref int) (* REJECTED *)
val () = assert p @ (r: ref int, =r)   (* ACCEPTED *)
```

```
val r1 = newref ()
(* r1 @ ref () *)
val r2 = r1
(* r1 @ ref () * r2 @ =r1 *)
(* r1 @ ref () * r2 = r1 *)
val () = r1 := 0
(* r1 @ ref int * r2 = r1 *)
val x2 = !r2 + 1
(* r1 @ ref int * r2 = r1 * x2 @ int *)
val p = (r1, r2)
(* r1 @ ref int * r2 = r1 * x2 @ int * p @ (=r1, =r2) *)
val () = assert p @ (ref int, ref int) (* REJECTED *)
val () = assert p @ (r: ref int, =r)   (* ACCEPTED *)
val () = assert p @ (=r, r: ref int)   (* ACCEPTED *)
```

Permissions by example

```
val r1 = newref ()
(* r1 @ ref () *)
val r2 = r1
(* r1 @ ref () * r2 @ =r1 *)
(* r1 @ ref () * r2 = r1 *)
val () = r1 := 0
(* r1 @ ref int * r2 = r1 *)
val x2 = !r2 + 1
(* r1 @ ref int * r2 = r1 * x2 @ int *)
val p = (r1, r2)
(* r1 @ ref int * r2 = r1 * x2 @ int * p @ (=r1, =r2) *)
val () = assert p @ (ref int, ref int) (* REJECTED *)
val () = assert p @ (r: ref int, =r)   (* ACCEPTED *)
val () = assert p @ (=r, r: ref int)   (* ACCEPTED *)
```

Why do this?

Permissions by example

```
val r1 = newref ()
(* r1 @ ref () *)
val r2 = r1
(* r1 @ ref () * r2 @ =r1 *)
(* r1 @ ref () * r2 = r1 *)
val () = r1 := 0
(* r1 @ ref int * r2 = r1 *)
val x2 = !r2 + 1
(* r1 @ ref int * r2 = r1 * x2 @ int *)
val p = (r1, r2)
(* r1 @ ref int * r2 = r1 * x2 @ int * p @ (=r1, =r2) *)
val () = assert p @ (ref int, ref int) (* REJECTED *)
val () = assert p @ (r: ref int, =r)   (* ACCEPTED *)
val () = assert p @ (=r, r: ref int)   (* ACCEPTED *)
```

Motivation

Imagine an imperative implementation of sets:

```
val make:   [a] () -> set a
val insert: [a] (set a, consumes a) -> ()
val merge:  [a] (set a, consumes set a) -> ()
```

Imagine an imperative implementation of sets:

```
val make:   [a] () -> set a
val insert: [a] (set a, consumes a) -> ()
val merge:  [a] (set a, consumes set a) -> ()
```

Then,

- `let s = make() in ...` produces `s @ set t`

Imagine an imperative implementation of sets:

```
val make:   [a] () -> set a
val insert: [a] (set a, consumes a) -> ()
val merge:  [a] (set a, consumes set a) -> ()
```

Then,

- `let s = make() in ...` produces `s @ set t`
- cannot do `merge(s, s);`

Imagine an imperative implementation of sets:

```
val make:   [a] () -> set a
val insert: [a] (set a, consumes a) -> ()
val merge:  [a] (set a, consumes set a) -> ()
```

Then,

- `let s = make() in ...` produces `s @ set t`
- cannot do `merge(s, s);`
- cannot do `merge(s1, s2); insert(s2, x);`

Imagine an imperative implementation of sets:

```
val make:   [a] () -> set a
val insert: [a] (set a, consumes a) -> ()
val merge:  [a] (set a, consumes set a) -> ()
```

Then,

- `let s = make() in ...` produces `s @ set t`
- cannot do `merge(s, s);`
- cannot do `merge(s1, s2); insert(s2, x);`
- cannot do `insert(s, x1)` and `insert(s, x2)` in independent threads.

Imagine an imperative implementation of sets:

```
val make:   [a] () -> set a
val insert: [a] (set a, consumes a) -> ()
val merge:  [a] (set a, consumes set a) -> ()
```

Then,

- `let s = make() in ...` produc
- cannot do `merge(s, s);`
- cannot do `merge(s1, s2); insert(s2, x);`
- cannot do `insert(s, x1)` and `insert(s, x2)` in independent threads.

```
error in sequential code:
protocol violation
```

Motivation

Imagine an imperative implementation of sets:

```
val make:   [a] () -> set a
val insert: [a] (set a, consumes a) -> ()
val merge:  [a] (set a, consumes set a) -> ()
```

Then,

- `let s = make() in ...` produc
- cannot do `merge(s, s);`
- cannot do `merge(s1, s2); insert(s2, x);`
- cannot do `insert(s, x1)` and `insert(s, x2)` in independent threads.

error in concurrent code:
data race

Permissions, in a nutshell

Like a program logic, Mezzo's *static* discipline is flow-sensitive.

- A *current* (set of) *permission*(s) exists *at each program point*.
- *Different* permissions exist at different points.
- There is no such thing as *the* type of a variable.

A permission has *layout* and *ownership* readings.

A permission is either *duplicable* or *affine*.

What this talk is not about

The paper and talk do *not* discuss:

- algebraic data types, which describe *tree-shaped* data,
- (static) regions, which can describe *non-tree-shaped* data,
- adoption & abandon, a *dynamic* alternative to regions,

and much more (ICFP 2013).

```
data list a =
  | Nil
  | Cons { head: a; tail: list a }

data mutable mlist a =
  | MNil
  | MCons { head: a; tail: mlist a }
```

```
val rec meld_aux [a]
  (xs: MCons { head: a; tail: mlist a },
   consumes ys: mlist a) : () =
  match xs.tail with
  | MNil  ->
      xs.tail <- ys
  | MCons ->
      meld_aux (xs.tail, ys)
  end
```

Censored

```
val rec append_aux [a] (consumes (
  dst: MCons { head: a; tail: () },
  xs: list a, ys: list a
)) : (| dst @ list a) =
  match xs with
  | Cons ->
      let dst' = MCons { head = xs.head; tail = () } in
      dst.tail <- dst';
      tag of dst <- Cons;
      append_aux (dst', xs.tail, ys)
  | Nil ->
      dst.tail <- ys;
      tag of dst <- Cons
  end
```

Censored

```
abstract region
val newregion: () -> region
abstract rref (rho : value) a
fact duplicable (rref rho a)
val newrref: (consumes x: a | rho @ region) -> rref rho a
val get: (r: rref rho a | duplicable a | rho @ region) -> a
val set: (r: rref rho a, consumes x: a | rho @ region) -> ()
```

Censored

```
val dfs [a] (g: graph a, f: a -> ()) : () =
  let s = stack::new g.roots in
  stack::work (s, fun (n: dynamic
                  | g @ graph a * s @ stack dynamic) : () =
    take n from g;
    if not n.visited then begin
      n.visited <- true;
      f n.content;
      stack::push (n.neighbors, s)
    end;
    give n to g
  )
```

Censored

So, what *are* the paper and talk about?

- extend Mezzo with *threads* and *locks;*
- describe a *modular*, *machine-checked* proof of
  - type soundness;
  - data race freedom.

Outline

- Threads, data races, and locks (by example)
- Mezzo's architecture
- The kernel and its proof (glimpses)
- Conclusion

```
open thread

val r = newref 0

val f (| r @ ref int) : () =

  r := !r + 1

val () =
  spawn f ;
  spawn f
```

```
open thread

val r = newref 0

val f (| r @ ref int) : () =

  r := !r + 1

val () =
  spawn f ;
  spawn f
```

f requires r @ ref int
and gives it back

```
open thread

val r = newref 0

val f (| r @ ref int) : () =

  r := !r + 1

val () =
  spawn f ;
  spawn f
```

spawn f requires r @ ref int
and does NOT give it back

```
open thread

val r = newref 0

val f (| r @ ref int) : () =

  r := !r + 1

val () =
  spawn f ;
  spawn f
```

TYPE ERROR!
(in fact, this code is racy)

Introducing synchronization

```
open thread
open lock
val r = newref 0
val l : lock (r @ ref int) = new()
val f () : () =
  acquire l;
  r := !r + 1;
  release l
val () =
  spawn f ;
  spawn f
```

```
open thread
open lock
val r = newref 0
val l : lock (r @ ref int) = new()
val f () : () =
  acquire l;
  r := !r + 1;
  release l
val () =
  spawn f ;
  spawn f
```

this consumes r @ ref int
the lock now mediates access to it

```
open thread
open lock
val r = newref 0
val l : lock (r @ ref int) = new()
val f () : () =
  acquire l;
  r := !r + 1;
  release l
val () =
  spawn f ;
  spawn f
```

f requires no permission
l @ lock (...) is duplicable

```
open thread
open lock
val r = newref 0
val l : lock (r @ ref int) = new()
val f () : () =
  acquire l;
  r := !r + 1;
  release l
val () =
  spawn f ;
  spawn f
```

acquiring the lock
produces r @ ref int

```
open thread
open lock
val r = newref 0
val l : lock (r @ ref int) = new()
val f () : () =
  acquire l;
  r := !r + 1;
  release l
val () =
  spawn f ;
  spawn f
```

r @ ref int allows updating r

```
open thread
open lock
val r = newref 0
val l : lock (r @ ref int) = new()
val f () : () =
  acquire l;
  r := !r + 1;
  release l
val () =
  spawn f ;
  spawn f
```

releasing the lock
consumes r @ ref int

```
open thread
open lock
val r = newref 0
val l : lock (r @ ref int) = new()
val f () : () =
  acquire l;
  r := !r + 1;
  release l
val () =
  spawn f ;
  spawn f
```

WELL-TYPED!
(yup, this code is race free)

```
(* A second-order function. *)

val hide : [a, b, s : perm] (
  f : (consumes a | s) -> b |
  consumes s
) -> (consumes a) -> b
```

```
(* A second-order function. *)

val hide : [a, b, s : perm] (
  f : (consumes a | s) -> b |
  consumes s
) -> (consumes a) -> b
```

hide is polymorphic in s
e.g., r @ ref int

```
(* A second-order function. *)

val hide : [a, b, s : perm] (
  f : (consumes a | s) -> b |
  consumes s
) -> (consumes a) -> b
```

hide takes a function f
which has a side effect on s

```
(* A second-order function. *)

val hide : [a, b, s : perm] (
  f : (consumes a | s) -> b |
  consumes s
) -> (consumes a) -> b
```

hide consumes s
which becomes owned by the lock

```
(* A second-order function. *)

val hide : [a, b, s : perm] (
  f : (consumes a | s) -> b |
  consumes s
) -> (consumes a) -> b
```

hide produces a new function
which has no advertised effect

```
open lock

val hide   [a, b, s : perm] (
  f : (consumes a | s) -> b |
  consumes s
)  : (consumes a) -> b =
  let l : lock s = new () in
  fun (consumes x : a) : b =
    acquire l;
    let y = f x in
    release l;
    y
```

Introducing synchronization, revisited

```
open thread
open hide
val r = newref 0
val f (| r @ ref int) : () =
  r := !r + 1

val f = hide f

val () =
  spawn f;
  spawn f
```

```
open thread
open hide
val r = newref 0
val f (| r @ ref int) : () =
  r := !r + 1

val f = hide f

val () =
  spawn f;
  spawn f
```

r @ ref int

```
open thread
open hide
val r = newref 0
val f (| r @ ref int) : () =
  r := !r + 1

val f = hide f

val () =
  spawn f;
  spawn f
```

```
r @ ref int
f @ (| r @ ref int) -> ()
```

```
open thread
open hide
val r = newref 0
val f (| r @ ref int) : () =
  r := !r + 1

val f = hide f

val () =
  spawn f;
  spawn f
```

```
f @ () -> ()
```

```
open thread
open hide
val r = newref 0
val f (| r @ ref int) : () =
  r := !r + 1

val f = hide f

val () =
  spawn f;
  spawn f
```

WELL-TYPED!
(yup, this code is race free)

Outline

- Threads, data races, and locks (by example)
- Mezzo's architecture
- The kernel and its proof (glimpses)
- Conclusion

A kernel:

- a λ-calculus with threads;
- affine, polymorphic, value-dependent, with type erasure.

Several extensions:

- mutable state: *references;*
- hidden state: *locks;*
- dynamic ownership control: *adoption and abandon.*

All *machine-checked* in Coq (14KLOC).

We wish to prove that well-typed programs:

- *do not go wrong*;
- *are data-race free*.

This is trivial - true of *all* programs - in the kernel calculus!

Subject reduction and *progress* are non-trivial results.

We set up their proof so that it is *robust* in the face of extensions.

We *parameterize* the kernel with:

- a type of *machine states* s;
- a type of *instrumented states* R, or *resources*;
  - which must form a *monotonic separation algebra;*
- a correspondence relation, $s \sim R$.

Subject reduction and progress hold *for all* such parameters.

Pseudo-Modularity

The kernel is *not* parameterized w.r.t. the extensions.

We add the extensions, one after another, on top of the kernel.

So, the Coq code is *monolithic*. Fortunately,

- each extension is (morally) *independent* of the others;
- the key statements *do not change* with extensions;
- only new proof cases appear.

Outline

- Threads, data races, and locks (by example)
- Mezzo's architecture
- The kernel and its proof (glimpses)
- Conclusion

Values and terms

A fairly unremarkable untyped λ-calculus with threads.

$$
\begin{array}{rcll}
\kappa & ::= & \texttt{value} \mid \texttt{term} \mid \texttt{soup} \mid \ldots & \text{(Kinds)} \\
v & ::= & x \mid \lambda x.t & \text{(Values)} \\
t & ::= & v \mid v\ t \mid \texttt{spawn}\ v\ v & \text{(Terms)}
\end{array}
$$

$$
\begin{array}{lll}
\textit{initial configuration} & & \textit{new configuration} \\
s \ / \ (\lambda x.t)\ v & \longrightarrow & s \quad / \ [v/x]t \\
 & & \\
s \ / \ E[t] & \longrightarrow & s' \quad / \ E[t'] \\
 & & \text{if } s \ / \ t \longrightarrow s' \ / \ t' \\
s \ / \ \texttt{thread}\ (t) & \longrightarrow & s' \quad / \ \texttt{thread}\ (t') \\
 & & \text{if } s \ / \ t \longrightarrow s' \ / \ t' \\
s \ / \ t_1 \parallel t_2 & \longrightarrow & s' \quad / \ t'_1 \parallel t_2 \\
 & & \text{if } s \ / \ t_1 \longrightarrow s' \ / \ t'_1 \\
s \ / \ t_1 \parallel t_2 & \longrightarrow & s' \quad / \ t_1 \parallel t'_2 \\
 & & \text{if } s \ / \ t_2 \longrightarrow s' \ / \ t'_2 \\
s \ / \ \texttt{thread}\ (D[\texttt{spawn}\ v_1\ v_2]) & \longrightarrow & s \quad / \ \texttt{thread}\ (D[()]) \parallel \texttt{thread}\ (v_1\ v_2)
\end{array}
$$

an abstract notion
of machine state

initial configuration		*new configuration*
$s \;/\; (\lambda x.t)\; v$	$\longrightarrow$	$s \;/\; [v/x]t$
$s \;/\; E[t]$	$\longrightarrow$	$s' \;/\; E[t']$ if $s \;/\; t \longrightarrow s' \;/\; t'$
$s \;/\; \mathtt{thread}\;(t)$	$\longrightarrow$	$s' \;/\; \mathtt{thread}\;(t')$ if $s \;/\; t \longrightarrow s' \;/\; t'$
$s \;/\; t_1 \parallel t_2$	$\longrightarrow$	$s' \;/\; t'_1 \parallel t_2$ if $s \;/\; t_1 \longrightarrow s' \;/\; t'_1$
$s \;/\; t_1 \parallel t_2$	$\longrightarrow$	$s' \;/\; t_1 \parallel t'_2$ if $s \;/\; t_2 \longrightarrow s' \;/\; t'_2$
$s \;/\; \mathtt{thread}\;(D[\mathtt{spawn}\; v_1\; v_2])$	$\longrightarrow$	$s \;/\; \mathtt{thread}\;(D[()]) \parallel \mathtt{thread}\;(v_1\; v_2)$

$$
\begin{array}{rcll}
\kappa & ::= & \ldots \mid \texttt{type} \mid \texttt{perm} & \text{(Kinds)} \\
T, U & ::= & x \mid =v \mid T \rightarrow T \mid (T \mid P) & \text{(Types)} \\
 & & \forall x : \kappa.T \mid \exists x : \kappa.T & \\
P, Q & ::= & x \mid v\,@\,T \mid \texttt{empty} \mid P \ast P & \text{(Permissions)} \\
 & & \forall x : \kappa.P \mid \exists x : \kappa.P & \\
 & & \texttt{duplicable}\ \theta & \\
\theta & ::= & T \mid P &
\end{array}
$$

The typing judgement

A traditional type system uses a list Γ of *type assumptions*:

$$\Gamma \vdash t : T$$

Mezzo splits it into a list K of *kind assumptions* and a *permission* P:

$$K, P \vdash t : T$$

This can be read like a Hoare triple: $K \vdash \{P\}\, t\, \{T\}$.

A typing judgement about a *running* program (or thread) depends on a resource R:

$$R, K, P \vdash t : T$$

R is the thread's *partial*, *instrumented view* of the machine state...

Resources

A resource is:

- *partial*: a resource could be, say, a heap fragment;
- *instrumented:* a resource could record whether each location is mutable or immutable.

Resources

A resource is:

- *partial*: a resource could be, say, a heap fragment;
- *instrumented:* a resource could record whether each location is mutable or immutable.

At this stage, though, resources are *abstract*.

What properties must we require of them?

Monotonic separation algebra

R	*resource*
	e.g., an instrumented heap fragment
	maps every address to $\lightning$, N, $X\,v$, or $D\,v$
$R_1 \star R_2$	*conjunction*
	e.g., requires separation at mutable addresses
	requires agreement at immutable addresses
$\widehat{R}$	*duplicable core*
	e.g., throws away mutable addresses
	keeps immutable addresses
$R_1 \lhd R_2$	*tolerable interference (rely)*
	e.g., allows memory allocation

Working with abstract resources

- Star $\star$ is commutative and associative.
- $R_1 \star R_2$ *ok* implies R_1 *ok*.
- $R \star \widehat{R} = R$.
- $R_1 \star R_2 = R$ and R *ok* imply $\widehat{R_1} = \widehat{R}$.
- $R \star R = R$ implies $R = \widehat{R}$.
- $\widehat{R} \star \widehat{R} = \widehat{R}$.
- $R \lhd R$.
- R_1 *ok* and $R_1 \lhd R_2$ imply R_2 *ok*.
- $R_1 \lhd R_2$ implies $\widehat{R_1} \lhd \widehat{R_2}$.
- rely preserves splits:

$$\frac{R_1 \star R_2 \lhd R' \qquad R_1 \star R_2 \; ok}{\exists R_1' R_2',\ R_1' \star R_2' = R' \wedge R_1 \lhd R_1' \wedge R_2 \lhd R_2'}$$

A small set of typing rules

Singleton
$$R; K; P \ \diamond \ v : =v$$

Frame
$$\frac{R; K; P \ \diamond \ t : T}{R; K; P \ * \ Q \ \diamond \ t : T \mid Q}$$

Function
$$\frac{\widehat{R}; K, x : \texttt{value}; P \ * \ x \,@\, T \vdash t : U}{R; K; (\texttt{duplicable}\ P) \ * \ P \ \diamond \ \lambda x.t : T \to U}$$

ForallIntro
$$\frac{\textit{t is harmless} \qquad R; K, x : \kappa; P \ \diamond \ t : T}{R; K; \forall x : \kappa.P \ \diamond \ t : \forall x : \kappa.T}$$

ExistsIntro
$$\frac{R; K; P \ \diamond \ v : [U/x]T}{R; K; P \ \diamond \ v : \exists x : \kappa.T}$$

Cut
$$\frac{R_2; K; P_1 \ * \ P_2 \ \diamond \ t : T \qquad R_1; K \Vdash P_1}{R_1 \ \star \ R_2; K; P_2 \ \diamond \ t : T}$$

ExistsElim
$$\frac{R; K, x : \kappa; P \vdash t : T}{R; K; \exists x : \kappa.P \vdash t : T}$$

SubLeft
$$\frac{K \vdash P_1 \leq P_2 \qquad R; K; P_2 \vdash t : T}{R; K; P_1 \vdash t : T}$$

SubRight
$$\frac{R; K; P \vdash t : T_1 \qquad K \vdash T_1 \leq T_2}{R; K; P \vdash t : T_2}$$

Application
$$\frac{R; K; Q \vdash t : T}{R; K; (v \,@\, T \to U) \ * \ Q \vdash v\ t : U}$$

Spawn
$$R; K; (v_1 \,@\, T \to U) \ * \ (v_2 \,@\, T) \vdash \texttt{spawn}\ v_1\ v_2 : \top$$

Selected typing rules

The kernel typing rules manipulate R abstractly.

$$\frac{\widehat{R}; K, x : \texttt{value}; P \ast x \,@\, T \vdash t : U}{R; K; (\texttt{duplicable}\ P) \ast P \vdash \lambda x.t : T \to U}$$

$$\frac{\begin{array}{c} R_2; K; P_1 \ast P_2 \vdash t : T \\ R_1; K \Vdash P_1 \end{array}}{R_1 \star R_2; K; P_2 \vdash t : T}$$

The kernel typing rules manipulate R abstractly.

$$\frac{\widehat{R}; K, x : \mathtt{value}; P \ast x \,@\, T \vdash t : U}{R; K; (\mathtt{duplicable}\ P) \ast P \vdash \lambda x.t : T \to U}$$

$$\frac{\begin{array}{c} R_2; K; P_1 \ast P_2 \vdash t : T \\ R_1; K \Vdash P_1 \end{array}}{R_1 \star R_2; K; P_2 \vdash t : T}$$

cannot capture an arbitrary resource R
can capture its duplicable core $\widehat{R}$

The kernel typing rules manipulate *R* abstractly.

$$\frac{\widehat{R}; K, x : \texttt{value}; P \ast x \,@\, T \vdash t : U}{R; K; (\texttt{duplicable}\ P) \ast P \vdash \lambda x.t : T \to U}$$

$$\frac{R_2; K; P_1 \ast P_2 \vdash t : T \qquad R_1; K \Vdash P_1}{R_1 \star R_2; K; P_2 \vdash t : T}$$

if a typing rule has two premises
then *R* must be split between them

Lemma (S.R., preliminary form)

$$\frac{\begin{array}{c} s_1 \;/\; t_1 \longrightarrow s_2 \;/\; t_2 \\ s_1 \sim R_1 \star R_1' \\ R_1; \varnothing; \texttt{empty} \vdash t_1 : T \end{array}}{\exists R_2 R_2' \left\{ \begin{array}{l} s_2 \sim R_2 \star R_2' \\ R_2; \varnothing; \texttt{empty} \vdash t_2 : T \\ R_1' \lhd R_2' \end{array} \right.}$$

one thread takes a step

Lemma (S.R., preliminary form)

$$\frac{\begin{array}{c} s_1 \;/\; t_1 \longrightarrow s_2 \;/\; t_2 \\ s_1 \sim R_1 \star R'_1 \\ R_1; \varnothing; \mathtt{empty} \vdash t_1 : T \end{array}}{\exists R_2 R'_2 \left\{ \begin{array}{l} s_2 \sim R_2 \star R'_2 \\ R_2; \varnothing; \mathtt{empty} \vdash t_2 : T \\ R'_1 \lhd R'_2 \end{array} \right.}$$

this thread's view is R_1
the other threads' view is R'_1

Lemma (S.R., preliminary form)

$$\frac{\begin{array}{c} s_1 \;/\; t_1 \longrightarrow s_2 \;/\; t_2 \\ s_1 \sim R_1 \star R'_1 \\ R_1; \varnothing; \texttt{empty} \vdash t_1 : T \end{array}}{\exists R_2 R'_2 \left\{ \begin{array}{l} s_2 \sim R_2 \star R'_2 \\ R_2; \varnothing; \texttt{empty} \vdash t_2 : T \\ R'_1 \lhd R'_2 \end{array} \right.}$$

this thread is well-typed
under its view

Lemma (S.R., preliminary form)

$$\frac{\begin{array}{c} s_1 \;/\; t_1 \longrightarrow s_2 \;/\; t_2 \\ s_1 \sim R_1 \star R_1' \\ R_1 ; \varnothing ; \mathtt{empty} \vdash t_1 : T \end{array}}{\exists R_2 R_2' \left\{ \begin{array}{l} s_2 \sim R_2 \star R_2' \\ R_2 ; \varnothing ; \mathtt{empty} \vdash t_2 : T \\ R_1' \lhd R_2' \end{array} \right.}$$

Lemma (S.R., preliminary form)

$$\frac{\begin{array}{c} s_1 \;/\; t_1 \longrightarrow s_2 \;/\; t_2 \\ s_1 \sim R_1 \star R_1' \\ R_1; \varnothing; \texttt{empty} \vdash t_1 : T \end{array}}{\exists R_2 R_2' \left\{ \begin{array}{l} s_2 \sim R_2 \star R_2' \\ R_2; \varnothing; \texttt{empty} \vdash t_2 : T \\ R_1' \lhd R_2' \end{array} \right.}$$

this thread's view and the other threads' view evolve

Lemma (S.R., preliminary form)

$$\frac{\begin{array}{c} s_1 \;/\; t_1 \longrightarrow s_2 \;/\; t_2 \\ s_1 \sim R_1 \star R_1' \\ R_1; \varnothing; \texttt{empty} \vdash t_1 : T \end{array}}{\exists R_2 R_2' \left\{ \begin{array}{l} s_2 \sim R_2 \star R_2' \\ R_2; \varnothing; \texttt{empty} \vdash t_2 : T \\ R_1' \lhd R_2' \end{array} \right.}$$

the new machine state agrees with the new views

Lemma (S.R., preliminary form)

$$\frac{\begin{array}{c} s_1 \;/\; t_1 \longrightarrow s_2 \;/\; t_2 \\ s_1 \sim R_1 \star R_1' \\ R_1; \varnothing; \mathtt{empty} \vdash t_1 : T \end{array}}{\exists R_2 R_2' \left\{ \begin{array}{l} s_2 \sim R_2 \star R_2' \\ R_2; \varnothing; \mathtt{empty} \vdash t_2 : T \\ R_1' \lhd R_2' \end{array} \right.}$$

`the thread remains well-typed under its view`

Lemma (S.R., preliminary form)

$$\frac{\begin{array}{c} s_1 \;/\; t_1 \longrightarrow s_2 \;/\; t_2 \\ s_1 \sim R_1 \star R'_1 \\ R_1; \varnothing; \texttt{empty} \vdash t_1 : T \end{array}}{\exists R_2 R'_2 \begin{cases} s_2 \sim R_2 \star R'_2 \\ R_2; \varnothing; \texttt{empty} \vdash t_2 : T \\ R'_1 \lhd R'_2 \end{cases}}$$

`the interference inflicted on the other threads is tolerable`

Theorem (Subject Reduction)

Reduction preserves well-typedness.

$$\frac{c_1 \longrightarrow c_2 \qquad \vdash c_1}{\vdash c_2}$$

A configuration c is *acceptable* if every thread:

- has reached an answer; or
- is able to make one step; or
- (after introducing locks) is waiting on a locked lock.

Theorem (Progress)

Every well-typed configuration is acceptable.

Cannot be stated for the kernel. We introduce references first.

There, writing requires an exclusive access right.

Hence, it is easy to prove that:

Theorem

A well-typed program cannot exhibit a data race.

- Threads, data races, and locks (by example)
- Mezzo's architecture
- The kernel and its proof (glimpses)
- Conclusion

Alias Types. Separation Logic. L^3. (And a lot more.)

Views (Dinsdale-Young *et al.*, 2013) are particularly relevant.

- extensible framework;
- monolithic machine state, composable views, agreement;
- while-language instead of a λ-calculus.

A few lessons

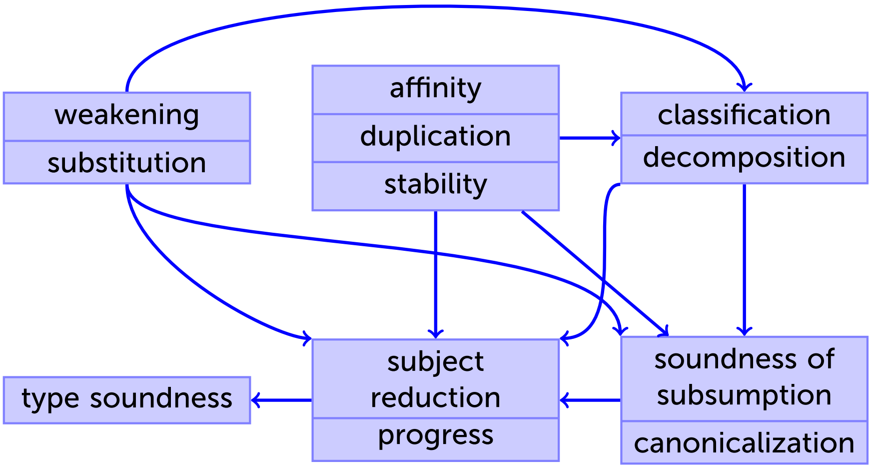
- The good old *syntactic approach* to type soundness works.
- Formalization *helps* clarify and simplify. *A lot.*
- In the end, it is "just" affine λ-calculus.

Thank you

More information in the paper and online:
`http://gallium.inria.fr/~protzenk/mezzo-lang/`

Try it out!

Road map


weakening
substitution
affinity
duplication
stability
classification
decomposition
type soundness
subject reduction
progress
soundness of subsumption
canonicalization

Dealing with binding

In Coq, we use only *one syntactic category*.

Well-kindedness distinguishes values, terms, types, etc.

- avoids a quadratic number of substitution functions!
- makes it easy to deal with dependency.

Binding encoded via de Bruijn indices.

Re-usable library, dblib.

The main hygiene lemmas have >90 cases and 4-line proofs.

```
data list a =
  | Nil
  | Cons { head: a; tail: list a }

data mutable mlist a =
  | MNil
  | MCons { head: a; tail: mlist a }
```

```
val rec meld_aux [a]
  (xs: MCons { head: a; tail: mlist a },
   consumes ys: mlist a) : () =
  match xs.tail with
  | MNil  ->
      xs.tail <- ys
  | MCons ->
      meld_aux (xs.tail, ys)
  end
```

```
val rec append_aux [a] (consumes (
  dst: MCons { head: a; tail: () },
  xs: list a, ys: list a
)) : (| dst @ list a) =
  match xs with
  | Cons ->
      let dst' = MCons { head = xs.head; tail = () } in
      dst.tail <- dst';
      tag of dst <- Cons;
      append_aux (dst', xs.tail, ys)
  | Nil ->
      dst.tail <- ys;
      tag of dst <- Cons
  end
```

```
abstract region
val newregion: () -> region
abstract rref (rho : value) a
fact duplicable (rref rho a)
val newrref: (consumes x: a | rho @ region) -> rref rho a
val get: (r: rref rho a | duplicable a | rho @ region) -> a
val set: (r: rref rho a, consumes x: a | rho @ region) -> ()
```

```
val dfs [a] (g: graph a, f: a -> ()) : () =
  let s = stack::new g.roots in
  stack::work (s, fun (n: dynamic
                 | g @ graph a * s @ stack dynamic) : () =
    take n from g;
    if not n.visited then begin
      n.visited <- true;
      f n.content;
      stack::push (n.neighbors, s)
    end;
    give n to g
  )
```