The theory of Mezzo

François Pottier

INRIA

IHP, April 2014

Acknowledgements

Jonathan Protzenko, Thibaut Balabonski,
Henri Chataing, Armaël Guéneau, Cyprien Mangin.

- Introduction
- The kernel
- Extensions
- Conclusion

Surface versus kernel

Concerning the syntax of types,

- Surface has the *name introduction* form `x: t`,
- which Kernel does not have;

Furthermore, the conventional reading of function types differs:

- Surface functions *do not consume* their arguments, except for the parts marked with **`consumes`**;
- Kernel has the opposite convention, which is standard in affine λ-calculi, hence no **`consumes`** keyword.

Recall the type of the `length` function for mutable lists.

```
[a] mlist a -> int
```

In Surface syntax, this could also be written:

```
[a]       (consumes xs : mlist a) ->
             (int | xs @ mlist a)
```

or, by exploiting universal quantification and a singleton type:

```
[a, xs : value]
    (=xs | consumes xs @ mlist a) ->
             (int | xs @ mlist a)
```

Erasing **consumes** yields a Kernel type that means the same thing.

A Surface pair of a value and a function that consumes it:

```
(x: a, (| consumes x @ a) -> ())
```

In Surface syntax, this could also be written:

```
{x : value} ((=x | x @ a), (| consumes x @ a) -> ())
```

This uses existential quantification and a singleton type.

Erasing `consumes` yields a Kernel type that means the same thing.

Outline

- Introduction
- The kernel
  - The untyped calculus
  - Type-checking inert programs
  - Type-checking running programs; resources
  - The path to type soundness
- Extensions
- Conclusion

The untyped calculus

A fairly unremarkable untyped λ-calculus.

$$
\begin{array}{rcll}
\kappa & ::= & \text{value} \mid \text{term} \mid \text{soup} \mid \ldots & \text{(Kinds)} \\
v & ::= & x \mid \lambda x.t & \text{(Values)} \\
t & ::= & v \mid v\ t \mid \text{spawn}\ v\ v & \text{(Terms)} \\
sp & ::= & \text{thread}\ (t) \mid sp \parallel sp & \text{(Soups)} \\
E & ::= & v\ [] & \text{(Shallow evaluation contexts)} \\
D & ::= & [] \mid E[D] & \text{(Deep evaluation contexts)}
\end{array}
$$

Values and terms

a variant of A-normal form

A fairly unremarkable untyped λ-calculus.

$$
\begin{array}{rcll}
\kappa & ::= & \text{value} \mid \text{term} \mid \text{soup} \mid \ldots & \text{(Kinds)} \\
v & ::= & x \mid \lambda x.t & \text{(Values)} \\
t & ::= & v \mid v\ t \mid \text{spawn}\ v\ v & \text{(Terms)} \\
sp & ::= & \text{thread}\ (t) \mid sp \parallel sp & \text{(Soups)} \\
E & ::= & v\ [] & \text{(Shallow evaluation contexts)} \\
D & ::= & [] \mid E[D] & \text{(Deep evaluation contexts)}
\end{array}
$$

a primitive construct
for spawning a new thread

A fairly unremarkable untyped λ-calculus.

$$
\begin{array}{rcll}
\kappa & ::= & \text{value} \mid \text{term} \mid \text{soup} \mid \ldots & \text{(Kinds)} \\
v & ::= & x \mid \lambda x.t & \text{(Values)} \\
t & ::= & v \mid v\ t \mid \text{spawn}\ v\ v & \text{(Terms)} \\
sp & ::= & \text{thread}\ (t) \mid sp \parallel sp & \text{(Soups)} \\
E & ::= & v\ [] & \text{(Shallow evaluation contexts)} \\
D & ::= & [] \mid E[D] & \text{(Deep evaluation contexts)}
\end{array}
$$

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

an abstract notion
of machine state

initial configuration — *new configuration*

$$s \;/\; (\lambda x.t)\; v \longrightarrow s \;/\; [v/x]t$$

$$s \;/\; E[t] \longrightarrow s' \;/\; E[t'] \quad \text{if } s \;/\; t \longrightarrow s' \;/\; t'$$

$$s \;/\; \text{thread}\;(t) \longrightarrow s' \;/\; \text{thread}\;(t') \quad \text{if } s \;/\; t \longrightarrow s' \;/\; t'$$

$$s \;/\; t_1 \parallel t_2 \longrightarrow s' \;/\; t_1' \parallel t_2 \quad \text{if } s \;/\; t_1 \longrightarrow s' \;/\; t_1'$$

$$s \;/\; t_1 \parallel t_2 \longrightarrow s' \;/\; t_1 \parallel t_2' \quad \text{if } s \;/\; t_2 \longrightarrow s' \;/\; t_2'$$

$$s \;/\; \text{thread}\;(D[\text{spawn}\; v_1\; v_2]) \longrightarrow s \;/\; \text{thread}\;(D[()]) \parallel \text{thread}\;(v_1\; v_2)$$

Type-checking inert programs

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

In the Coq formalisation, only *one syntactic category*.

Well-kindedness serves to distinguish 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.

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

$$\Gamma \vdash t : T$$

Here, it is split 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 traditional type system uses a list Γ of *type assumptions*:

$$\Gamma \vdash t : T$$

Here, it is split into a list K of *kind assumptions* and a *permission* P:

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

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

precondition

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

$$\Gamma \vdash t : T$$

Here, it is split into a list K of *kind assumptions* and a *permission* P:

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

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

postcondition

What is needed to type-check an *inert* program?

- one introduction rule for each type construct (5 of them);
- one rule for each term construct (2 of them);
- a few non-syntax-directed rules (Cut, ExistsElim, Sub);
- and a bunch of subsumption rules.

More is needed to check a *running* program; discussed later on.

A variable x has type $=x$ in the absence of *any* assumption.

$$K; P \vdash v : {=}v$$

The introduction rule for $T \mid Q$ is also the *frame rule*.

$$\frac{K; P \vdash t : T}{K; P * Q \vdash t : T \mid Q}$$

lambda *separately* extends K and P.

$$\frac{K, x : \text{value}; P * x \mathbin{@} T \vdash t : U}{K; (\text{duplicable } P) * P \vdash \lambda x.t : T \rightarrow U}$$

The *duplicable* facts that hold when the function is defined remain valid when the function is invoked.

a kind assumption

lambda *separately* extends K and P.

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

The *duplicable* facts that hold when the function is defined remain valid when the function is invoked.

a part of the precondition

lambda *separately* extends K and P.

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

The *duplicable* facts that hold when the function is defined remain valid when the function is invoked.

lambda *separately* extends K and P.

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

The *duplicable* facts that hold when the function is defined remain valid when the function is invoked.

`this is a permission!`

Universal quantifier introduction is restricted to *harmless* terms.

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

They include values, memory allocation, but not *lock* allocation.

The well-known interaction between polymorphism and mutable state is really between polymorphism and *hidden* state.

Existential quantifier introduction.

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

Syntax-directed rules for terms (1/2)

Function application.

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

Syntax-directed rules for terms (1/2)

Function application.

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

an assumption about a value
expressed as part of the precondition

Spawning a thread is a like a function call,

$$K; (v_1 \mathbin{@} T \rightarrow U) \ast (v_2 \mathbin{@} T) \vdash \mathsf{spawn}\ v_1\ v_2 : \top$$

but produces a unit result.

Cut hides a part of the precondition, P_1, that happens to be "true".

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

Non-syntax-directed rules (1/3)

Cut hides a part of the precondition, P_1, that happens to be "true".

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

permission interpretation judgement discussed later on

Existential quantifier elimination.

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

Subsumption is Hoare's rule of consequence.

$$\frac{K \vdash P_1 \leq P_2 \qquad K; P_2 \vdash t : T_1 \qquad K \vdash T_1 \leq T_2}{K; P_1 \vdash t : T_2}$$

Many rules. (More than 50 in the full system.) Excerpt:

$$\forall x : \kappa.P \leq [U/x]P$$

$$(v \,@\, T) \ast P \equiv v \,@\, T \mid P$$

$$v \,@\, T_1 \rightarrow T_2 \leq v \,@\, (T_1 \mid P) \rightarrow (T_2 \mid P)$$

$$(\text{duplicable } P) \ast P \leq P \ast P$$

$$\text{empty} \leq \text{duplicable } {=}v$$

$$\text{empty} \leq \text{duplicable } (T \rightarrow U)$$

$$\text{empty} \leq \text{duplicable } (\text{duplicable } \theta)$$

This axiomatization is neither *minimal* nor *complete*.

Type-checking running programs; resources

We wish to prove that *well-typed programs do not go wrong*.

But that is true of *all* programs in this trivial calculus!

We must organize the proof so that it is *robust* in the face of extensions: references, locks, adoption and abandon, etc.

Towards type soundness

We would like to prove that this affine type system *keeps correct track of ownership*, in some sense.

But there are *no resources* in this trivial calculus!

We need an *abstract notion of resource*, to be later instantiated.

E.g., a resource could be a heap fragment that one owns.

Axiomatization of resources

We need some tools to reason abstractly about resources.

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
$R_1 \triangleleft R_2$	*tolerable interference (rely)*
	e.g., allows memory allocation

We also need a *consistency* predicate *R ok*.

- 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:

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

Technical note

In Coq, a *type class* of *monotonic separation algebras*.

Currently 7 instances, and combinations thereof!

You want $\star$ to be represented as a *total function*.

Thomas Braibant's *AAC* plugin is very useful.

We assume a notion of *agreement* between a machine state s and a resource R:

$$s \sim R$$

E.g., if s is a heap and R an instrumented heap (fragment), then they must agree on the content of every address.

Typing judgements with resources

A typing judgement about a *running* thread must be parameterized with a resource R:

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

It reflects the thread's *view* of the machine state.

Its partial knowledge of, and assumptions about, the global state.

Typing rules with resources

The previous typing rules are extended with a parameter R.

The extension is non-trivial in two cases:

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

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

The previous typing rules are extended with a parameter R.

The extension is non-trivial in two cases:

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

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

one owns R when the function is defined
but only $\widehat{R}$ when the function is invoked

Typing rules with resources

The previous typing rules are extended with a parameter R.

The extension is non-trivial in two cases:

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

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

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

The previous typing rules are extended with a parameter R.

The extension is non-trivial in two cases:

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

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

```
permission interpretation judgement:
```
R_1 justifies P_1

The interpretation of permissions

The judgement $R; K \Vdash P$ gives meaning to permissions.

It is analogous to the semantics of separation logic, $h \Vdash F$.

The interpretation of permissions

a “semantic” object

The judgement $R; K \Vdash P$ gives meaning to permissions.

It is analogous to the semantics of separation logic, $h \Vdash F$.

a syntactic object

The judgement $R; K \Vdash P$ gives meaning to permissions.

It is analogous to the semantics of separation logic, $h \Vdash F$.

The interpretation of permissions

$$\frac{R_1; K; P \Vdash v : T \quad R_2; K \Vdash P}{R_1 \star R_2; K \Vdash v \,@\, T}$$

$$R; K \Vdash \text{empty}$$

$$\frac{R_1; K \Vdash P_1 \qquad R_2; K \Vdash P_2}{R_1 \star R_2; K \Vdash P_1 * P_2}$$

$$\frac{\theta \text{ is duplicable}}{R; K \Vdash \text{duplicable } \theta}$$

$$\frac{R; K, x : \kappa \Vdash P}{R; K \Vdash \forall x : \kappa . P}$$

$$\frac{R; K \Vdash [U/x]P}{R; K \Vdash \exists x : \kappa . P}$$

The interpretation of permissions

$v \mathbin{@} T$ holds if v has type T
mutual induction between the judgements

$$\frac{R_1; K; P \Vdash v : T \qquad R_2; K \Vdash P}{R_1 \star R_2; K \Vdash v \mathbin{@} T} \qquad R; K \Vdash \text{empty} \qquad \frac{R_1; K \Vdash P_1 \qquad R_2; K \Vdash P_2}{R_1 \star R_2; K \Vdash P_1 * P_2}$$

$$\frac{\theta \textit{ is duplicable}}{R; K \Vdash \text{duplicable } \theta} \qquad \frac{R; K, x : \kappa \Vdash P}{R; K \Vdash \forall x : \kappa . P} \qquad \frac{R; K \Vdash [U/x]P}{R; K \Vdash \exists x : \kappa . P}$$

The interpretation of permissions

we require a "canonical" derivation of $v : T$
i.e., one that does not use Sub or ExistsElim

$$\frac{R_1; K; P \Vdash v : T \qquad R_2; K \Vdash P}{R_1 \star R_2; K \Vdash v \,@\, T}$$

$$R; K \Vdash \text{empty}$$

$$\frac{R_1; K \Vdash P_1 \qquad R_2; K \Vdash P_2}{R_1 \star R_2; K \Vdash P_1 * P_2}$$

$$\frac{\theta \textit{ is duplicable}}{R; K \Vdash \text{duplicable } \theta}$$

$$\frac{R; K, x : \kappa \Vdash P}{R; K \Vdash \forall x : \kappa.P}$$

$$\frac{R; K \Vdash [U/x]P}{R; K \Vdash \exists x : \kappa.P}$$

every resource justifies empty:
affine interpretation

$$\frac{R_1; K; P \Vdash v : T \quad R_2; K \Vdash P}{R_1 \star R_2; K \Vdash v @ T}$$

$$R; K \Vdash \text{empty}$$

$$\frac{R_1; K \Vdash P_1 \qquad R_2; K \Vdash P_2}{R_1 \star R_2; K \Vdash P_1 * P_2}$$

$$\frac{\theta \textit{ is duplicable}}{R; K \Vdash \text{duplicable } \theta}$$

$$\frac{R; K, x : \kappa \Vdash P}{R; K \Vdash \forall x : \kappa.P}$$

$$\frac{R; K \Vdash [U/x]P}{R; K \Vdash \exists x : \kappa.P}$$

The interpretation of permissions

syntactic conjunction is interpreted by
"semantic" star

$$\frac{R_1; K; P \Vdash v : T \qquad R_2; K \Vdash P}{R_1 \star R_2; K \Vdash v \,@\, T}$$

$$R; K \Vdash \text{empty}$$

$$\frac{R_1; K \Vdash P_1 \qquad R_2; K \Vdash P_2}{R_1 \star R_2; K \Vdash P_1 * P_2}$$

$$\frac{\theta \textit{ is duplicable}}{R; K \Vdash \text{duplicable } \theta}$$

$$\frac{R; K, x : \kappa \Vdash P}{R; K \Vdash \forall x : \kappa . P}$$

$$\frac{R; K \Vdash [U/x]P}{R; K \Vdash \exists x : \kappa . P}$$

The interpretation of permissions

object-level predicate interpreted by
meta-level predicate (not fully satisfactory)

$$\frac{R_1; K; P \Vdash v : T \qquad R_2; K \Vdash P}{R_1 \star R_2; K \Vdash v @ T}$$

$$R; K \Vdash \text{empty}$$

$$\frac{R_1; K \Vdash P_1 \qquad R_2; K \Vdash P_2}{R_1 \star R_2; K \Vdash P_1 * P_2}$$

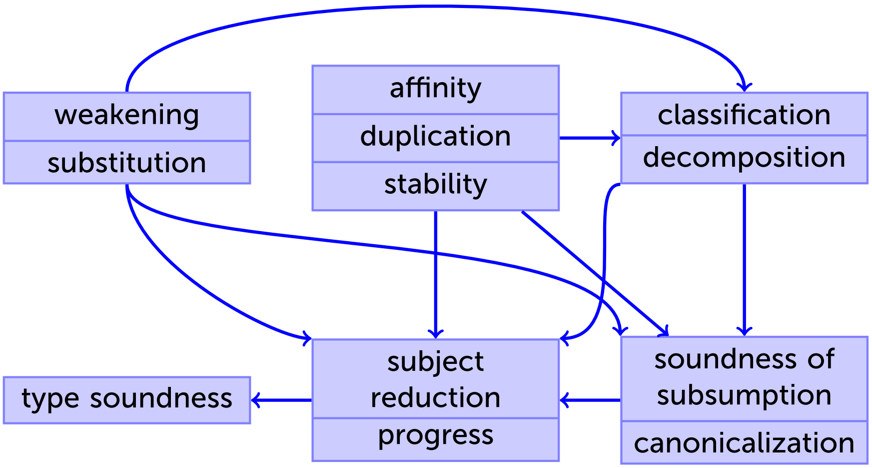
$$\frac{\theta \textit{ is duplicable}}{R; K \Vdash \text{duplicable } \theta}$$

$$\frac{R; K, x : \kappa \Vdash P}{R; K \Vdash \forall x : \kappa.P}$$

$$\frac{R; K \Vdash [U/x]P}{R; K \Vdash \exists x : \kappa.P}$$

The path to type soundness

Road map


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

Lemma (Substitution)

Let κ *be* value*,* type*, or* perm*. Typing is preserved by the substitution of an element u of kind* κ *for a variable of kind* κ*.*

$$\frac{R; K, x : \kappa; P \vdash t : T}{R; K; [u/x]P \vdash [u/x]t : [u/x]T}$$

The proof of this lemma involves 92 cases (as of now)...

Technical note

The proof of this lemma involves 92 cases (as of now)...

... and the proof script takes up 4 lines.

```
apply the_great_mutind; intros; subst; simpl_subst_goal;
try closed; try econstructor (solve [
  eauto 7 with insert_insert insert_concat
               lift_subst subst_subst j_substitution ]).
```

Of course, the hint databases must be carefully crafted.

One must sometimes reason by induction on the *size* of a type derivation.

The typing judgement is indexed with a natural integer.

We prove that substitution is size-preserving.

Lemma (Affinity)

Typing is preserved under the addition of unnecessary resources.

$$\frac{R_1; K; P \vdash t : T \qquad R_1 \star R_2 \; ok}{R_1 \star R_2; K; P \vdash t : T}$$

Lemma (Duplication)

Duplicable permissions can be justified by duplicable resources.

$$\frac{R; K \Vdash P \qquad R \text{ ok} \qquad P \text{ is duplicable}}{\widehat{R}; K \Vdash P}$$

The proof was difficult. Miraculous result?

Lemma (Stability)

Typing is preserved by tolerable interference $\lhd$.

$$\frac{R_1; K; P \vdash t : T \qquad R_1\ ok \qquad R_1 \lhd R_2}{R_2; K; P \vdash t : T}$$

One such lemma per type constructor. For functions:

Lemma (Classification)

Among the values, only λ-abstractions admit a function type.

$$\frac{R; K \Vdash v \,@\, T \to U}{\exists x,\ \exists t,\ v = \lambda x.t}$$

Easy to prove, because the hypothesis is a *canonical* derivation.

One such lemma per type constructor. For functions:

Lemma (Decomposition)

If $\lambda x.t$ has type $T \rightarrow U$, then t has type U under the assumption $x \,@\, T$.

$$\frac{R; K \Vdash \lambda x.t \,@\, T \rightarrow U \qquad R \; ok}{\widehat{R}; K, x : \text{value}; x \,@\, T \vdash t : U}$$

Easy to prove, because the hypothesis is a *canonical* derivation.

Soundness of subsumption

Lemma (Soundness of subsumption)

Permission subsumption is sound:

$$\frac{K \vdash P \leq Q \qquad R; K \Vdash P \qquad R \text{ ok}}{R; K \Vdash Q}$$

$R; K \Vdash P$ is canonical: classification and decomposition apply.

The *only* lemma where the subsumption rules play a role.

Only *one case* per subsumption rule.

It is easy to add new rules. A form of “semantic subtyping”?

Lemma (Canonicalization)

If v has type T under an empty precondition, then there is a canonical derivation of this fact.

$$\frac{R; K; \text{empty} \vdash v : T \qquad R\ ok}{R; K \Vdash v \mathbin{@} T}$$

The proof relies on

- Substitution, to eliminate ExistsElim;
- Soundness of Subsumption, to eliminate Sub.

Lemma (S.R., preliminary form)

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

one thread takes a step

Lemma (S.R., preliminary form)

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

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

Lemma (S.R., preliminary form)

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

this thread is well-typed under its view

Lemma (S.R., preliminary form)

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

Lemma (S.R., preliminary form)

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

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

Lemma (S.R., preliminary form)

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

the new machine state agrees with the new views

Lemma (S.R., preliminary form)

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

the thread remains well-typed under its view

Lemma (S.R., preliminary form)

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

the interference inflicted on the other threads is tolerable

Lemma (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 either has reached a value or is able to take a step; i.e., *no thread has gone wrong*.

Lemma (Progress)

Every well-typed configuration is acceptable.

$$\frac{\vdash c}{c \textit{ is acceptable}}$$

Well-typed programs do not go wrong.

Theorem (Type Soundness)

Assume $void; \varnothing; \text{empty} \vdash t : T$. *Then, by executing* $initial \;/\; t$, *one can reach only acceptable configurations.*

Outline

- Introduction
- The kernel
- Extensions
  - References
  - Locks
  - Adoption and abandon
- Conclusion

What's an extension?

An extension typically involves:

- *syntax*, *dynamic semantics*:
  - new terms;
  - new machine state components;
  - new reduction rules.
- *static semantics* of *inert* programs:
  - new types;
  - new typing rules, new subsumption rules.
- *static semantics* of *running* programs:
  - new resource components;
  - yet more typing rules!
- *proofs*:
  - new proof cases in the main lemmas; new auxiliary lemmas.

References

Why references?

References are simplified memory blocks:

- only one field;
- no tag;
- mutable or immutable; freezing is supported.

New terms:

$$
\begin{array}{lcl}
v & ::= & \ldots \mid \ell \\
t & ::= & \ldots \mid \mathsf{newref}\ v \mid !v \mid v := v
\end{array}
$$

New machine state component:

- a *heap* maps an initial segment of $\mathbb{N}$ to values.

New reduction rules:

initial config.		*new configuration*	*side condition*
h / newref v	$\longrightarrow$	$h +\!\!+ v$ / *limit* h	
h / $!\ell$	$\longrightarrow$	h / v	$h(\ell) = v$
h / $\ell := v'$	$\longrightarrow$	$h[\ell \mapsto v']$ / $()$	$h(\ell) = v$

New types:

$$\begin{array}{rcl} T & ::= & \ldots \mid \text{ref}_m\ T \\ m & ::= & D \mid X \end{array}$$

New typing rules:

$$R; K; v \mathbin{@} T \vdash \text{newref } v : \text{ref}_m\ T$$

$$R; K; (\text{duplicable } T) * (v \mathbin{@} \text{ref}_m\ T) \vdash !v : T \mid (v \mathbin{@} \text{ref}_m\ T)$$

$$R; K; (v \mathbin{@} \text{ref}_X\ T) * (v' \mathbin{@} T') \vdash v := v' : \top \mid (v \mathbin{@} \text{ref}_X\ T')$$

New subsumption rules:

$$v \,@\, \mathsf{ref}_m\ T \equiv \exists x : \mathsf{value}.((v \,@\, \mathsf{ref}_m\ {=}x) * (x \,@\, T))$$

$$\frac{T \leq U}{v \,@\, \mathsf{ref}_m\ T \leq v \,@\, \mathsf{ref}_m\ U}$$

Type-checking running programs

New resource component:

- An *instrumented heap* maps memory locations to instrumented values.
- An *instrumented value* is $\lightning$, N, $D\,v$, or $X\,v$.
- The composition of resources satisfies:

$$
\begin{array}{rcl}
N \star X\,v & = & X\,v \\
N \star N & = & N \\
D\,v \star D\,v & = & D\,v
\end{array}
$$

Separation at mutable locations; *agreement* at immutable locations.

Agreement between a value and an instrumented value:

v and m v agree

(Just ignore the mutability flag.)

Agreement between raw and instrumented heaps ($s \sim R$): pointwise.

Type-checking running programs

New typing rule for memory locations:

$$\frac{R_1; K \Vdash v \,@\, T \qquad R_2(\ell) = m\, v}{R_1 \star R_2; K; P \vdash \ell : \mathsf{ref}_m\, T}$$

Introduces (gives meaning to) the type $\mathsf{ref}_m\ T$,
by connecting it with an *instrumented heap fragment* $R_1 \star R_2$:

- R_2 guarantees that ℓ holds some value v;
- if m is X, R_2 guarantees exclusive knowledge of this fact;
- *and* (separately) R_1 guarantees that v has type T.

Theorem

Well-typed programs do not go wrong.

"Just" a matter of dealing with the new proof cases.

A *data race* occurs when two distinct threads are ready to access a single location, and one of the accesses is a write.

Theorem

Well-typed programs are data race free.

The proof is immediate: writing requires exclusive ownership.

$$X\, v_1 \star m\, v_2 = \lightning$$

Locks

Syntax and dynamic semantics

New terms:

$$\begin{array}{rcl} v & ::= & \ldots \mid k \\ t & ::= & \ldots \mid \text{newlock} \mid \text{acquire } v \mid \text{release } v \end{array}$$

New machine state component:

- a *lock heap* maps an initial segment of $\mathbb{N}$ to U (unlocked) or L (locked).

New reduction rules:

initial config.		*new configuration*	*side condition*
kh / newlock	$\longrightarrow$	$kh +\!\!+ L$ / *limit kh*	
kh / acquire k	$\longrightarrow$	$kh[k \mapsto L]$ / $()$	$kh(k) = U$
kh / release k	$\longrightarrow$	$kh[k \mapsto U]$ / $()$	$kh(k) = L$

Type-checking inert programs

New types:

$$T \quad ::= \quad \ldots \mid \mathsf{lock}\ P \mid \mathsf{locked}$$

New typing rules:

$$R; K; Q \vdash \mathsf{newlock} : \exists x : \mathsf{value}.(=x \mid (x \,@\, \mathsf{lock}\ P) * (x \,@\, \mathsf{locked}))$$

$$R; K; v \,@\, \mathsf{lock}\ P \vdash \mathsf{acquire}\ v : \top \mid P * (v \,@\, \mathsf{locked})$$

$$R; K; P * (v \,@\, \mathsf{locked}) * (v \,@\, \mathsf{lock}\ P) \vdash \mathsf{release}\ v : \top$$

Type-checking running programs

New resource component:

- An *instrumented lock heap* maps lock addresses to instrumented lock statuses.
- An *instrumented lock status* is a pair of:
  - a *lock invariant*: a closed permission P;
  - an *access right*: one of $\lightning$, N, and X.
- The composition of resources satisfies:

$$\begin{aligned} P \star P &= P \\ N \star X &= X \\ N \star N &= N \end{aligned}$$

Agreement on the lock invariant; *separation* concerning the ownership of a locked lock.

New resource component:

- An *instrumented lock heap* maps lock addresses to instrumented lock statuses.
- An *instrumented lock status* is a pair of:
  - a *lock invariant*: a closed permission P;
  - an *access right*: one of $\lightning$, N, and X.
- The composition of resources satisfies:

$$\begin{aligned} P \star P &= P \\ N \star X &= X \\ N \star N &= N \end{aligned}$$

syntax!

Agreement on the lock invariant; *separation* concerning the ownership of a locked lock.

Type-checking running programs

Agreement between a lock status and an instrumented lock status:

$$U \text{ and } (P, N) \text{ agree}$$
$$L \text{ and } (P, X) \text{ agree}$$

(Just ignore the invariant *P*.)

Pointwise agreement between raw and instrumented lock heaps is written *s and R agree*.

On top of this, *a more elaborate notion of agreement* is defined:

$$\frac{\textit{s and } R \star R' \textit{ agree} \qquad R' ; \varnothing \Vdash \text{hidden invariants of } (R \star R')}{s \sim R}$$

With this definition, the type soundness statements are unchanged.

> the conjunction of the invariants
> of all presently unlocked locks

On top of this, *a more elaborate notion of agreement* is defined:

$$\frac{s \text{ and } R \star R' \text{ agree} \qquad R' ; \varnothing \Vdash \text{hidden invariants of } (R \star R')}{s \sim R}$$

With this definition, the type soundness statements are unchanged.

> a fragment of the instrumented state that justifies this conjunction

On top of this, *a more elaborate notion of agreement* is defined:

$$\frac{s \text{ and } R \star R' \text{ agree} \qquad R', \varnothing \Vdash \text{hidden invariants of } (R \star R')}{s \sim R}$$

With this definition, the type soundness statements are unchanged.

Type-checking running programs

> the fragment of the instrumented state that remains visible to the program

On top of this, *a more elaborate notion of agreement* is defined:

$$\frac{\begin{array}{c} s \text{ and } R \star R' \text{ agree} \\ R' ; \varnothing \Vdash \text{hidden invariants of } (R \star R') \end{array}}{s \sim R}$$

With this definition, the type soundness statements are unchanged.

Type-checking running programs

New typing rules for lock addresses:

$$\frac{R(k) = (P, \_)}{R; K; Q \vdash k : \text{lock } P} \qquad \frac{R(k) = (\_, X)}{R; K; Q \vdash k : \text{locked}}$$

Introduce (give meaning to) the types lock P and locked.

A configuration is now *acceptable* if every thread:

- has reached a value,
- is able to take a step,
- or *is waiting on a lock* that is currently held.

The type discipline does not prevent deadlocks.

Theorem

Well-typed programs do not go wrong.

"Just" a matter of dealing with the new proof cases.

Adoption and abandon

No new values.

New terms:

$$t \quad ::= \quad \ldots \mid \text{give } v_1 \text{ to } v_2 \mid \text{take } v_1 \text{ from } v_2 \mid \text{fail} \mid \text{take! } v_1 \text{ from } v_2$$

Updated machine state component:

- the heap maps a memory location to a pair of *an adopter pointer* $p ::= \mathit{null} \mid \ell$ and a value.

New reduction rules:

initial config.		*new configuration*		*side condition*
h / give ℓ to ℓ'	$\longrightarrow$	$h[\ell \mapsto \langle\, \ell' \mid v \,\rangle]$	/ ()	$h(\ell) = \langle\, p \mid v \,\rangle$
h / take ℓ from ℓ'	$\longrightarrow$	h	/ take! ℓ from ℓ'	$h(\ell) = \langle\, \ell' \mid v \,\rangle$
h / take ℓ from ℓ'	$\longrightarrow$	h	/ fail	$h(\ell) = \langle\, p \mid v \,\rangle \wedge p \neq \ell'$
h / take! ℓ from ℓ'	$\longrightarrow$	$h[\ell \mapsto \langle\, \mathit{null} \mid v \,\rangle]$	/ ()	$h(\ell) = \langle\, p \mid v \,\rangle$
s / E[fail]	$\longrightarrow$	s	/ fail	

Note that `take` does *not* need an atomic implementation.

Type-checking inert programs

New types:

$$T \quad ::= \quad \ldots \mid \text{adoptable} \mid \text{unadopted} \mid \text{adopts } T$$

Intuitively,

- v @ adoptable is `v @ dynamic`; it is duplicable;
  - guarantees the existence of v's `adopter` field;
  - allows an attempt to **`take`** v from its adopter.
- v @ unadopted means we own v as a potential adoptee; *affine*;
  - guarantees that v's `adopter` field exists and is *null*;
  - allows to **`give`** v to some adopter.
- v @ adopts T means we own v as an adopter; it is *affine*;
  - asserts that every adoptee of v has type T;
  - represents the collective ownership of all such adoptees.

Type-checking inert programs

Modified typing rule for memory allocation:

$$R; K; v \mathbin{@} T \vdash \textbf{newref } v : \exists x : \text{value}.(=x \mid (x \mathbin{@} \text{ref}_m\ T) * (x \mathbin{@} \text{adopts } \bot) * (x \mathbin{@} \text{unadopted}))$$

The value *x* produced by newref *v*:

- is the address of a memory block, as before;
- can be used as an adopter (and presently has no adoptees);
- can be used as an adoptee (i.e., is presently not adopted).

New typing rules for adoption and abandon:

$$\begin{array}{l} R; K; (v_2 \mathbin{@} \text{adopts } U) * (v_1 \mathbin{@} U) * (v_1 \mathbin{@} \text{unadopted}) \\ \quad \vdash \textbf{give } v_1 \textbf{ to } v_2 : \top \mid \\ \quad (v_2 \mathbin{@} \text{adopts } U) \end{array}$$

$$\begin{array}{l} R; K; (v_2 \mathbin{@} \text{adopts } U) * (v_1 \mathbin{@} \text{adoptable}) \\ \quad \vdash \textbf{take } v_1 \textbf{ from } v_2 : \top \mid \\ \quad (v_2 \mathbin{@} \text{adopts } U) * (v_1 \mathbin{@} U) * (v_1 \mathbin{@} \text{unadopted}) \end{array}$$

Type-checking inert programs

New subsumption rules:

$$\text{empty} \leq \text{duplicable adoptable}$$

$$v \mathbin{@} \text{unadopted} \leq (v \mathbin{@} \text{unadopted}) * (v \mathbin{@} \text{adoptable})$$

$$\frac{T \leq U}{v \mathbin{@} \text{adopts } T \leq v \mathbin{@} \text{adopts } U}$$

Type-checking running programs

New resource component:

- A *raw adoption resource* maps a memory location to a pair of an adoptee status and an adopter status.
- An *adoptee status* is $\lightning$, N, or $X\ p$.
- An *adopter status* is $\lightning$, N, or X.

Auxiliary definitions:

- $R \vdash \ell$ *is adoptable* iff ℓ is in the domain of R.
- $R \vdash \ell$ *is unadopted* iff R maps ℓ to $(X\ null, \_)$.
- $R \vdash \ell'$ *is an adopter* iff R maps ℓ' to $(\_, X)$.
- $R \vdash \vec{\ell}$ *are the adoptees of* ℓ' iff:
  - $R \vdash \ell'$ *is an adopter*; and
  - $\vec{\ell}$ lists *the* addresses ℓ such that $R \vdash \ell$ *is adopted by* ℓ';

Type-checking running programs

We would like "$\cdot \vdash \vec{\ell}$ *are the adoptees of* ℓ'" to be affine, i.e.:

$$\frac{R_1 \vdash \vec{\ell} \textit{ are the adoptees of } \ell'}{R_1 \star R_2 \vdash \vec{\ell} \textit{ are the adoptees of } \ell'}$$

But this does *not* hold.

R_2 could own an adoptee of ℓ'.

There would be a *dangling adopter edge* out of R_2.

We avoid this issue by forbidding dangling adopter edges.

An adoption resource R is *round* if

$R \vdash \ell$ is adopted by ℓ' implies $R \vdash \ell'$ is an adopter.

Roundness is preserved by $\star$ and by $\lhd$, which means we can work in the subset of round resources.

Three new typing rules for memory locations!

$$\frac{R \vdash \ell \textit{ is adoptable}}{R; K; P \vdash \ell : \text{adoptable}} \qquad \frac{R \vdash \ell \textit{ is unadopted}}{R; K; P \vdash \ell : \text{unadopted}}$$

$$\frac{\begin{array}{c} R_1 \vdash \vec{\ell} \textit{ are the adoptees of } \ell' \\ R_2; K \Vdash \vec{\ell} \,@\, U \end{array}}{R_1 \star R_2; K; P \vdash \ell' : \text{adopts } U}$$

They give meaning to the three new types.

New typing rules for terms:

$$R; K; P \vdash \text{fail} : T$$

$$\frac{R; K \Vdash \ell' \mathbin{@} \text{adopts } U \qquad R \vdash \ell \textit{ is adopted by } \ell'}{\begin{array}{l} R; K; P \vdash \text{take! } \ell \text{ from } \ell' : \top \mid \\ \qquad (\ell' \mathbin{@} \text{adopts } U) * (\ell \mathbin{@} U) * (\ell \mathbin{@} \text{unadopted}) \end{array}}$$

Theorem

Well-typed programs do not go wrong.

"Just" a matter of dealing with the new proof cases.

Outline

The Coq proof is currently 14K non-blank, non-comment lines.

- de Bruijn index library (2K) (reusable);
- MSA library (2K) (reusable);
- kernel (4K);
- references, locks, adoption and abandon (6K).

What is missing

An earlier version of the proof had the following features:

- memory blocks with *multiple fields*;
- memory blocks with a *tag*; tag update instruction;
- *sum types*; `match` instruction;
- (parameterized) *iso-recursive types*.

We need to add them back in.

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

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

Conclusion

Concerning the meta-theory:

- The good old *syntactic approach* to type soundness works.
- Formalization *helped tremendously* clarify and simplify the design.

Concerning Mezzo:

- *Type inference* and type error reports need more research.
- Does Mezzo help write correct programs?
  Does it help prove programs correct?

Thank you

More information online:

`http://gallium.inria.fr/~protzenk/mezzo-lang/`