The theory of Mezzo

François Pottier

INRIA

IHP, April 2014
Acknowledgements

Outline

• Introduction
• The kernel
• Extensions
• Conclusion
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 \( \lambda \)-calculi, hence no *consumes* keyword.
Recall the type of the `length` function for mutable lists.

\[
[a] \text{mlist } a \rightarrow \text{int}
\]

In Surface syntax, this could also be written:

\[
[a] \quad (\text{consumes } xs : \text{mlist } a) \rightarrow \\
\quad (\text{int } \mid xs @ \text{mlist } a)
\]

or, by exploiting universal quantification and a singleton type:

\[
[a, xs : \text{value}] \\
\quad (=xs \mid \text{consumes } xs @ \text{mlist } a) \rightarrow \\
\quad (\text{int } \mid xs @ \text{mlist } a)
\]

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) -&gt; ())

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 $\lambda$-calculus.

$$\kappa ::= \text{value} \mid \text{term} \mid \text{soup} \mid \ldots \quad \text{(Kinds)}$$

$$v ::= x \mid \lambda x.t \quad \text{(Values)}$$

$$t ::= v \mid v \ t \mid \text{spawn} \ v \ v \quad \text{(Terms)}$$

$$sp ::= \text{thread} \ (t) \mid sp \ || \ sp \quad \text{(Soups)}$$

$$E ::= v \ [] \quad \text{(Shallow evaluation contexts)}$$

$$D ::= [] \mid E[D] \quad \text{(Deep evaluation contexts)}$$
A fairly unremarkable untyped λ-calculus.

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

A fairly unremarkable untyped $\lambda$-calculus.

\[
\kappa ::= \text{value} \mid \text{term} \mid \text{soup} \mid \ldots \quad \text{(Kinds)}
\]

\[
\nu ::= x \mid \lambda x.t \quad \text{(Values)}
\]

\[
t ::= \nu \mid \nu t \mid \text{spawn } \nu \nu \quad \text{(Terms)}
\]

\[
sp ::= \text{thread } (t) \mid sp \parallel sp \quad \text{(Soups)}
\]

\[
E ::= \nu [] \quad \text{(Shallow evaluation contexts)}
\]

\[
D ::= [] \mid E[D] \quad \text{(Deep evaluation contexts)}
\]

a primitive construct for spawning a new thread
### Operational semantics

**initial configuration**

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

\[ \longrightarrow s / [v/x]t \]

**new configuration**

\[ s / E[t] \]

\[ \longrightarrow s' / E[t'] \]

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 \[ s / t_2 \longrightarrow s' / t'_2 \]

\[ s / \text{thread} \ (D[\text{spawn} \ v_1 \ v_2]) \]

\[ \longrightarrow s / \text{thread} \ (D[()]) \parallel \text{thread} \ (v_1 \ v_2) \]
initial configuration

\[ s / \lambda x. t \ v \]

\[ s / E[t] \]

\[ s / \text{thread}(t) \]

\[ s / t_1 \parallel t_2 \]

\[ s / \text{thread}(D[\text{spawn} v_1 v_2]) \]

new configuration

\[ \rightarrow s / [v/x] t \]

\[ \rightarrow s' / E[t'] \quad \text{if } s / t \rightarrow s' / t' \]

\[ \rightarrow s' / \text{thread}(t') \quad \text{if } s / t \rightarrow s' / t' \]

\[ \rightarrow s' / t'_1 \parallel t_2 \quad \text{if } s / t_1 \rightarrow s' / t'_1 \]

\[ \rightarrow s' / t_1 \parallel t'_2 \quad \text{if } s / t_2 \rightarrow s' / t'_2 \]

\[ \rightarrow s / \text{thread}(D[()] \parallel \text{thread}(v_1 v_2) \]

an abstract notion of machine state
Type-checking inert programs
Types and permissions

\[ \kappa ::= \ldots \mid \text{type} \mid \text{perm} \quad \text{(Kinds)} \]

\[ T, U ::= x \mid =v \mid T \rightarrow T \mid (T \mid P) \quad \text{(Types)} \]
\[ \forall x : \kappa.T \mid \exists x : \kappa.T \]

\[ P, Q ::= x \mid v @ T \mid \text{empty} \mid P * P \quad \text{(Permissions)} \]
\[ \forall x : \kappa.P \mid \exists x : \kappa.P \]
\[ \text{duplicable } \theta \]

\[ \theta ::= T \mid P \]
In the Coq formalisation, only one syntactic category. Well-kindness 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 $\Gamma$ of \textit{type assumptions}:

$$\Gamma \vdash t : T$$

Here, it is split into a list $K$ of \textit{kind assumptions} and a \textit{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 $\Gamma$ of \textit{type assumptions}:

$$\Gamma \vdash t : T$$

Here, it is split into a list $K$ of \textit{kind assumptions} and a \textit{permission} $P$:

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

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

\textbf{precondition}
The type discipline

A traditional type system uses a list $\Gamma$ 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 \textit{any} assumption.

$$K; P \vdash v : =v$$
The introduction rule for $T \mid Q$ is also the **frame rule**.

$$
\frac{K; P \vdash t : T}{K; P \star Q \vdash t : T \mid Q}
$$
lambda \textit{separately} extends $K$ and $P$.

\[
K, x : \text{value}; P \ast x @ T \vdash t : U
\]

\[
K; (\text{duplicable } P) \ast P \vdash \lambda x.t : T \rightarrow U
\]

The \textit{duplicable} facts that hold when the function is defined remain valid when the function is invoked.
lambda \textit{separately} extends $K$ and $P$.

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

The \textit{duplicable} facts that hold when the function is defined remain valid when the function is invoked.
lambda separately extends $K$ and $P$.

\[
K, x : \text{value}; P \ast x \in T \vdash t : U
\]
\[
K; (\text{duplicable } P) \ast P \vdash \lambda x.t : T \rightarrow U
\]

The duplicable facts that hold when the function is defined remain valid when the function is invoked.
lambda \textit{separately} extends \( K \) and \( P \).

\[
\begin{align*}
K, x : \text{value}; & \quad P \ast x @ T \vdash t : U \\
K; (\text{duplicable } P) \ast P \vdash \lambda x. t : T \rightarrow U
\end{align*}
\]

The \textit{duplicable} facts that hold when the function is defined remain valid when the function is invoked.

\textbf{this is a permission!}
Universal quantifier introduction is restricted to *harmless* terms.

\[
\begin{align*}
t \text{ is harmless} \\
K, x : \kappa; P \vdash t : T \\
\hline
K; \forall x : \kappa. P \vdash t : \forall x : \kappa. T
\end{align*}
\]

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.

\[ K; P \vdash v : [U/x]T \]

\[ K; P \vdash v : \exists x : \kappa. T \]
Function application.

\[
K; Q \vdash t : T \\
\frac{K; (v @ T \to U) \ast Q \vdash v t : U}{K; Q \vdash t : T}
\]
Function application.

\[
K; Q \vdash t : T \\
K; (v @ T \rightarrow U) \ast Q \vdash v \ t : U
\]

an assumption about a value expressed as part of the precondition
Syntax-directed rules for terms (2/2)

Spawning a thread is like a function call,

\[ K; (v_1 @ T \rightarrow U) \ast (v_2 @ T) \vdash \text{spawn } v_1 \; v_2 : \top \]

but produces a unit result.
Cut hides a part of the precondition, $P_1$, that happens to be “true”.

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

$$\frac{}{K; P_2 \vdash t : T}$$
Cut hides a part of the precondition, $P_1$, that happens to be "true".

\[
\frac{K; P_1 \ast P_2 \vdash t : T}{\frac{K \models P_1}{K; P_2 \vdash t : T}}
\]

Permission interpretation judgement discussed later on.
Existential quantifier elimination.

\[ K, x : \kappa; P \vdash t : T \]

\[ K; \exists x : \kappa. P \vdash t : T \]
Subsumption is Hoare's rule of consequence.

\[
\frac{K \vdash P_1 \leq P_2 \quad K; P_2 \vdash t : T_1 \quad K \vdash T_1 \leq T_2}{K; P_1 \vdash t : T_2}
\]
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 \textit{minimal} nor \textit{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.
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 \( \\downarrow, N, X_v, \) or \( D_v \)

- \( R_1 \star R_2 \) conjunction
  - e.g., requires separation at mutable addresses
  - requires agreement at immutable addresses

- \( \hat{R} \) 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 \( R \text{ ok} \).
Axiomatization of resources

- Star $\star$ is commutative and associative.
- $R_1 \star R_2 \ ok$ implies $R_1 \ ok$.
- $R \star \hat{R} = R$.
- $R_1 \star R_2 = R$ and $R \ ok$ imply $\hat{R}_1 = \hat{R}$.
- $R \star R = R$ implies $R = \hat{R}$.
- $\hat{R} \star \hat{R} = \hat{R}$.
- $R \triangleleft R$.
- $R_1 \ ok$ and $R_1 \triangleleft R_2$ imply $R_2 \ ok$.
- $R_1 \triangleleft R_2$ implies $\hat{R}_1 \triangleleft \hat{R}_2$.
- rely preserves splits:

$$R_1 \star R_2 \triangleleft R' \quad R_1 \star R_2 \ ok$$

$$\exists R'_1R'_2, \ R'_1 \star R'_2 = R' \land R_1 \triangleleft R'_1 \land R_2 \triangleleft R'_2$$
In Coq, a *type class* of *monotonic separation algebras*. Currently 7 instances, and combinations thereof!
You want ⋆ 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.
The previous typing rules are extended with a parameter $R$. The extension is non-trivial in two cases:

\[
\begin{align*}
\frac{\hat{R}; K, x : \text{value}; P * x @ T \vdash t : U}{R; K; (\text{duplicable } P) * P \vdash \lambda x. t : T \rightarrow U}
\end{align*}
\]
The previous typing rules are extended with a parameter $R$. The extension is non-trivial in two cases:

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

one owns $R$ when the function is defined but only $\hat{R}$ when the function is invoked
The previous typing rules are extended with a parameter $R$. The extension is non-trivial in two cases:

- If a typing rule has two premises, then $R$ must be split between them.

\[
\begin{align*}
\hat{R}; K, x : \text{value}; P \times x @ T & \vdash t : U \\
R; K; (\text{duplicable } P) \times P & \vdash \lambda x.t : T \rightarrow U
\end{align*}
\]
The previous typing rules are extended with a parameter $R$. The extension is non-trivial in two cases:

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

$$
\frac{R_2; K; P_1 \ast P_2 \vdash t : T}{R_1; K \parallel P_1}
\quad
\frac{R_1 \ast R_2; K; P_2 \vdash t : T}{R_1 \parallel R_2; K; P_2 \vdash t : T}
$$

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

The judgement $R, K \vdash P$ gives meaning to permissions.

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

a “semantic” object
The interpretation of permissions

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

It is analogous to the semantics of separation logic, $h \models F$. 
The interpretation of permissions

\[
\begin{align*}
R_1; K; P & \vdash v : T \\
R_2; K & \vdash P \\
\overline{R_1 \star R_2; K \vdash v \, @ \, T}
\end{align*}
\]

\[
\begin{align*}
\theta & \text{ is duplicable} \\
\overline{R; K \vdash \text{duplicable } \theta}
\end{align*}
\]

\[
\begin{align*}
R; K; x : \kappa & \vdash P \\
\overline{R; K \vdash \forall x : \kappa. P}
\end{align*}
\]

\[
\begin{align*}
R; K & \vdash [U/x]P \\
\overline{R; K \vdash \exists x : \kappa. P}
\end{align*}
\]

\[
\begin{align*}
R_1; K & \vdash P_1 \\
R_2; K & \vdash P_2 \\
\overline{R_1 \star R_2; K \vdash P_1 \star P_2}
\end{align*}
\]
The interpretation of permissions

\( \nu \at T \) holds if \( \nu \) has type \( T \)

mutual induction between the judgements

\[
\begin{align*}
R_1; K; P \vdash \nu : T \\
R_2; K \vdash P \\
\frac{R_1 \ast R_2; K \vdash \nu @ T}{R; K \vdash \text{empty}} \\
\end{align*}
\]

\( \theta \) is duplicable

\[
\frac{R; K \vdash \text{duplicable } \theta}{R; K \vdash \text{duplicable } \theta}
\]

\[
\begin{align*}
R_1; K; x : \kappa \vdash P \\
\frac{R; K \vdash \forall x : \kappa. P}{R; K \vdash \exists x : \kappa. P}
\end{align*}
\]
The interpretation of permissions

we require a “canonical” derivation of \( v : T \)
i.e., one that does not use Sub or ExistsElim

\[
\frac{R_1; K; P \vdash v : T}{R_2; K \vdash P}
\]
\[
\frac{R_1 \times R_2; K \vdash v @ T}{\emptyset}
\]

\[
\frac{\theta \text{ is duplicable}}{R; K \vdash \text{duplicable } \theta}
\]

\[
\frac{R; K, x : \kappa \vdash P}{R; K \vdash \forall x : \kappa. P}
\]

\[
\frac{R; K \vdash [U/x]P}{R; K \vdash \exists x : \kappa. P}
\]
The interpretation of permissions

every resource justifies empty: affine interpretation

\[
\begin{align*}
R_1; K; P & \not\models v : T \\
R_2; K & \not\models P \\
\frac{}{R_1 \star R_2; K \not\models v @ T}
\end{align*}
\]

\[
\begin{align*}
\theta \text{ is duplicable} \quad & \quad R; K, x : \kappa & \not\models P \\
R; K & \not\models \forall x : \kappa. P \\
\end{align*}
\]

\[
\begin{align*}
R_1; K & \not\models P_1 \\
R_2; K & \not\models P_2 \\
\frac{}{R_1 \star R_2; K \not\models P_1 \star P_2}
\end{align*}
\]

\[
\begin{align*}
R; K & \not\models \exists x : \kappa. P \\
\frac{}{R; K \models [U/x]P}
\end{align*}
\]
The interpretation of permissions

syntactic conjunction is interpreted by “semantic” star

\[
\begin{align*}
R_1; K; P \models v : T \\
R_2; K \models P \\
\vdash R_1 \star R_2; K \models v@T
\end{align*}
\]

\[
\begin{align*}
R; K \models empty \\
R_1; K \models P_1 \\
R_2; K \models P_2 \\
\vdash R_1 \star R_2; K \models P_1 \star P_2
\end{align*}
\]

\[
\begin{align*}
\theta \text{ is duplicable} \\
\vdash R; K \models \text{duplicable } \theta
\end{align*}
\]

\[
\begin{align*}
R; K, x : \kappa \models P \\
\vdash R; K \models \forall x : \kappa. P
\end{align*}
\]

\[
\begin{align*}
R; K \models [U/x]P \\
\vdash R; K \models \exists x : \kappa. P
\end{align*}
\]
The interpretation of permissions

\[ R_1; K; P \not\models v : T \]
\[ \frac{R_2; K \models P}{R_1 \times R_2; K \models v @ T} \]

\[ \theta \text{ is duplicable} \]
\[ \frac{R; K \not\models \text{duplicable } \theta}{R; K \not\models \theta} \]

\[ R_1; K \models P_1 \]
\[ R_2; K \models P_2 \]
\[ \frac{R_1 \times R_2; K \models P_1 \times P_2}{R_1 \times R_2; K \models \emptyset} \]

\[ \text{object-level predicate interpreted by meta-level predicate (not fully satisfactory)} \]

\[ R; K \models \forall x : \kappa. P \]
\[ R; K \models [U/x]P \]
\[ R; K \models \exists x : \kappa. P \]
The path to type soundness
Road map

- weakening
- substitution
- affinity
- duplication
- stability
- classification
- decomposition
- subject
- reduction
- progress
- soundness of
- subsumption
- canonicalization
- type soundness
Lemma (Substitution)

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

$$
\frac{R; K, x : \kappa; P \vdash t : T}{R; K; [u/x]P \vdash [u/x]t : [u/x]T}
$$
The proof of this lemma involves 92 cases (as of now)...

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 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.*

\[
\frac{R_1; K; P \vdash t : T \quad R_1 \star R_2 \text{ ok}}{R_1 \star R_2; K; P \vdash t : T}
\]
**Lemma (Duplication)**

_Duplicable permissions can be justified by duplicable resources._

\[
\begin{align*}
R; K \not\models P & \quad R \text{ ok} & \quad P \text{ is duplicable} \\
\hline
\end{align*}
\]

\[
\begin{align*}
\hat{R}; K \not\models P
\end{align*}
\]

The proof was difficult. Miraculous result?
Lemma (Stability)

Typing is preserved by tolerable interference $\triangleleft$.

\[
\frac{R_1; K; P \vdash t : T \quad R_1 \text{ ok} \quad R_1 \triangleleft R_2}{R_2; K; P \vdash t : T}
\]
One such lemma per type constructor. For functions:

**Lemma (Classification)**

*Among the values, only $\lambda$-abstractions admit a function type.*

\[
R; K \vdash v @ T \to U \\
\exists x, \exists t, v = \lambda x.t
\]

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* \( x @ T \).

\[
\frac{
R; K \vdash \lambda x. t @ T \rightarrow U \quad R \ ok
}{
\widehat{R}; K, x : \text{value}; x @ T \vdash t : U
}
\]

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

Lemma (Soundness of subsumption)

Permission subsumption is sound:

\[
\frac{K \vdash P \leq Q \quad R; K \vdash P \quad R \text{ ok}}{R; K \vdash Q}
\]

\(R; K \vdash P\) 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.

\[
\begin{align*}
R; K; \text{empty} & \vdash v : T & R \text{ ok} \\
\hline
R; K & \not\vdash v @ T
\end{align*}
\]

The proof relies on

- Substitution, to eliminate ExistsElim;
- Soundness of Subsumption, to eliminate Sub.
Lemma (S.R., preliminary form)

\[ s_1 / t_1 \rightarrow s_2 / t_2 \]

\[ s_1 \sim R_1 \ast R'_1 \]

\[ R_1; \emptyset; \text{empty} \vdash t_1 : T \]

\[ \exists R_2 R'_2 \begin{cases} 
    s_2 \sim R_2 \ast R'_2 \\
    R_2; \emptyset; \text{empty} \vdash t_2 : T \\
    R'_1 \triangle R'_2
\end{cases} \]
Subject reduction

Lemma (S.R., preliminary form)

\[
\frac{s_1 / t_1 \rightarrow s_2 / t_2 \quad s_1 \sim R_1 \ast R'_1 \quad R_1; \emptyset; \text{empty} \vdash t_1 : T}{\exists R_2 R'_2 \begin{cases} s_2 \sim R_2 \ast R'_2 \\ R_2; \emptyset; \text{empty} \vdash t_2 : T \\ R'_1 \triangleleft R'_2 \end{cases}}
\]

one thread takes a step
Subject reduction

Lemma (S.R., preliminary form)

\[ s_1 \vdash t_1 \rightarrow s_2 \vdash t_2 \]

\[ s_1 \sim R_1 \star R'_1 \]

\[ R_1; \emptyset; \text{empty} \vdash t_1 : T \]

\[ \exists R_2R'_2 \left\{ \begin{array}{l}
  s_2 \sim R_2 \star R'_2 \\
  R_2; \emptyset; \text{empty} \vdash t_2 : T \\
  R'_1 \triangleleft R'_2
\end{array} \right. \]

this thread's view is \( R_1 \)
the other threads' view is \( R'_1 \)
Lemma (S.R., preliminary form)

\[
\begin{align*}
  s_1 / t_1 & \rightarrow s_2 / t_2 \\
  s_1 \sim R_1 \star R_1'
  \\
  R_1; \emptyset; \text{empty} & \triangleright t_1 : T
\end{align*}
\]

\[
\exists R_2 R_2' \left\{ 
  \begin{align*}
    s_2 & \sim R_2 \star R_2' \\
    R_2; \emptyset; \text{empty} & \triangleright t_2 : T \\
    R_1' & \triangleleft R_2'
  \end{align*}
\right\}
\]

\(\text{this thread is well-typed under its view}\)
Lemma (S.R., preliminary form)

\[
\begin{align*}
& s_1 / t_1 \longrightarrow s_2 / t_2 \\
& s_1 \sim R_1 \star R_1' \\
& R_1; \emptyset; \text{empty} \vdash t_1 : T \\
\end{align*}
\]

\[
\exists R_2 R_2' \begin{cases}
  s_2 \sim R_2 \star R_2' \\
  R_2; \emptyset; \text{empty} \vdash t_2 : T \\
  R_1' \preceq R_2'
\end{cases}
\]

this thread's view and the other threads' view evolve
Lemma (S.R., preliminary form)

\[
\begin{align*}
  s_1 / t_1 & \rightarrow s_2 / t_2 \\
  s_1 & \sim R_1 \star R'_1 \\
  R_1; \emptyset; \text{empty} & \vdash t_1 : T \\
  \exists R_2 R'_2 \quad \left\{ 
    \begin{array}{l}
      s_2 \sim R_2 \star R'_2 \\
      R_2; \emptyset; \text{empty} \vdash t_2 : T \\
      R'_1 \triangleleft R'_2
    \end{array}
  \right.
\end{align*}
\]

the new machine state agrees with the new views
Lemma (S.R., preliminary form)

\[ s_1 / t_1 \longrightarrow s_2 / t_2 \]

\[ s_1 \sim R_1 \star R'_1 \]

\[ R_1; \emptyset; \text{empty} \vdash t_1 : T \]

\[ \exists R_2 R'_2 \begin{cases} 
    s_2 \sim R_2 \star R'_2 \\
    R_2; \emptyset; \text{empty} \vdash t_2 : T \\
    R'_1 \triangleleft R'_2 
\end{cases} \]

the thread remains well-typed under its view
Lemma (S.R., preliminary form)

\[
\frac{s_1 \, / \, t_1 \rightarrow s_2 \, / \, t_2}{s_1 \sim R_1 \, \star \, R'_1}
\]

\[
\frac{R_1; \emptyset; \text{empty} \vdash t_1 : T}{\exists R_2 R'_2 \begin{cases} s_2 \sim R_2 \, \star \, R'_2 \\ R_2; \emptyset; \text{empty} \vdash t_2 : T \\ R'_1 \triangleleft R'_2 \end{cases}}
\]

the interference inflicted on the other threads is tolerable
Lemma (Subject Reduction)

Reduction preserves well-typedness.

\[
\frac{c_1 \rightarrow c_2}{\vdash c_1} \vdash c_2
\]
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.*

$$\vdash c$$

$c$ is acceptable
Well-typed programs do not go wrong.

**Theorem (Type Soundness)**

_Assume void; \emptyset; empty \vdash t : T. Then, by executing initial / t, one can reach only acceptable configurations._
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
References are simplified memory blocks:

- only one field;
- no tag;
- mutable or immutable; freezing is supported.
Syntax and dynamic semantics

New terms:

\[ v ::= \ldots \mid \ell \]
\[ t ::= \ldots \mid \text{newref } v \mid !v \mid v := v \]

New machine state component:

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

New reduction rules:

\[
\begin{array}{ccc}
\text{initial config.} & \text{new configuration} & \text{side condition} \\
\hline
h / \text{newref } v & \longrightarrow h + v & \text{limit } h \\
h / !\ell & \longrightarrow h & / v \\
h / \ell := v' & \longrightarrow h[\ell \mapsto v'] & / () \\
\end{array}
\]
Type-checking inert programs

New types:

\[
T ::= \ldots \mid \text{ref}_m T \\
\]
\[
m ::= D \mid X
\]

New typing rules:

\[
R; K; v @ T \vdash \text{newref } v : \text{ref}_m T
\]
\[
R; K; (\text{duplicable } T) \ast (v @ \text{ref}_m T) \vdash !v : T \mid (v @ \text{ref}_m T)
\]
\[
R; K; (v @ \text{ref}_X T) \ast (v' @ T') \vdash v := v' : \top \mid (v @ \text{ref}_X T')
\]
New subsumption rules:

\[ v \@ \text{ref}_m \ T \equiv \exists x : \text{value.} ((v \@ \text{ref}_m = x) \times (x @ T)) \]

\[ T \leq U \]

\[ v \@ \text{ref}_m \ T \leq v \@ \text{ref}_m \ U \]
New resource component:

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

\[
N \star X v = X v \\
N \star N = N \\
D v \star D v = D v
\]

*Separation* at mutable locations; *agreement* at immutable locations.
Agreement between a value and an instrumented value:

\[ v \text{ and } m v \text{ agree} \]

(Just ignore the mutability flag.)

Agreement between raw and instrumented heaps \((s \sim R)\): pointwise.
New typing rule for memory locations:

\[
R_1; K \vdash v @ T \quad R_2(\ell) = m v
\]

\[
R_1 \star R_2; K; P \vdash \ell : \text{ref}_m T
\]

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

- \( R_2 \) guarantees that \( \ell \) 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.

\[ X v_1 \star m v_2 = \frac{7}{\text{ }} \]
Locks
Syntax and dynamic semantics

New terms:

\[ v ::= \ldots \mid k \]
\[ t ::= \ldots \mid \text{newlock} \mid \text{acquire } v \mid \text{release } v \]

New machine state component:

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

New reduction rules:

<table>
<thead>
<tr>
<th>initial config.</th>
<th>new configuration</th>
<th>side condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>( kh / \text{newlock} )</td>
<td>( kh \leftrightarrow L )</td>
<td>limit ( kh )</td>
</tr>
<tr>
<td>( kh / \text{acquire } k )</td>
<td>( kh[k \mapsto L] )</td>
<td>()</td>
</tr>
<tr>
<td>( kh / \text{release } k )</td>
<td>( kh[k \mapsto U] )</td>
<td>()</td>
</tr>
</tbody>
</table>

\[ kh(k) = U \]
\[ kh(k) = L \]
Type-checking inert programs

New types:

\[ T ::= \ldots \mid \text{lock } P \mid \text{locked} \]

New typing rules:

\[ R; K; Q \vdash \text{newlock} : \exists x : \text{value.}(=x \mid (x @ \text{lock } P) \ast (x @ \text{locked})) \]

\[ R; K; v @ \text{lock } P \vdash \text{acquire } v : \top \mid P \ast (v @ \text{locked}) \]

\[ R; K; P \ast (v @ \text{locked}) \ast (v @ \text{lock } P) \vdash \text{release } v : \top \]
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 $\frac{1}{2}$, $N$, and $X$.
- The composition of resources satisfies:

\[
\begin{align*}
P \star P &= P \\
N \star X &= X \\
N \star N &= N
\end{align*}
\]

*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 \( \frac{1}{2} \), \( N \), and \( X \).
- The composition of resources satisfies:

  \[
  P \star P = P \\
  N \star X = X \\
  N \star N = N
  \]

*Agreement* on the lock invariant; *separation* concerning the ownership of a locked lock.
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 \( s \text{ and } R \text{ agree} \).
On top of this, *a more elaborate notion of agreement* is defined:

\[ s \text{ and } R \star R' \text{ agree} \]

\[ R'; \emptyset \not\models \text{ hidden invariants of } (R \star R') \]

\[ s \sim R \]

With this definition, the type soundness statements are unchanged.
On top of this, a more elaborate notion of agreement is defined:

\[
\text{the conjunction of the invariants of all presently unlocked locks}
\]

\[
s \text{ and } R \star R' \text{ agree }
\]

\[
R'; \emptyset \not\models \text{ hidden invariants of } (R \star R')
\]

\[
s \sim R
\]

With this definition, the type soundness statements are unchanged.
On top of this, a more elaborate notion of agreement is defined:

\[
s \text{ and } R \star R' \text{ agree }
\]

\[
R', \emptyset \mid \models \text{ hidden invariants of } (R \star R')
\]

\[
\therefore s \sim R
\]

With this definition, the type soundness statements are unchanged.
On top of this, a more elaborate notion of agreement is defined:

\[
\begin{align*}
&\text{the fragment of the instrumented state} \\
&\text{that remains visible to the program}
\end{align*}
\]

\[
\begin{align*}
s \text{ and } R \ast R' \text{ agree} \\
R'; \emptyset \not\models \text{hidden invariants of } (R \ast R') \\
\hline
s \sim R
\end{align*}
\]

With this definition, the type soundness statements are unchanged.
New typing rules for lock addresses:

\[
\begin{align*}
R(k) &= (P, _) \\
\frac{R; K; Q \vdash k : \text{lock } P}{R; K; Q \vdash k : \text{lock } P}
\end{align*}
\]

\[
\begin{align*}
R(k) &= (_, X) \\
\frac{R; K; Q \vdash k : \text{locked}}{R; K; Q \vdash k : \text{locked}}
\end{align*}
\]

*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:

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

Updated machine state component:

- the heap maps a memory location to a pair of an adopter pointer \( p := \text{null} \mid \ell \) and a value.
Syntax and dynamic semantics

New reduction rules:

<table>
<thead>
<tr>
<th>initial config.</th>
<th>new configuration</th>
<th>side condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$h / \text{give } \ell \text{ to } \ell'$</td>
<td>$h[\ell \mapsto \langle \ell'</td>
<td>v \rangle]$ / ()</td>
</tr>
<tr>
<td>$h / \text{take } \ell \text{ from } \ell'$</td>
<td>$h$ / take! $\ell$ from $\ell'$</td>
<td>$h(\ell) = \langle \ell'</td>
</tr>
<tr>
<td>$h / \text{take } \ell \text{ from } \ell'$</td>
<td>$h$ / fail</td>
<td>$h(\ell) = \langle p</td>
</tr>
<tr>
<td>$h / \text{take! } \ell \text{ from } \ell'$</td>
<td>$h[\ell \mapsto \langle \text{null}</td>
<td>v \rangle]$ / ()</td>
</tr>
<tr>
<td>$s / E[\text{fail}]$</td>
<td>$s$ / fail</td>
<td></td>
</tr>
</tbody>
</table>

Note that take does not need an atomic implementation.
New types:

\[ T ::= \ldots \mid \text{adoptable} \mid \text{unadopted} \mid \text{adopts } T \]

Intuitively,

- \( v @ \text{adoptable} \) is \( v @ \text{dynamic} \); it is duplicable;
  - guarantees the existence of \( v \)'s adopter field;
  - allows an attempt to take \( v \) from its adopter.

- \( v @ \text{unadopted} \) 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.

- \( v @ \text{adopts } T \) 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.
Modified typing rule for memory allocation:

\[
R; K; v @ T \vdash \text{newref } v : \exists x : \text{value.}(=x \mid
\quad (x @ \text{ref}_m T) \ast (x @ \text{adopts } \bot) \ast (x @ \text{unadopted}))
\]

The value \( x \) produced by \( \text{newref } v \):

- 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:

$R; K; (v_2 @ adopts U) \ast (v_1 \@ U) \ast (v_1 \@ unadopted)$

$\vdash give v_1 to v_2 : \top |$

$(v_2 @ adopts U)$

$R; K; (v_2 @ adopts U) \ast (v_1 \@ adoptable)$

$\vdash take v_1 from v_2 : \top |

(v_2 @ adopts U) \ast (v_1 \@ U) \ast (v_1 \@ unadopted)$
New subsumption rules:

\[
\text{empty} \leq \text{duplicable adoptable}
\]

\[
\nu \dpr \text{unadopted} \leq (\nu \dpr \text{unadopted}) \times (\nu \dpr \text{adoptable})
\]

\[
\frac{T \leq U}{\nu \dpr \text{adopts } T \leq \nu \dpr \text{adopts } U}
\]
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 $\leftarrow$, $N$, or $X$.
- An *adopter status* is $\leftarrow$, $N$, or $X$. 
Auxiliary definitions:

- $R \vdash \ell$ is adoptable iff $\ell$ is in the domain of $R$.
- $R \vdash \ell$ is unadopted iff $R$ maps $\ell$ to $(X \text{ null}, \_)$.
- $R \vdash \ell'$ is an adopter iff $R$ maps $\ell'$ to $(\_, X)$.
- $R \vdash \vec{\ell}$ are the adoptees of $\ell'$ iff:
  - $R \vdash \ell'$ is an adopter; and
  - $\vec{\ell}$ lists the addresses $\ell$ such that $R \vdash \ell$ is adopted by $\ell'$;
We would like "\( \vdash \vec{\ell} \text{ are the adoptees of } \ell' \)" to be affine, i.e.:

\[
\begin{align*}
R_1 & \vdash \vec{\ell} \text{ are the adoptees of } \ell' \\
R_1 \star R_2 & \vdash \vec{\ell} \text{ are the adoptees of } \ell'
\end{align*}
\]

But this does \textit{not} hold.

\( R_2 \) could own an adoptee of \( \ell' \).

There would be a \textit{dangling adopter edge} out of \( R_2 \).
We avoid this issue by forbidding dangling adopter edges. An adoption resource $R$ is *round* if

$$R \vdash \ell \text{ is adopted by } \ell' \text{ implies } R \vdash \ell' \text{ is an adopter.}$$

Roundness is preserved by $\star$ and by $\triangleleft$, which means we can work in the subset of round resources.
Three new typing rules for memory locations!

\[
\begin{align*}
R \vdash \ell & \text{ is adoptable} \\
R \vdash \ell & \text{ is unadopted} \\
R_1 & \vdash \vec{\ell} \text{ are the adoptees of } \ell' \\
R_2; K & \vdash \vec{\ell} \circled@ U \\
R_1 \star R_2; K & \vdash \ell' \text{ : adopts } U
\end{align*}
\]

They give meaning to the three new types.
New typing rules for terms:

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

\[ R; K \vdash \ell' @ \text{adopts } U \quad R \vdash \ell \text{ is adopted by } \ell' \]

\[ R; K; P \vdash \text{take! } \ell \text{ from } \ell' : T \mid (\ell' @ \text{adopts } U) * (\ell @ U) * (\ell @ \text{unadopted}) \]
Theorem

Well-typed programs do not go wrong.

“Just” a matter of dealing with the new proof cases.
Outline

- Introduction
- The kernel
- Extensions
- Conclusion
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).
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 $\lambda$-calculus.
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?
More information online:
http://gallium.inria.fr/~protzenk/mezzo-lang/