Safety of mutref + $\forall / \exists$, problem + intuitive solution

From ”Semantics of Types for Mutable State” by Amal Jamil Ahmed

Dmitrii Mikhailovskii, Marijn van Wezel

Introduction: syntax

Consider

$$\lambda^M := \lambda 2 + \text{mutref} + \exists + \texttt{unit},$$

or formally,

$$\begin{aligned} e &::= x \mid l \mid \texttt{unit} \mid \lambda x.e \mid e_1 e_2 \mid \texttt{new}(e) \mid !e \mid e_1 := e_2 \mid \\ &\quad\ \Lambda.e \mid e[] \mid \texttt{pack}\, e \mid \texttt{unpack}\, e_1\ \texttt{as}\ x\ \texttt{in}\ e_2, \\ v &::= l \mid \texttt{unit} \mid \lambda x.e \mid \Lambda.e \mid \texttt{pack}\, v, \\ x &\in Var, \\ l &\in Loc. \end{aligned}$$

Introduction: operational semantics

Given a store S,

- $dom(S)$ is a set of allocated locations.
- It can be extended with ***new***.

Then operational semantics is given by an abstract machine with states (S, e), where e is a closed term, and

$$(S, e) \mapsto_M (S', e')$$

is a single operational step.

Operational semantics: rules

$$\frac{(S, e_1) \mapsto_M (S', e_1')}{(S, e_1 e_2) \mapsto_M (S', e_1' e_2)} \text{(MO-app1)}$$

$$\frac{(S, e_2) \mapsto_M (S', e_2')}{(S, (\lambda x.e_1) e_2) \mapsto_M (S', (\lambda x.e_1) e_2')} \text{(MO-app2)}$$

$$\frac{}{(S, (\lambda x.e) v) \mapsto_M (S', e[v/x])} \text{(MO-app3)}$$

$$\frac{(S, e) \mapsto_M (S', e')}{(S, \texttt{new}\ e) \mapsto_M (S', \texttt{new}\ e')} \text{(MO-new1)}$$

$$\frac{l \notin dom(S)}{(S, \texttt{new}\ v) \mapsto_M (S[l \mapsto v], l)} \text{(MO-new2)}$$

Operational semantics: rules

$$\frac{(S, e) \mapsto_M (S', e')}{(S, !e) \mapsto_M (S', !e')} \text{(MO-deref1)}$$

$$\frac{l \in dom(S)}{(S, !l) \mapsto_M (S, S(l))} \text{(MO-deref2)}$$

$$\frac{(S, e_1) \mapsto_M (S', e_1')}{(S, e_1 := e_2) \mapsto_M (S', e_1' := e_2)} \text{(MO-assign1)}$$

$$\frac{(S, e_2) \mapsto_M (S', e_2')}{(S, v_1 := e_2) \mapsto_M (S', v_1 := e_2')} \text{(MO-assign2)}$$

$$\frac{l \in dom(S)}{(S, l := v) \mapsto_M (S[l \mapsto v], \texttt{unit})} \text{(MO-assign3)}$$

Operational semantics: rules

$$\frac{(S, e) \mapsto_M (S', e')}{(S, e[]) \mapsto_M (S', e'[])} \text{ (MO-tapp1)}$$

$$\frac{}{(S, (\Lambda.e)[]) \mapsto_M (S, e)} \text{ (MO-tapp2)}$$

$$\frac{(S, e) \mapsto_M (S', e')}{(S, \mathtt{pack}\ e) \mapsto_M (S', \mathtt{pack}\ e')} \text{ (MO-pack)}$$

$$\frac{(S, e_1) \mapsto_M (S', e_1')}{(S, \mathtt{unpack}\ e_1\ \mathtt{as}\ x\ \mathtt{in}\ e_2) \mapsto_M (S', \mathtt{unpack}\ e_1'\ \mathtt{as}\ x\ \mathtt{in}\ e_2)} \text{ (MO-unpack1)}$$

$$\frac{}{(S, \mathtt{unpack}\ (\mathtt{pack}\ v)\ \mathtt{as}\ x\ \mathtt{in}\ e_2) \mapsto_M (S, e_2[v/x])} \text{ (MO-unpack1)}$$

Problem

The model for λ^l restricts updates of allocated cells. Consider the following program:

```
let ... in                          % store is S
let _ = x := y in                   % store is S'
e_rest
```

with the following derivation:

$$\Gamma = \{x \mapsto \texttt{ref}\ \tau_1,\ y \mapsto \tau_1,\ z \mapsto \tau_2\},$$

$$\Gamma \vDash_M \texttt{let}\ \_ = x := y\ \texttt{in}\ e_{rest} : \tau'$$

$$\Gamma \vDash_M e_{rest} : \tau'$$

Problem

According to λ^l, we have that

$$x :_S \texttt{ref}\ \tau_1,\ y :_S \tau_1,\ z :_S \tau_2$$

and

$$x :_{S'} \texttt{ref}\ \tau_1,\ y :_{S'} \tau_1,\ z :_{S'} \tau_2.$$

Hence, we need to be able to prove that types remain the same after evaluation of

$$x := y.$$

However, we cannot guarantee that $z : \tau_2$, as it may be an aliased location of x.

Problem

Hence, we need to be able to prove that types remain the same after evaluation of

$$x := y.$$

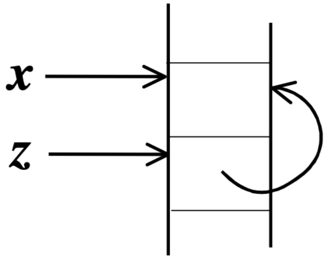

Figure 1: Visualization of state S

Store typing

To allow only weak (type-preserving) updates to mutable references, we introduce a *store typing* Ψ, which keeps tracks of the type τ of updates allowed at a location $\ell \in Loc$.

We say that

$$\Psi(\ell) = \tau$$

if ℓ is an allocated location and updates of type τ are allowed at ℓ.

Store typing

We say that ℓ has type **ref** τ if and only if:

- ℓ is an allocated location, and,
- the value in the store S at location ℓ currently has type τ, and,
- the *store typing* says that the allowed update type for location ℓ is τ.

Store typing

We can formalize the type of mutable references **ref** τ as:

$$\textbf{ref}\ \tau \stackrel{\text{def}}{=} \{\ \langle S, \Psi, \ell \rangle \mid \ell \in \text{dom}(\Psi) \wedge \Psi(\ell) = \tau \wedge \langle S, \Psi, S(\ell) \rangle \in \tau\ \}$$

Since $\Psi(\ell) = \tau$ implies that $\ell \in \text{dom}(\Psi)$ and we implicitly assume that S satisfies Ψ, we can simplify this type to:

$$\textbf{ref}\ \tau \stackrel{\text{def}}{=} \{\ \langle \Psi, \ell \rangle \mid \Psi(\ell) = \tau\ \}$$

We can see that the type of a value depends on Ψ and the value itself (n.b. a location ℓ is also a value).

An inconsistent model

Recall that the type of a value depends on the store typing Ψ and the value itself. The types of these objects are:

$$\begin{aligned} StoreType &= Loc \xrightarrow{\text{fin}} Type \\ Type &= StoreType \times Val \to \mathcal{P} \end{aligned}$$

where $\mathcal{P}$ is the type of propositions (*true* or *false*).

The definition of $Type$ is *circular* and has an inconsistent cardinality: the set of types must be bigger than itself.

Stratification

Consider our latest definition of **ref**:

$$\textsf{ref}\ \tau \overset{\text{def}}{=} \{\ \langle \Psi, \ell \rangle \mid \Psi(\ell) = \tau\ \}$$

Notice that to determine the members of **ref** τ, we only need to look at $\Psi(\ell) = \tau$. This suggests there exists an ordering, which can be used to *stratify* our types.

Stratification

To stratify our types, we:

- Divide our types into levels 0 through ∞, and,
- Let a type at level *k* rely only the store typing that maps to types at level *j*, where $j < k$.

$$\begin{aligned} Type_0 &= Unit \\ StoreType_k &= Loc \xrightarrow{\text{fin}} Type_k \\ Type_{k+1} &= StoreType_k \times Val \to \mathcal{P} \end{aligned}$$

Syntax-based stratification

To determine whether a type τ belongs to $Type_k$ for $k \geq 0$, we use the following set of rules:

- $\tau = \bot \Rightarrow \tau \in Type_0$
- τ is a primitive type (`unit` or `bool`) $\Rightarrow \tau \in Type_1$
- $\tau \in Type_k \Rightarrow$ **ref** $\tau \in Type_{k+1}$
- $\tau \in Type_k \Rightarrow \tau \in Type_{k+1}$

Syntax-based stratification

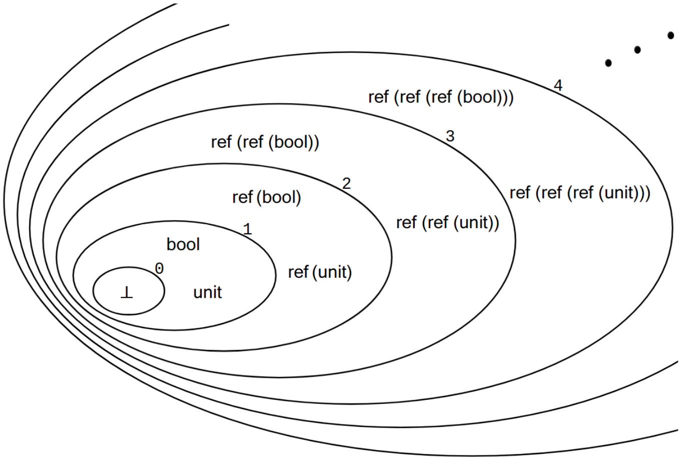

Figure 2: Type hierarchy based on syntax of types

Problems with syntax-based stratification

Consider $\exists \alpha.$ **ref** α: if we assume that $\alpha \in Type_k$, we can conclude that $\exists \alpha.$ **ref** $\alpha \in Type_{k+1}$ (the $+1$ comes from the extra **ref**).

However, since α is a type variable, we don't know the level of the type that instantiates α.

Problems with syntax-based stratification

Furthermore, we want to model impredicative quantified types as well. This means that α, which we assumed to be at level k, may be instantiated with $\exists\alpha.\ \textbf{ref}\ \alpha$ itself, which we concluded was at level $k + 1$.

Since $k \neq k + 1$, this leads to a contradiction: there is no finite level that is guaranteed to contain $\exists\alpha.\ \textbf{ref}\ \alpha$.

Problems with syntax-based stratification

Suppose we *do know* of a finite level *n* that contains $\exists\alpha.\ \textsf{ref}\ \alpha$, then we another problem:

- In the existential case ($\exists$), knowing the witness type is contained in level *n* breaks the abstraction;
- In the universal case ($\forall$), knowing that level *n* must contain the type gives a form of bounded (not impredicative) polymorphism.

Stratification based on semantic approximation

Instead of using the *syntactic complexity* of a type to stratify on, we shall use semantic approximation for our stratification.

- $\langle \Psi^{k-1}, v \rangle \in \tau^k$ means that v *looks* like it belong to τ for at least k steps;
- We call this k the approximation index;
- Levels correspond to approximations of a type's behaviour;
- To determine the members of **ref** τ to approximation k, we only need to look at a store typing that maps types to approximation $k - 1$.

Stratification based on semantic approximation

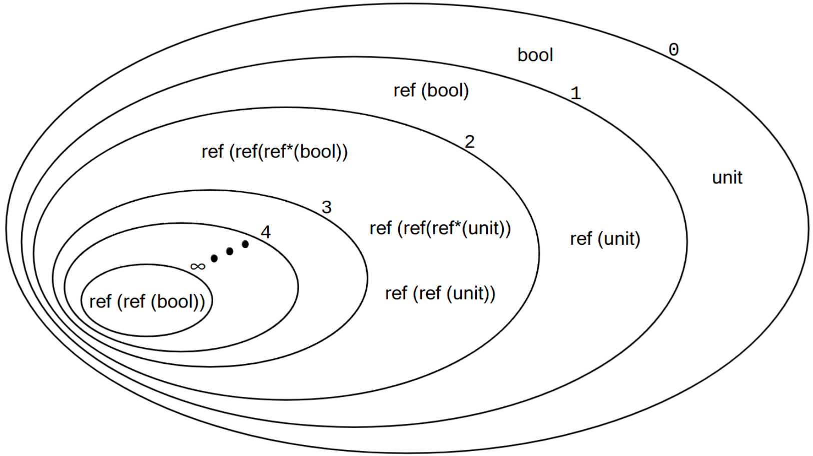

Figure 3: Approximations of type ref(ref($bool$))

Stratification based on semantic approximation

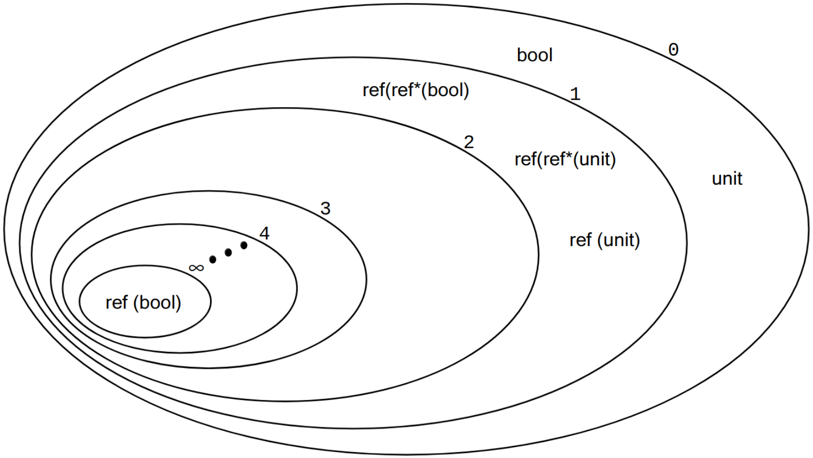

Figure 4: Approximations of type ref($bool$)

Stratification based on semantic approximation

Our original observation that led us to this type hierarchy was that τ is smaller than **ref** τ. With semantic approximation, ”smaller” means that less steps are necessary to show that an assumption about a type is wrong.

It is still true that τ is smaller than **ref** τ, since the latter requires an additional dereferencing step.

Stratification based on semantic approximation

Unlike the previous approach, semantic approximation *is* able to model (impredicative) quantified types. Suppose we need to assign a level to:

$$\exists\alpha.\ \textsf{ref}\ \alpha$$

Since the level of our types is now bound to the operational semantics of the language (i.e. the number of steps), we can just assign it the number of remaining steps as the level.

Conclusion

In conclusion, to model weak updates to mutable references, we:

- Introduced the concept of a *store typing*, which lead to an inconsistent model;
- Tried using syntax-based stratification to solve this, but this did not work for quantified types;
- Introduced the concept of an *approximation index* to stratify our types.