Type safety of Recursive types

Jeroen Kool & Cárolos Laméris

Recursive data structures

Example Coq:

```
Inductive bintree : Set :=
  | leaf : nat -> bintree
  | node : bintree -> bintree -> bintree.
```

Recursive data structures

Example Coq:

```
Inductive bintree : Set :=
  | leaf : nat -> bintree
  | node : bintree -> bintree -> bintree.
```

Naive implementation Rust:

```
struct Node
{
    value: u32,                 // unsigned 32-bit integer
    left: Option<Node>,
    right: Option<Node>
}
```

Recursive data structures

Compiler complains:

```
| struct Node
| ^^^^^^^^^^ recursive type has infinite size

| left: Option<Node>,
|       ------------ recursive without indirection
| right: Option<Node>
|       ------------ recursive without indirection
|
help: insert some indirection (e.g., a `Box`, `Rc`,
      or `&`) to make `Node` representable
```

Recursive data structures

Following the compilers advice:

```
struct Node
{
    value: u32,
    left: Option<Box<Node>>,
    right: Option<Box<Node>>
}
```

Now the Compiler doesn't complain.[1]

[1] For a detailed discussion see 15.1 of the Rust documentation.

Recursive data structures

Untyped λ-calculus can capture recursiveness:

$$Y \coloneqq \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$$

Recursive data structures

Untyped λ-calculus can capture recursiveness:

$$Y := \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$$

$$(YM) \twoheadrightarrow_\beta M(YM) \twoheadrightarrow \ldots$$

Recursive data structures

Untyped λ-calculus can capture recursiveness:

$$Y \coloneqq \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$$

$$(YM) \twoheadrightarrow_\beta M(YM) \twoheadrightarrow \ldots$$

$$\Omega \coloneqq (\lambda x.xx)(\lambda x.xx)$$

Recursive data structures

Untyped λ-calculus can capture recursiveness:

$$Y \coloneqq \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$$

$$(YM) \twoheadrightarrow_\beta M(YM) \twoheadrightarrow \ldots$$

$$\Omega \coloneqq (\lambda x.xx)(\lambda x.xx)$$

$$\Omega \to_\beta \Omega \to_\beta \ldots$$

Fixed-point type constructor

We will look at the tree example. We call the leaf, the unit, and the tree, α.

$$\begin{aligned}\alpha &= 1 + (int \times \alpha \times \alpha) \\ &= 1 + (int \times (int \times \alpha \times \alpha) \times (int \times \alpha \times \alpha)) \\ &\vdots\end{aligned}$$

Fixed-point type constructor

We will look at the tree example. We call the leaf, the unit, and the tree, α.

$$\begin{aligned}\alpha &= 1 + (int \times \alpha \times \alpha) \\ &= 1 + (int \times (int \times \alpha \times \alpha) \times (int \times \alpha \times \alpha)) \\ &\vdots\end{aligned}$$

We can define a recursive function:

$$F = \lambda\alpha : type.1 + (int \times \alpha \times \alpha).$$

Fixed-point type constructor

$$F = \lambda\alpha : type.1 + (int \times \alpha \times \alpha)$$

We want to find the fixed point, which, by definition, is t such that

$$t = F(t).$$

Fixed-point type constructor

$$F = \lambda\alpha : type.1 + (int \times \alpha \times \alpha)$$

We want to find the fixed point, which, by definition, is t such that

$$t = F(t).$$

The fixed point of this function is written as

$$\mu\alpha.F(\alpha)$$

Here μ is a fixed-point type constructor.

Fixed-point type constructor

$$F = \lambda\alpha : type.1 + (int \times \alpha \times \alpha)$$

We want to find the fixed point, which, by definition, is t such that

$$t = F(t).$$

The fixed point of this function is written as

$$\mu\alpha.F(\alpha)$$

Here μ is a fixed-point type constructor. As the above is the fixed point by definition, it should be equal to F applied to itself:

$$\mu\alpha.F(\alpha) = F\left(\mu\alpha.F(\alpha)\right)$$

Fixed-point type constructor

We now have:

$$\mu\alpha.F(\alpha) = F\left(\mu\alpha.F(\alpha)\right)$$

To let us make this look a bit more like types by substituting $F(\alpha) = \tau$.

$$\mu\alpha.\tau = F(\mu\alpha.\tau)$$

Fixed-point type constructor

We now have:

$$\mu\alpha.F(\alpha) = F\left(\mu\alpha.F(\alpha)\right)$$

To let us make this look a bit more like types by substituting $F(\alpha) = \tau$.

$$\mu\alpha.\tau = F(\mu\alpha.\tau)$$

We rewrite the right hand side of the above equation:

$$F(\mu\alpha.\tau) = 1 + (int \times \mu\alpha.\tau \times \mu\alpha.\tau) = \tau[\mu\alpha.\tau/\alpha].$$

Fixed-point type constructor

We now have:

$$\mu\alpha.F(\alpha) = F\left(\mu\alpha.F(\alpha)\right)$$

To let us make this look a bit more like types by substituting $F(\alpha) = \tau$.

$$\mu\alpha.\tau = F(\mu\alpha.\tau)$$

We rewrite the right hand side of the above equation:

$$F(\mu\alpha.\tau) = 1 + (int \times \mu\alpha.\tau \times \mu\alpha.\tau) = \tau[\mu\alpha.\tau/\alpha].$$

We get

$$\mu\alpha.\tau = \tau[\mu\alpha.\tau/\alpha]$$

`fold` and `unfold`

We introduce the recursive type $\mu\alpha.\tau$ to our language. We can shift to an expanded version $\tau[\mu\alpha.\tau/\alpha]$.

`fold` and `unfold`

We introduce the recursive type $\mu\alpha.\tau$ to our language. We can shift to an expanded version $\tau[\mu\alpha.\tau/\alpha]$.

Expanding is called `unfold` and contracting `fold`

$$\mu\alpha.\tau \xrightarrow{\texttt{unfold}} \tau[\mu\alpha.\tau/\alpha]$$
$$\&$$
$$\tau[\mu\alpha.\tau/\alpha] \xrightarrow{\texttt{fold}} \mu\alpha.\tau$$

`fold` and `unfold`

We introduce the recursive type $\mu\alpha.\tau$ to our language. We can shift to an expanded version $\tau[\mu\alpha.\tau/\alpha]$.

Expanding is called `unfold` and contracting `fold`

$$\mu\alpha.\tau \xrightarrow{\texttt{unfold}} \tau[\mu\alpha.\tau/\alpha]$$
$$\&$$
$$\tau[\mu\alpha.\tau/\alpha] \xrightarrow{\texttt{fold}} \mu\alpha.\tau$$

For example:

$$tree = \mu\alpha.1 + (int \times \alpha \times \alpha)$$
$$\texttt{fold} \uparrow \quad \downarrow \texttt{unfold}$$
$$1 + (int \times (\mu\alpha.1 + (int \times \alpha \times \alpha)) \times (\mu\alpha.1 + (int \times \alpha \times \alpha)))$$

`fold` and `unfold`

The syntax of simple typed lambda calculus (STLC) with recursive types is defined as follows:

$$
\begin{aligned}
\tau &::= \ldots \mid \mu\alpha.\tau \\
e &::= \ldots \mid \texttt{fold}\ e \mid \texttt{unfold}\ e \\
v &::= \ldots \mid \texttt{fold}\ v \\
E &::= \ldots \mid \texttt{fold}\ E \mid \texttt{unfold}\ E
\end{aligned}
$$

`fold` and `unfold`

Further, the following **evaluation rule** is added:

$$\frac{}{\texttt{unfold}\ (\texttt{fold}\ v) \to v}$$

`fold` and `unfold`

Further, the following **evaluation rule** is added:

$$\frac{}{\texttt{unfold}\ (\texttt{fold}\ v) \to v}$$

Finally, the following **typing judgements** are added:

$$\frac{\Gamma \vdash e : \tau[\mu\alpha.\tau/\alpha]}{\Gamma \vdash \texttt{fold}\ e : \mu\alpha.\tau}\textsc{Fold} \qquad \frac{\Gamma \vdash e : \mu\alpha.\tau}{\Gamma \vdash \texttt{unfold}\ e : \tau[\mu\alpha.\tau/\alpha]}\textsc{Unfold}$$

`fold` and `unfold`

$$T \coloneqq \mu\alpha.\alpha \to \alpha,\ \omega = \lambda x : T.(\texttt{unfold}\ x)x,\ \Omega = \omega(\texttt{fold}\ \omega)$$

`fold` and `unfold`

$$T := \mu\alpha.\alpha \to \alpha,\ \omega = \lambda x : T.(\texttt{unfold}\ x)x,\ \Omega = \omega(\texttt{fold}\ \omega)$$

$$\Omega = \omega(\texttt{fold}\ \omega) = [\lambda x : T.(\texttt{unfold}\ x)x](\texttt{fold}\ \omega)$$

$$\to [\texttt{unfold}\ (\texttt{fold}\ \omega)](\texttt{fold}\ \omega) \to \omega(\texttt{fold}\ \omega) = \Omega$$

`fold` and `unfold`

$$T := \mu\alpha.\alpha \to \alpha,\ \omega = \lambda x : T.(\texttt{unfold}\ x)x,\ \Omega = \omega(\texttt{fold}\ \omega)$$

$$\frac{\Gamma \vdash e : \tau[\mu\alpha.\tau/\alpha]}{\Gamma \vdash \texttt{fold}\ e : \mu\alpha.\tau}\ \textsc{Fold} \qquad \frac{\Gamma \vdash e : \mu\alpha.\tau}{\Gamma \vdash \texttt{unfold}\ e : \tau[\mu\alpha.\tau/\alpha]}\ \textsc{Unfold}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{x : T \vdash x : T}\ \text{Var}}{x : T \vdash \texttt{unfold}\ x : T \to T}\ \text{Unfold} \qquad \dfrac{}{x : T \vdash x : T}\ \text{Var}}{x : T \vdash (\texttt{unfold}\ x)x : T}\ \text{App}}{\bullet \vdash \omega : T \to T}\ \text{Abs}$$

`fold` and `unfold`

$$T := \mu\alpha.\alpha \to \alpha,\ \omega = \lambda x : T.(\texttt{unfold}\ x)x,\ \Omega = \omega(\texttt{fold}\ \omega)$$

$$\frac{\Gamma \vdash e : \tau[\mu\alpha.\tau/\alpha]}{\Gamma \vdash \texttt{fold}\ e : \mu\alpha.\tau}\ \textsc{Fold} \qquad \frac{\Gamma \vdash e : \mu\alpha.\tau}{\Gamma \vdash \texttt{unfold}\ e : \tau[\mu\alpha.\tau/\alpha]}\ \textsc{Unfold}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{x : T \vdash x : T}\ \text{Var}}{x : T \vdash \texttt{unfold}\ x : T \to T}\ \text{Unfold} \qquad \dfrac{}{x : T \vdash x : T}\ \text{Var}}{x : T \vdash (\texttt{unfold}\ x)x : T}\ \text{App}}{\bullet \vdash \omega : T \to T}\ \text{Abs}$$

$$\dfrac{\begin{array}{c}\vdots \\ \bullet \vdash \omega : T \to T\end{array} \qquad \dfrac{\begin{array}{c}\vdots \\ \bullet \vdash \omega : T \to T\end{array}}{\bullet \vdash \texttt{fold}\ \omega : T}\ \text{Fold}}{\bullet \vdash \omega(\texttt{fold}\ \omega) : T}\ \text{App}$$

`fold` and `unfold`

$$T \coloneqq \mu\alpha.\alpha \to X,\ \zeta \coloneqq \lambda x : T.f[(\texttt{unfold}\ \ x)x]$$

`fold` and `unfold`

$$T \coloneqq \mu\alpha.\alpha \to X,\ \zeta \coloneqq \lambda x : T.f[(\texttt{unfold}\ x)x]$$

- $\bullet \vdash Y \coloneqq \lambda f : X \to X.\zeta(\texttt{fold}\ \zeta) : (X \to X) \to X$

`fold` and `unfold`

$$T \coloneqq \mu\alpha.\alpha \to X,\ \zeta \coloneqq \lambda x : T.f[(\texttt{unfold}\ x)x]$$

$$\bullet \vdash Y \coloneqq \lambda f : X \to X.\zeta(\texttt{fold}\ \zeta) : (X \to X) \to X$$

$$M : X \to X, \quad (Y\,M) \to M(Y\,M)$$

Reviewing interpretations

Recall:

$$\mathcal{V}[\![\sigma]\!] \coloneqq \begin{cases} \{\texttt{true}, \texttt{false}\} & \text{if } \sigma = \texttt{bool}, \\ \{\lambda x : \tau_1.e;\ \forall v \in \mathcal{V}[\![\tau_1]\!].\, e[v/x] \in \mathcal{E}[\![\tau_2]\!]\} & \text{if } \sigma = \tau_1 \to \tau_2, \\ \{\text{inl}\, v;\ \mathcal{V}[\![\tau_1]\!]\} \cup \{\text{inr}\, v;\ \mathcal{V}[\![\tau_2]\!]\} & \text{if } \sigma = \tau_1 + \tau_2. \end{cases}$$

$$\mathcal{E}[\![\tau]\!] \coloneqq \{e;\ \forall e'.\, e \to^* e' \wedge \text{irred}(e) \implies e' \in \mathcal{V}[\![\tau]\!]\}.$$

Reviewing interpretations

$$\mathcal{V}[\![\mu\alpha.\tau]\!] \coloneqq \left\{ \texttt{fold}\, v;\, \texttt{unfold}\,(\texttt{fold}\, v) \in \mathcal{E}\left[\!\!\left[\tau\left[\frac{\mu\alpha.\tau}{\alpha}\right]\right]\!\!\right] \right\}$$

Reviewing interpretations

$$\mathcal{V}[\![\mu\alpha.\tau]\!] \coloneqq \left\{\texttt{fold}\, v;\, \texttt{unfold}\,(\texttt{fold}\, v) \in \mathcal{E}\left[\!\!\left[\tau\left[\frac{\mu\alpha.\tau}{\alpha}\right]\right]\!\!\right]\right\}$$

$$= \left\{\texttt{fold}\, v;\, v \in \mathcal{V}\left[\!\!\left[\tau\left[\frac{\mu\alpha.\tau}{\alpha}\right]\right]\!\!\right]\right\}$$

Reviewing interpretations

$$\mathcal{V}[\![\mu\alpha.\tau]\!] \coloneqq \left\{\texttt{fold}\, v;\, \texttt{unfold}\,(\texttt{fold}\, v) \in \mathcal{E}\left[\!\!\left[\tau\left[\frac{\mu\alpha.\tau}{\alpha}\right]\right]\!\!\right]\right\}$$

$$= \left\{\texttt{fold}\, v;\, v \in \mathcal{V}\left[\!\!\left[\tau\left[\frac{\mu\alpha.\tau}{\alpha}\right]\right]\!\!\right]\right\}$$

$$\mathcal{V}[\![\mu\alpha.\alpha \to \alpha]\!] = \{\texttt{fold}\, v;\, v \in \mathcal{V}[\![(\mu\alpha.\alpha \to \alpha) \to (\mu\alpha.\alpha \to \alpha)]\!]\}$$

Reviewing interpretations

$$\mathcal{V}[\![\mu\alpha.\tau]\!] := \left\{\texttt{fold}\, v;\, \texttt{unfold}\,(\texttt{fold}\, v) \in \mathcal{E}\left[\!\!\left[\tau\left[\frac{\mu\alpha.\tau}{\alpha}\right]\right]\!\!\right]\right\}$$

$$= \left\{\texttt{fold}\, v;\, v \in \mathcal{V}\left[\!\!\left[\tau\left[\frac{\mu\alpha.\tau}{\alpha}\right]\right]\!\!\right]\right\}$$

$$\mathcal{V}[\![\mu\alpha.\alpha \to \alpha]\!] = \{\texttt{fold}\, v;\, v \in \mathcal{V}[\![(\mu\alpha.\alpha \to \alpha) \to (\mu\alpha.\alpha \to \alpha)]\!]\}$$

$$\mathcal{V}[\![(\mu\alpha.\alpha \to \alpha) \to (\mu\alpha.\alpha \to \alpha)]\!] =$$

$$\left\{\lambda x : \mu\alpha.\alpha \to \alpha.e;\, \forall v \in \mathcal{V}[\![\mu\alpha.\alpha \to \alpha]\!],\, e[v/x] \in \mathcal{E}[\![\mu\alpha.\alpha \to \alpha]\!]\right\}$$

Reviewing interpretations

$$\mathcal{E}_k[\![\tau]\!] \coloneqq \{e;\, \forall j < k.\, \forall e'.\, e \to^j e' \wedge \mathrm{irred}(e') \implies e' \in \mathcal{V}_{k-j}[\![\tau]\!]\}$$

Reviewing interpretations

$$\mathcal{E}_k[\![\tau]\!] \coloneqq \{e;\, \forall j < k.\, \forall e'.\, e \to^j e' \wedge \mathrm{irred}(e') \implies e' \in \mathcal{V}_{k-j}[\![\tau]\!]\}$$

$$\mathcal{V}_k[\![\mu\alpha.\tau]\!] \coloneqq \left\{\mathtt{fold}\, v;\, \forall j < k.\, v \in \mathcal{V}_j \left[\!\!\left[\tau \left[\frac{\mu\alpha.\tau}{\alpha}\right]\right]\!\!\right]\right\}$$

Reviewing interpretations

$$\mathcal{V}_k[\![\sigma]\!] := \begin{cases} \{\texttt{true}, \texttt{false}\} & \text{if } \sigma = \texttt{bool}, \\ \{\lambda x : \tau_1.e;\ \forall j \leq k.\, \forall v \in \mathcal{V}_j[\![\tau_1]\!].\, e[v/x] \in \mathcal{E}_j[\![\tau_2]\!]\} & \text{if } \sigma = \tau_1 \to \tau_2, \\ \{\mathrm{inl}\, v;\ \mathcal{V}_k[\![\tau_1]\!]\} \cup \{\mathrm{inr}\, v;\ \mathcal{V}_k[\![\tau_2]\!]\} & \text{if } \sigma = \tau_1 + \tau_2, \end{cases}$$

Reviewing interpretations

Example: Consider the term:

$$M \coloneqq \lambda x : \texttt{bool}.x(\lambda y : \texttt{bool}.y)$$

Reviewing interpretations

Example: Consider the term:

$$M \coloneqq \lambda x : \texttt{bool}.x(\lambda y : \texttt{bool}.y)$$

$$M \in \mathcal{V}_0[\![\texttt{b} \to \texttt{b}]\!] = \{\lambda x : \texttt{b}.e;\ \forall v \in \mathcal{V}_0[\![\texttt{b}]\!].\ e[v/x] \in \mathcal{E}_0[\![\texttt{b}]\!]\}$$

Reviewing interpretations

Example: Consider the term:

$$M \coloneqq \lambda x : \texttt{bool}.x(\lambda y : \texttt{bool}.y)$$

$$M \in \mathcal{V}_0[\![\texttt{b} \to \texttt{b}]\!] = \{\lambda x : \texttt{b}.e;\ \forall v \in \mathcal{V}_0[\![\texttt{b}]\!].\ e[v/x] \in \mathcal{E}_0[\![\texttt{b}]\!]\}$$

$$\mathcal{E}_0[\![\texttt{b}]\!] = \{e;\ \forall j < 0.\forall e'.\ e \to^j e' \wedge \text{irred}(e') \implies e' \in \mathcal{V}_{0-j}[\![\tau]\!]\}.$$

Reviewing interpretations

Example: Consider the term:

$$M \coloneqq \lambda x : \texttt{bool}.x(\lambda y : \texttt{bool}.y)$$

$$M \in \mathcal{V}_0[\![\texttt{b} \to \texttt{b}]\!] = \{\lambda x : \texttt{b}.e;\ \forall v \in \mathcal{V}_0[\![\texttt{b}]\!].\ e[v/x] \in \mathcal{E}_0[\![\texttt{b}]\!]\}$$

$$\mathcal{E}_0[\![\texttt{b}]\!] = \{e;\ \forall j < 0.\forall e'.\ e \to^j e' \wedge \text{irred}(e') \implies e' \in \mathcal{V}_{0-j}[\![\tau]\!]\}.$$

$$M \notin \mathcal{V}_1[\![\texttt{b} \to \texttt{b}]\!] = \{\lambda x : \texttt{b}.e;\ \forall j \leq 1.\forall v \in \mathcal{V}_j[\![\texttt{b}]\!].\ e[v/x] \in \mathcal{E}_j[\![\texttt{b}]\!]\}$$

Reviewing interpretations

Example: Consider the term:

$$M \coloneqq \lambda x : \texttt{bool}.x(\lambda y : \texttt{bool}.y)$$

$$M \in \mathcal{V}_0[\![\texttt{b} \to \texttt{b}]\!] = \{\lambda x : \texttt{b}.e;\ \forall v \in \mathcal{V}_0[\![\texttt{b}]\!].\ e[v/x] \in \mathcal{E}_0[\![\texttt{b}]\!]\}$$

$$\mathcal{E}_0[\![\texttt{b}]\!] = \{e;\ \forall j < 0.\forall e'.\ e \to^j e' \wedge \text{irred}(e') \implies e' \in \mathcal{V}_{0-j}[\![\tau]\!]\}.$$

$$M \notin \mathcal{V}_1[\![\texttt{b} \to \texttt{b}]\!] = \{\lambda x : \texttt{b}.e;\ \forall j \leq 1.\forall v \in \mathcal{V}_j[\![\texttt{b}]\!].\ e[v/x] \in \mathcal{E}_j[\![\texttt{b}]\!]\}$$

$$\mathcal{E}_1[\![\texttt{b}]\!] \coloneqq \{e;\ \forall j < 1.\forall e'.\ e \to^j e' \wedge \text{irred}(e') \implies e' \in \mathcal{V}_{1-j}[\![\texttt{b}]\!]\}$$

Type safety

Definition.

$$\mathcal{G}_k[\![\bullet]\!] = \emptyset$$

$$\mathcal{G}_k[\![\Gamma, x : \tau]\!] = \{\gamma[x \mapsto v] \mid \Gamma \in \mathcal{G}_k[\![\Gamma]\!] \wedge v \in \mathcal{V}_k[\![\tau]\!]\}$$

$$\Gamma \models e : \tau := \forall k \geq 0. \forall \gamma \in \mathcal{G}_k[\![\Gamma]\!]. \gamma(e) \in \mathcal{E}_k[\![\tau]\!]$$

Type safety

Definition.

$$\mathcal{G}_k[\![\bullet]\!] = \emptyset$$

$$\mathcal{G}_k[\![\Gamma, x : \tau]\!] = \{\gamma[x \mapsto v] \mid \Gamma \in \mathcal{G}_k[\![\Gamma]\!] \wedge v \in \mathcal{V}_k[\![\tau]\!]\}$$

$$\Gamma \models e : \tau := \forall k \geq 0. \forall \gamma \in \mathcal{G}_k[\![\Gamma]\!]. \gamma(e) \in \mathcal{E}_k[\![\tau]\!]$$

For type safety we need:

- Fundamental property
- Entails type safety, i.e. $\bullet \models e : \tau \Longrightarrow \text{safe}(e)$

Type safety

Definition.

$$\mathcal{G}_k[\![\bullet]\!] = \emptyset$$

$$\mathcal{G}_k[\![\Gamma, x : \tau]\!] = \{\gamma[x \mapsto v] \mid \Gamma \in \mathcal{G}_k[\![\Gamma]\!] \wedge v \in \mathcal{V}_k[\![\tau]\!]\}$$

$$\Gamma \models e : \tau := \forall k \geq 0. \forall \gamma \in \mathcal{G}_k[\![\Gamma]\!]. \gamma(e) \in \mathcal{E}_k[\![\tau]\!]$$

For type safety we need:

- Fundamental property
- Entails type safety, i.e. $\bullet \models e : \tau \Longrightarrow \text{safe}(e)$

That this entails type safety is proved as in Chapter 3 (last presentation).

Fundamental property

For the fundamental property we need monotonicity.

Lemma (Monotonicity).

If $v \in \mathcal{V}_k[\![\tau]\!]$ and $j \leq k$, then $v \in \mathcal{V}_j[\![\tau]\!]$.

Fundamental property

For the fundamental property we need monotonicity.

Lemma (Monotonicity).

If $v \in \mathcal{V}_k[\![\tau]\!]$ and $j \leq k$, then $v \in \mathcal{V}_j[\![\tau]\!]$.

This can be proven by the cases

- $\tau = bool$
- $\tau = \tau_1 \to \tau_2$
- $\tau = \mu\alpha.\tau$

Fundamental property

For the fundamental property we need monotonicity.

Lemma (Monotonicity).

If $v \in \mathcal{V}_k[\![\tau]\!]$ and $j \leq k$, then $v \in \mathcal{V}_j[\![\tau]\!]$.

This can be proven by the cases

- $\tau = bool$
- $\tau = \tau_1 \to \tau_2$
- $\tau = \mu\alpha.\tau$

We will prove the case $\tau = \mu\alpha.\tau$

Proof monotonicity lemma

Assume $v \in \mathcal{V}_k[\![\mu\alpha.\tau]\!]$ and $j \leq k$, we then need to show $v \in \mathcal{V}_j[\![\mu\alpha.\tau]\!]$.

Proof monotonicity lemma

Assume $v \in \mathcal{V}_k[\![\mu\alpha.\tau]\!]$ and $j \leq k$, we then need to show $v \in \mathcal{V}_j[\![\mu\alpha.\tau]\!]$.

Because $v \in \mathcal{V}_k[\![\mu\alpha.\tau]\!]$, we know, by the definition of $\mathcal{V}_k[\![\mu\alpha.\tau]\!]$, that there exists a v' such that $v = \texttt{fold}\ v'$, with $\forall n < k.v' \in \mathcal{V}_n[\![\tau[\mu\alpha.\tau/\alpha]]\!]$.

Proof monotonicity lemma

Assume $v \in \mathcal{V}_k[\![\mu\alpha.\tau]\!]$ and $j \leq k$, we then need to show $v \in \mathcal{V}_j[\![\mu\alpha.\tau]\!]$.

Because $v \in \mathcal{V}_k[\![\mu\alpha.\tau]\!]$, we know, by the definition of $\mathcal{V}_k[\![\mu\alpha.\tau]\!]$, that there exists a v' such that $v = \texttt{fold}\ v'$, with $\forall n < k.v' \in \mathcal{V}_n[\![\tau[\mu\alpha.\tau/\alpha]]\!]$.

Because $j \leq k$ we also know that $\forall n < j.v' \in \mathcal{V}_n[\![\tau[\mu\alpha.\tau/\alpha]]\!]$ and so $v \in \mathcal{V}_j[\![\mu\alpha.\tau]\!]$.

Fundamental property

Theorem (Fundamental property).

If $\Gamma \vdash e : \tau$, then $\Gamma \models e : \tau$.

Fundamental property

Theorem (Fundamental property).

If $\Gamma \vdash e : \tau$, then $\Gamma \models e : \tau$.

Proof by induction on the typing derivations.

We will give the prove overview for the Fold case.

Proof of fundamental property of Fold case

For the Fold case, we assume $\Gamma \vdash \texttt{fold}\ e : \mu\alpha.\tau$ and need to show that $\Gamma \models \texttt{fold}\ e : \mu\alpha.\tau$.

Proof of fundamental property of Fold case

For the Fold case, we assume $\Gamma \vdash \texttt{fold}\ e : \mu\alpha.\tau$ and need to show that $\Gamma \models \texttt{fold}\ e : \mu\alpha.\tau$.

We take some $k \geq 0$ and $\gamma \in \mathcal{G}_k[\![\Gamma]\!]$. We need to show that $\gamma(\texttt{fold}\ e) \in \mathcal{E}_k[\![\mu\alpha.\tau]\!]$ which is the same as $\texttt{fold}\ \gamma(e) \in \mathcal{E}_k[\![\mu\alpha.\tau]\!]$.

Proof of fundamental property of Fold case

For the Fold case, we assume $\Gamma \vdash \texttt{fold}\ e : \mu\alpha.\tau$ and need to show that $\Gamma \models \texttt{fold}\ e : \mu\alpha.\tau$.

We take some $k \geq 0$ and $\gamma \in \mathcal{G}_k[\![\Gamma]\!]$. We need to show that $\gamma(\texttt{fold}\ e) \in \mathcal{E}_k[\![\mu\alpha.\tau]\!]$ which is the same as $\texttt{fold}\ \gamma(e) \in \mathcal{E}_k[\![\mu\alpha.\tau]\!]$.

Take some $j < k$ with $\texttt{fold}\ \gamma(e) \to^j e'$ and $\mathrm{irred}(e')$, then we need to show $e' \in \mathcal{V}_{k-j}[\![\mu\alpha.\tau]\!]$.

Proof of fundamental property of Fold case

For the Fold case, we assume $\Gamma \vdash \texttt{fold}\ e : \mu\alpha.\tau$ and need to show that $\Gamma \models \texttt{fold}\ e : \mu\alpha.\tau$.

We take some $k \geq 0$ and $\gamma \in \mathcal{G}_k[\![\Gamma]\!]$. We need to show that $\gamma(\texttt{fold}\ e) \in \mathcal{E}_k[\![\mu\alpha.\tau]\!]$ which is the same as $\texttt{fold}\ \gamma(e) \in \mathcal{E}_k[\![\mu\alpha.\tau]\!]$.

Take some $j < k$ with $\texttt{fold}\ \gamma(e) \to^j e'$ and $\mathrm{irred}(e')$, then we need to show $e' \in \mathcal{V}_{k-j}[\![\mu\alpha.\tau]\!]$.

Then we know that $\gamma(e) \to^j e_1$ with $\texttt{fold}\ e_1 = e'$ and $\mathrm{irred}(e_1)$ and we have to show that $\texttt{fold}\ e_1 \in \mathcal{V}_{k-j}[\![\mu\alpha.\tau]\!]$.

Proof of fundamental property of Fold case

For the Fold case, we assume $\Gamma \vdash \texttt{fold}\ e : \mu\alpha.\tau$ and need to show that $\Gamma \models \texttt{fold}\ e : \mu\alpha.\tau$.

We take some $k \geq 0$ and $\gamma \in \mathcal{G}_k[\![\Gamma]\!]$. We need to show that $\gamma(\texttt{fold}\ e) \in \mathcal{E}_k[\![\mu\alpha.\tau]\!]$ which is the same as $\texttt{fold}\ \gamma(e) \in \mathcal{E}_k[\![\mu\alpha.\tau]\!]$.

Take some $j < k$ with $\texttt{fold}\ \gamma(e) \to^j e'$ and $\mathrm{irred}(e')$, then we need to show $e' \in \mathcal{V}_{k-j}[\![\mu\alpha.\tau]\!]$.

Then we know that $\gamma(e) \to^j e_1$ with $\texttt{fold}\ e_1 = e'$ and $\mathrm{irred}(e_1)$ and we have to show that $\texttt{fold}\ e_1 \in \mathcal{V}_{k-j}[\![\mu\alpha.\tau]\!]$.

By the induction hypothesis, $\Gamma \models e : \tau[\mu\alpha.\tau/\alpha]$, we know that $e_1 \in \mathcal{V}_{k-j}[\![\tau[\mu\alpha.\tau/\alpha]]\!]$. With the monotonicity lemma and definition of $\mathcal{V}_{k-j}[\![\mu\alpha.\tau]\!]$ we can show that $\texttt{fold}\ e_1 \in \mathcal{V}_{k-j}[\![\mu\alpha.\tau]\!]$.

Additional thoughts

If we forget the recursion, how do we retrieve the old interpretations?

Additional thoughts

If we forget the recursion, how do we retrieve the old interpretations?

$$\bigcap_{k\geq 0} \mathcal{V}_k[\![\tau]\!] = \mathcal{V}[\![\tau]\!],$$

$$\bigcap_{k\geq 0} \mathcal{E}_k[\![\tau]\!] = \mathcal{E}[\![\tau]\!].$$

Additional thoughts

If we forget the recursion, how do we retrieve the old interpretations?

$$\bigcap_{k \geq 0} \mathcal{V}_k[\![\tau]\!] = \mathcal{V}[\![\tau]\!],$$

$$\bigcap_{k \geq 0} \mathcal{E}_k[\![\tau]\!] = \mathcal{E}[\![\tau]\!].$$

We get that the fundamental property hold for all the typing derivations other than Fold and Unfold.

Additional thoughts

If we forget the recursion, how do we retrieve the old interpretations?

$$\bigcap_{k \geq 0} \mathcal{V}_k[\![\tau]\!] = \mathcal{V}[\![\tau]\!],$$

$$\bigcap_{k \geq 0} \mathcal{E}_k[\![\tau]\!] = \mathcal{E}[\![\tau]\!].$$

We get that the fundamental property hold for all the typing derivations other than Fold and Unfold.

And we still have the counterexample that Robbert gave last Thursday that if something is semantically well typed it doesn't have to be syntactically well typed:

```
if true then true else (true true)
```

Conclusion

- Recursive types
- Fixed-point type constructor
- Interpretations with step-index
- How semantically well typed implies safe for recursive types

Questions?