Safety of STLC with recursive types

Lucas van Kasteren s1039270
Rico te Wechel s4773039
27-11-2023

Naive way to "add" recursion

- Recursive types can be used to capture potentially infinite data structures.

- To demonstrate the utility of recursive types we use the

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

- Now suppose we try to type this term:

$$(\lambda x :?.xx)(\lambda x :?.xx)$$

Naive way to “add” recursion

- We recall the syntax of STLC

$$\tau ::= \text{bool} \mid \tau \to \tau$$
$$e ::= x \mid \text{true} \mid \text{false} \mid \text{if } e \text{ then } e \text{ else } e \mid \lambda x : \tau.\ e \mid e\ e$$
$$v ::= \text{true} \mid \text{false} \mid \lambda x : \tau.\ e$$
$$E ::= [] \mid \text{if E then } e \text{ else } e \mid \text{E } e \mid v \text{ E}$$

- What do we expect the Ω combinator to have for its type?

- Later we will see why recursive types can be helpful in typing the Ω combinator

Example data structure 'tree'

- Let us consider the inductive definition of tree

```
Inductive tree : Set :=
  | leaf : unit -> tree
  | node : int -> tree -> tree -> tree.
```

- We can rewrite this definition:

  *type tree = unit + int * tree * tree*

- Unfold the definition:
  *unit + int * tree * tree = unit + (int * (unit + (int * tree * tree)) * (unit + (int * tree * tree)))*

- We can define a fixpoint, which is a function f for which $x = f(x) \text{ for all } x \in dom(f)$.

- For tree we take some F such that *tree = F(tree)*

Example data structure 'tree'

Recall: *tree = F(tree)*

- For the sake of clarity we will use *tree* = *α* and as *unit type 1:*

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

- We use F in the recursive constructor μ:

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

- We substitute τ for F(α):

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

- Rewriting:

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

Formalizing STLC with recursive types

- We (again) recall the syntax of STLC

$$\tau ::= \text{bool} \mid \tau \to \tau$$
$$e ::= x \mid \text{true} \mid \text{false} \mid \text{if } e \text{ then } e \text{ else } e \mid \lambda x : \tau.\ e \mid e\ e$$
$$v ::= \text{true} \mid \text{false} \mid \lambda x : \tau.\ e$$
$$E ::= [] \mid \text{if E then } e \text{ else } e \mid \text{E } e \mid v \text{ E}$$

- Now we extend it with recursive types

$$\tau ::= 1 \mid \text{bool} \mid \tau \to \tau \mid \alpha$$
$$e ::= x \mid \langle\rangle \mid \text{true} \mid \text{false} \mid \lambda x.\ e \mid e\ e \mid \text{fold } e \mid \text{unfold } e$$
$$v ::= \langle\rangle \mid \text{true} \mid \text{false} \mid \lambda x : \tau.\ e \mid \text{fold } v$$
$$E ::= [] \mid \text{if E then } e_1 \text{ else } e_2 \mid \text{E } e \mid v \text{ E} \mid \text{fold } E \mid \text{unfold } E$$

E-FOLD

$$\frac{}{\text{unfold}(\text{fold } v) \mapsto v}$$

Typing the term Ω

- We add the following typing judgements:

$$\textsc{T-Unit}\quad \frac{}{\Gamma \vdash \langle\rangle : 1}$$

$$\textsc{T-Fold}\quad \frac{\Gamma \vdash e : \tau[\mu\alpha.\tau/\alpha]}{\Gamma \vdash \text{fold } e : \mu\alpha.\tau}$$

$$\textsc{T-Unfold}\quad \frac{\Gamma \vdash e : \mu\alpha.\tau}{\Gamma \vdash \text{unfold } e : \tau[\mu\alpha.\tau/\alpha]}$$

- Now how do we type $\Omega = (\lambda x.xx)(\lambda x.xx)$
- We define $\mathrm{SA} \triangleq \lambda x : ?.\ xx.$
- Hence we get Ω = SA SA

Typing the term Ω continued

- First we type x and say that $x : \mu\alpha.\ \alpha \to \tau$
- Now unfolding this type once gives $(\mu\alpha.\ \alpha \to \tau) \to \tau$

$$\text{T-UNFOLD}\quad \frac{\Gamma \vdash e : \mu\alpha.\tau}{\Gamma \vdash \text{unfold } e : \tau[\mu\alpha.\tau/\alpha]}$$

- We can *encode* the self application with $\lambda x : \mu\alpha.\ \alpha \to \tau.\ (\text{unfold } x)\ x$
- Hence SA is well typed, i.e. $\cdot \vdash \text{SA} : (\mu\alpha.\ \alpha \to \tau) \to \tau$
- Finally, if we encode $\Omega \triangleq \text{SA}\ (\text{fold SA})$
- Thus $\cdot \vdash \Omega : \tau$

Type Safety recap

- Type safety = "Well-typed programs do not go wrong" or "well-typed programs do not get *stuck*"

- Formally we say

$$\text{safe}(e) \triangleq \forall e'.\ e \mapsto^* e' \Rightarrow \text{val}(e') \vee (\exists e''.\ e' \mapsto e'')$$

- To prove type safety for STLC we recall the following

$$\begin{aligned}
\mathcal{V}[\![\text{bool}]\!] &\triangleq \{\text{true}, \text{false}\} \\
\mathcal{V}[\![\tau_1 \to \tau_2]\!] &\triangleq \{\lambda x : \tau_1.\ e \mid \forall v \in \mathcal{V}[\![\tau_1]\!].\ e[v/x] \in \mathcal{E}[\![\tau_2]\!]\} \\
\mathcal{E}[\![\tau]\!] &\triangleq \{e \mid \forall e'.\ e \mapsto^* e' \wedge \text{irred}(e') \Rightarrow e' \in \mathcal{V}[\![\tau]\!]\}
\end{aligned}$$

- Type safety is the best we get

The recursive type case

- To prove type safety for STLC extended with recursive types we might try to extend the following way

$$\mathcal{V}[\![\mu\alpha.\tau]\!] \triangleq \{\text{fold } v \mid \text{unfold } (\text{fold } v) \in \mathcal{E}[\![\tau[\mu\alpha.\tau/\alpha]]\!]\}$$

- But..

$$\text{E-FOLD}\quad \frac{}{\text{unfold}(\text{fold } v) \mapsto v}$$

- So this means we get

$$\mathcal{V}[\![\mu\alpha.\tau]\!] \triangleq \{\text{fold } v \mid v \in \mathcal{V}[\![\tau[\mu\alpha.\tau/\alpha]]\!]\}$$

- Problem: This breaks well-foundedness

STLC type safety enabling recursive types

- Solution: Step-indexed logical relations

$$\mathcal{V}_k[\![\text{bool}]\!] \triangleq \{\text{true}, \text{false}\}$$

$$\mathcal{V}_k[\![\tau_1 \rightarrow \tau_2]\!] \triangleq \{\lambda x : \tau_1.\ e \mid \forall j < k.\ \forall v \in \mathcal{V}_j[\![\tau_1]\!].\ e[v/x] \in \mathcal{E}_j[\![\tau_2]\!]\}$$

$$\mathcal{V}_k[\![\mu\alpha.\tau]\!] \triangleq \{\text{fold } v \mid \forall j < k.\ v \in \mathcal{V}_j[\![\tau[\mu\alpha.\tau/\alpha]]\!]\}$$

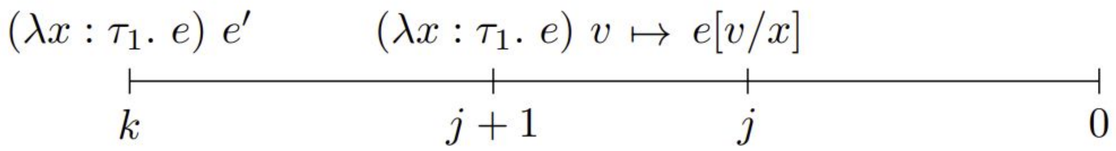
$$\mathcal{E}_k[\![\tau]\!] \triangleq \{e \mid \forall j < k.\ \forall e'.\ e \mapsto_j e' \wedge \text{irred}(e') \Rightarrow e' \in \mathcal{V}_{k-j}[\![\tau]\!]\}$$

$$\mathcal{G}_k[\![\cdot]\!] \triangleq \emptyset$$

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

Understanding the relations

$$\mathcal{V}_k[\![\tau_1 \to \tau_2]\!] \triangleq \{\lambda x : \tau_1.\ e \mid \forall j < k.\ \forall v \in \mathcal{V}_j[\![\tau_1]\!].\ e[v/x] \in \mathcal{E}_j[\![\tau_2]\!]\}$$



$$\mathcal{E}_k[\![\tau]\!] \triangleq \{e \mid \forall j < k.\ \forall e'.\ e \mapsto_j e' \wedge \mathrm{irred}(e') \Rightarrow e' \in \mathcal{V}_{k-j}[\![\tau]\!]\}$$

Recap: The fundamental property

- Recall

(A) For all terms e if $\cdot \vdash e : \tau$ then $\cdot \vDash e : \tau$

(B) For all terms e if $\cdot \vDash e : \tau$ then safe(e)

$$\mathcal{G}[\![\cdot]\!] \triangleq \{\emptyset\}$$

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

$$\Gamma \vDash e : \tau \triangleq \forall \gamma \in \mathcal{G}[\![\Gamma]\!], \gamma(e) \in \mathcal{E}[\![\tau]\!]$$

Fundamental property allowing recursive types

- Now

$$\Gamma \vDash e : \tau \triangleq \forall \gamma \in \mathcal{G}[\![\Gamma]\!], \gamma(e) \in \mathcal{E}[\![\tau]\!]$$

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

- Proof
  - Monotonicity lemma
  - Induction on typing judgment

Lemma (Monotonicity). *If* $v \in \mathcal{V}_k[\![\tau]\!]$ *and* $j \leq k$ *then* $v \in \mathcal{V}_j[\![\tau]\!]$.

T-FOLD

$$\frac{\Gamma \vdash e : \tau[\mu\alpha.\tau/\alpha]}{\Gamma \vdash \text{fold } e : \mu\alpha.\tau}$$

$$\mathcal{E}_k[\![\tau]\!] \triangleq \{e \mid \forall j < k.\ \forall e'.\ e \mapsto_j e' \land \text{irred}(e') \Rightarrow e' \in \mathcal{V}_{k-j}[\![\tau]\!]\}$$

Thank you!