Compositional Programming with Full Iso-recursive Types

Litao Zhou <ltzhou@cs.hku.hk>
Supervised by Bruno C. d. S. Oliveira
The University of Hong Kong

The Expression Problem

How to resolve the dilemma posed by the modular extension for both **data type variants** and **their operations** in object-oriented and functional programming

Compositional Programming

A **statically typed** modular programming language that addresses the Expression Problem, featuring:

✓ Compositional interface (enabled by intersection types)
✓ Nested trait composition (via the merge operator)
✗ Support for recursive types

Expressions

```
type LitSig<Exp> = {
  Lit : Int -> Exp
}
```

```
evalLit (self: Top) = {
  Lit (val: Int) =
    { eval = val };
};
```

```
doubleLit<Exp> (self: LitSig<Exp>) = {
  Lit (val: Int) = {
    double = self.Lit (val + val)
};};
```

```
type AddSig<Exp> = {
  Add : Exp -> Exp
            -> Exp
}
```

```
evalAdd (self: Top) = {
  Add (l r: Eval) =
    { eval = l.eval + r.eval }
};
```

```
doubleAdd<Exp> (self: AddSig<Exp>) = {
  Add (l r: Double<Exp>) = {
    double = self.Add l.double r.double
};};
```

```
type Eval = {eval : Int}
```

```
type Double<Exp> = {double : Exp}
```

Operations

Iso-recursive Types

Recursive types can be used to model objects:

```
interface IExp {
  eval   : Int,
  double : IExp }
```

represented as

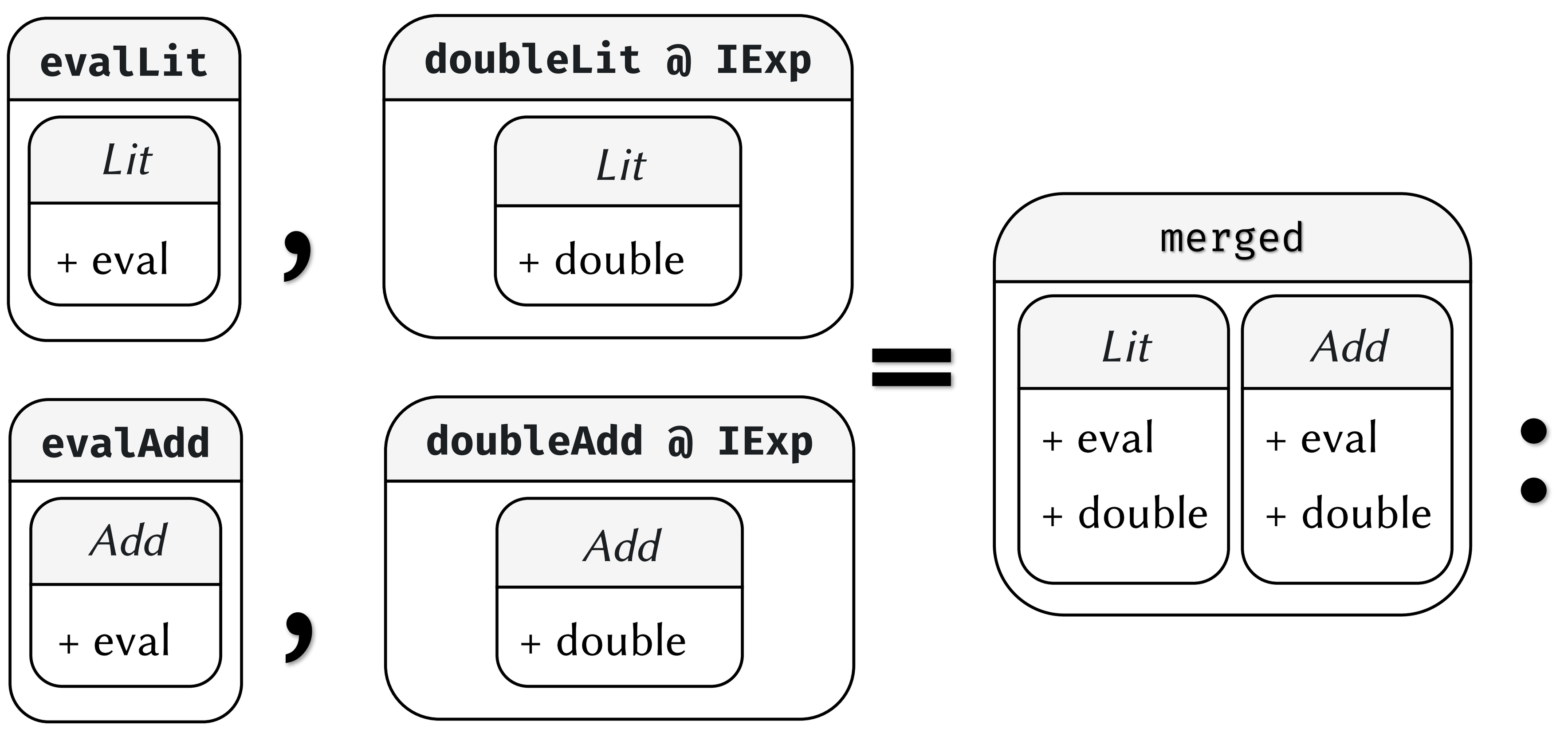
```
type IExp = μ a. {
  eval   : Int,
  double : a    }
```

Iso-recursive types and their unfoldings are converted via term level constructs:

$\boxed{\vdash e : A}$ *(Iso-recursive typing)*

$$\textsc{IsoTyp-fold}\ \frac{\vdash e : A[\mu\alpha.A/\alpha]}{\vdash \textbf{fold}\ [\mu\alpha.A] e : \mu\alpha.A} \qquad \textsc{IsoTyp-unfold}\ \frac{\vdash e : \mu\alpha.A}{\vdash \textbf{unfold}\ [\mu\alpha.A] e : A[\mu\alpha.A/\alpha]}$$

Challenge: Iso-recursive Types are not Enough



With standard iso-recursive types, we cannot insert fold operators to get the desired typing, since we implement the components of the folded expression separately.

Actual type

```
{Lit : Int  -> IExp_ufd
 Add : IExp -> IExp
            -> IExp_ufd}
```

How? ↪

Desired type

```
{Lit : Int  -> IExp
 Add : IExp -> IExp
            -> IExp}
```

where `IExp_ufd` is the unfolding of `IExp`
`IExp_ufd = {eval:Int, double:IExp}`

Solution: Full Iso-recursive Types

We present a novel formulation of iso-recursive typing. Now, (un)foldings can take place anywhere within an expression, enabled by a **casting operator**

$$\begin{array}{lll} e ::= & \ldots \mid \textbf{cast}[c]e & \textit{(Expressions)} \\ v ::= & \ldots \mid \textbf{cast}[\textsf{fold}_A]v \mid \textbf{cast}[c_1 \to c_2]v & \textit{(Values)} \\ c ::= & \textsf{unfold}_A \mid \textsf{fold}_A \mid \textsf{id} \mid c_1 \to c_2 \mid \ldots & \textit{(Casting operators)} \end{array}$$

We develop a calculus with record types, disjoint intersection types, BCD subtyping, iso-recursive types, and the casting operators, which adopts a call-by-value small step semantics.

$\boxed{e \hookrightarrow e'}$ *(Reduction)*

$$\textsc{Red-cast}\ \frac{e \hookrightarrow e'}{\textbf{cast}\ [c] e \hookrightarrow \textbf{cast}\ [c] e'} \qquad \textsc{Red-castelim}\ \frac{}{\textbf{cast}\ [\textsf{unfold}_A] (\textbf{cast}\ [\textsf{fold}_A] v) \hookrightarrow v}$$

We formally prove the soundness of the type system in Coq.

$\boxed{\vdash A \hookrightarrow B : c}$ *(Type casting)*

$$\textsc{TCast-arrow}\ \frac{\vdash A_2 \hookrightarrow A_1 : c_1 \qquad \vdash B_1 \hookrightarrow B_2 : c_2}{\vdash A_1 \to B_1 \hookrightarrow A_2 \to B_2 : c_1 \to c_2}$$

$$\textsc{TCast-fold}\ \frac{}{\vdash A[\mu\alpha.A/\alpha] \hookrightarrow \mu\alpha.A : \textsf{fold}_{\mu\alpha.A}}$$

$$\textsc{TCast-id}\ \frac{}{\vdash A \hookrightarrow A : \textsf{id}}$$

$\boxed{\vdash e : A}$ *(Full iso-recursive typing)*

$$\textsc{Typ-cast}\ \frac{\vdash e : A \qquad \vdash A \hookrightarrow B : c}{\vdash \textbf{cast}\ [c] e : B}$$

For example, with the cast operator,
`{Lit : id -> fold`$_{\texttt{IExp}}$`}`,
`{Add : id -> id -> fold`$_{\texttt{IExp}}$`}`
can be applied to the `merged` expression and achieve the desired typing

Future Work

- By extending $\lambda^{+}_{\mu i}$ to **polymorphism**, we can achieve the full power of Compositional Programming
- Our encoding can also be applied to modularly compose objects with **binary methods**
- Full iso-recursive types provides a new (and perhaps more direct) way to **relate iso- and equi-recursive types**