tF example

- Last time: finally got a type-safe universal language (tF)

```t
\[ t ::= x \mid b \mid \mathrm{fn} \ x \ . \ e \mid e_1 \ e_2 \mid \langle e_1, e_2 \rangle \mid \text{first} \ e \mid \text{rest} \ e \mid \text{inl} \ e \mid \text{inr} \ e \mid \text{case} \ e_0 \ e_1 \ e_2 \mid \text{rec} \ y \ . \ e \mid \text{let} \ x = e_1 \ \text{in} \ e_2 \]
```

\[ \tau ::= B \mid \tau_1 \rightarrow \tau_2 \mid \tau_1^* \tau_2 \mid \tau_1 + \tau_2 \]

```
let factorial = (rec fact: int \rightarrow int.
  fn n. if (n<2) 1 (fact(n-1)*n)) in : int
```

Other examples:

- How to assign a type to \((\lambda \ x \ (x \ x))\)?
- Can write terms that don’t get stuck:
  \((\lambda \ x \ (x \ x)) \ (\lambda \ y \ #f)\)
  \((\lambda \ x \ (x \ x)) \ (\lambda \ y \ #f)\)
- Need solution to \(\tau = \tau \rightarrow \text{bool}\)
Fixed point type constructor
- Want to solve equations of form \( X = \tau \) where \( X \) is
  type variable mentioned in type expression \( \tau \)
- Type constructor \( \mu X. \tau \) produces this solution
  – analogue of \( \text{rec } x \; e \) for types
  = ML datatype declaration
- Can define useful types:
  \[
  \begin{align*}
  \text{tree} & \triangleq \mu T. \; T \times T + \text{int} \\
  \text{nat} & \triangleq \mu N. \; \text{unit} + N \\
  0 & \triangleq \text{inl}(\#u), 1 \triangleq \text{inr}(\text{inl}(\#u)), 2 \triangleq \text{inr}(\text{inr}(\text{inl}(\#u))) \ldots \\
  \text{successor} & \triangleq \lambda n: \text{nat}. \; \text{inr}(n)
  \end{align*}
  \]

Closed vs. Open recursion
- Many modern languages allow types to refer to
  one another arbitrarily (even to other source file!)
- Open recursion: type expression not closed
  Requires fixed point over all types in scope
- Example:
  \[
  \begin{align*}
  \text{class Node} & \{ \\
  \text{Edge}[\] & \text{ outgoing_edges}; \\
  \text{Edge}[\] & \text{ incoming_edges}; \\
  \text{Node from; } & \text{ Node to; }
  \}
  \end{align*}
  \[
  \begin{align*}
  \text{class Edge} & \{ \\
  \text{Node from; } & \text{ Node to; }
  \}
  \end{align*}
  \]
  \[
  \begin{align*}
  \text{Node} & \triangleq \mu N. \; \text{array}[N] \times \text{array}[N] \\
  \text{Edge} & \triangleq \mu E. \; (\text{array}[E] \times \text{array}[E]) \times (\text{array}[E] \times \text{array}[E])
  \end{align*}
  \]

Type equivalence
- Problem: types no longer have one unique
  syntactic form
  \( \mu X. \tau \) is solution to \( X = \tau \) can substitute \( \mu X. \tau \) for \( X \) wherever it appears in \( \tau \)
- Unfolding of type \( \mu X. \tau \) is equivalent type
  \( \tau[\mu X. \tau / X] \)
  - implicit notion of equivalence: type expressions are
    fully substitutable for each other
  - weaker notion: types are isomorphic, expressions
    must be explicitly mapped between types

Abstract and concrete views
- unfold operator allows access to internals of value of
  recursive type; fold operator packages concrete value as
  abstract value
- Winskel: abs/rep
- fold/unfold are bijection: types are isomorphic

Typing, evaluation rules
\[
\begin{align*}
\Gamma \vdash e : \tau & \Rightarrow \mu X. \tau / X \\
\Gamma \vdash \text{fold}_{\mu X. \tau} \; e : \mu X. \tau \\
\Gamma \vdash e : \mu X. \tau \\
\Gamma \vdash \text{unfold} \; e : \tau[\mu X. \tau / X]
\end{align*}
\]

Example
- Goal: show \((\lambda x \; (x \; x))\) can be typed as
  \( \mu T. \; T \rightarrow \text{bool} \)
- Add declarations, explicit fold and unfold:
  \[
  \begin{align*}
  \text{fold}_{\mu T. \; T \rightarrow \text{bool}} (\lambda x: (\mu T. \; T \rightarrow \text{bool}) \rightarrow \text{bool}) \rightarrow (\text{unfold} \; x) \; x
  \end{align*}
  \]

\[
\begin{align*}
\Gamma \vdash x : (\mu T. \; T \rightarrow \text{bool}) & \Rightarrow x : (\mu T. \; T \rightarrow \text{bool}) \\
\Gamma \vdash \text{fold}_{\mu T. \; T \rightarrow \text{bool}} (\lambda x: (\mu T. \; T \rightarrow \text{bool}) \rightarrow \text{bool}) & \Rightarrow (\text{unfold} \; x) \; x : \mu T. \; T \rightarrow \text{bool}
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash \text{fold}_{\mu T. \; T \rightarrow \text{bool}} (\lambda x: (\mu T. \; T \rightarrow \text{bool}) \rightarrow \text{bool}) \rightarrow (\text{unfold} \; x) \; x : \mu T. \; T \rightarrow \text{bool}
\end{align*}
\]
Capturing the untyped λ

- All lambda calculus expressions can be used in any context
  - Evaluation never gets stuck
- Type of lambda calculus terms:
  \[ \Lambda = \mu X. X \to X \equiv (\mu X. X \to X) \to (\mu X. X \to X) \]
- Translating λ to λ→µ:
  \[ (\lambda x : \Lambda . e) = (\text{fold } (\lambda X : \Lambda . \varnothing (e))) \]
  \[ (\varnothing (e_1) \varnothing (e_2)) = (\text{unfold } \varnothing (e_1) \varnothing (e_2)) \]

fold and unfold in real languages

- Java, Modula-3: recursive type and unfolding are substitutable: fold/unfold supplied automatically as needed
- ML: datatype constructor is fold, match operation provides implicit unfold for each arm of the sum
- CLU, C requires explicit use of operators to shift between views (up/down, (&/*))

Note:
fold and unfold are combined with other features (ML: sums, CLU & Java: classes/modules, C: references)

Y operator

- Can use recursive types to write Y operator as ordinary expression
- Deugr recf : τ as Y τ (λf : τ . er)!

Constructing a model

- Our semantics models τ as domain \( \mathcal{J} [\tau] \)
- How do we model the new type constructor?
  \( \mathcal{J} [\mu X. \tau] = ? \)
  Since \( \mu X. \tau \equiv \tau \{ \mu X. \tau / X \} \), we expect isomorphism to hold in domains as well:
  \( \mathcal{J} [\mu X. \tau] \cong \mathcal{J} \{ \mu X. \tau / X \} \)
- Example: natural numbers
  \( N = \mathcal{J} [\mu N. \text{unit} + N] \cong \mathcal{J} \{ \text{unit} + (\mu N. \text{unit} + N) \} \)
  \( N \cong \text{unit} + N \)
- Modeling these types requires solutions to domain equations we have been using all along

Recursive domain constructor

- Idea: assume a constructor for recursive domains: \( \mu D . \mathcal{J}(D) \)
  - Functor \( \mathcal{J} \) maps one domain into another domain
  - If \( D = \mu X. \mathcal{J}(X) \), \( \mathcal{J}(D) \) produces a domain related to \( D \) by continuous functions \( \text{up} \) and \( \text{down} \) that are inverses of one another
  \[
  \begin{align*}
  d_0 \sqsubseteq d_1 \sqsubseteq d_2 \ldots & \Rightarrow \text{up}(\sqcup d_i) = \sqcup \text{up}(d_i) \\
  e_0 \sqsubseteq e_1 \sqsubseteq e_2 \ldots & \Rightarrow \text{down}(\sqcup e_i) = \sqcup \text{down}(e_i)
  \end{align*}
  \]

Interpreting types

- Types can define names; need type environment \( \chi : \text{Type} \to \text{Domain} \) to define inductively
  \( \mathcal{J}[\text{unit}]_\chi = \top \)
  \( \mathcal{J}[\text{int}]_\chi = \mathbb{Z} \)
  \( \mathcal{J}[\text{X}]_\chi = \chi(X) \)
  \( \mathcal{J}[\tau_1 \to \tau_2]_\chi = \mathcal{J}[\tau_1]_\chi \to \mathcal{J}[\tau_2]_\chi \)
  \( \mathcal{J}[\mu X. \tau]_\chi = \mu \mathcal{D}. \mathcal{J}[\tau]_\chi[X \to \mathcal{D}] \)