Aggregate Data Structures

COS 441
Princeton University
Fall 2004
Data-Structures for MinML

• Products Types
  – n-tuples general case, will study 0-tuple (unit) and 2-tuples (pairs)

• Sums Types
  – Tagged types (We can just study binary tags)

• Recursive Types
  – Lists, Trees, … etc.
Product Types

Types \( \tau ::= \text{unit} \mid \tau_1 \times \tau_2 \)

Expressions \( e ::= () \mid \text{check } e_1 \text{ is } () \text{ in } e_2 \mid (e_1, e_2) \mid \text{split } e_1 \text{ as } (x, y) \text{ in } e_2 \)

Values \( v ::= () \mid (v_1, v_2) \)
Products: Static Semantics

\( \Gamma \vdash () : \text{unit} \)

\[
\begin{array}{c}
\Gamma \vdash e_1 : \text{unit} \quad \Gamma \vdash e_2 : \tau_2 \\
\Gamma \vdash \text{check } e_1 \text{ is } () \text{ in } e_2 : \tau_2
\end{array}
\]

\[
\begin{array}{c}
\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \\
\Gamma \vdash (e_1, e_2) : \tau_1 \ast \tau_2
\end{array}
\]

\[
\begin{array}{c}
\Gamma \vdash e_1 : \tau_1 \ast \tau_2 \quad \Gamma, x: \tau_1, y: \tau_2 \vdash e_2 : \tau \\
\Gamma \vdash \text{split } e_1 \text{ as } (x, y) \text{ in } e_2 : \tau
\end{array}
\]
Products: Dynamic Semantics

\[
\text{check}() \text{ is } () \text{ in } e \rightarrow e
\]

\[
e_1 \rightarrow e'_1
\]

\[
\text{check}_1 \text{ is } () \text{ in } e_2 \rightarrow \text{check}_1' \text{ is } () \text{ in } e_2
\]

\[
\text{split}_1 (v_1, v_2) \text{ as } (x, y) \text{ in } e \rightarrow \{v_1, v_2/x, y \} e
\]

\[
e_1 \rightarrow e'_1
\]

\[
\text{split}_e_1 \text{ as } (x, y) \text{ in } e_2 \rightarrow \text{split}_e_1' \text{ as } (x, y) \text{ in } e_2
\]

\[
\begin{align*}
e_1 & \rightarrow e'_1 \\
(e_1, e_2) & \rightarrow (e'_1, e_2)
\end{align*}
\]

\[
\begin{align*}
e_2 & \rightarrow e'_2 \\
(v_1, e_2) & \rightarrow (v_1, e'_2)
\end{align*}
\]
Sum Types

Types  \( \tau ::= \tau_1 + \tau_2 \)

Expressions  \( e ::= \text{inl}_{\tau_1 + \tau_2}(e_1) | \text{inr}_{\tau_1 + \tau_2}(e_2) | \text{case}_\tau e_0 \text{ of } \text{inl}(x : \tau_1) \Rightarrow e_1 | \text{inr}(y : \tau_2) \Rightarrow e_2 \)

Values  \( \nu ::= \text{inl}_{\tau_1 + \tau_2}(\nu_1) | \text{inr}_{\tau_1 + \tau_2}(\nu_2) \)
Sums: Static Semantics

\[ \Gamma \vdash e_1 : \tau_1 \]

\[ \Gamma \vdash \text{inl}_{\tau_1 + \tau_2} (e_1) : \tau_1 + \tau_2 \]

\[ \Gamma \vdash e_2 : \tau_2 \]

\[ \Gamma \vdash \text{inl}_{\tau_1 + \tau_2} (e_2) : \tau_1 + \tau_2 \]

\[ \Gamma \vdash e_0 : \tau_1 + \tau_2 \quad \Gamma, x_1: \tau_1 \vdash e_1 : \tau \quad \Gamma, x_2: \tau_2 \vdash e_2 : \tau \]

\[ \Gamma \vdash \text{case}_\tau e_0 \text{ of } \text{inl}(x_1: \tau_1) \Rightarrow e_1 \mid \text{inr}(x_2: \tau_2) \Rightarrow e_2 : \tau \]
Sums: Dynamic Semantics

\[
\frac{e \leftrightarrow e'}{\text{in}l_{\tau_1 + \tau_2}(e) \leftrightarrow \text{in}l_{\tau_1 + \tau_2}(e')}
\]

\[
\frac{e \leftrightarrow e'}{\text{in}r_{\tau_1 + \tau_2}(e) \leftrightarrow \text{in}r_{\tau_1 + \tau_2}(e')}
\]

\[
\text{case}_\tau \text{in}l_{\tau_1 + \tau_2}(v) \text{ of } \text{in}l(x_1: \tau_1) \Rightarrow e_1 \mid \text{in}r(x_2: \tau_2) \Rightarrow e_2 \mapsto \{v/x_1\}e_1
\]

\[
\text{case}_\tau \text{in}r_{\tau_1 + \tau_2}(v) \text{ of } \text{in}l(x_1: \tau_1) \Rightarrow e_1 \mid \text{in}r(x_2: \tau_2) \Rightarrow e_2 \mapsto \{v/x_2\}e_2
\]
Recursive Types

Types \( \tau ::= t \mid \text{rect is } \tau \)

Expressions \( e ::= \text{roll}(e) \mid \text{unroll}(e) \)

Values \( v ::= \text{roll}(v) \)
Rec. Types: Static Semantics

\[
\Gamma \vdash e : \{rect\ is\ \tau/t\}\tau \\
\frac{}{\Gamma \vdash \text{roll}(e) : \text{rect is } \tau}
\]

\[
\frac{}{\Gamma \vdash e : \text{rect is } \tau} \\
\frac{}{\Gamma \vdash \text{unroll}(e) : \{\text{rect is } \tau/t\}\tau}
\]
datatype nat = Zero | Succ of nat

e : rec t is (Zero + Succ of t)
unroll(e) : {(rec t is (Zero + Succ of t))/t}
            (Zero + Succ of t)
unroll(e) :
            (Zero + Succ of (rec t is (Zero + Succ of t)))
Rec. Types: Dynamic Semantics

\[
\text{unroll}(\text{roll}(v)) \leftrightarrow v
\]

\[
e \leftrightarrow e' \\
\text{unroll}(e) \leftrightarrow \text{unroll}(e')
\]

\[
e \leftrightarrow e' \\
\text{roll}(e) \leftrightarrow \text{roll}(e')
\]
Derived Types

\[ \text{bool} \cong \text{unit} + \text{unit} \]

\[ \text{ilist} \cong \\
\text{rec } t \text{ is } (\text{unit} + (\text{int} \times t)) \]
Examples: Booleans

\[ \text{true} \equiv \text{inl}_{\text{unit} + \text{unit}}(()), \]
\[ \text{false} \equiv \text{inr}_{\text{unit} + \text{unit}}(()), \]
\[ \text{if}(e, e_1, e_2) \equiv \]
\[ \quad \text{case}_\tau e_0 \text{ of} \]
\[ \quad \quad \text{inl}(x: \text{unit}) \Rightarrow e_1 \]
\[ \quad \quad \text{inr}(y: \text{unit}) \Rightarrow e_2 \]
Examples: Lists

<table>
<thead>
<tr>
<th>ilist ≡</th>
</tr>
</thead>
<tbody>
<tr>
<td>rec t is (unit + (int * t))</td>
</tr>
</tbody>
</table>

| nil ≡ roll(inl unit +(int * ilist)(())) |

| cons(e₁, e₂) ≡ |
| roll(inr unit +(int * ilist)((e₁, e₂))) |
Examples: Lists

\[
\begin{align*}
\text{listcase } e \text{ of } & \\
\quad \text{nil} & \Rightarrow e_1 \mid \text{cons}(x,y) & \Rightarrow e_2 \\
\cong & \\
\text{case unroll}(e_0) \text{ of } & \\
\quad \text{inl}(\_\text{:unit}) & \Rightarrow e_1 \\
\mid \text{inr}(z\text{:int } \ast \text{ ilist}) & \Rightarrow \\
\quad \text{split } z \text{ as (x,y) in } & e_2
\end{align*}
\]
Roll/Unroll

• The roll and unroll primitives are there just to make the proof a bit easier and to guarantee syntax directed checking.
• They are not needed in a realistic implementation only act as a notational fiction.
• Compare to inl/inr which have a real computational purpose.
Some Subtle Bugs

• Harper’s notes present rules/syntax that are “buggy”
• Rules presented in a fashion that makes implementing type checking in a bottom up way impossible
• Also contains some redundant info
• Abstract syntax corrected in homework
Checking vs Inference

• We can view the relation $\Gamma \vdash e : \tau$ in several ways
• A predicate on environments expressions and types
  
  \[
  \text{val check : (env * exp * typ) -> bool}
  \]
• A function partial function that computes the type of an expression
  
  \[
  \text{val infer : (env * exp) -> typ}
  \]
Type Annotations

• Abstract syntax is “decorated” with type information to make checking/inference syntax directed

• More sophisticated implementations of checkers can avoid the need for type information
Case

Can infer $\tau$ by computing type of $e_1$ and $e_2$

$$\frac{\Gamma \vdash e_0 : \tau_1 + \tau_2 \quad \Gamma, x_1 : \tau_1 \vdash e_1 : \tau \quad \Gamma, x_2 : \tau_2 \vdash e_2 : \tau}{\Gamma \vdash \text{case}_\tau e_0 \text{ of } \text{inl}(x_1 : \tau_1) \Rightarrow e_1 | \text{inr}(x_2 : \tau_2) \Rightarrow e_2 : \tau}$$

redundant

Worth keeping for better error reporting.
Recursive Types

If we compute the type for e still must “guess” $\tau$ to check rule

\[
\frac{\Gamma \vdash e : \{\text{rect is } \tau/t\} \tau}{\Gamma \vdash \text{roll}(e) : \text{rect is } \tau}
\]

Homework annotates roll to avoid need to “guess”
Example

\[ \text{roll}(\text{inl}_{\text{unit} + (\text{rec } t \text{ is } (\text{unit} \times t))}(\text{unit})) : \text{rec } t \text{ is } \tau \]

What should \( \tau \) be for the above to type check?
Example

\[
\begin{align*}
\text{roll}(\text{inl}_{\text{unit} + (\text{rec}\ t\ \text{is}\ (\text{unit} + t))}(\text{unit})) : \text{rec}\ t\ \text{is}\ \tau \\
\text{What should}\ \tau\ \text{be for the above to type check?}
\end{align*}
\]

\[
\begin{align*}
\tau &= (\text{unit} + t) \\
\{\text{rec}\ t\ \text{is}\ (\text{unit} + t) / t\} (\text{unit} + t) &= \text{unit} + \text{rec}\ t\ \text{is}\ (\text{unit} + t)
\end{align*}
\]
Summary

• Can express many interesting type using these primitive type constructors
• Algorithmic issues of type-checking sometime clutter the presentation of the rules