Semantics for MinML

COS 441
Princeton University
Fall 2004

Fib in MiniML

fun fib(n:int):int is
    if <(n,2) then 1
    else +(apply(fib,-(n,1)),
        apply(fib,-(n,2)))

MinML Abstract Syntax

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Operator</th>
<th>Arity</th>
</tr>
</thead>
<tbody>
<tr>
<td>fun(x:τ₁):τ₂ is</td>
<td>int</td>
<td>()</td>
</tr>
<tr>
<td></td>
<td>bool</td>
<td>()</td>
</tr>
<tr>
<td></td>
<td>→</td>
<td>(0,0)</td>
</tr>
<tr>
<td></td>
<td>n</td>
<td>()</td>
</tr>
<tr>
<td></td>
<td>o</td>
<td>(0,...,0)</td>
</tr>
<tr>
<td></td>
<td>fun</td>
<td>(0,0,2)</td>
</tr>
<tr>
<td></td>
<td>apply</td>
<td>(0,0)</td>
</tr>
<tr>
<td></td>
<td>true</td>
<td>()</td>
</tr>
<tr>
<td></td>
<td>false</td>
<td>()</td>
</tr>
</tbody>
</table>

Type Environment

• Type environment (Γ) maps names to type
• [x:τ] extends environment defined as

$$\Gamma[x:τ](y) = \begin{cases} \tau & \text{if } y = x \\ \Gamma(y) & \text{otherwise} \end{cases}$$

• Typing rule for names is

$$\Gamma(x) = τ \quad \Gamma \vdash x : τ$$

ABT for Fib in MiniML

fun(int,int,
    (fib,n,if(<(n,2),1,
        +(apply(fib,-(n,1)),
            apply(fib,-(n,2))))))
Integers

\[ \Gamma \vdash n : \text{int} \]
\[ \Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int} \]
\[ \Gamma \vdash + (e_1, e_2) : \text{int} \]
\[ \Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int} \]
\[ \Gamma \vdash - (e_1, e_2) : \text{int} \]
\[ \Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int} \]
\[ \Gamma \vdash * (e_1, e_2) : \text{int} \]

Booleans

\[ \Gamma \vdash \text{false} : \text{bool} \quad \Gamma \vdash \text{true} : \text{bool} \]
\[ \Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int} \]
\[ \Gamma \vdash \langle e_1, e_2 \rangle : \text{bool} \]
\[ \Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int} \]
\[ \Gamma \vdash \langle e_1, e_2 \rangle : \text{bool} \]
\[ \Gamma \vdash e : \text{bool} \quad \Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau \]
\[ \Gamma \vdash \text{if } e \text{ then } e_1 \text{ else } e_2 : \tau \]

Booleans

\[ \Gamma \vdash \text{false} : \text{bool} \quad \Gamma \vdash \text{true} : \text{bool} \]
\[ \Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int} \]
\[ \Gamma \vdash \langle e_1, e_2 \rangle : \text{bool} \]
\[ \Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int} \]
\[ \Gamma \vdash \langle e_1, e_2 \rangle : \text{bool} \]
\[ \Gamma \vdash e : \text{bool} \quad \Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau \]
\[ \Gamma \vdash \text{if } e \text{ then } e_1 \text{ else } e_2 : \tau \]

Functions

\[ \Gamma \vdash f : \tau_1 \rightarrow \tau_2 \]
\[ \binom{x_1 : \tau_1}{\Gamma} \vdash c : \tau_2 \quad (f, x \notin \text{dom}(\Gamma)) \]
\[ \Gamma \vdash \text{fun } f : (x_1 : \tau_1 : \tau_2) \quad \text{is } c : \tau_1 \rightarrow \tau_2 \]
\[ \Gamma \vdash c_1 : \tau_1 \rightarrow \tau \quad \Gamma \vdash c_2 : \tau_2 \]
\[ \Gamma \vdash \text{apply}(c_1, c_2) : \tau \]

Inversion Theorems

\[ \text{Theorem 9.1 (Inversion)} \]
\[ 1. \text{If } \Gamma \vdash x : \tau, \text{ then } \Gamma(x) = \tau. \]
\[ 2. \text{If } \emptyset \vdash x : \tau, \text{ then } \tau = \text{int}. \]
\[ 3. \text{If } \emptyset \vdash \text{true} : \tau, \text{ then } \tau = \text{bool}, \text{ and similarly for false.} \]
\[ 4. \text{If } \Gamma \vdash \text{if } \text{true} \text{ else } \text{false} : \tau, \text{ then } \Gamma \vdash e : \text{bool}, \Gamma \vdash e_1 : \tau \quad \text{and} \quad \Gamma \vdash e_2 : \tau. \]
\[ 5. \text{If } \Gamma \vdash \text{fun } f : (x_1 : \tau_1) : \text{int} : \tau, \text{ then } \Gamma [x_1 : \tau_1] [x_1 : \tau_1] \vdash c : \tau_2 \quad \text{and} \quad \tau = \tau_1 \rightarrow \tau_2. \]
\[ 6. \text{If } \Gamma \vdash \text{apply}(e_1, e_2) : \tau, \text{ then there exists } \tau_2 \text{ such that } \Gamma \vdash e_1 : \tau_2 \rightarrow \tau \quad \text{and} \quad \Gamma \vdash e_2 : \tau_2. \]

Substitution and Weakening

\[ \text{Lemma 9.2} \]
\[ 1. \text{Typing is not affected by “junk” in the symbol table. If } \emptyset \vdash x : \tau \text{ and} \quad \emptyset \vdash c : \tau, \text{ then } \emptyset \vdash c : \tau. \]
\[ 2. \text{Substitution for a variable with type } \tau \text{ by an expression of the same type doesn’t affect typing. If } \Gamma [x : \tau] \vdash c' : \tau' \quad \text{and} \quad \Gamma \vdash c : \tau, \text{ then} \quad \emptyset \vdash c' [x : \tau]. \]
Transition Semantics
Set of states are *closed* expressions
Set of final states (F) are *values* (v)
\[ v ::= \text{true} \mid \text{false} \mid n \mid \text{fun}(\tau_1, \tau_2, (l, x, e)) \]

Primitive “Instructions”
\[ + (m, n) \mapsto m + n \]
\[ \text{if true then } \mathcal{e}_1 \text{ else } \mathcal{e}_2 \mapsto \mathcal{e}_1 \]
\[ \text{if false then } \mathcal{e}_1 \text{ else } \mathcal{e}_2 \mapsto \mathcal{e}_2 \]
\[ (v = \text{fun}(f : \tau_1 : \tau_2 : e)) \]
\[ \text{apply}(v, v_1) \mapsto \{v, v_1/f, x\}e \]

Search Rules
\[ \mathcal{e}_1 \mapsto \mathcal{e}'_1 \]
\[ + (\mathcal{e}_1, \mathcal{e}_2) \mapsto + (\mathcal{e}'_1, \mathcal{e}_2) \]
\[ \mathcal{e}_2 \mapsto \mathcal{e}'_2 \]
\[ + (v_1, \mathcal{e}_2) \mapsto + (v_1, \mathcal{e}'_2) \]
\[ v \mapsto v' \]
\[ \text{if } \mathcal{e}_1 \text{ then } \mathcal{e}_1 \text{ else } \mathcal{e}_2 \mapsto \text{if } \mathcal{v}' \text{ then } \mathcal{e}_1 \text{ else } \mathcal{e}_2 \]

MinML Summary
• Hopefully what you saw was completely unsurprising and boring
  – It will get less boring when you have to implement the semantics for homework in SML
• Now that we have a very simple base language we can explore extensions to the language

Extending MinML
• We will consider extensions to the MinML
  – Adding real numbers
  – Adding lists
  – Adding lists and real numbers
• We will examine the impact on the static and dynamic semantics as well as discuss design alternatives and issues for each extension

MinML with real
• This is easy right?
• Just add new operators or *overload* existing operators
• Automatic or explicit conversion between real and int?
• Examine simplest design first than consider more complicated ones
A Simple Design for Reals

Just add new operators and constants

Approach OCaml (ML Variant) uses

\[
\begin{align*}
\text{Real Num's} & : m \equiv 0.0 \mid 0.5 \mid 3.145 \ldots \\
\text{Real Op's} & : \text{ro} \equiv \text{+} \mid \text{-} \mid \text{*} \mid \text{=} \mid \text{<} \\
\text{Types} & : \tau \equiv \ldots \mid \text{real} \\
\text{Expr's} & : e \equiv \ldots \mid \text{i2r(e)} \mid \text{r2i(e)}
\end{align*}
\]

Typing for Simple Reals

\[
\begin{align*}
\text{Typing for Simple Reals} & : \\
\Gamma & \vdash m : \text{real} \\
\Gamma & \vdash e_1 : \text{int} \\
\Gamma & \vdash e_2 : \text{real} \\
\Gamma & \vdash \text{i2r(e)} : \text{real} \\
\Gamma & \vdash \text{r2i(e)} : \text{int} \\
\end{align*}
\]

Step Function for Simple Reals

\[
\begin{align*}
\text{Step Function for Simple Reals} & : \\
\text{i2r(n)}(n) & \mapsto n.0 \\
\text{r2i(n,m)} & \mapsto n \\
\text{+}(m_1,m_2) & \mapsto m_1 + \text{real } m_2 \\
\end{align*}
\]

A Simple Design for Reals

- Straightforward no complications
- Type checking syntax directed
- Extensions to dynamic semantics
- Arguably clumsy from a user perspective
- New operators for every type
- Possibly O(n^2) coercions with more numeric types

Reals with Overloading

- Overload the existing operators
- Automatically promote integers to real but not the reverse
- Basically the Java design

\[
\begin{align*}
\text{Real Num's} & : m \equiv 0.0 \mid 0.5 \mid 3.145 \ldots \\
\text{Types} & : \tau \equiv \ldots \mid \text{real} \\
\text{Expr's} & : e \equiv \ldots \mid \text{trunc(e)}
\end{align*}
\]

Typing for Overloaded Reals

\[
\begin{align*}
\text{Typing for Overloaded Reals} & : \\
\Gamma & \vdash m : \text{real} \\
\Gamma & \vdash e_1 : \text{int} \\
\Gamma & \vdash e_2 : \text{real} \\
\Gamma & \vdash \text{trunc(e)} : \text{int} \\
\Gamma & \vdash +(e_1,e_2) : \text{real} \\
\Gamma & \vdash -(e_1,e_2) : \text{bool} \\
\end{align*}
\]
Step Fun. for Overloaded Reals

\[
\begin{align*}
n & \mapsto n.0 \\
\text{trunc}(n.m) & \mapsto n \\
+ (m_1,m_2) & \mapsto m_1 +_{\text{real}} m_2 \\
\vdots & \vdots
\end{align*}
\]

Reals with Overloading

- Many issues with our extensions
- Type checking not syntax directed
  - But can choose unambiguous order or rule applications to get “sensible” system
  - Bottom up type checking algorithm
- Dynamic semantics non-deterministic
  - Bigger problem here semantics is not clear when ints are promoted to reals
  - +\((1,2)\) \mapsto 3.0 and +\((1,2)\) \mapsto 3

Fixing Dynamic Semantics

- Rewrite semantics so that ints are promoted to reals only when used in context expecting a real
- Mirrors intuition of type-checking rules
- Suggests we must distinguish between reals and integers at runtime
  - Not a problem for an interpret but bigger problem for compiler
  - Messy number for rules!

Reals with Overloading (Fixed)

\[
\begin{align*}
\text{trunc}(n.m) & \mapsto n \\
+ (m_1,m_2) & \mapsto n_1.0 +_{\text{real}} m_2 \\
+ (m_1,m_2) & \mapsto m_1 +_{\text{real}} n_2.0 \\
\vdots & \vdots
\end{align*}
\]

Translation Semantics

- Avoid complicated dynamic semantics by using original simpler semantics of and type-directed translation of program
- Best of both worlds
  - User gets overloading
  - Compiler and proof hacker’s get simpler systems

Type Directed Translation

\[
\begin{align*}
\Gamma \vdash (m \Rightarrow m) &: \text{real} \\
\Gamma \vdash (n \Rightarrow n) &: \text{int} \\
\Gamma \vdash (e \Rightarrow e') &: \text{int} \\
\Gamma \vdash (\text{trunc}(e \Rightarrow e')) &: \text{real} \\
\Gamma \vdash (e_1,e_2) &: \text{real} \\
\Gamma \vdash (e_3,e_4) &: \text{real} \\
\Gamma \vdash (+(e_1,e_2) \Rightarrow +(e_3,e_4)) &: \text{real} \\
\Gamma \vdash (e_5,e_6) &: \text{int} \\
\Gamma \vdash (e_7,e_8) &: \text{int} \\
\Gamma \vdash +(e_5,e_6) \Rightarrow +(e_7,e_8)) &: \text{int} \\
\vdots & \vdots
\end{align*}
\]
MinML with (τ) list

Simple ML generic list with hd/tl

Types τ ::= ... | (τ) list

Expr's e ::= ...

| nil | cons(e₁, e₂)
| hd(e) | tl(e) | isNil(e)

Typing MinML with (τ) list

Γ ⊢ nil : (τ) list

Γ ⊢ e₁ : τ, Γ ⊢ e₂ : (τ) list

Γ ⊢ cons(e₁, e₂) : (τ) list

Γ ⊢ e : (τ) list

Γ ⊢ hd(e) : τ

Γ ⊢ tl(e) : (τ) list

Γ ⊢ e : (τ) list

Γ ⊢ isNil(e) : bool

Step Fun. for MinML with (τ) list

hd(cons(V₁, V₂)) → V₁

tl(cons(V₁, V₂)) → V₂

MinML with (τ) list and casel

Simple ML generic list with casel

Types τ ::= ... | (τ) list

Expr's e ::= ...

| nil | cons(e₁, e₂)
| casel(e) of
| nil → e₁
| or cons(x:τ, y) → e₂
| end

MinML with (τ) list

- Type checking not syntax directed
  - nil has many types
  - Can use weaker form of type inference to assign type to nil or raise error if it is not determinable
- casel version is less error prone and avoids messing up the dynamic semantics with corner cases

MinML with real and (τ) list

<table>
<thead>
<tr>
<th>Lists hd/tl</th>
<th>Lists casel</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reals</td>
<td></td>
</tr>
<tr>
<td>Separate Ops</td>
<td></td>
</tr>
<tr>
<td>Overloaded Ops</td>
<td></td>
</tr>
</tbody>
</table>
MinML with real and (τ) list

<table>
<thead>
<tr>
<th>Lists</th>
<th>hd/tl</th>
<th>Lists</th>
<th>casel</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reals Separate Ops</td>
<td>No unpleasant interactions</td>
<td>No unpleasant interactions</td>
<td></td>
</tr>
<tr>
<td>Reals Overloaded Ops</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

What’s Going On?

- In the real type system all the primitive leaf rules/axioms for expressions provided us one unique type
- Bottom up reading of the expression could resolve overloading ambiguities
- Type inference for “nil” violates that assumption breaking overloading

What can we do about it?

- Give up on either overloading or type-inference!
- Complain when our algorithm gets too confused
  - Choose sane defaults (e.g. SML + is by default (int * int) -> int)
  - Allow optional annotations to help checker and override defaults
  - Pragmatic engineering and design issues
- Adopt Haskell type classes which allow for polymorphic runtime overloading!

Lessons Learned

- We can easily specify languages at a high-level with formal techniques
- We can reason about the design and interaction of systems incrementally
- The formalizes are just an aid to force you to think about all the details that come up when trying to do good language design