CS301
Session 11

Agenda

- Discussion: midterm exam - take-home or in-class?
- Interlude: common type constructors
- Type soundness

Common type constructors

Things we could add to Impcore

- Array is a type constructor, not a single type
- We're familiar with other type constructors from the garden-variety programming languages we use all the time
- ...but now is a good time to analyze them in a language-independent way
- Our typing rules will assume just one type environment $\Gamma$
Three common type constructors

- (First-class) functions
- Products
- Sums

First-class functions

- Type constructor \( \rightarrow \)
  - Infix, two arguments: \( \tau_1 \rightarrow \tau_2 \)
  - Formation rule:
    \[
    \tau_1 \text{ and } \tau_2 \text{ are types} \\
    \tau_1 \rightarrow \tau_2 \text{ is a type}
    \]

Typing rules for functions

- Introduction
  \[
  \Gamma \{ x \mapsto \tau \} \vdash e : \tau' \\
  \Gamma \vdash \text{LAMBDA}(x : \tau, e) : \tau \rightarrow \tau'
  \]
- Elimination
  \[
  \Gamma \vdash e_1 : \tau \rightarrow \tau' \quad \Gamma \vdash e_2 : \tau \\
  \Gamma \vdash \text{APPLY}(e_1, e_2) : \tau'
  \]

Products (pairs)

- Constituent types need not be the same
- Various, "tuple", "struct", "record"
- Can be used to model objects (in the OO sense)
- Formation
  \[
  \tau_1 \text{ and } \tau_2 \text{ are types} \\
  \tau_1 \times \tau_2 \text{ is a type}
  \]
Typing rules for products

- Introduction
  \[ \Gamma ⊢ e_1 : τ_1 \quad \Gamma ⊢ e_2 : τ_2 \]
  \[ \Gamma ⊢ \text{PAIR}(e_1, e_2) : τ_1 \times τ_2 \]

- Elimination
  \[ \Gamma ⊢ e : τ_1 \times τ_2 \]
  \[ \Gamma ⊢ \text{fst}(e) : τ_1 \]

(And similarly for the second element)

An elegant elim rule

- Like a pattern match
  \[ \Gamma ⊢ e : τ_1 \times τ_2 \quad \Gamma\{x_1 \mapsto τ_1, x_2 \mapsto τ_2\} ⊢ e' : τ \]
  \[ \Gamma ⊢ \text{LETPAIR}(x_1, x_2, e, e') : τ \]

Generalizing pairs

- In ML and related languages pairs are generalized to records with named fields
- Your homework contains a similar problem about sum types
- Formation:
  \[ τ_1 \ldots τ_n \text{ are types} \]
  \[ \{\text{name}_1 : τ_1, \ldots, \text{name}_n : τ_n\} \text{ is a type} \]

Typing records

- Introduction
  \[ \{\text{name}_1 : τ_1, \ldots, \text{name}_n : τ_n\} \text{ is a type} \]
  \[ \Gamma ⊢ e_1 : τ_1, \ldots, \Gamma ⊢ e_n : τ_n \]
  \[ \Gamma ⊢ \text{RECORD}(\text{name}_1 = e_1, \ldots, \text{name}_n = e_n) : \{\text{name}_1 : τ_1, \ldots, \text{name}_n : τ_n\} \]

- Elimination
  \[ \Gamma ⊢ e : \{\text{name}_1 : τ_1, \ldots, \text{name}_n : τ_n\} \]
  \[ \text{name} = \text{name}_i, 1 \leq i \leq n \]
  \[ \Gamma ⊢ \text{GETFIELD}(\text{name}, e) : τ_i \]
More elegant elim rule

- Again like a pattern match

\[
\begin{align*}
\Gamma & \vdash e : \{\text{name}_1 : \tau_1, \ldots, \text{name}_n : \tau_n\} \\
\Gamma & \{x_1 \mapsto \tau_1, \ldots, x_n \mapsto \tau_n\} \vdash e' : \tau
\end{align*}
\]

\[
\Gamma \vdash \text{LETRECORD}(x_1, \ldots, x_n, e, e') : \tau
\]

Sum types

- A type that unions other types together
- Like C unions, but safer because you can always tell what's there
- Like simple ML datatypes (no recursion)
- Formation rule

\[
\begin{align*}
\tau_1 \text{ and } \tau_2 \text{ are types} \\
\tau_1 + \tau_2 \text{ is a type}
\end{align*}
\]

Typing rules for unions

- Introduction

\[
\begin{align*}
\Gamma & \vdash e : \tau_1 \quad \tau_2 \text{ is a type} \\
\Gamma & \vdash \text{LET}_{\tau_2}(e) : \tau_1 + \tau_2
\end{align*}
\]

\[
\begin{align*}
\Gamma & \vdash e : \tau_2 \quad \tau_1 \text{ is a type} \\
\Gamma & \vdash \text{RIGHT}_{\tau_1}(e) : \tau_1 + \tau_2
\end{align*}
\]

Typing rules for unions(2)

- Elimination: like case or switch

\[
\begin{align*}
\Gamma & \vdash e : \tau_1 + \tau_2 \\
\Gamma & \{x_1 \mapsto \tau_1\} \vdash e_1 : \tau \\
\Gamma & \{x_2 \mapsto \tau_2\} \vdash e_2 : \tau
\end{align*}
\]

\[
\Gamma \vdash \text{case } e \text{ of LEFT}(x_1) \Rightarrow e_1 | \RIGHT(x_2) \Rightarrow e_2 : \tau
\]
About type soundness

Why trust a type system?

- Given a complex enough type system, we might be unable to see whether it behaves reasonably
- Language designers prove type soundness both to increase trust and to be explicit about what guarantees the type system provides

What is type soundness?

- A kind of claim we make about the relationship between the typing rules and the evaluation rules
- Loosely, "well-typed programs don't go wrong"
- Sample corollaries:
  - Functions always receive the right number and kind of arguments
  - No array access is out of bounds (a more advanced kind of type system)

Machinery needed for soundness

- The meaning of a type $\llbracket \tau \rrbracket$ is a set of values
- Examples
  - $\llbracket \text{INT} \rrbracket = \{ \text{NUMBER}(n) \mid n \text{ is an integer} \}$
  - $\llbracket \text{BOOL} \rrbracket = \{ \text{BOOL}(\#t), \text{BOOL}(\#f) \}$
- This gives us a notation for the set of things a well typed expression is allowed to evaluate to
Proper environments

If $\Gamma$ and $\rho$ are typing and value environments, respectively, we say $\rho$ agrees with $\Gamma$ whenever, for every $x$ in $\text{dom}(\Gamma)$,

1. $x$ is also in $\text{dom}(\rho)$, and
2. $\rho(x) \in \llbracket \Gamma(x) \rrbracket$

A soundness claim

If

1. $\Gamma$ and $\rho$ are typing and value environments, and
2. $\rho$ agrees with $\Gamma$, and
3. $\Gamma \vdash e : \tau$ and $\langle \rho, e \rangle \Downarrow v$,

then $v \in \llbracket \tau \rrbracket$
CS301
Session 12

Agenda

- Side trip: the semantics of defining and applying recursive functions
- Introduction to polymorphic type systems
- A polymorphic type system for uScheme

Recursive functions
How can recursion work?

- Rule: all names are evaluated by looking them up in an environment
- How do we arrange for the name $f$ to be meaningful in:

\[
\text{(define f (n e) }
\begin{align*}
&\text{ (if (= e 0) 1 } \\
&\text{ (* n (f n (- e 1))))}
\end{align*}
\]

Simple case: Impcore

- Functions are not first-class; special function environment
- We just bind the function name to a piece of abstract syntax

\[
\langle \text{DEFINE}(f, (x_1, \ldots, x_n), e), \xi, \phi \rangle \rightarrow \langle \xi, \phi \{ f \mapsto \text{USER}(\langle x_1, \ldots, x_n \rangle, e) \} \rangle
\]

Impcore function application

- By the time we use a recursive function, its definition is already in the function environment

\[
\phi(f) = \text{USER}(\langle x_1, \ldots, x_n \rangle, e)
\]

\[
\begin{align*}
&x_1, \ldots, x_n \text{ all distinct} \\
&\langle e_1, \xi_0, \phi, \rho_0 \rangle \Downarrow \langle v_1, \xi_1, \phi, \rho_1 \rangle \\
&\vdots \\
&\langle e_n, \xi_n-1, \phi, \rho_n-1 \rangle \Downarrow \langle v_n, \xi_n, \phi, \rho_n \rangle \\
&\langle e, \xi_n, \phi, \{ x_1 \mapsto v_1, \ldots, x_n \mapsto v_n \} \rangle \Downarrow \langle v, \xi', \phi, \rho_n \rangle \\
&\langle \text{APPLY}(f, e_1, e_2, \ldots, e_n), \xi_0, \phi, \rho_0 \rangle \Downarrow \langle v, \xi', \phi, \rho_n \rangle
\end{align*}
\]

First-class functions

- What about uScheme? How do we make sure the name of the recursive function is properly bound in the body?
Functions

- Lambdas evaluate to closures

\[
\begin{align*}
\text{functions} & : l_1, \ldots, l_n \not\in \text{dom } \sigma \\
\rho' & = \rho \{ x_1 \mapsto l_1, \ldots, x_n \mapsto l_n \} \\
\sigma_0 & = \sigma \{ l_1 \mapsto \text{unspecified}, \ldots, l_n \mapsto \text{unspecified} \} \\
\langle e_1, \rho', \sigma_0 \rangle & \Downarrow \langle v_1, \sigma_1 \rangle \\
\vdots \\
\langle e_n, \rho', \sigma_{n-1} \rangle & \Downarrow \langle v_n, \sigma_n \rangle \\
\langle e, \rho', \sigma_n \{ l_1 \mapsto v_1, \ldots, l_n \mapsto v_n \} \rangle & \Downarrow \langle v, \sigma' \rangle \\
\langle \text{LETREC}((x_1, e_1, \ldots, x_n, e_n), e), \rho, \sigma \rangle & \Downarrow \langle v, \sigma' \rangle
\end{align*}
\]

Local recursive definition

- When a recursive function is applied, how/where is its name bound?

\[
\begin{align*}
l_1, \ldots, l_n & \not\in \text{dom } \sigma \\
\rho' & = \rho \{ x_1 \mapsto l_1, \ldots, x_n \mapsto l_n \} \\
\sigma_0 & = \sigma \{ l_1 \mapsto \text{unspecified}, \ldots, l_n \mapsto \text{unspecified} \} \\
\langle e_1, \rho', \sigma_0 \rangle & \Downarrow \langle v_1, \sigma_1 \rangle \\
\vdots \\
\langle e_n, \rho', \sigma_{n-1} \rangle & \Downarrow \langle v_n, \sigma_n \rangle \\
\langle e, \rho', \sigma_n \{ l_1 \mapsto v_1, \ldots, l_n \mapsto v_n \} \rangle & \Downarrow \langle v, \sigma' \rangle \\
\langle \text{LETREC}((x_1, e_1, \ldots, x_n, e_n), e), \rho, \sigma \rangle & \Downarrow \langle v, \sigma' \rangle
\end{align*}
\]

Top-level recursive definitions

- Left as an exercise...do it!
Polymorphic type systems

Perspective
- Flexibility of dynamic typing (Scheme) both a blessing and a curse
- Great for small systems, prototypes, and god-like programmers
- Not so great for large systems, production code, trusted code, teams of ordinary mortals

Limitations of monomorphic typing
- Example from typed Impcore: list processing functions

Where we're going
- Introduce polymorphic type system with static type checking
- Now we can write one version of length with type
  \[ \forall \alpha. \alpha \text{ list} \rightarrow \text{int} \]
  \[(\text{forall } (\text{'a}) (\text{function } ((\text{list } \text{'a})) \text{ int}))\]
- This will be flexible enough to type a lot of the programs we want - almost a "sweet spot"
- ...but terribly verbose and impossible to use
Why?

- Why torture ourselves with this type system?
- To motivate type inference as in ML and related languages
- The real "sweet spot": polymorphic type system, plus type inference, yields a terse, flexible language with robust guarantees suitable for production programming
- Used in ML, OCaml, Haskell, etc. etc.

Type variables

- A new kind of variable that stands for an unknown type
- Actual types are supplied by type instantiation, a.k.a. type application
- Type variables are bound in types by ∀(abstractly), or forall (concretely)
- Bound in expressions by TYPAMBDA (abstractly), or type-lambda (concretely)

Idea: lambda for types

- You've seen this before: Java/C++ generics
- Quantified types: ∀α1, ..., αn . τ
  (forall ('a1 ... 'an) type)
- Type abstraction: TYPAMBDA(α1, ..., αn, e)
  (type-lambda ('a1 ... 'an) exp)
- Type application: TYAPPLY(e, τ1, ..., τn)
  (@ exp typel ... typen)

Quantified types

-> length
<procedure> : (forall ('a) (function ((list 'a)) int))
-> cons
<procedure> : (forall ('a) (function ('a (list 'a)) (list 'a)))
-> car
<procedure> : (forall ('a) (function ((list 'a)) 'a))
-> cdr
<procedure> : (forall ('a) (function ((list 'a)) (list 'a)))
-> '()' () : (forall ('a) (list 'a))
Type instantiation

-> (val length-int (@ length int))
length-int : (function ((list int)) int)

-> (val length-bool (@ length bool))
length-bool : (function ((list bool)) int)

-> (val nil-bool (@ '() bool))
() : (list bool)

Instantiation substitutes actual types for type variables

Type abstraction

-> (val-rec (forall ('a) (function ((list 'a)) int))
     len (type-lambda ('a)
       (lambda (((list 'a) l))
         (if ((@ null? 'a) l) 0
          (+ 1 ((@ len 'a) ((@ cdr 'a) l))))))
     len : (forall ('a) (function ((list 'a)) int))
-> (@ len int)
<procedure> : (function ((list int)) int)
-> ((@ len int) '(1 2 3))
3 : int