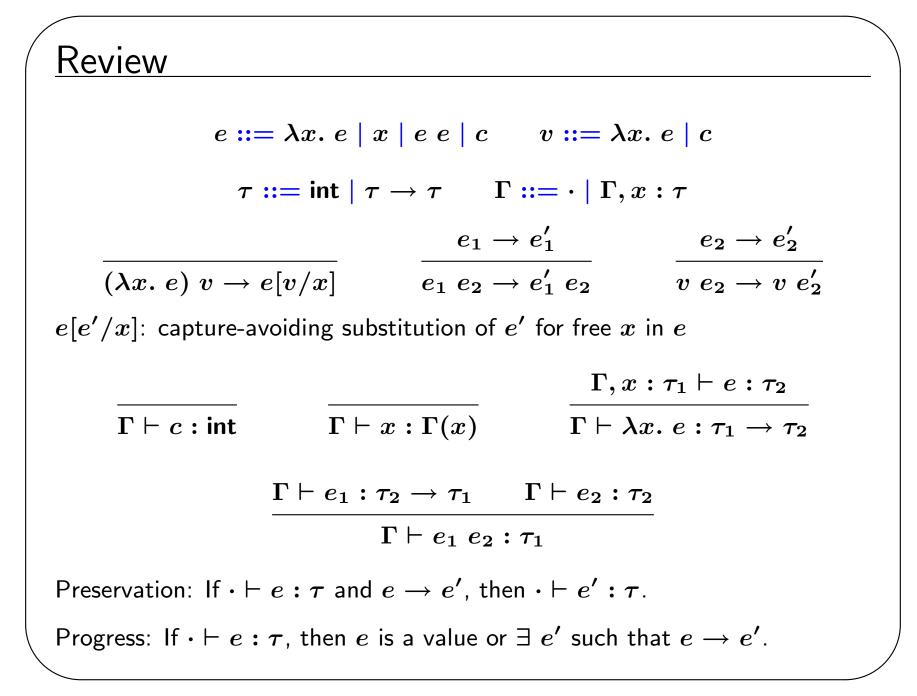
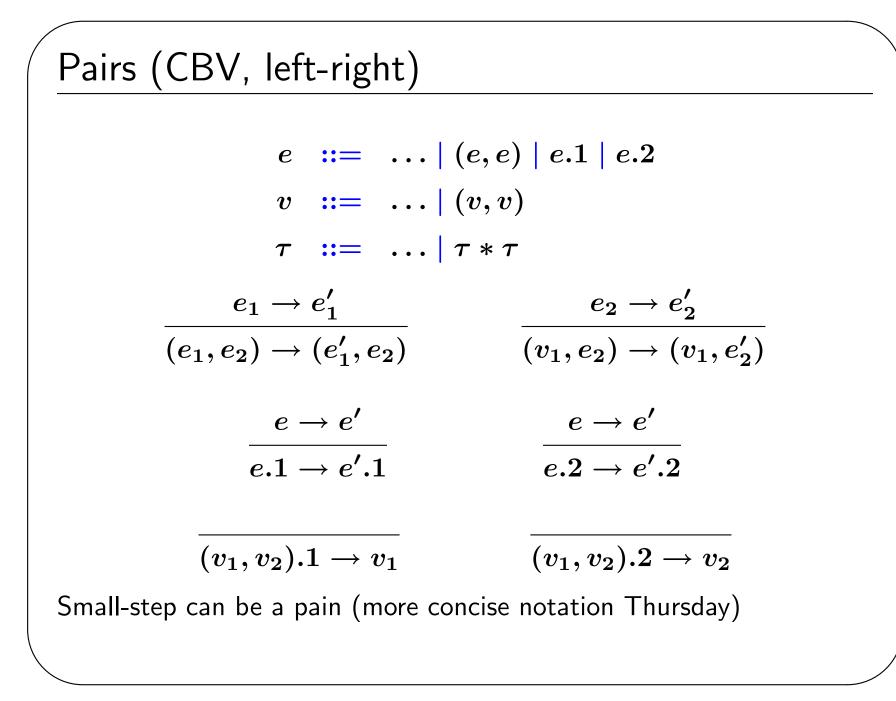
CSE 505: Concepts of Programming Languages

Dan Grossman Fall 2005 Lecture 9— More ST λ C Extensions; Notes on Termination

<u>Outline</u>

- Continue extending ST λ C data structures, recursion
- Discussion of "anonymous" types
- Consider termination informally
- Next time: Curry-Howard Isomorphism, Evaluation Contexts, Abstract Machines







 $rac{\Gammadashearrow e_1: au_1 \qquad \Gammadashearrow e_2: au_2}{\Gammadashaarrow (e_1,e_2): au_1* au_2}$

$\Gamma dash e: au_1 * au_2$	$\Gamma \vdash e: \tau_1 \ast \tau_2$
$\Gamma \vdash e.1:\tau_1$	$\Gamma \vdash e.2:\tau_2$

Canonical Forms: If $\cdot \vdash v : \tau_1 * \tau_2$, then v has the form (v_1, v_2) .

Progress: New cases using C.F. are v.1 and v.2.

Preservation: For primitive reductions, inversion gives the result *directly*.

<u>Records</u>

Records seem like pairs with named fields

Fields do *not* α -convert.

Names might let us reorder fields, e.g.,

 $\cdot \vdash \{l_1 = 42; l_2 = \mathsf{true}\} : \{l_2 : \mathsf{bool}; l_1 : \mathsf{int}\}.$

Nothing wrong with this, but many languages disallow it. (Why? Run-time efficiency and/or type inference)

(Caml has only named record types with disjoint fields.)

More on this when we study *subtyping*

<u>Sums</u>

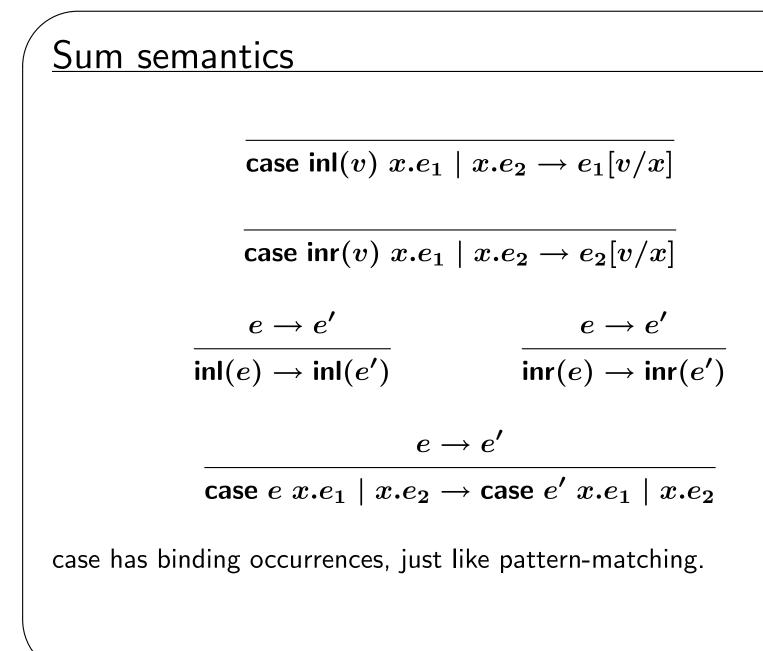
What about ML-style datatypes:

type t = A | B of int | C of int*t

- 1. Tagged variants (i.e., discriminated unions)
- 2. Recursive types
- 3. Type constructors (e.g., type 'a mylist = ...)
- 4. Names the type

Today we'll model just (1) with (anonymous) sum types:

 $e ::= \dots | \operatorname{inl}(e) | \operatorname{inr}(e) | \operatorname{case} e x.e | x.e$ $v ::= \dots | \operatorname{inl}(v) | \operatorname{inr}(v)$ $\tau ::= \dots | \tau_1 + \tau_2$



Sum Type-checkingInference version (not trivial to infer; can require annotations)
$$\Gamma \vdash e : \tau_1$$
 $\Gamma \vdash e : \tau_1$ $\Gamma \vdash inl(e) : \tau_1 + \tau_2$ $\Gamma \vdash e : \tau_1 + \tau_2$ $\Gamma \vdash e : \tau_1 + \tau_2$ $\Gamma \vdash e : \tau_1 + \tau_2$ $\Gamma \vdash case \ e \ x.e_1 \mid x.e_2 : \tau$ C.F.: If $\cdot \vdash v : \tau_1 + \tau_2$, then either v has the form $inl(v_1)$ and $\cdot \vdash v_1 : \tau_1$ or ...The rest is induction and substitution.Can encode booleans with sums. E.g., bool = int + int, true = inl(0), false = inr(0).

Base Types, in general

What about floats, strings, enums, ...? Could add them all or do something more general...

Parameterize our language/semantics by a collection of *base types* (b_1, \ldots, b_n) and *primitives* $(c_1 : \tau_1, \ldots, c_n : \tau_n)$.

```
Examples: concat : string\rightarrowstring\rightarrowstring
toInt : float\rightarrowint
```

```
"hello" : string
```

For each primitive, *assume* if applied to values of the right types it produces a value of the right type.

Together the types and assumed steps tell us how to type-check and evaluate $c_i v_1 \dots v_n$ where c_i is a primitive.

We can prove soundness once and for all given the assumptions.

Recursion

We won't prove it, but every extension so far preserves termination. A Turing-complete language needs some sort of loop. What we add won't be encodable in $ST\lambda C$.

```
E.g., let rec f x = e
```

Do typed recursive functions need to be bound to variables or can they be anonymous?

In Caml, you need variables, but it's unnecessary:

$$e := \dots | \text{fix } e$$

$$rac{e
ightarrow e'}{\operatorname{\mathsf{fix}} e
ightarrow \operatorname{\mathsf{fix}} e'} \qquad \qquad rac{\operatorname{\mathsf{fix}} \lambda x. \ e
ightarrow e[\operatorname{\mathsf{fix}} \lambda x. \ e/x]}{\operatorname{\mathsf{fix}} \lambda x. \ e
ightarrow e[\operatorname{\mathsf{fix}} \lambda x. \ e/x]}$$

Using fix

It works just like let rec, e.g.,

fix λf . λn . if n < 1 then 1 else n * (f(n-1))

Note: You can use it for mutual recursion too.

Pseudo-math digression

Why is it called fix? In math, a fixed-point of a function g is an x such that g(x) = x.

Let g be λf . λn . if n < 1 then 1 else n * (f(n-1)).

If g is applied to a function that computes factorial for arguments $\leq m$, then g returns a function that computes factorial for arguments $\leq m+1$.

Now g has type (int \rightarrow int) \rightarrow (int \rightarrow int). The fix-point of g is the function that computes factorial for *all* natural numbers.

And fix g is equivalent to that function. That is, fix g is the fix-point of g.

Typing fix

 $\frac{\Gamma \vdash e: \tau \to \tau}{\Gamma \vdash \mathsf{fix} \; e: \tau}$

Math explanation: If e is a function from τ to τ , then fix e, the fixed-point of e, is some τ with the fixed-point property. So it's something with type τ .

Operational explanation: fix λx . e' becomes $e'[\text{fix } \lambda x$. e'/x]. The substitution means x and fix λx . e' better have the same type. And the result means e' and fix λx . e' better have the same type.

Note: Proving soundness is straightforward!

General approach

We added lets, booleans, pairs, records, sums, and fix. Let was syntactic sugar. Fix made us Turing-complete by "baking in" self-application. The others *added types*.

Whenever we add a new form of type au there are:

- Introduction forms (ways to make values of type au)
- Elimination forms (ways to use values of type au)

What are these forms for functions? Pairs? Sums?

When you add a new type, think "what are the intro and elim forms"?

Anonymity

We added many forms of types, all *unnamed* a.k.a. *structural*.

Many real PLs have (all or mostly) *named* types:

- Java, C, C++: all record types (or similar) have names (omitting them just means compiler makes up a name)
- Caml sum-types have names.

A never-ending debate:

- Structual types allow more code reuse, which is good.
- Named types allow less code reuse, which is good.
- Structural types allow generic type-based code, which is good.
- Named types allow type-based code to distinguish names, which is good.

The theory is often easier and simpler with structural types.

Termination

Surprising fact: If $\cdot \vdash e : \tau$ in the ST λ C with all our additions *except* fix, then there exists a v such that $e \rightarrow^* v$.

That is, all programs terminate.

So termination is trivially decidable (the constant "yes" function), so our language is not Turing-complete.

Proof is in the book. It requires cleverness because the size of expressions does *not* "go down" as programs run.

Non-proof: Recursion in λ calculus requires some sort of self-application. Easy fact: For all Γ , x, and τ , we *cannot* derive $\Gamma \vdash x \ x : \tau$.