

CIS 500

Software Foundations

Fall 2002

4 November

## Administrivia

---

- ◆ Reminder: Prof. Pierce out of town Nov. 5 - 14
  - ◆ No office hours Nov 5, 7, 12, or 14
  - ◆ 3PM recitation cancelled on Nov 11 - go to Max's in Towne 307 instead
- ◆ Next Wednesday: Midterm II
  - ◆ Covering Chapters 1-16 (concentrating on 9-16), except 12 and 15.6.
  - ◆ There **will** be a question about the proof of type safety for the simply typed lambda-calculus with references. Make sure you understand it completely.
  - ◆ In general, the questions on the second midterm will be somewhat harder/deeper than the first. It will also be somewhat shorter.

## Subtyping and Lists

---

$$\frac{S_1 <: T_1}{\text{List } S_1 <: \text{List } T_1} \quad (\text{S-LIST})$$

i.e., `List` is a covariant type constructor.

## Subtyping and References

---

$$\frac{S_1 <: T_1 \quad T_1 <: S_1}{\text{Ref } S_1 <: \text{Ref } T_1} \quad (\text{S-REF})$$

I.e., `Ref` is **not** a covariant (nor a contravariant) type constructor.

## References again

---

Observation: a value of type `Ref T` can be used in two different ways: as a **source** for values of type `T` and as a **sink** for values of type `T`.

## References again

---

Observation: a value of type `Ref T` can be used in two different ways: as a **source** for values of type `T` and as a **sink** for values of type `T`.

Idea: Split `Ref T` into three types:

- ◆ `Source T`: reference cell with “read capability”
- ◆ `Sink T`: reference cell with “write capability”
- ◆ `Ref T`: cell with both capabilities

## Modified Typing Rules

---

$$\frac{\Gamma \mid \Sigma \vdash t_1 : \text{Source } T_{11}}{\Gamma \mid \Sigma \vdash !t_1 : T_{11}}$$

(T-DEREF)

$$\frac{\Gamma \mid \Sigma \vdash t_1 : \text{Sink } T_{11} \quad \Gamma \mid \Sigma \vdash t_2 : T_{11}}{\Gamma \mid \Sigma \vdash t_1 := t_2 : \text{Unit}}$$

(T-ASSIGN)

# Subtyping rules

---

$$\frac{S_1 <: T_1}{\text{Source } S_1 <: \text{Source } T_1} \quad (\text{S-SOURCE})$$
$$\frac{T_1 <: S_1}{\text{Sink } S_1 <: \text{Sink } T_1} \quad (\text{S-SINK})$$
$$\text{Ref } T_1 <: \text{Source } T_1 \quad (\text{S-REFSOURCE})$$
$$\text{Ref } T_1 <: \text{Sink } T_1 \quad (\text{S-REFSINK})$$



# Capabilities

---

Other kinds of capabilities (e.g., send and receive capabilities on communication channels, encrypt/decrypt capabilities of cryptographic keys, ...) can be treated similarly.

# Coercion semantics

---

[skip]

## Intersection Types

The inhabitants of  $T_1 \wedge T_2$  are terms belonging to **both**  $S$  and  $T$ —i.e.,  $T_1 \wedge T_2$  is an order-theoretic **meet** (greatest lower bound) of  $T_1$  and  $T_2$ .

$$T_1 \wedge T_2 <: T_1 \quad (\text{S-INTER1})$$

$$T_1 \wedge T_2 <: T_2 \quad (\text{S-INTER2})$$

$$\frac{S <: T_1 \quad S <: T_2}{S <: T_1 \wedge T_2} \quad (\text{S-INTER3})$$

$$S \rightarrow T_1 \wedge S \rightarrow T_2 <: S \rightarrow (T_1 \wedge T_2) \quad (\text{S-INTER4})$$

# Intersection Types

---

Intersection types permit a very flexible form of **finitary overloading**.

$+ : (\text{Nat} \rightarrow \text{Nat} \rightarrow \text{Nat}) \wedge (\text{Float} \rightarrow \text{Float} \rightarrow \text{Float})$

This form of overloading is **extremely powerful**.

Every strongly normalizing untyped lambda-term can be typed in the simply typed lambda-calculus with intersection types.

→ type reconstruction problem is undecidable

Intersection types have not been used much in language designs (too powerful!), but are being intensively investigated as type systems for **intermediate languages** in highly optimizing compilers (cf. Church project).

# Union types

---

Union types are also useful.

$T_1 \vee T_2$  is an **untagged** (non-disjoint) union of  $T_1$  and  $T_2$

No tags  $\rightarrow$  no **case** construct. The only operations we can safely perform on elements of  $T_1 \vee T_2$  are ones that make sense for **both**  $T_1$  and  $T_2$ .

N.b.: untagged **union** types in C are a source of type safety violations precisely because they ignore this restriction, allowing any operation on an element of  $T_1 \vee T_2$  that makes sense for **either**  $T_1$  or  $T_2$ .

Union types are being used recently in type systems for XML processing languages (cf. XDuce, Xtatic).

# Metatheory of Subtyping (Preview)

## Syntax-directed rules

In the simply typed lambda-calculus (without subtyping), each rule can be “read from bottom to top” in a straightforward way.

$$\frac{\Gamma \vdash t_1 : T_{11} \rightarrow T_{12} \quad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1 \ t_2 : T_{12}} \quad (\text{T-APP})$$

If we are given some  $\Gamma$  and some  $t$  of the form  $t_1 \ t_2$ , we can try to find a type for  $t$  by

1. finding (recursively) a type for  $t_1$
2. checking that it has the form  $T_{11} \rightarrow T_{12}$
3. finding (recursively) a type for  $t_2$
4. checking that it is the same as  $T_{11}$

Technically, the reason this works is that We can divide the “positions” of the typing relation into **input positions** ( $\Gamma$  and  $t$ ) and **output positions** ( $T$ ).

- ◆ For the input positions, all metavariables appearing in the premises also appear in the conclusion (so we can calculate inputs to the “subgoals” from the subexpressions of inputs to the main goal)
- ◆ For the output positions, all metavariables appearing in the conclusions also appear in the premises (so we can calculate outputs from the main goal from the outputs of the subgoals)

$$\frac{\Gamma \vdash t_1 : T_{11} \rightarrow T_{12} \quad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1 \ t_2 : T_{12}} \quad (\text{T-APP})$$



## Syntax-directed sets of rules

---

The second important point about the simply typed lambda-calculus is that the **set** of typing rules is syntax-directed, in the sense that, for every “input”  $\Gamma$  and  $t$ , there one rule that can be used to derive typing statements involving  $t$ .

E.g., if  $t$  is an application, then we must proceed by trying to use T-APP. If we succeed, then we have found a type (indeed, the unique type) for  $t$ . If it fails, then we know that  $t$  is not typable.

→ no backtracking!

## Non-syntax-directedness of typing

When we extend the system with subtyping, both aspects of syntax-directedness get broken.

1. The set of typing rules now includes **two** rules that can be used to give a type to terms of a given shape (the old one plus T-SUB)

$$\frac{\Gamma \vdash t : S \quad S <: T}{\Gamma \vdash t : T} \quad (\text{T-SUB})$$

2. Worse yet, the new rule T-SUB itself is not syntax directed: the inputs to the left-hand subgoal are exactly the same as the inputs to the main goal!

(Hence, if we translated the typing rules naively into a typechecking function, the case corresponding to T-SUB would cause divergence.)

## Non-syntax-directedness of subtyping

Moreover, the subtyping relation is not syntax directed either.

1. There are **lots** of ways to derive a given subtyping statement.
2. The transitivity rule

$$\frac{S <: U \quad U <: T}{S <: T} \quad (\text{S-TRANS})$$

is badly non-syntax-directed: the premises contain a metavariable (in an “input position”) that does not appear at all in the conclusion.

To implement this rule naively, we’d have to **guess** a value for **U**!

# What to do?

---

## What to do?

---

1. Observation: We don't **need** 1000 ways to prove a given typing or subtyping statement — one is enough.  
→ Think more carefully about the typing and subtyping systems to see where we can get rid of “excess flexibility”
2. Use the resulting intuitions to formulate new “algorithmic” (i.e., syntax-directed) typing and subtyping relations
3. Check (i.e., prove) that the algorithmic relations are “the same as” the original ones in an appropriate sense.

## What to do?

---

1. Observation: We don't **need** 1000 ways to prove a given typing or subtyping statement — one is enough.  
→ Think more carefully about the typing and subtyping systems to see where we can get rid of “excess flexibility”
2. Use the resulting intuitions to formulate new “algorithmic” (i.e., syntax-directed) typing and subtyping relations
3. Check (i.e., prove) that the algorithmic relations are “the same as” the original ones in an appropriate sense.

Details: next time.

## Exceptions (Chapter 14)

## Motivation

---

Most programming languages provide some mechanism for interrupting the normal flow of control in a program to signal some exceptional condition.

Note that it is always **possible** to program without exceptions — instead of raising an exception, we return `None`; instead of returning result `x` normally, we return `∃(x)`. But now we need to wrap every function application in a `case` to find out whether it returned a result or an exception.

→ much more convenient to build this mechanism into the language.



## Varieties of non-local control

---

There are **many** ways of adding “non-local control flow”

- ◆ `exit(1)`
- ◆ `goto`
- ◆ `setjmp/longjmp`
- ◆ `raise/try` (or `catch/throw`) in many variations
- ◆ `callcc` / continuations
- ◆ more esoteric variants (cf. many Scheme papers)

## Varieties of non-local control

---

There are **many** ways of adding “non-local control flow”

- ◆ `exit(1)`
- ◆ `goto`
- ◆ `setjmp/longjmp`
- ◆ `raise/try` (or `catch/throw`) in many variations
- ◆ `callcc` / continuations
- ◆ more esoteric variants (cf. many Scheme papers)

Let's begin with the simplest of these.

## An “abort” primitive

First step: raising exceptions (but not catching them).

$t ::= \dots$	terms
$\text{error}$	run-time error

Evaluation

$\text{error } t_2 \longrightarrow \text{error}$  (E-APPERR1)

$v_1 \text{ error} \longrightarrow \text{error}$  (E-APPERR2)

Typing

$\Gamma \vdash \text{error} : T$  (T-ERROR)

## Typing errors

---

Note that the typing rule for `error` allows us to give it *any* type `T`.

$$\Gamma \vdash \text{error} : T \quad (\text{T-ERROR})$$

This means that both

```
if x>0 then 5 else error
```

and

```
if x>0 then true else error
```

will typecheck.

## Syntax-directedness

---

However this rule

$$\Gamma \vdash \text{error} : T \quad (\text{T-ERROR})$$

has a problem from the point of view of implementation: it is not syntax-directed!

## An alternative typing rule

---

In a system with subtyping and a minimal `Bot` type, we can give `error` a better typing:

$$\Gamma \vdash \text{error} : \text{Bot} \quad (\text{T-ERROR})$$

(Of course, what we've really done is just pushed the complexity of the old `error` rule onto the `Bot` type! We'll return to this point later.)

## Type safety

---

The **preservation** theorem requires no changes when we add **error**: if a term of type **T** reduces to **error**, that's fine, since **error** has every type **T**.

## Type safety

---

The **preservation** theorem requires no changes when we add **error**: if a term of type **T** reduces to **error**, that's fine, since **error** has every type **T**.

Progress, though, requires a little more care.



## Progress

First, note that we do **not** want to extend the set of values to include **error**, since this would make our new rule for propagating errors through applications.

$$v_1 \text{ error} \longrightarrow \text{error} \quad (\text{E-APPERR2})$$

overlap with our existing computation rule for applications:

$$(\lambda x:T_{11}.t_{12}) \ v_2 \longrightarrow [x \mapsto v_2]t_{12} \quad (\text{E-APPABS})$$

e.g., the term

$(\lambda x:\text{Nat}.0) \text{ error@}$

might evaluate to either **0** (which would be wrong) or **error** (what we want).

## Progress

---

Instead, we keep `error` as a non-value normal form, and refine the statement of progress to explicitly mention the possibility that terms may evaluate to `error` instead of to a value.

THEOREM [PROGRESS]: Suppose `t` is a closed, well-typed normal form. Then either `t` is a value or `t = error`.

# Catching exceptions

$t ::= \dots$

terms

$\text{try } t \text{ with } t$

trap errors

## Evaluation

$\text{try } v_1 \text{ with } t_2 \longrightarrow v_1$

(E-TRYV)

$\text{try error with } t_2$

$\longrightarrow t_2$

(E-TRYERROR)

$t_1 \longrightarrow t'_1$

$\text{try } t_1 \text{ with } t_2$

$\longrightarrow \text{try } t'_1 \text{ with } t_2$

(E-TRY)

## Typing

$$\frac{\Gamma \vdash t_1 : T \quad \Gamma \vdash t_2 : T}{\Gamma \vdash \text{try } t_1 \text{ with } t_2 : T} \quad (\text{T-TRY})$$

# Exceptions carrying values

$t ::= \dots$   
 $\text{raise } t$

terms  
 $\text{raise exception}$

## Evaluation

$(\text{raise } v_{11}) \ t_2 \longrightarrow \text{raise } v_{11}$  (E-APPRaise1)

$v_1 \ (\text{raise } v_{21}) \longrightarrow \text{raise } v_{21}$  (E-APPRaise2)

$$\frac{t_1 \longrightarrow t'_1}{\text{raise } t_1 \longrightarrow \text{raise } t'_1}$$
 (E-RAISE)

$$\begin{aligned} &\text{raise } (\text{raise } v_{11}) \\ &\longrightarrow \text{raise } v_{11} \end{aligned}$$
 (E-RAISERAISE)

$\text{try } v_1 \text{ with } t_2 \longrightarrow v_1$

(E-TRYV)

$\text{try raise } v_{11} \text{ with } t_2$   
 $\longrightarrow t_2 \ v_{11}$

(E-TRYRAISE)

$$\frac{t_1 \longrightarrow t'_1}{\text{try } t_1 \text{ with } t_2 \longrightarrow \text{try } t'_1 \text{ with } t_2}$$

(E-TRY)

# Typing

---

$$\frac{\Gamma \vdash t_1 : T_{\text{exn}}}{\Gamma \vdash \text{raise } t_1 : T} \quad (\text{T-EXN})$$

$$\frac{\Gamma \vdash t_1 : T \quad \Gamma \vdash t_2 : T_{\text{exn}} \rightarrow T}{\Gamma \vdash \text{try } t_1 \text{ with } t_2 : T} \quad (\text{T-TRY})$$