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

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

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

Subtyping and References

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

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 cabability"
- `Sink T`: reference cell with "write cabability"
- `Ref T`: cell with both capabilities

Modified Typing Rules

$$\frac{\Gamma \mid \Sigma \vdash \mathtt{t_1} : \mathtt{Source}\ \mathtt{T_{11}}}{\Gamma \mid \Sigma \vdash \mathtt{!t_1} : \mathtt{T_{11}}} \quad \text{(T-Deref)}$$

$$\frac{\Gamma \mid \Sigma \vdash \mathtt{t_1} : \mathtt{Sink}\ \mathtt{T_{11}} \qquad \Gamma \mid \Sigma \vdash \mathtt{t_2} : \mathtt{T_{11}}}{\Gamma \mid \Sigma \vdash \mathtt{t_1}\mathtt{:=}\mathtt{t_2} : \mathtt{Unit}} \quad \text{(T-Assign)}$$

Subtyping rules

$$\frac{\mathtt{S_1} <: \mathtt{T_1}}{\mathtt{Source}\ \mathtt{S_1} <: \mathtt{Source}\ \mathtt{T_1}} \quad \text{(S-Source)}$$

$$\frac{\mathtt{T_1} <: \mathtt{S_1}}{\mathtt{Sink}\ \mathtt{S_1} <: \mathtt{Sink}\ \mathtt{T_1}} \quad \text{(S-Sink)}$$

$$\mathtt{Ref}\ \mathtt{T_1} <: \mathtt{Source}\ \mathtt{T_1} \quad \text{(S-RefSource)}$$

$$\mathtt{Ref}\ \mathtt{T_1} <: \mathtt{Sink}\ \mathtt{T_1} \quad \text{(S-RefSink)}$$

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.

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

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

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

$$S \to T_1 \wedge S \to T_2 <: S \to (T_1 \wedge T_2) \qquad \text{(S-INTER4)}$$

Intersection Types

Intersection types permit a very flexible form of finitary overloading.

$+ : (\mathtt{Nat} \to \mathtt{Nat} \to \mathtt{Nat}) \wedge (\mathtt{Float} \to \mathtt{Float} \to \mathtt{Float})$

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.
>
> $\longrightarrow$ 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 $\longrightarrow$ no `case` construct. The only operations we can safely perform on elements of $T_1 \backslash/ T2$ 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 ignores 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.

$$\frac{\Gamma \vdash t_1 : T_{11} \to T_{12} \qquad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1\ t_2 : T_{12}} \quad \text{(T-App)}$$

If we are given some Γ 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 $T_{11} \to T_{12}$
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 (Γ 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)

$$\frac{\Gamma \vdash t_1 : T_{11} \to T_{12} \qquad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1\ t_2 : T_{12}} \quad \text{(T-App)}$$

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" Γ 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)

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

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

$$\frac{S <: U \qquad U <: T}{S <: T} \qquad \text{(S-Trans)}$$

   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 ::=	...	terms
	error	run-time error

Evaluation

$$\texttt{error}\ t_2 \longrightarrow \texttt{error} \qquad \text{(E-AppErr1)}$$

$$v_1\ \texttt{error} \longrightarrow \texttt{error} \qquad \text{(E-AppErr2)}$$

Typing

$$\Gamma \vdash \texttt{error} : T \qquad \text{(T-Error)}$$

Typing errors

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

$$\Gamma \vdash \texttt{error} : T \qquad \text{(T-Error)}$$

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

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

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:

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

(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 litte 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.

$$\texttt{v}_1\ \texttt{error} \longrightarrow \texttt{error} \qquad \text{(E-AppErr2)}$$

overlap with our existing computation rule for applications:

$$(\lambda\texttt{x}:\texttt{T}_{11}.\texttt{t}_{12})\ \texttt{v}_2 \longrightarrow [\texttt{x} \mapsto \texttt{v}_2]\texttt{t}_{12} \qquad \text{(E-AppAbs)}$$

e.g., the term

`(λx:Nat.0) 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	::=	...	terms
		try t with t	trap errors

Evaluation

$$\texttt{try}\ \texttt{v}_1\ \texttt{with}\ \texttt{t}_2 \longrightarrow \texttt{v}_1 \qquad \text{(E-TryV)}$$

$$\texttt{try}\ \texttt{error}\ \texttt{with}\ \texttt{t}_2 \longrightarrow \texttt{t}_2 \qquad \text{(E-TryError)}$$

$$\frac{\texttt{t}_1 \longrightarrow \texttt{t}_1'}{\texttt{try}\ \texttt{t}_1\ \texttt{with}\ \texttt{t}_2 \longrightarrow \texttt{try}\ \texttt{t}_1'\ \texttt{with}\ \texttt{t}_2} \qquad \text{(E-Try)}$$

Typing

$$\frac{\Gamma \vdash \texttt{t}_1 : \texttt{T} \qquad \Gamma \vdash \texttt{t}_2 : \texttt{T}}{\Gamma \vdash \texttt{try}\ \texttt{t}_1\ \texttt{with}\ \texttt{t}_2 : \texttt{T}} \qquad \text{(T-Try)}$$

Exceptions carrying values

`t`	::=	...	terms
		`raise t`	raise exception

Evaluation

$$\texttt{(raise } v_{11}\texttt{) } t_2 \longrightarrow \texttt{raise } v_{11} \qquad \text{(E-AppRaise1)}$$

$$v_1 \texttt{ (raise } v_{21}\texttt{)} \longrightarrow \texttt{raise } v_{21} \qquad \text{(E-AppRaise2)}$$

$$\frac{t_1 \longrightarrow t_1'}{\texttt{raise } t_1 \longrightarrow \texttt{raise } t_1'} \qquad \text{(E-Raise)}$$

$$\texttt{raise (raise } v_{11}\texttt{)} \longrightarrow \texttt{raise } v_{11} \qquad \text{(E-RaiseRaise)}$$

$$\texttt{try } v_1 \texttt{ with } t_2 \longrightarrow v_1 \qquad \text{(E-TryV)}$$

$$\texttt{try raise } v_{11} \texttt{ with } t_2 \longrightarrow t_2 \; v_{11} \qquad \text{(E-TryRaise)}$$

$$\frac{t_1 \longrightarrow t_1'}{\texttt{try } t_1 \texttt{ with } t_2 \longrightarrow \texttt{try } t_1' \texttt{ with } t_2} \qquad \text{(E-Try)}$$

Typing

$$\frac{\Gamma \vdash t_1 : T_{exn}}{\Gamma \vdash \texttt{raise } t_1 : T} \qquad \text{(T-Exn)}$$

$$\frac{\Gamma \vdash t_1 : T \qquad \Gamma \vdash t_2 : T_{exn} \to T}{\Gamma \vdash \texttt{try } t_1 \texttt{ with } t_2 : T} \qquad \text{(T-Try)}$$