FJ syntax (terms and values)

\[
\begin{align*}
t & ::= & \text{terms} \\
& | & x \\
& | & t.f \\
& | & t.m(\tau) \\
& | & \text{new } C(\tau) \\
& | & (C) \: t \\
\end{align*}
\]

\[
\begin{align*}
v & ::= & \text{values} \\
& | & \text{new } C(\tau) \\
\end{align*}
\]

Syntax (methods and classes)

\[
\begin{align*}
K & ::= & \text{constructor declarations} \\
& | & C(\tau) \: \{ \text{super}(\tau); \text{this}.f=\tau; \} \\
M & ::= & \text{method declarations} \\
& | & C \: m(\tau) \: \{ \text{return } t; \} \\
CL & ::= & \text{class declarations} \\
& | & \text{class } C \: \text{extends } C \{ \text{C } \tau; \: K \: M \}
\end{align*}
\]

Administrivia

- People taking the 380 final (or another) early on the 20th can start the CIS500 exam at noon. Let me know by email if you plan to do this.
Subtyping

Assume we have a (global, fixed) class table $CT$ mapping class names to definitions.

$CT(C) - \text{class } C \text{ extends } D \{ \ldots \}$

$C \subseteq D$

$C \subseteq C$

$C \subseteq D \quad D \subseteq E$

$C \subseteq E$

Field lookup

$\text{fields}(\text{Object}) = \emptyset$

$CT(C) - \text{class } C \text{ extends } D \{ \mathbb{F}; \ K \ \mathbb{M} \}$

$\text{fields}(D) = \mathbb{F}, \mathbb{G}$

$\text{fields}(C) = \mathbb{F}, \mathbb{G}, \mathbb{I}$

Method type lookup

$CT(C) - \text{class } C \text{ extends } D \{ \mathbb{F}; \ K \ \mathbb{M} \}$

$B = \{ \mathbb{F} \} \{ \text{return } t_i \} \in \mathbb{M}$

$mtype(m, C) = \mathbb{F} \rightarrow B$

$CT(C) - \text{class } C \text{ extends } D \{ \mathbb{F}; \ K \ \mathbb{M} \}$

$m \text{ is not defined in } \mathbb{M}$

$mtype(m, C) = mtype(m, D)$

Method body lookup

$CT(C) - \text{class } C \text{ extends } D \{ \mathbb{F}; \ K \ \mathbb{M} \}$

$B = \{ \mathbb{F} \} \{ \text{return } t_i \} \in \mathbb{M}$

$mbody(m, C) = (\pi, t)$

$CT(C) - \text{class } C \text{ extends } D \{ \mathbb{F}; \ K \ \mathbb{M} \}$

$m \text{ is not defined in } \mathbb{M}$

$mbody(m, C) = mbody(m, D)$
Valid method overriding

\[
\text{mtype}(m, D) \Rightarrow D_0 \text{ implies } C \Rightarrow D \text{ and } C_0 \Rightarrow D_0
\]

\[
\text{override}(m, D, C \Rightarrow C_0)
\]

The example again

class A extends Object {
    A() {
        super();
    }
}

class B extends Object {
    B() {
        super();
    }
}

class Pair extends Object {
    Object fst;
    Object snd;
    Pair(Object fst, Object snd) {
        super();
        this.fst = fst;
        this.snd = snd;
    }
    Pair setfst(Object newfst) {
        return new Pair(newfst, this.snd);
    }
}

Evaluation

Projection:

\[
\text{new Pair(new A(), new B()).snd} \rightarrow \text{new B()}
\]
Evaluation

Casting:

(Pair) new Pair(new A(), new B()) → new Pair(new A(), new B())

Method invocation:

\[
\begin{align*}
\text{new Pair}(\text{new A()}, \text{new B()}).\text{setfst}(\text{new B()}) \\
\quad \rightarrow [\text{newfst} \mapsto \text{new B()}, \\
\quad \text{this} \mapsto \text{new Pair}(\text{new A()}, \text{new B()})] \\
\quad \text{new Pair}(\text{newfst}, \text{this}.\text{snd})
\end{align*}
\]

(i.e., \text{new Pair}(\text{new B()}, \text{new Pair}(\text{new A()}, \text{new B()}).\text{snd})

Evaluation rules

\[
\begin{align*}
\text{fields}(\text{C}) = C \quad \text{I} \\
(\text{new } C(\tau)).f_i \mapsto v_i & \quad \text{(E-PROJNEW)} \\
\text{mbody}(m, C) = (x, t_0) \\
(\text{new } C(\tau)).m(\pi) & \rightarrow [x \mapsto \pi, \text{this } \mapsto \text{new } C(\tau)|t_0] & \text{(E-INVKNEW)} \\
C \triangleleft D & \rightarrow (D)(\text{new } C(\tau)) \rightarrow \text{new } C(\tau) & \text{(E-CASTNEW)}
\end{align*}
\]

plus some congruence rules...
Notes

FJ has no rule of subsumption (because we want to follow Java). The typing rules are algorithmic.

(Where would this make a difference?...)

Typing rules

\[
\begin{align*}
\frac{t_0 \rightarrow t'_0}{t_0.f \rightarrow t'_0.f} & \quad \text{(E-Field)} \\
\frac{t_0 \rightarrow t'_0}{t_0.m(e) \rightarrow t'_0.m(e)} & \quad \text{(E-InvK-Recv)} \\
\frac{t_i \rightarrow t'_i}{v_0.m(\tau, t_i, e) \rightarrow v_0.m(\tau, t'_i, e)} & \quad \text{(E-InvK-Arg)} \\
\frac{t_i \rightarrow t'_i}{\text{new } C(\tau, t_i, e) \rightarrow \text{new } C(\tau, t'_i, e)} & \quad \text{(E-New-Arg)} \\
\frac{t_0 \rightarrow t'_0}{(C)t_0 \rightarrow (C)t'_0} & \quad \text{(E-Cast)}
\end{align*}
\]

Typing
Typing rules

\[
\Gamma \vdash t_0 : C_0 \quad \text{fields}(C_0) \rightarrow C_i \\
\Gamma \vdash t_0.f_i : C_i
\]  
\hspace{1cm} (T-\text{FIELD})

\[
\Gamma \vdash t_0 : D \quad D <: C \\
\Gamma \vdash (C)t_0 : C
\]  
\hspace{1cm} (T-\text{UCAST})

\[
\Gamma \vdash t_0 : D \\
\Gamma \vdash t_0 ; C <: D \quad C \not{:} D \\
\Gamma \vdash (C)t_0 : C
\]  
\hspace{1cm} (T-\text{DCAST})

Why two cast rules?

Typing rules

\[
\Gamma \vdash t_0 : D \\
\Gamma \vdash (C)t_0 : C
\]  
\hspace{1cm} (T-\text{UCAST})

\[
\Gamma \vdash t_0 : D \quad C <: D \quad C \not{:} D \\
\Gamma \vdash (C)t_0 : C
\]  
\hspace{1cm} (T-\text{DCAST})

Why two cast rules? Because that's how Java does it!

Note that this rule "has subsumption built in" — i.e., the typing relation in FJ is written in the algorithmic style of TAPL chapter 16, not the declarative style of chapter 15.
Typing rules

\[
\begin{align*}
\Gamma \vdash t_0 : C_0 \\
\text{mtype}(\pi, C_0) \rightarrow \Delta \\
\Gamma \vdash E : C \quad C \vdash : \Delta \\
\Gamma \vdash t_0.\pi(\Delta) : C
\end{align*}
\] (T-InvK)

Note that this rule “has subsumption built in” — i.e., the typing relation in FJ is written in the algorithmic style of TAPL chapter 16, not the declarative style of chapter 15.

Why? Because Java does it this way!

Java typing is algorithmic

The Java typing relation is defined in the algorithmic style, for (at least) two reasons:

1. In order to perform static overloading resolution, we need to be able to speak of “the type” of an expression
2. We would otherwise run into trouble with typing of conditional expressions

Let’s look at the second in more detail...

Java typing must be algorithmic

We haven’t included them in FJ, but full Java has both interfaces and conditional expressions.

The two together actually make the declarative style of typing rules unworkable!
Java conditionals

\[ t_1 \in \text{bool} \quad t_2 \in T_2 \quad t_3 \in T_3 \]
\[ t_1 \ ? t_2 : t_3 \in ? \]

Actual Java rule (algorithmic):

\[ t_1 \in \text{bool} \quad t_2 \in T_2 \quad t_3 \in T_3 \]
\[ t_1 \ ? t_2 : t_3 \in \min(T_2, T_3) \]

More standard (declarative) rule:

\[ t_1 \in \text{bool} \quad t_2 \in T \quad t_3 \in T \]
\[ t_1 \ ? t_2 : t_3 \in T \]

Algorithmic version:

\[ t_1 \in \text{bool} \quad t_2 \in T_2 \quad t_3 \in T_3 \]
\[ t_1 \ ? t_2 : t_3 \in T_2 \lor T_3 \]

Requires joins!
Java has no joins

But, in full Java (with interfaces), there are types that have no join!
E.g.:

```java
interface I {...}
interface J {...}
interface K extends I, J {...}
interface L extends I, J {...}
```

K and L have no join (least upper bound) — both I and J are common upper bounds, but neither of these is less than the other.

So: algorithmic typing rules are really our only option.

---

FJ Typing rules

- fields(C) = ⊤ ⊢
- \( \Gamma \vdash \mathcal{E} : C \quad C \preceq D \)
- \( \Gamma \vdash \text{new } C(\mathcal{E}) : C \)

(T-New)

Typing rules (methods, classes)

```latex
\begin{align*}
\mathbf{K}: \mathcal{C}, \text{this}: C &\vdash t_0 : E_0 \quad E_0 \preceq C_0 \\
\text{CT}(C) - \text{class } C \text{ extends } D \{\ldots\} &\quad \text{override}(m, D, C \rightarrow C_0) \\
&\quad C_0 = (\mathcal{C} \mathbf{x}) \{\text{return } t_0;\} \text{ OK in } C
\end{align*}
```

```latex
\begin{align*}
\mathbf{K} - C(\mathcal{S} \mathbf{g}, C \mathcal{T}) - \text{super} (\mathcal{S}); \text{this}\mathcal{F} = \mathcal{F}; &\quad \text{fields}(D) - \mathcal{S} \mathbf{g} \quad \mathcal{M} \text{ OK in } C \\
&\quad \text{class } C \text{ extends } D \{\mathcal{C} \mathcal{F}; K \mathcal{M}\} \text{ OK}
\end{align*}
```
Preservation

Theorem [Preservation]: If $\Gamma \vdash t : C$ and $t \rightarrow t'$, then $\Gamma \vdash t' : C'$ for some $C' < C$.

Proof: Straightforward induction.

---

Preservation

Theorem [Preservation]: If $\Gamma \vdash t : C$ and $t \rightarrow t'$, then $\Gamma \vdash t' : C'$ for some $C' < C$.

Proof: Straightforward induction. ???

---

Preservation?

Surprise: well-typed programs can step to ill-typed ones!
(How?)

---

Preservation?

Surprise: well-typed programs can step to ill-typed ones!
Preservation?

Surprise: well-typed programs can step to ill-typed ones!
(How?)

\[
(A)(\text{Object})\text{new } B() 
\rightarrow (A)\text{new } B()
\]

Solution: “Stupid Cast” typing rule

Add another typing rule, marked “stupid” to

\[
\Gamma \vdash t_0 : D 
\quad C \not\subseteq D 
\quad D \not\subseteq C \\
\begin{array}{l}
\text{stupid warning} \\
\Gamma \vdash (C)t_0 : C
\end{array}
\]

(T-SCAST)

Solution: “Stupid Cast” typing rule

Add another typing rule, marked “stupid” to

\[
\Gamma \vdash t_0 : D 
\quad C \not\subseteq D 
\quad D \not\subseteq C \\
\begin{array}{l}
\text{stupid warning} \\
\Gamma \vdash (C)t_0 : C
\end{array}
\]

(T-SCAST)

Correspondence with Java

Let’s try to state precisely what we mean by “FJ corresponds to Java”:

Claim:

1. Every syntactically well-formed FJ program is also a syntactically well-formed Java program.
2. A syntactically well-formed FJ program is typable in FJ (without using the T-SCAST rule) iff it is typable in Java.
3. A well-typed FJ program behaves the same in FJ as in Java. (E.g., evaluating it in FJ diverges iff compiling and running it in Java diverges.)

Of course, without a formalization of full Java, we cannot prove this claim. But it’s still very useful to say precisely what we are trying to accomplish—in particular, it provides a rigorous way of judging counterexamples.

(Cf. “conservative extension” between logics.)
Alternative approaches to casting

- Loosen preservation theorem
- Use big-step semantics

Problem: well-typed programs can get stuck.
How?

Cast failure:
(A)new Object()
## Formalizing Progress

Solution: Weaken the statement of the progress theorem to

A well-typed FJ term is either a value or can reduce one step or is stuck at a failing cast.

Formalizing this takes a little more work...

## Evaluation Contexts

\[
E ::= \begin{align*}
&[] \\
&E.f \\
&E.m(v) \\
&v.m(r,E,E) \\
&\text{new } C(r,E,E) \\
&(C)E
\end{align*}
\]

evaluation contexts

- hole
- field access
- method invocation (receiver)
- method invocation (arg)
- object creation (arg)
- cast

Evaluation contexts capture the notion of the “next subterm to be reduced,” in the sense that, if \( t \rightarrow t' \), then we can express \( t \) and \( t' \) as \( t = E[r] \) and \( t' = E[r'] \) for a unique \( E, r, \) and \( r' \), with \( r \rightarrow r' \) by one of the computation rules \( E\text{-PROJNEW}, E\text{-INVKNEW}, \) or \( E\text{-CASTNEW} \).

## Progress

**Theorem** [Progress]: Suppose \( t \) is a closed, well-typed normal form. Then either (1) \( t \) is a value, or (2) \( t \rightarrow t' \) for some \( t' \), or (3) for some evaluation context \( E \), we can express \( t \) as \( t = E(C)(\text{new } D(r)) \), with \( D \not\subseteq C \).

CIS 500, 27 November