CIS 500

Software Foundations

Fall 2005

Novemeber 30

Object Encodings

Last time, we talked about encoding objects in the typed lambda calculus with records, recursion, references and subtyping.

We have a litte more to talk about this topic, but let's work through an example to see where we are.

Example from last time

```
class SetCounter {
   protected int x = 1;
   int get() { return x; }
   void set(int i) { x = i; return; }
   void inc() { this.set(this.get() + 1); return; }
}

class InstrCounter extends SetCounter {
   protected int a = 0;
   void set(int i) { a++; super.set(i); return; }
   int accesses() { return a; }
}
```



```
SetCounter = {get:Unit→Nat, set:Nat→Unit, inc:Unit→Unit};
CounterRep = {x:Ref Nat};

setCounterClass =
  λr:CounterRep.
     λthis:SetCounter.
       {get = λ_:Unit. !(r.x),
        set = λi:Nat.  r.x:=i,
        inc = λ_:Unit. this.set (succ(this.get unit))};

newSetCounter =
  λ_:Unit. let r = {x=ref 1} in
              fix (setCounterClass r);
```

```
InstrCounter = {get:Unit→Nat, set:Nat→Unit,
                inc:Unit→Unit, accesses:Unit→Nat};

InstrCounterRep = {x:Ref Nat, a:Ref Nat};

instrCounterClass =
  λr:InstrCounterRep.
    λthis:InstrCounter.
      let super = setCounterClass r this in
        {get = super.get,
         set = λi:Nat. (r.a:=succ(!(r.a)); super.set i),
         inc = super.inc,
         accesses = λ_:Unit. !(r.a)};
newInstrCounter =
  λ_:Unit. let r = {x=ref 1, a=ref 0} in
             fix (instrCounterClass r);
```


One more refinement...

A small fly in the ointment

The implementation we have given for instrumented counters has a problem because calling the object creation function

```
newInstrCounter =
  λ_:Unit. let r = {x=ref 1, a=ref 0} in
             fix (instrCounterClass r);
```

will cause the evaluator to diverge!

Intuitively (see TAPL for details), the problem is the "unprotected" use of `this` in the call to `setCounterClass` in `instrCounterClass`:

```
instrCounterClass =
  λr:InstrCounterRep.
    λthis: InstrCounter.
      let super = setCounterClass r this in
        ...
```


To see why this diverges, consider a simpler example:

```
ff = λf:Nat→Nat.
       let f′ = f in
       λn:Nat. 0
```

$\Longrightarrow$ `ff : (Nat→Nat) → (Nat→Nat)`

Now:

```
 fix ff   ⟶   ff (fix ff)
          ⟶   let f′ = (fix ff) in λn:Nat. 0
          ⟶   let f′ = ff (fix ff) in λn:Nat. 0
          ⟶   let f′ = (let f′ = (fix ff) in λn:Nat. 0) in λn:Nat. 0
          ⟶   uh oh...
```

One possible solution

Idea: "delay" `this` by putting a dummy abstraction in front of it...

```
setCounterClass =
  λr:CounterRep.
  λthis: Unit→SetCounter.
    λ_:Unit.
      {get = λ_:Unit. !(r.x),
       set = λi:Nat.  r.x:=i,
       inc = λ_:Unit. (this unit).set(succ((this unit).get unit))};
```

$\Longrightarrow$

```
setCounterClass : CounterRep → (Unit→SetCounter) → (Unit→SetCounter)

newSetCounter =
  λ_:Unit. let r = {x=ref 1} in
              fix (setCounterClass r) unit;
```


Similarly:

```
instrCounterClass =
  λr:InstrCounterRep.
  λthis: Unit→InstrCounter.
    λ_:Unit.
      let super = setCounterClass r this unit in
         {get = super.get,
          set = λi:Nat. (r.a:=succ(!(r.a)); super.set i),
          inc = super.inc,
          accesses = λ_:Unit. !(r.a)};

newInstrCounter =
  λ_:Unit. let r = {x=ref 1, a=ref 0} in
              fix (instrCounterClass r) unit;
```


Success

This works, in the sense that we can now instantiate `instrCounterClass` (without diverging!), and its instances behave in the way we intended.

Success (?)

This works, in the sense that we can now instantiate `instrCounterClass` (without diverging!), and its instances behave in the way we intended.

However, all the "delaying" we added has an unfortunate side effect: instead of computing the "method table" just once, when an object is created, we will now re-compute it every time we invoke a method!

Section 18.12 in TAPL shows how this can be repaired by using references instead of `fix` to "tie the knot" in the method table.

Recap

Multiple representations

All the objects we have built in this series of examples have type `Counter`.

But their internal representations vary widely.

Encapsulation

An object is a record of functions, which maintain common internal state via a shared reference to a record of mutable instance variables.

This state is inaccessible outside of the object because there is no way to name it. (Instance variables can only be named from inside the methods.)

Subtyping

Subtyping between object types is just ordinary subtyping between types of records of functions.

Functions like `inc3` that expect `Counter` objects as parameters can (safely) be called with objects belonging to any subtype of `Counter`.

Inheritance

Classes are data structures that can be both extended and instantiated.

We modeled inheritance by copying implementations of methods from superclasses to subclasses.

Each class

- waits to be told a record `r` of instance variables and an object `this` (which should have the same interface and be based on the same record of instance variables)
- uses `r` and `this` to instantiate its superclass
- constructs a record of method implementations, copying some directly from `super` and implementing others in terms of `this` and `super`.

The `this` parameter is "resolved" at object creation time using `fix`.

Where we are...

The essence of objects

- Dynamic dispatch
- Encapsulation of state with behavior
- Behavior-based subtyping
- Inheritance (incremental definition of behaviors)
- Access of super class
- "Open recursion" through `this`

What's missing

The peculiar status of classes (which are both run-time and compile-time things)

Named types with declared subtyping

Recursive types

Run-time type analysis (casting, etc.)

(...lots of other stuff)

Modeling Java

Models in General

No such thing as a "perfect model" — The nature of a model is to abstract away from details!

So models are never just "good": they are always "good for some specific set of purposes."

Models of Java

Lots of different purposes $\longrightarrow$ lots of different kinds of models

- Source-level vs. bytecode level
- Large (inclusive) vs. small (simple) models
- Models of type system vs. models of run-time features (not entirely separate issues)
- Models of specific features (exceptions, concurrency, reflection, class loading, ...)
- Models designed for extension

Featherweight Java

Purpose: model the "core OO features" and their types and nothing else.

History:

- Originally proposed by a Penn PhD student (Atsushi Igarashi) as a tool for analyzing GJ ("Java plus generics")
- Since used by many others for studying a wide variety of Java features and proposed extensions

Things left out

- Reflection, concurrency, class loading, inner classes, ...

Things left out

- Reflection, concurrency, class loading, inner classes, ...
- Exceptions, loops, ...

Things left out

- Reflection, concurrency, class loading, inner classes, ...
- Exceptions, loops, ...
- Interfaces, overloading, ...

Things left out

- Reflection, concurrency, class loading, inner classes, ...
- Exceptions, loops, ...
- Interfaces, overloading, ...
- Assignment (!!)

Things left in

- Classes and objects
- Methods and method invocation
- Fields and field access
- Inheritance (including open recursion through `this`)
- Casting

Example

```
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); }
}
```


Conventions

For syntactic regularity...

- Always include superclass (even when it is `Object`)
- Always write out constructor (even when trivial)
- Always call `super` from constructor (even when no arguments are passed)
- Always explicitly name receiver object in method invocation or field access (even when it is `this`)
- Methods always consist of a single `return` expression
- Constructors always
  - Take same number (and types) of parameters as fields of the class
  - Assign constructor parameters to "local fields"
  - Call `super` constructor to assign remaining fields
  - Do nothing else

Formalizing FJ

Nominal type systems

Big dichotomy in the world of programming languages:

- Structural type systems:
  - What matters about a type (for typing, subtyping, etc.) is just its structure.
  - Names are just convenient (but inessential) abbreviations.
- Nominal type systems:
  - Types are always named.
  - Typechecker mostly manipulates names, not structures.
  - Subtyping is declared explicitly by programmer (and checked for consistency by compiler).

Advantages of Structural Systems

Somewhat simpler, cleaner, and more elegant (no need to always work wrt. a set of "name definitions")

Easier to extend (e.g. with parametric polymorphism)

Caveat: when recursive types are considered, some of this simplicity and elegance slips away...

Advantages of Nominal Systems

Recursive types fall out easily

Using names everywhere makes typechecking (and subtyping, etc.) easy and efficient

Type names are also useful at run-time (for casting, type testing, reflection, ...).

Java (like most other mainstream languages) is a nominal system.

Representing objects

Our decision to omit assignment has a nice side effect...

The only ways in which two objects can differ are (1) their classes and (2) the parameters passed to their constructor when they were created.

All this information is available in the `new` expression that creates an object. So we can identify the created object with the `new` expression.

Formally: object values have the form `new C(`$\overline{\texttt{v}}$`)`

FJ Syntax

Syntax (terms and values)

`t` ::=		*terms*
	`x`	*variable*
	`t.f`	*field access*
	`t.m(`$\overline{\texttt{t}}$`)`	*method invocation*
	`new C(`$\overline{\texttt{t}}$`)`	*object creation*
	`(C) t`	*cast*
`v` ::=		*values*
	`new C(`$\overline{\texttt{v}}$`)`	*object creation*

Syntax (methods and classes)

`K` ::=		*constructor declarations*
	`C(`$\overline{\texttt{C}}$ $\overline{\texttt{f}}$`) {super(`$\overline{\texttt{f}}$`); this.`$\overline{\texttt{f}}$`=`$\overline{\texttt{f}}$`;}`	
`M` ::=		*method declarations*
	`C m(`$\overline{\texttt{C}}$ $\overline{\texttt{x}}$`) {return t;}`	
`CL` ::=		*class declarations*
	`class C extends C {`$\overline{\texttt{C}}$ $\overline{\texttt{f}}$`; K `$\overline{\texttt{M}}$`}`	

Subtyping

Subtyping

As in Java, subtyping in FJ is declared.

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

$$\frac{CT(\texttt{C}) = \texttt{class C extends D \{...\}}}{\texttt{C} <: \texttt{D}}$$

$$\texttt{C} <: \texttt{C}$$

$$\frac{\texttt{C} <: \texttt{D} \qquad \texttt{D} <: \texttt{E}}{\texttt{C} <: \texttt{E}}$$

More auxiliary definitions

From the class table, we can read off a number of other useful properties of the definitions (which we will need later for typechecking and operational semantics)...

Fields lookup

$$\mathit{fields}(\texttt{Object}) = \emptyset$$

$$\frac{CT(\texttt{C}) = \texttt{class C extends D \{}\overline{\texttt{C}}\ \overline{\texttt{f}}\texttt{; K }\overline{\texttt{M}}\texttt{\}} \qquad \mathit{fields}(\texttt{D}) = \overline{\texttt{D}}\ \overline{\texttt{g}}}{\mathit{fields}(\texttt{C}) = \overline{\texttt{D}}\ \overline{\texttt{g}}, \overline{\texttt{C}}\ \overline{\texttt{f}}}$$

Method type lookup

$$\frac{CT(\texttt{C}) = \texttt{class C extends D \{}\overline{\texttt{C}}\ \overline{\texttt{f}}\texttt{; K }\overline{\texttt{M}}\texttt{\}} \qquad \texttt{B m (}\overline{\texttt{B}}\ \overline{\texttt{x}}\texttt{) \{return t;\}} \in \overline{\texttt{M}}}{\mathit{mtype}(\texttt{m}, \texttt{C}) = \overline{\texttt{B}} \to \texttt{B}}$$

$$\frac{CT(\texttt{C}) = \texttt{class C extends D \{}\overline{\texttt{C}}\ \overline{\texttt{f}}\texttt{; K }\overline{\texttt{M}}\texttt{\}} \qquad \texttt{m is not defined in } \overline{\texttt{M}}}{\mathit{mtype}(\texttt{m}, \texttt{C}) = \mathit{mtype}(\texttt{m}, \texttt{D})}$$

Method body lookup

$$\frac{CT(\texttt{C}) = \texttt{class C extends D \{}\overline{\texttt{C}}\ \overline{\texttt{f}}\texttt{; K }\overline{\texttt{M}}\texttt{\}} \qquad \texttt{B m (}\overline{\texttt{B}}\ \overline{\texttt{x}}\texttt{) \{return t;\}} \in \overline{\texttt{M}}}{mbody(\texttt{m}, \texttt{C}) = (\overline{\texttt{x}}, \texttt{t})}$$

$$\frac{CT(\texttt{C}) = \texttt{class C extends D \{}\overline{\texttt{C}}\ \overline{\texttt{f}}\texttt{; K }\overline{\texttt{M}}\texttt{\}} \qquad \texttt{m is not defined in } \overline{\texttt{M}}}{mbody(\texttt{m}, \texttt{C}) = mbody(\texttt{m}, \texttt{D})}$$

Valid method overriding

$$\frac{mtype(\texttt{m}, \texttt{D}) = \overline{\texttt{D}} \to \texttt{D}_0 \text{ implies } \overline{\texttt{C}} = \overline{\texttt{D}} \text{ and } \texttt{C}_0 = \texttt{D}_0}{override(\texttt{m}, \texttt{D}, \overline{\texttt{C}} \to \texttt{C}_0)}$$

Evaluation

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:

```
new Pair(new A(), new B()).snd  ⟶  new B()
```


Evaluation

Casting:

```
(Pair)new Pair(new A(), new B())  ⟶  new Pair(new A(), new B())
```


Evaluation

Method invocation:

```
new Pair(new A(), new B()).setfst(new B())
⟶ [ newfst ↦ new B(),
    this ↦ new Pair(new A(),new B()) ]
  new Pair(newfst, this.snd)
```

i.e., `new Pair(new B(), new Pair(new A(), new B()).snd)`


```
((Pair) (new Pair(new Pair(new A(),new B()), new A())
          .fst).snd
⟶ ((Pair)new Pair(new A(),new B())).snd
⟶ new Pair(new A(), new B()).snd
⟶ new B()
```

Evaluation rules

$$\frac{\mathit{fields}(\mathtt{C}) = \overline{\mathtt{C}}\ \overline{\mathtt{f}}}{(\mathtt{new\ C}(\overline{\mathtt{v}}))\mathtt{.f_i} \longrightarrow \mathtt{v_i}} \quad \text{(E-ProjNew)}$$

$$\frac{\mathit{mbody}(\mathtt{m}, \mathtt{C}) = (\overline{\mathtt{x}}, \mathtt{t_0})}{(\mathtt{new\ C}(\overline{\mathtt{v}}))\mathtt{.m}(\overline{\mathtt{u}}) \longrightarrow [\overline{\mathtt{x}} \mapsto \overline{\mathtt{u}}, \mathtt{this} \mapsto \mathtt{new\ C}(\overline{\mathtt{v}})]\mathtt{t_0}} \quad \text{(E-InvkNew)}$$

$$\frac{\mathtt{C} <: \mathtt{D}}{(\mathtt{D})(\mathtt{new\ C}(\overline{\mathtt{v}})) \longrightarrow \mathtt{new\ C}(\overline{\mathtt{v}})} \quad \text{(E-CastNew)}$$

plus some congruence rules...

$$\frac{\mathtt{t_0} \longrightarrow \mathtt{t_0'}}{\mathtt{t_0.f} \longrightarrow \mathtt{t_0'.f}} \quad \text{(E-Field)}$$

$$\frac{\mathtt{t_0} \longrightarrow \mathtt{t_0'}}{\mathtt{t_0.m}(\overline{\mathtt{t}}) \longrightarrow \mathtt{t_0'.m}(\overline{\mathtt{t}})} \quad \text{(E-Invk-Recv)}$$

$$\frac{\mathtt{t_i} \longrightarrow \mathtt{t_i'}}{\mathtt{v_0.m}(\overline{\mathtt{v}}, \mathtt{t_i}, \overline{\mathtt{t}}) \longrightarrow \mathtt{v_0.m}(\overline{\mathtt{v}}, \mathtt{t_i'}, \overline{\mathtt{t}})} \quad \text{(E-Invk-Arg)}$$

$$\frac{\mathtt{t_i} \longrightarrow \mathtt{t_i'}}{\mathtt{new\ C}(\overline{\mathtt{v}}, \mathtt{t_i}, \overline{\mathtt{t}}) \longrightarrow \mathtt{new\ C}(\overline{\mathtt{v}}, \mathtt{t_i'}, \overline{\mathtt{t}})} \quad \text{(E-New-Arg)}$$

$$\frac{\mathtt{t_0} \longrightarrow \mathtt{t_0'}}{\mathtt{(C)t_0} \longrightarrow \mathtt{(C)t_0'}} \quad \text{(E-Cast)}$$

Typing

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

$$\frac{\texttt{x:C} \in \Gamma}{\Gamma \vdash \texttt{x} : \texttt{C}} \quad \text{(T-Var)}$$

Typing rules

$$\frac{\Gamma \vdash \texttt{t}_0 : \texttt{C}_0 \qquad \mathit{fields}(\texttt{C}_0) = \overline{\texttt{C}}\ \overline{\texttt{f}}}{\Gamma \vdash \texttt{t}_0.\texttt{f}_i : \texttt{C}_i} \quad \text{(T-Field)}$$

Typing rules

$$\frac{\Gamma \vdash \texttt{t}_0 : \texttt{D} \qquad \texttt{D} <: \texttt{C}}{\Gamma \vdash \texttt{(C)t}_0 : \texttt{C}} \quad \text{(T-UCast)}$$

$$\frac{\Gamma \vdash \texttt{t}_0 : \texttt{D} \qquad \texttt{C} <: \texttt{D} \qquad \texttt{C} \neq \texttt{D}}{\Gamma \vdash \texttt{(C)t}_0 : \texttt{C}} \quad \text{(T-DCast)}$$

Why two cast rules?

Typing rules

$$\frac{\Gamma \vdash \texttt{t}_0 : \texttt{D} \qquad \texttt{D} <: \texttt{C}}{\Gamma \vdash \texttt{(C)t}_0 : \texttt{C}} \quad \text{(T-UCast)}$$

$$\frac{\Gamma \vdash \texttt{t}_0 : \texttt{D} \qquad \texttt{C} <: \texttt{D} \qquad \texttt{C} \neq \texttt{D}}{\Gamma \vdash \texttt{(C)t}_0 : \texttt{C}} \quad \text{(T-DCast)}$$

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

Typing rules

$$\frac{\Gamma \vdash t_0 : C_0 \quad mtype(m, C_0) = \overline{D} \to C \quad \Gamma \vdash \overline{t} : \overline{C} \quad \overline{C} <: \overline{D}}{\Gamma \vdash t_0.m(\overline{t}) : C} \qquad \text{(T-Invk)}$$

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.

Typing rules

$$\frac{\Gamma \vdash t_0 : C_0 \quad mtype(m, C_0) = \overline{D} \to C \quad \Gamma \vdash \overline{t} : \overline{C} \quad \overline{C} <: \overline{D}}{\Gamma \vdash t_0.m(\overline{t}) : C} \qquad \text{(T-Invk)}$$

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!

Typing rules

$$\frac{\Gamma \vdash t_0 : C_0 \quad mtype(m, C_0) = \overline{D} \to C \quad \Gamma \vdash \overline{t} : \overline{C} \quad \overline{C} <: \overline{D}}{\Gamma \vdash t_0.m(\overline{t}) : C} \qquad \text{(T-Invk)}$$

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!

But why does Java do 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

$$\frac{t_1 \in \texttt{bool} \qquad t_2 \in T_2 \qquad t_3 \in T_3}{t_1 \texttt{ ? } t_2 \texttt{ : } t_3 \in \texttt{?}}$$

Java conditionals

$$\frac{t_1 \in \texttt{bool} \qquad t_2 \in T_2 \qquad t_3 \in T_3}{t_1 \texttt{ ? } t_2 \texttt{ : } t_3 \in \texttt{?}}$$

Actual Java rule (algorithmic):

$$\frac{t_1 \in \texttt{bool} \qquad t_2 \in T_2 \qquad t_3 \in T_3}{t_1 \texttt{ ? } t_2 \texttt{ : } t_3 \in \mathit{min}(T_2, T_3)}$$

More standard (declarative) rule:

$$\frac{t_1 \in \texttt{bool} \qquad t_2 \in T \qquad t_3 \in T}{t_1 \texttt{ ? } t_2 \texttt{ : } t_3 \in T}$$

More standard (declarative) rule:

$$\frac{t_1 \in \texttt{bool} \qquad t_2 \in T \qquad t_3 \in T}{t_1 \texttt{ ? } t_2 \texttt{ : } t_3 \in T}$$

Algorithmic version:

$$\frac{t_1 \in \texttt{bool} \qquad t_2 \in T_2 \qquad t_3 \in T_3}{t_1 \texttt{ ? } t_2 \texttt{ : } t_3 \in T_2 \vee T_3}$$

Requires joins!

Java has no joins

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

E.g.:

```
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

$$\frac{\mathit{fields}(\texttt{C}) = \overline{\texttt{D}}\ \overline{\texttt{f}} \qquad \Gamma \vdash \overline{\texttt{t}} : \overline{\texttt{C}} \qquad \overline{\texttt{C}} \texttt{ <: } \overline{\texttt{D}}}{\Gamma \vdash \texttt{new C}(\overline{\texttt{t}}) : \texttt{C}} \quad \text{(T-New)}$$

Typing rules (methods, classes)

$$\frac{\overline{\texttt{x}} : \overline{\texttt{C}}, \texttt{this} : \texttt{C} \vdash \texttt{t}_0 : \texttt{E}_0 \qquad \texttt{E}_0 \texttt{ <: } \texttt{C}_0 \qquad \mathit{CT}(\texttt{C}) = \texttt{class C extends D } \{\ldots\} \qquad \mathit{override}(\texttt{m}, \texttt{D}, \overline{\texttt{C}} \to \texttt{C}_0)}{\texttt{C}_0\ \texttt{m}\ (\overline{\texttt{C}}\ \overline{\texttt{x}})\ \{\texttt{return t}_0\texttt{;}\}\ \texttt{OK in C}}$$

$$\frac{\texttt{K} = \texttt{C}(\overline{\texttt{D}}\ \overline{\texttt{g}}, \overline{\texttt{C}}\ \overline{\texttt{f}})\ \{\texttt{super}(\overline{\texttt{g}})\texttt{; this.}\overline{\texttt{f}} = \overline{\texttt{f}}\texttt{;}\} \qquad \mathit{fields}(\texttt{D}) = \overline{\texttt{D}}\ \overline{\texttt{g}} \qquad \overline{\texttt{M}}\ \texttt{OK in C}}{\texttt{class C extends D } \{\overline{\texttt{C}}\ \overline{\texttt{f}}\texttt{; K } \overline{\texttt{M}}\} \texttt{ OK}}$$

Properties

Progress

Progress

Problem: well-typed programs can get stuck.

How?

Progress

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

$\mathtt{E}$ ::=		*evaluation contexts*
	$[\,]$	*hole*
	$\mathtt{E.f}$	*field access*
	$\mathtt{E.m}(\overline{\mathtt{t}})$	*method invocation (receiver)*
	$\mathtt{v.m}(\overline{\mathtt{v}},\mathtt{E},\overline{\mathtt{t}})$	*method invocation (arg)*
	$\mathtt{new\ C}(\overline{\mathtt{v}},\mathtt{E},\overline{\mathtt{t}})$	*object creation (arg)*
	$\mathtt{(C)E}$	*cast*

Evaluation contexts capture the notion of the "next subterm to be reduced," in the sense that, if $\mathtt{t} \longrightarrow \mathtt{t'}$, then we can express $\mathtt{t}$ and $\mathtt{t'}$ as $\mathtt{t} = \mathtt{E}[\mathtt{r}]$ and $\mathtt{t'} = \mathtt{E}[\mathtt{r'}]$ for a unique $\mathtt{E}$, $\mathtt{r}$, and $\mathtt{r'}$, with $\mathtt{r} \longrightarrow \mathtt{r'}$ by one of the computation rules E-ProjNew, E-InvkNew, or E-CastNew.

Progress

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

Preservation

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

Proof: Straightforward induction.

Preservation

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

Proof: Straightforward induction. ???

Preservation?

Preservation?

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

(How?)

Preservation?

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

(How?)

```
(A)(Object)new B()  ⟶  (A)new B()
```

Solution: "Stupid Cast" typing rule

Add another typing rule, marked "stupid" to

$$\frac{\Gamma \vdash t_0 : D \qquad C \not<: D \qquad D \not<: C \qquad \textit{stupid warning}}{\Gamma \vdash (C)t_0 : C} \qquad \text{(T-SCast)}$$

Solution: "Stupid Cast" typing rule

Add another typing rule, marked "stupid" to

$$\frac{\Gamma \vdash t_0 : D \qquad C \not<: D \qquad D \not<: C \qquad \textit{stupid warning}}{\Gamma \vdash (C)t_0 : C} \qquad \text{(T-SCast)}$$

This is an example of a modeling technicality; not very interesting or deep, but we have to get it right if we're going to claim that the model is an accurate representation of (this fragment of) Java.

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