Foundations of Software Winter Semester 2007

Week 13 December 11

December 11, 2007 - version 1.0

Plan

PREVIOUSLY: objects through translation TODAY: Featherweight Java NEXT: Featherweight Scala



Last week

The lectures last week developed a series of increasingly sophisticated examples of "OO-style programming" in a typed lambda-calculus.

Multiple representations

All the objects in all the examples have type Counter (and sometimes more specific types).

But their internal representations vary widely.

Encapsulation

An object is a record of function. Hidden, internal state is implemented via capturing variables that hold ref's.

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. To delay the binding as long as possible, this fix needs to happen late.

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 (wrt. Java, say)

We haven't really captured the peculiar status of *classes* (which are both run-time and compile-time things) — we've captured the run-time aspect, but not the way in which classes get used as *types* in Java.

Also not named types with declared subtyping

Nor recursive types

Nor run-time type analysis (casting, etc.)

(... nor lots of other stuff)

Modeling Java

About models (of things 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" [or "bad"]: they are always "good [or bad] 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 "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"), which later became Java 1.5
- 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, ...
- 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 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



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 (without generics) uses nominal types only.

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 $C(\overline{v})$

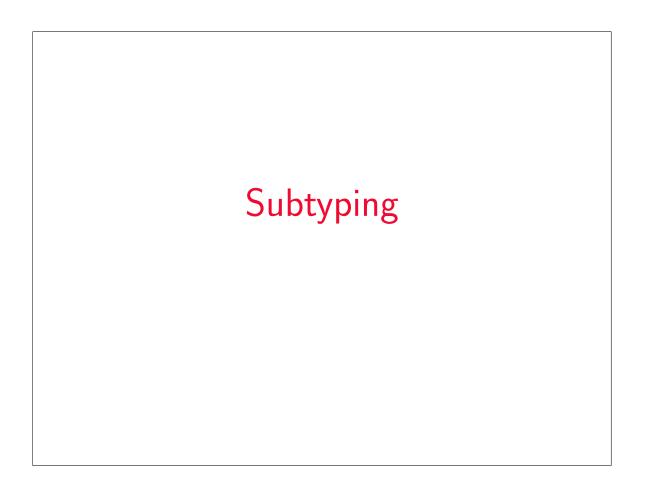


Syntax (terms and values) t ::= x t.f t.m(t) new C(t) (C) t v ::= new C(\overline{v})

terms variable field access method invocation object creation cast

values object creation Syntax (methods and classes)

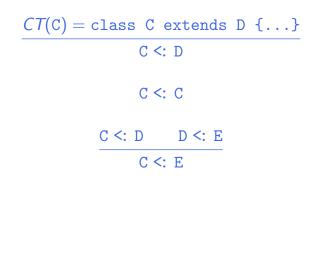
```
K :::= c(\overline{C} \ \overline{f}) \ \{super(\overline{f}); \ this.\overline{f}=\overline{f};\} constructor declarationsM :::= method declarationsC \ m(\overline{C} \ \overline{x}) \ \{return \ t;\} class declarationsCL :::= class C \ extends C \ \{\overline{C} \ \overline{f}; \ K \ \overline{M}\} class declarations
```



Subtyping

As in Java, subtyping in FJ is *declared*.

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



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)...

Field(s) lookup

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

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

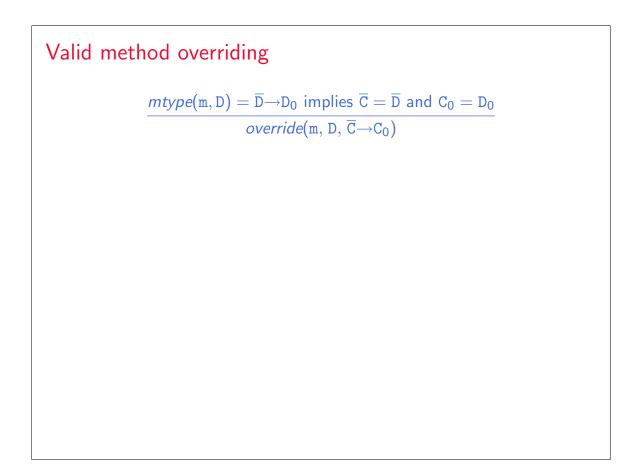
Method type lookup $\mathcal{CT}(C) = class \ C \ extends \ D \ \{\overline{C} \ \overline{f}; \ K \ \overline{M}\}$
 $\underline{B \ m \ (\overline{B} \ \overline{x}) \ \{return \ t;\} \in \overline{M} \ mtype(m, C) = \overline{B} \rightarrow B}$ $\mathcal{CT}(C) = class \ C \ extends \ D \ \{\overline{C} \ \overline{f}; \ K \ \overline{M}\}$
 $\underline{m \ is \ not \ defined \ in \ \overline{M} \ mtype(m, C) = mtype(m, D)}$

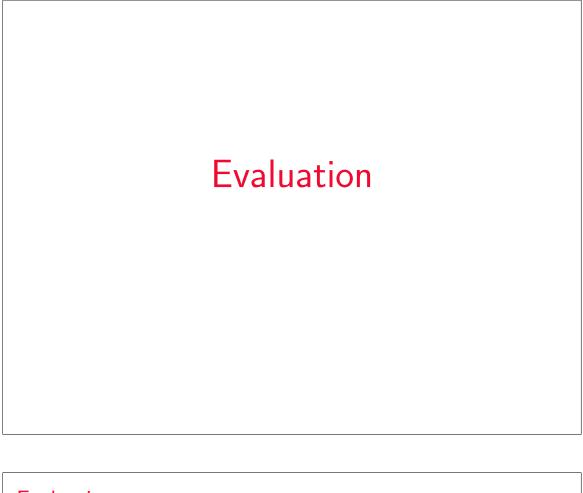
Method body lookup

 $\begin{array}{c} \textit{CT}(\texttt{C}) = \texttt{class C} \texttt{ extends D} \{\overline{\texttt{C}} \ \overline{\texttt{f}}\texttt{; K} \ \overline{\texttt{M}}\} \\ \hline \texttt{B m} \ (\overline{\texttt{B}} \ \overline{\texttt{x}}) \ \{\texttt{return t}\texttt{;}\} \in \overline{\texttt{M}} \\ \hline \hline \textit{mbody}(\texttt{m},\texttt{C}) = (\overline{\texttt{x}},\texttt{t}) \end{array}$

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

mbody(m, C) = mbody(m, D)





Evaluation						
Projectio	n:					
new	Pair(new	A(), new	B()).snd	\longrightarrow	new B()	



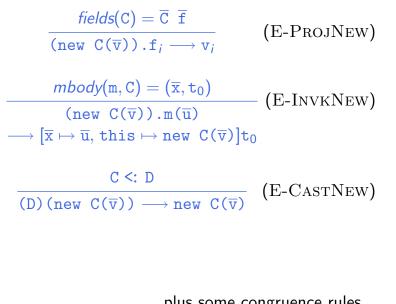
Casting:

Evaluation Order

FJ uses a call-by-value evaluation order (like lambda-calculus and Java):

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

Evaluation rules



$$\frac{t_{0} \longrightarrow t'_{0}}{t_{0}.f \longrightarrow t'_{0}.f} \qquad (E-FIELD)$$

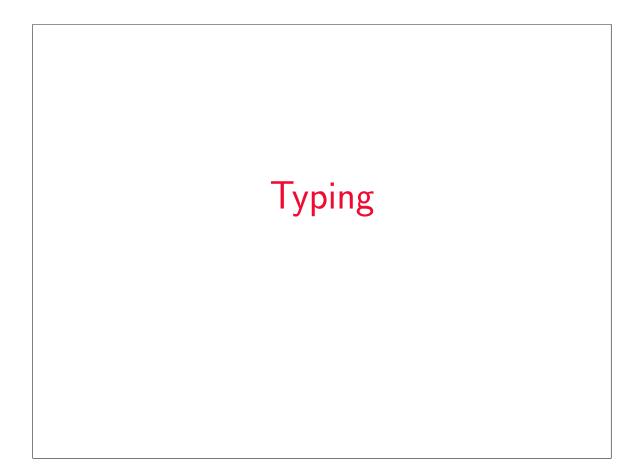
$$\frac{t_{0} \longrightarrow t'_{0}}{t_{0} \dots (t) \longrightarrow t'_{0} \dots (t)} \qquad (E-INVK-RECV)$$

$$\frac{t_{i} \longrightarrow t'_{i}}{t_{0} \dots (t), t_{i}, t) \longrightarrow v_{0} \dots (t), t'_{i}, t)} (E-INVK-ARG)$$

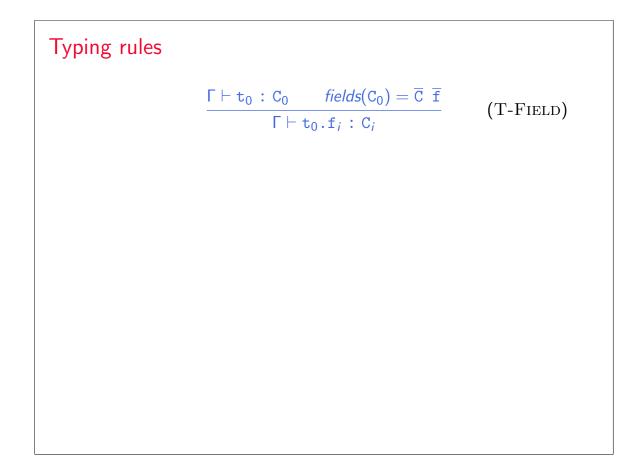
$$\frac{t_{i} \longrightarrow t'_{i}}{new \ C(\overline{v}, t_{i}, t) \longrightarrow new \ C(\overline{v}, t'_{i}, t)} (E-NEW-ARG)$$

$$\frac{t_{0} \longrightarrow t'_{0}}{(C)t_{0} \longrightarrow (C)t'_{0}} \qquad (E-CAST)$$

plus some congruence rules...



Typing rules		
	$\frac{\mathbf{x} : \mathbf{C} \in \Gamma}{\Gamma \vdash \mathbf{x} : C}$	(T-VAR)



Typing rules						
	$\frac{\Gamma \vdash t_0 : D D <: C}{\Gamma \vdash (C)t_0 : C}$	(T-UCAST)				
	$\frac{t_0: D C <: D C \neq D}{\Gamma \vdash (C)t_0: C}$	(T-DCAST)				
Why two cast rules?						

Typing rules

$$\frac{\Gamma \vdash t_0 : D \quad D <: C}{\Gamma \vdash (C)t_0 : C} \qquad (T-UCAST)$$

$$\frac{\Gamma \vdash t_0 : D \quad C <: D \quad C \neq D}{\Gamma \vdash (C)t_0 : C} \qquad (T-DCAST)$$

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

From the transformation of transformation o

Typing rules

$$\begin{bmatrix} \Gamma \vdash t_0 : C_0 \\ mtype(m, C_0) = \overline{D} \rightarrow C \\ \hline \Gamma \vdash \overline{t} : \overline{C} & \overline{C} <: \overline{D} \\ \hline \Gamma \vdash t_0 . m(\overline{t}) : C \end{bmatrix}$$
(T-INVK)

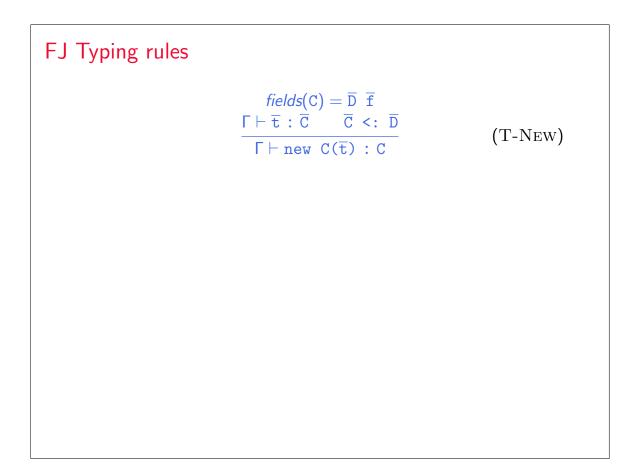
Note that this rule "has subsumption built in" — i.e., the typing relation in FJ is written in the *algorithmic* style of TAPL chapter 16, not the declarative style of chapter 15.

Why? Because Java does it this way!

Typing rules $\Gamma \vdash t_0 : C_0$
 $mtype(m, C_0) = \overline{D} \rightarrow C$
 $\Gamma \vdash \overline{t} : \overline{C} \quad \overline{C} <: \overline{D}$
 $\Gamma \vdash t_0 . m(\overline{t}) : C$ (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? After the break...

Typing rules (methods, classes)

$$\begin{split} \overline{\mathbf{x}} : \overline{\mathbf{C}}, \, \texttt{this} : \mathbf{C} \vdash \mathbf{t}_0 : \mathbf{E}_0 & \mathbf{E}_0 <: \mathbf{C}_0 \\ CT(\mathbf{C}) = \texttt{class } \mathbf{C} \text{ extends } \mathbf{D} \ \{\dots\} \\ \hline \\ \frac{\textit{override}(\mathbf{m}, \mathbf{D}, \overline{\mathbf{C}} \rightarrow \mathbf{C}_0)}{\mathbf{C}_0 \ \mathbf{m} \ (\overline{\mathbf{C}} \ \overline{\mathbf{x}}) \ \{\texttt{return } \mathbf{t}_0;\} \ \mathsf{OK} \ \texttt{in } \mathbf{C} \\ \hline \\ K = \mathbf{C}(\overline{\mathbf{D}} \ \overline{\mathbf{g}}, \ \overline{\mathbf{C}} \ \overline{\mathbf{f}}) \ \{\texttt{super}(\overline{\mathbf{g}}); \ \texttt{this}.\overline{\mathbf{f}} = \overline{\mathbf{f}};\} \\ \hline \\ \frac{\textit{fields}(\mathbf{D}) = \overline{\mathbf{D}} \ \overline{\mathbf{g}} & \overline{\mathbf{M}} \ \mathsf{OK} \ \texttt{in } \mathbf{C} \\ \hline \\ \hline \\ \texttt{class } \mathbf{C} \ \texttt{extends } \mathbf{D} \ \{\overline{\mathbf{C}} \ \overline{\mathbf{f}}; \ K \ \overline{\mathbf{M}}\} \ \mathsf{OK} \end{split}$$



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!

This model accounts for extensions that the model does not actually mention. We are studying the core of a *language family*.

Java conditionals

$$\frac{\mathtt{t}_1\in\mathtt{bool}\qquad \mathtt{t}_2\in\mathtt{T}_2\qquad \mathtt{t}_3\in\mathtt{T}_3}{\mathtt{t}_1~?~\mathtt{t}_2~:~\mathtt{t}_3\in?}$$

Java conditionals

More standard (declarative) rule:

 $\frac{\mathtt{t}_1\in\mathtt{bool}\quad \mathtt{t}_2\in\mathtt{T}\quad \mathtt{t}_3\in\mathtt{T}}{\mathtt{t}_1~?~\mathtt{t}_2~:~\mathtt{t}_3\in\mathtt{T}}$

Java conditionalsMore standard (declarative) rule: $\frac{t_1 \in bool \quad t_2 \in T \quad t_3 \in T}{t_1 ? t_2 : t_3 \in T}$ Algorithmic version: $\frac{t_1 \in bool \quad t_2 \in T_2 \quad t_3 \in T_3}{t_1 ? t_2 : t_3 \in T_2 \lor T_3}$ Requires joins!

Java has no joins

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

E.g.:

```
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.

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

 $E ::= \begin{bmatrix} \\ & E \\ & m(\overline{t}) \\ & v \\ & m(\overline{v}, E, \overline{t}) \\ & new \ C(\overline{v}, E, \overline{t}) \\ & (C) E \end{bmatrix}$

evaluation contexts hole field access method invocation (rcv) method invocation (arg) object creation (arg) cast

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

Progress

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

Proof: Straightforward induction.

Progress

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

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: Standard substitution lemma, and then straightforward induction.

Preservation

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

Proof: Standard substitution lemma, and then straightforward induction. *???*

Preservation?

Preservation?

Surprise: well-typed programs *can* step to ill-typed ones! The substitution lemma does not hold.

(How?)

Preservation?

Surprise: well-typed programs *can* step to ill-typed ones! The substitution lemma does not hold.

(How?)

(A)(Object)new B() \longrightarrow (A)new B()

Solution: "Stupid Cast" typing rule

Add another typing rule, marked "stupid" to indicate that an implementation should generate a warning if this rule is used.

 $\frac{\Gamma \vdash t_0 : D \quad C \not\leq D \quad D \not\leq C}{stupid \ warning}$ $\frac{\Gamma \vdash (C)t_0 : C}{\Gamma \vdash C}$

Solution: "Stupid Cast" typing rule

Add another typing rule, marked "stupid" to indicate that an implementation should generate a warning if this rule is used.

 $\frac{\Gamma \vdash t_0 : D \quad C \not\leq D \quad D \not\leq C}{\frac{stupid \ warning}{\Gamma \vdash (C)t_0 : C}} \qquad (7)$

(T-SCAST)

(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.

Alternative approaches to casting

- Loosen preservation theorem
- Use big-step semantics

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.
- 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—e.g., it provides a rigorous way of judging counterexamples. (Cf. "conservative extension" between logics.)