From Simple Types to Objects

6.894 (OODL) Spring 2001

Gregory T. Sullivan
MIT AI Lab

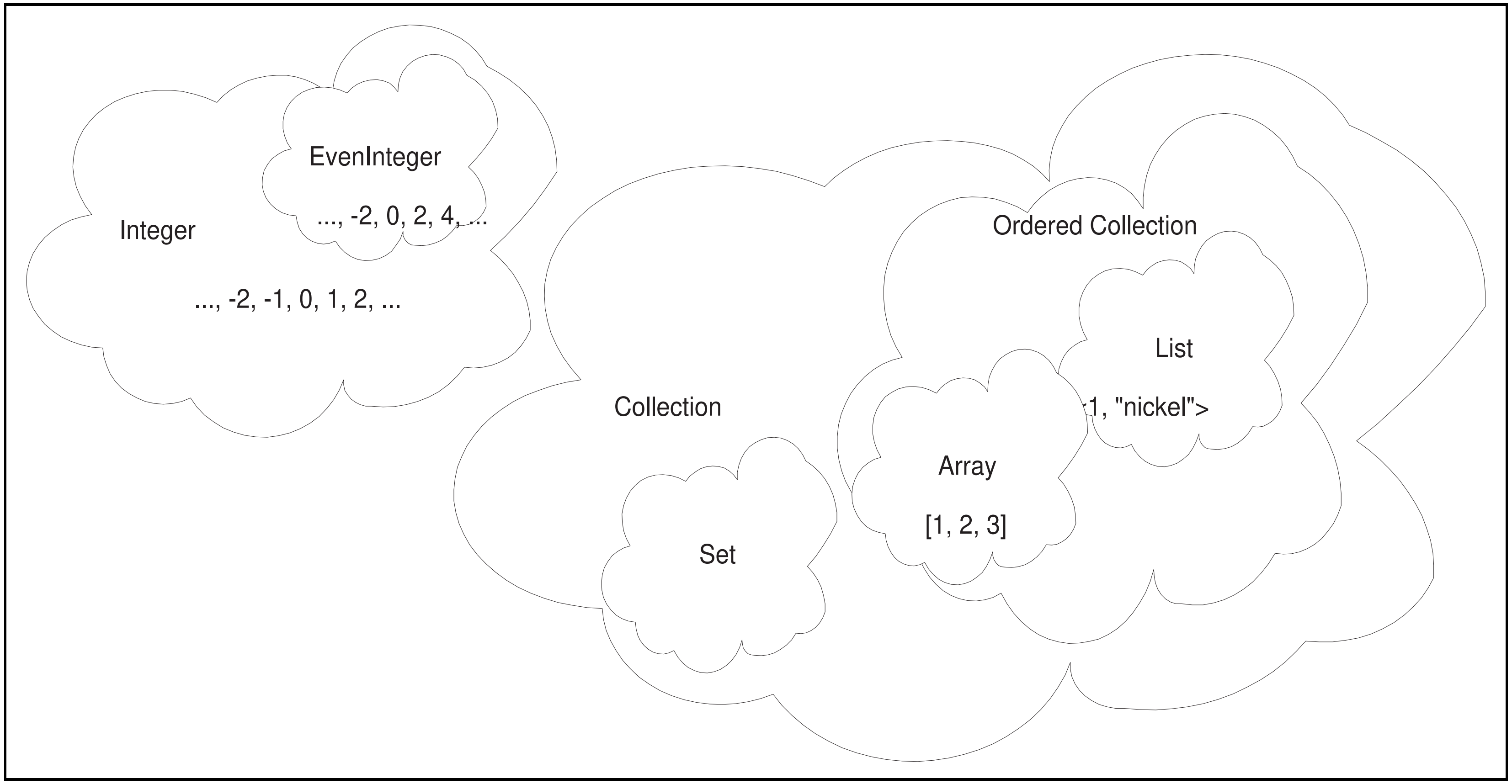
Overview

We will rapidly cover a lot of material, skipping over vast chasms of theory.

- What are Types?
- Why Types?
- Features for OO Languages:
  - Functions
  - Records
  - Subtyping

What are Types?

Types are abstractions of runtime values.



Why Types?

- Safety:
  "Well-typed programs cannot go wrong." – Robin Milner

```
int fun foo (int x) {
  // type checker will detect error at compile time
  return x + ``hello'';
}
```

- Documentation, Abstraction

```
int fun foo (Collection c1, Collection c2) {
  ... append(c1, c2) ...
}
```

- Efficiency

```
int fun foo (int x, int y) {
  x + y;  // can use integer + here
}
```

Types are used to rule out certain classes of runtime errors.

Type Inference

Is the following code type-correct?

```
fun (x, y) {
  x + y;
}
```

Hard to say.

How about now:

```
fun (int z) {
  let f = fun (x, y) {
            x + y;
          }
  in {
    f(z, 4);
  }
}
```

Absolutely!
Infer first of all that f does not escape. Then note all calls (only 1) to f are known, and know that all arg.s are integer.

Challenges (Limits)

```
bool fun foo (int x) { (x > 5) | (x < 15) }
int fun bar (int x) {
  if (foo(x))
    { x + 2; }
  else
    { ``a string''; } }
```

`foo` will always return true, so the else branch of the conditional in `bar` will never be executed, so bar should be type correct. How about:

```
foo =
  let i = 1 in
  fun (x) {
    i := i + 1;
    if (i mod 2) = 0 {
      string_append(x, ``there'');
    } else {
      x * 2; }}
```

On the first, third, fifth, etc. time it is called, `foo` has the type `int->int`. On the second, fourth, etc. time it is called, `foo` has the type `string->string`. This sort of type is very difficult to state precisely in any current type system.

Type Regimes and Programming Languages

Strongly Typed: program behavior is well-defined. This includes ML, Scheme, Dylan, Java, but not C, C++.

Statically Typed: (ML, Java) a certain amount of type checking is done at compile time. Rules out certain runtime errors, such as "no such method". How about "division by 0", array bounds checking, etc.? Upside: more confidence, errors caught sooner, less need for tagged values. Downside: rules out some correct programs (as in the preceding slide).

Dynamically Typed: almost all checking is done at runtime. Upside: more flexible. Downside: late detection of errors, need runtime type information (tags).

* note that statically checked strongly typed languages may still include dynamic type checks. In Java, cast downs. But also array bounds checks, div. by 0, etc.

A Simple Language Of Booleans and Integers

Reduction Rules (aka *Dynamic Semantics*)

$$\frac{e_0 \rightarrow e_0'}{\textsf{if}\ e_0\ \textsf{then}\ e_1\ \textsf{else}\ e_2 \rightarrow \textsf{if}\ e_0'\ \textsf{then}\ e_1\ \textsf{else}\ e_2}$$

$\textsf{if true then}\ e_1\ \textsf{else}\ e_2 \rightarrow e_1$

$\textsf{if false then}\ e_1\ \textsf{else}\ e_2 \rightarrow e_2$

$$\frac{e_1 \rightarrow e_1'}{e_1\ +\ e_2 \rightarrow e_1' + e_2} \qquad \frac{e_2 \rightarrow e_2'}{e_1\ +\ e_2 \rightarrow e_1 + e_2'}$$

$m + n \rightarrow s$ where s is the sum of integers m and n.

$m < n \rightarrow b$ where b is $\textsf{true}$ if integer m is less than integer n, and $\textsf{false}$ otherwise.

reduction is nondeterministic in choice of operand to +

A Simple Language, Continued

Typing Rules (aka *Static Semantics*)

$\mathsf{true}\ : Bool \qquad \mathsf{false}\ : Bool \qquad \dfrac{e_0\ : Bool \quad e_1\ :\ \tau \quad e_2\ :\ \tau}{\mathsf{if}\, e_0\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2\ :\ \tau}\ \text{(cond)}$

$$\frac{e_1\ : Int \quad e_2\ : Int}{e_1\ +\ e_2\ : Int}\ \text{(plus)} \qquad \frac{e_1\ : Int \quad e_2\ : Int}{e_1\ <\ e_2\ : Bool}\ \text{(lt)}$$

$n\ : Int$ for all integer literals n

Can we do better (or different) on conditional? Union types. If subtyping (and all joins), could use LUB.

Simple Types, continued

Safety, or **Soundness** property:

For a given term e, if $e\ :Int$, then there exists some integer n such that $e \xrightarrow{*} n$.

That is, terms don't get "stuck".

Example: Derivation of $(\mathsf{if}(3 < 4) 2\, \mathsf{else}\, 3) + 5\ :Int$

$$\dfrac{\dfrac{\dfrac{3\ :Int \quad 4\ :Int}{3 < 4\ :Bool} \qquad 2\ :Int \quad 3\ :Int}{\mathsf{if}(3 < 4) 2\, \mathsf{else}\, 3\ :Int} \qquad 5\ :Int}{(\mathsf{if}(3 < 4) 2\, \mathsf{else}\, 3) + 5\ :Int}$$

Functions, and Function Types

Grammar for types: $T ::= \mathsf{int} \mid \mathsf{bool} \mid T \to T$

Typing: (now includes type assumptions $A : \mathit{ID} \to \mathit{Type}$)

$$\frac{A[x : T'] \vdash e : T}{A \vdash \mathsf{fun}(x : T')\{e\} : T' \to T} \text{ (abs)} \qquad A[x : T] \vdash x : T \text{ (var)}$$

$$\frac{A \vdash e_2 : T' \quad A \vdash e_1 : T' \to T}{A \vdash e_1(e_2) : T} \text{ (app)}$$

Reduction:

$$\frac{e_1 \to e_1'}{e_1(e_2) \to e_1'(e_2)} \qquad \frac{e_2 \to e_2'}{\mathsf{fun}(x : T)\{e_0\}(e_2) \to \mathsf{fun}(x : T)\{e_0\}(e_2')}$$

$$\mathsf{fun}(x : T)\{e_0\}(V) \to e_0[x \mapsto V] \qquad (\beta)$$

"Value" terms are now distinguished by syntactic class V, which contains integers, true, false, and function terms.

Exercise:

```
fun(f : int -> int){ f(3) } (fun(x : int){ x + 4 }) : ?
```

Why need type assumptions? (because functions introduce bindings).
Note that this is call-by-value as opposed to call-by-name (or lazy) reduction.
Have we maintained safety?

```
x:int |- x + 4 : int             f:int->int |- f(3) : int
--------------------             ------------------------------
fun(x : int){ x + 4 } : int->int  fun(f : int -> int){ f(3) } : (int->int)->int
-----------------------------------------------------------------------------
fun(f : int -> int){ f(3) } (fun(x : int){ x + 4 }) :
```

Records, and Record Types

Motivation: objects are records, with methods as record elements.

`{x:int = 3; y:int = 4; setx = fun(x2:int).  x := x2}`

Has record type:

`{x :  int; y :  int; setx :  int->void}`

What if we want to return the updated record from `setx`? What is the type?

`{ ...; setx = fun(this, newx:int).  x := newx ; this }`

Has *recursive* type:

`rec X . {x :  int; y :  int; setx :  (X, int)->X}`

Another option is to use a notion of *self type*.

Subtyping

Principle of Safe Substitution:

> If S is a subtype of T, then any term of type S may safely be used in a context where a term of type T is required.

Suppose we add the type $Nat \leq Int$ to our set of basic types. The following should typecheck:

```
fun foo (x : Int) { x + 2; }

fun bar (x : Nat) { foo(x); }

Nat i = 4;
bar(i);
```

New typing rules:

$$\frac{n \geq 0}{n\ : Nat} \qquad \frac{e\ :\ S \quad S \leq T}{e\ :\ T}\ \text{(subsumption)}$$

Subtyping, continued

Typechecking body of `fun bar (x:Nat){foo(x);}`:

$$\dfrac{\dfrac{\cdots}{A \vdash foo : Int \to Int} \qquad \dfrac{A[x : Nat] \vdash x : Nat \quad Nat \leq Int}{A[x : Nat] \vdash x : Int}}{A[x : Nat] \vdash foo(x) : Int}$$

A function should operate on subtypes of its declared argument types.

parametric polymorphism: The same code running on arguments of different types.

ad-hoc polymorphism: Different code being invoked depending on the argument types (ex: +). Leads to *generic functions* and *multimethod dispatch*.

Subtyping Functions

When is a function

$$f_1 = \mathsf{fun}(x : s)\{\ldots\} : s \to s'$$

substitutable for a function

$$f_2 = \mathsf{fun}(x : t)\{\ldots\} : t \to t'$$

That is, when can we replace a call $f_2(v)$ with a call $f_1(v)$? Clearly $s' \leq t'$; that is, f_1 must return subtypes of what f_2 returns. For argument types, f_1 must accept everything that f_2 accepts (and possibly more), so $t \leq s$. To be precise,

$$\frac{t \leq s \quad s' \leq t'}{(s \to s') \leq (t \to t')}$$

Function subtyping is *contravariant* in the argument position.

Typechecking with Subtyping

Example: Assume $S \leq T$

```
void foo(x : S){...}
T bar(x : T){x}

S y = new S();
foo(bar(y));  // is bar(y) guaranteed to be of type S?
```

$$\dfrac{\dfrac{\dfrac{y : S \vdash y\ :\ S \quad S < T}{y : S \vdash y\ :\ T} \quad \dfrac{\cdots}{\mathsf{fun}(x\ :\ T)x\ :\ T \to T}}{y : S \vdash bar(y)\ :\ T} \quad \mathsf{fun}(x\ :\ S)\ldots\ :\ S \to ?}{y : S \vdash foo(bar(y))\ :?}$$

Problem:(loss of information) We can't legally consider `bar` to be of type $S \to S$, even though for this use, it is.

Reminder: this comes up all the time in OO programs!

How to regain information?

Several choices:

- More complex type systems: we want to say that

  `fun bar(x : T){x}`

  has type $\forall S \leq T.S \rightarrow S$. So-called "2nd order type systems", such as $F_{\leq}$. Note that type inference for $F_{\leq}$ is undecidable.

- Insert dynamic casts / type checks.

  `foo(as S bar(y))`

- Polyvariant Type Inference: Instead of analyzing the body of a function once, with its variable typed as declared, analyze the body of a function many times – maybe once per call site, with the variable typed with the argument type. Ex: Agesen's CPA analysis [Age96]

Agesen's Cross Product Algorithm

Summary: a method is analyzed for each possible tuple of *concrete* argument types. For example, if the class hierarchy looks like:

```
   A       D
 B   C   E   F
```

And suppose for a given call site `x.f(y)`, $x : A$, $y : D$ and there are two applicable methods, f_1 and f_2. Then the bodies of both f_1 and f_2 would be analyzed four times each – once for each pair of concrete types $\{(B, E), (B, F), (C, E), (C, F)\}$. To avoid redundant re-analysis, a cache of typings is kept for each method.

Benefits: Enables inlining, customization (method cloning).

Disadvantages: Possibly exponential.

Subtyping Records

A record of type r is substitutable for a record of type r' if type r has all of the fields (and possibly more) or type r' and all those fields agree on their types. More formally:

$$\frac{\{l'_1 \ldots l'_m\} \subseteq \{l_1 \ldots l_m\} \qquad t_i \leq t'_i \quad \text{for } 1 \leq i \leq m}{\{l_1 : t_1; \ldots; l_n : t_n\} \ \leq \ \{l'_1 : t'_1; \ldots; l'_m : t'_m\}}$$

This assumes some sort of canonical (e.g. lexicographic) ordering on labels.

Binary Methods

Consider two class types, given as recursive record types:

$$Point = \{x : Int; y : Int; equal : Point \to Bool\}$$
$$ColorPoint = \{x : Int; y : Int; c : String; equal : ColorPoint \to Bool\}$$

equal is a *binary method.*

One would hope that $ColorPoint \leq Point$, but they are not, as the type of *equal* in *ColorPoint* is not a subtype of the type of *equal* in *Point*.

This is not a problem with the subtyping rules – they are all required for soundness.

Binary Method Example

The canonical ex. of how things go bad: Suppose the type system did allow that ColorPoint <= Point.

```
foo (Point p1) {
  Point p2 = new Point();
// will crash when p1 (a colorPoint) asks for p2's color:
  p1.equal(p2);
}
main () {
  foo(new ColorPoint());
}
```

Solutions to Binary Method Problem

C++: Do not allow argument type to change in subclass. That is, *ColorPoint* must be defined as:

$ColorPoint = \{x : Int; y : Int; c : String; equal : Point \rightarrow Bool\}$

Use Functions instead of Methods: This is similar to Java's static methods. Also very common in C++.

Double Dispatching: [Ingalls 86] Modern version is the "visitor" pattern [Gamma et al.].

Generic Functions and Multimethod Dispatch: as in CLOS, Dylan. Yeah!

More Expressive Type Systems: Matching [Bruce], 2nd order type systems. More complex types spawn need for type inference.

Mutable Records and Subtyping

Suppose we have two record types:

$$r_1 = \{l : T\}$$
$$r_2 = \{l : S\}$$

$$S \leq T \implies r_2 \leq r_1$$

Now think about:

```
bool fun f (x : r_1) {
  x.l = a-T;
  true;
}
... f(an-r_2) ...
```

Certainly r_1's are fine as arguments to f. So the fact that an instance of r_2 is not OK means that substitutivity is violated, thus r_2 could not be a subtype of r_1.

The reason is that we've hidden the "setter" signatures, which expose the violation.

Hugh Greene reminds me: In Dylan, Limited(Collection, of: S) $\leq$ Limited(Collection, of: T), and we have the same problem.

Java Subtyping

Suppose $B < A$ in:

```
class C extends Object {
    A x = new A(); void setx(A x2) { x = x2; } }
class D extends D      {
    B x = new B(); void setx(B x2) { x = x2; } }
```

What actually happens in Java?

```
class Test2 extends Object {
    public static void foo (C c1) {
        c1.setx(new A());
    }
    public static void main (String[] args) {
        D aD = new D();
        foo(aD);
        System.out.println(aD.x.getClass());
    }
} // class Test2
```

Prints “class B”.

Java Subtyping Restrictions

In Java, the following gets an error, with B a subclass of A:

```
class C extends Object {
    A x = new A();
    A getx () { return(x); }
}
class D extends C {
    B x = new B();
    B getx () { return(x); }
}
```

gets

```
D.java:3: getx() in D cannot override getx() in C;
    attempting to use incompatible return type
found   : B
required: A
    B getx () { return(x); }
      ^
```

Why is that?

Adding Polymorphism to Types

Motivation: Polymorphic lists. Universal quantification (abstraction) over type variable, System F (Girard 72, Reynolds 74). ML.

cons: $\forall X \,.\, (X \times \mathit{List}(X)) \to X$, isnil: $\forall X \,.\, \mathit{List}(X) \to \mathit{bool}$
head: $\forall X \,.\, \mathit{List}(X) \to X$, tail: $\forall X \,.\, \mathit{List}(X) \to \mathit{List}(X)$

Concrete syntax can be a bit cumbersome:

```
map = λX. λY.
        λf:X → Y.
            (fix λm:(List X) → (List Y).
                     λl:List X .
                         if isnil [X] l
                            then nil [Y]
                            else cons [Y] (f (head [X] l))
                                          (m (tail [X] l))))
            as (List X) → (List Y);
```

Note that type reconstruction is undecidable in System F.

Polymorphism with Subtyping and records

Motivation (from Benjamin Pierce's manuscript) – simple universal quantification is not enough.

Write a function that takes record containing `a` and returns original record plus next `a`:

```
f = λx:{a:Nat} .  {orig=x, asucc=succ(x.a)};
```

```
⇒{a:Nat} → {orig:{a:Nat}, asucc:Nat}
```

If we apply it to a record `rab = {a=0, b=true}`,

```
f rab
```

```
⇒{orig={a=0,b=true}, asucc=1}:{orig:{a:Nat}, asucc:Nat}
```

we lose the `b` field in the result type. We can't solve this with universal quantification:

```
fpoly = λX.λx:X. {orig=x, asucc=succ(x.a)};
```

will not typecheck

$\underline{F_<}$

$F_<$, Cardelli and Wegner, 1985. Bounded quantification in CLU 1977.

Can write:

```
fpoly = λX < {a:Nat} .  λx:X. {orig=x, asucc=succ(x.a)};
```

$\Rightarrow$ `∀ X < {a:Nat} .  X → {orig:X, asucc:Nat};`

The subtyping rule for quantified types is analagous to that for functions:

$$\frac{A \vdash T_1 < S_1, \quad A, X < T_1 \vdash S_2 < T_2}{A \vdash (\forall X < S_1 \,.\, S_2) < (\forall X < T_1 \,.\, T_2)}$$

If the 2nd premise gives you pause, it should. In fact it renders subtyping undecidable (Pierce, 94)

Not There Yet

Motivation (from Mitchell FOPL 96,98):

`type ordered =` $\{$`less: ordered` $\rightarrow$ `bool`$\}$

Note that type is recursive, unlike $\{$`a:Nat`$\}$ from previous ex.

Can write in $F_<$ a sorting function with type:

`sort:` $\forall$ `T < ordered . List(T)` $\rightarrow$ `List(T)`

But this isn't as useful as we'd like, because it's hard to define subtypes of `ordered`!

`type string =` $\{$`less : string` $\rightarrow$ `bool, concat: string` $\rightarrow$ `string, ...` $\}$

is not a subtype of `ordered` because of our old nemisis, contravariant subtyping of function types.

F-bounded Subtype Polymorphism

Type bounds given by a function (F) on types.

Ex. (from Mitchell FOPL), the two named type functions:

`type ordered =` $\{$`less :  ordered` $\rightarrow$ `bool`$\}$

`type string =` $\{$`less:  string` $\rightarrow$ `bool, ...`$\}$

Define `F` from types to types:

`F =` λ`T .` $\{$`less:  T` $\rightarrow$ `bool`$\}$

Now we can write a sort function with the type:

`sort :` $\forall$ `T < F(T) . List(T)` $\rightarrow$ `List(T)`

Check that `string < F(string)`:

`F(string) =` $\{$`less :  string` $\rightarrow$ `bool`$\}$

and clearly `string` $\leq$ `F(string)`.

See [LTS01] for the use of F^{ω} (no subtyping) as the basis for a typed intermediate language.

More Types

Intersection & Union types – see Church group at BU.

Existential Types – applications to data hiding (ADTs, modules).

Still to Come

Typechecking Multimethods

Predicate Dispatching

Reading material

[Mit96] – advanced coverage of typed λ-calculi, semantics, and types.

[AC96] – lots of calculi.

[Cas97] – includes a formal theory of multiple dispatch.

[PS93] – accessible, if incomplete.

[GM94] – especially record-based typing.

Any papers by Cardelli, B. Pierce, etc. Conferences: ECOOP, POPL, FOOL, TIC workshop, TLCA.

To Do

* examples of each binary method solution

* generic/parameterized types

* representations of types at runtime. Accessing types of objects. While static checking may enable removing type info., may need it for debugging anyway.

* other, more esoteric, reasons to keep type info around at runtime: mobile code type checking.

References

[AC96] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer-Verlag, New York, 1996.

[Age96] Ole Agesen. *Concrete Type Inference: Delivering Object-Oriented Applications*. Ph.D. thesis, Stanford University, December 1996.

[Cas97] Giuseppe Castagna. *Object-Oriented Programming: A Unified Foundation*. Progress in Theoretical Computer Science. Birkhauser, Boston, 1997.

[GM94] Carl A. Gunter and John C. Mitchell, editors. *Theoretical Aspects of Object-Oriented Programming*. Fondations of Computing. The MIT Press, Cambridge, MA, 1994.

[LTS01] Christopher League, Valery Trifonov, and Zhong Shao. Type-preserving compilation of featherweight java. In *Foundations of Object-Oriented Languages (FOOL8)*, London, January 2001.

[Mit96] John C. Mitchell. *Foundations for Programming Languages.* MIT Press, Cambridge, MA, 1996.

[PS93] Jens Palsberg and Michael I. Schwartzbach. *Object-Oriented Type Systems.* John Wiley, 1993.