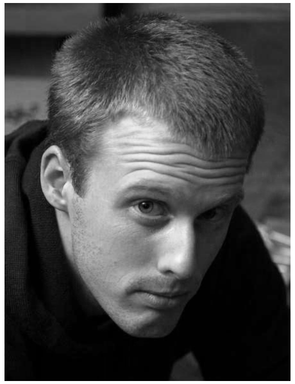
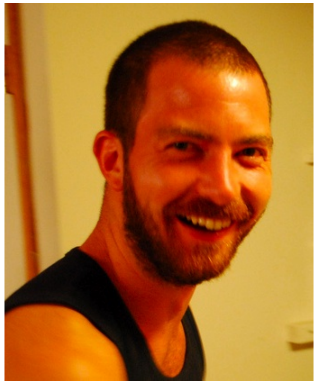
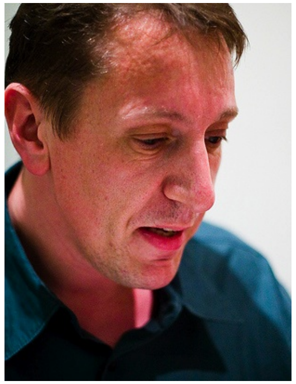
Like Types

aka. integrating typed and untyped code in Thorn

Francesco Zappa Nardelli

INRIA Paris-Rocquencourt, MOSCOVA research team

Tobias Wrigstad | Sylvain Lebresne | Johan Östlund | Jan Vitek

Purdue University

ParSec   June 4, 2009

“Scripting” languages are:

1. maximally permissive: *anything goes, until it doesn’t*;

2. maximally modular: *a program can be run even when crucial pieces are missing*;

These features enable *rapid prototyping* of software.

Perl, Python, Ruby, JavaScript, etc... are widely used.

Some scripting languages features

- Return objects of different types depending on some value;
- methods can take arguments of different types;

  `fun typeMe (x,y) -> if x then y + 1 else y ^ "hola";`

- overloading of `method_missing`
  (in db, regexps on the method name to implement different queries);
- changing classes at run-time (add or delete a method, modify inheritance);

Remark: these are *inherently hard to type*.

Remark: prototypes are often used as production code

In production code, *types would be useful*:

- untyped code is hard(er) to navigate;
- higher loads of data make speed a pressing issue.

Common approach:

- rewrite the untyped program in a statically typed language (e.g., C++, Java).

Better:

incremental addition of type annotations (or module-by-module migration).

(Untyped) Point

A `Point` declaration in *Thorn** (a new scripting language from Purdue and IBM):

```
class Point(var x, var y) {
  fun getX() = x;
  fun getY() = y;
  fun move(p) { x := p.getX(); y := p.getY() }
}
```

(`x` and `y` are fields, and `Point` is both a class name and a trivial two argument constructor.)

```
o = Point(0,0);   # create a point
a = Point(5,6);   # create another point
a.move(o);        # move point a to point o
```

* IBM systematically choses ugly names to minimise the risk of copyright conflicts.

Partially typed point

Suppose that we want to annotate `Point` to make the coordinates integers:

```
class Point(var x : Int, var y : Int) {
  fun getX() : Int = x;
  fun getY() : Int = y;
  fun move(p) { x := p.getX(); y := p.getY() }
}
```

We want the method `move` to accept any object, with the hope that if the actual object provides `getX` and `getY` method that return integers, the program *should* run just fine...

Extensive literature? Short (and partial) review.

The type systems of *Strongtalk* (Bracha and Griswold), *TypePlug* (Haldiman et al.), *BabyJ* (Anderson and Drossopoulou), $\mathbf{Ob}^{?}_{<:}$ (Siek and Taha), leave us with two options:

1. omit the type of `p`: flexible but unhelpful;

2. type `p` as `Point`: safe but inflexible. For instance, it forbids:

```
class Coordinate(var x: Int, var y: Int) {
  fun getX(): Int = x;
  fun getY(): Int = y;
}
```

```
p = Point(0,0);
c = Coordinate(5,6);
p.move(c)
```

Structural subtyping

Strongtalk, TypePlug, and $\mathbf{Ob}^{?}_{<:}$, support *structural subtyping*.

Apparently quite flexible: if `p:Point`, then any object that *structurally conforms* to `Point` can be passed as an argument to `move`.

But `Coordinate` is not a structural subtype of `Point`. Solution: invent more general types e.g.

```
class XY {
  fun getX(): Int;
  fun getY(): Int;
}                                           fun move (p:XY) { ... }
```

Result on large programs: large family of types that must be kept in synch and *have no meaning to the programmer*.

Soft typing

Idea (Cartwright and Fagan, 1991):

> *infer the minimal constraints* (similar to the class `XY`), and either warn (and insert the appropriate run-time check) or reject the program.

Problems:

- requires structural subtyping or a complete subtype hierarchy;
- a typo in a method name generates a bogus constraint (hard to debug);
- no help from IDEs;
- compile-time optimisations hard.

Gradual typing

Idea (Siek and Taha, 2006):

> whenever we go from untyped to typed code, *insert the appropriate cast*.

For instance, the last line of the program

```
class Foo { fun bar(x: Int) x*x; }
f:Foo = Foo();
f.bar(xyzzy); # does not type check
```

is compiled as `f.bar((Int) xyzzy)`.

Doubt: what do casts do at runtime?

Gradual typing and run-time wrappers

```
class Ordered {fun compare(o:Ordered):Int;}
class SubString {fun sub(o:String):Bool;}
fun sort(x: [Ordered]):[Ordered] = ...
fun filter(x: [SubString]):[SubString] = ...
```

- Testing that an object has type `[Ordered]` is done in *linear time*;
- arrays are *mutable*: checking the type at the beginning of `sort` is not enough.

Only option: enclose datas in *run-time wrappers*:

```
fun  plentyOfWrappers ( f: dyn ) {
  f':[SubString] = filter(sort(f));
  # f' = ([SubString])([Ordered])f
  v:SubString = f'[0];
  # v =  (Substring)(Ordered)f'[0] }
```

Our design principles

1. Permissive: *try to accept as many programs as possible*;

2. Modular: *be as modular as possbble*;

3. Reward good behaviour:

programmer effort rewarded either with performance
or clear correctness guarantee.

Like types

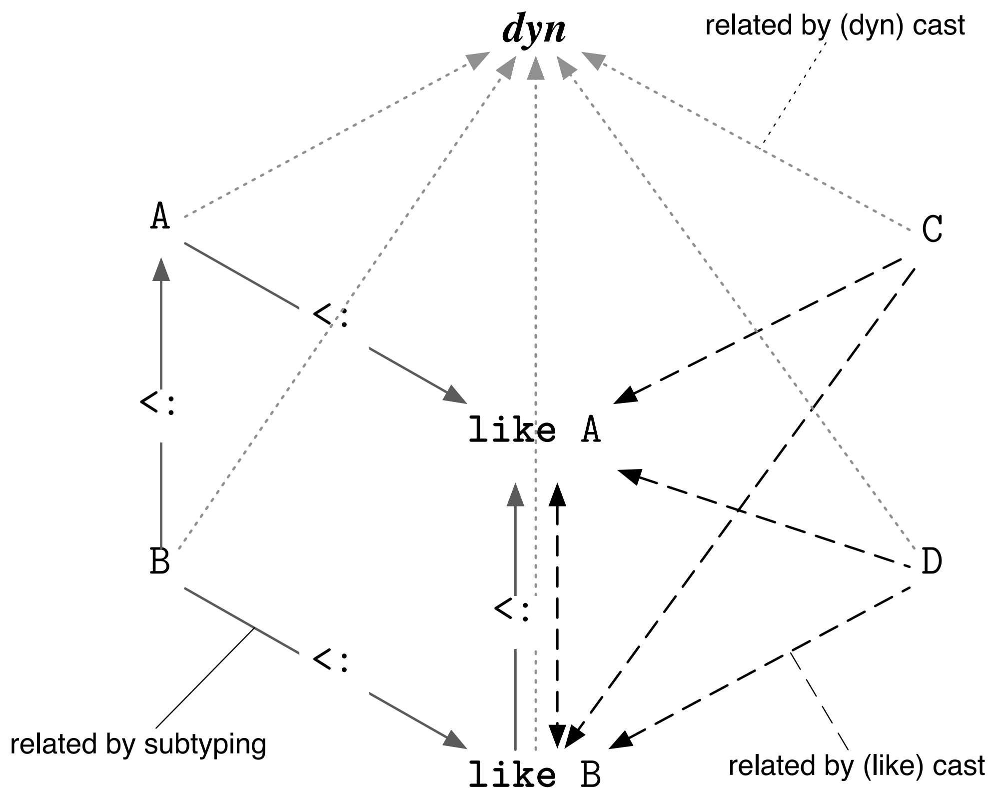
- For each class name `C`, introduce a `like C` type;
- the compiler checks that all *operations* on an object of type `like C` are well-typed if the object had type `C`;
- the run-time does not restrict binding of variables of type `like C` and checks at run-time that the invoked method exists.

A well-typed example:

```
fun move(p: like Point) {
  x := p.getX(); # 1
  y := p.getY(); # 2
  # p.hog();     # 3 compile time error
}
```

```
p = Point (0,0);
c = Coordinate(5,6);
p.move(c)
```

Like types: the big picture



Like types

- *A unilateral promise as to how a value will be treated locally*;
- allows most of the regular static checking machinery;
- allows the flexibility of structural subtyping;
- *concrete types can stay concrete, so more aggressive optimisations are possible*;
- allow reusing type names as semantics tags;
- *interact nicely with generics*.

Wrapping untyped objects in like types

```
class Cell(var contents) {
  fun get() = contents;
  fun set(c) { contents := c }
}

class IntCell {
    fun get():Int;
    fun set(c:Int);
}

p: like IntCell = (like IntCell) Cell(0);
```

Sort-of simple union types

```
fun typeMe(a,b) {
  if (a)    # treat b as a Foo
  else      # treat b as a Bar
}

class Foo_Or_Bar extends Foo, Bar;

fun typeMe(a:bool, b:like Foo_Or_Bar) {
  if (a)    # treat b as a Foo
  else      # treat b as a Bar
}
```

Metatheory: miniThorn

Basically an imperative version of FJ, with classes and methods defined as:

$$\mathbf{class}\ C\ \mathbf{extends}\ D\ \{\ \mathit{fds}\ ;\ \mathit{mds}\ \}$$

$$t\ m\ (t_1\ x_1\ ..\ t_k\ x_k)\ \{\ s\ ;\ \mathbf{return}\ x\ \}$$

Let C range over class names. Types are defined as

$$t\ ::=\ C\ |\ \textsf{\textbf{like}}\ C\ |\ \mathbf{dyn}$$

and statements include method invocation and casts, denoted respectively as

$$x = y\,.\,m\,(y_1\,..\,y_n) \qquad \text{and} \qquad x = (t)\,y.$$

Typing of method invocation

$$\frac{\begin{array}{l}\Gamma \vdash y : C \;\vee\; \Gamma \vdash y : \textsf{\textbf{like}}\ C \\ \mathbf{mtype}(m,\ C) = t_1 \mathinner{..} t_k \rightarrow t' \\ \Gamma \vdash y_1 <: t_1 \quad .. \quad \Gamma \vdash y_k <: t_k \\ \Gamma \vdash x : t \\ t' <: t\end{array}}{\Gamma \vdash x = y\,.\,m\,(y_1 \mathinner{..} y_k)}$$

$$\frac{\begin{array}{l}\Gamma \vdash y : \mathbf{dyn} \\ \Gamma \vdash y_1 : t_1 \quad .. \quad \Gamma \vdash y_k : t_k \\ \Gamma \vdash x : \mathbf{dyn}\end{array}}{\Gamma \vdash x = y\,.\,m\,(y_1 \mathinner{..} y_k)}$$

If the target object has a concrete or like type, then the type of the actual arguments is statically checked against the method type. This check is not (cannot be) performed if the target object has a dynamic type.

Run-time state

Imagine that $x : C$, y : **like** D, and $z : \mathbf{dyn}$ are aliased to the same object at location p. An environment F records variables mapped to *stack-values* sv:

$$x \mapsto p \qquad y \mapsto (\textbf{like } D)p \qquad z \mapsto (\mathbf{dyn})p$$

A state of the run-time is defined by a heap H of locations mapped to objects

$$p \;\mapsto\; C(f_1 = sv_1; ..; f_n = sv_n)$$

and a stack S of activation records

$$\langle F_1 | s_1 \rangle ... \langle F_n | s_n \rangle \;.$$

Run-time invariants

1. Objects in the heap are always well-formed:

$$\frac{H(p) = D(...) \wedge D <: C}{\mathcal{T}_H(p) = C} \qquad \frac{D <: C}{\mathcal{T}_H((\textsf{like}\ D)p) = \textsf{like}\ C} \qquad \frac{}{\mathcal{T}_H((\mathbf{dyn})p) = \mathbf{dyn}}$$

$H(p) = C(f_1 = sv_1; ..; f_n = sv_n)$ implies $\mathcal{T}_H(sv_i) = \mathbf{ftype}(C, f_i)$.

2. Relation between static types, stack values, and heap:

Static type	Stack value	Object in the heap
C	p	$H(p) = D(...)$ and $D <: C$
like C	(**like** C)p	$H(p) = D(...)$
$\mathbf{dyn}$	$(\mathbf{dyn})p$	$H(p) = D(...)$

Semantics (1)

Method invocation on an object that statically has a concrete type C:

$$\frac{\begin{array}{l} F(y) = p \\ H(p) = C(...) \\ \textbf{mbody}\,(m,\ C) = x_1 \,..\, x_n \,.\, s_0\,;\, \textbf{return}\, x_0 \\ F(y_1) = sv_1 \quad .. \quad F(y_n) = sv_n \end{array}}{\begin{array}{l} H \mid \langle\, F \mid x = y\,.\,m\,(y_1 \,..\, y_n)\,;\ s \,\rangle\, S \longrightarrow \\ \quad H \mid \langle\, [\,]\,[x_1 \mapsto sv_1 \,..\, x_n \mapsto sv_n]\,[\mathit{this} \mapsto p] \mid s_0\,;\,\textbf{return}\, x_0 \,\rangle\, \langle\, F \mid x = \mathit{ret}\,;\ s \,\rangle\, S \end{array}}$$

Semantics (2)

Method invocation on an object that statically has type **like** C:

$$\frac{\begin{array}{l} F(y) = (\textbf{like}\ C)\ p \\ H(p) = D(...) \\ \textbf{mtype}\,(m,\ C) = \textbf{mtype}\,(m,\ D) \\ \textbf{mbody}\,(m,\ D) = x_1 \mathinner{..} x_n \,.\, s_0\ ;\ \textbf{return}\, x_0 \\ F(y_1) = sv_1 \quad .. \quad F(y_n) = sv_n \end{array}}{\begin{array}{l} H \mid \langle\, F \mid x = y\,.\,m\,(y_1 \mathinner{..} y_n)\ ;\ s\,\rangle\, S \ \longrightarrow \\ \quad H \mid \langle\, [\,]\,[x_1 \mapsto sv_1 \mathinner{..} x_n \mapsto sv_n]\,[\mathit{this} \mapsto p] \mid s_0\ ;\ \textbf{return}\, x_0\,\rangle\,\langle\, F \mid x = \mathit{ret}\ ;\ s\,\rangle\, S \end{array}}$$

Semantics (3)

Method invocation on an object that statically has type **dyn**:

$$\frac{\begin{array}{l} F(y) = (\mathbf{dyn})\, p \\ H(p) = C(\ldots) \\ \mathbf{mtype}\,(m,\ C) = t_1 \mathinner{..} t_n \ \rightarrow\ t \\ \mathbf{mbody}\,(m,\ C) = x_1 \mathinner{..} x_n \,.\, s_0\,;\, \mathbf{return}\, x_0 \\ F(y_1) = sv_1 \quad \mathinner{..} \quad F(y_n) = sv_n \\ \mathcal{T}_H\,(sv_1) <:\ t_1 \quad \mathinner{..} \quad \mathcal{T}_H\,(sv_n) <:\ t_n \end{array}}{\begin{array}{l} H \mid \langle\, F \mid x = y\,.\,m\,(y_1 \mathinner{..} y_n)\,;\ s \,\rangle\, S \ \longrightarrow \\ \qquad H \mid \langle\, [\,]\,[x_1 \mapsto sv_1 \mathinner{..} x_n \mapsto sv_n]\,[\mathit{this} \mapsto p] \mid s_0\,;\, \mathbf{return}\, x_0 \,\rangle\, \langle\, F \mid x = (\mathbf{dyn})\, \mathit{ret}\,;\ s \,\rangle\, S \end{array}}$$

Semantics (4)

The run-time does not need chains of wrappers, as it only needs to record the *static view* that a variable has of an object:

$$\frac{\begin{array}{c}\Gamma \vdash y : t_2 \\ \Gamma \vdash x : \textbf{like } C\end{array}}{\Gamma \vdash x = (\textbf{like } C)\, y}$$

$$\frac{F(y) = w\, p}{H \mid \langle\, F \mid x = (\textbf{like } C)\, y\, ;\, s\,\rangle\, S \longrightarrow H \mid \langle\, F\, [x \mapsto (\textbf{like } C)\, p] \mid s\,\rangle\, S}$$

Nice properties

Preservation the run-time invariant is preserved through reductions;

Progress if a program is stuck, then it attempted to execute $x = y.m(y_1, .., y_n)$ and $\Gamma(y) =$ **like** C or $\Gamma(y) = \mathbf{dyn}$, or *(...usual conditions on null-pointers and downcasts...)*.

Implementation without run-time wrappers

The run-time implements *three dispatch functions*:

$x = y.m(y_1, .., y_n)$ dispatch without any run-time type check;

$x = y.{}_{\textsf{like}\ C}\ m(y_1, .., y_n)$ check that the method m exists in the actual object, and has the type declared in C;

$x = y.{}_{\mathbf{dyn}} m(y_1, .., y_n)$ check that the method m exists in the actual object, and that the type of the arguments is compatible with the type of m.

Given an program and a type derivation, we compile method invocations to the appropriate dispatch function:

$$
\textbf{compile}\ y.m(y_1, .., y_n)\ \Gamma \;=\; \begin{array}{ll}
y.m(y_1, .., y_n) & \text{if } \Gamma(y) = C \\
y.{}_{\textsf{like}\ C}\ m(y_1, .., y_n) & \text{if } \Gamma(y) = \textbf{like}\ C \\
y.{}_{\mathbf{dyn}} m(y_1, .., y_n) & \text{if } \Gamma(y) = \mathbf{dyn}
\end{array}
$$

Correctness of compilation

Let σ_s range over well-typed states of the semantics and let σ_i range over well-typed states of the implementation.

We say that $\sigma_s \lhd \sigma_i$ if σ_i is obtained from σ_s by

1. erasing all the wrappers;

2. compiling all the statements that appear in all the stack frames.

Theorem.
If $\sigma_s \lhd \sigma_i$ and $\sigma_s \longrightarrow \sigma_s'$ then there exists a σ_i' such that $\sigma_i \longrightarrow \sigma_i'$ and $\sigma_s' \lhd \sigma_i'$.

Rewards

```
fun a(i, j) = 1.0 / (((i + j) * (i + j + 1) >> 1) + i + 1);
```

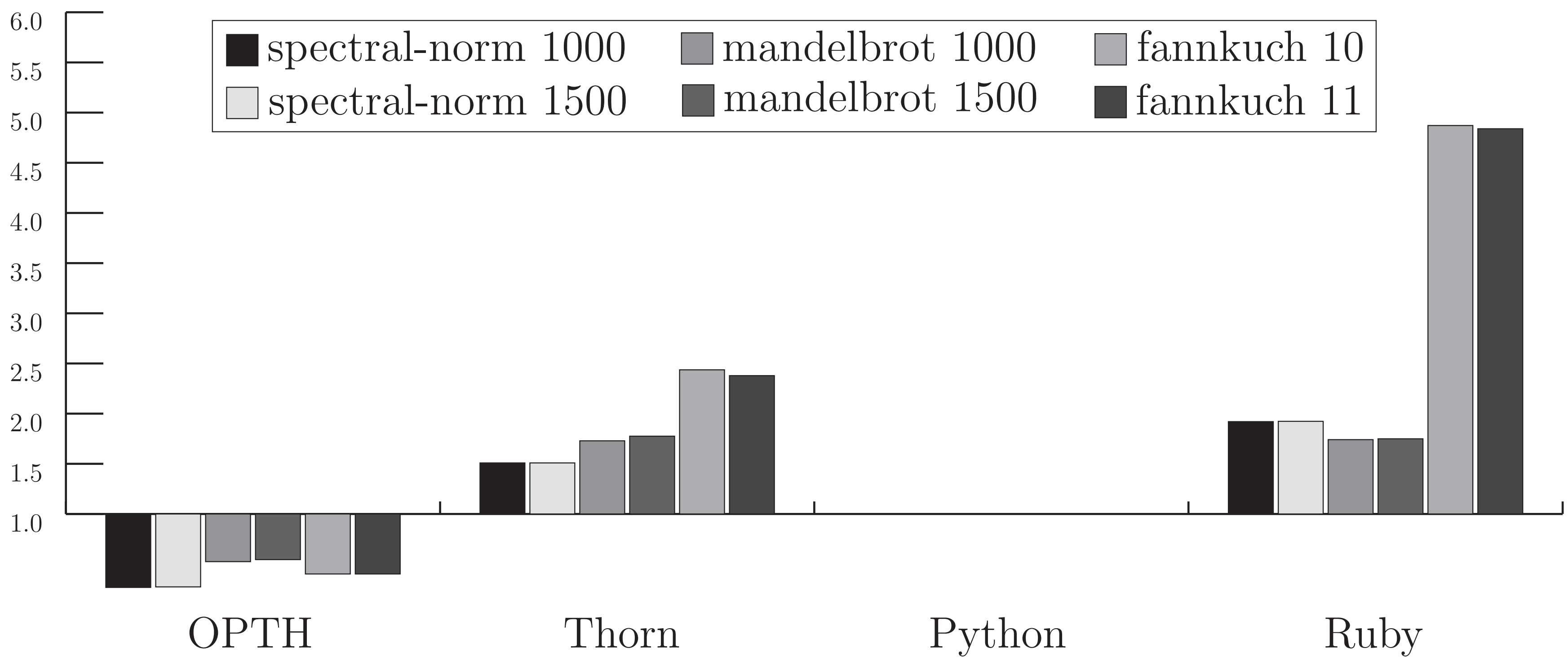
`i,j:dyn` 87 bytecode instructions, 8 new frames, 8 new objects

`i,j:Int32` 29 bytecode instructions, 0 new frames, 1 new object

`i,j:like Int32` 42 bytecode instructions, 3 new frames, 3 new objects

And in practice?

More rewards



Experience: porting *Pwiki* from Python to typed Thorn

- About 1000 lines of code and 1000 lines of libraries;
- at first, we typed all the function arguments with like types;
  it was always possible to run the program, even when only part of it had annotations
- then we strenghtened the annotations, using concrete types whenever possible;
  some parts of the code were left untyped
- found one error (a test `s < 10` where `s` is always string).

Conclusion

Like types represent a sweet spot in the design space of language features for incremental hardening of software.

Still not enough experience to draw strong conclusions.