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):

```java
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 chooses ugly names to minimise the risk of copyright conflicts.
Partially typed point

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

```kotlin
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), 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:

```kotlin
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 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.

```kotlin
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

```java
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

```kotlin
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 possible*;

3. Reward good behaviour:

   *programmer effort rewarded either with performance or clear correctness guarantee.*
**Like types**

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

A well-typed example:

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

\[ \text{p = Point (0,0);} \]
\[ \text{c = Coordinate(5,6);} \]
\[ \text{p.move(c)} \]
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

```kotlin
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);
```
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:

```plaintext
class C extends D { fds ; mds }

let tm (t₁ x₁ .. tₖ xₖ) { s ; return x }
```

Let $C$ range over class names. Types are defined as

```plaintext
t ::= C | like C | dyn
```

and statements include method invocation and casts, denoted respectively as

```plaintext
x = y . m (y₁ .. yₙ) and x = (t) y.
```
Typing of method invocation

\[\Gamma \vdash y : C \lor \Gamma \vdash y : \text{like } C\]
\[\text{mtype}(m, C) = t_1 .. t_k \rightarrow t'\]
\[\Gamma \vdash y_1 <: t_1 \ldots \Gamma \vdash y_k <: t_k\]
\[\Gamma \vdash x : t\]
\[t' <: t\]
\[\Gamma \vdash x = y . m(y_1 .. y_k)\]
\[\Gamma \vdash y : \text{dyn}\]
\[\Gamma \vdash y_1 : t_1 \ldots \Gamma \vdash y_k : t_k\]
\[\Gamma \vdash x : \text{dyn}\]
\[\Gamma \vdash x = y . m(y_1 .. y_k)\]

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

\[
\begin{align*}
  x & \mapsto p \\
  y & \mapsto (\textbf{like} \ D)p \\
  z & \mapsto (\textbf{dyn})p
\end{align*}
\]

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

\[
\begin{align*}
p & \mapsto C(f_1 = sv_1; \ldots; f_n = sv_n)
\end{align*}
\]

and a stack \( S \) of activation records

\[
\langle F_1|s_1\rangle \ldots \langle F_n|s_n\rangle .
\]
Run-time invariants

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

\[ H(p) = D(...) \land D <: C \quad \frac{D <: C}{\mathcal{T}_H(p) = C} \]

\[ \mathcal{T}_H((\text{like } D)p) = \text{like } C \quad \mathcal{T}_H((\text{dyn})p) = \text{dyn} \]

\[ H(p) = C(f_1 = s v_1; \ldots; f_n = s v_n) \text{ implies } \mathcal{T}_H(s v_i) = \text{ftype}(C, f_i). \]

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

<table>
<thead>
<tr>
<th>Static type</th>
<th>Stack value</th>
<th>Object in the heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>( C )</td>
<td>( p )</td>
<td>( H(p) = D(...) ) and ( D &lt;: C )</td>
</tr>
<tr>
<td>( \text{like } C )</td>
<td>( (\text{like } C)p )</td>
<td>( H(p) = D(...) )</td>
</tr>
<tr>
<td>( \text{dyn} )</td>
<td>( (\text{dyn})p )</td>
<td>( H(p) = D(...) )</td>
</tr>
</tbody>
</table>
Method invocation on an object that statically has a concrete type $C$:

\[
F(y) = p \\
H(p) = C(\ldots) \\
\text{mbody}(m, C) = x_1 \ldots x_n \cdot s_0; \text{return } x_0 \\
F(y_1) = sv_1 \ldots F(y_n) = sv_n
\]

\[
H | \langle F | x = y \cdot m(y_1 \ldots y_n); s \rangle S \rightarrow \\
H | \langle [] [x_1 \mapsto sv_1 \ldots x_n \mapsto sv_n] [\text{this} \mapsto p] | s_0; \text{return } x_0 \rangle \langle F | x = \text{ret}; s \rangle S
\]
Semantics (2)

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

\[
F(y) = (\text{like } C) \ p \\
H(p) = D(\ldots) \\
\text{mtype}(m, C) = \text{mtype}(m, D) \\
\text{mbody}(m, D) = x_1 \ldots x_n \ . \ s_0 \ . \ \text{return} \ x_0 \\
F(y_1) = s v_1 \ \ldots \ F(y_n) = s v_n
\]

\[
H \ | \ ( F \ | \ x = y \ . \ m \ (y_1 \ldots y_n) \ ; \ s ) S \rightarrow \\
H \ | \ ( [] [ x_1 \mapsto s v_1 \ldots x_n \mapsto s v_n ] [ \text{this} \mapsto p ] \ | \ s_0 ; \ \text{return} \ x_0 ) ( F \ | \ x = \text{ret} \ ; \ s ) S
\]
Method invocation on an object that statically has type \texttt{dyn}:

\[ F(y) = (\texttt{dyn}) p \]
\[ H(p) = C(\ldots) \]
\[ \texttt{mtype}(m, C) = t_1 \ldots t_n \rightarrow t \]
\[ \texttt{mbody}(m, C) = x_1 \ldots x_n \cdot s_0; \texttt{return} x_0 \]
\[ F(y_1) = sv_1 \ldots F(y_n) = sv_n \]
\[ T_H(sv_1) <: t_1 \ldots T_H(sv_n) <: t_n \]

\[
H \mid \langle F \mid x = y . m (y_1 \ldots y_n) ; s \rangle S \rightarrow \\
H \mid \langle [] [x_1 \mapsto sv_1 \ldots x_n \mapsto sv_n] [\texttt{this} \mapsto p] \mid s_0 ; \texttt{return} x_0 \rangle \langle F \mid x = (\texttt{dyn}) \texttt{ret} ; s \rangle S
\]
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:

\[
\begin{align*}
\Gamma & \vdash y : t_2 \\
\Gamma & \vdash x : \text{like } C \\
\hline
\Gamma & \vdash x = (\text{like } C) y
\end{align*}
\]

\[
F(y) = w \ p \\
\hline
H \ | \ \langle F \ | \ x = (\text{like } C) y ; s \rangle S \rightarrow H \ | \ \langle F [x \mapsto (\text{like } C) p] \ | \ s \rangle S
\]
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 $\Gamma(y) = \text{like } C$ or $\Gamma(y) = \text{dyn}$, or (...)usual conditions on null-pointers and downcasts...).
Implementation without run-time wrappers

The run-time implements *three dispatch functions*:

\[ x = y.m(y_1, \ldots, y_n) \] dispatch without any run-time type check;

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

\[ x = y.\text{dyn} \ m(y_1, \ldots, y_n) \] 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 a program and a type derivation, we compile method invocations to the appropriate dispatch function:

\[
\text{compile } y.m(y_1, \ldots, y_n) \ \Gamma = \begin{cases} 
  y.m(y_1, \ldots, y_n) & \text{if } \Gamma(y) = C \\
  y.\text{like } C \ m(y_1, \ldots, y_n) & \text{if } \Gamma(y) = \text{like } C \\
  y.\text{dyn} \ m(y_1, \ldots, y_n) & \text{if } \Gamma(y) = \text{dyn}
\end{cases}
\]
Correctness of compilation

Let \( \sigma_s \) range over well-typed states of the semantics and let \( \sigma_i \) range over well-typed states of the implementation.

We say that \( \sigma_s \prec \sigma_i \) if \( \sigma_i \) is obtained from \( \sigma_s \) by

1. erasing all the wrappers;
2. compiling all the statements that appear in all the stack frames.

**Theorem.**
If \( \sigma_s \prec \sigma_i \) and \( \sigma_s \rightarrow \sigma'_s \) then there exists a \( \sigma'_i \) such that \( \sigma_i \rightarrow \sigma'_i \) and \( \sigma'_s \prec \sigma'_i \).
Rewards

$$\text{fun } a(i, j) = \frac{1.0}{((i + j) \times (i + j + 1) \gg 1) + i + 1};$$

\begin{array}{l}
i, j: \text{dyn} \quad 87 \text{ bytecode instructions, 8 new frames, 8 new objects} \\
i, j: \text{Int32} \quad 29 \text{ bytecode instructions, 0 new frames, 1 new object} \\
i, j: \text{like Int32} \quad 42 \text{ bytecode instructions, 3 new frames, 3 new objects}
\end{array}

And in practice?
More rewards

- spectral-norm 1000
- spectral-norm 1500
- mandelbrot 1000
- mandelbrot 1500
- fannkuch 10
- fannkuch 11

Bar chart comparing performance across different languages and benchmarks.
Experience: porting \textit{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 strengthened 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.