Type systems. Why? WHY?

Jonathan Protzenko

jonathan.protzenko@inria.fr

Gallium (the nerds!)

INRIA Junior Seminar

Plan

1. Introduction
2. What is typing?
3. Let's do some math!
4. So what do I do?

Programming

Pretty much everyone has to do it (unfortunately).

Before programming

Young PhD student wants to write a numerical simulation.

(Real programmers use C++).

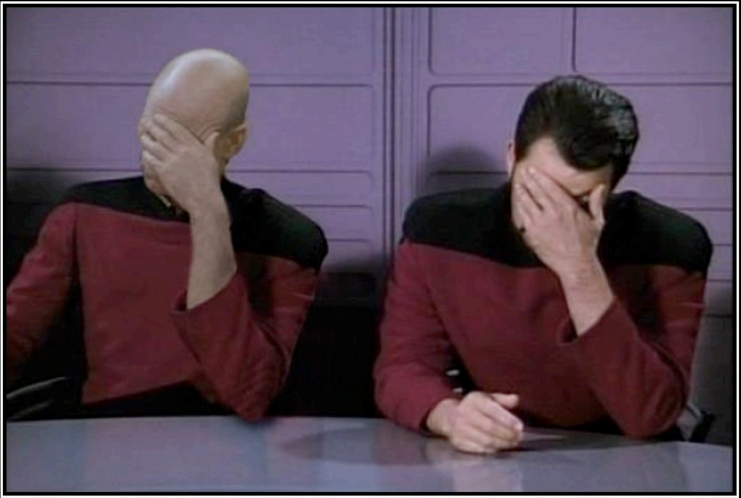
```cpp
#include <vector>

class B {
  int& foo;
};

int main() {
  std::vector<B> vec;
  B elt;
  vec.push_back(elt);
}
```

Easy?

```
test.cpp:3:7: error: cannot define the implicit default assignment
    operator for 'B', because non-static reference member 'foo' can't
      use default assignment operator
class B {
      ^
/usr/include/c++/4.6/bits/stl_vector.h:834:4: note: in instantiation
      member function
    'std::vector<B, std::allocator<B> >::_M_insert_aux' requested he
          _M_insert_aux(end(), __x);
          ^
test.cpp:10:7: note: in instantiation of member function 'std::vector
      std::allocator<B> >::push_back' requested here
  vec.push_back(elt);
      ^
test.cpp:4:8: note: declared here
  int& foo;
       ^
/usr/include/c++/4.6/bits/vector.tcc:317:16: note: implicit default
      assignment operator for 'B' first required here
          *__position = __x_copy;
```



DOUBLE FACEPALM

FOR WHEN ONE FACEPALM DOESN'T CUT IT

(I had to use \footnotesize to fit the error on the screen...)

```
test.cpp: In instantiation of ‘void std::vector<_Tp,
  _Alloc>::_M_insert_aux(std::vector<_Tp, _Alloc>::iterator, const
  _Tp&) [with _Tp = B; _Alloc = std::allocator<B>; std::vector<_Tp,
  _Alloc>::iterator = __gnu_cxx::__normal_iterator<B*, std::vector<B>
  >; typename std::_Vector_base<_Tp, _Alloc>::pointer = B*]’:
/usr/include/c++/4.7/bits/stl_vector.h:893:4:   required from ‘void
  std::vector<_Tp, _Alloc>::push_back(const value_type&) [with _Tp =
  B; _Alloc = std::allocator<B>; std::vector<_Tp, _Alloc>::value_type
  = B]’
test.cpp:10:20:   required from here
test.cpp:3:7: error: non-static reference member ‘int& B::foo’, can’t
  use default assignment operator
In file included from /usr/include/c++/4.7/vector:70:0,
                 from test.cpp:1:
/usr/include/c++/4.7/bits/vector.tcc:336:4: note: synthesized method
  ‘B& B::operator=(const B&)’ first required here
```

There are people working hard to make sure you get these errors.

People working on *type systems*.

I want to convince you that there's a good reason for type systems.

Plan

1. Introduction
2. What is typing?
3. Let's do some math!
4. So what do I do?

Typing?

Making sure you don't mix oranges with apples.

Since 1968! (Algol)

For performance

with typing

Source code.

```
class Orange {
  int size;
  color color;
}

int main () {
  Orange o(8cm, red);
  print(o.size);
}
```

Compiled code.

```
o = allocate_block(2)
set(offset(o, 0), 8cm)
set(offset(o, 1), red)
print_int(offset(o, 0))
```

Source code.

```
function main () {
  var o = {
    size: 8cm,
    color: red,
    origin: "spain",
    ...
  };
  console.log(o.size);
}
```

Compiled code.

Without typing

```
o = create_dictionary()
... (several lines) ...
set_key(o, "size", 8cm)
set_key(o, "color", red)
check(o, is_dictionary)
check(o, has_key, "size")
call_print(fetch_key(o, "size"))

print(thing):
 depending_on_the_type_of(thing):
    if integer:
      print_int(thing)
    if ...
```

For performance

A type describes the *shape of an object*.

type = memory representation
⇒ better generated code
⇒ better performance

Types help the compiler

We just saw *static typing*.

Dynamic languages are harder to compile, because you have to *check the types* at run-time.

For the programmer

For the speed of development

Types won't even allow you to *write* some buggy code.

Should this code be allowed?

```
void print(Orange o) {
  cout << o.flavor << endl;
}
```

With typing

```
test2.cpp:11:13: error: no member named 'flavor'
  cout << o.flavor << endl;
          ~ ^
1 error generated.
```

Error when compiling the code.

Without typing

Error when running.

Let's hope your code is well-tested...

Types help the programmer

A type system can rule out programming mistakes *in advance*.

Example

If I change the `size` field into a `diameter` field...

The compiler will flag all the locations in the source code that need to be changed.

With typing

Testing

Sample program

```
if (planets are aligned) {
  // ...
  print(o.flavor);
} else {
  // ...
  print(o.size);
}
```

Testing only covers a *fraction* of the program.

(Exponential number of configurations to test!)

An exhaustive analysis

Strong, static typing applies to the *whole* program.

Other reasons

Typing enables...

- reasoning about who-modifies-what (C++ `const` keyword) in a *modular* fashion,
- hiding internal representation through type *abstraction*,
- easy refactoring of the code,
- better support for other tools (IDEs, analyzers)...

Plan

How do people like me reason on type systems?

VAR
$$K; x @ t \vdash x : t$$

LET
$$\frac{K; P \vdash e_1 : t_1 \qquad K, x : \mathsf{term}; x @ t_1 \vdash e_2 : t_2}{K; P \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : t_2}$$

FUNCTION
$$\frac{K, \vec{X} : \vec{\kappa}, x : \mathsf{term}; P * x @ t_1 \vdash e : t_2 \qquad P \text{ is duplicable}}{K; P \vdash \mathsf{fun}\ [\vec{a} : \vec{\kappa}]\ (x : t_1) : t_2 = e : \forall(\vec{X} : \vec{\kappa})\ t_1 \to t_2}$$

INSTANTIATION
$$\frac{K; P \vdash e : \forall(X : \kappa)\ t_1}{K; P \vdash e : [T_2/X]t_1}$$

APPLICATION
$$K; x_1 @ t_2 \to t_1 * x_2 @ t_2 \vdash x_1\ x_2 : t_1$$

TUPLE
$$K; \vec{x} @ \vec{t} \vdash (\vec{x}) : (\vec{t})$$

NEW
$$\frac{\mathsf{A}\ \{\vec{f}\} \text{ is defined}}{K; \vec{x} @ \vec{t} \vdash \mathsf{A}\ \{\vec{f} = \vec{x}\} : \mathsf{A}\ \{\vec{f} : \vec{t}\}}$$

READ
$$\frac{t \text{ is duplicable} \qquad P \text{ is } x @ \mathsf{A}\ \{F[f : t]\}\ \mathsf{adopts}\ u}{K; P \vdash x.f : (t \mid P)}$$

WRITE
$$\frac{\mathsf{A}\ \{\ldots\} \text{ is exclusive}}{K; x_1 @ \mathsf{A}\ \{F[f : t_1]\}\ \mathsf{adopts}\ u * x_2 @ t_2 \vdash x_1.f \leftarrow x_2 : (\mid x_1 @ \mathsf{A}\ \{F[f : t_2]\}\ \mathsf{adopts}\ u)}$$

MATCH
$$\frac{\text{for every } i, \quad K; P \vdash \mathsf{let}\ p_i = x\ \mathsf{in}\ e_i : t}{K; P \vdash \mathsf{match}\ x\ \mathsf{with}\ \vec{p} \to \vec{e} : t}$$

WRITETAG
$$\frac{\mathsf{A}\ \{\ldots\} \text{ is exclusive} \qquad \mathsf{B}\ \{\vec{f'}\} \text{ is defined} \qquad \#\vec{f} = \#\vec{f'}}{K; x @ \mathsf{A}\ \{\vec{f} : \vec{t}\}\ \mathsf{adopts}\ u \vdash \mathsf{tag\ of}\ x \leftarrow B : (\mid x @ \mathsf{B}\ \{\vec{f'} : \vec{t}\}\ \mathsf{adopts}\ u)}$$

GIVE
$$\frac{t_2\ \mathsf{adopts}\ t_1}{K; x_1 @ t_1 * x_2 @ t_2 \vdash \mathsf{give}\ x_1\ \mathsf{to}\ x_2 : (\mid x_2 @ t_2)}$$

TAKE
$$\frac{t_2\ \mathsf{adopts}\ t_1}{K; x_1 @ \mathsf{dynamic} * x_2 @ t_2 \vdash \mathsf{take}\ x_1\ \mathsf{from}\ x_2 : (\mid x_1 @ t_1 * x_2 @ t_2)}$$

FAIL
$$K; P \vdash \mathsf{fail} : t$$

SUB
$$\frac{K; P_2 \vdash e : t_1 \qquad P_1 \leq P_2 \qquad t_1 \leq t_2}{K; P_1 \vdash e : t_2}$$

FRAME
$$\frac{K; P_1 \vdash e : t}{K; P_1 * P_2 \vdash e : (t \mid P_2)}$$

EXISTSELIM
$$\frac{K, X : \kappa; P \vdash e : t}{K; \exists(X : \kappa)\ P \vdash e : t}$$

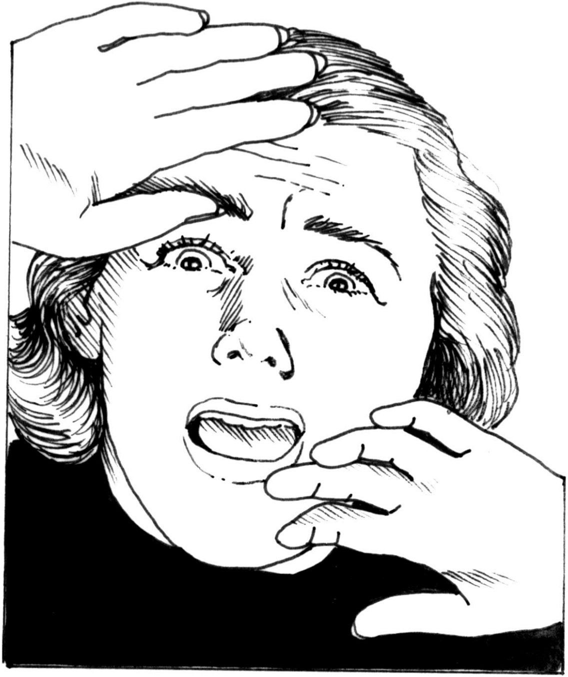
Figure 4. Typing rules

LETTUPLE
$$\frac{(\vec{t}) \text{ is duplicable} \qquad K, \vec{x} : \mathsf{term}; P * x @ (\vec{t}) * \vec{x} @ \vec{t} \vdash e : t}{K; P * x @ (\vec{t}) \vdash \mathsf{let}\ (\vec{x}) = x\ \mathsf{in}\ e : t}$$

LETDATAMATCH
$$\frac{(\vec{t}) \text{ is duplicable} \qquad K, \vec{x} : \mathsf{term}; P * x @ \mathsf{A}\ \{\vec{f} : \vec{t}\}\ \mathsf{adopts}\ u * \vec{x} @ \vec{t} \vdash e : t}{K; P * x @ \mathsf{A}\ \{\vec{f} : \vec{t}\}\ \mathsf{adopts}\ u \vdash \mathsf{let}\ \mathsf{A}\ \{\vec{f} = \vec{x}\} = x\ \mathsf{in}\ e : t}$$

LETDATAMISMATCH
$$\frac{\mathsf{A} \text{ and } \mathsf{B} \text{ belong to a common algebraic data type}}{K; P * x @ \mathsf{A}\ \{\vec{f} : \vec{t}\}\ \mathsf{adopts}\ u \vdash \mathsf{let}\ \mathsf{B}\ \{\vec{f'} = \vec{x}\} = x\ \mathsf{in}\ e : t}$$

LETDATAUNFOLD
$$\frac{x @ \mathsf{A}\ \{\vec{f} : \vec{t}\}\ \mathsf{adopts}\ u \text{ is an unfolding of } T\ \vec{T} \qquad K; P * x @ \mathsf{A}\ \{\vec{f} : \vec{t}\}\ \mathsf{adopts}\ u \vdash \mathsf{let}\ \mathsf{A}\ \{\vec{f} = \vec{x}\} = x\ \mathsf{in}\ e : t}{K; P * x @ T\ \vec{T} \vdash \mathsf{let}\ \mathsf{A}\ \{\vec{f} = \vec{x}\} = x\ \mathsf{in}\ e : t}$$



Formally...

These are called *derivation rules*.

Here's an example:

$$\frac{x \text{ instance of class } C \qquad C \text{ has a field } f \text{ of type } t}{x.f \text{ has type } t}$$

(Top part: hypotheses. Bottom part: conclusion.)

Formally...

These are called *derivation rules*.

Here's an example:

$$\frac{o \text{ instance of class } \mathit{Orange} \qquad \mathit{Orange} \text{ has a field size of type int}}{o.\text{size has type int}}$$

(Top part: hypotheses. Bottom part: conclusion.)

Two important rules

Let's switch to ML, the family of languages that are being studied in my field.

App

$$\frac{\Gamma \vdash f : \tau_1 \to \tau_2 \qquad \Gamma \vdash x : \tau_1}{\Gamma \vdash f\, x : \tau_2}$$

Fun

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \text{fun}\ x \to e : \tau_1 \to \tau_2}$$

This is what we call a typing judgement.

Is a program well-typed?

Provide a proof derivation, that is, a tower of rules ending with axioms.

$$
\dfrac{
  \dfrac{
    \dfrac{
      \dfrac{}{x : \mathsf{int} \vdash x : \mathsf{int}}\;\text{Var}
      \qquad
      \dfrac{}{x : \mathsf{int} \vdash 1 : \mathsf{int}}\;\text{Constant}
    }{x : \mathsf{int} \vdash x + 1 : \mathsf{int}}\;\text{Plus}
  }{\varepsilon \vdash \mathsf{fun}\ x \to x + 1 : \mathsf{int} \to \mathsf{int}}\;\text{Fun}
  \qquad
  \dfrac{}{\varepsilon \vdash 42 : \mathsf{int}}\;\text{Constant}
}{\varepsilon \vdash (\mathsf{fun}\ x \to x + 1)\ 42 : \mathsf{int}}\;\text{App}
$$

Why all the pain?

We want to assert that a program is well-typed because of the following theorem:

Well-typed programs don't go wrong.

Where « wrong » means: run into a segmentation fault.

Proving this theorem requires...

1. Defining what it means for a program to run (« operational semantics »)
2. Proving that the types remain the same during execution (« subject reduction »)
3. Proving that the program actually does something (« progress »)

Operational semantics

Defines how to *perform a computation*.

For the purposes of the proof, we define a notion of *substitution*, where we *replace* a variable with an expression.

$$\text{let } x = e_1 \text{ in } e_2 \rightsquigarrow e_2[e_1/x]$$

(real programs aren't compiled that way!)

Operational semantics

The various reduction steps of a small code snippet:

```
let x = 2 + 2 in
let y = x * x in
sqrt y
```

Operational semantics

The various reduction steps of a small code snippet:

```
let x = 4 in
let y = x * x in
sqrt y
```

Operational semantics

The various reduction steps of a small code snippet:

```
let y = 4 * 4 in
sqrt y
```

Operational semantics

The various reduction steps of a small code snippet:

```
let y = 16 in
sqrt y
```

Operational semantics

The various reduction steps of a small code snippet:

```
sqrt 16
```

Operational semantics

The various reduction steps of a small code snippet:

```
4
```

Subject reduction

If the program is well-typed, it won't end up in an ill-typed state.

```
let y = 16 in
sqrt "ilovethejuniorseminar"
```

Subject reduction (traditional)

We then show that if $e \leadsto e'$ and $\Gamma \vdash e : \tau$, then $\Gamma \vdash e' : \tau$, i.e. the types remain throughout execution.

No surprises!

Progress

The program is either:

1. in a configuration where there exists a reduction that we cannot compute (segmentation fault):

   ```
   2 + "coucou"
   ```

2. or in a configuration where we can always reduce (in the middle of a computation):

   ```
   2 + 2
   ```

3. or in a configuration where we can no longer reduce (result of a computation):

   ```
   4
   ```

Combining all three notions

The combination of *operational semantics*, *subject reduction* and *progress* gives the original result, called **type soundness**:

> Well-typed programs don't segfault.

This is a result that we achieve through the use of a *type system*.

How do you determine whether a program is well-typed?

You need an algorithm!

This is not an algorithm

Fun

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \mathsf{fun}\ x \rightarrow e : \tau_1 \rightarrow \tau_2}$$

You need to *know* what you want to prove *before* proving it.

How do you do it?

- Either require type annotations from the programmer, like in C++,
- or have the system « guess automatically » the types, like in ML (type inference).

What is a good type-checking algorithm?

- I'm writing a type-checking algorithm. If the algorithms says « yes », is the program well-typed? (Correctness)
- I'm writing a type-checking algorithm. If the algorithms says « no », is the program ill-typed? (Completeness)

After type-checking…

Compiling the program

The type-checking gives theorems for the *original program*.

What about the compiled code?

Another big topic

My team also focuses on *compiler certification*.

We don't want the compiler to ruin all the good work of the type-checker.

Plan

1. Introduction
2. What is typing?
3. Let's do some math!
4. So what do I do?

Reasoning on state

There is an implicit notion of *state* in programs.

```
int* x = new int;

delete x;
```

Reasoning on state

There is an implicit notion of *state* in programs.

```
int* x = new int;

delete x;
```

x goes from *valid pointer* to *invalid pointer*

Reasoning on state

There is an implicit notion of *state* in programs.

```
int* x = new int;
// x: int*
delete x;
// x: int*
```

However, the type system just says *pointer*.

Reasoning on state

There is an implicit notion of *state* in programs.

```
int* x = new int;
// x: valid int*
delete x;
// x: invalid
```

However...

Traditional type systems provide no facilities for reasoning about the *state* of a program.

We want types to talk about the state an object is in.

Why is it difficult?

If the type of an object changes, who *sees* the change?

Why is it difficult?

```
int* x = new int;
// x: valid int*
int* y = x;
// x: valid int*, y: valid int*
// ... (several lines of code) ...
// x: valid int*, y: valid int*
delete x;
// x: invalid, y: valid int*
delete y;
// apocalypse
```

Why is it difficult?

Do *x* and *y* *point* to the same thing?

Unsolvable problem. We need a type system with *restrictions*.

General idea

```
int* x = new int;
// x: valid int*
int* y = x;
// x: valid int*, y = x
// ... (several lines of code) ...
// x: valid int*, y = x
delete x;
// x: invalid, y = x
delete y;
// error: y is invalid
```

General idea

- We need to keep track of *aliasing*.
- We have a notion of *ownership*.

Thank you

So long and thanks for all the fish!