

# Typesystems For DSLs

**Markus Voelter**

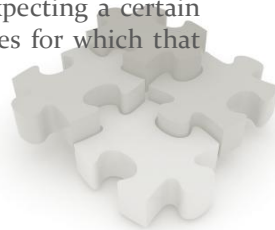
independent/itemis  
voelter@acm.org  
@markusvoelter  
<http://voelter.de>

## Typesystem (from Wikipedia)

In computer science, a type system may be defined as a tractable syntactic framework for classifying phrases according to the kinds of values they compute.

A type system **associates types with each computed value**. By examining the flow of these values, a type system attempts to prove that no type errors can occur.

The type system in question **determines what constitutes a type error**, but a type system generally seeks to guarantee that operations expecting a certain kind of value are not used with values for which that operation makes no sense.



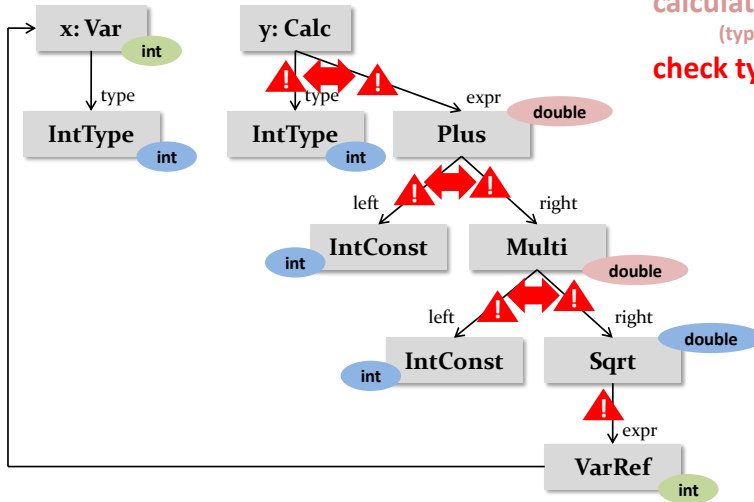
```
var x : int;
calc y : int = 1 + 2 * sqrt(x)
```

declare fixed types

derive types...

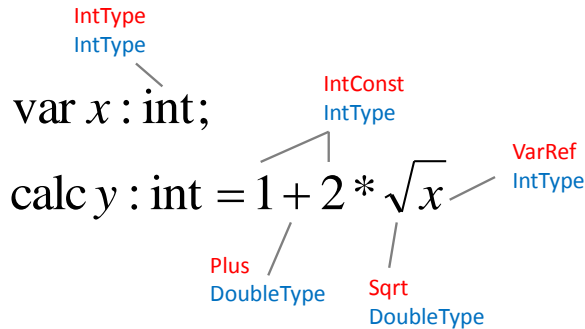
calculate common  
(type hierarchies)

check types



By the way:

# Type != Meta Class



Aren't  
Typesystems  
just another set of



Constraints?

Yes, but:  
way more complicated;  
special approach useful

Three main approaches:

- 1 Recursion
- 2 Unification
- 3 Pattern Matching

# 1 Recursion

---

**general:**

Recursion is the process of repeating items in a self-similar way

**computer science:**

in which it refers to a method of defining functions in which the function being defined is applied within its own definition


# 1 Recursion

---

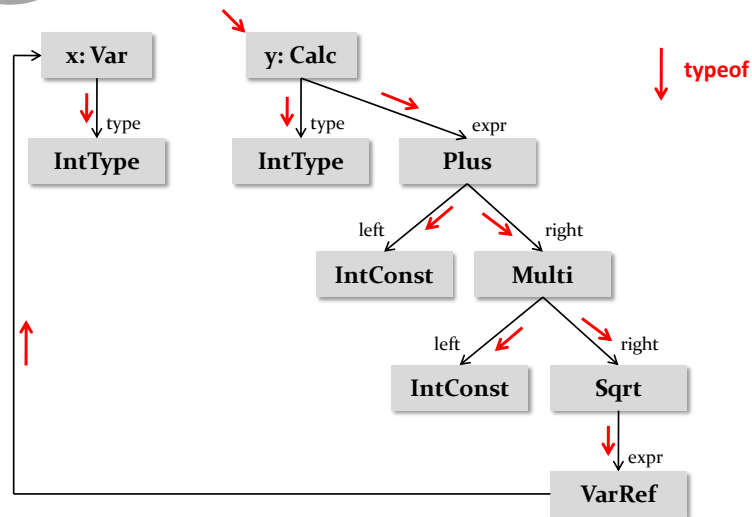
**computer science:**

in which it refers to a method of defining functions in which the function being defined is applied within its own definition

```
unsigned int factorial(unsigned int n)
{
    if (n <= 1)
        return 1;
    else
        return n * factorial(n-1);
}
```



# 1 Recursion



# 1 Recursion

**typeof** := function that returns the type of an element for a given element

**typeof** := element -> type-of-the-element

In the **recursive** approach, it does it by drawing on the types of „related“ elements  $p_1 \dots p_n$

**typeof(e)** :=  $f(\text{typeof}(p_1), \text{typeof}(p_2) \dots \text{typeof}(p_n))$

# 1 Recursion

---

```

var i: int
var i: int = 42
var i: int = 33.33
var i = 42
  
```

} LocalVarDecl:  
`var` name=ID `:` type=Type (`=` init=Expr)?

```

typeof( LocalVarDecl ) {
  if ( type != null && init != null ) {
    ensure typeof( init ) is-same-as typeof( type ) ||
      typeof( init ) is-subtype-of typeof( type )
    return typeof( type )
  }
  ...
  
```

# 1 Recursion

---

LocalVarDecl:  
`var` name=ID `:` type=Type (`=` init=Expr)?

```

typeof( LocalVarDecl ) {
  if ( type != null && init != null ) {
    ensure typeof( init ) is-same-as typeof( type ) ||
      typeof( init ) is-subtype-of typeof( type )
    return typeof( type )
  }
  if ( type == null && init != null ) { return typeof( init ) }
  if ( type != null && init == null ) { return typeof( type ) }
  if ( type == null && init == null ) { raise error }
  }
  
```

# 1 Recursion

LocalVarDecl:

```
`var` name=ID `:` type=Type (`=` init=Expr)?
```

Derivation and Propagation

```
ensureSameOrSub( LocalVarDecl.init, LocalVarDecl.type )
useTypeOfFeature( LocalVarDecl, LocalVarDecl.type )
else useTypeOfFeature( LocalVarDecl, LocalVarDecl.init )
else error
```

Constraints

(do nothing if C.x == null)

# Xtext/TS

# 1 Recursion

Now for Xtext 2.0

With new typing DSL

Static consistency  
checks, custom navigation,  
templates, etc.

```

expts.ts
typesystem expr.typesys.ExprTypesystem
ecore file "platform:/resource/expr/src-gen/expr/ExprDemo.ecore"
language package expr.exprDemo.ExprDemoPackage

@section "Basics"

// float is a subtype of string
subtype IntType base FloatType

// primitive types use clones of themselves as their type
typeof Type + -> clone
// string literals have string type
typeof StringLiteral -> StringType
typeof Equals -> BoolType
typeof NumberLiteral -> javacode
// variable declarations and formulas use have the type of their type
typeof VarDecl -> feature type {
  ensureCompatibility type <=: init
}
typeof Formula -> feature type {
  ensureCompatibility type <=: expr
}
typeof Symbol -> abstract
// a symbol reference has the type of the symbol it references
typeof SymbolRef -> feature a symbol
// plus must have ints or floats on either side, and the two
// have to be compatible. Type of Plus is the common supertype
// of left and right
typeof Plus -> common left right {
  ensureType left <=: IntType, FloatType
  ensureType right <=: IntType, FloatType
  ensureCompatibility left <=> right
}
// same for multi...
typeof Multi -> common left right {
  ensureType left <=: IntType, FloatType
  ensureType right <=: IntType, FloatType
  ensureCompatibility left <=> right
}

```

# Xtext

<http://code.google.com/a/eclipselabs.org/p/xtext-typesystem/>

# Demo



## Xtext/TS

### 1 Recursion

## Xtypes

*a DSL for writing type systems for  
Xtext languages*

Lorenzo Bettini

<http://xtypes.sourceforge.net/>

```
rule TIntConstant
derives
  G |- var IntConstant i : var Type int
from
  var IntType intType
  $int := $intType

rule TStringConstant
derives
  G |- var StringConstant s : var Type string
from
  var StringType stringType
  $string := $stringType
```

```
rule TFieldOk
derives
  G |- var Field f : 'OK'
from
  // checks that there are no duplicate field in the hierarchy
  var Class C := (Class) container($f)
  !exists inheritedField in getall($C.extends, fields, extends) {
    $inheritedField.name = $f.name
  } error='duplicate field in base class'
  !exists otherField in $C.fields {
    $otherField.name = $f.name
    $otherField != $f
  } error='duplicate field in the same class'
```



## 2 Unification

---

Unification is an operation [..] which produces from [..] logic terms a substitution which [..] makes the terms equal modulo some equational theory.

- (1)  $2 * x == 10$
- (2)  $x + x == 10$
- (3)  $x + y == 2 * x + 5$

set of linear equations

## 2 Unification

---

Unification is an operation [..] which produces from [..] logic terms a substitution which [..] makes the terms equal modulo some equational theory.

- (1)  $2 * x == 10$
- (2)  $x + x == 10$
- (3)  $x + y == 2 * x + 5$

set of linear equations

**$x := 5$**   
 **$y := 10$**

- (1)  $2 * 5 == 10$
- (2)  $5 + 5 == 10$
- (3)  $5 + 10 == 2 * 5 + 5$

# 2 Unification

```

var i: int
var i: int = 42
var i: int = 33.33
var i = 42
  
```

} LocalVarDecl:  
`var` name=ID `:` type=Type (`=` init=Expr)?

**typeof**( LocalVarDecl.type ) **:<=:** **typeof**( LocalVarDecl.init )  
**typeof**( LocalVarDecl ) **::=:** **typeof**( LocalVarDecl.type )

(do nothing if C.x == null)

Constraints and Derivation Rules  
at the same time!

# 2 Unification

**typeof**( LocalVarDecl.type ) **:<=:** **typeof**( LocalVarDecl.init )  
**typeof**( LocalVarDecl ) **::=:** **typeof**( LocalVarDecl.type )

var i: int	<b>typeof</b> ( int ) <b>:&lt;=:</b> <b>typeof</b> ( -null- )	→ ignore
	<b>typeof</b> ( T ) <b>::=:</b> <b>typeof</b> ( int )	→ T := int
var i: int = 42	<b>typeof</b> ( int ) <b>:&lt;=:</b> <b>typeof</b> ( int )	→ ok
	<b>typeof</b> ( T ) <b>::=:</b> <b>typeof</b> ( int )	→ T := int
var i: int = 33.33	<b>typeof</b> ( int ) <b>:&lt;=:</b> <b>typeof</b> ( double )	→ error!
	<b>typeof</b> ( T ) <b>::=:</b> <b>typeof</b> ( int )	→ T := int
var i = 42	<b>typeof</b> ( U ) <b>:&lt;=:</b> <b>typeof</b> ( int )	→ U := int
	<b>typeof</b> ( T ) <b>::=:</b> <b>typeof</b> ( U )	→ T := int

# 2 Unification

---

```
var i: int[]
```

```
var i: int[] = {1, 2, 3}
```

```
var i = {1, 2, 3}
```

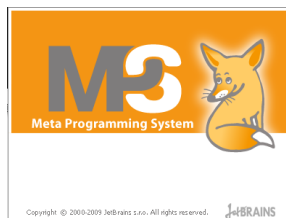
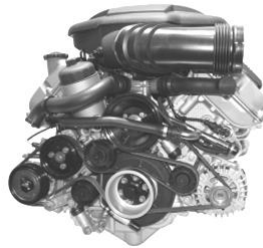
```
type var t
```

```
init.elements.foreach{ e | t <::= typeof(e) }
```

```
typeof( LocalVarDecl.type ) <::= t
```

```
typeof( LocalVarDecl ) ::= typeof( LocalVarDecl.type )
```

## Demo



# 3 Pattern Matching

```

var i: int
var i: int = 42
var i: int = 33.33
var i = 42
  
```

} LocalVarDecl:  
`var` name=ID `:` type=Type (`=` init=Expr)?

typeof( init )	typeof( type )	typeof( LocalVarDecl )
int	int	int
int	-	int
-	int	int
-	-	<error>
int	double	int
double	int	<error>

# 3 Pattern Matching

```

var i: int
var i: int = 42
var i: int = 33.33
var i = 42
  
```

} LocalVarDecl:  
`var` name=ID `:` type=Type (`=` init=Expr)?

typeof( init )	typeof( type )	typeof( LocalVarDecl )
t	-	t
-	t	t
t	u :<= t	u
-	-	<error>

upper case: unbound, free type vars

lower case: bound type vars

# 3 Pattern Matching

```

var i: int
var i: int = 42
var i: int = 33.33
var i = 42
  
```

} LocalVarDecl:  
`var` name=ID `:` type=Type (`=` init=Expr)?

<b>typeof( init )</b>	<b>typeof( type )</b>	<b>typeof( LocalVarDecl )</b>
t	-	t
-	t	t
t	u :<= t	u
-	-	<error>

upper case: unbound, free type vars  
lower case: bound type vars

**Spoofox**

No Demo 😊



**Spoofox**

# THE END.

## .coordinates

**web** [www.voelter.de](http://www.voelter.de)  
**email** [voelter@acm.org](mailto:voelter@acm.org)  
**twitter** [markusvoelter](#)

**xing** [http://www.xing.com/profile/Markus\\_Voelter](http://www.xing.com/profile/Markus_Voelter)  
**linkedin** <http://www.linkedin.com/pub/0/377/a31>