Typesystems For DSLs

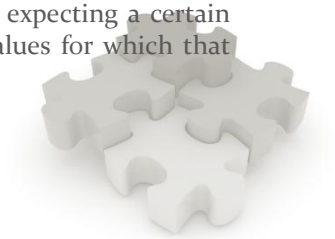
Markus Voelter
independent/itemis
voelter@acm.org
@markusvoelter
http://voelter.de

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.

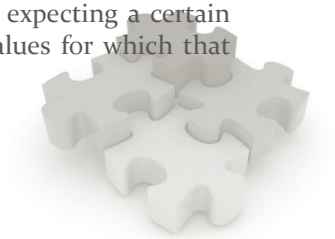


Typesystem (from Wikipedia)

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.

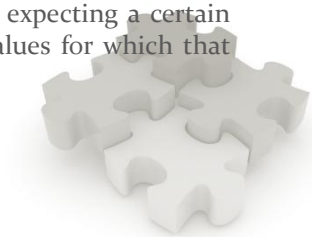


Typesystem (from Wikipedia)

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.

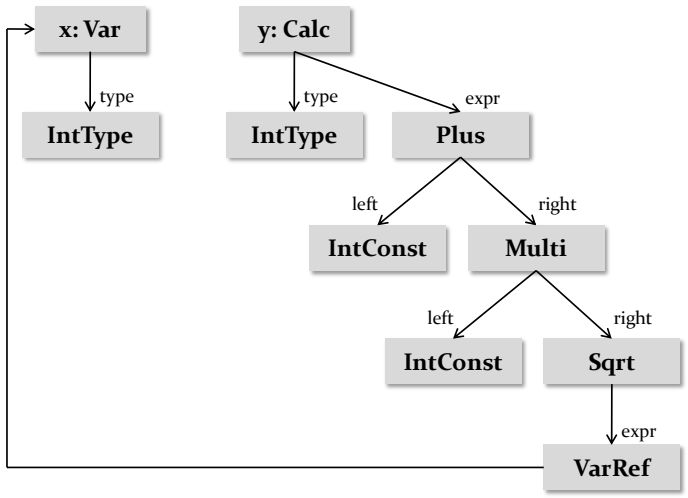


$\text{var } x : \text{int};$

$\text{calc } y : \text{int} = 1 + 2 * \sqrt{x}$

$\text{var } x : \text{int};$

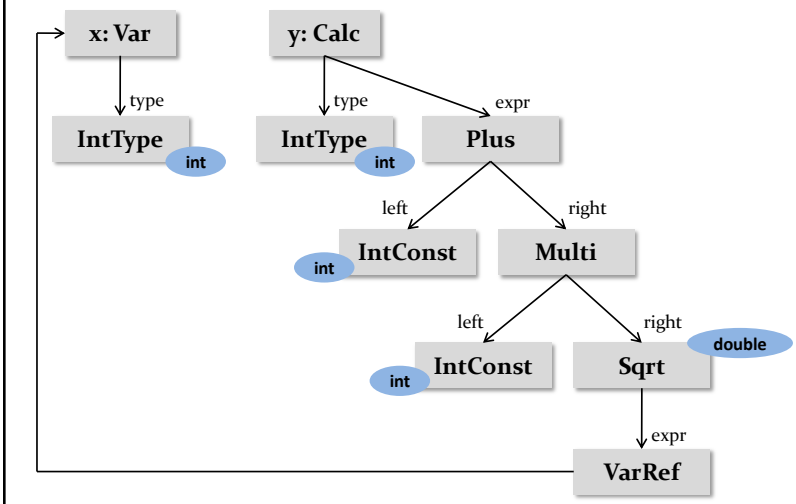
$\text{calc } y : \text{int} = 1 + 2 * \sqrt{x}$



var x : int;

calc y : int = $1 + 2 * \sqrt{x}$

declare fixed types

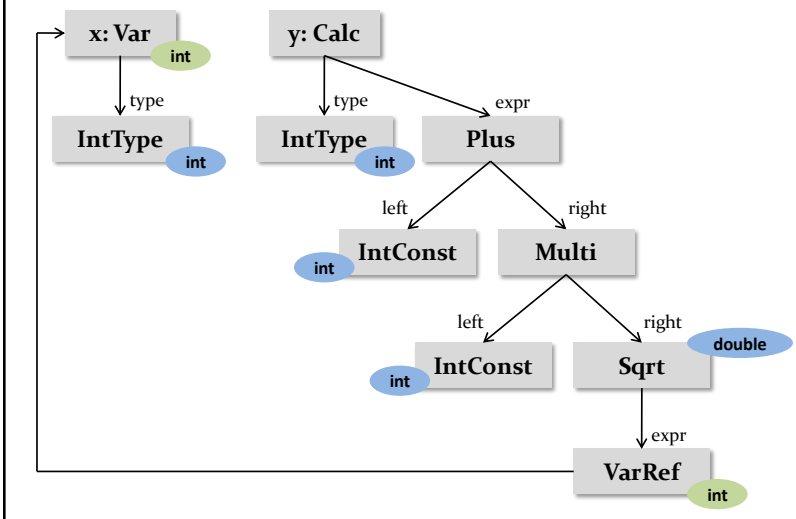


var x : int;

calc y : int = $1 + 2 * \sqrt{x}$

declare fixed types

derive types



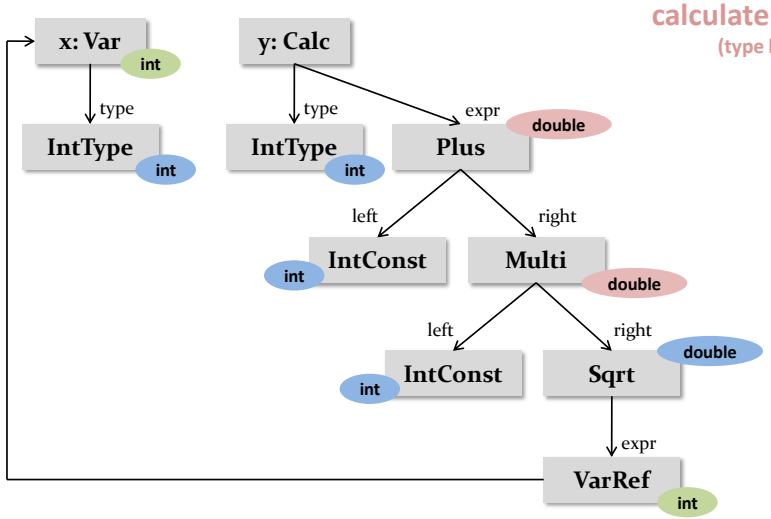
var x : int;

calc y : int = $1 + 2 * \sqrt{x}$

declare fixed types

derive types

calculate common (type hierarchies)



var x : int;

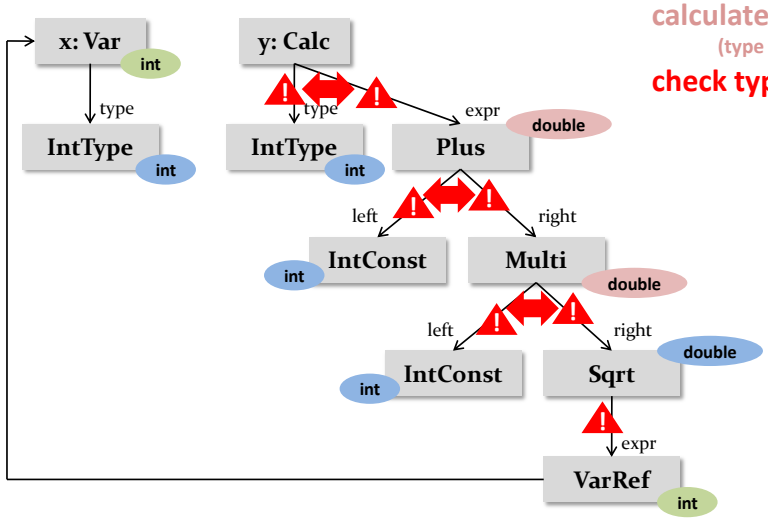
calc y : int = $1 + 2 * \sqrt{x}$

declare fixed types

derive types

calculate common (type hierarchies)

check types



By the way:

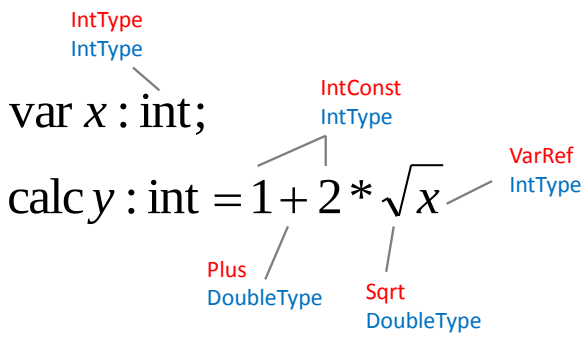
Type != Meta Class

$$\text{var } x : \text{int};$$

$$\text{calc } y : \text{int} = 1 + 2 * \sqrt{x}$$

By the way:

Type != Meta Class



Aren't
Typesystems
just another set of
Constraints?

Aren't
Typesystems
just another set of
Constraints?

Yes, but:
way more complicated;
special approach useful

Context

Intended for

static

Typesystems

Context

Especially useful
for languages
with
Expressions

Three main approaches:

1. Recusion
2. Unification
3. Pattern Matching

Recusion

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

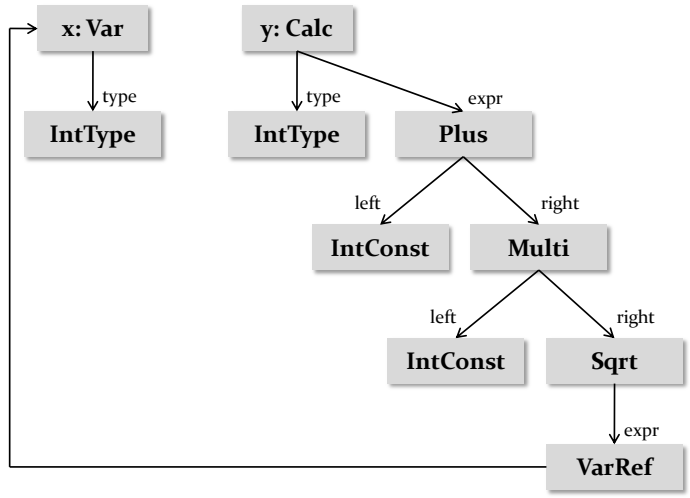
Recusion

computer science:

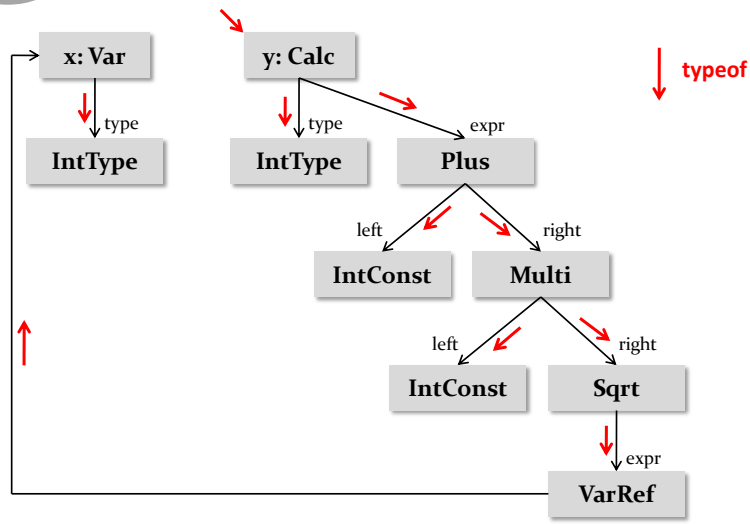
in which it refers to a method of defining functions in which the function being defined is applied within its own definition

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

1 Recusion



1 Recusion



1 Recusion

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

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

1 Recusion

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 p1 … p*n*

typeof(e) := f(typeof(p1), typeof(p2) … typeof(p*n*))

1 Recusion

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

1 Recusion

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

}

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

Recusion

```
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 )
 }
 …
```

Recusion

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 Recusion

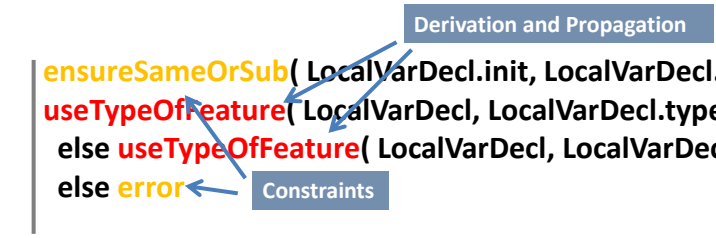
LocalVarDecl:
`var` name=ID `:` type=Type (`=` init=Expr)?

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

(do nothing if C.x == null)

1 Recusion

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



(do nothing if C.x == null)

1 Recusion

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 Recusion

Now for Xtext 2.0

With new typing DSL

Static consistency checks, custom navi-gation, templates, etc.

expr.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 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 Recusion

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.

2 Unification

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

???

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)?

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)

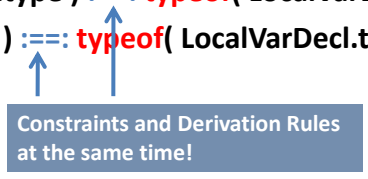
2 Unification

```
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)



2 Unification

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

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

var i: int

typeof(int) :<=: typeof(-null-) → ignore
typeof(T) :==: typeof(int) → T := int

2 Unification

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

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

var i: int

typeof(int) :<=: typeof(-null-) → ignore
typeof(T) :==: typeof(int) → T := int

var i: int = 42

typeof(int) :<=: typeof(int) → ok
typeof(T) :==: typeof(int) → T := int

2 Unification

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

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

var i: int	typeof(int) :<=: typeof(-null-)	→ ignore
	typeof(T) :==: typeof(int)	→ T := int
var i: int = 42	typeof(int) :<=: typeof(int)	→ ok
	typeof(T) :==: typeof(int)	→ T := int
var i: int = 33.33	typeof(int) :<=: typeof(double)	→ error!
	typeof(T) :==: typeof(int)	→ T := int

2 Unification

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

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

var i: int	typeof(int) :<=: typeof(-null-)	→ ignore
	typeof(T) :==: typeof(int)	→ T := int
var i: int = 42	typeof(int) :<=: typeof(int)	→ ok
	typeof(T) :==: typeof(int)	→ T := int
var i: int = 33.33	typeof(int) :<=: typeof(double)	→ error!
	typeof(T) :==: typeof(int)	→ T := int
var i = 42	typeof(U) :<=: typeof(int)	→ U := int
	typeof(T) :==: typeof(U)	→ T := int

2 Unification

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

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

var i: int	typeof(int) :<=: typeof(-null-)	→ ignore
	typeof(T) :==: typeof(int)	→ T := int
var i: int = 42	typeof(int) :<=: typeof(int)	→ ok
	typeof(T) :==: typeof(int)	→ T := int
var i: int = 33.33	typeof(int) :<=: typeof(do	
	typeof(T) :==: typeof(int)	
var i = 42	typeof(U) :<=: typeof(int)	
	typeof(T) :==: typeof(U)	

2 Unification

var i: int[]

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

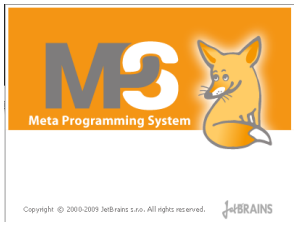
var i = {1, 2, 3}

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



2 Unification

Unification is more powerful:

```
factorial( int i ) {
  if ( i == 0 ) return 1;
  return i * factorial(i-1);
}
```

Recursion cannot be used to derive the return type of factorial, since it calls itself.

2 Unification

Unification is more powerful:

```
int factorial( int i ) {
  if ( i == 0 ) return 1;
  return i * factorial(i-1);
}
```

Recursion cannot be used to derive the return type of factorial, since it calls itself, so recursive functions need a type specification.

2 Unification

```
factorial( int i ) {
  if ( i == 0 ) return 1;
  return i * factorial(i - 1);
}
```

2 Unification

```
T   int
factorial( int i ) {
  if ( i == 0 ) return 1;
                 int
  return i * factorial(i - 1);
                          int
          T
}
```

2 Unification

```
T   int
factorial( int i ) {
  if ( i == 0 ) return 1;
                  int
  return i * factorial(i - 1);
                 int
             T   int
}
```

typeof(BinOp) :<=: typeof(BinOp.left)
typeof(BinOp) :<=: typeof(BinOp.right)

2 Unification

```
T   int
factorial( int i ) {
  if ( i == 0 ) return 1;
          int      int
  return i * factorial(i - 1);
                  int
             T   int
}
```

typeof(BinOp) :<=: typeof(BinOp.left)
typeof(BinOp) :<=: typeof(BinOp.right)
typeof(Return) :<=: typeof(Return.expr)

2 Unification

```
int   int
factorial( int i ) {
  if ( i == 0 ) return 1;
  return i * factorial(i - 1);
}
```

int int int int int int

```
typeof( BinOp ) :<=: typeof( BinOp.left )
typeof( BinOp ) :<=: typeof( BinOp.right )
typeof( Return ) :<=: typeof( Return.expr )

type var t_ret;
for (Return r in FunctionDecl.allReturns):
  t_ret :<=: typeof( r );
```

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)?
```

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)?

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

Spoofax

No Demo ☺

Spoofax

THE END.

.coordinates

web www.voelter.de
email voelter@acm.org
twitter markusvoelter

xing http://www.xing.com/profile/Markus_Voelter
linkedin http://www.linkedin.com/pub/0/377/a31