

Statix: Declarative Type System Specification

Eelco Visser

WG2.16 | Portland | February 2019

Based on:

Van Antwerpen, Bach Poulsen, Rouvoet, Visser

Scopes as Types

PACMPL 2 (OOPSLA), 2018

Type System Specification

Context

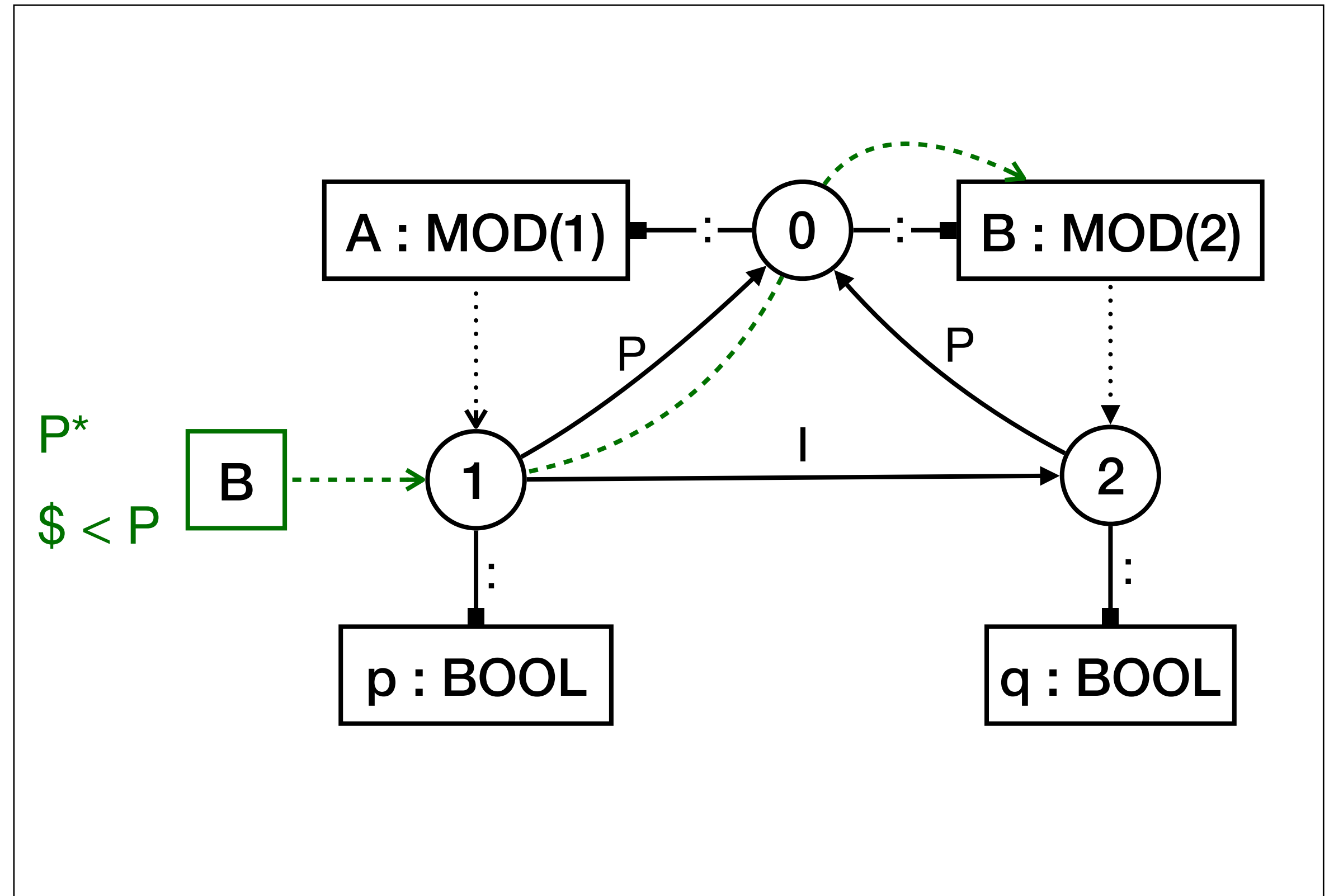
- Language workbench
- High-level language specification
- Abstract from implementation concerns (generate automatically)

Type systems

- Type constraints
- Name resolution \Rightarrow scope graphs
- Staging

Scope Graphs

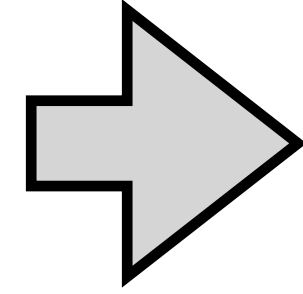
```
module A {  
  import B  
  def p : bool = ~q  
}  
module B {  
  def q : bool = true  
}
```



Statix By Example

Arithmetic Expressions: Concrete and Abstract Syntax

```
> 1 + 2 * 3
```

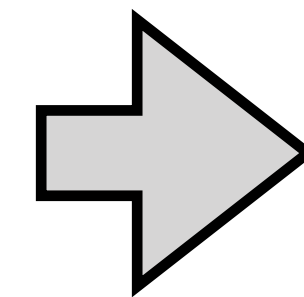


```
Program(  
  [Exp(  
    Add(Int("1"), Mul(Int("2"), Int("3")))  
  )]  
)
```

context-free syntax // arithmetic

```
Exp.Int      = <<INT>>  
Exp.Add      = <<Exp> + <Exp>> {left}  
Exp.Sub      = <<Exp> - <Exp>> {left}  
Exp.Mul      = <<Exp> * <Exp>> {left}  
Exp.Eq       = <<Exp> == <Exp>> {non-assoc}
```

SDF3



constructors // arithmetic

```
Int          : INT -> Exp  
Add          : Exp * Exp -> Exp  
Sub          : Exp * Exp -> Exp  
Mul          : Exp * Exp -> Exp  
Eq           : Exp * Exp -> Exp
```

Statix

Arithmetic Expressions: Abstract Syntax

```
> 1 + 2 * 3
```

```
constructors // arithmetic
```

```
Int      : INT -> Exp  
Add      : Exp * Exp -> Exp  
Sub      : Exp * Exp -> Exp  
Mul      : Exp * Exp -> Exp  
Eq       : Exp * Exp -> Exp
```

Arithmetic Expressions: Type Predicate

```
> 1 + 2 * 3
```

```
constructors // arithmetic
```

```
Int      : INT -> Exp
Add      : Exp * Exp -> Exp
Sub      : Exp * Exp -> Exp
Mul      : Exp * Exp -> Exp
Eq       : Exp * Exp -> Exp
```

```
sorts TYPE constructors
```

```
INT     : TYPE
BOOL    : TYPE
FUN     : TYPE * TYPE -> TYPE
```

```
rules // expressions
```

```
typeOfExp : scope * Exp -> TYPE
```

```
typeOfExp(s, Int(i)) = INT().
```

```
typeOfExp(s, Add(e1, e2)) = INT() :-  
  typeOfExp(s, e1) == INT(),  
  typeOfExp(s, e2) == INT().
```

```
. . .
```

```
typeOfExp(s, Eq(e1, e2)) = BOOL() :- {T}  
  typeOfExp(s, e1) == T,  
  typeOfExp(s, e2) == T.
```

Variable Definitions

```
def a = 0
def b = a + c
def b = 1 + d
def c : Int = 0
def e : Bool = 1
> a + b + c
```

```
sorts Program constructors
Program : list(Decl) -> Program
```

```
sorts Decl constructors
Def      : Bind -> Decl
Exp      : Exp -> Decl
```

```
sorts Bind constructors
Bind     : ID * Exp -> Bind
TBind   : ID * Type * Exp -> Bind
```

```
relations
typeOfDecl : occurrence -> TYPE
```


Variable Definitions

```
def a = 0
def b = a + c
def b = 1 + d
def c : Int = 0
def e : Bool = 1
> a + b + c
```

sorts Program constructors

Program : list(Decl) -> Program

sorts Decl constructors

Def : Bind -> Decl

Exp : Exp -> Decl

sorts Bind constructors

Bind : ID * Exp -> Bind

TBind : ID * Type * Exp -> Bind

relations

typeOfDecl : occurrence -> TYPE

rules

programOK : Program

programOK(Program(decls)) :- {s}

new s,
declsOk(s, decls).

rules // declarations

declOk : scope * Decl

declsOk maps declOk(*, list(*))

declOk(s, Def(bind)) :-
bindOk(s, s, bind).

declOk(s, Exp(e)) :- {T}
typeOfExp(s, e) == T.

Variable Definitions: Declarations

```
def a = 0
def b = a + c
def b = 1 + d
def c : Int = 0
def e : Bool = 1
> a + b + c
```

sorts Program constructors

Program : list(Decl) -> Program

sorts Decl constructors

Def : Bind -> Decl

Exp : Exp -> Decl

sorts Bind constructors

Bind : ID * Exp -> Bind

TBind : ID * Type * Exp -> Bind

relations

typeOfDecl : occurrence -> TYPE

rules // bindings

bindOk : scope * scope * Bind

bindsOk maps bindOk(*, *, list(*))

bindOk(s_bnd, s_ctx, Bind(x, e)) :- {T}
typeOfExp(s_ctx, e) == T,
s_bnd -> Var{x@x} with typeOfDecl T.

bindOk(s_bnd, s_ctx, TBind(x, t, e)) :- {T}
typeOfType(s_ctx, t) == T,
typeOfExp(s_ctx, e) == T,
s_bnd -> Var{x@x} with typeOfDecl T.

Variable Definitions: Name Resolution

```
def a = 0
def b = a + c
def b = 1 + d
def c : Int = 0
def e : Bool = 1
> a + b + c
```

```
sorts Program constructors
  Program : list(Decl) -> Program

sorts Decl constructors
  Def      : Bind -> Decl
  Exp      : Exp -> Decl

sorts Bind constructors
  Bind     : ID * Exp -> Bind
  TBind    : ID * Type * Exp -> Bind

relations
  typeOfDecl : occurrence -> TYPE
```

```
rules // variables
```

```
typeOfExp(s, Var(x)) = T :- {p d }
  typeOfDecl of Var{x@x} in s l-> [(p, (d, T))].
```

```
rules // bindings
```

```
bindOk : scope * scope * Bind
```

```
bindsOk maps bindOk(*, *, list(*))
```

```
bindOk(s_bnd, s_ctx, Bind(x, e)) :- {T}
  typeOfExp(s_ctx, e) == T,
  s_bnd -> Var{x@x} with typeOfDecl T.
```

```
bindOk(s_bnd, s_ctx, TBind(x, t, e)) :- {T}
  typeOfType(s_ctx, t) == T,
  typeOfExp(s_ctx, e) == T,
  s_bnd -> Var{x@x} with typeOfDecl T.
```

Lexical Scope: Functions

```
def i = 3

def inc = fun(x : Int) { x + i }

> inc 2
```

```
constructors // functions
Fun  : ID * Type * Exp -> Exp
App  : Exp * Exp -> Exp
```

```
sorts TYPE constructors
INT  : TYPE
BOOL : TYPE
FUN  : TYPE * TYPE -> TYPE
```

```
rules // functions
```

```
typeOfExp(s, Fun(x, t, e)) = FUN(T, S) :- {s_fun}
  typeOfType(s, t) == T,
  new s_fun,
  s_fun -P-> s,
  s_fun -> Var{x@x} with typeOfDecl T,
  typeOfExp(s_fun, e) == S.
```

```
typeOfExp(s, App(e1, e2)) = T :- {S}
  typeOfExp(s, e1) == FUN(S, T),
  typeOfExp(s, e2) == S.
```

Lexical Scope: Sequential Let

```
def a = 0
def b = 1
def c = 2

> let
  a = c;
  b = a;
  c = b
in
  a + b + c
```

```
rules // let bindings
```

```
typeOfExp(s, Let(binds, e)) = T :- {s_let}
  new s_let,
  sbindsOk(s, s_let, binds),
  typeOfExp(s_let, e) == T.
```

```
rules // bindings
```

```
sbindsOk : scope * scope * list(Bind)
```

```
sbindsOk(s, s_fin, []) :-
  s_fin -P-> s.
```

```
sbindsOk(s, s_fin, [bind | binds]) :- {s_mid}
  new s_mid, s_mid -P-> s,
  bindOk(s_mid, s, bind),
  sbindsOk(s_mid, s_fin, binds).
```

Lexical Scope: Parallel Let

```
def a = 0
def b = 1
def c = 2

> letpar
  a = c;
  b = a;
  c = b
in
  a + b + c
```

```
rules // let bindings

typeOfExp(s, LetPar(binds, e)) = T :- {s_let}
  new s_let, s_let -P-> s,
  bindsOk(s_let, s, binds),
  typeOfExp(s_let, e) == T.

bindOk : scope * scope * Bind
bindsOk maps bindOk(*, *, list(*))

bindOk(s_bnd, s_ctx, Bind(x, e)) :- {T}
  typeOfExp(s_ctx, e) == T,
  s_bnd -> Var{x@x} with typeOfDecl T.
```

Lexical Scope: Recursive Let

```
> letrec
  odd = fun(x : Int) {
    if x == 0 then false
    else even(x - 1)
  };
  even = fun(x : Int) {
    if x == 0 then true
    else odd(x - 1)
  }
in
  even(3)
```

```
rules // let bindings

typeOfExp(s, LetRec(binds, e)) = T :- {s_let}
  new s_let, s_let -P-> s,
  bindsOk(s_let, s_let, binds),
  typeOfExp(s_let, e) == T.

bindOk : scope * scope * Bind
bindsOk maps bindOk(*, *, list(*))

bindOk(s_bnd, s_ctx, Bind(x, e)) :- {T}
  typeOfExp(s_ctx, e) == T,
  s_bnd -> Var{x@x} with typeOfDecl T.
```

Modules

```
module A {  
  import B  
  def a = 4  
  def c = b + 4  
}  
module B {  
  import A  
  def b = a + 3  
}
```


Modules: Scopes as Types

```
module A {  
  import B  
  def a = 4  
  def c = b + 4  
}  
module B {  
  import A  
  def b = a + 3  
}
```

```
rules // modules  
declOk(s, Module(m, decls)) :- {s_mod}  
  new s_mod, s_mod -P-> s,  
  s -> Mod{m@m} with typeOfDecl MOD(s_mod),  
  declsOk(s_mod, decls).  
  
declOk(s, Import(m)) :- {p d s_mod}  
  typeOfDecl of Mod{m@m} in s l-> [(p, (d, MOD(s_mod)))],  
  s -I-> s_mod.
```

```
sorts TYPE constructors  
INT : TYPE  
BOOL : TYPE  
FUN : TYPE * TYPE -> TYPE  
MOD : scope -> TYPE
```

```
name-resolution  
labels P I R  
  
resolve Var filter pathMatch[P* (R* | I*)]  
  min pathLt[$ < I, $ < P, I < P, R < P]  
  
resolve Mod filter pathMatch[P P* I*]  
  min pathLt[$ < I, $ < P, I < P, R < P]
```

Records

```
record Point {  
  x : Int  
  y : Int  
}  
  
def p : Point  
  = new Point { x = 1, y = 2}  
  
> p.x + p.y  
  
def z = 3  
  
> with p do x + y + z
```

Record Type Declaration: Scopes as Types

```
record Point {  
  x : Int  
  y : Int  
}  
  
def p : Point  
  = new Point { x = 1, y = 2 }  
  
> p.x + p.y  
  
def z = 3  
  
> with p do x + y + z
```

```
sorts TYPE constructors  
INT   : TYPE  
BOOL  : TYPE  
FUN   : TYPE * TYPE -> TYPE  
REC   : scope -> TYPE
```

```
rules // record type  
  
declOk(s, Record(x, fdecls)) :- {s_rec}  
  new s_rec,  
  fdeclsOk(s_rec, s, fdecls),  
  s -> Var{x@x} with typeOfDecl REC(s_rec).  
  
fdeclOk : scope * scope * FDecl  
fdeclsOk maps fdeclOk(*, *, list(*))  
  
fdeclOk(s_bnd, s_ctx, FDecl(x, t)) :- {T}  
  typeOfType(s_ctx, t) == T,  
  s_bnd -> Var{x@x} with typeOfDecl T.
```

Record Literals

```
record Point {  
  x : Int  
  y : Int  
}  
  
def p : Point  
  = new Point { x = 1, y = 2 }  
  
> p.x + p.y  
  
def z = 3  
  
> with p do x + y + z
```

```
sorts TYPE constructors  
INT   : TYPE  
BOOL  : TYPE  
FUN   : TYPE * TYPE -> TYPE  
REC   : scope -> TYPE
```

```
rules // records construction
```

```
typeOfExp(s, New(x, fbinds)) = REC(s_rec) :- {p d}  
  typeOfDecl of Var{x@x} in s l-> [(p, (d, REC(s_rec)))],  
  fbindsOk(s, s_rec, fbinds).
```

```
fbindOk : scope * scope * FBind  
fbindsOk maps fbindOk(*, *, list(*))
```

```
fbindOk(s, s_rec, FBind(x, e)) :- {p d T}  
  typeOfExp(s, e) == T,  
  typeOfDecl of Var{x@x} in s_rec l-> [(p, (d, T))].
```

Record Projection

```
record Point {  
  x : Int  
  y : Int  
}  
  
def p : Point  
  = new Point { x = 1, y = 2 }  
  
> p.x + p.y  
  
def z = 3  
  
> with p do x + y + z
```

```
sorts TYPE constructors  
INT   : TYPE  
BOOL  : TYPE  
FUN   : TYPE * TYPE -> TYPE  
REC   : scope -> TYPE
```

```
rules // record projection  
  
typeOfExp(s, Proj(e, x)) = T :- {p d s_rec S}  
  typeOfExp(s, e) == S,  
  proj(S, x) == T.  
  
proj : TYPE * ID -> TYPE  
  
proj(REC(s_rec), x) = T :- {p d}  
  typeOfDecl of Var{x@x} in s_rec l-> [(p, (d, T))].
```

With Record

```
record Point {  
  x : Int  
  y : Int  
}  
  
def p : Point  
  = new Point { x = 1, y = 2 }  
  
> p.x + p.y  
  
def z = 3  
  
> with p do x + y + z
```

```
sorts TYPE constructors  
INT   : TYPE  
BOOL  : TYPE  
FUN   : TYPE * TYPE -> TYPE  
REC   : scope -> TYPE
```

```
rules // with record value
```

```
typeOfExp(s, With(e1, e2)) = T :- {s_with s_rec}  
  typeOfExp(s, e1) == REC(s_rec),  
  new s_with,  
  s_with -P-> s, s_with -R-> s_rec,  
  typeOfExp(s_with, e2) == T.
```

```
name-resolution
```

```
labels P I R
```

```
resolve Var filter pathMatch[P* (R* | I*)]  
           min pathLt[$ < I, $ < P, I < P, R < P]
```

Type References

```
record Point {  
  x : Int  
  y : Int  
}  
  
def translate : Point -> Point -> Point  
= fun(p: Point){ fun(d: Point) {  
  new Point{  
    x = p.x + d.x,  
    y = p.y + d.y }  
} }  
  
def p : Point = new Point { x = 1, y = 2}  
  
> translate(p)(p)
```

```
rules // types  
  
typeOfType : scope * Type -> TYPE  
  
typeOfType(s, IntT()) = INT().  
  
typeOfType(s, BoolT()) = BOOL().  
  
typeOfType(s, FunT(t1, t2)) =  
  FUN(typeOfType(s, t1), typeOfType(s, t2)).  
  
typeOfType(s, RecT(x)) = REC(s_rec) :- {p d}  
  typeOfDecl of Var{x@x}  
  in s l-> [(p, (d, REC(s_rec)))].
```

What Else

Other Contributions

In the paper

- Typing rules for STLC+records, System F, Featherweight Java
- Resolution calculus of scope graphs
- Declarative semantics of Statix
- Description of solver algorithm of Statix

In the artifact

- Implementation of scope graphs and Statix
- Executable specs of STLC+records, System F, Featherweight Generic Java