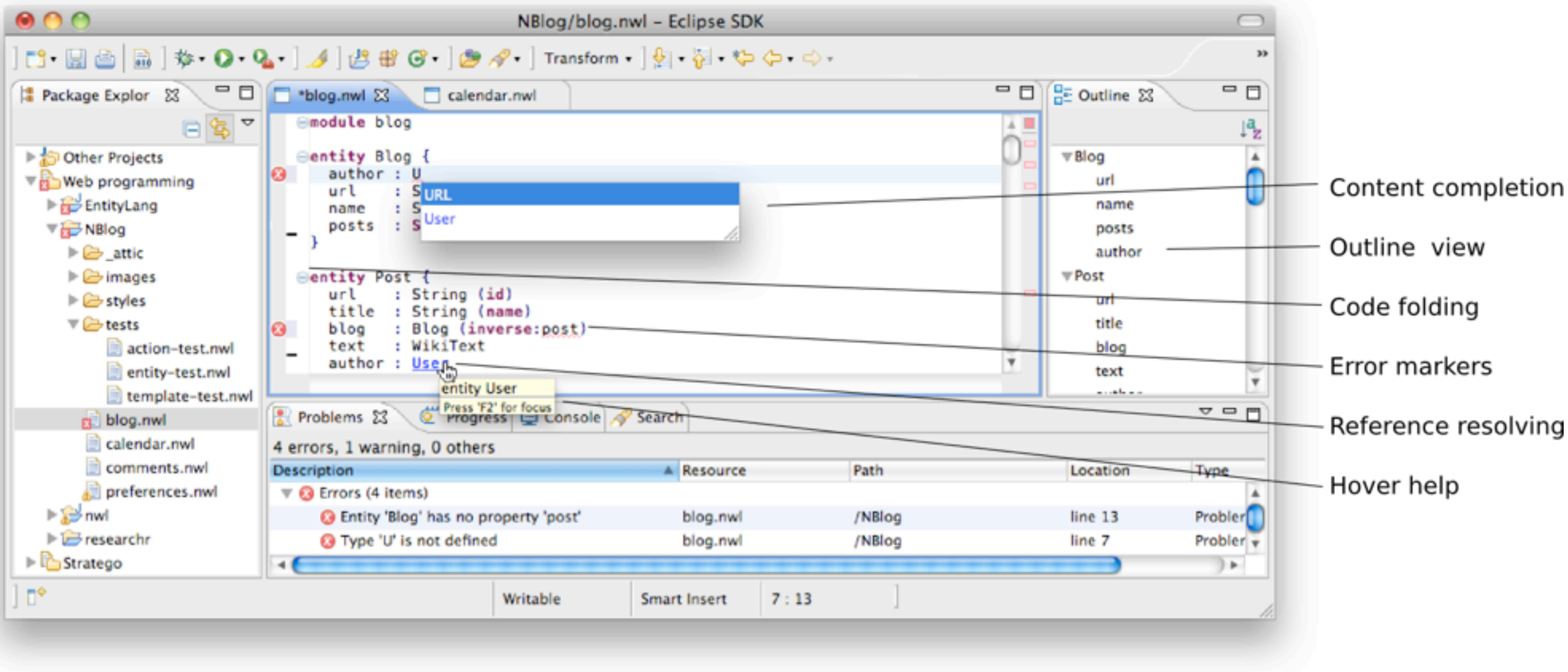
Declarative Name Binding and Scope Rules

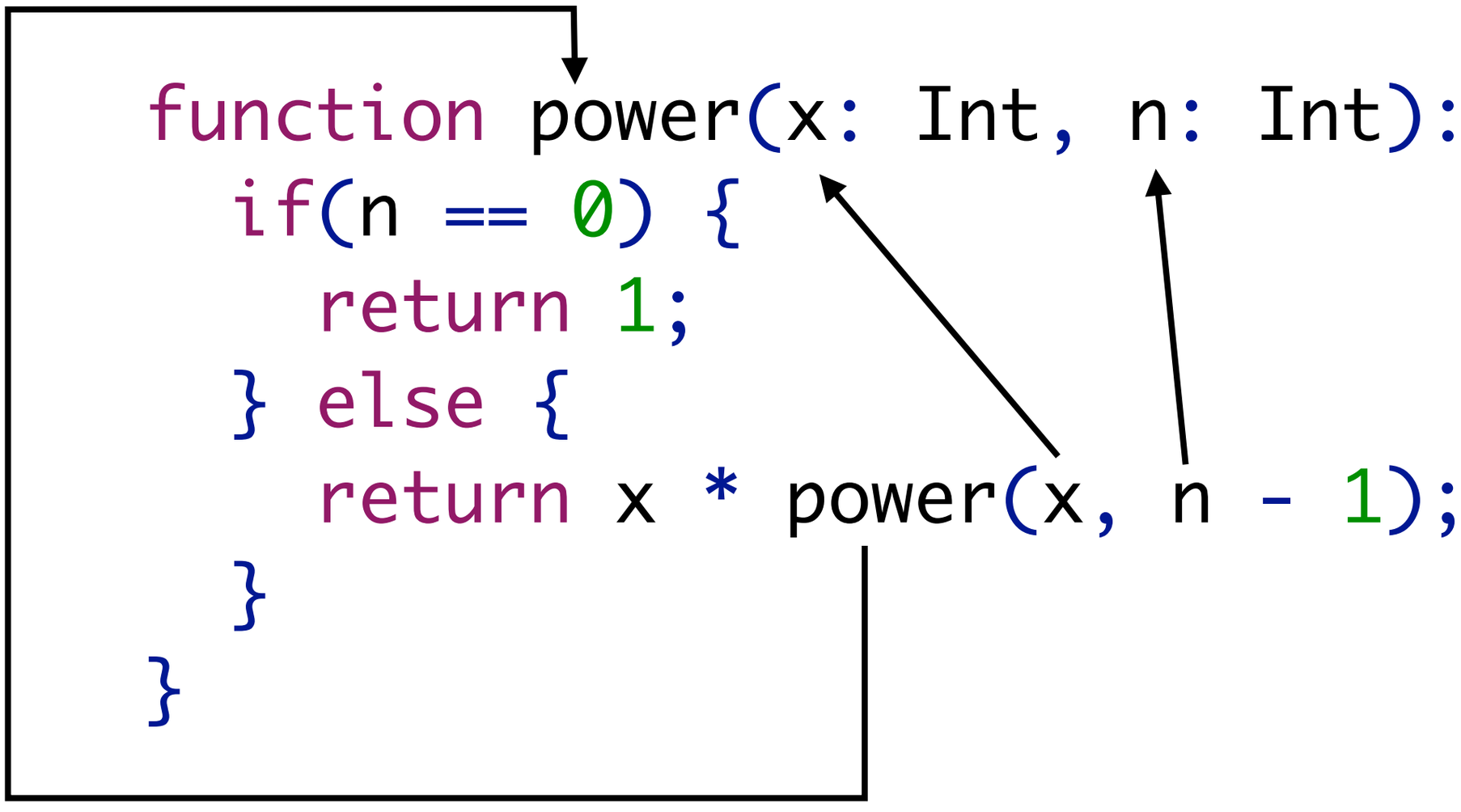
Gabriël Konat, Lennart Kats, Guido Wachsmuth,
Eelco Visser

TU Delft
Delft University of Technology

Context: Spoofax Language Workbench



Name Binding and Scope



Name Binding and Scope

Name Binding and Scope

static checking

editor services

transformation

refactoring

code generation

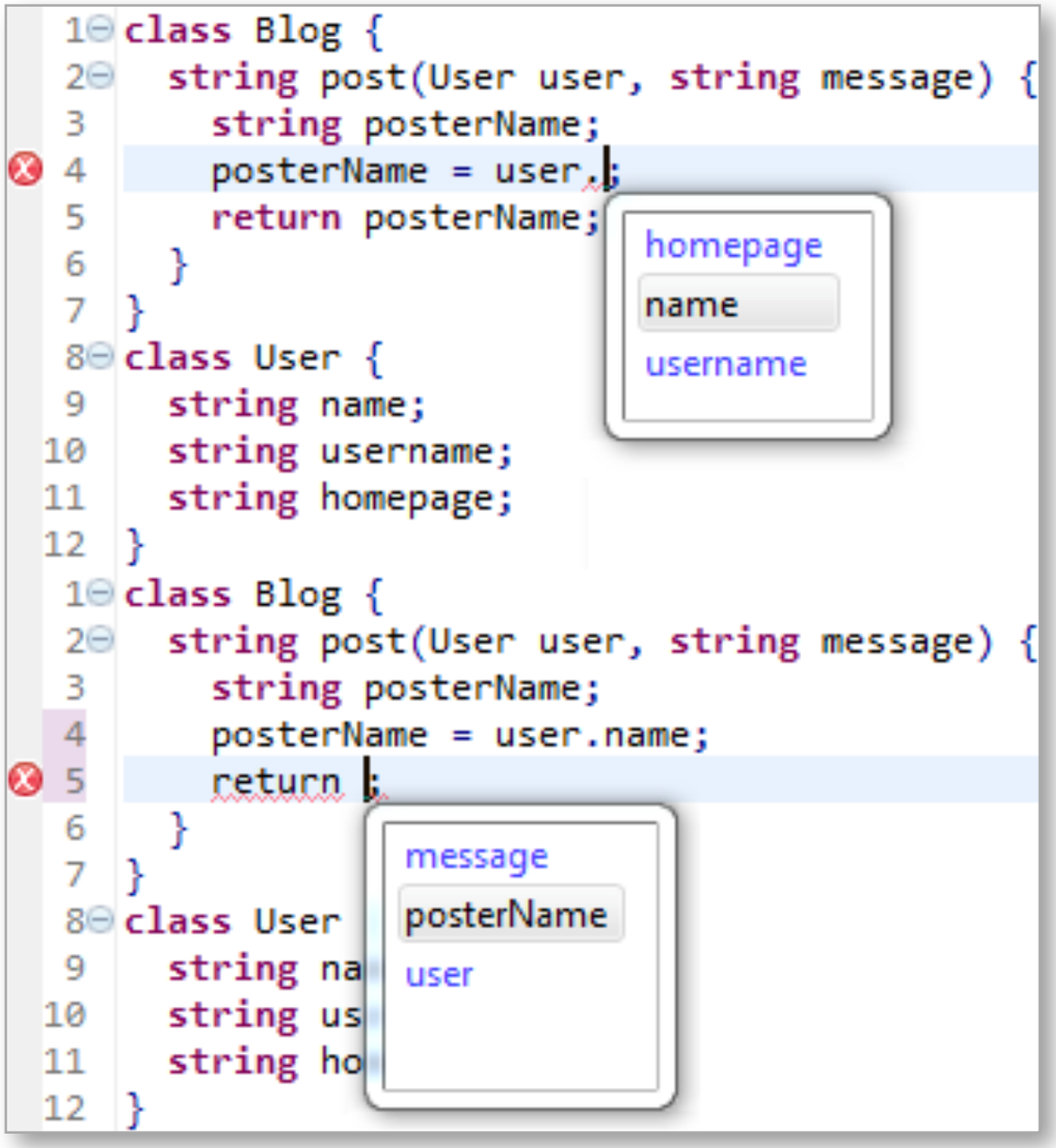
```
1  class User {
2    string name;
3  }
4  class Blog {
5    string post(User user, string message) {
6      posterName = "name";
7      string posterName;
8      posterName = user.nam;
9      string posterName = user.name;
10     return posterName;
11   }
12 }
```

Reference Resolution (Navigation)

```
1  class User {
2    string name;
3  }
4  class Blog {
5    string post(User user, string message) {
6      string posterName;
7      posterName = user.name;
8      return posterName;
9    }
10 }
```

```
1  class User {
2    string name;
3  }
4  class Blog {
5    string post(User user, string message) {
6      string posterName;
7      posterName = user.name;
8      return posterName;
9    }
10 }
```

Code Completion



Classical Approaches

Type Systems

(Env $\emptyset$)

$$\frac{}{\emptyset \vdash \diamond}$$

(Env x)

$$\frac{\Gamma \vdash A \qquad x \notin dom(\Gamma)}{\Gamma, x{:}A \vdash \diamond}$$

(Type Const)

$$\frac{\Gamma \vdash \diamond \qquad K \in Basic}{\Gamma \vdash K}$$

(Type Arrow)

$$\frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \to B}$$

(Val x)

$$\frac{\Gamma', x{:}A, \Gamma'' \vdash \diamond}{\Gamma', x{:}A, \Gamma'' \vdash x{:}A}$$

(Val Fun)

$$\frac{\Gamma, x{:}A \vdash M : B}{\Gamma \vdash \lambda x{:}A.M : A \to B}$$

(Val Appl)

$$\frac{\Gamma \vdash M : A \to B \qquad \Gamma \vdash N : A}{\Gamma \vdash M\,N : B}$$

Source: Luca Cardelli. Type Systems. In Handbook of Computer Science and Engineering, Chapter 103. CRC Press, 1997.

Modular Structural Operational Semantics

$$\frac{U = \{\rho, \ldots\}, \quad \rho(x) = con}{x \ -U\rightarrow \ con}$$

$$\frac{U = \{\rho, \sigma, \ldots\}, \quad \rho(x) = l, \quad \sigma(l) = con}{x \ -U\rightarrow \ con}$$

$$\frac{e_0 \ -X\rightarrow \ e_0'}{e_0 \ bop \ e_1 \ -X\rightarrow \ e_0' \ bop \ e_1}$$

$$\frac{e_1 \ -X\rightarrow \ e_1'}{con_0 \ bop \ e_1 \ -X\rightarrow \ con_0 \ bop \ }$$

$$\frac{bop = +, \quad n = n_0 + n_1}{n_0 \ bop \ n_1 \longrightarrow n}$$

$$\frac{bop = -, \quad n_0 < n_1, \quad U = \{\varepsilon, .}{n_0 \ bop \ n_1 \ -\{\varepsilon{=}err, \ldots\}\rightarrow \ err}$$

$$\frac{d \ -X\rightarrow \ d'}{\mathbf{let}\, d\, \mathbf{in}\, e \ -X\rightarrow \ \mathbf{let}\, d'\, \mathbf{in}\, e}$$

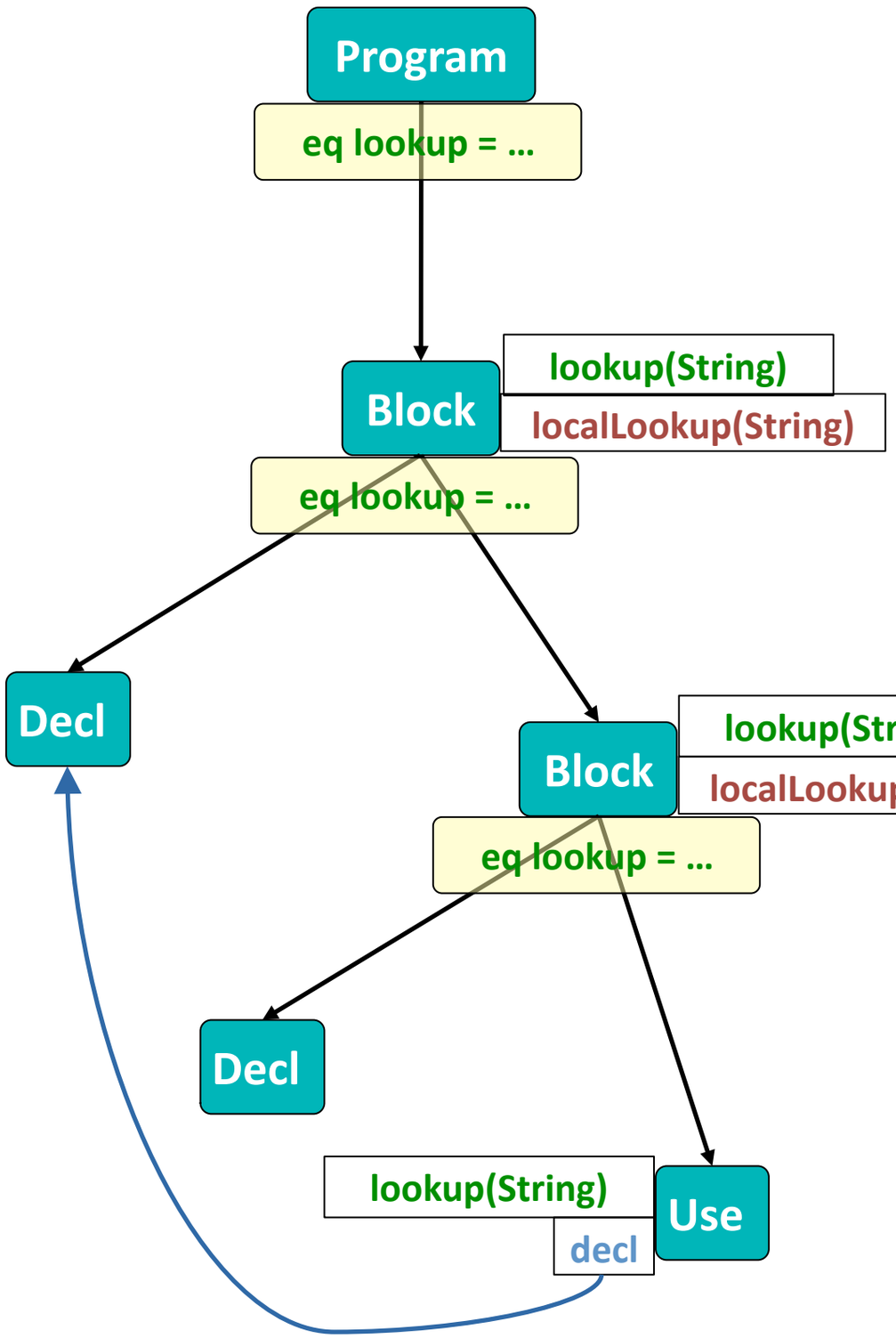
$$\frac{e \ -\{\rho{=}\rho[\rho_0], \ldots\}\rightarrow \ e'}{\mathbf{let}\, \rho_0\, \mathbf{in}\, e \ -\{\rho, \ldots\}\rightarrow \ \mathbf{let}\, \rho_0\, \mathbf{in}\, e}$$

$$\mathbf{let}\, \rho_0\, \mathbf{in}\, con \longrightarrow con$$

Source: Peter Mosses. Modular SOS: Differences from SOS. First APPSEM-II Workshop, Nottingham, March 2003

Reference Attribute Grammars

Source: Görel Hedin




```
eq Program.getBlock().lookup(String id) =
  null;
```

```
syn Block.localLookup(String id) {
  for (Decl d : getDecls()) {
    if (d.getID().equals(id))
      return d;
  }
  return null;
}
```

```
eq Block.getChild(int i).lookup(String id) {
  Decl d = localLookup(id);
  if (d != null) return d;
  return lookup(id);
}
```

```
syn Decl Use.decl = lookup(getID());
inh Decl Use.lookup(String);
```

Rewriting Strategies with Dynamic Rules

```
rename-top = alltd(rename)

rename :
  |[ var x : srt ]| -> |[ var y : srt ]|
  where y := <add-naming-key(|srt)> x

rename :
  |[ define page x (farg1*) { elem1* } ]| ->
  |[ define page x (farg2*) { elem2* } ]|
  where {| Rename
         : farg2* := <map(rename-page-arg)> farg1*
         ; elem2* := <rename-top> elem1*
         |}

rename = Rename

add-naming-key(|srt) : x -> y
  where y := x{<newname> x}
      ; rules (
          Rename : Var(x) -> Var(y)
          TypeOf : y -> srt
        )
```

What is the Problem?

What is the Problem?

Mental model:

Name binding is defined in terms of
programmatic encoding of name resolution

Rules encoded in many language operations

Abstractions try to reduce overhead
of the programmatic encoding

Name Binding in Xtext

```
grammar DomainModel with Terminals

Domainmodel :
  elements += Type*
;
Type:
  DataType | Entity
;
DataType:
  'datatype' name = ID
;
Entity:
  'entity' name = ID ('extends' superType = [Entity])? '{'
    features += Feature*
  '}'
;
Feature:
  many?='many'? name = ID ':' type = [Type]
;
```

Name Binding Language

Demo of NBL in Spoofax

Definitions and Reference

```
Class(_, c, _, _):
  defines Class c

ClassType(c) :
  refers to Class c

Base(c) :
  refers to Class c
```

```
class C {
}

class D {
  C x;
}

class E : D {
  C x;
}
```

Design Choice: Grammar Annotations

```
"class" Type@=ID "{" ClassBodyDecl* "}" -> Definition {"Class"}

Type@ID                                 -> Type {"ClassType"}
```

```
Class(_, c, _) :
  defines Class c

ClassType(c) :
  refers to Class c
```

Unique and Non-Unique Definitions

```
Class(NonPartial(), c, _, _) :
  defines unique Class c

Class(Partial(), c, _, _) :
  defines non-unique Class c

Base(c) :
  refers to Class c

ClassType(c) :
  refers to Class c
```

```
class D {
  C1 x;
}
class C1 {
  D d;
}
class C2: C1 {
  Int i;
}
partial class C3: C2 {
  Int j;
}
partial class C3 {
  Int k;
}
```

Namespaces

```
namespaces
  Class Method Field Variable

Class(NonPartial(), c, _, _):
  defines unique Class c

Field(_, f) :
  defines unique Field f

Method(_, m, _, _):
  defines unique Method m
Call(m, _) :
  refers to Method m

VarDef(_, v):
  defines unique Variable v
VarRef(x):
   refers to Variable x
   otherwise refers to Field x
```

```
class x {
  int x;
  void x() {
    int x; x = x + 1;
  }
}
```

Scope

```
Class(NonPartial(), c, _, _):
  defines unique Class c
  scopes Field, Method

Class(Partial(), c, _, _):
  defines non-unique Class c
  scopes Field, Method

Method(_, m, _, _):
  defines unique Method m
  scopes Variable

Block(_):
  scopes Variable
```

```
class C {
  int y;
  void m() {
    int x;
    x = y + 1;
  }
}

class D {
  void m() {
    int x;
    int y;
    { int x; x = y + 1; }
    x = y + 1;
  }
}
```

C# Namespaces are Scopes

```
Namespace(n, _):
  defines Namespace n
  scopes Namespace, Class
```

```
namespace N {
  class N {}
  namespace N {
    class N {}
  }
}
```

Imports

```
Using(qn):
  imports Class from Namespace ns
  where qn refers to Namespace ns

Base(c):
  imports Field
     from Class c {transitive}
  imports Method
     from Class c {transitive}
```

```
using N;

namespace M {
  class C { int f; }
}

namespace O {
  class E: M.C {
    void m() {}
  }
  class F:E { }
}
```

Definition Context

```
Foreach(t, v, exp, body):
  defines Variable v
      of type t
      in body
```

```
class C {
  void m(int[] x) {
    foreach (int x in x)
      WriteLine(x);
  }
}
```

Interaction with Type System (1)

```
FieldAccess(e, f):
  refers to Field f in c
  where e has type ClassType(c)
```

```
class C {
  int i;
}

class D {
  C c;
  int i;
  void init() {
    i = c.i;
  }
}
```

Interaction with Type System (2)

```
Method(t, m, ps, _) :
  defines Method m of type t
```

```
MethodCall(e, m, _):
  refers to Method m
      of type t in c
  where e has type ClassType(c)
```

```
class C {
  int i;
  int get() { return i; }
}

class D {
  C c;
  int i;
  void init() {
    i = c.get();
  }
}
```

Interaction with Type System (3)

```
Method(t, m, ps, _) :
  defines unique Method m
       of type (ts, t)
  where ps has type ts

Param(t, p):
  defines unique Variable p
       of type t

MethodCall(e, m, es):
  refers to Method m
      of type (ts, t) in c
  where e has type ClassType(c)
  where es has type ts
```

```
class C {
  void m() {}
  void m(int x) {}
  void m(bool x) {}
  void m(int x, int y) {}
  void m(bool x, bool y) {}
  void x() {
    m();
    m(42);
    m(true);
    m(21, 21);
    m(true, false);
  }
}
```

Name Binding

`defines`	class partial class type inheritance
`refers`	namespace using
`namespaces`	
`scopes`	method field variable parameter block
`imports`	

Name Binding Language in Spoofax (*)

names.nd

```
namespaces
  Program Namespace Using
  Class Function Field Variable

rules // Namespaces

  Namespace(x, _) :
    defines non-unique Namespace x
    scopes Namespace, Class

  UsingPart(x) :
    defines non-unique Using x
    refers to Namespace x
    imports Class from Namespace x {current-file}

  UsingPart(u, x) :
    refers to Namespace x in Namespace y
    where u refers to Namespace y

rules // Classes

  Class(x, _) :
    defines unique Class x of type Type(x)
    scopes Field, Function

  Class(x, y, _) :
    defines unique Class x of type Type(x)
    refers to Class y
    scopes Field, Function
    imports Field from Class y {transitive}
```

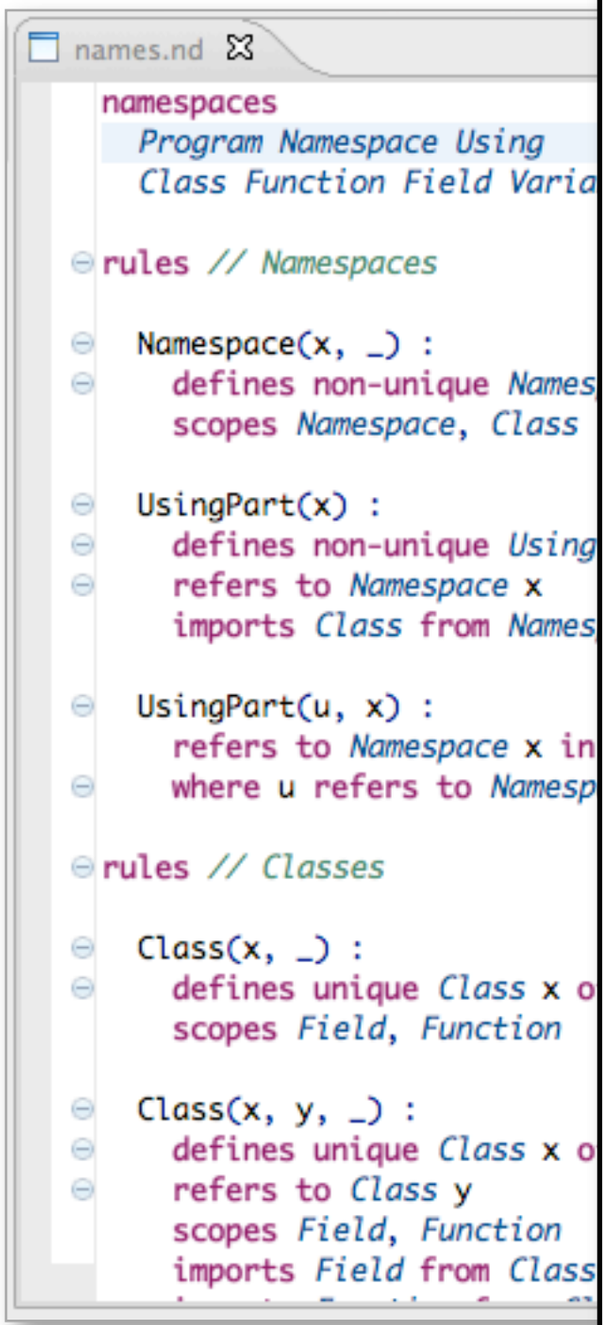
```
1  class Blog {
2    string post(User user, string message) {
3      string posterName;
4      posterName = user.;
5      return posterName;
6    }
7  }
8  class User {
9    string name;
10   string username;
11   string homepage;
12 }
```

homepage
name
username

```
1  class Blog {
2    string post(User user, string message) {
3      string posterName;
4      posterName = user.name;
5      return ;
6    }
7  }
8  class User
9    string na
10   string us
11   string ho
12 }
```

message
posterName
user

Name Binding Language in Spoofax (*)



derivation of editor services

checking

code completion

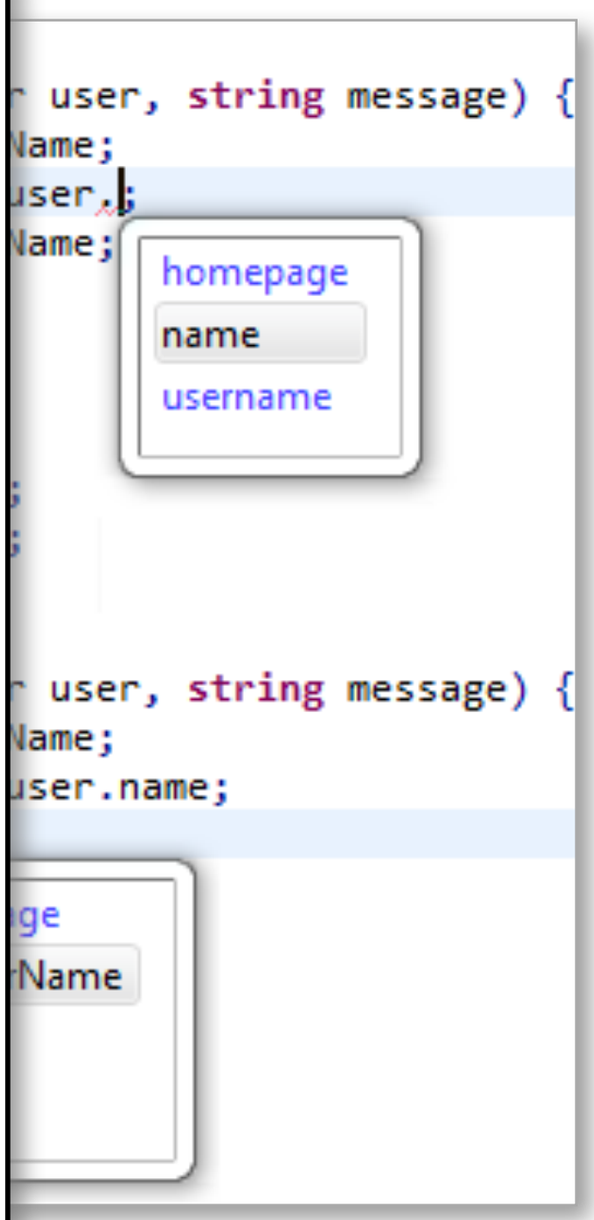
reference resolution

multi-file analysis

parallel evaluation

incremental evaluation

(*) alpha; NBL is compiled to Stratego implementation of name resolution algorithm (ask me for pointers)



Outlook

Language definition = CFG + NBD + TS + DS

Single source for

Language reference manual

Efficient (incremental) compiler

Execution engine (interpreter)

Integrated development environment