Onward! Paper:

A Language Designer's Workbench

A One-Stop-Shop for Implementation and Verification of Language Designs

Eelco Visser[1], Guido Wachsmuth[1], Andrew Tolmach[2], Pierre Neron[1], Vlad Vergu[1], Augusto Passalaqua[1], Gabriël Konat[1]

1 TU Delft Delft University of Technology

2 Portland State University

Syntax
with **SDF3**

- The syntax of a language defines the structure of the text representation of valid programs.
- A **parsing algorithm** that generates the abstract syntax tree from the text source code is usually the only definition of the syntax rules.
- SDF3 uses both templates, to define context free grammar productions including layout for pretty printing, and declarative rules for disambiguation.

Name Binding
with **NaBL**

- The name binding rules of a language describes how identifiers refer to their definition.
- A **resolution algorithm** is usually implicit and only appears inside the compiler or the type checker.
- NaBL uses rules relying on the following basic language independent concepts to identify definitions, references, and scopes to restrict the visibility of definitions.

Type System
with **TS**

- The type system assign types to the different elements of a programs and describes how these elements can be connected safely.
- A derived **type checking/inference** algorithm can be used in the IDE and the compiler to verify the static correctness of a program.
- TS inductive rules define the type system; these rules can refer to the type of the definitions from NaBL.

Dynamic Semantics
with **DynSem**

- The dynamic semantics of a language describe the dynamic behavior of the programs on a concrete machine.
- Often the **compiler** or **interpreter** implementation stands as the only definition of the dynamic semantics.
- In DynSem, the semantics are defined by declarative rules based on the framework of implicitly-modular structural operational semantics developed by P. Mosses.

PCF.sdf3

```
templates

  Exp.Var = [[ID]]
  Exp.App = [[Exp] [Exp]] {left}
  Exp.Fun = [
  fun [Param] (
      [Exp]
    )
  ]
  Exp.Fix = [
  fix [Param] (
     [Exp]
  )
  ]

context-free priorities
  Exp.App > Exp.Mul
  > {left: Exp.Add Exp.Sub}
  > Exp.Ifz
```

names.nab

```
namespaces Variable

binding rules

  Var(x) :
    refers to Variable x

  Param(x, t) :
  defines Variable x of type t

  Fun(p, e) :
    scopes Variable

  Fix(p, e) :
    scopes Variable

  Let(x, t, e1, e2) :
    defines Variable x
    of type t in e2
```

types.ts

```
type rules // binding

  Var(x) : t
  where definition of x : t

  Param(x, t) : t

  Fun(p, e) : FunType(tp, te)
  where p : tp and e : te

  App(e1, e2) : tr
  where e1 : FunType(tf, tr)
  and e2 : ta and tf == ta
  else error "type mismatch" on e2

  Fix(p, e) : tp
  where p : tp and e : te
  and tp == te
  else error "type mismatch" on p
```

semantics.ds

```
rules

  E env |- Var(x) --> v
  where env[x] --> T(e, env'),
        E env' |- e --> v

  E env |- App(e1, e2) --> v
  where
  E env |- e1 --> C(x, e, env'),
  E {x |--> T(e2, env), env'} |-
    e --> v

  E env |-
  Fun(Param(x,t),e) --> C(x,e,env)

  Ifz(e1, e2, e3) --> v
  where e1 --> I(i),
        i = 0,
        e2 --> v
```

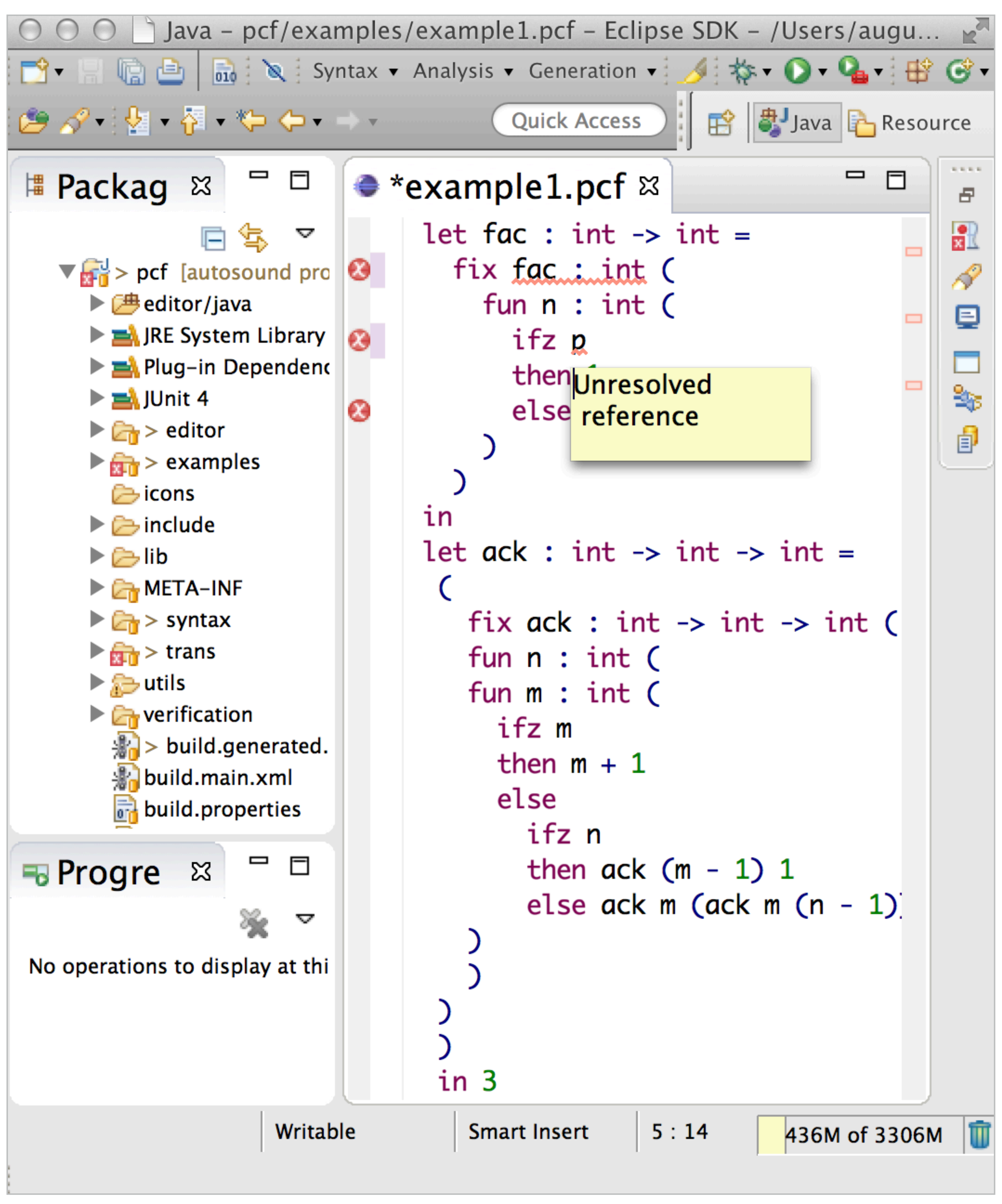
Automatic generation of new languages machinery from simple declarative rules

Development (Eclipse)

Easily write and edit programs
with an
Eclipse plugin for interactive development

Spoofax extends Eclipse to connect syntactic and semantics editor services. These services give feedback as the programmer types; they include:

- syntactic highlighting
- code views
- program navigation through references
- semantic code completion
- error detection (unresolved variables, type errors…)



Execution (Java)

Efficiently execute programs
with a
Java-based abstract syntax tree interpreter

Implicit structural operational semantics rules from DynSem are transformed into constructor specific rules:

```
  Ifz(e1, e2, e3) --> v
  where e1 --> I(i),
     [i = 0, e2 --> v] + [i≠0, e3 --> v]
```

The evaluation methods directly derive from such rules.

```
 public class Ifz_3_Node
   extends AbstractNode implements I_Exp
   public I_Exp _1, _2, _3;
   public Value evaluate
            (I_InterpreterFrame frame){
     I_InterpreterFrame env = frame;
     I_Exp e1 = this._1;
     I_Exp e2 = this._2;
     I_Exp e3 = this._3;
     Value v1 = e1.evaluate(env);
     if (v1 instanceof I_1_Node) {
       I_1_Node c_0 = (I_1_Node) v1;
       int i = c_0._1;
       if (i != 0) {
         return e3.evaluate(env);
       } else {
         if (i == 0) {
           return e2.evaluate(env);
         } else {
           throw
            new
             InterpreterException(
               “Premise failed”);
```

Verification (Coq)

Verify the correctness of the definitions
with a
model and proofs of type safety in Coq

The generated model includes:

- Term definition with a well-formedness predicate
- A lookup relation to represent name resolution
- An inductive typing predicate
- An inductive environment-based semantics relation.

```
Inductive C := (* constructors *)
| ParamC | FunC | FixC | AppC | IfzC …

Inductive term : Type :=
| Co : C → list term → term
| Id : I → term (* I is identifier type *)

Inductive ws_term : sort → term → Prop :=
| Co_ws cn s ss ts :
  sig cn = (ss,s) → (* sig is signature *)
  Forall2 ws_term ss ts →
  ws_term s (Co cn ts)…

Inductive wtyped : term → term → Prop := …

Inductive eval : env → term → val → Prop:=


Theorem type_preservation :
  forall e v ty,
   ws_term ExpS ExpS ⇒
    sound e ⇒
    eval e v ⇒
    wtyped e ty ⇒
    val_type v ty
```