Language Engineering Project

2017-2018

Eelco Visser

IN4333 Language Engineering Project
TU Delft
April 2018

TU Delft

Learning Goals

Prerequisites

- In the precursor compiler construction course (IN4303), students learn the basics of language engineering by building a complete definition for a small programming language.

Applying language engineering

- In this course, students learn to apply language engineering principles and tools to a real (domain-specific) programming language in a new project, i.e. without following a path set out in detail by the instructor.

Real world compilers

- Thus, they will experience that the design of a real programming languages requires trade-offs and compromises. Typically, students implement a previously existing language. But designing a new language is also an option provided there is a good plan that is discussed _before_ the course starts.

Language Engineering Project (IN4333)

Follow-up course to Compiler Construction (IN4303)

- Explore language design and implementation
- Implement a programming language or domain-specific language in Spoofax
- Projects are not pre-designed like the MiniJava project of the compilers course
- Investigate requirements of a programming language based on documentation, literature, etc.
- Focus on a particular aspect of language implementation
  - dive deeper into syntax definition, type systems, interpreters, or code generation

Today

- Course process and milestones
- Possible projects
- Reflection on study of compilers
- What is going on in language engineering research?

Process

Study Goals

- Learn to apply language engineering principles and tools to a real (domain-specific) programming language. Explore the definition of all aspects of a programming language: syntax, name binding, type analysis, transformations, code generation.

Literature and Study Materials

- Documentation of the design and implementation of a specific language
- Papers about language definition techniques

Education method

- This is a project course. Students deepen their language engineering skills and insights by building a complete language definition. Students work in teams of two on the definition of a (domain-specific) programming language using the Spoofax Language Workbench. Assistance and feedback is provided during weekly lab hours. The project should span the full life cycle of language implementation including a test suite, IDE, code generator, and distribution of the result as an Eclipse plugin.

Assessment

Assessment

- The work is assessed based on a code review of the language definition, a written report about the project, and a presentation in the final project workshop.

The final grade is based on the following components:

- Github repository with language project (40%)
- Written report about the project (30%)
- Presentation (slides) (30%)

Schedule

Weekly labs on Friday morning in DW-PC 1(010BG)

- Get assistance with technical issues
- Discuss design questions

Milestones

- M1: Stage complete
- M2: Core language
- M3: Maximize coverage
- M4: Plugin
- M5: Presentation
- M6: Paper

Workshop

- Presentation with demonstration for each project
- Date: Wednesday June 27

Code Review

- Date: Friday June 29

Schedule

- April 25: Kick-off (today): Select project
- April 26: Project start
- April 27: Koningsdag
- May 4*: Lab 1: M1: Stage Complete
- May 11: Hemelvaart
- May 18: Lab 2
- May 25*: Lab 3: M1: Core Language
- June 1: Lab 4
- June 8: Lab 5
- June 15*: Lab 6: M2: Maximize coverage
- June 22: Lab 7: M3: Plugin
- June 27*: Workshop: Presentation with demonstration (M3)
- June 29*: Code Reviews
- July 13: Deadline for submitting paper, slides, code

Beyond the Lab

Lab is not enough

- Course = 140 hours
- Labs = 28 hours
- Kick-off lecture = 2 hours
- Workshop = 8 hours
- Code review = 1 hour (max)
- => 101 hours of home work

Communication

- #spoofax-users channel on Slack
- Email: e.visser@tudelft.nl, j.denkers@tudelft.nl

MetaBorgCube

- Code in repository on github
- Continuous integration in Travis (Jasper?)

Milestones

M0: Project Selection

Select a project

- Coordinate with me
- One team per project option

Set objective

- What is the goal of the project?
- What stages should your language definition support?
  - Syntax definition (always)
  - Desugaring
  - Type system
  - Interpreter
  - Code generator
  - Optimizer
  - Refactoring

M1: Stage Complete

Github repository

- MetaBorgCube: https://github.com/metaborgcube
- We will provide a repository

Stage complete

- Spoofax project set-up
- A minimal definition for each stage of language definition
- Syntax, statics, dynamics, code generation, …
- Test set up for each stage
- Stages depend on goal of your project

Target language

- If you make a code generator:
- what is your target language?
- how to generate code? (string templates, AST + pretty-printer)

M2: Core Language

Identify a core language

- Essential core of of the language
- Complete enough to be able to `run' programs
- (run = depends on goal of your project)

Implement all stages for core language

- Syntax definition
- Type system
- Interpreter
- …

M3: Maximize Coverage

Extend coverage of language definition

- Extend core language towards full definition of the language
- Consider interesting features of the language to support

Additional editor services

- Refactoring?

Testing

- Define positive and negative test cases

M4: Plugin

Plugin

- Produce an Eclipse plugin that can be used by others

Mopping up

- Pay your technical debt
- Remove dead code
- Terminology, naming
- Clean up code (structure, formatting): is code readable, accessible?

Documentation

- README with architecture overview, instructions, sources, etc.
- Comments where necessary

M5: Presentation with Demonstration

Language

- Language background: domain, features,

Demonstration

- Show us that/how the plugin works

Language definition

- Coverage
- Limitations
- Technical issues
- Interesting encodings
- …

M6: Paper

Write a paper about your project

- What were the objectives?
- What have you achieved?
- What problems did you encounter?
- What lessons did you learn about programming languages, compilers, …?
- What interesting techniques / encodings / … did you apply / develop?
- Not: a listing of all your code

Target Software Language Engineering (SLE) conference

- https://conf.researchr.org/track/sle-2018/papers
- Use the submission format for SLE18: 12 pages in ACM SIGPLAN Conference Format LaTeX style

Start early

- Set up the paper now, collect observations as you go

Assessment

Soundness

- Is the language definition correct?
- Does editor give correct error messages?
- Does execution produce correct results
- => Tests

Completeness

- To what extent does language definition cover the language?

Usability

- Does the IDE provide useful error messages and navigation?

Code quality

- Is the code readable, understandable, and well organised?
- micro-level: the names and formatting used for individual rules
- macro-level: the organization of modules in the project
- Does the project build from the source in the git repository?

What is a Compiler?

What is a Programming Language?

What is a programming language for?

- Programming machines to solve problems for humans

How do programming languages help us achieve that?

- Abstraction: programmer can ignore irrelevant details
- Safety: guarantee safety properties
- Performance: develop and run programs efficiently
- => language design principles

How do you make a programming language?

- Define syntax and semantics
- Implement a compiler and/or interpreter and/or IDE that incorporate syntax and semantics

Compiler Architecture: A Standardized Pipeline?

Parsing

- from text to abstract syntax tree

Type checking

- statically check types of operators

Further static analysis

- check other context-sensitive properties (e.g. non-null)

Optimizations

- transform programs to improve performance or memory usage

Code generation

- translate program to low-level target code

Language run-time system

- components needed at run time such as garbage collection

Variability in Compiler Architecture

Parsing

- Many parsing algorithms (LL, LR, GLL, GLR, PEG), parser generators, parser combinators

Type checking

- direct traversal, constraints, attribute grammars, symbol tables, AST as symbol table (reference attributes)

Further static analysis

- many specialized algorithms for specific analyses for specific languages

Optimizations

- peep-hole optimization, loop fusion, inlining, constant propagation, code motion, and many more

Code generation

- directly target native assembly language for some architecture, target common IR (LLVM) with back-ends for different architectures, target virtual machine (JVM), generate code in high-level language (C, Java)

Language run-time system

- garbage collection algorithms, run-time checks

Studying Compiler Construction

Traditional Perspective

- Language = implementation of compiler
- Important: algorithms to implement compiler components
- Better algorithms =>
  - faster compiler
  - faster run-time
  - better analysis
  - more reliable analysis
- Increase reach of soundness / completeness
- Compiler Construction = Selection and implementation of algorithms for each compiler

Language Workbench Perspective

- Separate language *definition* from language *implementation*
- Definition using high-level declarative meta-language
  - language engineer reasons about design of language
- Automatically derive implementation from definition in meta-language
  - language workbench engineer reasons about implementation of meta-language

Current Language Engineering Research

"Software systems are the engines of modern information society. Our ability to cope with the increasing complexity of software systems is limited by the programming languages we use to build them. Bridging the gap between domain concepts and the implementation of these concepts in a programming language is one of the core challenges of software engineering. Modern programming languages have considerably reduced this gap, but often still require low-level programmatic encodings of domain concepts. Or as Alan Perlis formulated it in one of his famous epigrams: '**A programming language is low level when its programs require attention to the irrelevant**'. A fixed set of (Turing Complete) programming constructs is sufficient to express all possible computations, but at the expense of considerable encoding that obfuscates the concepts under consideration. **Linguistic abstraction can be used as a tool to capture our emerging understanding of domains of computation.**"

E. Visser,
Understanding Software through Linguistic Abstraction.
Science of Computer Programming (2015)

(Declarative Language) Definition

Declarative Languages

- Design domain-specific languages that abstract from irrelevant details (for that domain)

WebDSL

- Web programming abstractions for data modelling, access control, page templates
- Static checking of consistency

IceDust

- Declarative data modelling with relations, native multiplicities, derived values, derived relations
- Generation of different implementation strategies

LeQuest

- Specification of interfaces of medical equipment in order to generate web-based training software

Oracle

- Green-Marl, PGQX: graph processing/querying languages

Océ

- DSLs for the digital printing domain

PixieDust: Models

```
model
  entity TodoList {
    todos : Todo* (inverse = Todo.list)
  }
  entity Todo {
    description : String
    finished    : Boolean
  }

view
  TodoList.view = div { ul { todos.itemView } }

  Todo.itemView = li {
    input[type="checkbox", value=finished]
    span { description }
  }
```

Ten Veen, Harkes, Visser. PixieDust: Declarative Incremental User Interface Rendering Through Static Dependency Tracking [WWW18]

PixieDust: Views

```
view
  TodoList {
    view : View = div {
      header
      ul { visibleTodos.itemView }
      footer
    }

    header : View = div {
      h1 { "Todos" }
      input[type="checkbox", value = allFinished,
                             onClick = toggleAll]
      StringInput[onClick = addTodo](input)
    }

    footer : View = div {
      todosLeft "items left"
      ul{
        visibilityButton(this, "All")
        visibilityButton(this, "Finished")
        visibilityButton(this, "Not finished")
      }
      if(count(finishedTodos) > 0)
        button[onClick = clearFinished]
    }
  }
  Todo {
    itemView : View = li { div {
      BooleanInput(finished)
      span { task }
      button[onClick=deleteTodo] { "X" }
    }}
  }
```

```
view
  TodoList {
    input : String = (init = ""    )
    show  : String = (init = "All")

    finishedTodos : Todo* =
      todos.filter(todo => todo.finished)
      (inverse = Todo.inverseFinishedTodos?)

    visibleTodos : Todo* =
      switch {
        case show == "All"      => todos
        case show == "Finished" => finishedTodos
        default => todos \ finishedTodos
      }
      (inverse = Todo
  }
```

```
view
  Todo {
    actions {
      toggleTodo: finished := !finished
      deleteTodo: list := null
    }
  }
  TodoList {
    actions {
      addTodo:
        todos += {description = input
                  finished = false}
        input := ""

      toggleAll: todos.finished := !allFinished
      clearFinished: todos -= finishedTodos
      setVisibility(to: String): show := to
    }
  }
```

Declarative (Language Definition)

Goal: A Language Designer's Workbench

- High-level declarative specification of syntax, static semantics, dynamic semantics
- Implementation-independent understanding
- Facilitate reasoning about / verification of properties
- Generate wide range of tools from single definition

Declarative Meta-Languages

- Tuned for sub-domain of language definition

SDF3: Declarative Syntax Definition

Declarative Syntax Definition

- Understand syntax in terms of structure
- Disambiguation separated from grammar productions (priority, associativity)

Recent and Ongoing Research

- Principled syntactic code completion using placeholders
- Safe and complete disambiguation of deep priorities
- Layout constraints for layout sensitive languages
- Performance of generalized parsing
- Incremental generalized parsing

Deep Priority Conflicts

workspace-tiger - disambdemo/examples/example02.aterm - Eclipse

disamb-demo.sdf3

```
module disamb-demo

imports Common

context-free start-symbols Exp

context-free syntax

  Exp.Int     = INT
  Exp.Var     = ID
  Exp.Add     = <<Exp> + <Exp>> {left}
  Exp.If      = <if(<Exp>) <Exp>>
  Exp.IfElse  = <if(<Exp>) <Exp> else <Exp>>
  Exp         = <(<Exp>)> {bracket}

  Exp.Match   = <match <Exp> with <Pat+>> {longest-match}
  Pat.Clause  = [[ID] -> [Exp]]

//context-free priorities
//
//  Exp.Add > Exp.IfElse > Exp.If > Exp.Match
```

example03.disam

```
a + if(1) 2 + 3
```

example01.aterm | example03.aterm

```
amb(
  [ Add(
      Add(Var("a"), If(Int("1"), Int("2")))
    , Int("3")
    )
  , Add(
      Var("a")
    , If(Int("1"), Add(Int("2"), Int("3")))
    )
  ]
)
```

example02.disam

```
if(1) if(2) 3 else 4
```

example02.aterm | example04.aterm

```
amb(
  [ IfElse(
      Int("1")
    , If(Int("2"), Int("3"))
    , Int("4")
    )
  , If(
      Int("1")
    , IfElse(Int("2"), Int("3"), Int("4"))
    )
  ]
)
```

Writable | Insert | 6 : 6

Deep Priority Conflicts

workspace-tiger - disambdemo/syntax/disamb-demo.sdf3 - Eclipse

disamb-demo.sdf3

```
module disamb-demo

imports Common

context-free start-symbols Exp

context-free syntax

  Exp.Int     = INT
  Exp.Var     = ID
  Exp.Add     = <<Exp> + <Exp>> {left}
  Exp.If      = <if(<Exp>) <Exp>>
  Exp.IfElse  = <if(<Exp>) <Exp> else <Exp>>
  Exp         = <(<Exp>)> {bracket}

  Exp.Match   = <match <Exp> with <Pat+>> {longest-match}
  Pat.Clause  = [[ID] -> [Exp]]

context-free priorities

  Exp.Add > Exp.IfElse > Exp.If > Exp.Match
```

example01.disam

```
if(1 + 2)
  if(3 + 4 + 5)
    6 + 7
  else
    8 + 9
```

example04.disam

```
match x with
  z -> 1 + match 5 with
          a -> a + x
          y -> 3 + 4
```

example01.aterm | example03.aterm

```
If(
  Add(Int("1"), Int("2"))
, IfElse(
    Add(Add(Int("3"), Int("4")), Int("5"))
  , Add(Int("6"), Int("7"))
  , Add(Int("8"), Int("9"))
  )
)
```

example02.aterm | example04.aterm

```
Match(
  Var("x")
, [ Clause(
      "z"
    , Add(
        Int("1")
      , Match(
          Int("5")
        , [ Clause("a", Add(Var("a"), Var("x")
          , Clause("y", Add(Int("3"), Int("4")
          ]
        )
      )
    )
  ]
```

Writable | Insert | 23 : 2

Layout Sensitive Syntax

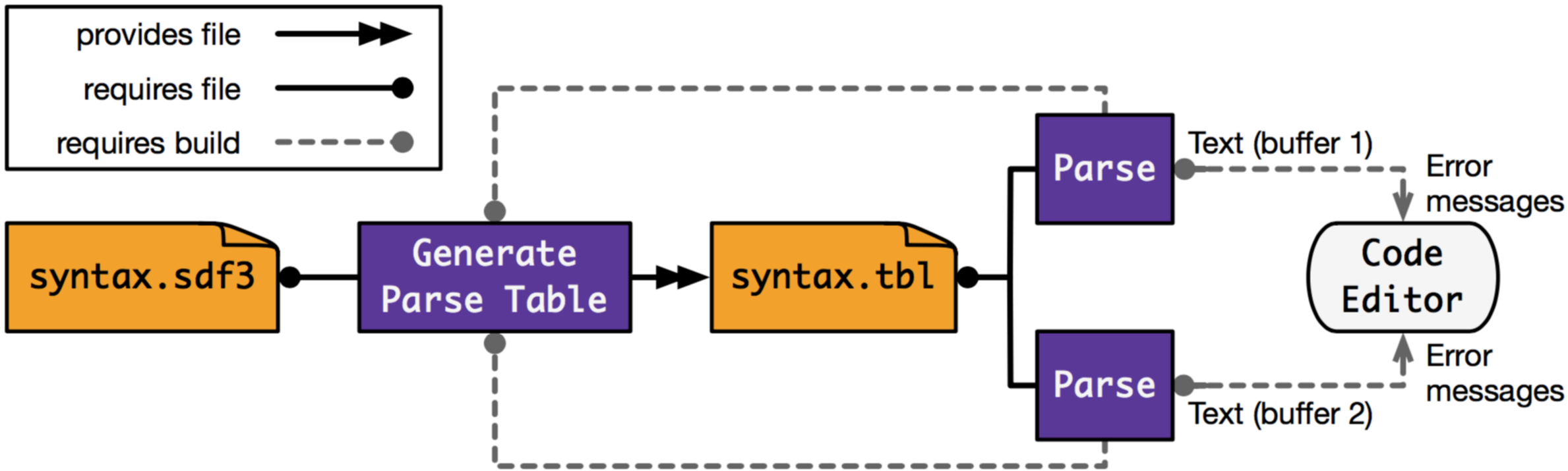
Model syntax of languages with layout sensitive syntax

- Haskell
- Python
- Ruby
- ...

Using *layout constraints* in SDF3

- Work in progress
- Prototype implementation in Spoofax Nightly

PIE: Incremental Interactive Pipelines



Konat, Erdweg, Visser. PIE: A Domain-Specific Language for Interactive Software Development Pipelines [<Programming> 2018]

```
class GenerateTable extends Builder<File, Out<File>> {
  static BuilderFactory<File, Out<File>, GenerateTable> factory =
    BuilderFactoryFactory.of(GenerateTable.class, File.class);
  GenerateTable(File syntaxFile) { super(syntaxFile); }
  @Override File persistentPath(File syntaxFile) {
    return new File("generate-table-" + hash(syntaxFile));
  }
  @Override Out<File> build(File syntaxFile) throws IOException {
    require(syntaxFile); File tblFile = generateTable(syntaxFile);
    provide(tblFile); return OutputPersisted.of(tblFile);
} }
class Parse extends Builder<Parse.Input, Out<ParseResult>> {
  static class Input implements Serializable {
    File tblFile; String text; BuildRequest tblReq;
    Input(File tblFile, String text, BuildRequest tblReq) {
      this.tblFile = tblFile; this.text = text; this.tblReq = tblReq;
    }
    boolean equals(Object o) {/* omitted */} int hashCode() {/* omitted */}
  }
  @Override Out<ParseResult> build(Input input) throws IOException {
    requireBuild(input.tblReq); require(input.tblFile);
    return OutputPersisted.of(parse(input.tblFile, input.text));
} } /* ... other required code omitted ... */
class UpdateEditor extends Builder<String, Out<ParseResult>> {
  @Override Out<ParseResult> build(String text) throws IOException {
    File syntaxFile = new File("syntax.sdf3");
    File tblFile = requireBuild(GenerateTable.factory, syntaxFile).val;
    BuildRequest tblReq = new BuildRequest(GenerateTable.factory, syntaxFile);
    return requireBuild(Parse.factory, new Parse.Input(tblReq, tblFile, text));
} } /* ... other required code omitted ... */
```

PIE pipeline for parser generation and editor updates

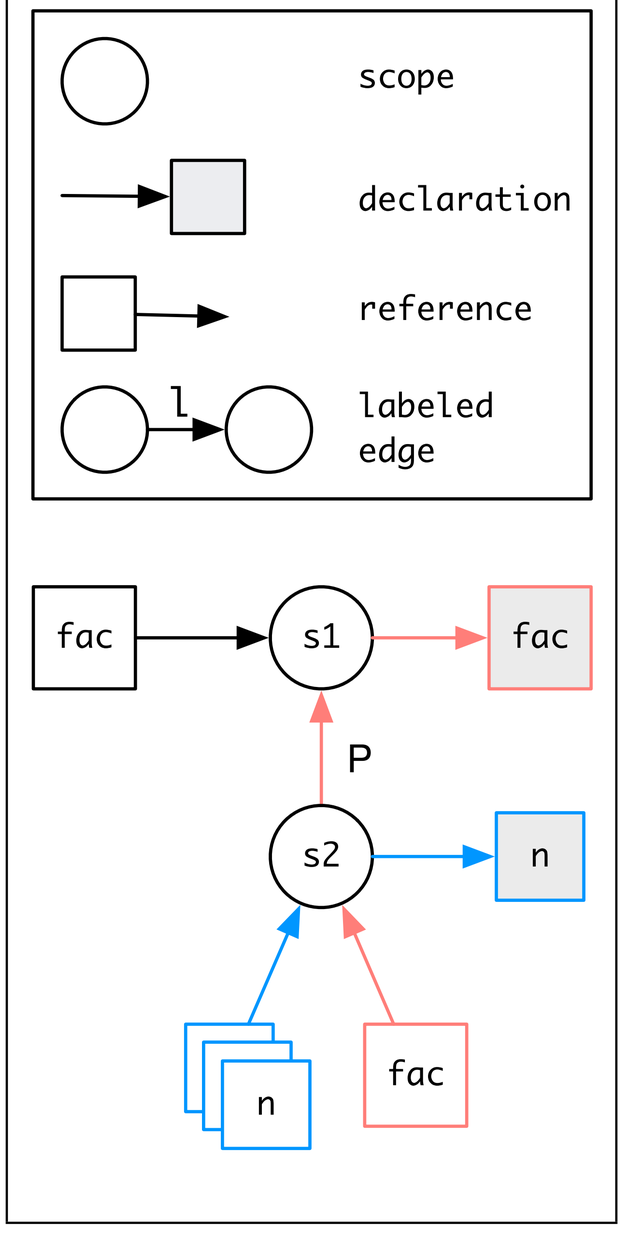
FlowSpec: Specification of Data-Flow Analysis [SLE17]

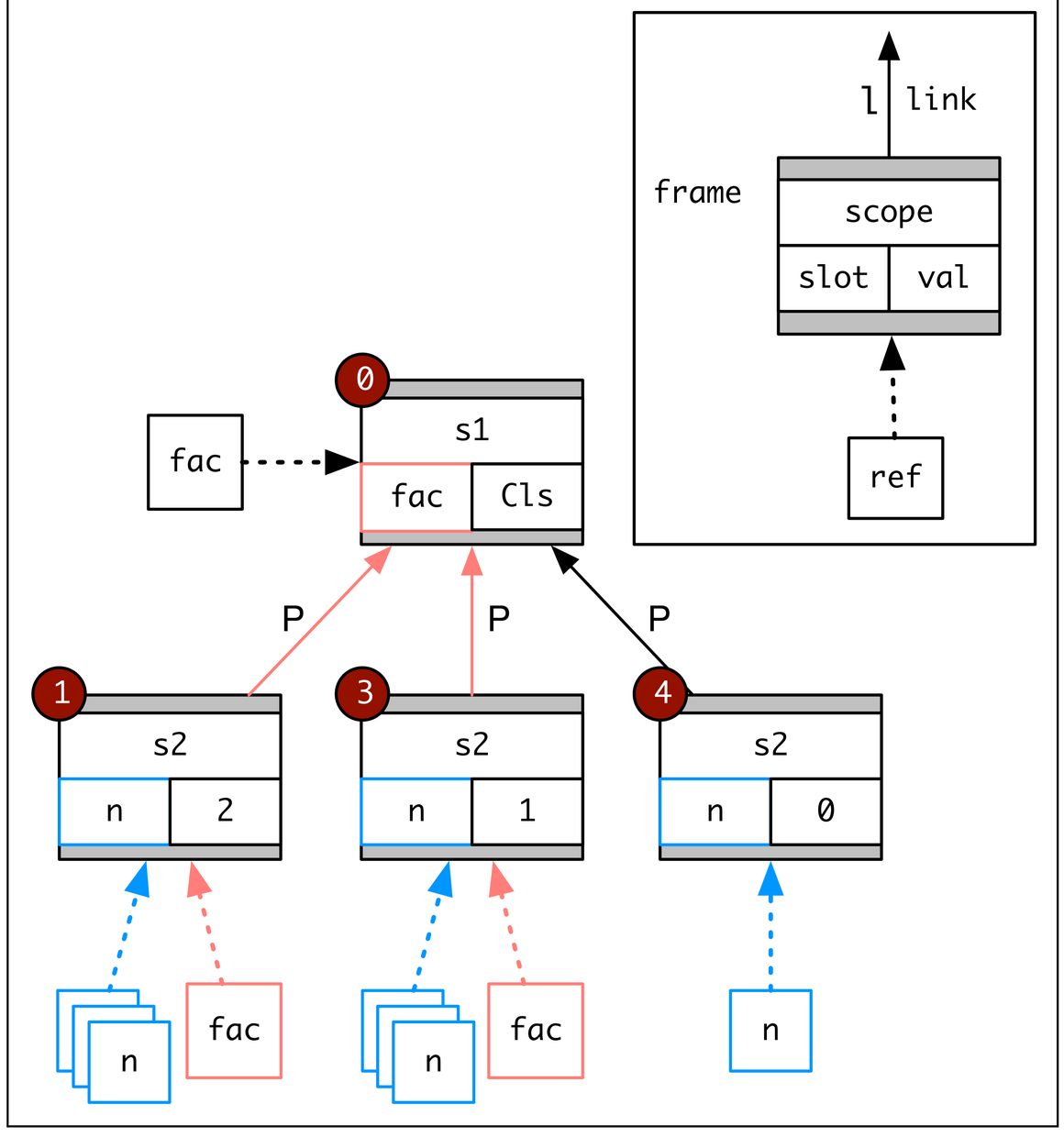
```
module control

control-flow rules
  Assign(_, e) = entry → e → this → exit

  Seq(s1, s2) = entry → s1 → s2 → exit

  IfThenElse(c, t, e) =
    entry → c → t → exit,
            c → e → exit

  While(c, b) = entry → c → b → c → exit

  BinOp(_, l, r) = entry → l → r → this → exit

  UnOp(_, e) = entry → e → this → exit

node VarRef(_)
node IntLit(_)
node True()
node False()
```

```
module availability

properties
  available: MustSet(term)
  external
    refs: Set(name)
property rules
  available(prev → Assign(n, _)) =
    { expr |
      expr ← available(prev),
      !(Var{n} in refs(expr) }

available(prev → e@BinOp(_,_,_)) =
  available(prev) ∪ {e}

available(prev → e@UnOp(_,_,_)) =
  available(prev) ∪ {e}

available(prev → _) = available(prev)
```

Name Binding: Scopes and Frames

```
letrec fac = 0                                   s1
  fun (n : Int) : Int { 2                        s2
    if (n == 0) {
      1 5
    } else {
      n * fac(n - 1) 3 4
    }
  }
in
  fac(2) 1
```





Intrinsically-Typed Definitional Interpreters [POPL18]

```
data Expr (Γ : Ctx) : Ty → Set where
  bool  :  Bool → Expr Γ bool
  num   :  ℤ → Expr Γ int
  var   :  ∀ {t} → t ∈ Γ → Expr Γ t
  if    :  ∀ {t} → Expr Γ bool →
           Expr Γ t → Expr Γ t →
           Expr Γ t
  plus  :  Expr Γ int → Expr Γ int →
           Expr Γ int

data Ty : Set where
  bool : Ty
  int  : Ty

Ctx = List Ty
```

```
data Val : Ty → Set where
  bool  :  Bool → Val bool
  num   :  ℤ → Val int

Env : Ctx → Set
Env Γ = All Val Γ

eval : ∀ {Γ t} → Expr Γ t → Env Γ → Val t
eval (bool b)   E    =  bool b
eval (num x)    E    =  num x
eval (var x)    E    =  lookup E x
eval (if c t e) E    =  case (eval c E) of λ{ (bool b) →
                        if b then (eval t E) else (eval e E)}
eval (plus e₁ e₂) E  =  case (eval e₁ E) of λ{ (num z₁) →
                        case (eval e₂ E) of λ{ (num z₂) →
                        num (z₁ + z₂) }}
```

Bach Poulsen, Rouvoet, Tolmach, Krebbers, Visser. Intrinsically-Typed Definitional Interpreters for Imperative Languages. POPL18

Intrinsically-Typed Definitional Interpreters [POPL18]

```
data Expr (s : Scope) : Ty → Set where
  unit  : Expr s unit
  var   : ∀ {t} → (s ↦ t) → Expr s t
  lam   : ∀ {s′ t u} → ⦃ shape : g s′ ≡ ( [ t ] , [ s ] ) ⦄ → Expr s′ u → Expr s (t ⇒ u)
  _·_   : ∀ {t u} → Expr s (t ⇒ u) → Expr s t → Expr s u
```

```
data Val : Ty → (Σ : HeapTy) → Set where
  unit  :  ∀ {Σ} → Val unit Σ
  ⟨_,_⟩ :  ∀ {Σ s s′ t u}⦃ shape : g s′ ≡ ([ t ] , [ s ]) ⦄ →
           Expr s′ u → Frame s Σ → Val (t ⇒ u) Σ
```

Binding with scopes and frames

```
eval : ℕ → ∀ {s t Σ} → Expr s t → M s (Val t) Σ
eval zero    _          =  timeout
eval (suc k) unit       =  return unit
eval (suc k) (var x)    =  getv x
eval (suc k) (lam e)    =  getFrame >>= λ f → return ⟨ e , f ⟩
eval (suc k) (e₁ · e₂)  =  eval k e₁ >>= λ{ ⟨ e , f ⟩ →
                           (eval k e₂ ^ f) >>= λ{ (v , f) →
                           init (sₑ e) (v :: []) (f :: []) >>= λ f′ →
                           usingFrame f′ (eval k e) }}
```

```
sₑ : ∀ {s t} → Expr s t → Scope
sₑ {s} _ = s
```

Statix: Static Semantics with Scope Graphs

NaBL2

- Generate scope graph + unification constraints from AST
- Special relations for storing program facts such as sub typing
- Language-independent constraint solver
- Cannot model parameterized types, structural types

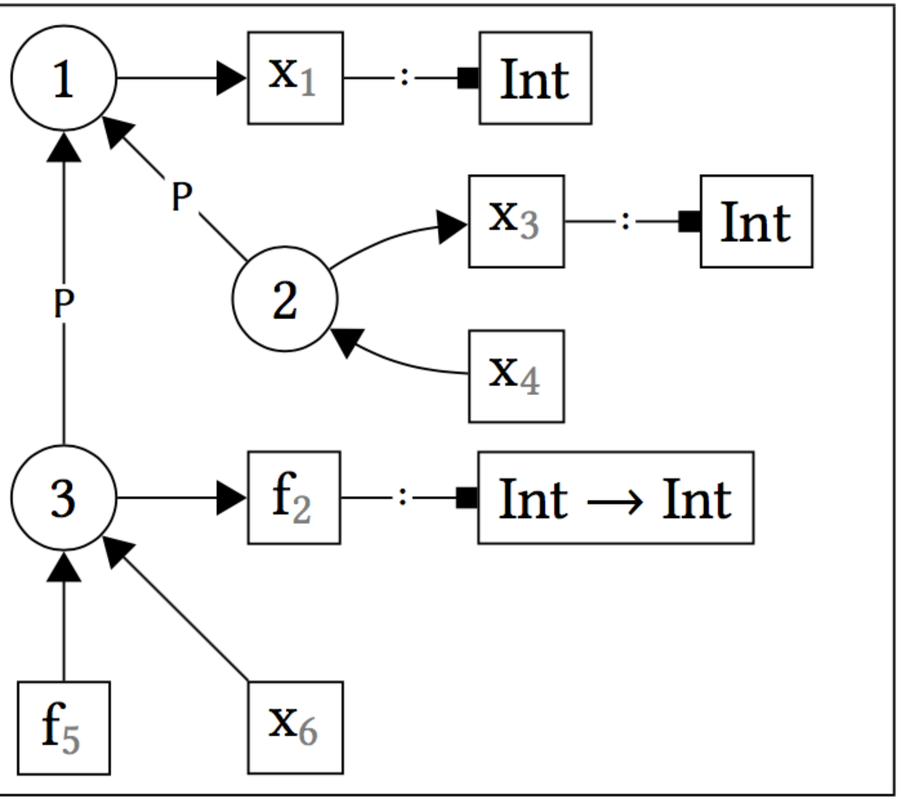
Statix

- User-defined constraints
- Define type comparison predicates
- Scoped relations
- Generalized scope graph queries
- Prototype implementation (syntax + type checking) in Spoofax Nightly

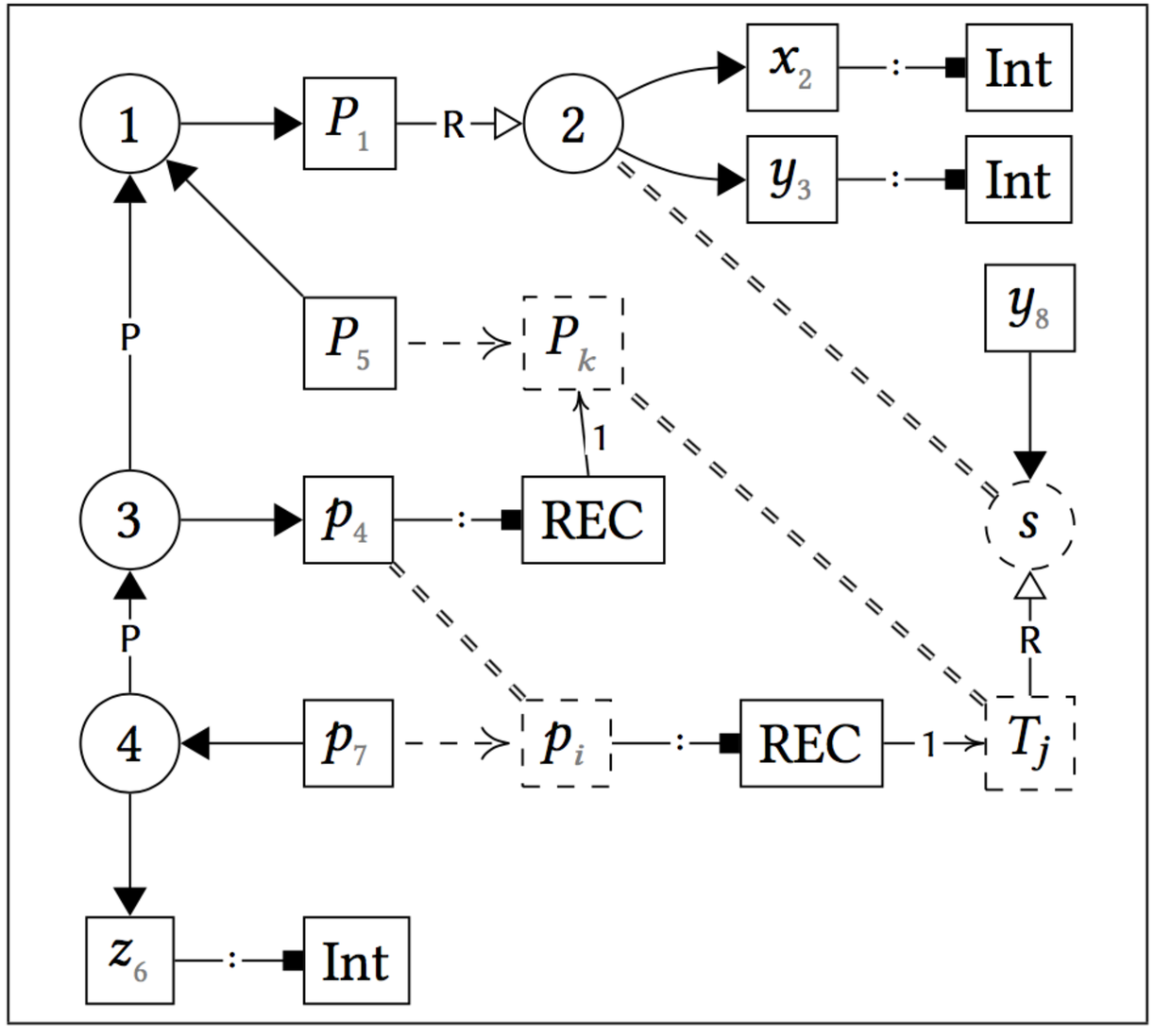
[Under submission]

Scopes with Types

```
let x1 = 14
    f2 = fun (x3: Int) { x4 + 1 }
in f5 x6
```

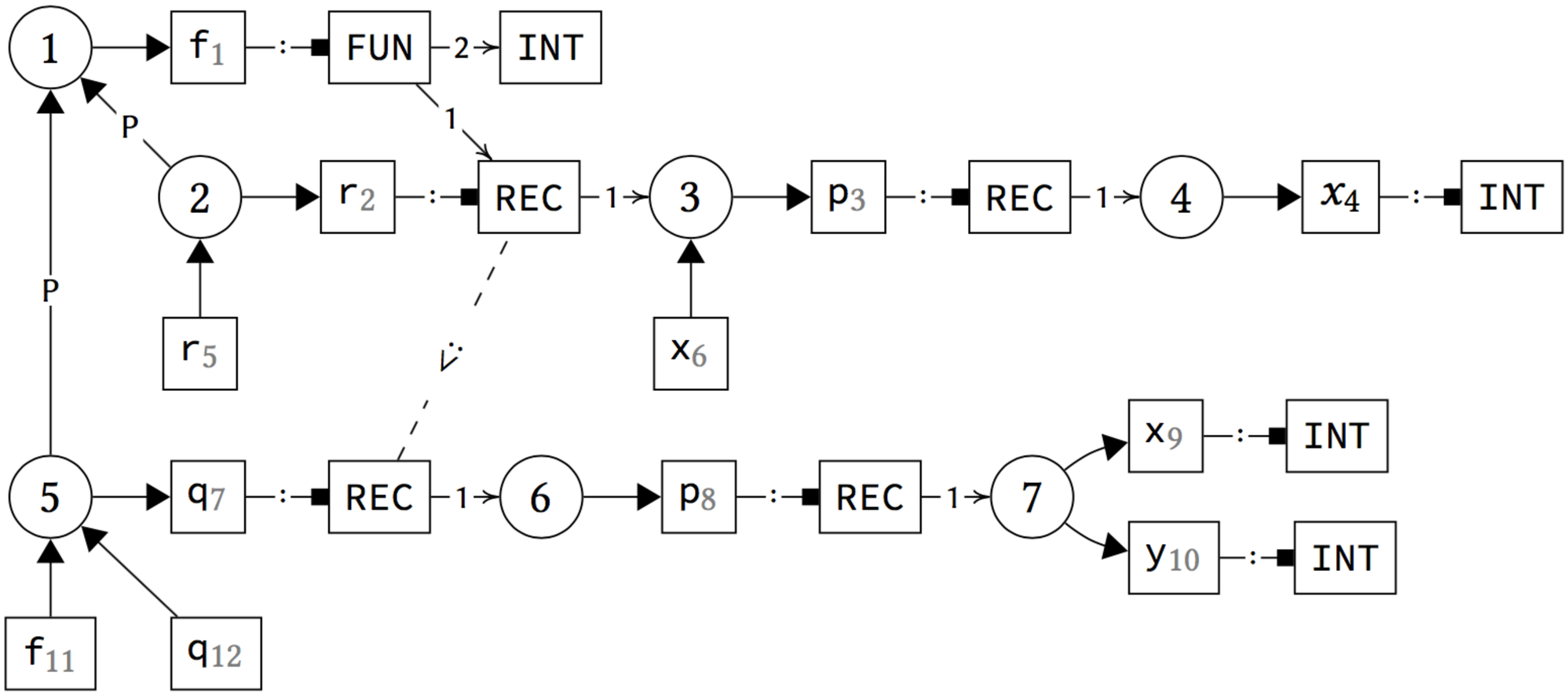



```
record P1 { x2: Int, y3: Int }
def p4 : Rec P5 = ...
def z6 : Int = p7.y8
```



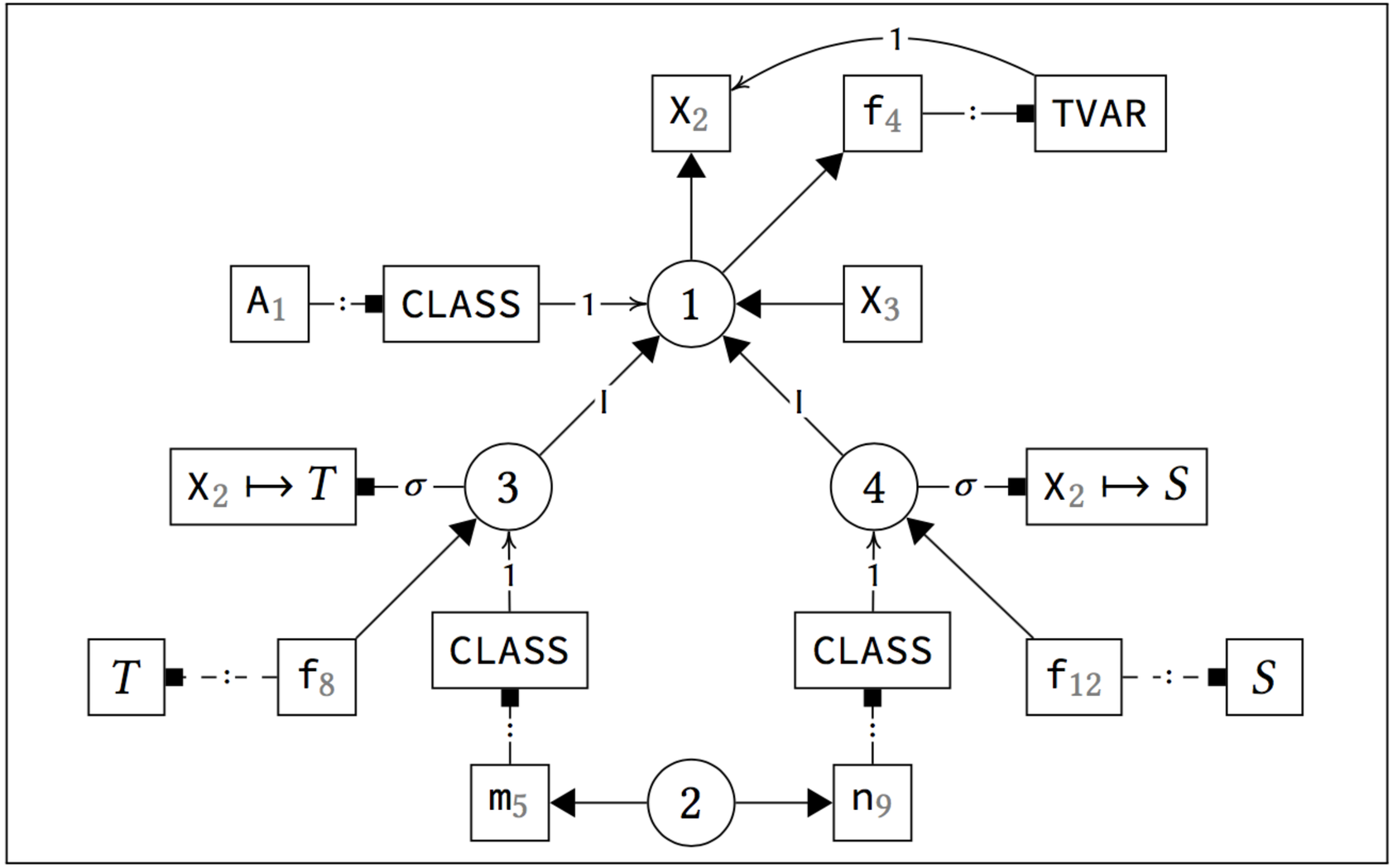
Scopes as Types: Modeling Structural Types

```
let f1 = fun(r2: {p3: {x4: Int}}) { r5.x6 }
    q7 = { p8 = { x9 = -7, y10 = 10 } }
in f11 q12
```



Scopes as Types: Modeling Generic Classes

```
class A1<X2> {
    X3 f4;
    ...
}
...
m5 = new A6<T>();
m7.f8;
n9 = new A10<S>();
n11.f12;
...
```



```
module stlcrec
// Static semantics of Simply Typed Lambda Calculus
signature
  sorts ID = string
  sorts BinOp
  sorts Exp constructors
    Num     : ID -> Exp
    BinExp  : Exp * BinOp * Exp -> Exp
    Fun     : ID * TypeExp * Exp -> Exp
    Var     : ID -> Exp
    App     : Exp * Exp -> Exp
  sorts TypeExp constructors
    NumType : TypeExp
    FunType : TypeExp * TypeExp -> TypeExp
  sorts Type constructors
    NUM   : Type
    FUN   : Type * Type -> Type
  relations
    typeOfDecl : occurrence -> Type
  namespaces
    Var  : string
  name-resolution
    labels P R
    resolve Var filter pathMatch[P*] min pathLt[$ < P]
```

```
rules

  typeOfExp : scope * Exp ->  Type

  typeOfExp(s, Num(_)) = NUM().

  typeOfExp(s, BinExp(e1, _, e2)) = NUM() :-
    typeOfExp(s, e1) == NUM(),
    typeOfExp(s, e2) == NUM().

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

  typeOfExp(s, Var(x)) = T :-
    typeOfDecl of Var{x@x} in s |-> [(_, T)].

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

```
rules

  typeOfTypeExp : scope * TypeExp ->  Type
  typeOfTypeExp(s, NumType()) = NUM().
  typeOfTypeExp(s, FunType(te1, te2)) = FUN(typeOfTypeExp(s, te1), typeOfTypeExp(s, te2)).

rules

  subType : Type * Type
  subType(NUM(), NUM()).
  subType(FUN(S1, T1), FUN(S2, T2)) :- subType(S2, S1), subType(T1, T2).
```

2017-2018 Projects

Research Angle

Case studies with Spoofax

- Make a neat and complete specification of (some aspect of) an existing language

Experimental

- Testing the edges of expressiveness of Spoofax

Type of Projects

Existing Language

- Implement an existing language
- Implementing helps in getting understanding of language
- Case study of Spoofax meta-languages

Language Extension

- Extend an existing language already implemented in Spoofax with new features

Existing Language Project

- Reimplement a language already implemented in (a previous version of) Spoofax
- Basic issues already explored
- Re-iteration can improve, use new features of Spoofax

New Language

- Design and implement a new (domain-specific) language
- Requires a well-developed idea to succeed in one quarter

2018 Project Themes

Layout sensitive syntax

- Make a solid language definition

Structural type systems

- Typescript

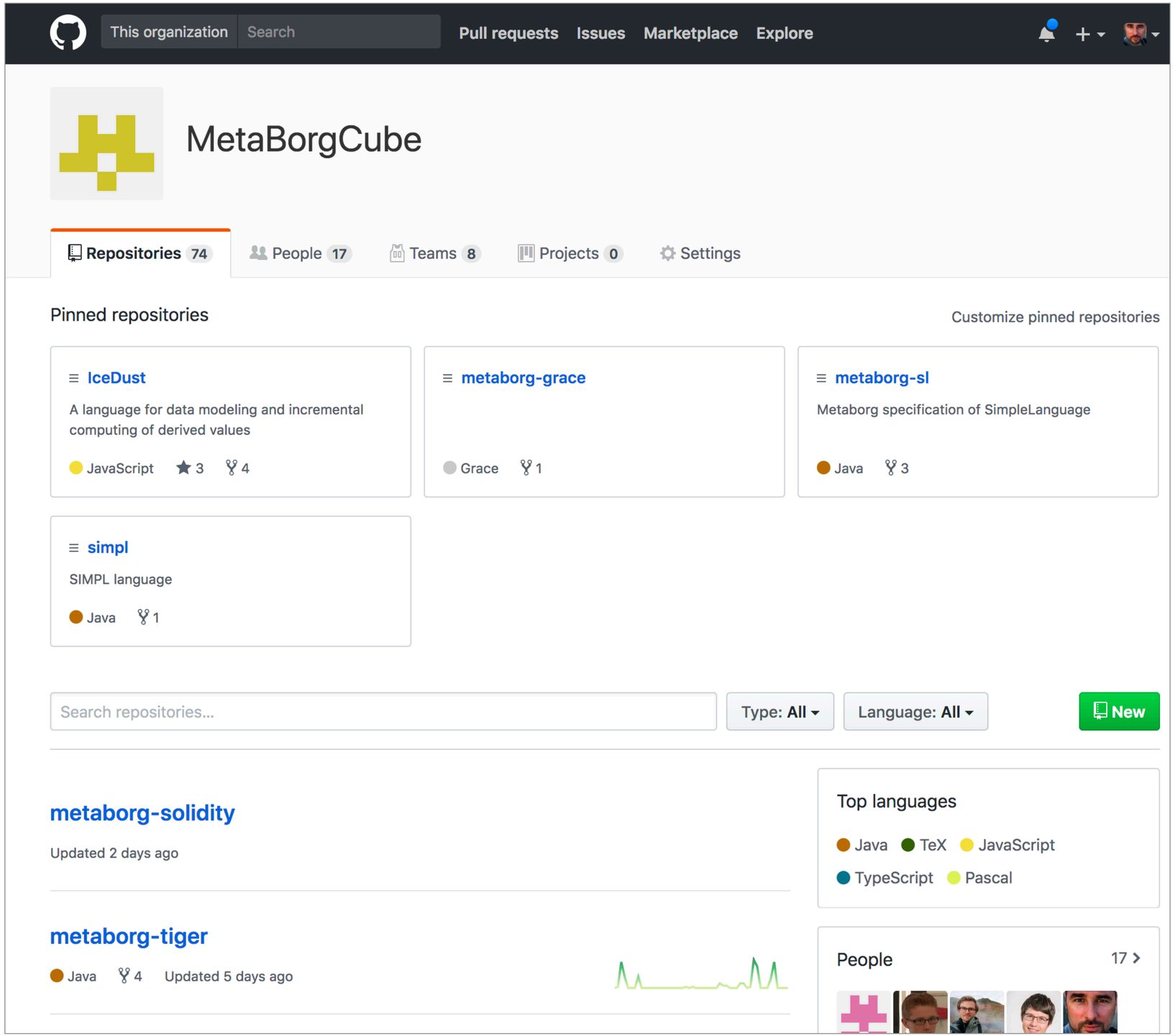
Gradual type systems

- What does that require?

Smart contracts

- Solidity

https://github.com/MetaBorgCube



Ada

Ada

- **Ada** is a structured, statically typed, imperative, wide-spectrum, and object-oriented high-level computer programming language, extended from Pascal and other languages.
- Commissioned by US Department of Defense around 1980
- https://en.wikipedia.org/wiki/Ada_(programming_language)

Nexperia

- https://www.nexperia.com/
- Uses Ada to program real-time embedded software

LEP 2017-2018

- Solid and complete syntax definition in SDF3
- Type system in Statix
- State machine DSL to Ada
- Basis for developing scriptable refactorings (large research project)

Python Syntax Definition

Python

- **Python** is an interpreted high-level programming language for general-purpose programming.
- Developed by by Guido van Rossum at CWI in Amsterdam in 1991

LEP 2017-2018

- Layout sensitive syntax
  - Case study for SDF3 with layout constraints
- Name binding
  - Python has ‘interesting’ name binding constructs
  - Definition in NaBL2 or Statix

TypeScript: Structural Types

TypeScript = JavaScript + Types

- Developed by Microsoft

LEP 2016-2017

- TypeScript prototype
- Small syntax definition
- Static semantics in NaBL2
- Limitation: expression of structural types
- https://github.com/MetaBorgCube/metaborg-typescript

LEP 2017-2018

- Solid syntax definition
- Type system in Statix
  - Caveat: solver for Statix under development
- Define critical test suite for testing structural types

C#

C# = Java for Windows

- Developed by Microsoft

NaBL case study

- Used small subset of C# as case study for NaBL1 design
- https://github.com/MetaBorgCube/spoofax-csharp

LEP 2017-2018

- Complete syntax definition
- Type system in Statix
  - Caveat: solver for Statix under development
- Code generation (to …?)

Océ

- Target for code generation at Océ

Go

Go = Cleaner systems programming

- Go is a statically typed language in the tradition of C, with memory safety, garbage collection, structural typing,[3] and CSP-style concurrent programming features added.
- Developed by Google

LEP 2015-2016

- Go to JavaScript compiler for a subset
- https://github.com/MetaBorgCube/metaborg-go

2017: NaBL2 Case Study

- Static semantics in NaBL2
- Challenge: structural types

LEP 2017-2018

- Solid and complete syntax definition
- Static semantics in Statix
- Compilation to …?

Solidity

Solidity = Programming Smart Contracts

- **Solidity** is a contract-oriented programming language for writing smart contracts.[1] It is used for implementing smart contracts[2] on various blockchain platforms
- It was developed by Gavin Wood, Christian Reitwiessner, Alex Beregszaszi, Liana Husikyan, Yoichi Hirai and several former Ethereum core contributors to enable writing smart contracts on blockchain platforms such as Ethereum.

Compiler Construction 2017-2018

- Syntax definition + NaBL2 type system
- By Jens Voortman

LEP 2017-2018

- Solid, complete syntax definition
- Type system in Statix
- Code generator?
- DynSem interpreter?

Grammar Workbench

Goal: Importing Grammars

- Reuse grammars in other grammar formalisms by translation to SDF3

LEP 2014-2015

- ANTLR in Spoofax
- https://github.com/MetaBorgCube/metaborg-antlr

LEP 2017-2018

- YACC, (E)BNF, …?

Pascal

Pascal

- 1970s educational language
- Developed by Niklaus Wirth

Goal

- Complete demonstration project of Spoofax

NaBL2 case study

- https://github.com/MetaBorgCube/metaborg-pascal

LEP 2016-2017

- Some extensions and improvements

LEP 2017-2018

- Solid and complete syntax definition
- Complete static semantics
- Compilation or interpretation

SASS

SASS

- https://sass-lang.com/

LEP 2017-2018

- Syntax definition
- Type checking?
- CSS generation

Automata Lab

Formal languages and automata

- Regular expression language
- Context-free grammar language
- Automata languages
  - Turing machines, DFA, NFA
  - Parse tables
- Algorithms
  - Conversion: Parser generation, normalization, determination, …
  - Execution: parsing, recognition, …

Goal: Supporting Education

- Formal languages and automata
- Compiler construction

What are you going to do?