Dependently Typed Languages in Statix

Jonathan Brouwer

Delft University of Technology, The Netherlands

April 26, 2023

Table of Contents

Background: What is Spoofax?

A language designer's workbench with everything you need to design a programming language.

- Declarative, using meta-languages

Background: What is Spoofax?

Numbers.sdf3

```
17 context-free syntax
18
19   Exp.Int      = IntConst
20
21   Exp.Uminus   = [- [Exp]]
22   Exp.Times    = [[Exp] * [Exp]]    {left}
23   Exp.Divide   = [[Exp] / [Exp]]    {left}
24   Exp.Plus     = [[Exp] + [Exp]]    {left}
25   Exp.Minus    = [[Exp] - [Exp]]    {left}
26
27   Exp.Eq       = [[Exp] = [Exp]]    {non-assoc}
28   Exp.Neq      = [[Exp] <> [Exp]]   {non-assoc}
29   Exp.Gt       = [[Exp] > [Exp]]    {non-assoc}
30   Exp.Lt       = [[Exp] < [Exp]]    {non-assoc}
31   Exp.Geq      = [[Exp] ⩾ [Exp]]    {non-assoc}
32   Exp.Leq      = [[Exp] ⩽ [Exp]]    {non-assoc}
33
34   Exp.And      = [[Exp] & [Exp]]    {left}
35   Exp.Or       = [[Exp] | [Exp]]    {left}
```

Background: What is Spoofax?

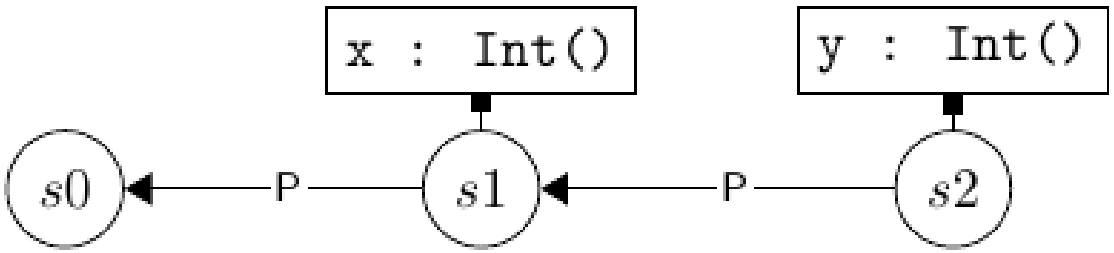
static-semantics.stx

```
356   typeOfExp(s, Int(i)) = INT() :-
357     @i.lit := i.
358
359 rules // operators
360
361   typeOfExp(s, Uminus(e)) = INT() :-
362     typeOfExp(s, e) == INT().
363
364   typeOfExp(s, Divide(e1, e2)) = INT() :-
365     typeOfExp(s, e1) == INT(),
366     typeOfExp(s, e2) == INT().
367
368   typeOfExp(s, Times(e1, e2)) = INT() :-
369     typeOfExp(s, e1) == INT(),
370     typeOfExp(s, e2) == INT().
371
372   typeOfExp(s, Minus(e1, e2)) = INT() :-
373     typeOfExp(s, e1) == INT(),
374     typeOfExp(s, e2) == INT().
```

Background: What is Spoofax?

Example

```
let x = 5;
let y = x + 3;
return x + y;
```



Background: What is Spoofax?

to-ir.str

```
16
17  to-ir-all = innermost(
18    to-ir +
19    to-ir-flatmap
20  )
21
22  // lhs |▷ rhs → lhs ; flatMap(lhs)
23  to-ir-flatmap: FlatMap(lhs, rhs) → Seq(lhs, Apply
24  // flatMap(lhs, flatMap(rhs)) → flatMap(lhs) ; fl
25  to-ir-flatmap: Apply(Var("flatMap"), [Seq(lhs, App
26    Seq(Apply(Var("flatMap"), [lhs]), Apply(Var("fla
27  // flatMap(lhs; rhs@(flatMap(_) ; _)) → flatMap(l
28  to-ir-flatmap: Apply(Var("flatMap"), [Seq(lhs, rhs
29    Seq(Apply(Var("flatMap"), [lhs]), rhs)
30
31  // Makes a strategy with an implicit input argumen
32  to-ir: StrategyDef(name, params, body) → Strategy
33  with inputVar := "__input"    // TODO: Generate un
34
```

Background: What are Dependent Types?

- Types may depend on values!

Example

```
concat : (A: Type) -> (n m : Nat) -> Vec A n -> Vec A m
         -> Vec A (n + m)
```

Background: What are Dependent Types?

Why are dependent types useful?

- Proof Assistants: Agda, Lean, Coq, etc

Sorted lists

```
sort : List t -> List t
sort_sorted : (v : List t) -> IsSorted (sort v)
```

Background: What are Dependent Types?

Type checking requires evaluation

Example 1

```
let T: Type = if false then Int else Bool end;
let b:  T = true;
```

Research Question

How suitable is Statix for defining a dependently-typed language?

- Will it be easier than doing it in Haskell?
- Prior work: Defining System F[1]

[1] Hendrik van Antwerpen et al. Scopes as types.

Why is this important?

Spoofax perspective

Developing a language with a complex type system tests the boundaries of what Spoofax can do.

Dependent Types perspective

Using a language workbench helps with rapid prototyping.

Primary Contribution: Calculus of Constructions in Statix

A lambda calculus with dependent types.

Syntax Definition

```
Expr.Type = "Type"
Expr.Var = ID
Expr.FnType = ID ":" Expr "->" Expr {right}
Expr.FnConstruct = "\\" ID ":" Expr "." Expr
Expr.FnDestruct = Expr Expr {left}
Expr.Let = "let" ID "=" Expr ";" Expr
```

Example

```
let f = \T: Type. \x: T. x;
f (Type -> Type) (\y: Type. y)
```

Type Checking: From inference rules to Statix code

$$\frac{\langle s \mid e \rangle : t_e \qquad \langle \mathsf{sPutSubst}(s, x, (s, e)) \mid b \rangle : t_b}{\langle s \mid \mathsf{Let}(x, e, b) \rangle : t_b}$$

Signature of *typeOfExpr*

```
typeOfExpr : scope * Expr -> Expr
```

Equivalent Statix code

```
typeOfExpr (s, Let(x, e, b)) = bt :-
  typeOfExpr (s, e) == et,
  typeOfExpr (sPutSubst (s, x, (s, e)), b) == bt
```

Type Checking: Environments & Context

Example

```
let T: Type = Bool;
\b: T. ???
```

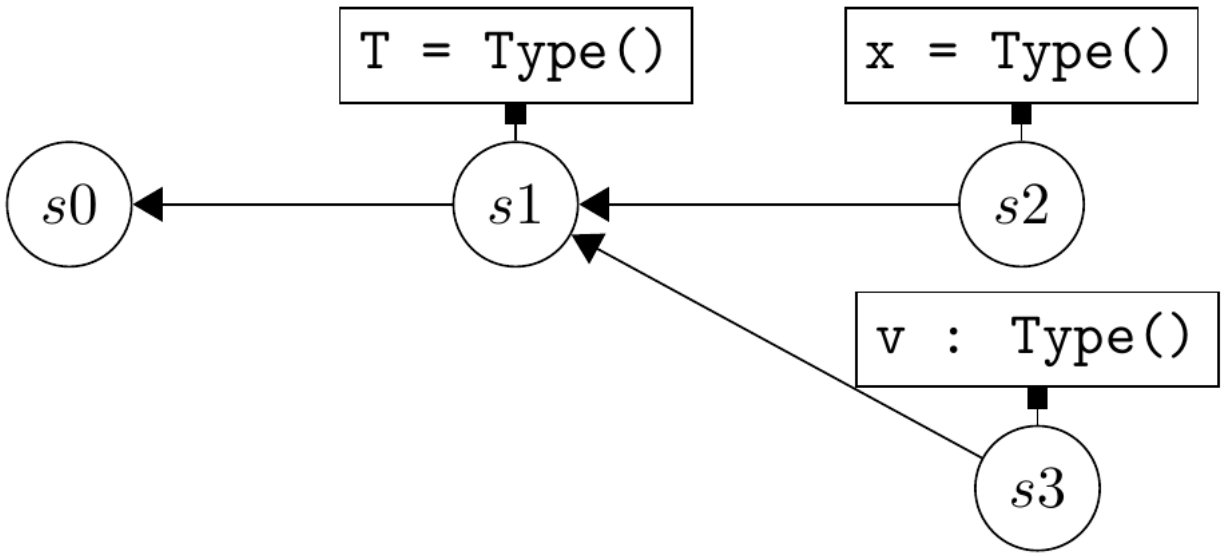
What information is needed?

1. T = Bool (Stored in Environment)
2. b : T (Stored in Context)

Type Checking: Environments & Context

How do we use scopes?

Scope graphs are used as a replacement for an environment and a context.



Type Checking: Requires Evaluation

Example from earlier

```
let T = if false then Int else Bool end;
let b: T = true;
```

Evaluation relations

```
betaHeadReduce : scope * Expr -> scope * Expr
betaReduce : scope * Expr -> Expr
expectBetaEq : (scope * Expr) * (scope * Expr)
```

Type Checking: Variable Capturing

What is the type of this expression?

```
\T : Type. \T : T. T
```

Type Checking: Variable Capturing

What is the type of this expression?

```
\T : Type. \T : T. T
```

The type

```
T : Type -> T : T -> T
```

Equivalent type under renaming: Not correct!

```
T : Type -> x : T -> x
```

Type Checking: Variable Capturing

Solutions

1. De Bruijn indices
2. Uniquifying names
3. Capture-avoiding substitution
4. **Using scopes to distinguish names**

The type

```
T : Type -> T : T -> T
```

The type (fixed)

```
T : Type -> T : (T, s0) -> (T, s0)
```

Extra contributions

Features

1. **Implemented Inference**
2. Implemented Inductive Data Types
3. Implemented Universes
4. Interpreter
5. Compiler to Clojure

Evaluation

1. **Comparison with implementation in Haskell**
2. Comparison with implementation in LambdaPi
3. Evaluation of Spoofax

On Extensibility

How to extend the language

1. Define parsing rules
2. Create a new .stx file
3. Define the relations: `typeOfExpr`, `betaReduceHead`, `expectBetaEq` and `betaReduce`

Comparison with Haskell

Type checking

```
data EnvEntry = NType Expr | NSubst (Env, Expr)
type Env = [EnvEntry]
tc :: Env -> Expr -> Either String Expr
```

Differences

- Distribution of definitions over files
- De Bruijn Indices vs Names
- Definition in Statix is approximately 20% longer
- Inference built-in to Statix

Term Inference

Example

```
let id = (\T : Type. \x: T. x);
id _ true
```

How it works

- When `expectBetaEq(e1, e2)` and `e1` or `e2` is free, infer!
- But in Statix we can't query whether variables are free.
- Solution: Wrap each free constructor in `Infer`.

Term Inference: Equational Unification

- We can declare two terms to be equal if they satisfy a certain relation (such as beta equality)
- We do this using *reduction rules*, such as:

Example of a rewrite rule

```
(\ x : T. b) a => b[x := a]
```

- Comparison with LambdaPi

Conclusions

- Statix is already as a great tool for prototyping dependently typed languages!
- Support for complex inference could be improved, for example using equational unification.