Dependently Typed Languages in Statix

Jonathan Brouwer

Delft University of Technology, The Netherlands

April 24, 2023

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

```
context-free syntax

  Exp.Int       = IntConst

  Exp.Uminus    = [- [Exp]]
  Exp.Times     = [[Exp] * [Exp]]    {left}
  Exp.Divide    = [[Exp] / [Exp]]    {left}
  Exp.Plus      = [[Exp] + [Exp]]    {left}
  Exp.Minus     = [[Exp] - [Exp]]    {left}

  Exp.Eq        = [[Exp] = [Exp]]    {non-assoc}
  Exp.Neq       = [[Exp] <> [Exp]]   {non-assoc}
  Exp.Gt        = [[Exp] > [Exp]]    {non-assoc}
  Exp.Lt        = [[Exp] < [Exp]]    {non-assoc}
  Exp.Geq       = [[Exp] >= [Exp]]   {non-assoc}
  Exp.Leq       = [[Exp] <= [Exp]]   {non-assoc}

  Exp.And       = [[Exp] & [Exp]]    {left}
  Exp.Or        = [[Exp] | [Exp]]    {left}
```

Background: What is Spoofax?

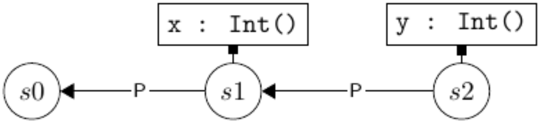
static-semantics.stx

```
  typeOfExp(s, Int(i)) = INT() :-
    @i.lit := i.

rules // operators

  typeOfExp(s, Uminus(e)) = INT() :-
    typeOfExp(s, e) == INT().

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

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

  typeOfExp(s, Minus(e1, e2)) = INT() :-
    typeOfExp(s, e1) == INT(),
    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

```
to-ir-all = innermost(
  to-ir +
  to-ir-flatmap
)

// lhs |▷ rhs → lhs ; flatMap(lhs)
to-ir-flatmap: FlatMap(lhs, rhs) → Seq(lhs, Apply
// flatMap(lhs, flatMap(rhs)) → flatMap(lhs) ; fl
to-ir-flatmap: Apply(Var("flatMap"), [Seq(lhs, App
  Seq(Apply(Var("flatMap"), [lhs]), Apply(Var("fla
// flatMap(lhs; rhs@(flatMap(_) ; _)) → flatMap(l
to-ir-flatmap: Apply(Var("flatMap"), [Seq(lhs, rhs
  Seq(Apply(Var("flatMap"), [lhs]), rhs)

// Makes a strategy with an implicit input argumen
to-ir: StrategyDef(name, params, body) → Strategy
with inputVar := "__input"    // TODO: Generate un
```

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

Type checking rules $\boxed{\langle s \mid e \rangle : t}$

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

$$\frac{\mathsf{sGetName}(s, x) = \mathsf{NType}(t)}{\langle s \mid \mathsf{Var}(x) \rangle : t} \qquad \frac{\mathsf{sGetName}(s, x) = \mathsf{NSubst}(s_e, e) \qquad \langle s_e \mid e \rangle : t}{\langle s \mid \mathsf{Var}(x) \rangle : t}$$

$$\frac{\begin{array}{c}\langle s \mid a \rangle : t_a \qquad t_a \underset{\beta}{=} \mathsf{Type}() \qquad \langle s \mid a \rangle \underset{\beta}{\Rightarrow} a' \\ \langle \mathsf{sPutType}(s, x, a') \mid b \rangle : t_b \qquad t_b \underset{\beta}{=} \mathsf{Type}()\end{array}}{\langle s \mid \mathsf{FnType}(x, a, b) \rangle : \mathsf{Type}()} \qquad \frac{\begin{array}{c}\langle s \mid a \rangle : t_a \qquad t_a \underset{\beta}{=} \mathsf{Type}() \qquad \langle s \mid a \rangle \underset{\beta}{\Rightarrow} a' \\ \langle \mathsf{sPutType}(s, x, a') \mid b \rangle : t_b\end{array}}{\langle s \mid \mathsf{FnConstruct}(x, a, b) \rangle : \mathsf{FnType}(x, a', t_b)}$$

$$\frac{\begin{array}{c}\langle s \mid f \rangle : t_f \qquad \langle s \mid t_f \rangle \; [] \underset{\beta h}{\Rightarrow} \langle s_f \mid \mathsf{FnType}(x, t_{da}, t_b) \rangle \\ \langle s \mid a \rangle : t_a \qquad t_a \underset{\beta}{=} \langle s_f \mid t_{da} \rangle \qquad \langle \mathsf{sPutSubst}(s_f, x, (s, a)) \mid t_b \rangle \underset{\beta}{\Rightarrow} t_b'\end{array}}{\langle s \mid \mathsf{FnDestruct}(f, a) \rangle : t_b'}$$

Figure 4.4: Rules for type checking the Calculus of Constructions

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. ???
```

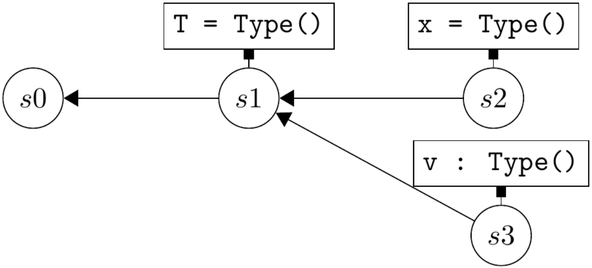
What information is needed?

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 relation

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

Type Checking: Requires Evaluation

Beta head-reduction rules

$$\boxed{\langle s_1 \mid e_1 \rangle\ \overline{p} \underset{\beta h}{\Rightarrow} \langle s_2 \mid e_2 \rangle}$$

$$\frac{}{\langle s \mid \mathsf{Type}() \rangle\ [\,] \underset{\beta h}{\Rightarrow} \langle s \mid \mathsf{Type}() \rangle}$$

$$\frac{\langle \mathsf{sPutSubst}(s, x, (s, e)) \mid b \rangle\ \overline{p} \underset{\beta h}{\Rightarrow} \langle s' \mid b' \rangle}{\langle s \mid \mathsf{Let}(x, e, b) \rangle\ \overline{p} \underset{\beta h}{\Rightarrow} \langle s' \mid b' \rangle}$$

$$\frac{\mathsf{sGetName}(s, x) = \mathsf{NSubst}(s_e, e) \qquad \langle s_e \mid e \rangle\ \overline{p} \underset{\beta h}{\Rightarrow} \langle s_{e'} \mid e' \rangle}{\langle s \mid \mathsf{Var}(x) \rangle\ \overline{p} \underset{\beta h}{\Rightarrow} \langle s_{e'} \mid e' \rangle}$$

$$\frac{\mathsf{sGetName}(s, x) = \mathsf{NType}(t)}{\langle s \mid \mathsf{Var}(x) \rangle\ \overline{p} \underset{\beta h}{\Rightarrow} \mathsf{rebuild}(s, \mathsf{Var}(x), \overline{p})}$$

$$\frac{}{\langle s \mid \mathsf{FnType}(x, a, b) \rangle\ [\,] \underset{\beta h}{\Rightarrow} \langle s \mid \mathsf{FnType}(x, a, b) \rangle}$$

$$\frac{}{\langle s \mid \mathsf{FnConstruct}(x, a, b) \rangle\ [\,] \underset{\beta h}{\Rightarrow} \langle s \mid \mathsf{FnConstruct}(x, a, b) \rangle}$$

$$\frac{\langle \mathsf{sPutSubst}(s, x, p) \mid b \rangle\ \overline{p} \underset{\beta h}{\Rightarrow} \langle s' \mid e' \rangle}{\langle s \mid \mathsf{FnConstruct}(x, \_, b) \rangle\ (p :: \overline{p}) \underset{\beta h}{\Rightarrow} \langle s' \mid e' \rangle}$$

$$\frac{\langle s \mid f \rangle\ (a :: \overline{p}) \underset{\beta h}{\Rightarrow} \langle s' \mid e' \rangle}{\langle s \mid \mathsf{FnDestruct}(f, a) \rangle\ \overline{p} \underset{\beta h}{\Rightarrow} \langle s' \mid e' \rangle}$$

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
- 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

Spoofax is a great tool for developing dependently typed languages!

- We can use scopes to represent environments and contexts
- Statix can still use improvements