

# Dependently Typed Languages in Statix

Jonathan Brouwer   Jesper Cockx   Aron Zwaan

Delft University of Technology, The Netherlands

April 20, 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

```
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().  
375
```

# Background: What is Spoofax?

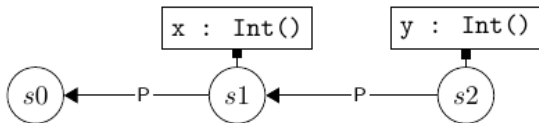
to-ir.str ✖

```
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 is Spoofax?

## Example

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



# Background: What are Dependent Types?

- Types may depend on values!

## Example

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

# 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;
```



# 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)
```

## Research Question

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

- Will it be easier than doing it in Haskell?
- Defining system F was a challenge<sup>1</sup>, will it even be possible?

---

<sup>1</sup>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 (T: Type -> Type) (\y: Type. y)
```

# Type Checking

## Type checking rules

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

$$\begin{array}{c}
 \frac{}{\langle s \mid \text{Type}() \rangle : \text{Type}()} \qquad \frac{\langle s \mid e \rangle : t_e \quad \langle \text{sPutSubst}(s, x, (s, e)) \mid b \rangle : t_b}{\langle s \mid \text{Let}(x, e, b) \rangle : t_b} \\
 \\
 \frac{\text{sGetName}(s, x) = \text{NType}(t)}{\langle s \mid \text{Var}(x) \rangle : t} \qquad \frac{\text{sGetName}(s, x) = \text{NSubst}(s_e, e) \quad \langle s_e \mid e \rangle : t}{\langle s \mid \text{Var}(x) \rangle : t} \\
 \\
 \frac{\langle s \mid a \rangle : t_a \quad t_a \stackrel{\beta}{=} \text{Type()} \quad \langle s \mid a \rangle \Rightarrow_{\beta} a' \quad \langle \text{sPutType}(s, x, a') \mid b \rangle : t_b \quad t_b \stackrel{\beta}{=} \text{Type}()}{\langle s \mid \text{FnType}(x, a, b) \rangle : \text{Type}()} \qquad \frac{\langle s \mid a \rangle : t_a \quad t_a \stackrel{\beta}{=} \text{Type()} \quad \langle s \mid a \rangle \Rightarrow_{\beta} a' \quad \langle \text{sPutType}(s, x, a') \mid b \rangle : t_b}{\langle s \mid \text{FnConstruct}(x, a, b) \rangle : \text{FnType}(x, a', t_b)} \\
 \\
 \frac{\langle s \mid f \rangle : t_f \quad \langle s \mid t_f \rangle \Rightarrow_{\beta h} \langle s_f \mid \text{FnType}(x, t_{da}, t_b) \rangle \quad \langle s \mid a \rangle : t_a \quad t_a \stackrel{\beta}{=} \langle s_f \mid t_{da} \rangle \quad \langle \text{sPutSubst}(s_f, x, (s, a)) \mid t_b \rangle \Rightarrow_{\beta} t'_b}{\langle s \mid \text{FnDestruct}(f, a) \rangle : t'_b}
 \end{array}$$

Figure 4.4: Rules for type checking the Calculus of Constructions

## Type Checking: From inference rules to Statix code

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

### 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  $T = \text{Bool}$  (Stored in Environment)
- ②  $x : T$  (Stored in Context)

# Type Checking: Environments & Context

## How do we use scopes?

A scope is used as a replacement for an environment and a context.

- One type of edge: `p`.
- One type of relation: `NameEntry`, which can be:
  - `NType`: Stores a type (Corresponds to a context)
  - `NSubst`: Stores a substitution (Corresponds to an environment)



# 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  
exactBetaEq : (scope * Expr) * (scope * Expr)
```

# Type Checking: Requires Evaluation

## Beta head-reduction rules

$$\langle s_1 \mid e_1 \rangle \bar{p} \Rightarrow_{\beta h} \langle s_2 \mid e_2 \rangle$$

$$\frac{}{\langle s \mid \text{Type}() \rangle \bar{\square} \Rightarrow_{\beta h} \langle s \mid \text{Type}() \rangle} \quad \frac{\langle \text{sPutSubst}(s, x, (s, e)) \mid b \rangle \bar{p} \Rightarrow_{\beta h} \langle s' \mid b' \rangle}{\langle s \mid \text{Let}(x, e, b) \rangle \bar{p} \Rightarrow_{\beta h} \langle s' \mid b' \rangle}$$

$$\frac{\text{sGetName}(s, x) = \text{NSubst}(s_e, e) \quad \langle s_e \mid e \rangle \bar{p} \Rightarrow_{\beta h} \langle s_e' \mid e' \rangle}{\langle s \mid \text{Var}(x) \rangle \bar{p} \Rightarrow_{\beta h} \langle s_e' \mid e' \rangle}$$

$$\frac{\text{sGetName}(s, x) = \text{NType}(t)}{\langle s \mid \text{Var}(x) \rangle \bar{p} \Rightarrow_{\beta h} \text{rebuild}(s, \text{Var}(x), \bar{p})} \quad \frac{}{\langle s \mid \text{FnType}(x, a, b) \rangle \bar{\square} \Rightarrow_{\beta h} \langle s \mid \text{FnType}(x, a, b) \rangle}$$

$$\frac{}{\langle s \mid \text{FnConstruct}(x, a, b) \rangle \bar{\square} \Rightarrow_{\beta h} \langle s \mid \text{FnConstruct}(x, a, b) \rangle}$$

$$\frac{\langle \text{sPutSubst}(s, x, p) \mid b \rangle \bar{p} \Rightarrow_{\beta h} \langle s' \mid e' \rangle}{\langle s \mid \text{FnConstruct}(x, \_, b) \rangle (p :: \bar{p}) \Rightarrow_{\beta h} \langle s' \mid e' \rangle} \quad \frac{\langle s \mid f \rangle (a :: \bar{p}) \Rightarrow_{\beta h} \langle s' \mid e' \rangle}{\langle s \mid \text{FnDestruct}(f, a) \rangle \bar{p} \Rightarrow_{\beta h} \langle s' \mid e' \rangle}$$

# 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

# 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