Dependently Typed Languages in Statix

Jonathan Brouwer

Delft University of Technology, The Netherlands

April 22, 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().
```

Background: What is Spoofax?

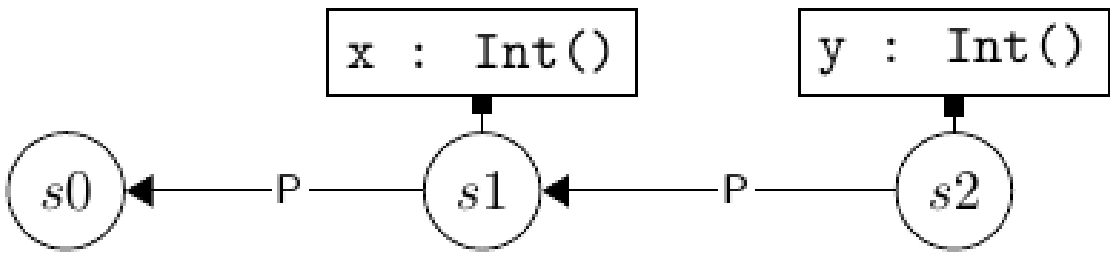
to-ir.str

```
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[1], will it even be possible?

[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 (T: Type -> Type) (\y: Type. y)
```

Type Checking

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

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

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

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

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

$$\frac{\langle s \mid f \rangle : t_f \qquad \langle s \mid t_f \rangle \; [] \underset{\beta h}{\Rightarrow} \langle s_f \mid \mathsf{FnType}(x, t_{da}, t_b) \rangle \qquad \langle s \mid a \rangle : t_a \qquad t_a \underset{\beta}{=} \langle s_f \mid t_{da} \rangle \qquad \langle \mathsf{sPutSubst}(s_f, x, (s, a)) \mid t_b \rangle \underset{\beta}{\Rightarrow} t'_b}{\langle s \mid \mathsf{FnDestruct}(f, a) \rangle : t'_b}$$

Figure 4.4: Rules for type checking the Calculus of Constructions

Type Checking: From inference rules to Statix code

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

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. 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
exectBetaEq : (scope * Expr) * (scope * Expr)
```

Type Checking: Requires Evaluation

Beta head-reduction rules

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

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

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

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

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

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

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**

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

Comparison with Haskell

Type checking

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

Differences

- Distribution of definitions over files
- De Bruijn Indices vs Names
- Inference built-in to Statix

On Extensibility

How to extend the language

1. Define the parsing rules for the new feature
2. Create a new .stx file for the type-checking rules (and import it)
3. Define the relations (typeOfExpr, betaReduceHead, expectBetaEq and betaReduce) in this file using the defining inference rules

Term Inference

Example

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

How it works

- When we encounter `expectBetaEq(e1, e2)` and `e1` or `e2` is free, we can infer it!
- 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

- A *logical framework* in which you can embed other programming languages
- Supports rewrite rules
- Cannot define *Let* because of limitations of LambdaPi

Comparison with LambdaPi: Example

Defining FnConstruct

```
symbol FnConstruct :
  Π (A : TmSort ),
  Π (B : TmType A → TmSort ),
  Π (f : Π (x : TmType A), TmType (B x)),
  TmType ( FnType A B);
```

Reduction rule for FnConstruct

```
rule FnConstruct _ _ $f → $f;
```

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