Dependently Typed Languages in Statix

Jonathan Brouwer   Jesper Cockx   Aron Zwaan

Delft University of Technology, The Netherlands

April 19, 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?

SDF3, Statix and Stratego (and more)

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().
```

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}
```

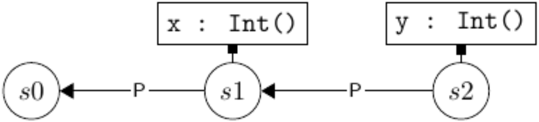
to-ir.str

```

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

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

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

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

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

Figure 4.4: Rules for type checking the Calculus of Constructions

Type Checking: From inference rules to Statix code

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

Equivalent Statix code

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

Type Checking

How do we use scopes?

A scope is used as a replacement for an environment and a context.
One edge `p`. One relation `name` → `NameEntry`, NameEntry is either:

- NType: Stores a type (Corresponds to a context)
- NSubst: Stores a substitution (Corresponds to an environment)

Type Checking: Requires Evaluation

Example 1

```
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)
```

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