

Generic Description of Well-Scoped, Well-Typed Syntaxes

Gergő Érdi

<http://gergo.erd.hu/>

Midlands Graduate School
April 2018.

Motivation

Handling languages in a dependently typed setting:

`data Form : Set`

`data Expr : Ctx0 → Set`

`data Tm : Ctx → Ty → Set`

Renaming & substitution

map : $\forall \{ \Gamma \ \Delta \}$
→ (Var $\Gamma \Rightarrow$ Var Δ)
→ (Tm $\Gamma \Rightarrow$ Tm Δ)

bind : $\forall \{ \Gamma \ \Delta \}$
→ (Var $\Gamma \Rightarrow$ Tm Δ)
→ (Tm $\Gamma \Rightarrow$ Tm Δ)

Renaming & substitution

$\text{map} : \forall \{ \Gamma \ \Delta \}$
 $\rightarrow (\text{Var } \Gamma \Rightarrow \text{Var } \Delta)$
 $\rightarrow (\text{Tm } \Gamma \Rightarrow \text{Tm } \Delta)$

$\text{bind} : \forall \{ \Gamma \ \Delta \}$
 $\rightarrow (\text{Var } \Gamma \Rightarrow \text{Tm } \Delta)$
 $\rightarrow (\text{Tm } \Gamma \Rightarrow \text{Tm } \Delta)$

+ a whole lot of proofs

Librarization

Definition of `map`/`bind`, and all proofs about them, depend on constructors of `Tm`

⇒ try adding e.g. `Bools` or `let` to the object language. . .

Syntax-generic programming

```
data Desc : Set1 where
  sg : (A : Set) → (A → Desc) → Desc
  node : (n : ℕ) {k : ℕ} → Shape n k
        → (wt : Vec Ty n → Vec Ty k → Ty → Set)
        → Desc
```

No constructor for **var**!

Syntax-generic programming

```
data Desc : Set1 where
  sg : (A : Set) → (A → Desc) → Desc
  node : (n : ℕ) {k : ℕ} → Shape n k
        → (wt : Vec Ty n → Vec Ty k → Ty → Set)
        → Desc
```

Storing
data in
nodes

No constructor for `var!`

New bound
variables

Well-typedness
predicate

Subterm
structure

Typing constraints

- ▶ Arbitrary proposition
- ▶ Don't need to be decidable
- ▶ Has *global view* \Rightarrow untyped representations are possible

Library provides:

- ▶ `ren` (OPE) and `sub` (list of terms)
- ▶ `ren`, `sub` proofs
- ▶ Unscoped, untyped, typed, PHOAS representations
- ▶ Type erasure, scope checking, ...

Demo

```
data `STLC : Set where
```

```
  `lam `app : `STLC
```

```
STLC : Desc
```

```
STLC = sg `STLC λ
```

```
  { `lam → sg Ty λ u →
```

```
    node 1 ([ [ bound ] ]) λ { [ u' ] [ t ] t' →  
      u' ≡ u ∧ t' ≡ u ▷ t }
```

```
; `app →
```

```
  node 0 [ [], [] ] λ { [] [ u , t ] t' →  
    u ≡ t ▷ t' }
```

```
}
```

Church style



- ▶ STLC normalization proof syntactically, from Software Foundations
- ▶ **let** \rightsquigarrow **lam** translation

TODO

Multiple contexts?

- ▶ Orthogonal (e.g. modal logic)
- ▶ Multi-sorted (e.g. System F)
- ▶ Incestuous (TT...) ?

TODO

Non-equality *wt* constraint use cases?

- ▶ Subtyping?
- ▶ Compositional typing unification?

Bidirectional typing?

Induction schema for a given **Desc**?

For more info

- ▶ Paper on Arxiv:
<https://arxiv.org/abs/1804.00119>
- ▶ (Submitted to LFMTP 2018)
- ▶ Questions?