Formalise Your Type System, Intrinsically

Problem Session (2023 April 27)

Xu Xue (HKU Programming Languages Group)

What is?

- PLFA Chapter 2.3 (Debruijn)
  - There are two fundamental approaches (extrinsic and intrinsic) to typed lambda calculi.
  - One approach, is to first define terms and then define types. Terms exist independent of types, and may have types assigned to them by separate typing rules.
  - Another approach, is to first define types and then define terms. Terms and type rules are intertwined, and it makes no sense to talk of a term without a type.
- TAPL Chapter 9.6 (Curry-Style vs. Church-Style)
  - Semantics is prior to typing: we first define the terms, then define a semantics showing how they behave, then give a type system that rejects some terms whose behaviors we don't like.
  - Typing is prior to semantics: to define terms and identify the well-typed terms, then give semantics just to these.

Problem Session

- PLFA (Wadler) promotes Intrinsic Typing by
  - Saying "Intrinsic typing is golden" (extrinsically-typed terms require about 1.6 times as much code as intrinsically-typed)
  - Giving a detailed type-sound formalisation of PCF (STLC + Nat + Fix)
  - Mentioning more language constructs (products, sum, let-binding…)
  - Presenting a non-trivial calculus formalisation (System Fw + iso-recursive types)

updates

System F in Agda, for Fun and Profit

James Chapman[1(✉)], Roman Kireev[1], Chad Nester[2], and Philip Wadler[2]

[1] IOHK, Hong Kong, Hong Kong
{james.chapman,roman.kireev}@iohk.io
[2] University of Edinburgh, Edinburgh, UK
{cnester,wadler}@inf.ed.ac.uk

Abstract. System F, also known as the polymorphic λ-calculus, is a typed λ-calculus independently discovered by the logician Jean-Yves Girard and the computer scientist John Reynolds. We consider $F_{\omega\mu}$, which adds higher-order kinds and iso-recursive types. We present the first complete, intrinsically typed, executable, formalisation of System $F_{\omega\mu}$ that we are aware of. The work is motivated by verifying the core language of a smart contract system based on System $F_{\omega\mu}$. The paper is a literate Agda script [14].

Problem Session

- Does this technique really fit well with more non-trivial features?
  - Subtyping (e.g., Intersection types)
  - Type inference (e.g., Bidirectional Typing)
  - Semantics (e.g., Type-directed Operational Semantics)

Let's see!

Table of Contents

- Agda tutorials (<5 mins) (Sorry 🥲)
  - Get comfortable with proof by constructions
- Review extrinsic formalisation of STLC
- Translate to intrinsic formalisation (1-by-1)
  - add more language structs
  - showcase type safety statement
- Discuss about the potential of applying intrinsic typing to λ_i

Agda Modules

Nat

_+_ is binary operator (func.)

```
module _ where

open import Data.Nat using (ℕ; zero; suc; _+_; _*_; _^_; _∸_)
open import Data.String using (String)
open import Relation.Binary.PropositionalEquality using (_≡_; _≢_; refl)
```

- Agda allows unicode to name anything.
  - Be not surprised when you see a variable named A<B (without any spaces)

Agda Datatypes

```
data Type : Set where
  Int : Type
  Arr : Type → Type → Type

data Term : Set where
  lit : ℕ → Term
  var : String → Term
  lam : String → Term → Term
  app : Term → Term → Term
```

Agda Snippets (Extrinsic Typing)

Prop

```
data ty : Context → Term → Type → Set where

  ty-lit : ∀ {Γ n}
    → ty Γ (lit n) Int

  ty-var : ∀ {Γ x A}
    → Γ ∋ x ⦂ A
    → ty Γ (var x) A

  ty-lam : ∀ {Γ x A B e}
    → ty (Γ , x ⦂ A) e B
    → ty Γ (lam x e) (Arr A B)

  ty-app : ∀ {Γ e₁ e₂ A B}
    → ty Γ e₁ (Arr A B)
    → ty Γ e₂ A
    → ty Γ (app e₁ e₂) B
```

implicit arguments (should be mentioned in explicit arguments)

explicit arguments

Proof by Construction

When finding a proof of a proposition, you are finding a inhabitant of a type.

- Term
  - (lit 2)
  - (app (lit 2) (lit 3))
  - …
- ty Γ (lit 4) Int
  - ty-lit

It's just like the "**apply ty-lit**" in Coq

Demo Time

Observations

Proof of Term	`app`	`(lam "x" (var "x"))`	`(lit 2)`
Proof of Typing	`ty-app`	`(ty-lam (ty-var Z))`	`ty-lit`

Almost Same!

- app corresponds to ty-app
- lam corresponds to ty-lam
- lit corresponds to ty-lit

Why not mix them together?

Intrinsic Typing (Principles)

- To define a type
- Then define a term which is dependent on those types
  - $\Gamma \vdash A$ (vs. Term)
  - it reads as "term of type A (under context Γ)"

Comparison

Extrinsic	Intrinsic
`(lit 4) : Term`	`(⊢lit 4) : Γ ⊢ Int`
`(lam "x" (var "x")) : Term`	`(⊢lam "x" (⊢var "x")) : Γ ⊢ Int ⇒ Int`

`e1 : Term`
`e2 : Term`
`(app e1 e2) : Term`
	`e1 : Γ ⊢ Int ⇒ Int`
	`e2 : Γ ⊢ Int`
	`(⊢app e1 e2) : Γ ⊢ Int`

The intuition of such formalisation is:
to construct any terms, we should specify its type first

To construct a well-typed variable

$$\frac{x : A \in \Gamma}{\Gamma \vdash x : A}$$

```
⊢var : ∀ {Γ A}
  → (x : Id)
  → Γ ∋ x ⦂ A
  → Γ ⊢ A
```

- Given a Γ and A
- Given a explicit variable name “x"
- Given a proof of “x” is in this Γ
- Then construct a well-typed variable of type A

To construct a well-typed lambda

$$\frac{\Gamma, x : A \vdash e : B}{\Gamma \vdash \lambda x \,.\, e : A \to B}$$

```
⊢lam : ∀ {Γ x A B}
  → Γ , x ⦂ A ⊢ B
  → Γ ⊢ A ⇒ B
```

- Given a Γ, A, B and a bound variable “x”
- Given a proof of well-typed body of type B in a extended Γ
- Then construct a well-typed lambda of type A to B

To construct a well-typed application

$$\frac{\Gamma \vdash e_1 : A \to B \quad \Gamma \vdash e_2 : A}{\Gamma \vdash e_1\ e_2 : B}$$

```
⊢app : ∀ {Γ A B}
  → Γ ⊢ A ⇒ B
  → Γ ⊢ A
  → Γ ⊢ B
```

- Given a Γ, A and B
- Given a proof of well-typed e1 of type A to B
- Given a proof of well-typed e2 of type A
- Then construct a well-typed application of type B

STLC (nominal)

```
data ty : Context → Term → Type → Set where

  ty-var : ∀ {Γ x A}
    → Γ ∋ x ⦂ A
    → ty Γ (var x) A

  ty-lam : ∀ {Γ x A B e}
    → ty (Γ , x ⦂ A) e B
    → ty Γ (lam x e) (Arr A B)

  ty-app : ∀ {Γ e₁ e₂ A B}
    → ty Γ e₁ (Arr A B)
    → ty Γ e₂ A
    → ty Γ (app e₁ e₂) B
```

```
data Type : Set where
  _⇒_ : Type → Type → Type

infix  4 _⊢_
data _⊢_ : Context → Type → Set where

  ⊢var : ∀ {Γ A}
    → (x : Id)
    → Γ ∋ x ⦂ A
    → Γ ⊢ A

  ⊢lam : ∀ {Γ x A B}
    → Γ , x ⦂ A ⊢ B
    → Γ ⊢ A ⇒ B

  ⊢app : ∀ {Γ A B}
    → Γ ⊢ A ⇒ B
    → Γ ⊢ A
    → Γ ⊢ B
```

STLC (nominal) + unit

```
data Type : Set where
  Unit : Type
  _⇒_ : Type → Type → Type
```

```
infix  4 _⊢_
data _⊢_ : Context → Type → Set where

  ⊢unit : ∀ {Γ}
    → Γ ⊢ Unit

  ⊢var : ∀ {Γ A}
    → (x : Id)
    → Γ ∋ x ⦂ A
    → Γ ⊢ A

  ⊢lam : ∀ {Γ x A B}
    → Γ , x ⦂ A ⊢ B
    → Γ ⊢ A ⇒ B

  ⊢app : ∀ {Γ A B}
    → Γ ⊢ A ⇒ B
    → Γ ⊢ A
    → Γ ⊢ B
```

STLC (nominal) + unit

You can construct well-typed terms

```
-- \x. x : Unit -> Unit
_ : ∅ ⊢ Unit ⇒ Unit
_ = ⊢lam (⊢var "x" Z)

-- (\x. x) 1 : Int
_ : ∅ ⊢ Int
_ = ⊢app (⊢lam (⊢var "x" Z)) (⊢int 1)
```

You can't construct ill-typed terms

```
-- (\x. x) 1 : Unit
_ : ∅ ⊢ Unit
_ = ⊢app (⊢lam (⊢var "x" Z)) ⊢unit
```

Well-typed terms reduces to well-typed terms

```
infix 2 _-→_

data _-→_ : ∀ {Γ A} → (Γ ⊢ A) → (Γ ⊢ A) → Set where

  r-app₁ : ∀ {Γ A B} {e₁ e₁' : Γ ⊢ A ⇒ B} {e₂ : Γ ⊢ A}
    → e₁ -→ e₁'
    → (⊢app e₁ e₂) -→ (⊢app e₁' e₂)


  r-app₂ : ∀ {Γ A B} {v : Γ ⊢ A ⇒ B} {e₂ e₂' : Γ ⊢ A}
    → Value v
    → e₂ -→ e₂'
    → (⊢app v e₂) -→ (⊢app v e₂')
```

- You get a preservation theorem for free

- You get a preservation theorem for free?
  - only after you done with the (non-trivial) definition of well-typed reductions
  - e.g., type-preserving substitution 🥲

FUNCTIONAL PEARL

Type-Preserving Renaming and Substitution

CONOR MCBRIDE

University of Nottingham

Abstract

I present a substitution algorithm for the simply-typed λ-calculus, represented in the style of Altenkirch and Reus (1999) which is statically guaranteed to respect scope and type. Moreover, I use a single traversal function, instantiated first to renaming, then to substitution. The program is written in Epigram (McBride & McKinna, 2004).

Case study: λ_i

Typing

$\boxed{\Gamma \vdash e \Leftrightarrow A}$ *(Bidirectional Typing)*

$$\frac{}{\Gamma \vdash i \Rightarrow \mathsf{Int}}\ \textsc{T-Lit} \qquad \frac{x : A \in \Gamma}{\Gamma \vdash x \Rightarrow A}\ \textsc{T-Var} \qquad \frac{\Gamma, x : A \vdash e \Leftarrow B}{\Gamma \vdash \lambda x.\, e : A \to B \Rightarrow A \to B}\ \textsc{T-Lam} \qquad \frac{\Gamma \vdash e \Rightarrow A}{\Gamma \vdash \{l = e\} \Rightarrow \{l : A\}}\ \textsc{T-Rcd}$$

$$\frac{\Gamma \vdash e_1 \Rightarrow A \quad \Gamma \vdash e_2 \Rightarrow B \quad A \ll B = C}{\Gamma \vdash e_1\, e_2 \Rightarrow C}\ \textsc{T-App} \qquad \frac{\Gamma \vdash e \Rightarrow A \quad A \ll l = B}{\Gamma \vdash e.l \Rightarrow B}\ \textsc{T-Proj} \qquad \frac{\Gamma \vdash e_1 \Rightarrow A \quad \Gamma \vdash e_2 \Rightarrow B}{\Gamma \vdash e_1 \,,,\, e_2 \Rightarrow A \,\&\, B}\ \textsc{T-Mrg} \qquad \frac{\Gamma \vdash e \Leftarrow A}{\Gamma \vdash e : A \Rightarrow A}\ \textsc{T-Ann}$$

$$\frac{\Gamma \vdash e \Rightarrow A \qquad A <: B}{\Gamma \vdash e \Leftarrow B}\ \textsc{T-Sub}$$

Fig. 4: Bi-directional typing. The bidirectional mode syntax is $\Leftrightarrow \;::=\; \Leftarrow | \Rightarrow$.

```
data _⊢_ : Context → Type → Set where

  ⊢int : ∀ {Γ}
    → ℕ
    → Γ ⊢ Int

  ⊢var : ∀ {Γ A x}
    → Γ ∋ x ⦂ A
    → Γ ⊢ A

  ⊢lam : ∀ {Γ}
    → (x : Id) → (A B : Type)
    → (Γ , x ⦂ A) ⊢ B
    → Γ ⊢ (A ⇒ B)

  ⊢app : ∀ {Γ A B C}
    → Γ ⊢ A
    → Γ ⊢ B
    → A << B ≡ C
    → Γ ⊢ C

  ⊢sub : ∀ {Γ A B}
    → Γ ⊢ A
    → A ≤ B
    → Γ ⊢ B

  ⊢ann : ∀ {Γ}
    → (A : Type)
    → Γ ⊢ A
    → Γ ⊢ A

  ⊢mrg : ∀ {Γ A B}
    → Γ ⊢ A
    → Γ ⊢ B
    → Γ ⊢ (A & B)
```

$$\frac{\Gamma, x : A \vdash e \Leftarrow B}{\Gamma \vdash \lambda x.\, e : A \to B \Rightarrow A \to B}\ \textsc{T-Lam}$$

```
⊢lam : ∀ {Γ}
  → (x : Id) → (A B : Type)
  → (Γ , x ⦂ A) ⊢ B
  → Γ ⊢ (A ⇒ B)
```

$$\frac{\Gamma \vdash e_1 \Rightarrow A \quad \Gamma \vdash e_2 \Rightarrow B \quad A \ll B = C}{\Gamma \vdash e_1\, e_2 \Rightarrow C}\ \textsc{T-App}$$

```
⊢app : ∀ {Γ A B C}
  → Γ ⊢ A
  → Γ ⊢ B
  → A << B ≡ C
  → Γ ⊢ C
```

$$\frac{\Gamma \vdash e \Leftarrow A}{\Gamma \vdash e : A \Rightarrow A}\ \textsc{T-Ann}$$

```
⊢ann : ∀ {Γ}
  → (A : Type)
  → Γ ⊢ A
  → Γ ⊢ A
```

$$\frac{\Gamma \vdash e \Rightarrow A \qquad A <: B}{\Gamma \vdash e \Leftarrow B}\ \textsc{T-Sub}$$

```
⊢sub : ∀ {Γ A B}
  → Γ ⊢ A
  → A ≤ B
  → Γ ⊢ B
```

Reduction

$\boxed{e \longmapsto e'}$ *(Small-Step Reduction)*

$$\textsc{Step-Int-Ann}\quad \frac{}{i \longmapsto i : \mathsf{Int}}$$

$$\textsc{Step-Arr-Ann}\quad \frac{}{\lambda x.\, e : A \to B \longmapsto (\lambda x.\, e : A \to B) : A \to B}$$

$$\textsc{Step-App}\quad \frac{(v_1 \bullet v_2) \hookrightarrow e}{v_1\, v_2 \longmapsto e}$$

$$\textsc{Step-Pv-Split}\quad \frac{A_1 \lhd A \rhd A_2}{p : A \longmapsto p : A_1 \,,,\, p : A_2}$$

$$\textsc{Step-Prj}\quad \frac{(v \bullet l) \hookrightarrow v'}{v.l \longmapsto v'}$$

$$\textsc{Step-Ann}\quad \frac{\neg e \in p \qquad e \longmapsto e'}{e : A \longmapsto e' : A}$$

$$\textsc{Step-Val-Ann}\quad \frac{v \longmapsto_A v'}{v : A \longmapsto v'}$$

$$\textsc{Step-App-L}\quad \frac{e_1 \longmapsto e_1'}{e_1\, e_2 \longmapsto e_1'\, e_2}$$

$$\textsc{Step-App-R}\quad \frac{e_2 \longmapsto e_2'}{v_1\, e_2 \longmapsto v_1\, e_2'}$$

$$\textsc{Step-Mrg-L}\quad \frac{e_1 \longmapsto e_1'}{e_1 \,,,\, e_2 \longmapsto e_1' \,,,\, e_2}$$

$$\textsc{Step-Mrg-R}\quad \frac{e_2 \longmapsto e_2'}{v_1 \,,,\, e_2 \longmapsto v_1 \,,,\, e_2'}$$

$$\textsc{Step-Rcd-R}\quad \frac{e \longmapsto e'}{\{l = e\} \longmapsto \{l = e'\}}$$

$$\textsc{Step-Prj-L}\quad \frac{e \longmapsto e'}{e.l \longmapsto e'.l}$$

Fig. 6: Operational Semantics

Reduction

```
data _⟶_ : ∀ {Γ A} → (Γ ⊢ A) → (Γ ⊢ A) → Set where
```

Reduction 1: Annotation

```
⊢ann : ∀ {Γ}
  → (A : Type)
  → Γ ⊢ A
  → Γ ⊢ A

⊢int : ∀ {Γ}
  → ℕ
  → Γ ⊢ Int
```

$$\frac{\textsc{Step-Int-Ann}}{i \longmapsto i : \mathsf{Int}}$$

```
step-int-ann : ∀ {Γ n}
  → (⊢int n) ⟶ ⊢ann Int (⊢int n)
```

Reduction 2: Casting

$$\textsc{Step-Val-Ann}\ \frac{v \longmapsto_A v'}{v : A \longmapsto v'}$$

1. Ask: what is the type of v and v'
2. v is of type B and v' should be of type A (after casting)
3. to construct a annotation, v should be of type A
4. need a subsumption rule and proof of B <: A

```
step-val-ann : ∀ {Γ A B} {v : Γ ⊢ B} {v' : Γ ⊢ A}
  → v - A ⇢ v'
  → (B≤A : B ≤ A)
  → (⊢ann A (⊢sub v B≤A)) ⟶ v'
```

We need a new judgment: casting!

Reduction 2: Casting

```
data _-_⇢_ : ∀ {Γ B} → (Γ ⊢ B) → (A : Type) → (Γ ⊢ A) → Set where
```

```
Lemma casting_preservation :
  ∀ v v' A B,
    value v →
    typing nil v Inf B →
    casting v A v' →
    ∃ C, typing nil v' Inf C ∧ isosub C A.
Proof.
```

Reduction 3: Application

$$\textsc{Step-App}\quad \frac{(v_1 \bullet v_2) \ \hookrightarrow \ e}{v_1\, v_2 \ \longmapsto \ e}$$

$$\textsc{T-App}\quad \frac{\Gamma \vdash e_1 \Rightarrow A \qquad \Gamma \vdash e_2 \Rightarrow B \qquad A \ll B = C}{\Gamma \vdash e_1\, e_2 \Rightarrow C}$$

```
data _·_[_]↝_ : ∀ {Γ A B C} → (Γ ⊢ A) → (Γ ⊢ B) → (A << B ≡ C) → (Γ ⊢ C) → Set where

step-app : ∀ {Γ A B C} {v₁ : Γ ⊢ A} {v₂ : Γ ⊢ B} {A<<B : A << B ≡ C} {e : Γ ⊢ C}
  → v₁ · v₂ [ A<<B ]↝ e
  → (⊢app v₁ v₂ A<<B) ⟶ e
```

```
Lemma papp_preservation_v :
  ∀ v vl e A B C,
    value v → value vl →
    typing nil v Inf A →
    typing nil vl Inf B →
    appsub (Some (Avt B)) A C →
    papp v (Av vl) e →
    (∃ D, typing nil e Inf D ∧ isosub D C).
Proof.
```

Problem Session

- Does this technique really fit well with more non-trivial features?
    - Subtyping (e.g., Intersection types)
        - Seems good, but terms are sometimes confused by subsumption rule (e.g., annotation)
    - Type inference (e.g., Bidirectional Typing)
    - Semantics (e.g., Type-directed Operational Semantics)
        - Good as long as the semantics respects a type-preservation principle

Conclusions (& opinions)

- Intrinsic typing is fancy in the perspective of proof engineering
  - especially when it's combined with debruijn intrinsic scoping
- It mixes terms construction with typing proof construction
  - is beneficial to saving code and forces you to always consider types
  - but reduction rules are messed with proof of type preservation
- I wouldn't recommend to adopt this technique
  - when your calculus is in a experimental phase
  - but it's a nice try to formalise classical ones where required theorems are already clear

Further reading

- Intrinsic typing in Coq (https://github.com/annenkov/stlcnorm)
- Full proof of PCF (PLFA Chapter Debruijn)
- Integration of language constructs (PLFA Chapter More)
- Discussion of bidirectional typing (PLFA Chapter Inference)
  - by defining a translate function from extrinsic typing to intrinsic
- Advanced language features (System *F* in Agda, for fun and profit)
  - Parametric polymorphism
  - Higher-order types
  - Iso-recursive types

Questions

- Q: Worry about more advanced feature, like dependent types?
- Q: Sub constructor previously didn't appear in the term, but now?
- Q: How to measure the equality of two terms (one term may have two derivations)?