Towards Tagless Interpretation of Stratified System F

Extended Abstract

Peter Thiemann
thiemann@acm.org
University of Freiburg
Germany

Marius Weidner
weidner@cs.uni-freiburg.de
University of Freiburg
Germany

Abstract

We explore the definition of an intrinsically typed interpreter for stratified System F in Agda.

Keywords: Agda, stratified System F, extensionality

ACM Reference Format:
Peter Thiemann and Marius Weidner. 2023. Towards Tagless Interpretation of Stratified System F: Extended Abstract. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 4 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Defining semantics is one of the key activities of a programming languages researcher. We learn that there are different styles of dynamics (small-step, big-step, denotational, just to name the most frequently used one), each with different trade-offs. When it comes to implementing or mechanizing semantics, there are further options to choose from, in particular if we are also interested in statics.

One important choice is whether we want to express the statics extrinsically or intrinsically, that is, do we want to start with untyped syntax and then define the statics as an afterthought, or do we integrate types with the syntax.

If we opt for intrinsically typed syntax, some properties are already paid for by construction. For instance, a small-step semantics for intrinsically typed syntax satisfies type preservation by construction. For another instance, consider specifying a denotational semantics by a compositional mapping from syntax to some semantic domain. With untyped syntax, the semantic domain has to lump the interpretations of different types together and distinguish them using type tags. But with intrinsically typed syntax the semantics can map into type-indexed semantic domains and thus elide type tags. This observation directly translates to tagless interpreters on intrinsically typed syntax, which elide tag checks at run time.

For concreteness, we show the well-known example of a tagless interpreter for the simply-typed lambda calculus implemented in Agda in Figure 1. We define the syntax as an inductive data type along with a compositional mapping to the semantic domain, spanned by Agda's natural number type and the function space. We define intrinsically typed syntax of expressions as an inductive datatype parameterized over a typing environment and indexed on the return type. For variables, we use de Bruijn indices into the typing environment.

The semantics of a typing environment is a run-time environment in the form of a heterogenous list of suitably typed values. With all that, we can define the semantics of an expression $\mathcal{E}[\![\_]\!]$ as a function from the semantics of a typing environment $\mathcal{G}[\![\_]\!]$ to the semantics of the type $\mathcal{T}[\![\_]\!]$. Clearly this definition also serves as a tagless interpreter for the simply-typed lambda calculus, which means that type preservation is also built into its definition. Moreover, as Agda accepts this definition as terminating, we know that evaluation of every simply-typed lambda term terminates; a non-trivial semantic property of the simply-typed lambda calculus.

Agda-encodings of intrinsically-typed interpreters have been explored quite a lot, but rarely in the context of polymorphic source languages. One possible reason is that the archetypical polymorphic lambda calculus, System F, cannot be embedded in Agda because of its impredicativity. This begs the question if we can develop a tagless interpreter for a predicative version of System F in Agda.

We answer this question affirmatively for Leivant's stratified version of the polymorphic lambda calculus [10]. The key idea of his calculus is to stratify the set of polymorphic types in levels such that universal quantification only ranges over strictly smaller levels. This restriction literally embodies predicativity and, as we will discover, the stratification corresponds directly to Agda's universe stratification.

2 Types

The definition of the type language for stratified System F is taken literally from Leivant's paper. It is defined as an inductive type parameterized over a level environment (that

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/XXXXXXX.XXXXXXX

```
module STLC where

open import Data.Nat using (ℕ; zero; suc)
open import Data.List using (List; []; _::_)

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

𝒯⟦_⟧ : Type → Set
𝒯⟦ nat ⟧ = ℕ
𝒯⟦ S ⇒ T ⟧ = 𝒯⟦ S ⟧ → 𝒯⟦ T ⟧

Env = List Type

data _∈_ : Type → Env → Set where
  here : ∀ {T Γ} → T ∈ (T :: Γ)
  there : ∀ {S T Γ} → S ∈ Γ → S ∈ (T :: Γ)

data Expr (Γ : Env) : Type → Set where
  con : ℕ → Expr Γ nat
  var : ∀ {T} → T ∈ Γ → Expr Γ T
  lam : ∀ {S T} → Expr (S :: Γ) T → Expr Γ (S ⇒ T)
  app : ∀ {S T} → Expr Γ (S ⇒ T) → Expr Γ S → Expr Γ T

data 𝒢⟦_⟧ : Env → Set where
  [] : 𝒢⟦ [] ⟧
  _::_ : ∀ {T Γ} → 𝒯⟦ T ⟧ → 𝒢⟦ Γ ⟧ → 𝒢⟦ T :: Γ ⟧

lookup : ∀ {T Γ} → T ∈ Γ → 𝒢⟦ Γ ⟧ → 𝒯⟦ T ⟧
lookup here (x :: _) = x
lookup (there x) (_ :: γ) = lookup x γ

ℰ⟦_⟧ : ∀ {Γ T} → Expr Γ T → 𝒢⟦ Γ ⟧ → 𝒯⟦ T ⟧
ℰ⟦ con n ⟧ γ = n
ℰ⟦ var x ⟧ γ = lookup x γ
ℰ⟦ lam e ⟧ γ = λ v → ℰ⟦ e ⟧ (v :: γ)
ℰ⟦ app e₁ e₂ ⟧ γ = ℰ⟦ e₁ ⟧ γ (ℰ⟦ e₂ ⟧ γ)
```

Figure 1. Simply typed lambda calculus, denotationally

assigns levels to free type variables) and indexed over the level of the type.

```
LEnv = List Level
data Type (Δ : LEnv) : Level → Set where
  nat : Type Δ zero
  _⇒_ : Type Δ ℓ → Type Δ ℓ′ → Type Δ (ℓ ⊔ ℓ′)
  `_ : ℓ ∈ Δ → Type Δ ℓ
  `∀ : ∀ ℓ → Type (ℓ :: Δ) ℓ′ → Type Δ (suc ℓ ⊔ ℓ′)
```

The unit type lives at level 0. Type variables live at their declared level. The level of a function type $S \Rightarrow T$ is the maximum of the levels of S and T. The level of a universal quantification at level l is the maximum of $l+1$ and the level of the body.

As for the simply-typed lambda calculus, we can define a compositional mapping from type syntax to Agda types.

```
𝒯⟦_⟧ : Type Δ ℓ → DEnv Δ → Set ℓ
𝒯⟦ nat ⟧ η = ℕ
𝒯⟦ T₁ ⇒ T₂ ⟧ η = 𝒯⟦ T₁ ⟧ η → 𝒯⟦ T₂ ⟧ η
𝒯⟦ ` α ⟧ η = apply-env η α
𝒯⟦ `∀ ℓ T ⟧ η = (D : Set ℓ) → 𝒯⟦ T ⟧ (ext-env D η)
```

Given a type at level l, this function returns an Agda type in Set l. To do so it needs a domain environment to interpret type variables. This environment gets extended in the last clause that maps universal quantification to a dependent function that takes an element of Set l and pushes it on the environment.

The type of the domain environment is interesting because its range type is unusual.

```
data DEnv : LEnv → Setω where
  [] : DEnv []
  _::_ : Set ℓ → DEnv Δ → DEnv (ℓ :: Δ)
```

As a value in the environment (the interpretation of a type, colloquially speaking) can live in a Set l, for any finite level l, we cannot assign the type any finite level. Hence, the type DEnv lives in the limit type Setω, which we use in this definition.

3 Expressions

Inspired by the encoding of System $F\omega$ by Chapman and coworkers [7], we define a unified environment for type variables and term variables. Type environments grow to the left.

```
data TEnv : LEnv → Set where
  ∅    : TEnv []
  _◁_ : Type Δ ℓ → TEnv Δ → TEnv Δ -- term variable
  _◁*_ : ∀ ℓ → TEnv Δ → TEnv (ℓ :: Δ) -- type variable
```

Membership of a term variable in a type environment is defined by the inn relation.

```
data inn : Type Δ ℓ → TEnv Δ → Set where
  here  : ∀ {T : Type Δ ℓ}{Γ}
        → inn T (T ◁ Γ)
  there : ∀ {T : Type Δ ℓ}{T′ : Type Δ ℓ′}{Γ}
        → inn T Γ → inn T (T′ ◁ Γ)
  tskip : ∀ {T : Type Δ ℓ}{Γ}
        → inn T Γ → inn (Twk T) (ℓ′ ◁* Γ)
```

In the last alternative, we skip over a type binding. Hence, the type T we find under the binding must be weakened to account for the extra type variables. Weakening is a special case of renaming, which is implemented as advocated by Benton and coworkers [4].

The type of expressions is now given as follows.

```
data Expr {Δ : LEnv} (Γ : TEnv Δ) : Type Δ ℓ → Set where
  #_  : ∀ (n : ℕ) → Expr Γ nat
  `_  : ∀ {T : Type Δ ℓ}
      → inn T Γ → Expr Γ T
  ƛ_  : ∀ {T : Type Δ ℓ}{T′ : Type Δ ℓ′}
      → Expr (T ◁ Γ) T′ → Expr Γ (T ⇒ T′)
  _·_ : ∀ {T : Type Δ ℓ}{T′ : Type Δ ℓ′}
      → Expr Γ (T ⇒ T′) → Expr Γ T → Expr Γ T′
  Λ   : ∀ (ℓ : Level) → {T : Type (ℓ :: Δ) ℓ′}
      → Expr (ℓ ◁* Γ) T → Expr Γ (`∀ ℓ T)
  _•_ : ∀ {T : Type (ℓ :: Δ) ℓ′}
      → Expr Γ (`∀ ℓ T) → (T′ : Type Δ ℓ)
      → Expr Γ (T [ T′ ]T)
```

Variables, lambda abstractions, and application are encoded just like for the simply-typed lambda calculus. Type abstraction takes a level l and a body where the new type variable is bound to l. Type application takes an expression with universal quantification at level l and a type T' of level l. It constructs an expression where type T' has been substituted in the body T of the quantified type. Substitution is defined as in PLFA [4, 9].

4 Semantics

It remains to define a compositional function from the expression syntax to the semantic domain that we already prepared in Section 2. We start with value environments.

```
Env : (Δ : LEnv) → TEnv Δ → DEnv Δ → Setω
Env Δ Γ η = ∀ {ℓ}{T : Type Δ ℓ} → inn T Γ → 𝒯⟦ T ⟧ η
```

Value environments are represented as functions—we could have done that in the simply-typed interpreter, too. They are indexed by a domain environment to be able to calculate the correct return type.

The definition of the interpretation function follows.

```
ℰ⟦_⟧ : ∀ {T : Type Δ ℓ}{Γ : TEnv Δ}
  → Expr Γ T → (η : DEnv Δ) → Env Δ Γ η → 𝒯⟦ T ⟧ η
ℰ⟦ # n ⟧ η γ = n
ℰ⟦ ` x ⟧ η γ = γ x
ℰ⟦ ƛ_ e ⟧ η γ = λ v → ℰ⟦ e ⟧ η (extend γ v)
ℰ⟦ e₁ · e₂ ⟧ η γ = ℰ⟦ e₁ ⟧ η γ (ℰ⟦ e₂ ⟧ η γ)
ℰ⟦ Λ ℓ e ⟧ η γ = λ D → ℰ⟦ e ⟧ (ext-env D η) (extend-tskip γ)
ℰ⟦ _•_ {T = T} e T′ ⟧ η γ =
  subst id (sym (single-subst-preserves T′ T))
    (ℰ⟦ e ⟧ η γ (𝒯⟦ T′ ⟧ η))
```

The cases for term variables, lambda abstraction, and application are similar to the simply-typed lambda calculus.

The first issue arises in the case for type abstraction. We interpret a type abstraction at level l as a function with argument type Set l. This argument has to be pushed onto the domain environment η and we have to account at the value level for the additional type variable in the type environment. The following function adapts the types.

```
extend-tskip : ∀ {Δ : LEnv}{Γ : TEnv Δ}{η : DEnv Δ}{D : Set ℓ}
  → Env Δ Γ η → Env (ℓ :: Δ) (ℓ ◁* Γ) (D :: η)
extend-tskip {η = η} {D = D} γ (tskip{T = T} x) =
  subst id (sym (ren*-preserves-semantics {ρ = wkᵣ}{η}{D :: η}
                (wkᵣ∈TRen* η D) T))
           (γ x)
```

The lemma we need in the rewrite clause proves that interpreting a weakened type in an extended domain environment gives the same result as interpreting the type in the orginal domain environment. The statement of this lemma is more general as it applies to arbitrary renamings:

```
ren*-preserves-semantics :
  ∀ {ρ : TRen Δ₁ Δ₂}{η₁ : DEnv Δ₁}{η₂ : DEnv Δ₂}
  → (ren* : TRen* ρ η₁ η₂) → (T : Type Δ₁ ℓ)
  → 𝒯⟦ Tren ρ T ⟧ η₂ ≡ 𝒯⟦ T ⟧ η₁
```

The argument $ren*$ roughly states that η_1 and η_2 are domain environments related by renaming ρ by precomposition $\eta_1 \equiv \eta_2 \circ \rho$. The proof of the lemma is by induction on T with an interesting subgoal in the case for universal quantification:

```
(T : Type (ℓ :: Δ₁) ℓ′) →
((D : Set ℓ) → 𝒯⟦ Tren (extᵣ ρ) T ⟧ (D :: η₂)) ≡
  ((D : Set ℓ) → 𝒯⟦ T ⟧ (D :: η₁))
```

We can show that the ranges of the function are equal with the inductive hypothesis. But the usual extensionality principle does not let us expose this equation. However, it can be used to prove a dependent extensionality principle (from the standard library), which enables us to complete the proof.

```
∀-extensionality :
  ∀ {a b}{A : Set a}{F G : (α : A) → Set b}
  → (∀ (α : A) → F α ≡ G α)
  → ((α : A) → F α) ≡ ((α : A) → G α)
```

The final case for type application opens two different cans of worms. First, the type of the right hand side does not match the expected type. Essentially, we have to prove that the composition of the meaning function for types commutes with substitution. Here $T[T']$ substitutes T' for the outermost variable of T.

```
single-subst-preserves :
  ∀ {η : DEnv Δ} (T′ : Type Δ ℓ) (T : Type (ℓ :: Δ) ℓ′)
  → 𝒯⟦ T [ T′ ]T ⟧ η ≡ 𝒯⟦ T ⟧ (𝒯⟦ T′ ⟧ η :: η)
```

Second, some steps in the proof involve equalities over entities of Setω. These cannot be handled with the standard definition of propositional equality which works parametrically for entities of Set l, for any l, but not for Setω. While it is easy to define these equalities, it is somewhat tedious to re-establish standard lemmas for transforming equality proofs like cong, subst, and trans to deal with Setω.

5 Related Work

Arjen Rouvoet's thesis [14] gives an excellent overview of the state of the art in intrinsically typed techniques for modeling language semantics. He pushed the limits of this technology in a range of papers with various coauthors [11, 15, 16, 18].

Giving semantics in an interpretive style is a defining feature of denotational semantics [17], but it can be traced back to Reynolds's idea of definitional interpreters [12].

The particular encoding of de Bruijn style variable representations used in this work dates back to work on nested datatypes [2, 5, 6], which subsequently lead to GADTs [8].

A tagless interpreter for a simply typed calculus was developed in Cayenne [3], but for extrinsically typed syntax (i.e., syntax and separate typing predicatepe).

Intrinsically typed encodings are further studied by Allais and others [1], who define a range of tagless functions including denotational semantics on intrinsically typed terms.

We draw some inspiration from the program implemented by Benton and coworkers [4]. Based on intrinsically typed syntax for a simply-typed lambda applied calculus, they define a big-step semantics and a set-theoretic denotational semantics. They prove soundness of the former semantics with respect to the latter as well as adequacy (using a logical relation). They also develop an expression encoding for System F, but the paper stops short of discussing its semantics.

6 Future Work

We are currently formalizing the small-step semantics of the language with the goal of proving an adequacy theorem for reduction with respect to the denotational semantics.

It would also be interesting to extend the stratified calculus by level quantification as in Agda's universe polymorphism.

References

[1] Guillaume Allais, James Chapman, Conor McBride, and James McKinna. 2017. Type-and-scope safe programs and their proofs. In *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs, CPP 2017*, Yves Bertot and Viktor Vafeiadis (Eds.). ACM, Paris, France, 195–207. https://doi.org/10.1145/3018610.3018613

[2] Thorsten Altenkirch and Bernhard Reus. 1999. Monadic Presentations of Lambda Terms Using Generalized Inductive Types. In *Computer Science Logic, CSL '99, 8th Annual Conference of the EACSL (Lecture Notes in Computer Science, Vol. 1683)*, Jörg Flum and Mario Rodríguez-Artalejo (Eds.). Springer, Madrid, Spain, 453–468. https://doi.org/10.1007/3-540-48168-0_32

[3] Lennart Augustsson and Magnus Carlsson. 1999. An exercise in dependent types: A well-typed interpreter. (May 1999). https://www.researchgate.net/publication/2610129 unpublished manuscript.

[4] Nick Benton, Chung-Kil Hur, Andrew Kennedy, and Conor McBride. 2012. Strongly Typed Term Representations in Coq. *J. Autom. Reason.* 49, 2 (2012), 141–159. https://doi.org/10.1007/s10817-011-9219-0

[5] Richard S. Bird and Lambert G. L. T. Meertens. 1998. Nested Datatypes. In *Mathematics of Program Construction, MPC'98 (Lecture Notes in Computer Science, Vol. 1422)*, Johan Jeuring (Ed.). Springer, Marstrand, Sweden, 52–67. https://doi.org/10.1007/BFb0054285

[6] Richard S. Bird and Ross Paterson. 1999. De Bruijn Notation as a Nested Datatype. *J. Funct. Program.* 9, 1 (1999), 77–91. https://doi.org/10.1017/s0956796899003366

[7] James Chapman, Roman Kireev, Chad Nester, and Philip Wadler. 2019. System F in Agda, for Fun and Profit. In *Mathematics of Program Construction - 13th International Conference, MPC 2019 (Lecture Notes in Computer Science, Vol. 11825)*, Graham Hutton (Ed.). Springer, Porto, Portugal, 255–297. https://doi.org/10.1007/978-3-030-33636-3_10

[8] James Cheney and Ralf Hinze. 2003. *First-Class Phantom Types.* Technical Report TR2003-1901. Cornell University. https://ecommons.cornell.edu/handle/1813/5614

[9] Wen Kokke, Jeremy G. Siek, and Philip Wadler. 2020. Programming language foundations in Agda. *Sci. Comput. Program.* 194 (2020), 102440. https://doi.org/10.1016/j.scico.2020.102440

[10] Daniel Leivant. 1991. Finitely Stratified Polymorphism. *Inf. Comput.* 93, 1 (1991), 93–113. https://doi.org/10.1016/0890-5401(91)90053-5

[11] Casper Bach Poulsen, Arjen Rouvoet, Andrew Tolmach, Robbert Krebbers, and Eelco Visser. 2018. Intrinsically-typed definitional interpreters for imperative languages. *Proc. ACM Program. Lang.* 2, POPL (2018), 16:1–16:34. https://doi.org/10.1145/3158104

[12] John C. Reynolds. 1975. User-defined Types and Procedural Data Structures as Complementary Approaches to Data Abstraction. In *New Directions in Algorithmic Languages*, Stephen A. Schumann (Ed.). INRIA, St. Pierre-de-Chartreuse, 309–317. Reprinted in [13].

[13] John C. Reynolds. 1994. User-defined Types and Procedural Data Structures as Complementary Approaches to Data Abstraction. In *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design*, Carl A. Gunter and John C. Mitchell (Eds.). MIT Press, Cambridge, MA, USA, 13–23. Originally published in [12].

[14] Arjen Rouvoet. 2021. *Correct by Construction Language Implementations.* Ph. D. Dissertation. Delft University of Technology, Netherlands. https://doi.org/10.4233/uuid:f0312839-3444-41ee-9313-b07b21b59c11

[15] Arjen Rouvoet, Robbert Krebbers, and Eelco Visser. 2021. Intrinsically typed compilation with nameless labels. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–28. https://doi.org/10.1145/3434303

[16] Arjen Rouvoet, Casper Bach Poulsen, Robbert Krebbers, and Eelco Visser. 2020. Intrinsically-typed definitional interpreters for linear, session-typed languages. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2020*, Jasmin Blanchette and Catalin Hritcu (Eds.). ACM, New Orleans, LA, USA, 284–298. https://doi.org/10.1145/3372885.3373818

[17] David A. Schmidt. 1986. *Denotational Semantics, A Methodology for Software Development.* Allyn and Bacon, Inc, Massachusetts. https://people.cs.ksu.edu/~schmidt/text/ds0122.pdf

[18] Cas van der Rest, Casper Bach Poulsen, Arjen Rouvoet, Eelco Visser, and Peter D. Mosses. 2022. Intrinsically-typed definitional interpreters à la carte. *Proc. ACM Program. Lang.* 6, OOPSLA2 (2022), 1903–1932. https://doi.org/10.1145/3563355