

Technical Outline

Eric Giovannini

April 14, 2023

1 The Overall Plan

The goal is to show graduality of the extensional gradual lambda calculus. The main theorem is the following:

Theorem 1.1 (Graduality). *If $M \sqsubseteq M'$, then*

- $M \Downarrow \text{iff } M' \Downarrow$.
- $M \Downarrow v_? \text{ iff } M' \Downarrow v'_?, \text{ where } v_? = \mathcal{U} \text{ or } v_? = v'_?.$

2 Syntax

We begin with a gradually-typed lambda calculus ($\text{Ext-}\lambda$), which is similar to the normal call-by-value gradually-typed lambda calculus, but differs in that it is actually a fragment of call-by-push-value specialized such that there are no non-trivial computation types. We do this for convenience, as either way we would need a distinction between values and effectful terms; the framework of call-by-push-value gives us a convenient language to define what we need.

We then observe that composition of type precision derivations is admissible, as is heterogeneous transitivity for term precision (via casts), so it will suffice to consider a new language ($\text{Ext-}\lambda^{\text{-trans}}$) in which we don't have composition of type precision derivations or transitivity of term precision.

We further observe that all casts, except those between Nat and $?$ and between $? \multimap ?$ and $?$, are admissible (we can define the cast of a function type functorially using the casts for its domain and codomain).

This means it is sufficient to consider a new language ($\text{Ext-}\lambda^{\text{-trans-cast}}$) in which instead of having arbitrary casts, we have injections from Nat and $? \multimap ?$ into $?$, and case inspections from $?$ to Nat and $?$ to $? \multimap ?$.

From here, we define an intensional GSTLC, so-named because it makes the intensional stepping behavior of programs explicit in the syntax. This is accomplished by adding a syntactic “later” type and a syntactic θ that maps terms of type later A to terms of type A .

2.1 Extensional (Ext- $\lambda^{\text{trans-cast}}$)

Value Types $A := \text{Nat} \mid ? \mid (A \multimap B)$

Computation Types $B := \text{Ret}A$

Value Contexts $\Gamma := \cdot \mid (\Gamma, x : A)$

Computation Contexts $\Delta := \cdot \mid \bullet : B \mid \Delta, x : A$

Values $V := \text{zro} \mid \text{suc } V \mid \text{Inj}_{\text{nat}}(V) \mid \text{Inj}_{\rightarrow}(V)$

Terms $M, N := \mathcal{U}_B \mid \text{Ret}V \mid \text{var } x = M \text{ in } N \mid \lambda x. M \mid V_f V_x \mid$

$\text{Case}_{\text{nat}}(V)\{\text{no} \rightarrow M_{\text{no}} \mid \text{nat}(n) \rightarrow M_{\text{yes}}\} \mid \text{Case}_{\rightarrow}(V)\{\text{no} \rightarrow M_{\text{no}} \mid \text{fun}(f) \rightarrow M_{\text{yes}}\}$

The value typing judgment is written $\Gamma \vdash V : A$ and the computation typing judgment is written $\Delta \vdash M : B$.

We define substitution for computation contexts as follows:

$$\frac{\delta : \Delta' \rightarrow \Delta \quad \Delta'|_V \vdash V : A}{(\delta, V/x) : \Delta' \rightarrow \Delta, x : A} \quad \cdot : \cdot \rightarrow \cdot \quad \frac{\Delta' \vdash M : B}{M : \Delta' \rightarrow \bullet : B}$$

The typing rules are as follows:

$$\begin{array}{c} \frac{}{\cdot, \Gamma \vdash \mathcal{U}_B : B} \quad \frac{}{\Gamma \vdash \text{zro} : \text{Nat}} \quad \frac{\Gamma \vdash M : \text{Nat}}{\Gamma \vdash \text{suc } M : \text{Nat}} \quad \frac{\cdot, \Gamma, x : A \vdash M : \text{Ret}A'}{\Gamma \vdash \lambda x. M : A \multimap A'} \\[10pt] \frac{\Gamma \vdash V_f : A \multimap A' \quad \Gamma \vdash V_x : A}{\cdot, \Gamma \vdash M N : \text{Ret}A'} \quad \frac{\Gamma \vdash V : A}{\cdot, \Gamma \vdash \text{ret}V : \text{Ret}A} \\[10pt] \frac{\Delta \vdash M : \text{Ret}A \quad \cdot, \Delta|_V, x : A \vdash N : B}{\Delta \vdash \text{var } x = M \text{ in } N : B} \quad \frac{\Gamma \vdash M : \text{Nat}}{\Gamma \vdash \text{Inj}_{\text{nat}}(M) : ?} \\[10pt] \frac{\Gamma \vdash M : (? \multimap ?)}{\Gamma \vdash \text{Inj}_{\rightarrow}(M) : ?} \quad \frac{\Gamma \vdash V : ? \quad \Delta, x : \text{Nat} \vdash M_{\text{yes}} : B \quad \Delta \vdash M_{\text{no}} : B}{\Delta \vdash \text{Case}_{\text{nat}}(M)\{\text{no} \rightarrow M_{\text{no}} \mid \text{nat}(n) \rightarrow M_{\text{yes}}\} : B} \\[10pt] \frac{\Gamma \vdash V : ? \quad \Delta, x : (? \multimap ?) \vdash M_{\text{yes}} : B \quad \Delta \vdash M_{\text{no}} : B}{\Delta \vdash \text{Case}_{\rightarrow}(M)\{\text{no} \rightarrow M_{\text{no}} \mid \text{fun}(f) \rightarrow M_{\text{yes}}\} : B} \end{array}$$

The equational theory is as follows:

$$\frac{\cdot, \Gamma, x : A \vdash M : \text{Ret}A' \quad \Gamma \vdash V : A}{(\lambda x.M)V = M[V/x]} \quad \frac{\Gamma \vdash V : A \multimap A}{\Gamma \vdash V = \lambda x.Vx}$$

$$\frac{\Gamma \vdash V : \text{Nat}}{\text{Case}_{\text{nat}}(\text{Inj}_{\text{nat}}(V))\{\text{no} \rightarrow M_{\text{no}} \mid \text{nat}(n) \rightarrow M_{\text{yes}}\} = M_{\text{yes}}[V/n]}$$

$$\frac{\Gamma \vdash V : ? \multimap ?}{\text{Case}_{\text{nat}}(\text{Inj}_{\rightarrow}(V))\{\text{no} \rightarrow M_{\text{no}} \mid \text{nat}(n) \rightarrow M_{\text{yes}}\} = M_{\text{no}}}$$

$$\Gamma, x : ? \vdash M = \text{Case}_{\text{nat}}(x)\{\text{no} \rightarrow M \mid \text{nat}(n) \rightarrow M[(\text{Inj}_{\text{nat}}(n))/x]\}$$

$$\frac{\bullet : \text{Ret}A, \Gamma \vdash M : B}{\bullet : \text{Ret}A, \Gamma \vdash M = \text{var } x = \bullet \text{ in } M[\text{ret } x]} \quad \text{var } x = \text{ret } V \text{ in } N = N[V/x]$$

Equivalent terms in the equational theory are considered equal.

We now discuss type and term precision. In our language, we do not have “term precision” but rather arbitrary monotone relations on types, which we denote by $A \circ \bullet B$. We have relations on value types, as well as on computation types. In addition, because we don’t have casts in our language, we do not have the usual cast rules specifying that casts are least upper bounds and greatest lower bounds. Instead, we have rules for our injection and case terms.

$$\begin{aligned} \text{Value Relations } R &:= \text{Nat} \mid ? \mid (R \multimap R) \mid \alpha_1 \leq_A \alpha_2 \\ \text{Computation Relations } S &:= \text{Lift}R \end{aligned}$$

2.2 Intensional

In the intensional syntax, we add a type constructor for later, as well as a syntactic θ term and a syntactic next term. We add rules for each of these, and also modify the rules for inj-arr and case-arr, since now the function is not $\text{Dyn} \multimap \text{Dyn}$ but rather $\triangleright (\text{Dyn} \multimap \text{Dyn})$.

We define an erasure function from intensional syntax to extensional syntax by induction on the intensional types and terms. The basic idea is that the syntactic type $\triangleright A$ erases to A , and next and θ erase to the identity.

3 Semantics

To give a semantics of the extensional, quotiented syntax ($\text{Ext-}\lambda^{-\text{trans-cast}}$), we proceed in a few steps:

1. We first give a denotational semantics of the intensional syntax $\text{Int-}\lambda$ using synthetic guarded domain theory, i.e., working internal to the topos of trees. The denotation of a value type A will be a poset $\llbracket A \rrbracket$, and the denotation of a computation type $\text{Ret}A$ will be the lift $L_{\mathcal{U}}\llbracket A \rrbracket$, (i.e., the free error domain on $\llbracket A \rrbracket$). We refer to this semantics as the *step-indexed semantics*.
2. We then show how to go from the above step-indexed semantics to a set-based semantics which is still intensional in that the denotations of terms that differ only in their number of “steps” will not be equal. The denotations here are a type of coinductive “Machine” that at each step of computation can either return a result, error, or continue running. We call this the *Machine semantics*. The passage from step-indexed semantics to the coinductive semantics will make use of clock quantification; see below for details.
3. Finally, we collapse the above intensional Machine-based semantics to an extensional semantics, so-called because at this point the information about the precise number of steps a Machine has taken has been lost. We care only whether the Machine runs to a value or error, or whether it diverges. Of course, we cannot in general decide whether a given Machine will halt, so the semantic objects here are pairs of a proposition P and a function taking a proof of P to an element of the poset.

The benefit to working synthetically in step 1 above is that the construction of the logical relation that proves canonicity for the intensional syntax can be carried out much like any normal, non-step-indexed logical relation.

4 Unary Canonicity

We first want to show soundness:

Lemma 4.1 (Soundness). *In the Machine semantics, none of the following are equal: \mathcal{U} , zro , $\text{suc}M$, ω , where ω represents a diverging program.*

Next we use a logical relations argument to establish the following:

Lemma 4.2. $M_i \Downarrow^n v? \text{ iff } M_i = \delta^n(\text{quote}(v?))$

We will need the following key property relating erasure to the semantics:

Lemma 4.3. *If $\lfloor M_i \rfloor = M$ and $\lfloor M'_i \rfloor = M'$, and $M = M'$, then $\llbracket M_i \rrbracket = \llbracket M'_i \rrbracket$.*

The idea of the proof is that since M_i and M'_i have equal erasures (in the extensional equational theory), we can recover from the proof that their erasures are equal a series of equational rules (e.g., β equality), and apply the intensional analogues of those rules to M_i to ensure that M_i and M'_i differ syntactically only in their number of θ 's.

5 Graduality