Synthetic Guarded Domain Theory + Gradual Typing

Eric Giovannini and Max New

University of Michigan

MPLSE Reading Group
March 24, 2023

Overview

What is Gradual Typing?

Gradually-typed languages combine static and dynamic typing in a single language and allow smooth interaction between both typed and untyped code.

This allows programmers to get the best of both worlds: they can start off programming in an untyped style and later annotate the code with types.

Doing so should not alter the semantics of the program!

Gradually-typed languages are usually compiled to **cast calculi** where the casts are made explicit.

Graduality

Gradual Guarantee (Siek et al. [7]): Key property for a language to be considered gradually-typed.

Adding type annotations should not change the semantics of the program, except to possibly introduce type errors.

Conversely: Removing type annotations should not change the behavior of the program.

Type and Term Precision

Type Precision: $A \sqsubseteq B$ means that A is more precise than B, or equivalently, B is more dynamic
Least precise type: ? (i.e., $A \sqsubseteq ?$ for all A)

Term precision: Extension of type precision to terms
Intuitively: $M \sqsubseteq N$ means "M behaves like N, but may error more"
For each type A, there is an error-term $\mho_A$ such that $\mho_A \sqsubseteq M$ for all $M : A$.

In the cast calculus, we allow casts between types A and B such that $A \sqsubseteq B$.

The Current Approach to Proving Graduality

Define a notion of *contextual error approximation* (two programs are equivalent, up to one erroring more than the other)

Construct a *logical relations model* and show that it is sound with respect to contextual error approximation.

This approach has been utilized by New and Ahmed [5] and New, Licata, and Ahmed [6].

Step Indexing

The logical relation must be *step-indexed* in order to deal with issues of non-wellfoundnedness i.e. we index the relation by a natural number representing the “fuel” we have left to observe the expression. Whenever a non well-founded operation takes place, we decrement the step-index.

This has a few downsides:

- Need to keep track of step index throughout the proofs
- Need two seaprate expression logical relations (one that counts steps on the left, and one on the right)
- Transitivity of the logical relation is not straightforward

What is SGDT?

SGDT is a logic/type theory with certain new axioms that internalize the notion of step-indexing.

There is an endofunctor $\triangleright : \mathtt{Type} \to \mathtt{Type}$, where $\triangleright A$ represents values of type A available one time step later.

There is a "delaying" function $\mathsf{next} : A \to \triangleright A$ that takes a value available now and views it as a value available later.

SGDT: Guarded Fixpoints

Fixpoint operator $\mathsf{fix} : (\triangleright A \to A) \to A$.

Idea: to construct an A "now", it suffices to assume we have an A "later" and use that to build an A "now".

When used for propositions, this is called "Löb-induction".

Fix satisfies the following unrolling equation:

$$\mathsf{fix}(f) = f(\mathsf{next}(\mathsf{fix}(f)))$$

Clocks and Clock Quantification

SGDT comes with a notion of clocks, abstract objects which keep track of time steps.

The operations above are with repsect to a given clock κ, e.g, we have $\triangleright^{\kappa}$.

The notion of *clock quantification* is crucial for encoding coinductive types using guarded recursion, an idea first introduced by Atkey and McBride [1].

The Topos of Trees Model

The denotational semantics of SGDT is in a category called the *topos of trees*, denoted $\mathcal{S} = \mathrm{Set}^{\omega^o}$.

Objects: presheaves over the ordered natural numbers, i.e., families $\{X_i\}$ of sets indexed by natural numbers, along with restriction maps $r_i^X \colon X_{i+1} \to X_i$.

Morphisms $\{X_i\}$ to $\{Y_i\}$: family of functions $f_i \colon X_i \to Y_i$ that commute with the restriction maps in the obvious way, that is, $f_i \circ r_i^X = r_i^Y \circ f_{i+1}$.

Denotations of Later, Next, and Fix

The type operator $\triangleright$ is defined on an object X by $(\triangleright X)_0 = 1$ and $(\triangleright X)_{i+1} = X_i$. The restriction maps are given by $r_0^{\triangleright} = !$, where ! is the unique map into 1, and $r_{i+1}^{\triangleright} = r_i^X$.

The morphism $\text{next}^X : X \to \triangleright X$ is defined pointwise by $\text{next}_0^X = !$, and $\text{next}_{i+1}^X = r_i^X$.

Given a morphism $f \colon \triangleright X \to X$, we define $\text{fix}\, f$ pointwise as $\text{fix}_i(f) = f_i \circ \cdots \circ f_0$.

Note that as defined, fix isn't actually a morphism in $\mathcal{S}$: what is its source? We need an object for functions from $\triangleright X \to X$. This is the internal hom $\triangleright X \Rightarrow X$.

We can then define $\text{fix} \colon (\triangleright X \Rightarrow X) \to X$; we omit the details.

Denotations of Later, Next, and Fix

In $\mathcal{S}$

$$
\begin{tikzcd}
X \arrow[d, "\mathrm{next}"] \\
\triangleright X
\end{tikzcd}
$$

In Set

$$
\begin{tikzcd}
X_0 \arrow[d, "!"'] & X_1 \arrow[l, "r_0^X"'] \arrow[d, "r_0^X"'] & X_2 \arrow[l, "r_1^X"'] \arrow[d, "r_1^X"'] & X_3 \arrow[l, "r_2^X"'] \arrow[d, "r_2^X"'] & \cdots \arrow[l] \\
1 & X_0 \arrow[l, "!"] & X_1 \arrow[l] & X_2 \arrow[l] & \cdots \arrow[l]
\end{tikzcd}
$$

Ticked Cubical Type Theory

In Ticked Cubical Type Theory [3], there is an additional sort called *ticks*.

Given a clock k, a tick $t : \mathsf{tick}\, k$ serves as evidence that one unit of time has passed according to the clock k.

The type $\triangleright^k A$ is represented as a function from ticks of a clock k to A.

The type A is allowed to depend on t, in which case we write $\triangleright_t^k A$ to emphasize the dependence.

The rules for tick abstraction and application are similar to those of dependent Π types.

GTLC: Syntax

Syntax

$$
\begin{aligned}
\text{Types } A, B &:= \text{Nat}, ?, (A \Rightarrow B) \\
\text{Terms } M, N &:= \mho_A, \text{zro}, \text{suc}\, M, (\lambda x.M), (M\,N), \\
&\quad (\langle B \leftarrowtail A \rangle M), (\langle A \twoheadleftarrow B \rangle M) \\
\text{Contexts } \Gamma &:= \cdot, (\Gamma, x : A)
\end{aligned}
$$

GTLC: Typing

$$\frac{}{\Gamma \vdash \mho_A : A} \qquad \frac{}{\Gamma \vdash \mathsf{zro} : \mathsf{Nat}} \qquad \frac{\Gamma \vdash M : \mathsf{Nat}}{\Gamma \vdash \mathsf{suc}\, M : \mathsf{Nat}}$$

$$\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x. M : A \Rightarrow B} \qquad \frac{\Gamma \vdash M : A \Rightarrow B \qquad \Gamma \vdash N : A}{\Gamma \vdash M\, N : B}$$

$$\frac{A \sqsubseteq B \qquad \Gamma \vdash M : A}{\Gamma \vdash \langle B \leftarrowtail A \rangle M : B} \qquad \frac{A \sqsubseteq B \qquad \Gamma \vdash M : B}{\Gamma \vdash \langle A \twoheadleftarrow B \rangle M : A}$$

GTLC: Type Precision

$$\frac{}{? \sqsubseteq ?}\ ? \qquad \frac{}{\mathrm{Nat} \sqsubseteq \mathrm{Nat}}\ \mathrm{Nat} \qquad \frac{}{\mathrm{Nat} \sqsubseteq ?}\ \mathit{Inj}_{\mathrm{Nat}}$$

$$\frac{A_i \sqsubseteq B_i \qquad A_o \sqsubseteq B_o}{(A_i \Rightarrow A_o) \sqsubseteq (B_i \Rightarrow B_o)} \Rightarrow \qquad \frac{(A_i \to A_o) \sqsubseteq (? \Rightarrow ?)}{(A_i \to A_o) \sqsubseteq ?}\ \mathit{Inj}_{\Rightarrow}$$

Precision Derivations:
For every $A \sqsubseteq B$, we have a *type precision derivation* $d : A \sqsubseteq B$ that is constructed using the rules above.

For any type A, we use A to denote the reflexivity derivation that $A \sqsubseteq A$, i.e., $A : A \sqsubseteq A$.

For type precision derivations $d : A \sqsubseteq B$ and $d' : B \sqsubseteq C$, we can define their composition $d' \circ d : A \sqsubseteq C$.

GTLC: Term Precision

Three kinds of rules: Congruence, Equational, and Cast Rules

Congruence rules: one per term constructor (except for casts)
Two examples (other rules omitted):

$$\frac{d : A \sqsubseteq B \qquad \Gamma^{\sqsubseteq}(x) = (A, B)}{\Gamma^{\sqsubseteq} \vdash x \sqsubseteq_e x : d}\ \text{Var}$$

$$\frac{d_i : A_i \sqsubseteq B_i \qquad d_o : A_o \sqsubseteq B_o \qquad \Gamma^{\sqsubseteq}, x : d_i \vdash M \sqsubseteq_e N : d_o}{\Gamma^{\sqsubseteq} \vdash \lambda x.M \sqsubseteq_e \lambda x.N : (d_i \Rightarrow d_o)}\ \text{Lambda}$$

GTLC: Term Precision

Equational Rules: Transitivity, β and η laws

$$\frac{\Gamma^{\sqsubseteq} \vdash M \sqsubseteq_e N : d \qquad \Gamma^{\sqsubseteq} \vdash N \sqsubseteq_e P : d'}{\Gamma^{\sqsubseteq} \vdash M \sqsubseteq_e P : d' \circ d} \text{ Transitivity}$$

$$\frac{\Gamma \vdash V : A_i \Rightarrow A_o}{\Gamma^{\sqsubseteq} \vdash \lambda x.(V\, x) \sqsupseteq\sqsubseteq_e V : A_i \Rightarrow A_o}\ \eta$$

GTLC: Term Precision

Cast Rules

$$\frac{d : A \sqsubseteq B \qquad \Gamma \vdash M : A}{\Gamma^{\sqsubseteq} \vdash M \sqsubseteq_e \langle B \leftarrowtail A \rangle M : d}\ \text{UpR}$$

$$\frac{d : A \sqsubseteq B \qquad \Gamma^{\sqsubseteq} \vdash M \sqsubseteq_e N : d}{\Gamma^{\sqsubseteq} \vdash \langle B \leftarrowtail A \rangle M \sqsubseteq_e N : B}\ \text{UpL}$$

(The other rules DnL, DnR are dual.)

The cast rules say that upcasts are least upper bounds, and dually, downcasts are greatest lower bounds.

Graduality for GTLC

Theorem (Graduality at Base Type)

If $\cdot \vdash M \sqsubseteq N : Nat$*, then*

1. *If* $N = \mho$*, then* $M = \mho$
2. *If* $N = V$*, then* $M = \mho$ *or* $M = V$*, where* $V = zro$ *or* $V = suc\, V'$
3. *If* $M = V$*, then* $N = V$

We also should be able to show that $\mho$, zro, and suc N are not equal.

Intensional GTLC

In addition to the above language, which we call the *extensional GTLC* (Ext-λC for short), we formalize the *intensional GTLC* (Int-λC for short).

Int-λC includes syntax to express “delayed” terms as terms, via the term constructor θ_s taking a term “later” to a term “now”.

$$\text{Terms } M, N := \mho_A, \ \ldots \ \theta_s(\tilde{M})$$

Typing:

$$\frac{\triangleright_t \ (\Gamma \vdash M_t : A)}{\Gamma \vdash \theta_s M : A}$$

Term Precision:

$$\frac{\triangleright_t \ (\Gamma^{\sqsubseteq} \vdash M_t \sqsubseteq_i N_t : d)}{\Gamma^{\sqsubseteq} \vdash \theta_s M \sqsubseteq_i \theta_s N : d}$$

Recall that $\triangleright_t$ is a dependent form of $\triangleright$ where the arugment is allowed to mention t. In particular, here we apply the tick t to the later-terms M and N to get "now"-terms M_t and N_t.

The Lift Monad

Datatype that represents computations that at each step can return a value (η), terminate with an error ($\mho$), or "think", i.e., defer the result to a later step (θ).

Definition (Lift Monad)

$$
\begin{aligned}
L_\mho A :=\ & \\
& \eta \colon A \to L_\mho A \\
& \mho \colon L_\mho A \\
& \theta \colon \ \triangleright (L_\mho A) \to L_\mho A
\end{aligned}
$$

There is a computation $\mathrm{fix}(\theta)$ of type $L_\mho A$; this represents a computation that thinks forever and never returns a value.

Notation: We define $\delta \colon L_\mho A \to L_\mho A$ by $\delta = \theta \circ \mathrm{next}$

Predomains and Monotone Functions

A **predomain** A consists of a type (which we denote $\langle A \rangle$) and a relation $\leq_A$ on A that satisfies the axioms of a partial ordering.

Since our types have an underlying order structure (representing the error ordering), we want to model types as partially-ordered sets in the semantics.

Then functions between terms will be modeled as *monotone* functions between their corresponding predomains.

We write $f : A \to_m B$ to indicate that f is a monotone function from A to B, i.e, for all $a_1 \leq_A a_2$, we have $f(a_1) \leq_B f(a_2)$.

Predomains

We define predomains for natural numbers, the dynamic type (which we denote D), and for monotone functions between predomains (which we denote $A_i \Rightarrow A_o$).

For Dyn, the underlying type is defined to be

$$\langle D \rangle = \mathbb{N} + \triangleright (D \to_m D)$$

This definition is valid because the occurrences of D are guarded by the $\triangleright$. The ordering is defined via guarded recursion by cases on the argument.

We also define a predomain for the "lifting" of a predomain by the $L_\mho$ monad. We denote this by $L_\mho A$.

Lock-Step Ordering and Weak Bisimilarity

For a predomain A, the ordering on $L_{\mho} A$ is called the "lock-step error ordering", denoted $l \lesssim l'$.
Intuitively: l is less than l' if they are in lock-step with regard to their intensional behavior, up to l erroring.

- $\eta\, x \lesssim \eta\, y$ if $x \leq_A y$.
- $\mho \lesssim l$ for all l
- $\theta\, \tilde{r} \lesssim \theta\, \tilde{r}'$ if $\triangleright_t\, (\tilde{r}_t \lesssim \tilde{r}'_t)$

We analogously define a lifting of a heterogeneous relation R between A and B to a relation $L(R)$ between $L_{\mho} A$ and $L_{\mho} B$.

Lock-Step Ordering and Weak Bisimilarity

We also define another ordering on $L_{\mho} A$, called "weak bisimilarity", written $l \approx l'$.
We say $l \approx l'$ if they are equivalent "up to delays".

$$\mho \approx \mho$$

$$\eta\, x \approx \eta\, y \text{ if } x \sim_A y$$

$$\theta\, \tilde{x} \approx \theta\, \tilde{y} \text{ if } \triangleright_t (\tilde{x}_t \approx \tilde{y}_t)$$

$$\theta\, \tilde{x} \approx \mho \text{ if } \theta\, \tilde{x} = \delta^n(\mho) \text{ for some } n$$

$$\theta\, \tilde{x} \approx \eta\, y \text{ if } (\theta\, \tilde{x} = \delta^n(\eta\, x)) \text{ for some } n \text{ and } x : \langle A \rangle \text{ such that } x \sim_A y$$

$$\mho \approx \theta\, \tilde{y} \text{ if } \theta\, \tilde{y} = \delta^n(\mho) \text{ for some } n$$

$$\eta\, x \approx \theta\, \tilde{y} \text{ if } (\theta\, \tilde{y} = \delta^n(\eta\, y)) \text{ for some } n \text{ and } y : \langle A \rangle \text{ such that } x \sim_A y$$

EP Pairs

We will model casts as *EP-pairs*.

Given predomains A and B, an EP-pair $c : A \rightsquigarrow B$ consists of $\mathrm{emb}_c(\cdot) \colon A \to B$ and $\mathrm{proj}_c(\cdot) \colon B \to L_{\mho} A$, and a monotone relation R_c between A and B.

The relation R_c should be related in a specific way to the embedding and projection functions.

EP Pairs

We have an identity EP-pair $\mathrm{id} : A \rightsquigarrow A$, with the embedding and projection equal to the identity and η, respectively.

Recall: $D \cong \mathbb{N} + \triangleright (D \to_m D)$

We have an EP-pair $\mathrm{Inj}_{\mathbb{N}}$, where the embedding is just inl and Projection checks if the value of type D is a nat and returns it, otherwise returns $\mho$.

EP Pairs

We have an EP-pair $\mathsf{Inj}_{\to} : (D \to L_{\mho} D) \rightsquigarrow D$. The embedding delays the function and injects into the sum type of D: $e(f) = \mathsf{inl}(\mathsf{next} f)$ The projection does case analysis on the value of type D, and if it is a nat, returns $\mho$, otherwise, it it's a delayed function $\tilde{f}$, it returns

$$\theta_t(\eta(\tilde{f}_t)).$$

For EP pairs $c_i : A_i \rightsquigarrow B_i$ and $c_o : A_o \rightsquigarrow B_o$ we have the EP-pair $c_i \Rightarrow c_o : (A_i \to_m A_o) \rightsquigarrow (B_i \to_m B_o)$.
The embedding and projection are defined functorially via the embeddings and projections of the domain and codomain.

EP Pairs: Semantics

We would like the semantic analogues of the cast rules to hold, e.g.,

$$\frac{c : A \leadsto B \qquad M : \langle B \rangle}{\mathrm{proj}_c(M) \quad L(R) \quad M}\ \mathrm{DnL}$$

Unfortunately, this does not hold, because the projection function for $\mathrm{Inj}_{\to}$ introduces a θ, and so the LHS and RHS are not in lock-step!

This problem leaks into the embedding functions as well via functoriality in the $c_i \Rightarrow c_o$ case.

Wait functions

To remedy this, we associate to each EP pair four "wait" functions that mirror the structure of the embedding and projection functions for their EP-pair.

$$
\begin{aligned}
w_l^e &: A \to_m A \\
w_r^e &: A \to_m A \\
w_l^p &: A \to_m L_{\mho} A \\
w_r^p &: A \to_m L_{\mho} A
\end{aligned}
$$

Each wait function appears in one of the four semantic analogues of the cast rules, i.e., the rule above becomes

$$
\frac{c : A \rightsquigarrow B \qquad M : \langle B \rangle}{\mathrm{proj}_c(M) \quad L(R) \quad w_r^p(c)(M)}\ \mathrm{DnL}
$$

Main Theorem

Theorem (Graduality at Base Type)

If $\cdot \vdash M_e \sqsubseteq_e N_e : Nat$*, then*

1. *If* $N_e = \mho$*, then* $M_e = \mho$
2. *If* $N_e = V$*, then* $M_e = \mho$ *or* $M_e = V$*, where* $V = zro$ *or* $V = suc\, V'$
3. *If* $M_e = V$*, then* $N_e = V$

Extensional Collapse

We define a "collapse" function $\lfloor \cdot \rfloor : \text{Int-}\lambda C \to \text{Ext-}\lambda C$ that "forgets" about the intensional delay information, i.e., all occurrences of θ_s are erased.

Every term M_e in Ext-λC will have a corresponding program M_i in Int-λC such that $\lfloor M_i \rfloor = M_e$.

Moreover, we will show that if $M_e \sqsubseteq_e M_e'$ in the extensional theory, then there exists terms M_i and M_i' such that $\lfloor M_i \rfloor = M_e$, $\lfloor M_i' \rfloor = M_e'$ and $M_i \sqsubseteq_i M_i'$ in the intensional theory.

The Current Picture

$$\begin{tikzcd}
\text{Int-}\lambda C \arrow[r, "\lfloor \cdot \rfloor"] \arrow[d, "{[\![\cdot]\!]^i}"'] & \text{Ext-}\lambda C \arrow[d, "??"] \\
\text{Int. Sem.} \arrow[r, "\text{collapse}"'] & \text{Ext. Sem.}
\end{tikzcd}$$

Benefits and Drawbacks

Positives:

- SGDT handles much of the tedious step-index reasoning
- Clarifies the underlying semantic and algebraic structure

Drawbacks:

- Intensional semantics is much more complicated (needed to introduce wait functions)
- Still need to work “analytically” with monotone functions
- Need to do a lot of manual “unfolding” of fixpoint definitions in Guarded Cubical Agda

References I

[1] Robert Atkey and Conor McBride.
Productive coprogramming with guarded recursion.
ACM SIGPLAN Notices 48, 9 (2013), 197-208.

[2] Lars Birkedal, Rasmus Ejlers Møgelberg, Jan Schwinghammer, and Kristian Støvring.
First steps in synthetic guarded domain theory: step-indexing in the topos of trees.
Logical Methods in Computer Science 8, 4 (2012).

[3] Rasmus Ejlers Møgelberg and Niccolò Veltri.
Bisimulation as path type for guarded recursive types.
Proc. ACM Program. Lang. 3, POPL Article 4 (January 2019)

[4] Rasmus E Møgelberg and Marco Paviotti.
Denotational semantics of recursive types in synthetic guarded domain theory.
Mathematical Structures in Computer Science 29, 3 (2019), 465-510.

References II

[5] Max S. New and Amal Ahmed.
Graduality from Embedding-Projection Pairs.
MProc. ACM Program. Lang. 2, ICFP, Article 73 (September 2018), 30 pages.

[6] Max S. New, Daniel R. Licata, and Amal Ahmed.
Gradual type theory.
Proc. ACM Program. Lang. 3, POPL, Article 15 (January 2019), 31 pages.

[7] Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, and John Tang Boyland
Refined Criteria for Gradual Typing
1st Summit on Advances in Programming Languages (SNAPL 2015).