A Guarded Syntactic Model of Gradual Dependent Types

Joey Eremondi

April 18, 2022

Overview

Two Problems - One Solution

Two Problems

- Implementing Gradual Dependent Types
- Denotational Semantics + Metatheory

One Soution

- Approximate Normalization + Translation to Static Dependent Types

Implementing Gradual Dependent Types

Don’t Reinvent the Wheel

Long-term goal: gradual types in a full-scale dependent language

Machinery

- Many parts to dependent type checking and compilation
  - Compile-time evaluation for comparisons
  - Unification/inference
  - Code generation/optimization
- Want to avoid re-implementing as much of this as possible

Efficiency

- Normalizing/comparing types is expensive
- Want to leverage existing techniques
  - E.g. Idris: experimental normalization by compilation to ChezScheme

Proposed Approach

Compile gradual dependent types to static dependent types *without changing the static core language*

- Can use existing normalization, unification, etc.

Challenge: Effects in gradual language

- Gradual languages: two effects
    - Failure - just model as special value in gradual language
    - Non-termination
- Dependent languages restricted in how non-termination is used
    - Ensures consistency and decidablity of type checking

The Idris model of Non-termination

Definitions marked as "partial"

- Are not checked for termination/productivity
- Allows
  - General recursion
  - Non-positive datatypes

At compile-time

- Can hide implementations so partial functions never normalized
- Conservative: some equivalent partial-terms may rejected as non-equal
  - need to normalize to *see* that are equal
- Ensures type-checking terminates

Problem with the Idris model

Ad-hoc, hard to reason about

- e.g. it's hard to prove that every gradual program's translation is well-typed
  - Need to reason about internal details of normalizer
- Want formalism of Idris-style non-termination, so can prove translation proves

We will use Guarded Type Theory as a theoretical model of Idris

- Lets us formalize the notion of "this value not normalized at compile time"
- More on this later

Translating with Approximate Normalization

How can we prevent non-termination during compile time normalization?

- Separate semantics for compile-time and run-time normalization
- Only difference: casts $\langle \mathbf{?} \Leftarrow (\mathbf{?} \to \mathbf{?}) \rangle \mathbf{f}$
  - Approx: reduces to $\lambda \mathbf{x}.\, \mathbf{f?}$

Translation:

- Model approx. **?** as strictly-positive $Unk = (1 \to Unk) + (Unk \times Unk) \ldots$
- Full translation produces pairs of approximate and exact
- Type computations only use approximate part

The Other Side: Denotational Semantics

Broad Motivation

Do gradual dependent types mean anything? Do they make sense?

What kind of reasoning principles hold for gradual dependent types?

What kind of guarantees can we give the programmer?

Why a syntactic model?

Want to prove that approximate normalization is terminating

- GDTL approach doesn't scale to inductives

Want to prove the GGs for Approx. Normalization

- GDTL Approach to errors was wrong
- GCIC approach simulation-based, complex
  - Even more complex when add approximation

Prove richer metatheory

Often need some sort of logical relation

If syntactic model is in consistent calculus, then can prove these things in the target theory itself (unlike $GCIC^{\mathcal{G}}$)

Theorems that show that gradual dependent types behave as expected

- EP-Pairs, or a version of them
  - Not needlessly producing ?
- Weak canonicity
  - Nothing gets stuck from gradual types
- Preservation of static propositional equalities
  - i.e. equal static values are equal in the model
  - Weaker version of full-abstraction

The Model

Model Approximate Normalization in Type Theory (MLTT?)

- Proves that all terms halt
- Decidabel type-checking

Model exact execution in Guarded Type Theory

- Consistent logic for describing (potentially) non-terminating terms
- Gives non-positive datatypes, can model **?** exactly

Then can prove things about the language using the model

Guarded Type Theory

Introduces:

- A "later" modality $\triangleright : \mathit{Type} \to \mathit{Type}$
- Operators $\mathit{next} : A \to \triangleright A$ and $\mathit{app} : \triangleright(A \to B) \to \triangleright A \to \triangleright B$
    - Arbitrary *guarded* fixed-points:
        - $\mathit{fix} : (\triangleright A \to A) \to A$
        - $\mathit{lob} : \mathit{fix}\, f = f\,(\mathit{next}\,(\mathit{fix}\, f))$ (but not definitionally)
    - Type lifter $\blacktriangleright : \triangleright \mathit{Type} \to \mathit{Type}$
    - Can be used to make a "lifting monad" $L\ A = A + \triangleright(L\ A)$

Gives us:

- Non-positive inductive datatypes
- General recursion, but only behind modality

Consistent: model in Topos of Trees

- Whatever that means

A Model in Guarded Type Theory

Universe à la Tarski

- Data-type of "codes" $\mathbb{C}_\ell : Type$
- "Elements-of" interpretations
  - $El_{approx} : \mathbb{C}_\ell \to Type$
  - $El_{exact} : \mathbb{C}_\ell \to Type$

Syntactic Model

1. Type semantics $\mathcal{T}[\![\mathbf{T}]\!] : \mathbb{C}_\ell$
2. Expression semantics: if $t : T$ then
   $\mathcal{E}[\![\mathbf{t}]\!] : (El_{approx}\, \mathcal{T}[\![\mathbf{T}]\!]) \times (L\, (El_{exact}\, \mathcal{T}[\![\mathbf{T}]\!]))$

Model of the unknown type

GTT allows for exact definition:

- $Unk = \mathit{fix}\,(\lambda(x : \triangleright Type).\,(Unk \times Unk) + (\blacktriangleright X \to Unk) + \ldots)$

Interpretation of casts must be guarded

- i.e. $cast : (c_1\ c_2 : \mathbb{C}_\ell) \to El_{exact}\ c_1 \to L\ (El_{exact}\ c_2)$

Mapping GTT to Idris

Straightforward mapping of GTT to partial Idirs

- fix becomes general recursion
- Guarded non-positive types just turn into partial non-positive types

$\mathit{fix}\, f = f\, (\mathit{next}\, (\mathit{fix}\, f))$ is *not definitional* in GTT:

- Know that type derivation never relies on normalizing non-terminating functions
- So neither does Idris typing