

Contributions of this paper for JFP Audience:

The axiomatic semantics of gradual type theory allows for a type theory focused on preserving type-based reasoning to be used as a framework for discussing *how* casts should behave if we care about such reasoning principles. The primary contribution is showing that the cast semantics forces certain design principles for casts with certain type constructors. For example, the "lazy" wrapping semantics for function type casts is forced if we care about the eta rule for function types.

The specification

for other casts however can be left open to different models where for example in Scheme-like GTT a cast between a pair of a boolean and an element of some type A , can be successfully cast to a sum type, $A+A$. In a certain sense this reminds me of how in languages like Reticulated Python, we may allow casts between different iterable types because of common functionality over the types (i.e. you can index into them).

One of the primary contributions is showing that gradual type theory by construction honors graduality, while honoring eta-rule based optimizations. This is in contrast to previous work on constructing gradual semantics for statically typed languages (like AGT and Cimini's work on automating the construction of the gradual type system and dynamic semantics), which break certain eta-rules, even if graduality is enforced by construction.

Strengths of the revision:

In general, I found the increased amount of exposition to be helpful across many sections. I think that the revision of the introduction to include different examples, (in particular eager vs. lazy product casts) to be enlightening.

I found Section 2 readable to begin with, but I appreciate the greater detail on how CBV and CBN are embedded in CBPV. There are some slight typos with the list of constructors in CBN. Similarly, I found the increased discussion of eta-equivalences for different type constructors aided in building intuition, although I think that the initial paragraph added on Beta rules may have been unnecessary.

In Section 3 I found that the increased exposition helped with certain parts, especially up until Section 3.5. The proofs for Lemma 3.5 and Lemma 3.6 are cleaner in appearance, although in some places there are some omissions of x' . The connection to the discussion of type constructors flows better after Theorem 3.6.

I'm not sure if it's just due to having read the paper once already, but I found the Section 3.6 about how the existence of a most-precise type to be easier to digest than in my initial review (though there were some sloppy mistakes such as broken references).

The updated related work adds discussion of optimization, which is related to the use of eta-rules for enabling program transformations.

Comments and concerns:

In the discussion of optimization,

I would have liked more discussion about how cast semantics that may be models of GTT (like threesomes) could allow for systems with efficient implementations of these models to exploit transformations in statically type code that may make a mixed-type configuration of a program more performant than the dynamically typed version.

Even just a sentence indicating on how other lines of research on the performance

optimization on casts are orthogonal and can be applied so long as they respect the design of GTT would be nice.

It would be great

if systems like Grift (that may be implementations of models of GTT) could allow for greater performance gains in mixed type configurations of programs over the dynamically typed configuration, since usually this isn't the case.

The initial presentation of the main graduality theorem could use some explanation (Theorem 5.8). On first appearance, some of the syntactic features of the judgment in the conclusion of the rules are a bit confusing.

Why is the explicit downcast on the right hand side of the precision relation necessary? Why is the explicit upcast on the left hand side of the precision relation in the latter rule necessary?

This is a bit confusing to read, especially

since the original statement of the gradual guarantee in previous work used a straightforward premise that terms were related by term precision. The proof of the gradual guarantee

later used rules for handling extra casts on the left or right expressions related by precision.

Is the statement of the theorem in this work

just related to the issue of homogeneous

term inequalities in CBPV*? At first glance homogeneous inequalities seem quite restrictive in that you couldn't relate two lambda expressions with different type annotations, but I assume that this is somehow taken care of by the translation into CBPV*. If the answer to these questions should be obvious, then I had trouble discerning it while reading and other readers may face the same issue.

There are various typos throughout the paper that appeared due to some modifications to syntax or added exposition. I list these typos and other small gripes below:

Section 1:

> "We argue that existing gradual type soundness theorems are only indirectly expressing..."

This seems like a stretch. In various common gradually typed settings, our programs may not crash if we remove type annotations (think operator overloading). I guess in that sense, the restriction enables us to rewrite the program in ways that would be unsound if the annotations were removed or optional typing was used. But the point seems to be more about *preventing unwanted behaviors* than enabling the exploitation of certain equivalences.

> Page 5: "The graduality principle states that..."

In practice, things like tagged values being encoded as objects can actually break graduality when used with certain dynamic language features like isinstance, unless handled carefully.

I thought that the scheme-like extension in Section 5 was nice in that a type-case expression worked with upcast values. Maybe make a connection to that development here.

> Page 6: "return 5, returning 5, ..."

Change these to "return true" and "returning true".

> Page 7: "So this example shows us that (1) ..."

What would happen if a call by value language used a lazy cast semantics on arguments in the absence of effects other than termination with an error?

> Page 9: Introduction of the lazy product

Here the lazy product is brought up to discuss eta laws for CBN features. But one thing essential to the work is the use of lazy products to define the dynamic computation type's interpretation. If there were some other CBN feature introduced into GTT to talk about embedding CBN, then would we be unable to satisfactorily determine the dynamic computation type's interpretation?

Section 2:

> Page 14: definition of complex values

Was there previous work on complex values, or is this a novel construction?
If there's previous work or some analogue, then cite it here.

> Page 15: trailing paren after "pure."

> Page 16: The stack with two holes in a product example.

Is the hole only used once because the laziness of the product means that a projection will choose which of the two subexpressions using the hole to actually run? A little bit more detail or an example would make this clearer.

> Page 17: For CBN with: 1, &, +, 0, ->, 1, &

Delete the latter 1 and &

> Page 18: "Eis"

Add a space after the E.

> Page 20: Figure 4

Add an underline under the B's in UMon

> Page 22: "where the type on the right is less precise than the type on the right"

Change the latter "right" to "left".

=====

Section 3:

> Theorem 3.2

Why does the last case assert precision while the others assert type?

> Page 30 "that all type constructors are monotone..."

"that" => "That"

> Page 31: On precision relation after "so transitivity gives the result"

Add an x' to the right of the cast on the middle part of the relation.

> Page 31: On precision relation after "identity extension gives the result"

Add an x' to the right of the cast on the middle part of the relation.

> Page 33: "The stipulation that some cast with the correct universal..."

Does this statement mean that implementations are free to choose different implementations of how to factor casts for whatever types they might add to GTT? Typically a ground type where a ? is added to each argument to a type constructor is used as the type to factor through.

> Page 37: "Given these two Theorems , , "

Fix the broken ref.

> Page 37: absurd z

Since the "absurd" term first appears here, maybe state exactly how this should differ from an error.

=====

Section 4:

> Page 39: "translation of any CBV type.."

Delete the extra period.

> Figure 11:

Add an underline to the F in the translation of casts.

> Page 41: "However, there is... However, we can..."

Try to rewrite to eliminate the repetitive use of "However,".

> Page 41: "bindXtoYinZ"

Fix the macro use here.

=====

Section 5:

> Page 42: "We translate GTT into a statically typed CBPV* language where the casts of GTT are translated to "contracts" in GTT"

Change the latter "GTT" to "CBPV*".

> Page 42: "CBPV , (defined in Section 6)"

Eliminate the space after CBPV.

> Page 44: Definition 5.1 part 2.

Replace the substitution for x with a substitution for z.

=====

Section 7:

Page 63: Figure 19

Why are there different arrows in the premise of the latter rule?

Page 67: Lemma 7.7 proof

In 2. Is the "ki" index intentional?

=====

Section 8:

> Page 73 "it means that code:"

What was meant to be written after code?

> Future work about verifying a cast on list types as mapping a cast.

Verifying cast behaviors for specific inductive types like immutable lists seems less difficult then verifying inductive types in general. Do certain mutable collections similar to lists pose any difficulties?

Currently, mutable vectors require a semantics similar to lazy products where getting and setting memory locations at specific indexes requires a cast each time. Are there any plans to investigate references using GTT?

