# Implementing the GTLC via Metaprogramming

By Andre Kuhlenschmidt

The semantics of the gradually typed lambda calculus (GTLC) are usually specified as a translation from the GTLC, which contains implicit casts, to a wholly explicit language. Since the definition is presented as syntax rewriting, I couldn’t help but implement the calculus in a syntax rewriting system. Therefore, my first project implements the GTLC in the Racket module system via the #lang interface and syntax macro rewriting. This implementation proves to be straight forward once implemented, but it is difficult to reason about the correctness of the implementation. I attempt to counter this difficulty in my second project where I implement the GTLC as an embedded language in a dependently typed programming language.

In each implementation I strive for a minimal implementation of the core typing rules and semantics. The language forms available are lambdas with typed or untyped arguments, application, variables, integers, booleans, if, and a few operators. While this is a minimal core it has been more than enough to see the beauty and beastliness of each implementation strategy.

The Racket Module System Implementation

Using the Racket module system to implement new languages is straightforward once you understand what hooks it provides and their correct uses. There is documentation for these hooks, but figuring out what to use without reading the reference front to back seems a little harder than it should be. For this project I spent roughly half of my time figuring out how to use the facilities available to achieve what I needed. Though once figured out the language forms presented few sources of future trouble.

The basic pattern of implementing a new language in racket is to reimplement the Racket core forms in terms of the originals so that new behavior may be attained. The largest hook in this direction is the #%module-begin form. This form gives the implementer the ability to change the behavior of modules using the new language as a whole.

In order to facilitate the reimplementation, Racket provides a sophisticated phasing mechanism that allows precise control of when and where identifiers are bound during macro expansion. Not having used phase levels before, I often found myself introducing identifiers at the wrong phase. Eventually I discovered that the Racket macro stepper has support for inspecting the bindings of identifiers at different levels. The phase level model is complex enough that, to decide what phase level I needed sometime I had to just guess and check with the stepper.

The precision of phase levels also leads to quite a few decisions about the design of syntax transforming programs and the structuring of code. It seems like the there are now two levers that you can pull in order to “go meta”. You can write something in a different module with a define and import them at a different phase level, or use one of the syntactic forms to shift their definition in module. This leads to a lot of refactoring when you realize that you need forms at different levels at the same time. This nuisance might be easily relieved by some mechanism of adhoc changes to an identifier’s phase level.

The main advantage of the Racket #lang implementation is that much of the structure has already been laid for creating new languages. Furthermore it seems that Racket’s module system and macro facilities are more loosely tied to the language than I would have suspected. One possible extension to this implementation of the GTLC is to use a different backend, besides the Racket compiler, for implementing the core forms of the GTLC language. I think that the racket module system has enough flexibility that this could be possible with only a few changes, and doing so would grant the user much of the tooling available to racketeers today.

Ultimately the main disadvantage of the Racket #lang implementation is that there are no strong guarantees that the implementation will actually produce code that is type correct. Even a trivial example, such as this project, went through numerous iterations due to bugs in the expanded code after figuring out how to use the metaprogramming facilities.

The Idris Interpreter Implementation

To counter the lack of type correctness in expanded code, I attempted a reimplementation in the programming language Idris. While Idris has some syntax extension ability, it has nothing as powerful as Racket’s module system. So ultimately the implementation technique that I used was to write an interpreter for the language via the methods presented in the Idris documentation. In this implementation, the GTLC source syntactic forms are mostly implemented as smart constructors that inspect inferred type information and create the explicitly typed core forms which are then interpreted.

The chief advantage of this implementation is that all GTLC Terms that typecheck are translated to code that is well typed in Idris. Because the core forms are explicitly typed, it is possible to write an interpreter that meets the specification of the cast calculus language and get strong guarantees that that language is correct. Further knowledge of dependent types might even allow stronger guarantees. Since Idris has dependent types it should be possible to prove that the translation to the cast calculus and subsequent execution live up to the criteria of gradually typed languages.

But all of these guarantees come at a cost. Programming in dependent types is far harder than I imagined. Type checking in dependently typed programming languages is necessarily conservative. It seemed that at every step of the way I was trying to decide whether I had written bad code or come up against one of the limitations of type checking. When I was convinced that I had correctly written something that is type sound, it became difficult to rethink the implementation into a form that the type checker could typecheck. Many of these problems have been resolved by factoring most of the complexity out of type-level computations, but there is still a logical hole in the program. While the definition of rtCasts itself typechecks, I have not been able to integrate it into the interpreter where the meta-variable ICast has been inserted to allow further progress.

Furthermore I have had only mild success at deriving proofs of type consistency. These proofs allow type information to be elided from terms in the syntax of the embedded gradually typed programs. They also allow consistent but unequal terms to be casted to the correct type. I have tried to use tactics to automatically search for these proof obligations, but the type elaborator does not always find the consistency relationship. I think that a custom tactic might be to achieve more consistent behavior.

Thoughts On Both Systems

Each of systems have very powerful facilities for metaprogramming. The Racket macro system has extraordinary flexibility to change the way that programs behave by translation of syntax and alteration of module interactions. The Idris programming language has a type system that is flexible enough to encode many complex and interesting types.

One of the really interesting bits that I discovered was that the two systems are not really all that different. Both Racket’s expander and Idris’s type elaborator convert the syntax of the programs to some normal form by incrementally rewriting language forms. Idris takes this a step further and type checks programs using syntactic equality of type level computation.

While I was coding I kept wondering if either system would benefit from the other’s knowledge. Perhaps syntax case would make for a wonderful and clear implementation of the dependent type system, or perhaps very expressive types could explain many of the behaviors of macro systems. Either way I could only imagine how painful debugging both macros and dependent types would be. Or would it? In some ways the Racket macro stepper is the same sort of tool that you would want in order to understand how type elaboration is failing.