Expressive and Strongly Type-Safe Code Generation

Thomas Winant, Jesper Cockx, and Dominique Devriese

imec-DistriNet, KU Leuven
firstname.lastname@cs.kuleuven.be

Meta-programs are useful to avoid writing boilerplate code, for polytypic programming, etc. However, when a meta-program passes the type checker, it does not necessarily mean that the programs it generates will be free of type errors, only that generating object programs will proceed without type errors. For instance, this well-typed Template Haskell [5] meta-program generates the ill-typed object program *not* `'X'`.

```
notX :: Q Exp
notX = [| not 'X' |]
```

Fortunately, Template Haskell will type-check the generated program after generation, and detect the type error. We call such meta-programming systems *weakly type-safe*. Even though weakly type-safe meta-programming suffices for guaranteeing that the resulting program is type-safe, it has important downsides. Type errors in the generated code are presented to the application developer, who may simply be a user of the meta-program. If the meta-program is part of a library, the developer has little recourse other than contacting the meta-program authors about the bug in their code. Moreover, composing weakly type-safe meta-programs can be brittle because the type of generated programs is not specified in a machine-checked way.

We are interested in what we call *strongly type-safe* meta-programming, which offers a stronger guarantee: when a meta-program is strongly type-safe, all generated programs are guaranteed to be type-safe too. As such, bugs in meta-programs are detected as early as possible. In fact, all arguments in favour of static typing can be made.

Existing strongly type-safe meta-programming systems like MetaML [6], Typed Template Haskell [3], and Scala's reflection API [1] offer this guarantee by providing *typed quotations*, i.e. quotations that are type-checked at meta-program compile-time. For example, when the faulty meta-program is rewritten using a typed quotation, the bug is detected at meta-program compile time.

```
notX :: Q (TExp Bool)
notX = [|| not 'X' ||]
```

Unfortunately, to offer this guarantee, these and other systems compromise on expressiveness [6, 2, 4]. In particular, while typed object expressions can be constructed using typed quotations, their *types* and *typing contexts* cannot. These are severe restrictions that make it impossible to develop strongly type-safe variants of many common weakly type-safe meta-programs.

A real world example that suffers from this restriction is the generation of *lenses* [7] for a record data-type. Such a meta-program is not expressible using state-of-the-art strongly type-safe meta-programming systems. To illustrate this restriction, consider a simplified sketch of what the Typed Template Haskell variant of *deriveLenses* would look like:

```
deriveLenses adt = map (λfield → deriveLens adt field) (fields adt)
deriveLens :: ADT → Field → Q (TExp (Lens ? ?))
deriveLens adt field = ...
```

The question marks should be replaced with the type of the record data type and the type of the field. Clearly, the type of the generated lens *depends* on the types of the record data type's

fields, which are only available at the value level in the meta-program. If Haskell had *dependent types*, one could write this signature:

$$deriveLens :: (adt :: ADT) \to (f :: Field\ adt) \to Q\ (TExp\ (LensType\ adt\ f))$$

Where *LensType* is a type-level function that calculates the type of the lens. Such a syntactic construction of the type of an object program is fundamentally impossible in Typed Template Haskell and other MetaML-like systems. Deeper inside the implementation of *deriveLens*, it gets worse as the types of generated expressions depend in more complex ways on the values *adt* and f, and they are also constructed in *contexts* that depend on them. The underlying reason for this limited expressiveness is that the meta-level type system of these systems is not powerful enough to express the naturally dependent types of many strongly type-safe meta-programs.

We propose a new design that delivers strong type-safety without compromising on expressiveness. Our first key design choice is to represent object programs by an inductive type family in an off-the-shelf dependently-typed language (Agda). This type family is indexed by the type of the program, and its type and variable contexts. Each of its constructors encodes one language construct, including its corresponding typing rule. Using this encoding, meta-programs construct object programs that are correct by construction. This approach is standard in dependently typed languages, yet it isn't commonly used by existing meta-programming systems.

Our second key design choice is to use a small explicitly-typed core language as the object language (in our case GHC Core), instead of the full surface language (which would be Haskell). This choice is based on the observation that surface languages are designed for programmers, not meta-programs. Their complex syntax, typing rules, type inference, and tendency to change make them ill-suited as object languages. In contrast, a core language such as GHC Core is designed to be used by the compiler. As a consequence, it is typically well-studied, small, explicitly typed, relatively stable, and has a full formal description, while remaining relatively close to the surface language. Using a core language instead of the full surface language is not an academic simplification, but a feature of our approach: it is a central design choice that we believe is essential to make our approach realistic to implement and use.

Our approach to strongly type-safe metaprogramming is based on existing technology (an off-the-shelf dependently-typed language and a standard encoding of the object language) but applies it in a new way. We have implemented is as a proof of concept for Haskell. Using our implementation, we developed strongly type-safe variants of existing real-world meta-programs: deriving lenses and deriving the `Eq` type class. In a fair comparison with the original meta-programs, our meta-programs count roughly the same number of SLOC. This shows that our approach is practical as well as simple, expressive, and strongly type-safe.

References

[1] Eugene Burmako. *Unification of Compile-Time and Runtime Metaprogramming in Scala.* PhD thesis, EPFL, 2017.

[2] Chiyan Chen and Hongwei Xi. Meta-programming through typeful code representation. In *ICFP*. ACM, 2003.

[3] Geoffrey Mainland. Type-safe runtime code generation with (Typed) Template Haskell. https://www.cs.drexel.edu/~mainland/2013/05/31/type-safe-runtime-code-generation-with-typed-template-haskell/.

[4] Geoffrey Mainland. Explicitly heterogeneous metaprogramming with MetaHaskell. In *ICFP*. ACM, 2012.

[5] Tim Sheard and Simon Peyton Jones. Template meta-programming for Haskell. *SIGPLAN Not.*, 37(12), December 2002.

[6] Walid Taha and Tim Sheard. Multi-stage programming with explicit annotations. *SIGPLAN Not.*, 32(12), December 1997.

[7] Twan van Laarhoven. CPS based functional references. http://twanvl.nl/blog/haskell/cps-functional-references, 2009.