Polarized Lambda-Calculus at Runtime, Dependent Types at Compile Time

András Kovács

University of Gothenburg Gothenburg, Sweden
andrask@chalmers.se

We describe a particular two-level type theory [3]:

- The object level is a polarized simply-typed calculus, with computation types (functions, computational products) and value types (inductive types, closures).
- The meta level is a standard dependent type theory.

This supports two-stage compilation, where we can use dependent types in the surface language, but only object-level constructions remain after staging. It appears to be an excellent setting for performance-focused staged programming.

The polarization lets us control closures and function arities in a fairly lightweight way. In particular, if we do not use the explicit closure type former, then all function calls can be compiled to calls and jumps to statically known code locations. Hence, we migth ask: how much can we reproduce from the abstraction tools of functional programming, without using any runtime closures? Perhaps surprisingly, closures are rarely used in an essential way.

- A *map* function for lists is meant to be inlined, and after inlining no runtime closures remain.
- Monadic binding in Haskell is a higher-order function. Without compiler optimizations, we get a big overhead from a deluge of runtime closures; but with optimizations we expect that no closures remain.

In the current work, we start to build up libraries in the mentioned two-level theory, aiming to minimize the usage of runtime closures and eliminate abstraction overheads. We want to shift work from general-purpose optimizing compilers to deterministic & extensible metaprogramming. For example, instead of expecting GHC to sufficiently inline our monadic code, we implement efficient monadic code generation ourselves.

Staged Monad Transformers

Despite recent competition from a variety of algebraic effect systems, monads and monad transformers remain the most widely used strongly-typed effect system, so it make sense for us to develop their staged flavor.

In our approach, we *don't use monads at all in the object language.* Instead, we use meta-level monads, and only convert down to the object level when runtime control dependencies force us to do so. Generally, the staged version of a monad is obtained by extending it with *code generation as an effect.* The code generation monad is the following:

$$\textbf{newtype}\ \mathsf{Gen} : \mathsf{MetaTy} \to \mathsf{MetaTy}$$
$$\textbf{newtype}\ \mathsf{Gen}\ A = \mathsf{Gen}\ (\{R : \mathsf{ObjTy}\} \to (A \to \Uparrow R) \to \Uparrow R)$$

Here, ObjTy is the universe of object-level types, MetaTy is a meta-level universe, and $\Uparrow R$ is (intuitively) the type of metaprograms which generate object expressions of type R.

This is a continuation monad where *eventually* we have to return an object-level value, so we can only "run" actions of type $\mathsf{Gen}\,(\Uparrow A)$ to extract a result with type $\Uparrow A$. However, while working in the monad, we are free to use both object-level and meta-level constructions, and introduce object-level let-binders. In general, for some monad transformer stack M, we obtain its staged version by putting a Gen at the bottom of the stack. For example, $\mathsf{State}\,(\Uparrow\mathsf{Int})$ becomes $\mathsf{StateT}\,(\Uparrow\mathsf{Int})\,\mathsf{Gen}$. In this monad in particular, we can modify an object expression in the state, and also generate object code via Gen.

Using our transformer library, we program in meta-monads, and insert conversions to and from object code at the points of dynamic dependencies (e.g. object-level function calls). Such conversions are defined in a compositional manner, by recursion on transformer stacks. Overall, this style of programming requires modest extra noise compared to Haskell, but it guarantees high-quality code output by staging.

Staged Stream Fusion

Stream fusion [1] is another application that we developed in this setting. The idea here is to give a convenient list-like interface for programming with meta-level state machines. Such machines can be turned into efficient object-level code as blocks of mutually recursive functions, without storing intermediate results in runtime lists. We demonstrate very concise solutions to two well-known problems.

First, we need to generate mutually recursive blocks. One part of the challenge is to have guaranteed well-scoping and well-typing, although the number (and types) of definitions in a mutual block is only known at staging time; see e.g. [4]. We also want to avoid any runtime overheads. We represent a mutual block as a single recursive definition with a computational product type. The polarization guarantees that this can be compiled without downstream overheads to an actual mutual block.

Second, fusion for arbitrary combinations of zipping and *concatMap* has been a long-standing challenge. Currently, the *strymonas* library [2] supports such fusion but its solution is much more complex than ours, and it also relies on mutable references in the object language. In contrast, we compile to pure object code, which also enables us to parameterize streams over arbitrary monads and embed monadic effects in stream definitions.

Our streams are tuples containing a type for an internal state, an initial state and a representation of state transitions. Moreover, the internal state is required to be a finite sum-of-products of object-level value types. Now, if such sums-of-products are closed under Σ-types, concatMap is easily definable by using a Σ-type of the family of internal states that we get from an $A \to \mathsf{Stream}\,B$ function. We construct such Σ-types from an internal *generativity axiom*, which expresses that certain metaprograms can't depend on object-level terms. This axiom does hold in the staging semantics and we can erase it during staging. From this, we construct *transient* Σ-types which end up as non-dependent product types after staging.

The ability to analyze object code has been often viewed as a desirable feature in metaprogramming. In contrast, we demonstrate a use-case for the explicit *lack* of intensional analysis, through our generativity axiom. This might be compared to *internal parametricity* statements in type theories.

References

[1] Duncan Coutts. *Stream fusion : practical shortcut fusion for coinductive sequence types*. PhD thesis, University of Oxford, UK, 2011.

[2] Tomoaki Kobayashi and Oleg Kiselyov. Complete stream fusion for software-defined radio. In Gabriele Keller and Meng Wang, editors, *Proceedings of the 2024 ACM SIGPLAN International Workshop on Partial Evaluation and Program Manipulation, PEPM 2024, London, UK, 16 January 2024*, pages 57–69. ACM, 2024.

[3] András Kovács. Staged compilation with two-level type theory. *Proc. ACM Program. Lang.*, 6(ICFP):540–569, 2022.

[4] Jeremy Yallop and Oleg Kiselyov. Generating mutually recursive definitions. In Manuel V. Hermenegildo and Atsushi Igarashi, editors, *Proceedings of the 2019 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM@POPL 2019, Cascais, Portugal, January 14-15, 2019*, pages 75–81. ACM, 2019.