Class-Dictionary Specialization with Rank-2 Polymorphic Functions

YONG QI FOO, National University of Singapore, Singapore
MICHAEL D. ADAMS, National University of Singapore, Singapore

Rank-2 polymorphism, when combined with type-class-constrained arguments, enables powerful abstractions and code reuse by allowing functions to accept arguments that are themselves ad-hoc polymorphic. Optimizing compilers like the Glasgow Haskell Compiler (GHC) use techniques like class-dictionary specialization and inlining for optimization. However, these techniques can falter when the rank-2 polymorphic functions are recursive. Specializing the polymorphic arguments of recursive rank-2 polymorphic function applications requires inlining the applied function to expose type and dictionary applications to the arguments, but the compiler's heuristic-driven inliner is reluctant to inline recursive functions, risking non-termination during compilation. This stalemate causes recursive rank-2 polymorphic functions to remain un-optimized and be left with a runtime penalty. Within the Haskell ecosystem, we identify this stalemate within three widely used libraries: the Scrap Your Boilerplate (SYB) and SYB With Class (SYB3) generic-programming libraries, and a fragment of the `lens` optics library. In these libraries, the combination of rank-2 polymorphism, type-class constraints and recursion is central to their implementation.

In this paper, we present a new optimization technique that breaks this stalemate and enables class-dictionary specialization with recursive rank-2 polymorphic functions. We introduce a partial evaluator that strategically applies the standard transformations of inlining, β-reduction and memoization to applications of rank-2 polymorphic functions that are *partially static*, i.e., whose arguments have some information known statically. This process exposes applications of its polymorphic arguments onto concrete types and dictionaries, which can be specialized by the standard compiler optimization pipeline. Additionally, we introduce type-constant folding to evaluate run-time type-equality tests statically, further leveraging static type information gained from partial evaluation. We implement our technique as a GHC plugin and demonstrate its effectiveness by resolving the performance bottlenecks in the aforementioned Haskell libraries. On SYB and SYB3 traversals, our technique achieves speedups of 43× on average (up to 155×) and 6.1× on average (up to 9.5×), respectively, matching the performance of their hand-written counterparts. On the fragment of the `lens` library containing the identified slowdowns, our technique achieves speedups of 1.6× on average (up to 2.1×).

CCS Concepts: • **Software and its engineering** → *General programming languages*; **Polymorphism**; **Software performance**; *Reusability*; • **Theory of computation** → *Program analysis*.

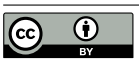
Additional Key Words and Phrases: partial evaluation, generic programming, scrap your boilerplate, scrap your boilerplate with class, rank-2 polymorphism

ACM Reference Format:
Yong Qi Foo and Michael D. Adams. 2026. Class-Dictionary Specialization with Rank-2 Polymorphic Functions. *Proc. ACM Program. Lang.* 10, OOPSLA1, Article 119 (April 2026), 28 pages. https://doi.org/10.1145/3798227

1 Introduction

The tension between abstraction and performance is a perennial challenge in language and compiler design. High-level abstractions promote code reuse, modularity and concision, but can introduce

Authors' Contact Information: Yong Qi Foo, National University of Singapore, Singapore, Singapore, yongqi@nus.edu.sg; Michael D. Adams, National University of Singapore, Singapore, Singapore, https://michaeldadams.org/.



ACM 2475-1421/2026/4-ART119
https://doi.org/10.1145/3798227

run-time overhead that is difficult for compilers to eliminate. *Rank-2 polymorphism* [36] is a prime example of such an abstraction. (Rank-2 polymorphic functions are functions that receive polymorphic arguments.) Coupled with type classes, rank-2 polymorphism enables expressive designs such as the *Scrap Your Boilerplate* (SYB) library [33, 34], the primary motivation behind our work. SYB is one of Haskell's most widely used and enduring generic-programming systems. SYB defines rank-2 polymorphic combinators for writing generic traversals concisely, freeing developers from the drudgery of writing repetitive "boilerplate" code. Its conceptual simplicity, ease of use and strong support from the Glasgow Haskell Compiler (GHC) [17] led to its widespread adoption, with over 97% (over 16,000) of all packages on the Hackage archive [24] directly or indirectly depending on it (October 2025). A particular appeal of SYB is its improvements in ergonomics for writing traversals. One of our benchmarks (§5) uses 118 Source Lines of Code (SLoC) to implement by hand, whereas SYB allows us to implement it with just 4 SLoC.

Despite their conceptual elegance, recursive rank-2 polymorphic functions can be slow because standard optimization techniques can be ineffective in their presence. The root of the problem is an optimization "roadblock" inherent to the compilation of recursive rank-2 polymorphic functions. Compilers like GHC specialize (ad-hoc) polymorphic functions by creating type- and dictionary-specific instances, eliminating overhead caused by class-dictionary passing [26] and enabling further downstream optimizations. However, suppose the compiler is trying to optimize the expression $f\,x$ where x is (ad-hoc) polymorphic and f is rank-2 polymorphic and recursive. (1) To specialize x, the compiler must expose type and dictionary arguments to x, such as by inlining [28] f. However, (2) inlining f is precisely what the compiler refuses to do because f is recursive, and unrestricted inlining of recursive functions risks infinite loops during compilation and code bloat [28].

The result is a stalemate: specialization demands inlining, but inlining is avoided in (loop-breaking) recursive functions. This stalemate leaves the expensive abstractions, like class-dictionary passing, higher-order function calls and polymorphic dispatch, intact in the compiled code. For these reasons, SYB is one of the slowest generic-programming libraries with documented slowdowns of one to two orders of magnitude [1–3, 6, 7, 39, 53, 58, 60, 61], and attempts to coerce GHC to aggressively perform general-purpose optimizations on SYB traversals have proven largely ineffective [39]. Beyond SYB, we have also identified this optimization "roadblock" in the Scrap Your Boilerplate With Class (SYB3) generic-programming library, the main variant of SYB, and a fragment of the popular `lens` optics library [32]. (On the Hackage archive, `lens` is one of the most-downloaded packages with over 420,000 downloads in total. For comparison, SYB has been downloaded approximately 340,000 times (October 2025).)

Previous attempts to optimize rank-2 polymorphic libraries like SYB have treated it as a domain-specific problem, targeting library-specific patterns instead of rank-2 polymorphism generally. Adams et al. [2, 3] developed a technique that applies transformations to SYB traversals to eliminate what the authors call "undesirable types" that are used by SYB combinators. While effective, this approach is brittle and coupled to the specifics of the SYB library. As examples, merely changing the *names* of the type classes used by SYB causes this technique to fail, and this technique does not work with SYB variants like SYB3. Its reliance on HERMIT [13, 52] means it no longer runs on programs built using modern versions of GHC. (As of October 2025, the latest version of GHC is 9.12.2, while HERMIT is supported by GHC versions only up to 7.10.3.) Private communication with Adams et al. also reveals that the optimization itself is slow, increasing compilation times drastically (compiling a relatively small sized module with the optimization takes *multiple seconds*), leaving it impractical for use on larger programs and integration into GHC.

In this paper, we resolve the optimization "roadblock" for rank-2 polymorphic functions. We present a novel partial evaluator that strategically applies standard program transformations: inlining, β-reduction and memoization, targeting *partially static applications* of rank-2 polymorphic

functions, which are applications of rank-2 polymorphic functions to arguments with some information known statically. The resulting code instantiates applications of the (ad-hoc) polymorphic arguments of rank-2 polymorphic functions with concrete type and dictionary arguments, revealing opportunities for class-dictionary specialization. However, run-time type-equality tests are not optimized by the compiler, despite the partial evaluator instantiating type-equality tests with statically known types. Thus, we also introduce a new type-specialization pass that performs *type-constant folding* to statically evaluate type-equality tests. Together with downstream optimization passes, these can eliminate most, if not all, of the overhead that typically plagues such abstractions. Since our optimization techniques perform only standard program transformations, they are simple to implement and integrate into a compiler.

We demonstrate the efficacy and generality of this technique by applying it to all three aforementioned libraries impacted by this slowdown. Benchmarks show that on SYB and SYB3 traversals, our technique completely eliminates their performance overhead, and transforms them into code that is equivalent to their hand-written, non-generic counterparts. (In fact, GHC's Common Subexpression Elimination (CSE) pass sometimes merges them.) This results in speedups of 43× and 6.1× on average (up to 155× and 9.5×) for SYB and SYB3 traversals, respectively, making their performance indistinguishable from their hand-written equivalents. On the fragment of `lens` displaying the optimization-resistant pattern, benchmarks show speedups of 1.63× on average (up to 2.14×) for one of its modules. Our benchmarks also show that our naïvely implemented optimization plugin incurs a 2–4× compilation time overhead (though each benchmark can still be compiled within milliseconds). Better implementation and tighter integration of our optimizations into the GHC compiler pipeline can likely reduce this overhead further.

The rest of this paper is organized as follows. §2 provides background on the optimization stalemate inherent to recursive rank-2 polymorphic functions, using SYB as a concrete example. §3 describes our partial evaluator that targets partially static applications of rank-2 polymorphic functions. §4 describes our type-constant folding pass to evaluate run-time type-equality tests statically. §5 shows benchmarks on SYB, SYB3 and `lens`, measuring the optimizations' effects on running time and compilation time. This paper ends with a discussion of our contributions (§6), related work (§7) and conclusions (§8). All code in this paper is written in Haskell, and GHC Core, one of GHC's intermediate representations of Haskell. An artifact for this paper is available, and is described in the Data-Availability Statement.

2 Scrap Your Boilerplate

SYB is a prime example of the optimization-resistant pattern which we optimize, and is the primary motivation behind our work. In this section, we give an overview of SYB to identify causes of slowdowns exhibited by this pattern, and reasons why GHC does not optimize them.

SYB is a generic-programming library for concisely expressing traversals to *transform*, *query* or *monadically transform* deep data structures. These traversals are generic in that they work over any data type. For instance, the function `everywhere (mkT (+ (1 :: Int)))` increments every integer occurring in any data structure. The same definition can be reused unchanged to increment every integer in a list of integers, a tree of integers or even a deep data structure representing companies with employees having integer-valued salaries.

SYB traversals are supported by three key ingredients: (1) an *alias* which lifts the "interesting" part of the traversal (which we call the traversal's *intent*) into a function that can be applied to any data type (§2.1), (2) a type class whose instances define how to "pass along" the traversal down to subterms of the type (§2.2) and (3) a *traversal scheme* that defines how to recursively traverse data structures (§2.3). Using these combinators, SYB traversals can be expressed concisely, dispensing with the "boilerplate" characteristic of hand-written traversals. For example, the `add` function in

```
import Data.Generics
add_Int :: Int → Int → Int
add_Int = (+)
add :: Int → [Int] → [Int]
add k = everywhere (mkT (add_Int k))
```

Fig. 1. An SYB traversal that performs addition on every integer in a list of integers.

```
ghci> mkT (add_Int 1) 2
3
ghci> mkT (add_Int 1) [2]
[2]
```

Fig. 2. Example use of *mkT*.

```
mkT :: ∀α. ∀β.
    (Typeable α, Typeable β)
    ⇒ (β → β) → α → α
mkT f = case cast f of
  Nothing → id
  Just g → g
```

Fig. 3. Definition of *mkT*.

Fig. 1 is an SYB traversal which adds some integer k to every Int in a list. No part of *add*'s definition contains the typical "boilerplate" of traversing through the list, as list traversal is handled by the SYB combinators *everywhere* and *mkT*. The intent of the traversal is add_{Int}, which specifies the essence of *add*: integer addition.

2.1 Type-Safe Casts, Typeable and Aliases

SYB exports combinators called *aliases* which *lifts* traversal intents into polymorphic functions. When applied to an "interesting" data structure, the lifted function behaves as the original intent, but when applied to a term of some other type, the lifted function defaults to some uninteresting behavior. As an example, the alias *mkT* (short for *make transformation*) lifts a transformation on a specific type into one that can be applied to *any* type. Given a transformation $f :: \tau \to \tau$ on a term of some type τ, $mkT\ f :: \forall\alpha.\ \text{Typeable}\ \alpha \Rightarrow \alpha \to \alpha$ is a lifted function such that when applied to an argument of type τ, $mkT\ f$ behaves as f, and when applied to an argument of any other type, $mkT\ f$ behaves as the identity function *id*. In essence, *mkT* allows the intent of a transformation to be applied to every subterm of a data structure, performing the actual transformation only when applied to terms of the target type.

Fig. 2 shows an example execution of *mkT*, where the add_{Int} function is defined in Fig. 1. Although add_{Int} can be applied only to Ints, $mkT\ (add_{\text{Int}}\ k)$ can be applied to any type. As shown in Fig. 2, applying this lifted function to integers performs addition, while applying it to a term of another type, like lists, leaves the term unchanged.

The definition of *mkT* is shown in Fig. 3. The ability for aliases like *mkT* to lift functions into polymorphic ones is supported by type-safe casts via the *cast* function, whose definition is shown in Fig. 4. Type-safe casts are enabled by the Typeable type class, which defines how types can have a term-level representation. The *eqTypeRep* function shown in Fig. 4 uses *sameTypeRep* to test the equality of two type representations, generating a coercion witnessing the equality of two types only when they have the same type representation. The coercion generated by *eqTypeRep* is used by *cast* to cast its argument into the desired type. Although the definition of *eqTypeRep* uses *unsafeCoerce*, its use here is type-safe since the coercion is produced only when the two type arguments α and β are actually equal.

Finally, SYB also exports other aliases for generic transformations such as *extT* (short for *extend transformation*), and aliases for generic queries and monadic transformations like *mkQ* and *mkM*. These aliases all operate on the same principle of type-safe casts.

2.2 The Data Type Class

SYB uses Haskell's Data type class, whose methods define how to "pass along" a traversal to the immediate subterms of a data type. Fig. 5 shows an excerpt of the Data type class definition. The *gmapT* method applies a transformation to the immediate subterms. The full Data definition also

```
cast :: ∀α. ∀β. (Typeable α, Typeable β) ⇒ α → Maybe β
cast x  | Just HRefl ← eqTypeRep tα tβ = Just x
        | otherwise                   = Nothing
  where { tα = typeRep :: TypeRep α; tβ = typeRep :: TypeRep β }

eqTypeRep :: ∀κ1. ∀κ2. ∀(α :: κ1). ∀(β :: κ2). TypeRep α → TypeRep β → Maybe (α :~~: β)
eqTypeRep a b  | sameTypeRep a b = Just (unsafeCoerce HRefl)
               | otherwise       = Nothing

sameTypeRep :: ∀κ1. ∀κ2. ∀(α :: κ1). ∀(β :: κ2). TypeRep α → TypeRep β → Bool
sameTypeRep a b = typeRepFingerprint a == typeRepFingerprint b
```

Fig. 4. Type-safe casts.

```
class Typeable α ⇒ Data α where
  gmapT :: (∀β. Data β ⇒ β → β) → α → α

  gfoldl :: ∀χ. (∀δ. ∀β. Data δ ⇒
                  χ (δ → β) → δ → χ β) →
                (∀γ. γ → χ γ) → α → χ α
```

Fig. 5. Data type class definition (excerpt).

```
instance Data Int where
  gmapT f x = x

instance Data α ⇒ Data [α] where
  gmapT f []       = []
  gmapT f (x : xs) = f x : f xs
```

Fig. 6. Example handwritten Data instances.

consists of methods like *gmapQ* for queries and *gmapM* for monadic transformations, as well as other combinators that behave similar to these.

Fig. 6 shows example hand-written instance definitions of Data for lists and Ints, each containing definitions of *gmapT*. These combinators are not recursive, only applying the argument f to the immediate subterms. The recursion seen in typical hand-written traversals are handled by SYB traversal schemes, which we discuss in §2.3. Crucially, *gmapT* is rank-2 polymorphic, i.e., it receives polymorphic functions as arguments. Rank-2 polymorphism is required for *gmapT*—the function argument must itself be polymorphic so that it can be applied to the immediate subterms of potentially different types. For example, in the Data type class instance definition for lists in Fig. 6, the *gmapT* method receives f and applies it to both x of type α and xs of type $[\alpha]$.

Hand-writing Data instances for large data structures is tedious. Thus, SYB supports **deriving** Data instances, which use default definitions (which we omit in Fig. 5 to save space) of *gmapT*. These default definitions are expressed in terms of the *gfoldl* combinator, an abstraction of all of the other combinators. The optimization techniques we present is agnostic towards a programmer's choice of giving explicit definitions of its combinators, or using the default definition of these combinators expressed via a derived *gfoldl*.

2.3 Traversal Schemes

Finally, the SYB library exports *traversal schemes* that specify methods of recursively traversing data structures. A typical traversal scheme for generic transformations is shown in Fig. 7: *everywhere* f applies a transformation f on all subterms in a data structure in top-down fashion. The *everywhere* traversal scheme produces generic traversals that can be applied to any data type with a Data instance. Traversal schemes are also rank-2 polymorphic like *gmapT*, but in addition, they are *recursive* since they traverse entire data structures. SYB also exports other traversal schemes that expose different ways to recursively traverse data structures and for different purposes.

```
everywhere :: (∀β. Data β ⇒ β → β) → (∀α. Data α ⇒ α → α)
everywhere f = let go = f ∘ gmapT go in go
```

Fig. 7. Definition of the *everywhere* traversal scheme.

2.4 Why Is SYB So Slow?

The slow performance of SYB traversals is frequently documented in the literature. Yakushev [58] benchmarked three SYB functions, and found them to be 36, 52, and 69 times slower than hand-written code. Chakravarty et al. [7] benchmarked SYB on three functions, finding them to be 45, 73, and 230 times slower than handwritten code. Brown and Sampson [6] developed a new generic-programming library, Alloy, as SYB was too slow to meet their performance requirements, and in their benchmarks found SYB to be 4 to 23 times slower than their own approach. Magalhães et al. [39] report SYB performing between 3 and 20 times slower than handwritten code. Adams and DuBuisson [1] developed Template Your Boilerplate, an implementation of SYB using Template Haskell, and report SYB performing between 10 and nearly 100 times slower than handwritten code. Sculthorpe et al. [53] benchmark SYB on two functions, finding it to be around 5 times slower than handwritten code. Adams et al. [2, 3] ran SYB on seven benchmarks and found SYB to perform between 5 and 70 times slower than handwritten code. Yallop [60, 61] developed an optimized variant of SYB in MetaOCaml [30] that uses multi-stage programming, and found the direct SYB port to OCaml to be up to 20 times slower than handwritten code. All of these papers conclude that SYB is one of the slowest generic-programming libraries.

Yallop [60, 61] observes that the causes of slowdowns exhibited by SYB traversals can be largely traced to (1) SYB testing for type equality at each subterm with aliases like *mkT*, (2) SYB using polymorphic overloaded type class methods like *gmapT*, and (3) SYB making indirect calls through arguments, not to statically known functions, as with *everywhere* and *gmapT*. For instance, the list [1, 2, 3, 4, 5] contains only five integers to perform addition on, but it actually has a total of ten subterms. This means that applying *add k* from Fig. 1 on this list involves ten calls to *everywhere* (to be more specific, ten calls to the static-argument-transformed function *go* in the definition of *everywhere*), ten calls to the overloaded *gmapT* involving dictionary lookup, and ten type-equality tests invoked by *mkT*, just to perform addition on five integers.

GHC has thus far been unable to optimize SYB traversals. In our tests, GHC 9.4.8 attempts some optimizations on SYB traversals but aborts quickly, while GHC 9.8.4 forgoes them entirely. To see why, consider the *add* function from Fig. 1. Ideally, the GHC specializer [26] should specialize *add* to [Int] to eliminate the overhead caused by class-dictionary passing and enable downstream optimizations. To do so, *everywhere* must be inlined to expose the type and dictionary arguments to *mkT* ($add_{\mathsf{Int}}\ k$) and *gmapT*. However, GHC hesitates to inline recursive functions like *everywhere* as it risks non-termination during compilation [28]. The result is a stalemate—specialization demands inlining, but inlining is avoided.

3 Partially Evaluating Partially Static Applications of Rank-2 Polymorphic Functions

To break the optimization stalemate caused by recursive rank-2 polymorphic functions, we introduce an optimization technique that exposes applications of polymorphic arguments to concrete types and dictionaries. This enables class-dictionary specialization and unlocks subsequent compiler optimizations that would otherwise be blocked. Our optimization targets Haskell programs de-sugared to GHC Core, equivalently, System F_C [56]. We use this intermediate representation as the basis for both our formalization and our implementation. An excerpt of the syntax of System F_C is shown in Fig. 8. Continuing our running example, GHC de-sugars the *add* function from Fig. 1

Binds

b	::=	$x :: \tau = e$	

Expressions

e	::=	x	▷ *Variable*
	|	ℓ	▷ *Literal*
	|	C	▷ *Constructor*
	|	$\Lambda\alpha :: \kappa.\ e$	▷ *Type Abstraction*
	|	$\lambda x :: \tau.\ e$	▷ *Term Abstraction*
	|	$e\ @\tau$	▷ *Type Application*
	|	$e_1\ e_2$	▷ *Term Application*
	|	**let** $\overline{b}$ **in** e_2	▷ *Let Binding*
	|	**case** e' **of** $\overline{p \to e}$	▷ *Pattern Match*
	|	$e \triangleright \gamma$	▷ *Cast*

Patterns

p	::=	$C\ \overline{x :: \tau}$	

Types

τ	::=	α	▷ *Type Variables*
	|	T	▷ *Type Constant*
	|	$\forall\alpha :: \kappa.\ \tau$	▷ *Polymorphism*
	|	$\tau_1\ \tau_2$	▷ *Type-level Application*

Kinds

κ	::=	$\star$	▷ *Base kind*
	|	$\kappa_1 \to \kappa_2$	▷ *Kind of Type Constructors*
	|	$\tau_1 \sim_\rho \tau_2$	▷ *Kind of Coercions*

Coercions

γ	::=	**refl** τ	▷ *Coercion Reflexivity*
	|	**sym** γ	▷ *Coercion Symmetry*
	|	**nth**^{i} γ	▷ *Coercion Decomposition*
	|	$\gamma_1\ @\ \gamma_2$	▷ *Coercion Instantiation*
	|	$\gamma_1\ ⨟\ \gamma_2$	▷ *Coercion Composition*

Fig. 8. System F_C syntax (excerpt). Overbars $\overline{x}$ represent sequences $x_1, \ldots, x_n$.

$$add = \lambda k :: \mathsf{Int}.\ everywhere\ (\Lambda\alpha.\ \lambda d' :: \mathsf{Data}\ \alpha.\ mkT\ @\alpha\ @\mathsf{Int}\ \ldots\ (add_{\mathsf{Int}}\ k))\ @[\mathsf{Int}]\ (\ldots)$$

Fig. 9. The *add* function from Fig. 1 de-sugared to System F_C. The kind of type arguments and class-dictionary terms are omitted for simplicity.

into the System F_C function shown in Fig. 9 (we omit some parts of the function for simplicity). In this section, we develop intuition behind our technique using our running example, then dive into technical details in §3.1 to §3.5.

Rank-2 polymorphic functions can receive polymorphic functions to apply them to different types. This allows more expressive designs but adds layers of indirection between polymorphic functions and the types and dictionaries they are eventually applied to. As explained in §2.4, this hides class-dictionary-specialization opportunities, particularly when the definition of the rank-2 polymorphic function is not unfolded due to recursion. The goal of our optimizer is thus to reveal class-dictionary-specialization opportunities in the presence of recursion and rank-2 polymorphism.

Our approach to doing so is to, repeatedly, forcefully unfold rank-2 polymorphic functions and simplify the resulting function application. However, blindly doing so on recursive functions leads to the same non-terminating behavior that the GHC inliner avoids. To solve this, instead of simplifying entire function applications, we selectively simplify only the portion of the function application containing (1) the unfolded function being applied, (2) type and dictionary arguments, and (3) the polymorphic function argument to specialize. In typical use-cases of rank-2 polymorphism, the set of these is finite (we discuss situations when this is violated in §6). This allows us to deal with recursion via *memoization*, which avoids unfolding and simplifying re-occurring expressions that have already been optimized in previous iterations of transformations. As we soon show via our running example, forcing inlining and simplification with memoization in this manner has the effect of specializing polymorphic functions to finitely many types, generating a series of type-specific mutually recursive functions. This is so that (1) the optimization pass terminates on typical use-cases, and (2) all program transformations used by this optimization are standard and easy to implement. Specifically, we use the inlining and simplification facilities of the GHC inliner [28], and memoization is done via introducing **let**-bindings and performing **let**-floating on the result, hoisting it to the broadest scope possible.

The central challenge, therefore, is not *how* code is transformed, but instead, *where* the transformations are applied. Clearly, given a rank-2 polymorphic function application, we inline the function being applied, and simplify the function application up to the type and class-dictionary arguments and the polymorphic function arguments. Crucially, transforming *every* such occurrence may not yield significant results and only cause code bloat. Instead, we target rank-2 polymorphic function applications that are *partially static*, i.e., have some information available at compile-time. This is so that we transform only the occurrences that have tangible specialization opportunities that downstream optimization passes can leverage.

However, as we soon show with our running example, targeting only rank-2 polymorphic functions is insufficient. Class-dictionary-specialization opportunities are hidden when there are layers of indirection between polymorphic functions and the dictionary arguments they are eventually applied to. Rank-2 polymorphic functions can cause this, but so do (rank-1) functions that eventually invoke rank-2 polymorphic functions. As such, our optimizer performs a transitive analysis to identify all functions that, when repeatedly optimized, eventually lead to a partially static application of a rank-2 polymorphic function, our main optimization target. This ensures we uncover all latent specialization opportunities within a call graph.

In summary, our optimization pass repeatedly performs iterations of the following procedure until no more optimization targets are found:

(1) *Identifying Optimization Targets* (§3.2). Determine which expression(s) are optimization targets. Optimization targets are partially static applications of rank-2 polymorphic functions, and functions that, when transformed, eventually lead to another optimization target.
(2) *Inlining* (§3.3). In a target function application, unfold the definition of the function being applied.
(3) *Simplification* (§3.4). Simplify the function application after inlining the applied function. For System F_C programs, this is just β-reduction.
(4) *Memoization* (§3.5). Place the simplified function application in a **let**-binding and **let**-float the binding to the broadest possible scope. This has the same effect of a *static-argument transformation* which reduces overhead from allocating function closures, and memoizes transformed expressions. Re-occurring optimization targets are replaced with its memoization.

In essence, our optimization pass is a *partial evaluator* [16, 27, 41]. We show how our partial evaluator works by optimizing *add* in Fig. 9.

Iteration 1 (Demonstration of Program Transformations). The first argument to *everywhere* is polymorphic, thus, an application of *everywhere* to a partially static first argument is an optimization target. In the definition of *add* in Fig. 9, the first argument of the application of *everywhere* is a lambda abstraction, considered partially static. Thus, we partially evaluate the function application by inlining *everywhere*, simplifying the function application, and memoizing the simplified expression by placing it in a **let** binding. These steps are illustrated in Fig. 10. For presentational reasons, we perform the memoization first to clearly demarcate the expression we are targeting. In practice, performing the memoization last has the same effect. To allow the memoized expression to be in the broadest scope possible, we **let**-float the generated binding. Note that if we had required the argument to *everywhere* to be *completely static*, we would be unable to identify any optimization targets since *k* is a lambda-bound variable of *add*, thus, dynamic (not static).

Iterations 2–4 (Transitively Identifying Optimization Targets). The type of *gmapT* is:

$$\forall\alpha.\ \mathsf{Data}\ \alpha \to (\forall\beta.\ \mathsf{Data}\ \beta \to \beta \to \beta) \to \alpha \to \alpha$$

The first two arguments are its type and dictionary arguments, respectively, and its third argument is a polymorphic function. This suggests that an application of *gmapT* with a partially static third

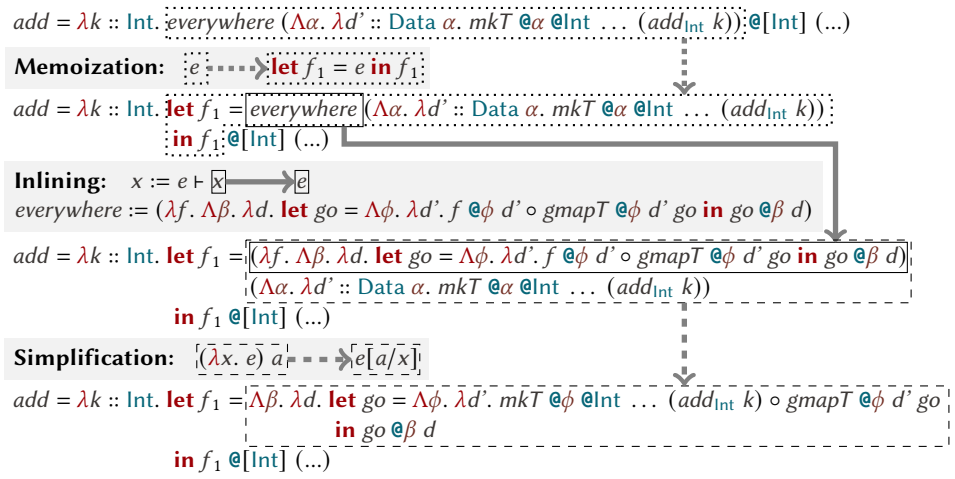


Fig. 10. One iteration of transformations on the application of *everywhere* in the definition of *add* from Fig. 9.

argument should be an optimization target. However, because *gmapT* is a class method, if the dictionary argument to *gmapT* (the second argument) is unknown, inlining *gmapT* has no benefit. Instead, we should consider an application of *gmapT* as an optimization target only if its dictionary argument is also partially static.

If we had considered only rank-2 polymorphic functions as optimization targets, we would be stuck. The program resulting from Iteration 1, shown in Fig. 10, has no optimization targets because the dictionary argument of the application of *gmapT* is not partially static. However, notice that based on the definition of *go*, applying *go* to a partially static second argument reveals a partially static application of *gmapT*. To illustrate this, using a bullet • to stand in as "some partially static term", applying *go* to • as the second argument gives us the rightmost expression of the following equivalence, which contains a partially static application of *gmapT*:

$$go\ \_\ \bullet \equiv (\Lambda\phi.\ \lambda d'.\ \cdots \circ gmapT\ @\phi\ d'\ go)\ \_\ \bullet \equiv \cdots \circ gmapT\ \_\ \bullet\ go$$

In fact, f_1 should also be an optimization target when applied to a partially static second argument, since partially evaluating that reveals a partially static application of *go* in its second argument:

$$f_1\ \_\ \bullet \equiv (\Lambda\beta.\ \lambda d.\ \ldots\ go\ @\beta\ d)\ \_\ \bullet \equiv \ldots\ go\ \_\ \bullet$$

Since the program resulting from Iteration 1 shown in Fig. 10 contains the function application f_1 @[Int] (...), which is partially static in its second argument (the expression (...) is the Data dictionary for [Int]), we perform the same transformations on f_1 @[Int] (...), illustrated as Iteration 2 in Fig. 11. As expected, doing so reveals a partially static application of *go*, which, again, we partially evaluate, shown as Iteration 3 in Fig. 11. Note that for concision of our presentation we inline f_2 and eliminate the dead **let** bindings of f_1 and f_2 before applying the usual transformations. In practice, eliminating **let** bindings prematurely undoes the memoization and static-argument transformation, and is avoided by our partial evaluator. These simplifications are instead left to the GHC simplifier [28], which runs only after partial evaluation has completed.

Finally, Iteration 3 reveals a partially static application of *gmapT*, our main optimization target. We partially evaluate this, as shown in Iteration 4 of Fig. 11. Notice that these iterations have successfully revealed applications of *mkT*, *go* and *gmapT* to concrete types such as Int and [Int], unlocking class-dictionary specialization opportunities.

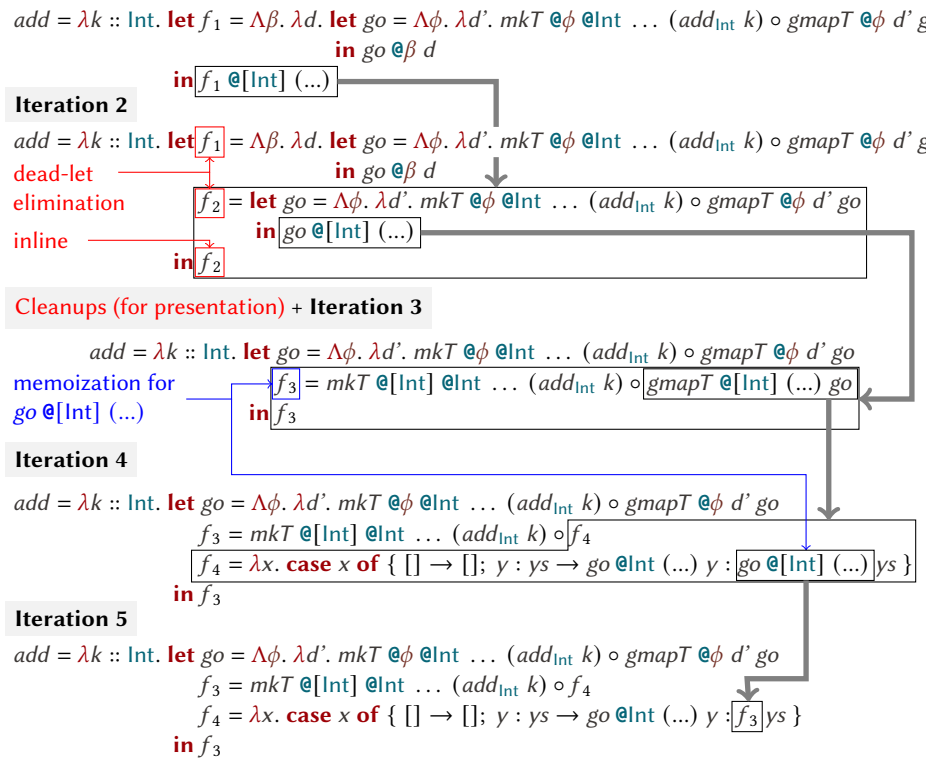


Fig. 11. Four more iterations of partial evaluation on the program produced in Fig. 10. Thick arrows indicate transformations performed in each iteration. Some cleanups are performed before running Iteration 3 to simplify the presentation.

In these iterations we have shown the conditions for when a function application should be considered an optimization target. Essentially, if a function f is polymorphic in its i^{th} argument, then an application of f to a partially static i^{th} argument (and dictionary argument, if any) is an optimization target. Moreover, if applying a function f to some specific partially static arguments reveals a partially static application of another optimization target, then an application of f to those partially static arguments is also an optimization target.

Iteration 5 (Memoization). As shown in Fig. 11, Iteration 4 reveals two more partially static applications of *go*. Since *go* is recursive and [Int] is a recursive data structure, as expected, f_4 also contains the expression *go* @[Int] (...) which we previously partially evaluated in Iteration 3. Had we continued to transform this function application like we have done previously, we would run into an infinite loop. Instead, since we have memoized the result of partially evaluating this function application as f_3 which is in scope, we replace *go* @[Int] (...) with f_3, shown as Iteration 5 in Fig. 11. In essence, memoization allows us to optimize recursive functions without (necessarily) running into non-termination.

Iterations 6–7. Partially evaluating the remaining partially static application of *go* from Fig. 11 and running one more iteration of partial evaluation yields the program shown in Fig. 12 (simplified for presentation) where there are no more optimization targets. This is the final result of partial evaluation, which is beginning to take on the structure of a hand-written, non-generic traversal!

$$
\begin{aligned}
add = \lambda k :: \mathsf{Int}.\ \mathbf{let}\ & f_3 = mkT\ \texttt{@}[\mathsf{Int}]\ \texttt{@}\mathsf{Int}\ \ldots\ (add_{\mathsf{Int}}\ k) \circ f_4 \\
& f_4 = \lambda x.\ \mathbf{case}\ x\ \mathbf{of}\ \{\ [] \to [];\ (y : ys) \to f_5\ y : f_3\ ys\ \} \\
& f_5 = mkT\ \texttt{@}\mathsf{Int}\ \texttt{@}\mathsf{Int}\ \ldots\ (add_{\mathsf{Int}}\ k) \\
\mathbf{in}\ & f_3
\end{aligned}
$$

Fig. 12. The (simplified) result of partially evaluating *add* from Fig. 9 to completion.

$$
\begin{array}{lrll}
\text{Definition/Binding Contexts:} & \Sigma & ::= & \cdot & \triangleright \textit{ Empty context} \\
& & \mid & \Sigma, x := e & \triangleright\ x \textit{ is defined as } e \\
& & \mid & \Sigma, \mathrm{bound}(x) & \triangleright\ x \textit{ is bound to a } \lambda/\Lambda
\end{array}
$$

Fig. 13. Definition/Binding contexts for keeping track of in-scope definitions and bound variables.

Rules 3.1: Partially Static Terms $\Gamma; \Sigma \vdash e$ p.s.

$$
\textbf{(PS:Bullet)}\ \frac{}{\Gamma; \Sigma \vdash \bullet \text{ p.s.}} \qquad
\textbf{(PS:Const)}\ \frac{c \text{ is a constant symbol}}{\Gamma; \Sigma \vdash c \text{ p.s.}} \qquad
\textbf{(PS:App)}\ \frac{\Gamma; \Sigma \vdash e_1 \text{ p.s.}}{\Gamma; \Sigma \vdash e_1\ e_2 \text{ p.s.}} \qquad
\textbf{(PS:Lam)}\ \frac{}{\Gamma; \Sigma \vdash \lambda x.e \text{ p.s.}} \qquad
\textbf{(PS:Var)}\ \frac{\Gamma; \Sigma \vdash x := e \qquad \Gamma; \Sigma \vdash e \text{ p.s.}}{\Gamma; \Sigma \vdash x \text{ p.s.}}
$$

$$
\textbf{(PS:Let)}\ \frac{\Gamma, x : \tau; \Sigma, x := e \vdash e' \text{ p.s.}}{\Gamma; \Sigma \vdash \mathbf{let}\ x = e\ \mathbf{in}\ e' \text{ p.s.}} \qquad
\textbf{(PS:Cast)}\ \frac{\Gamma; \Sigma \vdash e \text{ p.s.}}{\Gamma; \Sigma \vdash e \triangleright \gamma \text{ p.s.}}
$$

The rest of this section formally specifies each component of our partial evaluator.

3.1 Definition/Binding Contexts

Throughout the rest of this section, we use judgments of the form $\Gamma; \Sigma \vdash \mathcal{J}$. Γ represents typing contexts and Σ represents definition/binding contexts of the form shown in Fig. 13. This is to maintain access to the definitions of variables, useful for identifying optimization targets and inlining.

3.2 Identifying Optimization Targets

3.2.1 Partially Static Terms. Rules 3.1 determine if a term is partially static. The judgment form $\Gamma; \Sigma \vdash e$ p.s. states that e is partially static. The rules PS:Const and PS:Lam state that constant symbols and lambda abstractions are always partially static, and the remaining rules describe how terms can be partially static based on their subterms. For presentational reasons later, we define a new constant symbol $\bullet$ to stand for "some partially static term", which by the rule PS:Bullet states that it is partially static. $\bullet$ is not a valid System F_C term and will not appear in the transformed programs, and is used only to decide whether a function is a target of optimization. Note that the judgment $\Gamma; \Sigma \vdash x := e$ (e.g., in PS:Var) states that x is defined as e.

3.2.2 Cast-Insensitive Function-Application Decomposition. Casts ($e \triangleright \gamma$) can obscure function applications. These occur primarily when types are declared with **newtype**. Since casts are "no-ops" and do not affect runtime semantics, we introduce cast-insensitive function-application decomposition operators to identify components of a function application. These operators are defined in Rules 3.2: $\|e\|$ obtains the number of arguments of e (rules App-Size:Var and App-Size:App), $\mathrm{fn}(e)$ obtains the function being applied in e (rules App-Fn:Var and App-Fn:App), and $\mathrm{arg}_i(e)$ obtains the i^{th} argument in e (rules App-Arg:Left and App-Arg:This).

Rules 3.2: Cast-Insensitive Function-Application Decomposition

$$\frac{x \text{ is a variable}}{\|x\| = 0}\ (\textbf{App-Size:Var}) \qquad \frac{\|e\| = n}{\|(e \triangleright \overline{\gamma})\ e'\| = n + 1}\ (\textbf{App-Size:App})$$

$$\frac{x \text{ is a variable}}{\mathrm{fn}(x) = x}\ (\textbf{App-Fn:Var}) \qquad \frac{\mathrm{fn}(e) = x}{\mathrm{fn}((e \triangleright \overline{\gamma})\ e') = x}\ (\textbf{App-Fn:App})$$

$$\frac{\|e\| \geq n \qquad \mathrm{arg}_n(e) = e''}{\mathrm{arg}_n((e \triangleright \overline{\gamma})\ e') = e''}\ (\textbf{App-Arg:Left}) \qquad \frac{\|e\| = n - 1}{\mathrm{arg}_n((e \triangleright \overline{\gamma})\ e') = e'}\ (\textbf{App-Arg:This})$$

Rules 3.3: Partially Static Function Applications $\Gamma; \Sigma \vdash e \ddagger f \triangleleft S$

$$\frac{\mathrm{fn}(e) = f \qquad \|e\| = n \qquad \emptyset \subset S \subseteq [1, n] \qquad \forall i \in S.\ \Gamma; \Sigma \vdash \mathrm{arg}_i(e)\ \mathrm{p.s.}}{\Gamma; \Sigma \vdash e \ddagger f \triangleleft S}\ (\textbf{PSA})$$

Rules 3.4: Partial-Evaluation Targets $\Gamma; \Sigma \vdash x \hookleftarrow A$

$$\frac{\Gamma \vdash x : \overline{\forall \alpha.}^{\,i \geq 0}\ \tau_{i+1} \to \cdots \to \tau_j \to \cdots \to \tau_n \qquad \tau_j = \forall \alpha.\tau \qquad j < n \qquad \Sigma \vdash x := e}{\Gamma; \Sigma \vdash x \hookleftarrow \{j\}}\ (\textbf{Target:Rank-2-Fn})$$

$$\frac{\Gamma \vdash x : \overline{\forall \alpha.}^{\,i \geq 0}\ \tau_{i+1} \to \cdots \to \tau_j \to \cdots \to \tau_n \qquad \tau_j = \forall \alpha.\tau \qquad \Sigma \vdash \mathrm{ClassOp}(x) \qquad i + 1 < j < n}{\Gamma; \Sigma \vdash x \hookleftarrow \{i + 1, j\}}\ (\textbf{Target:Rank-2-Class-Op})$$

$$\frac{\Gamma; \Sigma \vdash x := \overline{\Lambda \alpha.}\ \overline{\lambda b_i.}^{\,>0}\ e \qquad e' \trianglelefteq e \qquad \Gamma; \Sigma \vdash f \hookleftarrow S \qquad \Gamma; \Sigma \not\vdash e' \ddagger f \triangleleft S \qquad \Gamma; \Sigma \vdash e'[\bullet/b_i \mid i \in K] \ddagger f \triangleleft S}{\Gamma; \Sigma \vdash x \hookleftarrow K}\ (\textbf{Target:Indirect})$$

3.2.3 Partially Static Function Applications. We use the judgment form $\Gamma; \Sigma \vdash e \ddagger f \triangleleft S$ to state that the expression e is an application of the function symbol f, such that for all i in S where S is a nonempty set of indices, the i^{th} argument is partially static. These judgments are derived using the PSA rule in Rules 3.3.

3.2.4 Partial-Evaluation Targets. Rules 3.4 shows rules for finding optimization targets, deriving judgments of the form $\Gamma; \Sigma \vdash x \hookleftarrow A$ which states that an application of x is an optimization target when for all i in A, the i^{th} argument is partially static. The Target:Rank-2-Fn rule requires the i^{th} argument of a function application to be partially static if it is polymorphic. The Target:Rank-2-Class-Op rule is similar, but additionally requires the dictionary argument to be partially static then the function being applied is a class method (given by the judgment $\Sigma \vdash \mathrm{ClassOp}(f)$). Lastly, the Target:Indirect rule makes a function f an optimization target when applying it to some partially static arguments reveals a partially static application of another optimization target (note that $e' \trianglelefteq e$ states that e' *occurs in/is a subexpression of* e). Crucially, Target:Indirect requires that the subexpression e' is not already a partially static application of the target. If it is already, it can be partially evaluated directly without invoking x.

3.3 Inlining

Partial evaluation performs inlining, which replaces a variable with its definition, and is a standard transformation that is also performed by the GHC simplifier [28]. We inline symbols in two ways, hence two inlining judgments of the form $\Gamma; \Sigma \vdash e \rightsquigarrow_\iota e'$ and $\Gamma; \Sigma \vdash e \rightsquigarrow_\delta e'$. The former is the main inlining judgment which inlines the function symbol of a function application. The latter

Rules 3.5: Inlining $\Gamma; \Sigma \vdash e \rightsquigarrow_\iota e'$

$$\frac{\Gamma; \Sigma \vdash f := e}{\Gamma; \Sigma \vdash f \rightsquigarrow_\iota e} \text{ (Inline:Fn)} \qquad \frac{\Gamma; \Sigma \vdash e \rightsquigarrow_\iota e'}{\Gamma; \Sigma \vdash (e \triangleright \overline{\gamma})\ e'' \rightsquigarrow_\iota (e' \triangleright \overline{\gamma})\ e''} \text{ (Inline:App)}$$

$$\frac{\Gamma; \Sigma \vdash \text{ClassOp}(f) \qquad \Gamma; \Sigma \vdash e'' \text{ is the dictionary argument to } f \qquad \Gamma; \Sigma \vdash e \rightsquigarrow_\iota e' \qquad \Gamma; \Sigma \vdash e'' \rightsquigarrow_\delta e'''}{\Gamma; \Sigma \vdash (e \triangleright \overline{\gamma})\ e'' \rightsquigarrow_\iota (e' \triangleright \overline{\gamma})\ e'''} \text{ (Inline:Class-Op-App)}$$

$$\text{fn}(e) = f$$

Rules 3.6: Inlining Dictionaries $\Gamma; \Sigma \vdash e \rightsquigarrow_\delta e'$

$$\frac{\Gamma; \Sigma \vdash f := e \qquad \Gamma; \Sigma \vdash \text{DFun}(e)}{\Gamma; \Sigma \vdash f \rightsquigarrow_\delta e} \text{ (Inline-Dict)} \qquad \frac{\Gamma; \Sigma \vdash f := e \qquad \Gamma; \Sigma \vdash e \rightsquigarrow_\delta e'}{\Gamma; \Sigma \vdash f \rightsquigarrow_\delta e'} \text{ (Inline-Dict:Defn)}$$

$$\frac{\Gamma, x : \tau; \Sigma, \text{bound}(x) \vdash e \rightsquigarrow_\delta e'}{\Gamma; \Sigma \vdash \lambda x.\ e \rightsquigarrow_\delta \lambda x.\ e'} \text{ (Inline-Dict:Abs)}$$

$$\frac{\Gamma; \Sigma \vdash e \rightsquigarrow_\delta e'}{\Gamma; \Sigma \vdash e\ e'' \rightsquigarrow_\delta e'\ e''} \text{ (Inline-Dict:App-Fn)} \qquad \frac{\Gamma; \Sigma \vdash e \rightsquigarrow_\delta e'}{\Gamma; \Sigma \vdash e \triangleright \gamma \rightsquigarrow_\delta e' \triangleright \gamma} \text{ (Inline-Dict:Cast)}$$

describes dictionary-argument inlining to reveal instance method definitions for partial evaluation to continue. The effect of inlining and simplifying class methods and their dictionary arguments is class-dictionary specialization [26].

The two sets of rules for inlining is shown in Rules 3.5 and Rules 3.6. Rules 3.5 state that the function symbol of a function application, and its dictionary argument (if the function is a class method) is inlined if its definition is in scope. Rules 3.6 describe the rules of inlining dictionary arguments where the predicate $\text{DFun}(e)$ determines if e is a function that constructs a class dictionary. Dictionary arguments are inlined to the point where the instance method definitions are exposed. Note that Rules 3.6 do not recursively inline dictionary arguments until the dictionary is fully elaborated, since this process neither terminates on recursive/undecidable instances nor is necessary for optimization. All of these rules change variable names to avoid name capture.

3.4 Simplification

Since System F_C is a typed lambda calculus, terms are simplified via β-reduction $e \rightsquigarrow_\beta e'$. The rules for β-reduction is given in Rules 3.7 (which define reduction of subterms) and Rules 3.8 (which define how function applications, casts, case expressions and let bindings are evaluated). These rules are not new and are taken from the operational semantics of System F_C as implemented in GHC [11]. We show these rules here to discuss its key aspects. Though the rules we present are extensive, implementing simplification in GHC is actually straightforward.

3.4.1 Cast Elimination. Although casts hide optimization targets, the simplification phase can eliminate as many casts in a type-safe manner as possible via the rules Simp:Trans-Cast, Simp:Let-Float-Cast and Simp:Case-Float-Cast, which groups casts together into a single coercion so that Simp:Rfl-Cast can drop reflexive casts.

3.4.2 A Note on Substitutions. Our version of β-reduction differs slightly from the form of β-reduction used by the GHC simplifier [28], where evaluating a function application produces a **let** binding that binds the parameter to the argument. The GHC simplifier does this to avoid duplicating expensive work, inlining the introduced definition of the variable only when there is no work duplication and/or the argument is inexpensive to evaluate. On the other hand, our partial evaluator performs substitution regardless of the number of occurrences of the bound variable in

Rules 3.7: Simplification (I) $e \rightsquigarrow_\beta e'$

$$\textbf{(Simp:Abs-Body)}\ \frac{e \rightsquigarrow_\beta e'}{\lambda x :: \tau . e \rightsquigarrow_\beta \lambda x :: \tau . e'} \qquad \textbf{(Simp:Ty-Abs-Body)}\ \frac{e \rightsquigarrow_\beta e'}{\Lambda \alpha :: \kappa . e \rightsquigarrow_\beta \Lambda \alpha :: \kappa . e'}$$

$$\textbf{(Simp:App-Fn)}\ \frac{e \rightsquigarrow_\beta e'}{e\ e'' \rightsquigarrow_\beta e'\ e''} \qquad \textbf{(Simp:App-Arg)}\ \frac{e \rightsquigarrow_\beta e'}{e''\ e \rightsquigarrow_\beta e''\ e'}$$

$$\textbf{(Simp:App-Ty-Arg)}\ \frac{e \rightsquigarrow_\beta e'}{e\ \tau \rightsquigarrow_\beta e'\ \tau} \qquad \textbf{(Simp:Cast)}\ \frac{e \rightsquigarrow_\beta e'}{e \triangleright \gamma \rightsquigarrow_\beta e' \triangleright \gamma} \qquad \textbf{(Simp:Let-Expr)}\ \frac{e \rightsquigarrow_\beta e'}{\textbf{let}\ \overline{b}\ \textbf{in}\ e \rightsquigarrow_\beta \textbf{let}\ \overline{b}\ \textbf{in}\ e'}$$

$$\textbf{(Simp:Let-Bind)}\ \frac{\overline{e_i \rightsquigarrow_\beta e_i'}}{\textbf{let}\ \overline{x_i :: \tau_i = e_i}\ \textbf{in}\ e \rightsquigarrow_\beta \textbf{let}\ \overline{x_i :: \tau_i = e_i'}\ \textbf{in}\ e} \qquad \textbf{(Simp:Case-Scr)}\ \frac{e \rightsquigarrow_\beta e'}{\textbf{case}\ e\ \textbf{of}\ \overline{p_i \to e_i} \rightsquigarrow_\beta \textbf{case}\ e'\ \textbf{of}\ \overline{p_i \to e_i}}$$

$$\textbf{(Simp:Case-Alt)}\ \frac{\overline{e_i \rightsquigarrow_\beta e_i'}}{\textbf{case}\ e\ \textbf{of}\ \overline{p_i \to e_i} \rightsquigarrow_\beta \textbf{case}\ e\ \textbf{of}\ \overline{p_i \to e_i'}}$$

Rules 3.8: Simplification (II) $e \rightsquigarrow_\beta e'$

$$\textbf{(Simp:Abs)}\ \frac{}{(\lambda x :: \tau .\ e_1)\ e_2 \rightsquigarrow_\beta e_1[e_2/x]} \qquad \textbf{(Simp:Ty-Abs)}\ \frac{}{(\Lambda \alpha :: \kappa .\ e)\ \tau \rightsquigarrow_\beta e[\tau/\alpha]} \qquad \textbf{(Simp:Rfl-Cast)}\ \frac{\gamma :: \tau \sim_\rho \tau}{e \triangleright \gamma \rightsquigarrow_\beta e}$$

$$\textbf{(Simp:Trans-Cast)}\ \frac{}{(e \triangleright \gamma_1) \triangleright \gamma_2 \rightsquigarrow_\beta e \triangleright (\gamma_1 \fatsemi \gamma_2)} \qquad \textbf{(Simp:App-Cast)}\ \frac{\gamma :: \tau_1 \to \tau_2 \sim_\rho \tau_1' \to \tau_2'}{(e_1 \triangleright \gamma)\ e_2 \rightsquigarrow_\beta (e_1\ (e_2 \triangleright \textbf{sym}\ (\textbf{nth}^0\ \gamma))) \triangleright (\textbf{nth}^1\ \gamma)}$$

$$\textbf{(Simp:App-Ty-Cast)}\ \frac{}{(e \triangleright \gamma)\ \tau \rightsquigarrow_\beta (e\ \tau) \triangleright (\gamma\ @\ \textbf{refl}\ \tau)} \qquad \textbf{(Simp:Case-Known-Con)}\ \frac{}{\textbf{case}\ C\ \overline{e_i}\ \textbf{of}\ \ldots C\ \overline{x_i : \tau_i} \to e \ldots \rightsquigarrow_\beta e[\overline{e_i/x_i}]}$$

$$\textbf{(Simp:Let-Float-App)}\ \frac{}{(\textbf{let}\ \overline{b}\ \textbf{in}\ e_1)\ e_2 \rightsquigarrow_\beta \textbf{let}\ \overline{b}\ \textbf{in}\ e_1\ e_2} \qquad \textbf{(Simp:Let-Float-Cast)}\ \frac{}{(\textbf{let}\ \overline{b}\ \textbf{in}\ e) \triangleright \gamma \rightsquigarrow_\beta \textbf{let}\ \overline{b}\ \textbf{in}\ e \triangleright \gamma}$$

$$\textbf{(Simp:Let-Float-Case)}\ \frac{}{\textbf{case}\ (\textbf{let}\ \overline{b}\ \textbf{in}\ e)\ \textbf{of}\ \overline{p_i \to e_i} \rightsquigarrow_\beta \textbf{let}\ \overline{b}\ \textbf{in}\ (\textbf{case}\ e\ \textbf{of}\ \overline{p_i \to e_i})} \qquad \textbf{(Simp:Case-Float-Cast)}\ \frac{}{(\textbf{case}\ e\ \textbf{of}\ \overline{p_i \to e_i}) \triangleright \gamma \rightsquigarrow_\beta \textbf{case}\ e\ \textbf{of}\ \overline{p_i \to e_i \triangleright \gamma}}$$

the body of the function at risk of duplicating work (Simp:Abs and Simp:Ty-Abs), so as to instantiate applications of polymorphic functions to concrete type and dictionary arguments, revealing more opportunities for further simplification and specialization. Note also that our substitutions rename variables to avoid name capture.

3.5 Partial Evaluation and Memoization

Finally, Rules 3.9 shows the rules for partially evaluating terms with memoization. The judgment form $\Gamma; \Sigma \vdash e \hookrightarrow e'$ states that e partially evaluates to e', and the judgment form $\Gamma; \Sigma \vdash \text{Memo}(e, x)$ states that the memoization of e is the variable x. The Partial-Eval:Opt-No-Memo rule states that if we find a partially static application of an optimization target that has not been partially evaluated before, we inline and simplify the application, then place it in a **let**-binding with a fresh identifier serving as a static-argument transformation and memoization. The Partial-Eval:Opt-Memo rule states that if we encounter an expression that has been memoized, and its memoization is in scope,

Rules 3.9: Partial Evaluation $\Gamma;\Sigma \vdash e \hookrightarrow e' \qquad \Gamma;\Sigma \vdash \mathrm{Memo}(e, e')$

(Partial-Eval:Opt-No-Memo)

$$\frac{\Gamma;\Sigma \vdash f \hookleftarrow S \qquad \Gamma;\Sigma \vdash e \ddagger f \triangleleft S \qquad \neg\exists x'.\, \Gamma;\Sigma \vdash \mathrm{Memo}(e, x') \qquad \Gamma;\Sigma \vdash e \rightsquigarrow_{\iota} e' \qquad e' \rightsquigarrow_{\beta} e''}{\Gamma;\Sigma \vdash e \hookrightarrow \textbf{let}\ x = e''\ \textbf{in}\ x \qquad \Gamma;\Sigma \vdash \mathrm{Memo}(e, x)}$$

(Partial-Eval:Opt-Memo)

$$\frac{\Gamma;\Sigma \vdash f \hookleftarrow S \qquad \Gamma;\Sigma \vdash e \ddagger f \triangleleft S \qquad \Gamma;\Sigma \vdash \mathrm{Memo}(e, x)}{\Gamma;\Sigma \vdash e \hookrightarrow x}$$

then we rewrite it to its memoization. Note that the memoization is also floated out to the broadest scope possible.

3.6 Correctness

Our partial evaluator performs only standard transformations–inlining, β-reduction, and memoization via **let** introduction and **let**-floating–which GHC also performs. These transformations are unconditionally type- and meaning-preserving in pure, lazy Haskell [28, 56]. In particular, (1) β-reduction is confluent and preserves types/semantics, (2) inlining preserves types/semantics and (3) **let** introduction/floating preserves types/semantics. Individually, these transformations preserves types/semantics, thus, $\Gamma;\Sigma \vdash e \hookrightarrow e'$ also preserves types and semantics by induction on its derivation. Semantics preservation of our partial evaluator is violated when side-effects are permitted, for example, with functions like *unsafePerformIO*, or when the code has strictness annotations which can change termination behavior of the partially evaluated program.

The program transformations performed in each iteration of partial evaluation terminates. (Recursive) functions and class dictionary arguments are inlined only once, and β-reduction in GHC Core always terminates [28]. Thus, each iteration of the partial evaluator terminates. However, like many partial evaluators, unrestricted repeated application of our partial evaluator until no more optimization targets are found is not guaranteed to terminate. We discuss non-termination in §6.

4 Type-Constant Folding

The partial evaluator presented in §3 breaks the optimization deadlock in recursive rank-2 polymorphic functions. This can be sufficient for the GHC optimization pipeline to specialize the residual program and perform further optimizations. For example, SYB3 [35], the main variant of SYB, uses Haskell type classes for type-specific function dispatch, superseding type-safe casts used by SYB. GHC can further optimize SYB3 traversals by performing class-dictionary specialization on the partially evaluated residual. However, we identify one optimization opportunity enabled by the partial evaluator that is not exploited by GHC: run-time type-equality tests on statically known types. In this section, we describe a new, simple optimization pass that reduces the overhead caused by type-equality tests. We build intuition by continuing our running example, using Fig. 12 as a starting point.

After our partial evaluator transforms SYB traversals to completion, GHC inlines and simplifies aliases like *mkT* to some extent, stopping short at *sameTypeRep*. For example, GHC simplifies *mkT* `@Int @Int` ... from Fig. 12 into the expression shown in Fig. 14. This expression contains an application of *sameTypeRep*, used to compare type representations at run-time.

GHC avoids inlining *sameTypeRep* because it frequently causes code bloat without performance improvements. The *sameTypeRep* function compares type-representation fingerprints, generated via `IO` actions and foreign function calls, thus fully simplifying *sameTypeRep* at the GHC Core/System

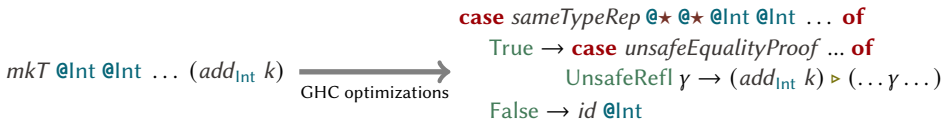


Fig. 14. Example of simplifications by the GHC simplifier on *mkT*.

F_C level is infeasible. To overcome this hurdle, we make two observations. Firstly, GHC guarantees that type representations uniquely define types by prohibiting users from writing bogus Typeable instances [48]. This guarantees that an application of *sameTypeRep* evaluates to True if and only if applied to two equal type arguments. Secondly, our partial evaluator expands the rank-2 polymorphic SYB combinators, instantiating SYB aliases with statically known types. Thus, the aliases simplify to *sameTypeRep* applied to statically known types too, giving us enough static information to evaluate type-equality tests at compile-time. For instance, without knowing how the type-representation fingerprints of Ints are computed, the application of *sameTypeRep* in Fig. 14 should obviously evaluate to True, since its two type arguments are equal.

Similar to how constant folding can evaluate 1 == 1 to True and 1 == 2 to False statically, we treat *sameTypeRep* as a primitive and perform a type-constant fold to evaluate type-equality tests at compile-time when static information is sufficient to give a result. For instance, in Fig. 14, *sameTypeRep* @★ @★ @Int @Int ... is replaced with True, while type-equality tests on disequal types like *sameTypeRep* @★ @★ @[Int] @Int ... is replaced with False. For SYB, this simple rewrite allows GHC to further simplify the residual into a form that closely resembles the intent of the traversal. For instance, as shown in Fig. 15, after re-writing the application of *sameTypeRep*, GHC can simplify the expression in Fig. 14 into just $add_{\mathsf{Int}}\ k$, i.e., applying $add_{\mathsf{Int}}\ k$ only on integers, as intended!

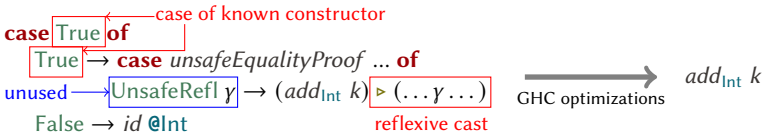


Fig. 15. Example of simplifications by the GHC simplifier on the residual of *mkT* after type-constant folding.

Rules 4.2 shows the rules for type-constant folding. The TCF:Eq rule states that when we find an application of *sameTypeRep* onto two equal types, we can immediately rewrite the application to True. On the other hand, the inverse is not true. When *sameTypeRep* is applied to different types, for example, Int and α, it is possible that type variables occurring in the type arguments are instantiated with types that make them equal. Thus, we perform the rewrites to False only when the type arguments are *always* disequal. This is given by Rules 4.1, which give criteria for when this is the case, i.e., when there are differing type-constant symbols. This is condition is quite weak and may not apply to all pairs of types that are always disequal. For our purposes, we do not require a more sophisticated analysis on types. Further note that these rules are justified as (1) *sameTypeRep* is never called on polymorphic or kind-polymorphic types (e.g., $\forall\alpha.\tau$) [48] and (2) type synonyms are fully elaborated.

To complete our running example, type-constant folding and downstream GHC optimizations re-writes *add* from Fig. 12 to become equivalent to hand-written, non-generic code. This is shown in Fig. 16, the result of optimizing *add* to completion.

RULES 4.1: TYPE DISEQUALITY $\tau \not\equiv \tau'$

(**TyDeq:Cons**)
$$\frac{C \neq D}{C\ \overline{\tau} \not\equiv D\ \overline{\tau}}$$

(**TyDeq:Arg**)
$$\frac{\tau_{1i} \not\equiv \tau_{2i}}{C\ \tau_{11}\ \ldots\ \tau_{1i}\ \ldots\ \tau_{1n} \not\equiv C\ \tau_{21}\ \ldots\ \tau_{2i}\ \ldots\ \tau_{2n}}$$

RULES 4.2: TYPE-CONSTANT FOLDING $e \rightarrowtail e'$

(**TCF:Eq**)
$$\frac{\tau_1 = \tau_2}{\mathit{sameTypeRep}\ \kappa_1\ \kappa_2\ \tau_1\ \tau_2\ e\ e' \rightarrowtail \mathsf{True}}$$

(**TCF:Deq**)
$$\frac{\tau_1 \not\equiv \tau_2}{\mathit{sameTypeRep}\ \kappa_1\ \kappa_2\ \tau_1\ \tau_2\ e\ e' \rightarrowtail \mathsf{False}}$$

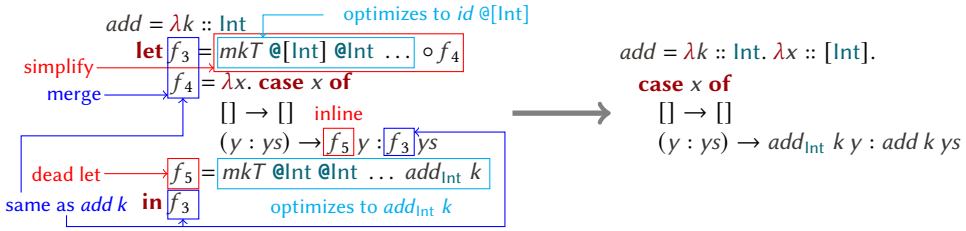


Fig. 16. The result of optimizing *add* from Fig. 9 to completion via type-constant folding and other GHC optimization passes.

Correctness. Type preservation of type-constant folding follows immediately since *sameTypeRep* always returns Bool. Semantic preservation follows immediately from the semantics of *sameTypeRep* and (dis)equality of types. Repeatedly applying type-constant folding to the program until no more eliminable type-equality tests occur is guaranteed to terminate due to the finiteness of programs, and that each rewrite does not introduce any more occurrences of eliminable expressions.

5 Benchmarks

We implemented our optimization techniques as a GHC plugin which transforms GHC Core (i.e., System F_C) programs. The plugin runs an initial GHC simplification pass first, followed by our partial evaluator (§3), then runs two iterations of our type-constant folding pass (§4) interleaved with other GHC optimization passes (we explain why we run this twice in §5.3). In this section, we describe our benchmarks for evaluating the effectiveness of our techniques by measuring our plugin's effects on running time and compilation times. Our benchmarks were developed using the Criterion benchmarking library [55],[1] were compiled with GHC version 9.8.4 using the `-O2` optimization level, and were run on a 3.4 GHz, 64-bit Intel i7 processor with 32 GB of RAM.

We benchmarked our plugin on the three libraries we found to be impacted by the optimization "roadblock": (1) SYB, (2) SYB3 and (3) `lens`. SYB3, like SYB, uses rank-2 polymorphism to "pass along" traversal functions to sub-terms. SYB3 is similar in purpose to SYB, but instead uses a constraint-polymorphic variant of Data that allows type-specific behavior of traversals to be written with type-class instances, superseding type-safe casts. SYB3 has no dependence on SYB, Typeable or Data. The `lens` library [32] is an optics library for defining lenses, prisms and traversals [15] (note that `lens` traversals are different to SYB/SYB3 traversals). The `lens` library consists of many modules; we benchmarked our plugin on `lens` using the library's own benchmark suites and found

[1]As of October 2025, Criterion has documented issues affecting measurement consistency (cf. GitHub (https://github.com/haskell/criterion) issues #60 and #166). This can shift both absolute timings and relative speedup magnitudes. For reference, for the fully optimized benchmarks, artifact reviewers reported deviations of ±15% on average, with the largest discrepancies being −30% to +40%.

```
add_Hand :: Int → [Int] → [Int]
add_Hand k [] = []
add_Hand k (x : xs) = k + x : add_Hand k xs
```

Fig. 17. Hand-written equivalent of *add* from Fig. 1.

Table 1. Benchmarked traversals.

Traversal Type	Data Structure		
	WTree	Company	Expr
Transformation	RmWeights	$\text{Inc}_{\text{Salary}}$	Inc_{Int}
Query	$\text{Select}_{\text{Int}}$	$\text{Select}_{\text{Float}}$	NumTypes
Monadic Transformation	RenumInt	AnonNames	DropCasts

that our plugin does not affect most of them, either because they do not use rank-2 polymorphic functions, or because GHC already successfully optimizes them without our plugin. However, running our optimization on the benchmarks for one of its modules, Control.Lens.Plated, showed an observable performance improvement. This module is a drop-in replacement of the Uniplate [44] library for simple traversals. The `lens` library also has no dependence on SYB.

Note that we have also benchmarked our plugin on other libraries that use rank-2 polymorphism, but found that our plugin has no effect. This is because the polymorphic argument to the rank-2 polymorphic function is not class-constrained. An example of this is *hoist* from recursion-schemes:

$$\mathit{hoist} :: (\text{Recursive}\ \sigma, \text{Corecursive}\ \tau) \Rightarrow (\forall \alpha.\text{Base}\ \sigma\ \alpha \to \text{Base}\ \tau\ \alpha) \to \sigma \to \tau$$

The *hoist* function is indeed rank-2 polymorphic and recursive. Our partial evaluator does transform this function, though in doing so, yields no performance improvement since no class-dictionary-specialization opportunities are exposed.

5.1 Benchmark Suites: SYB/SYB3

Some of the benchmarks we use are from the Haskell generic-programming literature. We use a generic-programming benchmark suite, GPBench [51], which also contains examples from the original SYB paper [33]. We also use benchmarks from the Template Your Boilerplate (TYB) [1] paper to benchmark generic monadic transformations. Some of the traversals in these suites are omitted since they test the same kind of traversal over the same data structures, or traverse over data structures that are of a similar size to the other data structures used.

Our benchmark suite tests our plugin on traversals over three data structures: (1) WTree is a weighted binary tree, taken from GPBench [51], (2) Company is a data structure of a company with departments and employees, taken from GPBench and is the main running example of the original SYB paper [33], and (3) Expr is a large excerpt of the data structure representing expressions in GHC Core, consisting of 18 (mostly) mutually recursive types and a total of 64 constructors.

We benchmark our optimization on three kinds of traversals: (1) generic transformations, (2) generic queries and (3) generic monadic transformations on these three data structures. Each traversal was implemented non-generically (Hand), using SYB (SYB) and using SYB3 (SYB3). The Hand versions are straightforward implementations with no extra performance tuning of the same algorithm expressed by the SYB traversals. As an example, a hand-written equivalent of *add* in Fig. 1 is $\mathit{add}_{\mathit{Hand}}$ in Fig. 17.

The traversals used are summarized in Table 1. These are:

(1) RmWeights is taken from GPBench [51]. It removes weights from a WTree. Due to SYB3's design, the SYB3 and hand-written implementations of this traversal are equivalent.
(2) $\text{Inc}_{\text{Salary}}$ is taken from the original SYB paper [33]. It increases salaries of employees in a Company.
(3) Inc_{Int} increments integer literals in an Expr.

(4) SELECT$_{\text{INT}}$ is taken from GPBench. It collects all the Ints in a WTree into a single list. Its implementation in GPBench uses two different algorithms for the Hand and SYB implementations. The Hand implementation uses a linear-time, accumulator-style traversal, while the SYB implementation uses a quadratic-time traversal. To ensure a fair comparison, we modified the Hand implementation to use a quadratic-time traversal as well. Due to SYB3's design, the SYB3 and hand-written implementations of this traversal are equivalent.
(5) SELECT$_{\text{FLOAT}}$ is analogous to SELECT$_{\text{INT}}$, but collects the Floats in a Company instead.
(6) NUMTYPES queries the number of types that occur in an Expr.
(7) RENUMINT is taken from the benchmarking suite of Template Your Boilerplate [1]. It uses a State monad to generate unique integers and renumbers the integers in a WTree.
(8) ANONNAMES uses a State monad to map every name in a Company to new unique names.
(9) DROPCASTS uses a Writer monad to drop casts from Expr while keeping track of how many casts have been dropped.

Both the SYB and SYB3 implementations were optimized with (1) our plugin without type-constant folding (SYB/SYB3 (partial evaluation)) and (2) our entire plugin (SYB/SYB3 (fully optimized)). The data structures are also compiled with all unfoldings exposed, a requirement for inlining. Note that all three data structures were re-defined with a List' data type which replaces the native list type [], since the unfoldings for the Data instance for native lists are not exposed by default. An alternative is to re-build GHC's base library with all unfoldings exposed, so that the Data instance definitions for native lists can be inlined—for the sake of simplifying replication efforts by the community, we did not use this approach.

5.2 Benchmark Suites: Lens (Plated)

We evaluated our optimizations using the Control.Lens.Plated module's own benchmark suite. The suite is centered around the *plate* method. This method, which retrieves the immediate self-similar descendants of a data structure, can be implemented manually, or via two generic implementations that use *gfoldl* from Data and methods from Typeable. The first, *tinplate*, is a straightforward, unoptimized implementation. The second, *uniplate*, is an optimized version that serves as the default for *plate*, and is optimized by pruning branches of the data structure that cannot contain self-similar descendants.

We divide these benchmarks around two categories: Hand, which uses a manual, hand-written implementation of *plate*, and Generic, which uses the library's generic implementations. We use four primary benchmarks applied to basic arithmetic expressions to measure the impact of our optimization techniques. The first, PLATE, benchmarks the *plate* method applied to a list of expressions. (The Generic version uses the *uniplate* implementation.) The second, TINPLATE, benchmarks *tinplate*; in the Hand version, this uses the hand-written *plate* implementation. The third, CLONE, applies a cloned version of *plate* to a list of expressions, and the fourth, PLATEFIB, applies *plate* to a single, large arithmetic expression. The Generic implementations were also optimized with (1) our plugin without type-constant folding (Generic (partial evaluation)) and (2) our entire plugin (Generic (fully optimized)).

5.3 Experimental Results

5.3.1 SYB/SYB3. Fig. 18 shows the experimental results for SYB and SYB3, normalized relative to the Hand version and are displayed on a logarithmic scale to accommodate the large differences between execution times. These results confirm previous results about the poor performance of SYB. The addition of our partial evaluator alone improves the performance of SYB traversals by up to 33×. When also optimized with type-constant folding, the performance improves up to an

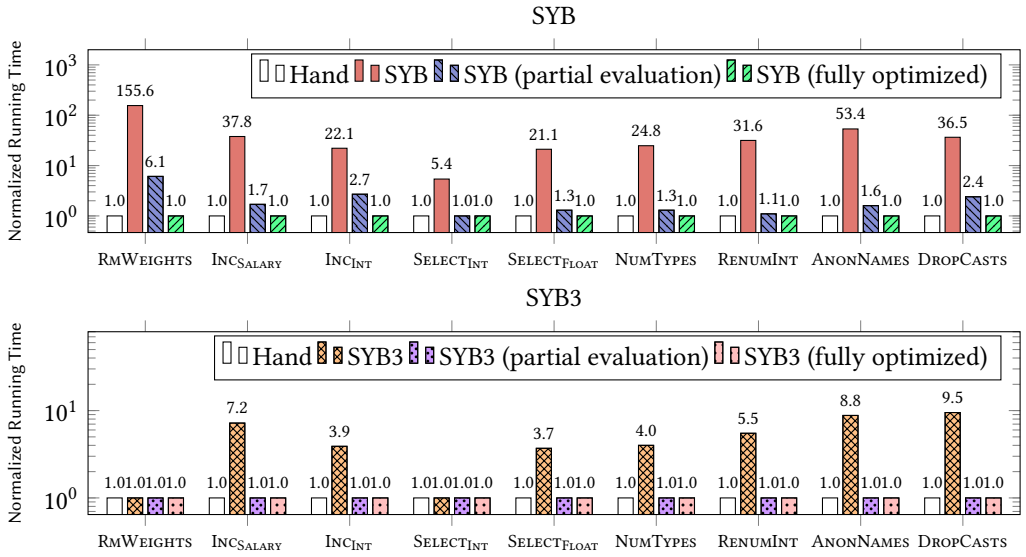


Fig. 18. Benchmarks on SYB and SYB3 traversals. Time axes in log scale. Lower is better.

additional 6×, and the performance of fully optimized SYB traversals is identical to that of their hand-written counterparts. Moreover, the SYB3 benchmarks show that just adding our partial evaluator to the GHC optimization pipeline allows the performance of SYB3 traversals to match that of their hand-written counterparts. (Optimizing SYB3 with the entire plugin achieves the same outcome.) Manual inspection of the generated GHC Core programs from running our optimizations shows that optimized traversals are equivalent to their hand-written counterparts. In fact, the GHC Common Subexpression Elimination pass sometimes merges them.

5.3.2 Lens (Plated). Fig. 19 shows the experimental results for Lens (Plated), normalized relative to the Hand version. Results show an average of 1.63× speed improvement (up to 2.14×) with our plugin. Originally, the partial evaluation and fully optimized versions had no performance differences because one function that invokes *sameTypeRep* was not inlined; adding an `INLINE` pragma to the implementation fixed this issue. Moreover, we initially used only a single type-constant folding pass, which yielded a smaller 1.3× average performance improvement. A manual inspection of the generated GHC Core revealed that some occurrences of *sameTypeRep* with concrete type arguments remained in the residual program. This is because other downstream GHC optimizations, when run after our partial evaluator and one type-constant folding pass, further instantiated *sameTypeRep* with concrete types. To address this, we ran the type-constant folding pass interleaved with the full GHC optimization pipeline, twice, achieving the results in Fig. 19. This means that our plugin actually runs the full GHC optimization pipeline more than once. To isolate the effect of running the GHC optimization pipeline multiple times, we ran all benchmarks with two full passes of the GHC optimization pipeline but without our optimizations. This showed no performance benefit over running the GHC optimization pipeline just once, confirming that our passes were responsible for the performance improvements. Finally, many optimization opportunities were missed due to missing unfoldings for the native list type, an issue we discuss in §6.

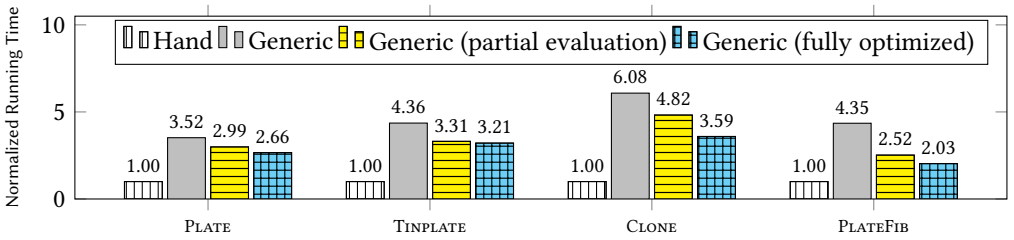


Fig. 19. Benchmarks for Lens (Plated). Lower is better.

Table 2. Compilation time benchmarks. Lower is better. Lens (Plated) benchmarks shows total time since it is defined in only one module.

Benchmark (SYB/SYB3)	**Average Time (ms)**
Hand	24.6
SYB	5.1
SYB3	28.3
SYB (fully optimized)	92.9
SYB3 (fully optimized)	75.8

Benchmark (Lens (Plated))	**Total Time (s)**
Hand	2.13
Generic	1.86
Generic (fully optimized)	2.67

5.4 Compilation Times

We also measured the effects of our plugin on compilation times, and the results are shown in Table 2. Note that we show *average* time to compile each module for SYB/SYB3 benchmarks, while we show *total* time for the `lens` benchmarks since the entire benchmark is defined in one module. Results show that the plugin incurs a 2–4× compilation time overhead. However, for SYB traversals, the overhead is significantly larger at approximately 18.2×. This is because GHC does not optimize SYB at all, whereas our plugin fully optimizes it and generates code whose size is equivalent to the hand-written counterparts. Comparing the compilation times for SYB with our plugin against that of the hand-written benchmarks, the difference is less drastic with a 3.8× average slowdown.

In order to isolate the effects of our optimization techniques for benchmarking, the techniques were implemented as a GHC plugin instead of being integrated directly into GHC. Furthermore, our partial evaluator was naïvely implemented, and each partial evaluation iteration re-traverses the entire program to identify optimization targets, even on portions of the program that have already been analyzed. (The partial evaluator runs up to 151 iterations for these benchmarks.) We believe that with direct integration of our optimization techniques into GHC, in particular, the simplifier and specializer, and with a less naïve implementation, the compilation time overhead can be reduced further.

6 Discussion

6.1 Missing Unfoldings

Our partial evaluator relies heavily on *inlining*, which requires that unfoldings (e.g., of `Data` instances) are exposed in interface files. GHC conservatively avoids exposing unfoldings of recursive definitions, such as the `gmapT` method for recursive types like lists (Fig. 6). This is not typically an issue since GHC's inliner avoids inlining recursive functions anyway, so its unfoldings are not needed. However, our partial evaluator also inlines recursive functions, hence, exposing all unfoldings is crucial. Users can force GHC to expose all unfoldings of a module by compiling it

```
inc :: Int → Int = (+ 1)

data T α = E | C (T (α, α))

inc_SYB, inc_Hand :: T Int → T Int

inc_SYB = everywhere (mkT inc)

inc_Hand = aux inc where
  aux :: ∀α.(α → α) → T α → T α
  aux f E = E
  aux f (C a) = C (aux (λ(x, y). (f x, f y)) a)
```

Fig. 20. Example of a program that cannot be optimized by our partial evaluator due to polymorphic recursion. The inc_{SYB} function is an SYB traversal over a polymorphically recursive data type T, and inc_{Hand} is its hand-written moral equivalent.

```
incProgressive_SYB, incProgressive_Hand :: [Int] → [Int]

incProgressive_SYB = aux id where
    aux :: (∀α. Data α ⇒ α → α) →
           (∀α. Data α ⇒ α → α)
    aux f = f ∘ gmapT (aux (mkT inc ∘ f))

incProgressive_Hand = aux id where
    aux :: ∀α. (α → Int) → [α] → [Int]
    aux _ [] = []
    aux f (x : xs) = (inc ∘ f) x : aux (inc ∘ f) x
```

Fig. 21. Example of a program that cannot be optimized by our partial evaluator due to unbounded term growth. The $incProgressive_{SYB}$ function is an SYB-style traversal that composes the traversal intent with *mkT inc* at every recursive call, and $incProgressive_{Hand}$ is its hand-written moral equivalent. The *inc* function is defined in Fig. 20.

with the `-fexpose-all-unfoldings` flag. (If the module whose unfoldings are to be exposed are in GHC's `base` library, re-compiling GHC might be required.) Our implementation also detects and warns when critical unfoldings are missing, alerting users to rectify this.

6.2 Non-termination

Unrestricted repeated inlining of recursive functions risks non-termination during compilation [28]. Since our partial evaluator relies on iterative inlining to expose optimization opportunities, we use memoization so that partially evaluating recursive function applications can terminate. Specifically, we memoize expressions consisting of the function being inlined, its type arguments, its dictionary arguments, and its polymorphic function arguments. Consequently, the termination of our algorithm relies on the set of these expressions being finite.

When the set of memoization entries is infinite, unrestricted repeated partial evaluation will not terminate. We identify two scenarios where this happens:

(1) *Polymorphic Recursion* [45]. The inc_{SYB} function shown in Fig. 20 is an SYB traversal over a polymorphically recursive datatype T α. Unrestricted and repeated partial evaluation of inc_{SYB} will encounter applications of *go* (the static-argument transformation of *everywhere*) with infinitely many type arguments, i.e., T Int, T (Int, Int), T ((Int, Int), (Int, Int)), and so on. Since the type argument changes at every recursive step, the memoization table never encounters a repeat hit.

(2) *Divergent Term-Level Specialization.* The $incProgressive_{SYB}$ function shown in Fig. 21 traverses a standard list but modifies its behavior at every recursive step by composing the traversal's intent with *mkT inc*. Here, unrestricted and repeated partial evaluation of $incProgressive_{SYB}$ will encounter applications of *aux* with infinitely many function arguments, i.e., *id*, *mkT inc* ∘ *id*, *mkT inc* ∘ (*mkT inc* ∘ *id*), and so on. While the types (and therefore, dictionary arguments) remain constant, the function term grows structurally indefinitely, and memoization similarly fails to encounter a hit.

Ideally, inc_{SYB} and $incProgressive_{SYB}$ would be transformed into their "hand-written" moral equivalents, inc_{Hand} and $incProgressive_{Hand}$, shown in Fig. 20 and Fig. 21, respectively. Such transformations require altering the type signature and internal structure of the function. However, this

is an approach that our partial evaluator is unable to do, since it performs only type-preserving inlining and simplification. This is a limitation of our work.

To ensure compiler stability in these cases, we adopt a standard safeguard similar to the "tick" limit in GHC's simplifier [28]. Our plugin enforces a user-configurable iteration limit on the partial evaluator. When this limit is reached, the partial evaluator aborts, preventing an infinite compilation loop. While this ensures termination, it results in suboptimal code for these cases, often manifesting as code bloat due to the partially unrolled, but ultimately abandoned, specialization attempts. An avenue for future work is to address these cases via heuristics to detect unproductive optimizations and rollback transformations, or by performing more aggressive structural transformations.

6.3 Integration and Generalizability

Our optimization passes performs standard transformations that are already done by GHC. Our work demonstrates how a *targeted* application of these transformations unlocks significant optimizations for problematic rank-2 polymorphic code. We believe this makes it easy to incorporate into GHC. In addition, the design patterns we exploit are not unique to SYB, and we have shown in §5 that they can be applied equally to SYB3 and a portion of `lens`. Furthermore, the abstractions that rank-2 polymorphism enables are elegant and widely used by the Haskell community, as seen by usage numbers for SYB and `lens`. Run-time type comparisons via `Typeable` and `TypeRep` or similar mechanisms are also used for dynamically typed exceptions [42], Cloud Haskell [12] and meta-programming [19, 43]. Future work can integrate our optimization passes into GHC and profile running time and optimization time on other libraries that use rank-2 polymorphism extensively. We also plan to extend type-constant folding to treat other primitive type operations uniformly to benefit broader use cases.

6.4 Stability of Optimization

An important consideration for compiler plugins is their sensitivity to the host compiler's evolution. Our optimization relies on specific syntactic patterns and internal API names within GHC, creating a dependency on the behavior of the standard simplification pipeline. For example, our analysis expects applications of the form $f\ g$ where f is rank-2 polymorphic and g is its polymorphic argument. Consider the common idiom $f\ \$\ g$. Currently, GHC's simplifier inlines the application operator ($) prior to partial evaluation, exposing the direct function application we expect. However, if a future GHC version were to delay or alter the inlining of $, our plugin instead only encounters the expression ($)\ $f\ g$, obscuring the target call site and preventing the optimization from firing. Similarly, type-constant folding relies on identifying specific GHC internal functions, such as *sameTypeRep*. If GHC modifies its `Typeable` implementation to use different primitives, our plugin would fail to identify and statically evaluate run-time type-equality tests.

Inspection testing [5] can address this potential fragility and ensure that the optimization fires reliably. Inspection testing allows library authors to assert properties about the generated Core at compile-time, causing the build to fail if optimizations do not trigger as expected. In the context of our plugin, users can define inspection obligations to verify that the final generated code is free of specific artifacts that signal a missed optimization. Specifically, a successful application of our partial evaluator should eliminate (1) rank-2 polymorphic functions like *gmapT*, which should be fully specialized and inlined, and (2) runtime type representations, i.e., uses of `TypeRep` (and associated equality checks), which should be statically resolved and erased. By asserting the absence of these terms in the final Core, inspection testing serves as a regression detection mechanism. If GHC evolves in a way that disrupts our plugin, whether through pipeline changes or API refactoring, the inspection tests will fail, alerting maintainers to the need for updates.

7 Related Work

7.1 Compiler Optimizations and Partial Evaluation

The GHC inliner [28] and specializer [26] are the foundation of our approach. Inlining unfolds definitions and simplifies expressions, revealing optimization opportunities. Together with the specializer, these can also remove the overhead of class-dictionary passing from using Haskell type classes. Our partial evaluator does not use new transformations, but strategically applies standard transformations to break the optimization stalemate for a specific, important class of programs. Constant folding in Haskell can be performed with rewrite rules [47]. However, rewrite rules cannot express our type-constant folding pass which involves an analysis on types.

Unsurprisingly, our partial evaluator is inspired by a long line of work on partial evaluation [16, 27, 41]. The binding-time analysis performed by our partial evaluator uses types and variable-binding information to decide which part of the code should be evaluated at compile time. Similar to Futamura [16] we use memoization to avoid optimizing expressions more than once; since we focus solely on enabling class-dictionary specialization, our optimizer targets only argument positions that involve types, dictionaries and polymorphic arguments, making the memoization simpler. Our partial evaluator can also be viewed as a limited form of *supercompilation* [57]. Similar to supercompilers, our partial evaluator can cause some code explosion [29], though, from our experiments, the code explosion exhibited by our partial evaluator is the effect of expanding/generating optimized, specialized code, as intended. Finally, to the best of our knowledge, our partial evaluator is the first that specifically targets rank-2 polymorphism.

7.2 Monomorphization

Several aspects of our work are closely related to *monomorphization*, a technique for implementing polymorphism in languages like C++ and Rust. Monomorphization specializes polymorphic functions for each concrete type instantiation, eliminating the overhead of uniform data representations (e.g., boxing). By eliminating dictionary passing in rank-2 polymorphic code, our partial evaluator achieves a similar effect, for instance, in SYB-style traversals, the partial evaluator generates specialized versions of the traversal for each encountered sub-term.

This relationship between monomorphization and rank-n polymorphism has been explored in recent literature. Griesemer et al. [18] formalizes the translation from Featherweight Generic Go to Featherweight Go by tracking type instantiations, while Lutze et al. [37] describes a monomorphization technique based on a type-flow analysis. Both approaches generalize to rank-n types and can detect non-monomorphizability caused by polymorphic recursion.

Our work differs from monomorphization in objectives and techniques. Our primary goal is to eliminate the overhead of dictionary-based overloading instead of providing a total compilation scheme for polymorphism. Typical monomorphization algorithms require a global, whole-program analysis. On the other hand, to minimize the impact on compilation time, our techniques operate locally and modularly, unfolding and specializing definitions only when demanded. Our residual programs can also remain polymorphic, allowing for partial specialization even when complete type information is unavailable.

7.3 Performant Generics

Our work was inspired by an attempt to develop a practical optimization for SYB-style- and generic programs. Magalhães et al. [39] ran benchmarks showing how the GHC optimizer—where inlining is done via the GHC Simplifier and does a lot of the heavy lifting—can optimize some generic programming libraries, although SYB remained around 10× slower than the handwritten version due to the optimization stalemate induced by the structure of SYB combinators. Magalhães [40]

showed that inlining via `INLINE` pragmas and rewrite rules [47] completely eliminates the overhead incurred by the `generic-deriving` library [38]. Alimarine and Smetsers [4] proved, by using typing to predict the structure of the result of a symbolic computation, that *symbolic evaluation* is able to eliminate a large amount of overhead in generic programs. Adams et al. [2, 3] then developed a domain-specific optimization of SYB-style programs by eliminating *undesirable types* also via partial evaluation, through a combination of inlining and symbolic evaluation based on domain-specific knowledge of SYB traversals. Just like our work, this technique can completely recover the performance of hand-written implementations. However, unlike these works, ours successfully optimizes SYB3 and generalizes to rank-2 polymorphism.

Other approaches sidestep the optimization problem for generic programs by designing different generic-programming libraries. Generic programming in Haskell follows two traditions: induction on the structure types [8–10, 20, 21, 23, 25, 46, 59], and the the generic fold of SYB [33–35]. These approaches were unified by the spine view [22]. While earlier works on generic programming focus on expressiveness, concision, ease-of-use, extensibility and modularity of generic-programming systems, more recent systems like Uniplate [44] and Alloy [6] also take performance into consideration. *Metaprogramming* has also been used to develop performant generics. Template Your Boilerplate [1] exposes a similar interface to SYB, except that the combinators generate the intended traversals at compile time via Template Haskell [54]. Yallop [60, 61] developed a port of SYB to MetaOCaml [30] that uses multi-stage programming to eliminate most of the overhead of SYB, and performs fixed-point elimination for selective traversals or branch pruning over mutually recursive traversals. Pickering et al. [50] also employs multi-stage programming techniques on `generics-sop` to develop `staged-sop` using Typed Template Haskell [49]. The `generic-lens` library [31], based on `lens`, exposes an API that supports SYB-style traversals whose performance matches hand-written equivalents.

8 Conclusion

We have presented a solution to the problem of enabling class-dictionary specialization in the presence of rank-2 polymorphism. Our technique breaks the optimization stalemate in which GHC's specializer and inliner are thwarted. The core of our approach is a partial evaluator that strategically forces the unfolding of these functions, using memoization to recover recursion and successfully optimize typical use cases. This process instantiates polymorphic function applications with concrete type and dictionary arguments, enabling class-dictionary specialization. We complement this with a type-constant folding pass that eliminates run-time type-equality tests.

The effectiveness of our method is demonstrated by its application to three widely used libraries that face this performance bottleneck: SYB, SYB3 and `lens`. Our optimization completely eliminates SYB and SYB3's performance overhead, transforming generic traversals into code that is operationally equivalent to hand-written, non-generic versions, achieving speedups of up to 155×, and improved generic implementations of certain fragments of the `lens` library by up to 2.1×. Our method allows developers to retain the conceptual simplicity and concision of libraries that use rank-2 polymorphism without sacrificing run-time performance. We believe the integration of this technique into compilers can help improve the viability of sophisticated abstractions in production-scale software.

Acknowledgments

This work is supported by the Singapore Ministry of Education under Project Number T1 251RES2422.

Data-Availability Statement

This paper is accompanied by an artifact online at:

- Zenodo: https://zenodo.org/records/18502345 (doi:10.5281/zenodo.18502345), which also contains a virtual machine for reproducing the results in this paper [14].
- GitHub: https://github.com/plilab/class-spec-rank2, commit hash prefix `3046fd9`.
- Software Heritage: https://archive.softwareheritage.org/swh:1:dir:642d137dd0c0b7ba724aa2dd26dd0fa7c1154aa9 (SWHID: swh:1:dir:642d137dd0c0b7ba724aa2dd26dd0fa7c1154aa9).

References

[1] Michael D. Adams and Thomas M. DuBuisson. 2012. Template your boilerplate: using template haskell for efficient generic programming. In *Proceedings of the 2012 Haskell Symposium* (Copenhagen, Denmark) *(Haskell '12)*. Association for Computing Machinery, New York, NY, USA, 13–24. doi:10.1145/2364506.2364509

[2] Michael D. Adams, Andrew Farmer, and José Pedro Magalhães. 2014. Optimizing SYB is easy!. In *Proceedings of the ACM SIGPLAN 2014 Workshop on Partial Evaluation and Program Manipulation* (San Diego, California, USA) *(PEPM '14)*. Association for Computing Machinery, New York, NY, USA, 71–82. doi:10.1145/2543728.2543730

[3] Michael D. Adams, Andrew Farmer, and José Pedro Magalhães. 2015. Optimizing SYB traversals is easy! *Science of Computer Programming* 112 (2015), 170–193. doi:10.1016/j.scico.2015.09.003 Selected and extended papers from Partial Evaluation and Program Manipulation 2014.

[4] Artem Alimarine and Sjaak Smetsers. 2004. Optimizing Generic Functions. In *Mathematics of Program Construction*, Dexter Kozen (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 16–31. doi:10.1007/978-3-540-27764-4_3

[5] Joachim Breitner. 2018. A promise checked is a promise kept: inspection testing. In *Proceedings of the 11th ACM SIGPLAN International Symposium on Haskell* (St. Louis, MO, USA) *(Haskell 2018)*. Association for Computing Machinery, New York, NY, USA, 14–25. doi:10.1145/3242744.3242748

[6] Neil C.C. Brown and Adam T. Sampson. 2009. Alloy: fast generic transformations for Haskell. In *Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell* (Edinburgh, Scotland) *(Haskell '09)*. Association for Computing Machinery, New York, NY, USA, 105–116. doi:10.1145/1596638.1596652

[7] Manuel Chakravarty, Gabriel Ditu, and Roman Leshchinskiy. 2009. Instant Generics: Fast and Easy.

[8] James Cheney and Ralf Hinze. 2002. A lightweight implementation of generics and dynamics. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell* (Pittsburgh, Pennsylvania) *(Haskell '02)*. Association for Computing Machinery, New York, NY, USA, 90–104. doi:10.1145/581690.581698

[9] Bruno C. d. S. Oliveira, Ralf Hinze, and Andres Löh. 2007. *Extensible and Modular Generics for the Masses* (1 ed.). Intellect, 199–216. doi:10.2307/j.ctv36xvknz.15

[10] Edsko de Vries and Andres Löh. 2014. True sums of products. In *Proceedings of the 10th ACM SIGPLAN Workshop on Generic Programming* (Gothenburg, Sweden) *(WGP '14)*. Association for Computing Machinery, New York, NY, USA, 83–94. doi:10.1145/2633628.2633634

[11] Richard A. Eisenberg. 2015. *System FC, as implemented in GHC*. Technical Report. University of Pennsylvania.

[12] Jeff Epstein, Andrew P. Black, and Simon Peyton-Jones. 2011. Towards Haskell in the cloud. In *Proceedings of the 4th ACM Symposium on Haskell* (Tokyo, Japan) *(Haskell '11)*. Association for Computing Machinery, New York, NY, USA, 118–129. doi:10.1145/2034675.2034690

[13] Andrew Farmer, Andy Gill, Ed Komp, and Neil Sculthorpe. 2012. The HERMIT in the machine: a plugin for the interactive transformation of GHC core language programs. In *Proceedings of the 2012 Haskell Symposium* (Copenhagen, Denmark) *(Haskell '12)*. Association for Computing Machinery, New York, NY, USA, 1–12. doi:10.1145/2364506.2364508

[14] Yong Qi Foo and Michael D. Adams. 2026. *Class-Dictionary Specialization With Rank-2 Polymorphic Functions (Artifact)*. doi:10.5281/zenodo.18502345

[15] J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. 2007. Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM Trans. Program. Lang. Syst.* 29, 3 (May 2007), 17–es. doi:10.1145/1232420.1232424

[16] Yoshihiko Futamura. 1983. Partial computation of programs. In *RIMS Symposia on Software Science and Engineering*, Eiichi Goto, Koichi Furukawa, Reiji Nakajima, Ikuo Nakata, and Akinori Yonezawa (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–35. doi:10.1007/3-540-11980-9_13

[17] GHC Team. 2024. Glasgow Haskell Compiler—The Glasgow Haskell Compiler. https://www.haskell.org/ghc. Accessed: 18 September 2024.

[18] Robert Griesemer, Raymond Hu, Wen Kokke, Julien Lange, Ian Lance Taylor, Bernardo Toninho, Philip Wadler, and Nobuko Yoshida. 2020. Featherweight go. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 149 (Nov. 2020), 29 pages. doi:10.1145/3428217

[19] Louis-Julien Guillemette and Stefan Monnier. 2008. A type-preserving compiler in Haskell. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming* (Victoria, BC, Canada) *(ICFP '08)*. Association for Computing Machinery, New York, NY, USA, 75–86. doi:10.1145/1411204.1411218

[20] Ralf Hinze. 2002. Polytypic values possess polykinded types. *Science of Computer Programming* 43, 2 (2002), 129–159. doi:10.1016/S0167-6423(02)00025-4 Mathematics of Program Construction (MPC 2000).

[21] Ralf Hinze. 2004. Generics for the masses. In *Proceedings of the Ninth ACM SIGPLAN International Conference on Functional Programming* (Snow Bird, UT, USA) *(ICFP '04)*. Association for Computing Machinery, New York, NY, USA, 236–243. doi:10.1145/1016850.1016882

[22] Ralf Hinze, Andres Löh, and Bruno C. d. S. Oliveira. 2006. "Scrap your boilerplate" reloaded. In *Proceedings of the 8th International Conference on Functional and Logic Programming* (Fuji-Susono, Japan) *(FLOPS'06)*. Springer-Verlag, Berlin, Heidelberg, 13–29. doi:10.1007/11737414_3

[23] Stefan Holdermans, Johan Jeuring, Andres Löh, and Alexey Rodriguez. 2006. Generic Views on Data Types. In *Mathematics of Program Construction*, Tarmo Uustalu (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 209–234. doi:10.1007/11783596_14

[24] Industrial Haskell Group. 2024. Hackage: Total Downloads. https://hackage.haskell.org/packages/browse?terms=generic. Accessed: 18 September 2024.

[25] Patrik Jansson and Johan Jeuring. 1997. PolyP—a polytypic programming language extension. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Paris, France) *(POPL '97)*. Association for Computing Machinery, New York, NY, USA, 470–482. doi:10.1145/263699.263763

[26] Mark P. Jones. 1995. Dictionary-free overloading by partial evaluation. *Lisp and Symbolic Computation* 8 (1995), 229–248. doi:10.1007/BF01019005

[27] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. 1993. *Partial Evaluation and Automatic Program Generation*. Prentice Hall International.

[28] Simon L. Peyton Jones and Simon Marlow. 2002. Secrets of the Glasgow Haskell Compiler inliner. *Journal of Functional Programming* 12 (2002), 393 – 434. doi:10.1017/S0956796802004331

[29] Peter A. Jonsson and Johan Nordlander. 2011. Taming code explosion in supercompilation. In *Proceedings of the 20th ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation* (Austin, Texas, USA) *(PEPM '11)*. Association for Computing Machinery, New York, NY, USA, 33–42. doi:10.1145/1929501.1929507

[30] Oleg Kiselyov. 2014. The Design and Implementation of BER MetaOCaml. In *Functional and Logic Programming*, Michael Codish and Eijiro Sumii (Eds.). Springer International Publishing, Cham, 86–102. doi:10.1007/978-3-319-07151-0_6

[31] Csongor Kiss, Matthew Pickering, and Nicolas Wu. 2018. Generic deriving of generic traversals. *Proc. ACM Program. Lang.* 2, ICFP, Article 85 (July 2018), 30 pages. doi:10.1145/3236780

[32] Edward A. Kmett. 2015. `lens`. https://hackage.haskell.org/package/lens. Accessed: 18 September 2024.

[33] Ralf Lämmel and Simon Peyton Jones. 2003. Scrap your boilerplate: a practical design pattern for generic programming. In *Proceedings of the 2003 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation* (New Orleans, Louisiana, USA) *(TLDI '03)*. Association for Computing Machinery, New York, NY, USA, 26–37. doi:10.1145/604174.604179

[34] Ralf Lämmel and Simon Peyton Jones. 2004. Scrap more boilerplate: reflection, zips, and generalised casts. In *Proceedings of the Ninth ACM SIGPLAN International Conference on Functional Programming* (Snow Bird, UT, USA) *(ICFP '04)*. Association for Computing Machinery, New York, NY, USA, 244–255. doi:10.1145/1016850.1016883

[35] Ralf Lämmel and Simon Peyton Jones. 2005. Scrap your boilerplate with class: extensible generic functions. In *Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming* (Tallinn, Estonia) *(ICFP '05)*. Association for Computing Machinery, New York, NY, USA, 204–215. doi:10.1145/1086365.1086391

[36] Daniel Leivant. 1983. Polymorphic type inference. In *Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (Austin, Texas) *(POPL '83)*. Association for Computing Machinery, New York, NY, USA, 88–98. doi:10.1145/567067.567077

[37] Matthew Lutze, Philipp Schuster, and Jonathan Immanuel Brachthäuser. 2025. The Simple Essence of Monomorphization. *Proc. ACM Program. Lang.* 9, OOPSLA1, Article 116 (April 2025), 27 pages. doi:10.1145/3720472

[38] José Pedro Magalhães. 2012. *Less Is More: Generic Programming Theory and Practice*. Ph. D. Dissertation. Universiteit Utrecht.

[39] José Pedro Magalhães, Stefan Holdermans, Johan Jeuring, and Andres Löh. 2010. Optimizing generics is easy!. In *Proceedings of the 2010 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation* (Madrid, Spain) *(PEPM '10)*. Association for Computing Machinery, New York, NY, USA, 33–42. doi:10.1145/1706356.1706366

[40] José Pedro Magalhães. 2013. Optimisation of Generic Programs Through Inlining. In *Implementation and Application of Functional Languages*, Ralf Hinze (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 104–121. doi:10.1007/978-3-642-41582-1_7

[41] Zohar Manna. 2003. *Mathematical Theory of Computation*. Dover Publications, Inc., USA.

[42] Simon Marlow, Simon Peyton Jones, Andrew Moran, and John Reppy. 2001. Asynchronous exceptions in Haskell. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation* (Snowbird, Utah, USA) *(PLDI '01)*. Association for Computing Machinery, New York, NY, USA, 274–285. doi:10.1145/378795.378858

[43] Trevor L. McDonell, Manuel M. T. Chakravarty, Vinod Grover, and Ryan R. Newton. 2015. Type-safe runtime code generation: accelerate to LLVM. In *Proceedings of the 2015 ACM SIGPLAN Symposium on Haskell* (Vancouver, BC, Canada) *(Haskell '15)*. Association for Computing Machinery, New York, NY, USA, 201–212. doi:10.1145/2804302.2804313

[44] Neil Mitchell and Colin Runciman. 2007. Uniform boilerplate and list processing. In *Proceedings of the ACM SIGPLAN Workshop on Haskell Workshop* (Freiburg, Germany) *(Haskell '07)*. Association for Computing Machinery, New York, NY, USA, 49–60. doi:10.1145/1291201.1291208

[45] Alan Mycroft. 1984. Polymorphic type schemes and recursive definitions. In *International Symposium on Programming*, M. Paul and B. Robinet (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 217–228. doi:10.1007/3-540-12925-1_41

[46] Ulf Norell and Patrik Jansson. 2003. Polytypic Programming in Haskell. In *Implementation of Functional Languages*, Phil Trinder, Greg J. Michaelson, and Ricardo Peña (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 168–184. doi:10.1007/978-3-540-27861-0_11

[47] Simon Peyton Jones, Andrew Tolmach, and Tony Hoare. 2001. Playing by the rules: rewriting as a practical optimisation technique in GHC. In *2001 Haskell Workshop* (2001 haskell workshop ed.). ACM SIGPLAN. https://www.microsoft.com/en-us/research/publication/playing-by-the-rules-rewriting-as-a-practical-optimisation-technique-in-ghc/

[48] Simon Peyton Jones, Stephanie Weirich, Richard A. Eisenberg, and Dimitrios Vytiniotis. 2016. *A Reflection on Types*. Springer International Publishing, Cham, 292–317. doi:10.1007/978-3-319-30936-1_16

[49] Matthew Pickering, Andres Löh, and Nicolas Wu. 2020. A Specification for Typed Template Haskell. https://mpickering.github.io/papers/specification-typed-th.pdf.

[50] Matthew Pickering, Andres Löh, and Nicolas Wu. 2020. Staged sums of products. In *Proceedings of the 13th ACM SIGPLAN International Symposium on Haskell* (Virtual Event, USA) *(Haskell 2020)*. Association for Computing Machinery, New York, NY, USA, 122–135. doi:10.1145/3406088.3409021

[51] Alexey Rodriguez, Johan Jeuring, Patrik Jansson, Alex Gerdes, Oleg Kiselyov, and Bruno C. d. S. Oliveira. 2008. Comparing libraries for generic programming in Haskell. In *Proceedings of the First ACM SIGPLAN Symposium on Haskell* (Victoria, BC, Canada) *(Haskell '08)*. Association for Computing Machinery, New York, NY, USA, 111–122. doi:10.1145/1411286.1411301

[52] Neil Sculthorpe, Andrew Farmer, and Andy Gill. 2013. The HERMIT in the Tree. In *Implementation and Application of Functional Languages*, Ralf Hinze (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 86–103. doi:10.1007/978-3-642-41582-1_6

[53] Neil Sculthorpe, Nicolas Frisby, and Andy Gill. 2014. KURE: A Haskell-Embedded Strategic Programming Language with Custom Closed Universes. *Journal of Functional Programming* 24, 4 (2014), 434–473. doi:10.1017/S0956796814000185

[54] Tim Sheard and Simon Peyton Jones. 2002. Template meta-programming for Haskell. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell* (Pittsburgh, Pennsylvania) *(Haskell '02)*. Association for Computing Machinery, New York, NY, USA, 1–16. doi:10.1145/581690.581691

[55] Bryan O' Sullivan. 2009. `criterion`. https://hackage.haskell.org/package/criterion. Accessed: 25 September 2025.

[56] Martin Sulzmann, Manuel M. T. Chakravarty, Simon Peyton Jones, and Kevin Donnelly. 2007. System F with type equality coercions. In *Proceedings of the 2007 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation* (Nice, Nice, France) *(TLDI '07)*. Association for Computing Machinery, New York, NY, USA, 53–66. doi:10.1145/1190315.1190324

[57] Valentin F. Turchin. 1986. The concept of a supercompiler. *ACM Trans. Program. Lang. Syst.* 8, 3 (June 1986), 292–325. doi:10.1145/5956.5957

[58] Alexey Rodriguez Yakushev. 2009. *Towards Getting Generic Programming Ready for Prime Time*. Ph. D. Dissertation. Utrecht University.

[59] Alexey Rodriguez Yakushev, Stefan Holdermans, Andres Löh, and Johan Jeuring. 2009. Generic programming with fixed points for mutually recursive datatypes. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming* (Edinburgh, Scotland) *(ICFP '09)*. Association for Computing Machinery, New York, NY, USA, 233–244. doi:10.1145/1596550.1596585

[60] Jeremy Yallop. 2016. Staging generic programming. In *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation* (St. Petersburg, FL, USA) *(PEPM '16)*. Association for Computing Machinery, New York, NY, USA, 85–96. doi:10.1145/2847538.2847546

[61] Jeremy Yallop. 2017. Staged generic programming. *Proc. ACM Program. Lang.* 1, ICFP, Article 29 (Aug 2017), 29 pages. doi:10.1145/3110273

Received 2025-10-10; accepted 2026-02-17