5-2009

The Design and Implementation of a Safe, Lightweight Haskell Compiler

Timothy Jan Chevalier
Portland State University

The Design and Implementation of a Safe, Lightweight Haskell Compiler

Timothy Jan Chevalier

Portland State University
Dept. of Computer Science
P.O. Box 751
Portland, OR 97207
tjc@cs.pdx.edu

Abstract. Typed programming languages offer safety guarantees that help programmers write correct code, but typical language implementations offer no proof that source-level guarantees extend to executable code. Moreover, typical implementations link programs with unsafe runtime system (RTS) code. I present a compiler for the functional language Haskell that preserves some of the properties of Haskell's type system. The soundness proof for the combination of the compiler and a verified RTS requires a proof that the compiler emits code that cooperates correctly with the RTS. In particular, the latter proof must address the boundary between the user program and the garbage collector. In this paper, I introduce a minimalist intermediate language type system that ensures that well-typed programs cannot make the garbage collector go wrong. I also discuss compilation strategies that allow simplifying the RTS. The work I present in this paper yields a simple and safe compilation system.

1   Introduction

When programmers program in a statically typed language, they feel confident that the compiler will reject programs that would do nonsensical things if executed. Such languages offer a promise that "well-typed programs won't go wrong": they have type systems that are sound (relative to a formal semantics), meaning that running a type-correct program only executes operations that make sense [1]. The type system defines what it means to "make sense".

Skeptical programmers feel less confident: they know that the source language's type system cannot prevent the compiler from transforming their program into one with a different meaning, or from linking that program with buggy runtime system (RTS) code to create a faulty executable. Typical compilers for typed languages have no safety proofs, generate untyped code (for example, C or assembly language) and link compiled code with RTS code written in unsafe languages such as C. A bug in the compiler or RTS could destroy the relationship between safety properties in source code and in executable code.

Programmers can regain some of their lost confidence when the RTS and compiler are proven to be safe. The benefits of such a proof are contingent on the correctness of the boundary between the RTS and compiled programs. More specifically, in this paper I look at the boundary between programs and one component of the RTS: the garbage collector. I focus on compiling Haskell, a typed functional language [2]. The Glasgow Haskell Compiler (GHC) [3] is a widely used implementation of Haskell. I use GHC's front-end to obtain code for my compiler, and I use GHC as a baseline for comparison.

Garbage collector bugs are just one example of many potential RTS flaws that could negate source-level safety guarantees. But they are an important one. For example, as recently as March 2009, a security researcher uncovered a garbage collector bug in the Mozilla Firefox Web browser that could allow an attacker to run arbitrary code on a victim's computer [4]. Regardless of how common such bugs are, for some applications, one crash or unsafe memory access is too many. Users of current state-of-the-art compilers, even for safe languages, have no way to guarantee that a compiler or RTS bug will not cause such behavior in their code. For example, GHC's RTS consists of over 30,000 lines of C code [5] [6], and a cursory search of the GHC bug tracker revealed several reports of collector bugs that caused GHC-compiled programs to crash [7].

Verifying the collector narrows this safety gap. The correctness of a collector requires a well-defined interface between compiled programs and the collector. The problem I address in this paper is how a compiler for Haskell can ensure that compiled code uses this interface correctly (given a verified collector).

Contributions In this work, I develop a lightweight compiler for Haskell that generates code for the PowerPC architecture that is statically guaranteed to make proper use of the garbage collector. When I say it is lightweight, I mean that I have tried to minimize the theoretical and implementation effort necessary to get the compiler working. This reduction in implementation effort has a performance cost; later in this paper, I discuss my efforts to determine that cost.

To frame the problem, I chose GHC and Compcert as the endpoints for the compiler; the choices of source and target language constrain some of the other design choices. GHC is a widely used, highly optimizing compiler for Haskell, and I take advantage of its front-end to obtain code in Core, GHC's intermediate language [3]. Compcert is a C compiler that has been proven to preserve the semantics of source programs; I take advantage of its back-end, which accepts code in the Cminor language and generates PowerPC assembly code [8].

My main contribution in this paper is the first set of performance results (that I know of) for a safety-preserving compiler for a functional language. These results suggest how much runtime overhead we must pay in exchange for increased confidence in program safety. Novel aspects of my compiler include:

- Its simple and lightweight nature. Other work on safety-preserving compilation involves complex type systems and heavyweight formalisms. However, such work has not yet produced a production-quality compiler for a functional language (see Section 2). My work is a different path to a complete,

high-assurance compiler for Haskell. (The compiler does not support all existing Haskell programs, because GHC compiles certain programs using features exposed in Core that are not supported by my compiler, such as foreign function calls and certain primitive operations.)

– The combination of a simple static type system with dynamic checks to support safety preservation. While other systems combine static and dynamic checks (see Section 4), the idea of using such a combination to simplify a safety-preserving compiler has not been studied, as far as I know.[1]

– The use of a strict intermediate language. Haskell is a language with call-by-need semantics: that is, it is lazy [2]. In a lazy language, every argument to a given function invocation is evaluated either zero or one times. Expressions are only evaluated if their values are demanded: that is, if a primitive operation requires their value. In contrast, in a strict language, if a function invocation is executed, then each of its arguments is evaluated exactly once. GHC, the dominant implementation of Haskell, maintains call-by-need semantics down to the lowest level of code and supports laziness in its RTS. In his account of the abstract machine on which GHC's RTS is based, Peyton Jones described handling updates (a key aspect of laziness) as "much the trickiest part of the Spineless Tagless G-machine" [9]. By compiling Haskell to a strict intermediate language, I obviate the need to support laziness in the RTS. Omitting this support can only make the RTS smaller and thus easier to verify. This compilation technique is not new, but has been relatively neglected (see Section 5).

Methods I designed a series of typed intermediate languages for the compiler middle-end; the type systems for these languages capture just the property I wish to check statically. I implemented the compiler as a series of transformations through these intermediate languages. I formalized the languages' static and dynamic semantics and proved that the compiler's transformations preserve safety. I ran the code that the compiler generates to test that it works (for the subset of Core that I have chosen to support). In order to determine whether my approach performs acceptably, I measured the performance of programs compiled by my system and compared these with results obtained from an existing mainstream compiler. I define acceptable performance as that of unoptimized GHC-compiled code. Because people use unoptimized code for practical work (for example, for day-to-day development when they do not want to wait for long compiles), a system that generates code that performs about as well as unoptimized code is good enough to be useful for at least some purposes. Because the additional performance cost of using my system arises partially from the dynamic safety checks it inserts, I measured the performance of code generated by my compiler both with these checks enabled, and with the checks disabled. Because my naïve implementation performed unacceptably badly, I tried to determine which optimizations needed to be implemented to achieve better perfor-

---

[1] However, the safety preservation proof for the entire compiler is not yet complete.

mance. I implemented these optimizations, most of which are standard compiler optimizations (see Section 7).

After implementing these optimizations, I found that my compiler now generates code that runs between 2 and 9 times slower than optimized GHC-compiled programs, for a small set of benchmarks (for more details, see Section 7). Safety checks add overhead of 5-18%. The optimizations also need to be proven safe in order to show that the same safety proofs I have done for the naïve compiler apply to the entire optimized system. However, the optimized system is still guaranteed to be safe in the sense that it typechecks code after every optimization pass, and well-typed code cannot execute unsafe operations.

Paper overview Section 2 provides necessary background and definitions. Section 3 presents the compilation pipeline. Rather than discussing the entire pipeline in detail, I focus on two key aspects: the simple type system (Section 4) and the handling of laziness (Section 5). Section 6 summarizes correctness results, and Section 7 gives performance results. Section 8 concludes. I discuss related work throughout the paper rather than in a separate section.

2   Garbage collection and safer compilation

Pointers, unboxed values, boxed values I define a *pointer* as a valid address that the running program can dereference to obtain a record that it allocated earlier during execution and has not freed, or a field of such a record. Program state (registers, the stack, and fields of heap-allocated records) can contain boxed or unboxed data. A value is *boxed* if it is represented as a pointer to heap-allocated data, and a value is *unboxed* if it is a self-contained bit pattern. By extension, a type is called a *boxed type* if its values have a boxed representation, and an *unboxed type* if its values have an unboxed representation [10].

Garbage collection Garbage collection is the automatic reclamation of heap storage space that is no longer needed by a running program [11]. In garbage collection jargon, the user application program that does useful work is called the *mutator* (as from the collector's perspective, all the user program does is allocate and alter memory that the collector is burdened with managing). A *tracing* garbage collector conservatively approximates the graph of heap data currently accessible by a running program (that is, the graph of live data) [12]. Tracing garbage collectors require a *root set* for their mutator programs: the set of pointers into the heap contained by program variables. They start from the root set and follow pointers recursively to determine the graph of reachable data. Conventionally, garbage collectors for functional language implementations are *precise*: they require information about which program values are pointers. Not all precise collectors require this information to be statically apparent: for example, collectors for Lisp [13] or ML [14] typically identify pointers through

dynamic tags. I target a precise collector that does need this information statically, so verifying the entire executable (including the collector) depends on verifying that programs declare their root sets correctly.

Safer compilation The strongest notion of "safety" we can imagine for a compiler is that it always generates code that is semantically equivalent to the source program, given a formal semantics that captures all of the program properties we care about. This notion is called semantic preservation. Providing a machine-checked proof that a compiler preserves semantics maximizes our confidence that the property holds: in that case, the only component of the system that users need to trust is the proof checker. Recently, Leroy accomplished this for C in the Compcert compiler [15]. Building such a compiler for Haskell is possible but daunting, especially since Haskell lacks a canonical formal semantics.

A more approachable alternative is to prove that the compiler preserves types. A type-preserving compiler comes with a proof that its transformations map well-typed programs onto other well-typed programs, possibly in the same language or possibly in a different one. Thus, a type-preserving compiler promises to generate well-typed code, but not necessarily to preserve program meaning. One example is the TAL compiler [16], which compiles code in System F (an archetypal simple functional language) to typed assembly language. Integrating an RTS with a TAL compiler requires either trusting the RTS (which I have argued is risky business) or implementing the RTS in TAL. It seems difficult to implement a garbage collector in a typed language, even a very low-level typed language like TAL. Recent work on designing a garbage-collecting typed assembly language addresses this challenge, though this approach has not been used with a functional language compiler [17].

Previous work on type-preserving compilation has effectively pushed the complexity of the source language type system as far down the compilation chain as possible. I adopt a different philosophy, and ask how much complexity we can *remove* from the source language type system. As of yet, there has been no production-quality TAL compiler for a functional language, which makes a more incremental approach seem appealing. (Recently, Chen and colleagues developed a production-quality optimizing TAL compiler for the object-oriented language C# [18], but that is the only example of such a compiler I know of.) Rather than building a fully type-preserving compiler for Core, I combine a very simple intermediate language type system with the insertion of dynamic checks to yield a compiler with a straightforward safety preservation argument.

3 The compilation pipeline

My compiler accepts Core code from GHC and emits Cminor code to be compiled by Compcert. The middle end is my contribution. The translations from Core through my compiler's intermediate languages preserve types; the safety argument for the compiler relies on Compcert's stronger safety guarantee to carry the type preservation property through to the executable code. Because
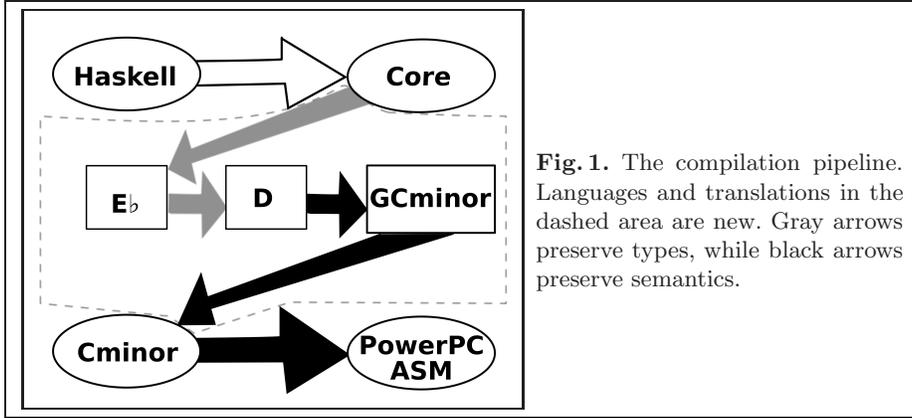
Fig. 1. The compilation pipeline. Languages and translations in the dashed area are new. Gray arrows preserve types, while black arrows preserve semantics.

it removes some of the type checks from the source language type system, the compiler must insert some additional runtime safety checks into compiled code.

GHC's front-end parses and typechecks Haskell source programs, then desugars them into Core, a typed intermediate language. GHC does most of its optimizations after translation to Core, and before translation to a lower-level untyped intermediate language [19]. To use GHC as a front-end, I take advantage of GHC's "External Core" feature, which produces a text-based representation of optimized Core code from a Haskell source file.[2] The advantage of using External Core is that it has well-defined static and dynamic semantics, as well as a stand-alone executable typechecker and interpreter, and so using GHC-compiled Core programs does not require trusting GHC itself [21]. Core was originally similar to System F, with the addition of *case* and *let* constructs [19]. More recent releases of GHC use a version of Core based on System F_C, which extends System F with type equality coercions [22].

Compcert is a certified compiler for a subset of C. It is *certified* because it comes with a machine-checked proof that it compiles C programs to semantically equivalent assembly-language programs [15]. Compcert targets the PowerPC platform. Compcert's back-end language, Cminor, is my compiler's target language. It is a C-like low-level intermediate language designed as a target for compiling C programs. It is untyped, except for distinguishing between integer and floating-point variables. Thus, Cminor's integer type encompasses both pointers and integers. In the rest of this paper, I explain how I attacked the problem of bridging the gap between Core and Cminor: in particular, reconciling Core's rich type system with Cminor's simple type system, and reconciling Core's call-by-need semantics with Cminor's call-by-value semantics.

---

[2] I use the version of External Core emitted by GHC version 6.10.1 [20]. With Aaron Tomb, I overhauled this feature, which had fallen into disrepair in a number of previous releases of GHC, and got it working in GHC 6.10.1. I also rewrote GHC's libraries for Core manipulation, on which my own compiler relies.

Figure 1 illustrates the compilation pipeline. The new languages in between Core and Cminor are E♭, D, and GCminor.[3] E♭ is a strict functional language with a very simple type system. In contrast with Core's rich type system, E♭ collapses all types that are represented by pointers into a single type □ (pronounced "box"). The translation from Core to E♭ makes laziness explicit with *force* and *delay* constructs. D is a strict first-order functional language that adds record types to E♭'s simple type system. Translating from E♭ to D mainly consists of closure conversion: that is, transforming definitions of first-class functions into top-level declarations of functions that take their free variables as arguments (packaged in a record) [23]. GCminor is a slight variation on Cminor that adds explicit allocation of memory in an implicitly garbage-collected heap. Translating from D to GCminor makes allocation operations and runtime checks explicit, as well as setting up the runtime layout of records. Allocations and function calls in GCminor are parameterized by root sets, allowing garbage collection to occur at any time: at any code point, the current root set is statically apparent.[4]

The translations from Core to E♭ and from E♭ to D are type-preserving, while the translations from D to GCminor and from GCminor to Cminor must be proven semantics-preserving, as the target languages are untyped. (While the first two proofs are complete, the second two are not.) Linking the Cminor code produced by my compiler with a verified garbage collector[5] yields the complete safety-preserving compilation system.

4 Type system

The type system of E♭, my compiler's first intermediate language, is designed to make it possible to verify that programs declare their root sets correctly. Thus, just enough information has to be statically manifest in a program to determine whether each program variable is a pointer. The type system I describe here captures the property that all reads or writes are to valid record fields. It is designed to be as simple as possible while still capturing that property. The features of Core's type system (including algebraic datatypes, parametric polymorphism, recursive types, and type equality coercions) are all individually well-understood, but they interact with each other in subtle ways that make it hard to remove one feature without removing many others. I saw no middle ground between completely maintaining Core's type system all the way down to a low level of abstraction, and simplifying it radically.

Thus, the type system of E♭ has only three types: Int, for unboxed (machine) integers; Float, for unboxed floating-point numbers; and □, for all values that are represented by pointers:

$$t \rightarrow Int \mid Float \mid \Box$$

---

[3] As Core is based on System F, the language names are inspired by the C minor scale, whose first four pitches are C, D, E♭, and F.

[4] GCminor, and the translations from D to GCminor and GCminor to Cminor, are due to Andrew McCreight and Andrew Tolmach.

[5] The garbage collector is by Andrew McCreight, building on his previous work [24].

Operationally, a value of type □ represents a pointer that code can dereference to yield a heap object that has a header and possibly some fields. Because the type system is not powerful enough to express the entire static semantics of Core, a well-typed E♭ program can evaluate to a checked runtime error if it uses data inconsistently with the semantics of Core (for example, by trying to apply a list as if it were a function). In other words, E♭ can type more programs than Core can: there are some "junk" E♭ programs that do not correspond to any valid Core program. So, abstractly, the formal semantics for E♭ attributes a meaning to these "junk" programs: the runtime error value. Concretely, compiling E♭ necessitates inserting runtime checks. You might ask why it is better for a program to halt with a checked runtime error than for it to crash. Fail-stop behavior, which runtime checks achieve, isolates errors and ensures that an erroneous program is killed before it can either do dangerous things such as reading memory that belongs to a different process, or silently return the wrong results.

E♭ examples The following fragment of E♭ code defines a higher-order function (call it `apply-head`) that takes another function, a list, and a default value, and applies the function to the head of the list if the list is non-empty. If the list is empty, then it returns the default value.

```
1    λ→Int (f:□) (z:Int) (xs:□) →
2       case xs of
3          Nil → z
4          Cons (y:□) (ys:□) → f□→Int y
```

Line 1 begins the definition of a function returning an `Int` ($\lambda_{\to Int}$) that takes three arguments: `f` and `xs`, with type □, and `z`, with type `Int`. The *case* expression (line 2) inspects the constructor with which `xs` was built and dispatches the appropriate alternative: line 3 if it was `Nil`, and line 4 if it was `Cons`. Line 4 binds the variables `y` to the head of the list and `ys` to the tail of the list, and applies the function argument `f` to `y`. The type tag □→`Int` on `f` says: "Check at runtime that `f` is a function that takes a □ argument and returns an `Int` result." (The notation □ →`Int` does not denote a function type, despite appearances: E♭ has no function types.) The *case* expression may fail with a checked runtime error if `xs` is a record tagged with a tag other than `Nil` or `Cons`, or if it evaluates to a function. Note that this is different from an equivalent function in Haskell, which would take an argument of type `[a]` (list of `a`, for any type `a`) and would guarantee by type safety that one of the *case* alternatives always matches.

This code is well-typed according to the typing rules of E♭, because the *case* expression is over a value with □ type; `f` was declared with type □; `y` was declared with type □, matching the type tag on the application; and the results of the two *case* alternatives both have the same type: `Int`. Note that although the application on line 4 says that we expect `f` to take a □ and return an `Int`, E♭'s type system *doesn't* check this property statically.

The following code typechecks, but would signal a runtime error:

```
apply-head□→Int→□→Int (λ→□ (x:□) → x) 1 ⟨Cons ⟨True⟩, ⟨Nil⟩⟩
```

The notation ⟨Cons ⟨True⟩, ⟨Nil⟩⟩ builds a record whose tag is Cons and whose two fields are also records (the empty record tagged with True and the empty record tagged with Nil). This code will pass the typechecker because all the arguments to apply-head have types consistent with the type annotation attached to its application, but at runtime, when the code for apply-head attempts to apply its first argument, it will fail with a checked runtime error, because the return type of the first argument to apply-head here is □, while the application on line 4 of the apply-head code expects f to return an Int.

In contrast, the typechecker rejects the following code statically:

apply-head$_{□→Int→□→Int}$ 1 ($\lambda_{→□}$ (x:□) → x) ⟨Cons ⟨True⟩, ⟨Nil⟩⟩

as it attempts to pass in an Int value for a □ argument and a □ value for an Int argument, contradicting the type tags attached to the application.

The same example also illustrates another kind of dynamic check necessitated by E♭'s type system. If this were Core code, we would know that the *case* expression on line 2 was exhaustive, because the type system guarantees statically that xs has type [a] for some type a and that every list constructor (Nil and Cons) has a corresponding alternative. However, E♭'s type system attributes a single type, □, to all values that *case* expressions can legally scrutinize. In effect, all tags are like branches of a single algebraic data type. So, there must always be an implicit default *case* alternative that signals a runtime error. You might think that this kind of check adds no additional cost, because code that dispatches *case* alternatives must check the tag of the scrutinee anyway, in order to choose the alternative to execute. However, adding the checks rules out the possibility of using jump tables (which are sometimes more effcient than decision trees) to compile *case* expressions [25].

A more serious problem arises with types such as Haskell's pair type:

```
1    λ→□ (p:□) →
2         case p of
3           Pair (x:□) (y:□) → x
```

This code is for the familiar fst function, which extracts the first component from a pair (written with the Pair tag here). In the equivalent code in Haskell or Core, the compiler would need to insert no tag checking code at all; compiled code could just extract the first field of p without checking its tag. But the implicit default alternative in E♭ makes this *case* expression more expensive to execute. The pair type is an example of a product type: an algebraic data type with only one constructor. Product types occur frequently in Haskell: for example, ordinary integers are defined as a product type that wraps an unboxed machine integer. Some of these checks can be eliminated by reconstructing some type information at the D stage (see the next section), but not others.

Figure 2 shows a selection of typing rules for E♭, which specify formally the principles that I've shown through examples. (The other rules are conventional.) The typing relation is of the usual form: $\Gamma \vdash_\Delta e : t$ (pronounced "the expression e has type t under typing assumptions Γ and Δ"). The variable environment Γ maps variables to types. The typing rules also depend on another environment,

9

$$\frac{\Gamma \cup \{v \mapsto t_1\} \vdash e : t_2}{\Gamma \vdash (\lambda_{\to t_2} \ (v : t_1). \ e) : \square} \tag{ELAM}$$

$$\frac{\Gamma \vdash e_1 : \square \qquad \Gamma \vdash e_2 : t_1}{\Gamma \vdash (e_{1 t_1 \to t_2} \ e_2) : t_2} \tag{EAPP}$$

$$\frac{\begin{array}{c} \Delta(tag) = \langle t_1, \cdots, t_n \rangle \\ \Gamma \vdash e_1 : t_1 \ \cdots \ \Gamma \vdash e_n : t_n \end{array}}{\Gamma \vdash \langle tag \ e_1, \cdots, e_n \rangle : \square} \tag{ERECORD}$$

$$\frac{\begin{array}{c} \Gamma \vdash e : \square \qquad \overrightarrow{\Delta(tag) = \langle t_1, \cdots, t_j \rangle} \\ \overrightarrow{\Gamma \cup \overrightarrow{\{v \mapsto t\}} \vdash e_{alt} : t} \end{array}}{\Gamma \vdash case \ e \ of \ \overrightarrow{(tag \ \overrightarrow{(v, t)} \to e_{alt})} : t} \tag{ECASE}$$

Fig. 2. Selected E♭ typing rules

Δ, that is generated as part of the Core-to-E♭ translation and is thus considered constant for a given program. This environment maps a record tag T to a list of types of the fields that a record tagged with T should have. Because Δ is constant for a given program (typechecking a subexpression does not extend it with new entries), I omit the Δ subscript from the rules in Figure 2.

D: types to eliminate runtime checks D's type system is the same as E♭'s type system, with the addition of record types. The E♭-to-D translation does closure conversion, as mentioned in Section 3. D implements both closures and E♭'s records with the same record construct, and record types allow efficient handling of explicit closures. Without record types and syntax for unchecked field access, extracting free variables from closures would be too expensive.

$$t \to Int \mid Float \mid \square \mid tag \qquad\qquad \Delta : tag \mapsto t*$$

Record types correspond to E♭'s record tags, so as before, the tag environment Δ is needed. Δ maps record types onto lists of types (of the record's fields).

The type system has a subtyping rule to allow record-typed expressions in contexts that expect a $\square$ type. Record types also allow eliminating some runtime checks for *case*s. If the compiler can prove that the scrutinee of a *case* expression must have a specific record type, then that type determines the *case* alternative that will execute at runtime; thus, it is safe to replace the entire *case* expression with the code for that alternative. This is similar to the conventional case-on-known-constructor optimization [19]. Why throw away type information in one compilation phase only to reconstruct it in a later one? Because discarding type information simplifies the translation, and reconstructing it is an optional optimization that does not affect the correctness of the system. In addition, known functions (those whose bodies are statically known) do not require runtime type checks. Identifying these functions is another optimization.
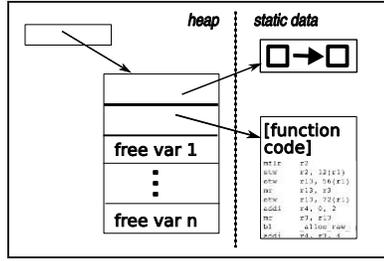
Fig. 3. The runtime layout of a heap record representing a closure. The first field of the record points to a type descriptor, which describes the argument type and the result type of the closure (both of which are □, in this case). The second field of the record points to the closure code, while the remaining fields give the closure's free variables.

Runtime checks: implementation In GCminor code, there is no distinction in the language between closure records and data records. A closure record represents a function: conceptually, a pointer to code, along with a list of free variables used in that code. The translation from D to GCminor sets up the concrete runtime layout for records shown in Figure 3, as well as inserting the code that does runtime checks. All records, including closure records, have an additional field pointing to a type descriptor, as shown in the figure.

The following is an example of Cminor code for a checked function call.[6] If runtime checks were disabled, only line 2 would remain: `f` would be called unconditionally. With safety checks disabled in this example, if `f` actually expected a pointer rather than an integer, a segmentation fault could occur. The safety check prevents this: it terminates the program before the bad call to `f` can occur.

```
1    if (37 == int32[int32[f - 4] + 8])
2      result = int32[f](0);
3    else
4      type_error();
```

In this code, `f` is the operator, represented by a closure record. In this example, the literal 37 (line 1) corresponds to the statically visible type tag that is part of the syntax for applications in D, and the memory access `int32[int32[f - 4] + 8]` corresponds to the dynamic type tag on the operator in an application. The code checks the static tag against the dynamic tag, calling the function if the tags match and signalling an error otherwise. The D-to-GCminor translation assigns these type tags. It can represent the type tags as small integers because there are only a fixed number of types in a given program.

Records for data constructions, as opposed to closures, have similar type tags; the code that checks these tags is folded into the code that does *case* dispatch. Thus, the theoretical upper bound for the cost of runtime checks amounts to at most one additional conditional statement per function call and *case* expression.[7] In Section 7 I try to find out what the added cost tends to be in practice.

---

[6] I have simplified the syntax of Cminor here to eliminate some irrelevant details.

[7] The cost would be higher with a back-end that supported jump tables, since runtime checks for *case*s rule out this possibility. Compcert does not support jump tables.

Related work Abadi and colleagues observed that in a statically typed language, it is useful to be able to inspect the type of a value at runtime, raising an exception if the type is erroneous [26]. They introduced a data type *Dynamic* with a constructor that pairs a value with a type tag, and a deconstructor (called *typecase*) that branches based on the type tag. E♭'s □ type is similar to the *Dynamic* type, and E♭'s application and *case* constructs serve a similar purpose to *typecase*. The □ type is also reminiscent of the exception type in Standard ML: a dynamically extensible datatype constructor [27].

By contrast with this work, which turns static checks into dynamic checks, work on *soft typing* attempts to turn dynamic checks into static checks. For example, Wright and Cartwright used soft typing to add partial static type-checking to Scheme [28]. Their work aims to provide better error checking for programmers and improve performance by eliminating some runtime checks.

The TAL compilation framework [16] requires no dynamic checks, because the type system for its target language is as powerful as the type system for its source language. To allow TAL code to interact with an untrusted collector, Vanderwaart and Crary defined a type system to describe the interface of a precise collector [29]. Also, Hawblitzel and Petrank presented an automatically verified garbage collector that uses a formal specification of the mutator interface [30]. Their specification of the mutator-collector interface models the source language type system directly. In general, TAL-based approaches require a more complex type system, but avoid additional dynamic checks.

5 Compiling laziness

In a language with call-by-need semantics (lazy evaluation), every function argument is evaluated at most once. For example, in the following code:

```
let f = λ x → x * x in f (37 + 42)
```

the value of $(37+42)$ is computed at the first reference to x in f. The second reference to x sees the cached value of the addition (79), rather than recomputing (37+42). Typically, Haskell implementations implement laziness by wrapping function arguments in *thunks*, or suspended computations. Functions expect their lazily evaluated arguments to be pointers to thunks. When the thread of execution demands the value of a thunk, code inserted by the compiler calls the code that does the suspended computation. Then, this code does an *update*: it overwrites the pointer to the thunk with a pointer to code that returns the computed value. Thus, further references to the same pointer will just return the same value, without recomputation [9].

One way to implement selective laziness in a strict language is to provide constructs that let programmers create thunks explicitly (*delay*) and demand their values explicitly (*force*). Borrowing from one presentation of *delay* and *force* in the Scheme language [31, p. 261], I use them to preserve the call-by-need semantics of Core in a translation to a language with call-by-value semantics.

12

Laziness in E♭ The following example, a lazy version of the `fst` example from Section 4, illustrates how E♭ handles laziness. (Because E♭ has call-by-value semantics, the earlier version of `fst` was strict.)

```
1    let fst = λ→□ (p:□) →
2        case !p of
3          Pair (x:□) (y:□) → !x in
4      fst /g□→□→□ a b/
```

where `a` and `b` are assumed to be bound somewhere in the enclosing scope. (I omit the type annotations on the application of `fst` and its binding, because here, `fst` is a known function and thus requires no dynamic checks on calls.)

The `!` (line 2) denotes *force*, and slashes (line 4) denote *delay*. The argument of `fst` is a thunk yielding a pair when it is evaluated, so it has to be forced (line 2) before use. Since any Core value can be a thunk, all references to □ variables have to be forced (line 3). The `!` operator evaluates its operand to weak-head normal form [9], so the first field of `p` still needs to be forced on line 3.

Laziness in D The following is the D code for the lazy `fst` function:

```
5    let fst = λ→□ (p:□) =
6      case FORCE(p) as p1 of
7        Pair → FORCE(p1#1)
```

`FORCE` is shorthand for a pseudo-macro that actually appears inline. The syntax (`p1#1`) on line 7 selects the first field from the record `p1`. D replaces pattern-matching on records with selection of numbered record fields. To allow for such selections to be typed, D's *case* expression names the result of evaluating the scrutinee (on line 6 above, `p1`). In this example, the typechecker would check the `Pair` alternative in an environment where `p1` is bound to type `Pair`; if there were more alternatives, then the typechecker would check each one in an environment where `p1` was bound to the corresponding type for the tag for each.

In D, a call to `fst` might look like:

`fst ⟨LAZY:(g□→□→□ a b)⟩`

This code invokes `fst` on a nested record: a record with tag `LAZY` whose single field is a pair (in this case, the result of a function g; suppose g, a and b are bound in the enclosing scope). The `LAZY` tag and code that uses it are generated by the E♭-to-D translation, as is the code for the `FORCE` pseudo-macro. The point is that *force* and *delay* are implemented inside the strict D language, with no need to specify the meaning of laziness in D's semantics or to add special support for updates and thunks to the RTS. A thunk is just a function that ignores its single argument and returns the value of the suspended computation.

Here is the code that implements the `FORCE` pseudo-macro for a variable x. Figure 4 illustrates the data structures that this code manipulates.[8]

---

[8] For readability, I have simplified D's syntax. D does not have whole-record assignment as shown on lines 4 and 7; rather, `x1`'s tag would be changed and its first field would be mutated in separate operations. Also, D does not have explicit sequencing as shown with a semicolon; rather, sequencing is implemented by *let* expressions.
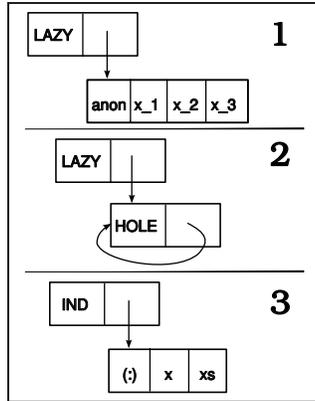
Fig. 4. The lifetime of a thunk. In stage 1, a thunk is a record with the LAZY tag that points to a thunk record with free variables and a tag that indicates the thunk code. In stage 2, during evaluation, the first field of the LAZY record points to a *blackhole*: a record with the HOLE tag that points to itself. In stage 3, the update has been done, and the record has the IND ("indirection" tag), pointing to the result.

```
1        case x as x1 of
2            LAZY → let thunk : □ = x1#1 in
3                       x1 ← ⟨ HOLE, x1 ⟩;
4                       let result : □ =
5                           thunk_Int→□ 0 in
6                       x1 ← ⟨ IND, result ⟩;
7                       result
8            IND  → x1#1
9            HOLE → error 0
10           default → x1
```

The first three *case* alternatives each correspond to one of the stages shown in Figure 4. (The steps shown in the figure are standard [9]; it is just the implementation that is non-standard.) The LAZY case corresponds to stage 1 in the figure; the HOLE case, to stage 2; and the IND case, to stage 3. (The code in the LAZY case sets up a "blackhole" structure as shown in stage 2 of the figure, which is necessary to prevent space leaks [32].) The default alternative (line 11) matches if none of the other alternatives match. This allows any value to be used in a context where a thunk is expected, which is useful for optimizations that change lazy to strict evaluation where it can do so without altering semantics.

This explicit desugaring of laziness has both advantages and disadvantages. It makes more information available for compiler transformations, and it allows removing support for laziness from the RTS. But it increases code size: the code for FORCE shown above is duplicated at every static reference to a lazy variable.

Related work Other studies in the Haskell literature have treated laziness as a construct to be desugared away early in compilation. For example, the Yale Haskell compiler compiled Haskell to a Lisp-like language [33]. It failed to achieve good performance due to limitations of Common Lisp's RTS [34]. Later, Faxén [35] and Boquist and Johnsson [36] built experimental optimizing

14

compilers based on the principle that making laziness explicit in the language allows more aggressive optimization. These projects contrast with my use of it to simplify the RTS. Some recent projects [37] [38] [39] have used Boquist's GRIN back-end as well, though not with RTS verification as a specific goal.

6   Correctness

I developed a formal static and dynamic semantics for Core, E♭, and D. I proved that the type systems of E♭ and of D are sound, and proved that the translation from Core to E♭ preserves types. Here, type preservation means that a well-typed Core expression is also well-typed when translated into E♭, and further, the type of the resulting E♭ expression is the result of translating the original Core expression's type into an E♭ type. An analogous statement is true about the translation from E♭ to D. For space reasons, I have omitted the formal statements of the semantics and of the safety proofs. The proofs follow the usual "safety = progress + preservation" equation [40], and are straighforward. These proofs could be mechanized, which I leave for future work. Complete assurance requires a proof that the D-to-GCminor translation and the GCminor-to-Cminor translations preserve semantics. This work remains to be done.

7   Performance results

My performance goal is not to generate equally high-performance code as the state of the art, but rather, to generate code that is both safe and usable. I define the threshold of usability as performing roughly as well as interpreted code or unoptimized code from a state-of-the-art compiler. Timings for a small sample of programs suggest that with optimizations disabled, recent versions of GHC produce code that is anywhere between 2 and 10 times slower than optimized code. As I go on to show, my compiler generates code whose performance falls within this range, as compared to the equivalent optimized GHC code.

Optimizations The naïve translations from Core to E♭ and from E♭ to D are simple, but to achieve the performance goal, I had to implement quite a few optimizations on E♭ and D code. Many of the optimizations are simple, and most are based on well-known compiler optimizations for functional languages. Note that proving the entire optimizing system correct requires proving that each optimization is type-preserving. I have not done this; even so, we can still be sure that the system generates well-typed code, since the typechecker runs after each transformation. Notable optimizations included using more efficient function calling conventions; code motion; eliminating *delay* and *force* operations where they can be proven statically to be unnecessary; eliminating *cases* on values whose tags are known statically; dead code elimination; and inlining. The results in the next section apply to optimized code.

15

Performance I ran the compiler (hereafter referred to as F2CM) on a sample of programs from the "spectral" section of the Haskell "nofib" benchmark suite [41]. The spectral programs are small kernel examples from larger, real programs. One program, `queens`, is from the "imaginary" section (toy programs). As a baseline, I ran GHC on the same set of programs, with optimizations enabled. The ten programs I chose are shown in Figure 5. Program size ranged from 26 to 685 lines of code (including whitespace and comments).

I had to modify most of the benchmarks slightly to avoid generating code with foreign function calls or certain primitive operations the back-end does not support. I compiled the benchmarks with a modified version of the GHC standard libraries that omits most I/O functions and others based on unsafe primitives or foreign calls. Also, GHC's `Integer` type (used for arbitrary-precision integers) is implemented by foreign calls to the GMP library, so I replaced that library with an alternative implementation, "integer-simple" [42]. Due to technical difficulties, when compiling the baseline GHC programs, I did not link them against the integer-simple library. So my comparisons may be skewed in favor of GHC, at least for programs that make heavy use of `Integer`s. Out of the 43 spectral programs in the current version of nofib, F2CM can compile 14 programs. For my experiments, I chose the ones that ran long enough to get reliable timings.

Figure 5 compares the running times of F2CM-compiled programs with those of GHC. All the timing numbers (for both F2CM and GHC) refer to an arithmetic mean over three runs. In general, F2CM generates significantly slower code than GHC: in this set of benchmarks, two times slower in the best case (`multiplier`) and nine times slower in the worst case (`queens`). With checks enabled, F2CM-compiled programs run on average (geometric mean) 4.3 times slower than the equivalent GHC-compiled programs. This is surprising, because GHC does most of its optimizations as Core-to-Core passes, and I am using optimized Core [19]. However, GHC's back-end is significantly optimized as well, whereas Compcert's back-end optimizes only lightly.

Runtime checks add significant but not overwhelming overhead: an average (geometric mean) of 10%, with 18% in the worst case. Other experiments (whose results are not shown here) suggested that checks on functions and on *case* expressions contributed about equally to the overhead of runtime checks. Conceivably, even more checks could be eliminated through more aggressive static analysis that identifies calls to known functions or *case*s on values built with known tags. I suspect, but cannot prove, that enough information is present statically in E♭ and D programs to obtain better performance; the compiler just isn't taking advantage of it yet. For example, statistics gathered in GHC showed that for a sample of Haskell benchmarks, an average of 79% of calls were to known functions [43]. It remains for future work to further clarify why the slowdowns are as high as they are. For now, though, these experiments show that F2CM generates code whose speed is within a factor of ten of GHC-generated code, for these benchmarks. Thus, I conclude that compiling Core to Cminor is a plausible strategy.
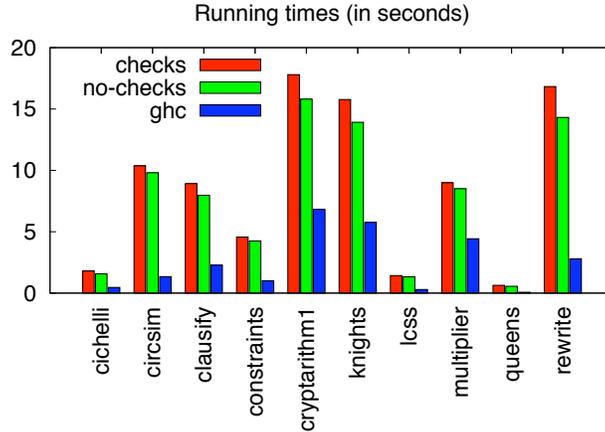
Fig. 5. Comparing running times with F2CM and GHC. The *y*-axis shows time in seconds. The "checks" bars (red) show the running times of F2CM-compiled programs with safety checks enabled. The "nochecks" bars (green) show the running times of F2CM-compiled programs with checks disabled. The "GHC" bars (blue) show the running times of the baseline GHC-compiled versions of the programs. In the experiments, F2CM compiled Core programs generated by GHC 6.9.20080918 (a development version of GHC, to which I made the necessary library changes), and the baseline GHC programs were from GHC 6.10.1. I always ran GHC with the `-O2` optimization flag. I ran both sets of programs with 128 MB of heap space. I ran all the programs on a Macintosh PowerBook G4 with a 1 GHz PowerPC processor and 512 MB of physical RAM, running Mac OS version 10.5.4.

8   Conclusion

What is the cost of assurance? My results show that safety checks add overhead of up to 18%, implying that the *explicit* safety features of the compiler add low overhead. Because I designed the compiler for simplicity, it may have features that are *implicitly* necessary for safety and that contribute additional cost. This cost is harder to measure. Even so, the overall slowdown compared to GHC was still less than a factor of ten for all the examples I tried, meeting my performance goal. Since I know of no other safety-preserving compilers for Haskell, my results comprise the first set of performance data for such a compiler.

Does simplicity really conflict with performance? So far, compiler designers have lacked compelling reasons to design for verifiability, and so no one has been motivated to answer this question. I have presented a compiler that, while simple, contributes no obvious intrinsic sources of overhead other than runtime checks. Though the overall overhead is high, I have not answered the question of whether this overhead is intrinsic to safety-preserving compilation or whether it can be overcome by cleverer compilation strategies. Only further performance

17

analysis and profiling can answer the question of whether this combination can perform well enough to persuade users to adopt it.

When you pay the cost of assurance, what do you get in return? Some people might wonder whether protection against the seemingly unlikely possibility of a bug in a production compiler or garbage collector is worth paying any additional cost. Whether safety is worth paying for depends on the cost of a safety violation. In a time of clickwrap licenses that shield software developers from accountability for bugs, developers may treat the cost of distributing unreliable code as an externality. If someday users' expectations for software reliability rise, developers will have to internalize this externality and the idea of trading performance for safety may become more compelling. There can be no ironclad safety guarantees, because safety proofs and proof checkers can contain errors as well, but the smaller the base of trusted code becomes, the more confident developers can be that their software will do what they expect it to.

References

1. Milner, R.: A theory of type polymorphism in programming. Journal of Computer and System Sciences **17**(3) (December 1978) 348–375
2. Peyton Jones, S., ed.: Haskell 98 Language and Libraries – The Revised Report. Cambridge University Press, Cambridge, England (2003)
3. The GHC Team: The Glasgow Haskell Compiler. `http://haskell.org/ghc` (April 2009)
4. The Mozilla Foundation: Mozilla foundation security advisory 2009-13. `http://www.mozilla.org/security/announce/2009/mfsa2009-13.html` (March 2009)
5. Marlow, S., Peyton Jones, S.: The new GHC/Hugs runtime system. `http://www.research.microsoft.com/~simonpj/Papers/new-rts.ps.gz` (1998)
6. The GHC Team: GHC commentary: The runtime system. `http://hackage.haskell.org/trac/ghc/wiki/Commentary/Rts` (January 2009)
7. The GHC Team: Ghc - Trac. `http://hackage.haskell.org/trac/ghc/` (April 2009)
8. Leroy, X.: The Compcert verified compiler. `http://compcert.inria.fr/doc/index.html` (April 2009)
9. Peyton Jones, S.L.: Implementing lazy functional languages on stock hardware: The Spineless Tagless G-machine. Journal of Functional Programming **2**(2) (1992) 127–202

10. Peyton Jones, S.L., Launchbury, J.: Unboxed values as first class citizens in a non-strict functional language. In Hughes, J., ed.: Proceedings of the Conference on Functional Programming and Computer Architecture, Cambridge, Massachusetts, USA, Springer-Verlag LNCS 523 (1991) 636–666

11. McCarthy, J.: Recursive functions of symbolic expressions and their computation by machine, part I. Commun. ACM **3**(4) (1960) 184–195

12. Jones, R.E.: Garbage Collection: Algorithms for Automatic Dynamic Memory Management. Wiley, Chichester (July 1996)

13. Wilson, P.R.: Uniprocessor garbage collection techniques. `ftp://ftp.cs.utexas.edu/pub/garbage/bigsurv.ps` (1992)

14. Tolmach, A.: Tag-free garbage collection using explicit type parameters. In: Proceedings of the 1994 ACM Conference on Lisp and Functional Programming, ACM Press (1994) 1–11

15. Leroy, X.: Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In: 33rd Symposium on Principles of Programming Languages, ACM Press (2006) 42–54

16. Morrisett, G., Walker, D., Crary, K., Glew, N.: From System F to typed assembly language. ACM Transactions on Programming Languages and Systems **21**(3) (1999) 527–568

17. Hawblitzel, C., Huang, H., Wittie, L., Chen, J.: A garbage-collecting typed assembly language. In: TLDI '07: Proceedings of the 2007 ACM SIGPLAN International Workshop on Types in Language Design and Implementation, New York, NY, USA, ACM (2007) 41–52

18. Chen, J., Hawblitzel, C., Perry, F., Emmi, M., Condit, J., Coetzee, D., Pratikaki, P.: Type-preserving compilation for large-scale optimizing object-oriented compilers. In: PLDI '08: Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation. (2008) 183–192

19. Peyton Jones, S.L.: Compiling Haskell by program transformation: A report from the trenches. In: European Symposium on Programming. (1996) 18–44

20. The GHC Team: Release notes for version 6.10.1. `http://www.haskell.org/ghc/docs/latest/html/users_guide/release-6-10-1.html` (November 2008)

21. Tolmach, A., Chevalier, T., the GHC Team: An external representation for the GHC Core language. `http://www.haskell.org/ghc/docs/latest/html/ext-core/core.pdf` (April 2009)

22. Sulzmann, M., Chakravarty, M.M.T., Peyton Jones, S., Donnelly, K.: System F with type equality coercions. In: TLDI '07: Proceedings of the 2007 ACM SIGPLAN International Workshop on Types in Language Design and Implementation, New York, NY, USA, ACM (2007) 53–66

23. Appel, A.W., Jim, T.: Continuation-passing, closure-passing style. In: Sixteenth ACM Symposium on Principles of Programming Languages, ACM Press (1989) 293–302

24. McCreight, A.: The Mechanized Verification of Garbage Collector Implementations. PhD thesis, Yale University (December 2008)

25. Appel, A.W.: Compiling with continuations. Cambridge University Press (1992)

26. Abadi, M., Cardelli, L., Pierce, B., Plotkin, G.: Dynamic typing in a statically typed language. ACM Trans. Program. Lang. Syst. **13**(2) (1991) 237–268

27. Milner, R., Tofte, M., Harper, R., MacQueen, D.: The Definition of Standard ML (Revised). MIT Press, Cambridge, MA, USA (1997)

28. Wright, A.K., Cartwright, R.: A practical soft type system for Scheme. In: In Proceedings of the 1994 ACM Conference on LISP and Functional Programming. (1994) 250–262
29. Vanderwaart, J.C., Crary, K.: A typed interface for garbage collection. In: Proc. 2003 ACM SIGPLAN International Workshop on Types in Language Design and Implementation, New York, NY, USA, ACM Press (2003) 109–122
30. Hawblitzel, C., Petrank, E.: Automated verification of practical garbage collectors. In: POPL '09: Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages, New York, NY, USA, ACM (2009) 441–453
31. Abelson, H., Sussman, G.J.: Structure and Interpretation of Computer Programs. first edn. The MIT Press (1985)
32. Jones, R.: Tail recursion without space leaks. Journal of Functional Programming **2** (1992)
33. Bloss, A., Hudak, P., Young, J.: Code optimizations for lazy evaluation. Lisp and Symbolic Computation **1**(2) (1988) 147–164
34. Hudak, P., Hughes, J., Peyton Jones, S., Wadler, P.: A history of Haskell: being lazy with class. In: HOPL III: Proceedings of the third ACM SIGPLAN conference on History of Programming Languages, New York, NY, USA, ACM (June 2007)
35. Faxén, K.F.: Analysing, Transforming and Compiling Lazy Functional Programs. PhD thesis, Royal Institute of Technology (June 1997)
36. Boquist, U., Johnsson, T.: The GRIN project: A highly optimising back end for lazy functional languages. In Kluge, W.E., ed.: Implementation of Functional Languages. Volume 1268 of Lecture Notes in Computer Science., Springer (1996) 58–84
37. Fokker, J., Dijkstra, A.: UHC/EHC structure. `http://www.cs.uu.nl/groups/ST/Projects/ehc/ehc-structure-doc.pdf` (April 2009)
38. Meacham, J.: Jhc Haskell compiler. `http://repetae.net/computer/jhc/` (April 2009)
39. Himmelstrup, D.: LHC Haskell compiler. `http://lhc.seize.it` (April 2009)
40. Pierce, B.C.: Types and programming languages. MIT Press, Cambridge, MA, USA (2002)
41. Partain, W.: The nofib benchmark suite of Haskell programs. In: Proceedings of the 1992 Glasgow Workshop on Functional Programming, London, UK, Springer-Verlag (1993) 195–202
42. The GHC Team: GHC developer wiki: Replacing GMP. `http://hackage.haskell.org/trac/ghc/wiki/ReplacingGMPNotes` (April 2009)
43. Marlow, S., Peyton Jones, S.: Making a fast curry: push/enter vs. eval/apply for higher-order languages. In: ICFP '04: Proceedings of the ninth ACM SIGPLAN International Conference on Functional Programming, Snowbird, UT, USA (2004) 4–15