Compiling functional code for system-level environment

Miroslav Kratochvíl[1]*

Department of Software Engineering, Charles University in Prague
kratochvil@ksi.mff.cuni.cz

Abstract: Compilation of programming languages based on λ–calculus to standalone, low-level machine code involves a challenge of removing automatic memory management that is usually required for supporting implicit allocations produced by currently available compilers. We propose a compilation approach that is able to convert pure, lazily-evaluated functional code to machine code suitable for running on bare hardware with no run-time support, and describe a Haskell-like programming language that demonstrates feasibility of the new method. Using the proposed approach, the missing ability to directly control the hardware and memory allocation may be added to purely functional languages, which otherwise have several advantages over traditional procedural languages, including easier verification and parallelization.

1 Introduction

Ideas from functional programming have always been influencing more traditional imperative programming languages. Apparent simplicity and expressiveness of λ–based constructions is reflected both in new features of some languages (notably the new functionality of C++11) or in whole new languages (Clojure, Rust, partially also in Swift or Nim). Functional languages, on the other hand, are being improved to acquire the benefits of languages with more direct control of resulting machine code, which may be required for reaching performance, efficiency, or binary compatibility goals — there has been much research aiming to make the high-level constructions more efficient, concerning e.g. memory allocation efficiency [7], type unboxing for performance [18] and various inlining methods.

We push both of these developments to one of possible meeting points — we demonstrate a language that satisfies the requirements from the system-level languages by having similar (mostly compatible) resulting code, execution and memory management model as C or C++, while supporting functional programming paradigms, type systems, syntax, and many practical programming constructions from Haskell and similar languages.

Major challenges. Such combination may easily lead to a direct contradiction, as the phenomena common in functional languages imply the need either for garbage collection or for other, possibly even more complicated run-time processing:

- Variables of recursive types (like lists and trees), that form a highly valued building block of functional programming, are quite difficult to be implemented efficiently without some kind of automatic deallocation decision. Common example of such decision may be seen in automatic handling of two singly-linked list objects that share a common 'tail'.
- A quite common technique in functional programming — generating functions on runtime by partial application and passing them around as first-class objects — is impossible in minimalist system-level conditions, as any code generator (or, equivalently, a compiler) can not be a part of the language runtime. Similar problem arises with code that implies need for lazy evaluation, which, if it can not be removed by inlining at compile time, is usually supported by run-time allocation of *thunks* in automatically managed memory.
- Arbitrarily deep recursion, a common method to run loops in functional programming, is usually supported by unbounded automatic allocation of a stack-like structure. In system-level programming, the programs and all recursion must fit into a standard, limited and unmanaged program stack segment.

Viability of a new language. The main motivation for creating a new language is to explore the possibilities that arise from the expressive power of a purely functional language applied to a full stack[1] of code that runs on bare hardware. The benefits may include easier high-level optimization and simplified static analysis.

In particular, performing partial evaluation of the functional code is a computationally easy, well developed and very effective method of optimization [20]. The lack of side effects (or correct embedding of side effects in a tangible construction) also simplifies derivation of semantic meaning of the code by reducing amount of variables that affect how the code is run.

Moreover, functional languages do not reflect any of the traditional paradigms that the system-level programming languages inherited: They are not designed specifically for register-based machines, nor uniprocessors, nor for sequential code execution[2], not even for preservation

*This work was supported by the grant 11562/2016 of the Charles University Grant Agency.

[1] We indirectly refer to the valuable property of C-like languages, that all dependencies of C (esp. the standard library) can again be written only in C.

[2] Any code that touches a common system-level primitive directly is almost necessarily sequential.

of call conventions and code structures shared by program parts. We consider lack of those properties a crucial benefit for simplification of automatic code processing, significant both for further elimination of variables affecting derivation of high-level optimization possibilities and, more notably, for automatic parallelization of code, as the compiler is not forced to perform a potentially sub-optimal decomposition of sequentially written code into paralellizable fibers.

Explicitly specified procedures and calls in block-structured code often represent some semantic value (like an API for module separation) that is unimportant or even harmful for resulting program structure — non-existence of any explicit code-structuring syntax leaves the choice of the call convention and separation of the code into subroutines on the compiler, which may produce better program while leaving the source semantically clean and readable.

We are further motivated by the recent high-performance results achieved by the language-centric functional programming approach, most notably the cache-oblivious memory allocation [2] or the generative approach to code parallelization [22].

1.1 Related research

We highlight several compilers that target a similar set of goals:

PreScheme — a language by Kelsey [15] is based on a compiler that transforms a simplified version of Scheme language to machine code. The approach chosen by authors is to completely replace all recursive types in the runtime with vectors, and to forcibly inline all code, so that no code-generating β–reductions present at runtime. As in other Scheme implementations, all evaluation is always eager and sequential structure of code is preserved.

Habit — a project of Portland State university HASP group [10] that states a need for similar system-programming language. Most recent publications from the project discuss the language features and provide a clear specification for implementation. To the best of our knowledge, no implementation of Habit is available yet.

Rust — a relatively new language that exploits techniques similar to linear types to stay both very efficient with memory allocations and safe against programmer errors. Compilation method and execution model chosen by Rust is similar to ours, with the exception of the imperative Rust syntax and the fact that the Rust runtime needs a garbage collector to work with recursive types from its standard library.

Swift — a recent product of Apple shows corporate interest in small, fast languages that reduce the memory-management overhead and include functional-programming improvements (notably pattern matching and a shifted view on classes, represented e.g. by Swift protocols and improved enumerated types).

Important theoretical result about evaluation methods, the fact that pure and eager functional languages can be shown inferior in terms of performance to languages that are either lazily evaluated or allow side-effects, was discussed by Bird, Jones and De Moor [1]. The low-level target environment of our language can not support lazy evaluation directly (although, in the code, the programmer can use lazy evaluation for any construction); we therefore must allow some well-contained amount of side effects.

We make extensive use of the knowledge that was gathered during the GHC development. Discussed compilation and optimization methods are usually based on the methods used to compile Haskell, as described for example in GHC Internals [13].

1.2 Approach

As a main goal we construct a simple language to demonstrate that there is no technical obstacle that would make system-level programming in a purely functional language impossible. To solve the aforementioned problems with run-time dependency on automatic memory management, we make following design choices:

- Recursive types are replaced by pointed types. We discuss the reasons and effects of this removal in section 2.3. As in C, management of all memory except the stack is done by programmer (directly or indirectly using library code) through pointed types or constructions based on them.

- Deep functional recursion is allowed, but it is replaced with tail recursion at all tail-call positions. Such trivial approach is used successfully in many compilers, including the GHC. See section 2.4 for details.

- Our main contribution is the method to run lazy evaluation without heap allocation of thunks. We store the thunks *in program stack* and apply them to function bodies that were modified at compile time to expect them as arguments, passed to them by a standard calling convention. We describe a fast, deterministic inference-based algorithm that converts functional code to such equivalently-behaving non-lazy form in section 2.2. Note that our solution works without any partial evaluation technique that is usually exploited for this purpose, but maintains the code in a form that is still able to be optimized and transformed by standard inlining algorithms.

For demonstration purposes, we only provide a simple type system (Hindley-Milner with several extensions)

and rely on the LLVM compiler framework to generate platform-specific code. Results are presented in section 3.

Although there is currently no guarantee that resulting programming language will be practical, we believe that low-level programming languages with high levels of abstraction are currently very favorable[3] and current development is producing a lot of small languages inspired by functional programming that target low-level goals (e.g. the Nim language), in which our resulting language could fit easily. Still, its main purpose is to serve as a future testbed for optimization and automatic parallelization techniques.

2 Language internals

The language is constructed similarly as other pure functional languages. We tightly follow the standard definition structure and functional syntax known from Haskell, with some simplifications (e.g. the syntax for type classes and related constructions is unnecessary for our purpose). Upon compilation, source code is type-checked, rewritten to non-polymorphic non-lazy equivalent, functions are lifted to form top-level blocks, and partially evaluated to certain extent for optimization.

The testing compiler finally emits a pack of LLVM intermediate code with several functions (e.g. `main`) exported to serve as entry points. LLVM framework is then used to compile the almost-machine LLVM code to an object file (or executable) of the target platform. Using LLVM as "assembly" allows us to simplify the compiler in three ways: We do not have to care about register allocation, spilling and raw stack operations, program can be easily linked with any code the LLVM bytecode is able to link with (notably any library that follows the C calling conventions, including the C standard library), and we can use the vast library of already-available low-level optimizations that can be run on LLVM bytecode.

2.1 Evaluation

In our case, the compilation of function evaluation is principally similar to that used in GHC — the function definitions that were optimized and lifted to top-level are compiled to form code blocks that obey a machine-level calling convention, and the machine code is generated for them. In contrast to GHC, our resulting code can only use a set of primitives applicable to system-level environment, notably it can not implicitly access any memory other than on the program stack. The two cases when GHC uses such allocation are the handling of boxed or recursive types (which we disallowed) and allocation of thunks for lazy evaluation, which must be worked around.

[3]This knowledge was gained by looking at the statistical distribution of programming languages used in software packages installed on an average Debian Linux system. [19]

Lazy evaluation Instead of thunk allocation in the STG[4] data structure, we will use a structure that is of predictable size and stored completely on stack. The only problematic part of such *static thunk* is its "meaning", or, technically, a description of the function that will evaluate the thunk. As the concept of function objects has no straightforward assembly-level representation, we replace it by a code address of a previously-prepared compiled function (or simply a "function pointer" to code generated at compile time) that is able to compute the actual result of the thunk. Rest of the static thunk consists of a tuple of the argument values that are expected by the pointed function on evaluation.

In resulting situation, the compiler must solve following new tasks:

- It has to ensure that the code is ready for passing the thunks around as function arguments or return values instead of simple values.
- It has to prepare thunk-evaluating functions for all occurrences of thunks in the code.

For example, consider this simple functional code:

```
f a = (+) a
g a b = a b
h a = g (f a) a
```

We will ignore the fact that any reasonable compiler would choose inlining with a far better result, and generate static thunks for demonstration. We progress as follows:

1. We will first derive the types of the code. Given initial basis $\Gamma \ni (+ : \mathbb{N} \to \mathbb{N} \to \mathbb{N})$, usual type inference would output following type assignments for the code: $\mathtt{f} : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$, $\mathtt{g} : \mathbb{N} \to \mathbb{N} \to \mathbb{N} \to \mathbb{N}$ and $\mathtt{h} : \mathbb{N} \to \mathbb{N}$.

   Instead, we reflect the need to see which functions must be called lazily and modify the type system to allow such expression. Specifically, the typing inferred for `f` is $\mathtt{f} : \mathbb{N} \to \textbf{thunk}(\{\mathbb{N}\}, \mathbb{N} \to \mathbb{N})$.

   The type expression means that `f` returns a thunk that can be used as a function of type $\mathbb{N} \to \mathbb{N}$ and already carries one argument of type $\mathbb{N}$ that will be passed to evaluating function. We will show how to derive such information later.

2. From that, the compiler sees that `f` must be called lazily, and creates an eagerly evaluable function that can be referenced from the thunk:

   ```
   f_eager t1 t2 = (+) t1 t2
   ```

3. It correspondingly translates the "inner" call `(f a)` in `h` to a thunk represented as a tuple $\big(\mathtt{address\_of}(\mathtt{f\_eager}), \mathtt{a}\big)$.

[4]Spineless Tagless G-Machine, the structure used to store all implicitly allocated data of programs compiled by GHC.

4. To translate the outer call, a new version of g, called `g_eager` that accepts a tuple in the above form as a second argument, is generated, and h is rewritten:

```
g_eager (fptr,a) t3 = fptr a t3
h a = g_eager (f_eager,a) a
```

   Resulting code now does not contain any lazy evaluation.

Overhead of static thunks. Compared to code inlining that would trivially solve the aforementioned example, static thunks may potentially present significant runtime overhead arising from the necessity to transfer function pointers and (possibly bulky) thunk data through the stack structures (such behavior may manifest as appearance of functions with surprisingly many arguments), and from the possibility that the thunk may get evaluated more than once. In our view, the first case is a counterweight to a similar situation present with inlining process — inlining may produce a program that runs faster, but usually for the price of code size. Allocating thunks produces possibly smaller code (because the compiler is not forced to inline all partial applications), but the required data transfers and indirections add overhead and reduce execution efficiency.

Compared to thunks allocated on heap, our approach profits from the simplicity of stack allocation, which is usually handled by one-instruction modification of a CPU register. General-purpose heap allocators may require hundreds of instructions and possibly produce several CPU cache misses.

Strictness. Strict method of evaluation is a common optimization in compilers of lazy functional languages, as it is usually cheaper to actually evaluate the result than to allocate the thunk and possibly re-evaluate it on each usage[5]. There are practical methods that automatically determine which results should be evaluated strictly, including the one from Haskell [12] or OCaml [24].

Our approach is simpler — because the need to produce specialized code with additional data transfers may impose significant performance overhead, we consider all calls strict by default, allowing lazy evaluation only when needed (i.e. where inlining was unable to remove it) or on programmer's choice.

Effect of strict evaluation on correctness. The common argument against strict evaluation — that it may prevent the program from halting — holds here. Moreover, our work with pointed types may cause a program crash if the statements are not evaluated in exact order (e.g. a null-pointer check followed by a dereference is a common construction, but may get evaluated in a reverse order if both parts were arguments of one function). We argue that our environment is not affected by those kinds of errors — all potentially harmful side effects (especially the out-of-stack memory operations) are contained in a monad environment, and we provide syntax for marking lazy code that the programmer can use to avoid possible infinite loops of eager evaluation.

[5] With the exception of popular counterexamples, including long lists that get only first few items instantiated.

$$\frac{x:\sigma \in \Gamma}{\Gamma \vdash x:\sigma}\ \text{Var}$$

$$\frac{\Gamma \vdash e:\tau \quad \sigma \in \mathtt{instances}_\Gamma(\tau)}{\Gamma \vdash e:\sigma}\ \text{Inst}$$

$$\frac{\Gamma \vdash e:\tau \quad \Gamma, x:\mathtt{close}(\tau) \vdash y:\sigma}{\Gamma \vdash \textbf{let}\ x = e\ \textbf{in}\ y:\sigma}\ \text{Let}$$

$$\frac{\Gamma, x_{1,2,\ldots,n}:\sigma_{1,2,\ldots,n} \vdash e:\tau}{\Gamma \vdash (\lambda x_{1,2,\ldots,n}.e):\textbf{thunk}(\{\varnothing\}, \sigma_1 \to \sigma_2 \to \cdots \to \sigma_n \to \tau)}\ \text{Abs}$$

$$\frac{\Gamma \vdash e:\textbf{thunk}(\Phi, \sigma \to \rho) \quad \Gamma \vdash x:\sigma}{\Gamma \vdash (e\ x):\textbf{thunk}(\Phi \parallel \sigma, \rho)}\ \text{App}$$

$$\frac{\Gamma \vdash e:\textbf{thunk}(\Phi, \rho) \quad \rho \in \mathscr{R}}{\Gamma \vdash e:\rho}\ \text{Eval}$$

Figure 1: Type inference rules based on Hindley-Milner type system. The original abstraction and application rules are modified to augment the simple function types with information about all possible thunk-evaluating functions.

2.2 Type system

The language is typed statically by a Hindley-Milner-style type inference as described by Damas and Milner [5]. While we use several non-trivial practical extensions (notably the statically defined overloading in a manner similar to that of Kaes [14] and support for recursion without explicit handling of the fixpoint operator) we only describe the minimal extension of the system that informs the compiler about the requirements for removing lazy evaluation. See Figure 1 for the modified inference rules.

In the figure, the functions `instances` and `close` are used exactly for the purposes of the original system — `close` introduces the $\forall$-polymorphism (to create a polytype), and $\mathtt{instances}_\Gamma$ removes it, by describing a set of new possible monotype instances with fresh type variables free in the basis Γ.

The information stored in a **thunk** covers

- the actual type of the function the thunk represents at current place
- one list of argument types for each thunk-evaluating function that must be able to get arguments required for its evaluation from this thunk.

Syntactically, the situation is described as follows. The structure:

$$\textbf{thunk}\left(\{(\tau_1, \tau_2, \ldots, \tau_n), (\sigma_1, \ldots), \ldots\}, \rho\right)$$

describes a thunk that returns (possibly functional) type ρ, and stores enough arguments to be evaluated either by function of type $\tau_1 \to \tau_2 \to \cdots \to \tau_n \to \rho$, or $\sigma_1 \to \cdots \to \rho$, etc. Argument lists can be empty, in that case we write them as $\varnothing$. Operation $\Phi \parallel \sigma$ in the figure produces a new set of argument lists, containing all lists from Φ with σ appended on the end.

Such set-based construction is necessary to overcome the possible incompatibility of thunks that can arise from a code similar to this:

```
succ :: N -> N
if :: Boolean -> a -> a -> a
f = succ
g x = (+) x
h a b = if a f (g b)
```

Because underlying code can not prepare the result of `h` for evaluation by derived thunk-evaluating functions of either `f` or `g`, it is necessary for the thunk to be universal. Its universal type is decided as **thunk**$(\{\varnothing, (\mathbb{N})\}, \mathbb{N} \to \mathbb{N})$. Consecutive application of a value of type $\mathbb{N}$ to the result of `h` produces **thunk**$(\{(\mathbb{N}), (\mathbb{N}, \mathbb{N})\}, \mathbb{N})$, which can be readily evaluated — the argument lists exactly correspond to the argument lists of `succ` and `(+)`. Note that it is not necessary to mark which argument list is going to be used on evaluation, as that information is also (implicitly) present in the code of the associated pointed function.

Unification of the thunk descriptions, required for the inference system to work, is done structurally in the type part, and by set union in the argument list. In our implementation, set union is handled by working with the set contents as with an separate object external to actual unification process, storing only a variable-like reference. Technical details are omitted.

There is a chance for non-determinism introduced in rules Abs and Eval, as it is not clear for the compiler whether e.g. the thunk should be evaluated on spot or passed down unmodified; or whether successive λ-abstractions should be converted to a single thunk or multiple smaller thunks that would eventually get evaluated successively. For our purpose, we use early evaluation of every thunk in the Eval case by completely disallowing already-evaluable thunks, and the all-at-once thunk construction in the Abs case. Such approach satisfies our requirements on the compiler; other alternatives were not considered as they bring "additional laziness" of the evaluation and measurement of their effect is not in the scope of this paper.

Thunk storage. Given a deterministic storage method, the size required for thunk structure can be computed at compile-time for any thunk type. Thunks can therefore be passed around as traditional function arguments and return values without any need for dynamic memory management.

Exact order in which the arguments are stored in the thunk is completely arbitrary, as long as the basic operations (addition of a new argument, set unification and extraction of any complete argument list) stay deterministic.

An easily implementable example is a tuple with just enough fields of every type so that each argument list can fit in, stored in tail-aligned order to allow easy addition of a newly applied argument to list tails.

Function specialization. After inference, function bodies specialized for thunk processing are created by a very gratifying side effect of the overloading resolution algorithm: In the same way in which e.g. an arithmetic function is specialized to support both integer and floating-point arguments, it can also be specialized to support thunks values. The difference in the code is then created on the place of the "invisible" application operator, instead of the more traditional place of an arithmetical operator.

2.3 Recursive types

Recursive types, such as auto-allocated lists and trees, form a basic building block of many current functional languages. Their removal in our language is justified by the addition of pointed ("reference") types that, like in other languages, are able to replicate the functionality to a practical extent. While other solutions for replacing the need for automatic deallocation exist (like the linear type systems), our choice is supported by following considerations:

- Almost all primitives used by a system-programming language require some computation with memory addresses (e.g. the system calls).
- Usage of first-class references is a fairly standard method to define complicated data structures.[6]
- Inclusion of pointers does not prohibit the possibility of future addition of automatic memory management by the programmer, just as with garbage collectors for C/C++ [3].
- All memory operations can be contained in a monad to produce a pure language. This can be further exploited to provide some automatic safety of memory operations, but forces the programmer to use complicated syntax for trivially-looking tasks.

As we are not aware of any officially recognized syntax that would allow to directly use pointers in a purely functional language, we reuse the `sizeOf`, `peek` and `poke` primitives from Haskell FFI library [4], that shares a similar set of goals. Allocation and deallocation functions are linked from standard C library, having type signatures `alloc :: IO (Ptr a)` and `free :: Ptr a -> IO ()`.[7]

[6] Moreover, naive data structure implementations in pure languages suffer from a fatal inefficiency resulting from the need to constantly reallocate immutable data. In those cases, non-trivial constructions such as the zipper [11] are required to produce effective code.

[7] `alloc` can decide about allocation size from previous type inference.

2.4 Recursion

The common model for transforming functional recursion to loops using tail calls fits our scheme with one exception — unbounded, non-tail recursion along a data structure will cause stack overflow much earlier than in a common functional language, where the "stack" structure is allocated dynamically on heap and the actual system stack holds only rarely-expanding structures. In case of Haskell, infinite non-tail recursion will cause a regular memory depletion error, stack overflows can happen only on evaluation of deeply-nested thunk values.[8]

The risk of unwanted stack overflows can be mitigated syntactically: Whenever the compiler would emit code that calls some function recursively using a non-tail call, i.e. when there is a loop in call graph that contains at least one non-tail call edge, it may abort the compilation and require the programmer to syntactically acknowledge the acceptance of the risk (or existence of some functionality that alleviates it) with a keyword at call site.

3 Implementation and results

Our demonstrational compiler is implemented as an expansion of the `tlc` compiler written in C++ [17]. We measure its performance in two ways: First, to measure the general performance of our approach, we compare the speed of simple implementations of two algorithms that solve relatively common problems with other compilers. Next, performance overhead of a synthetic case of static thunk usage is tested on a small program that runs an equivalent of standard Haskell `foldl` function on a list structure.

The exact test problems for first comparison are:

- Insertion of 2^{20} pseudo-random elements into a binary search tree, and
- 2^{28} rounds of TEA cipher [23].

We have created simple implementations of both test algorithms in C, Haskell and in our testing language, and compared the execution speed of the code produced by each compiler. The measurements are shown in the figure 2. Results show almost identical running times for C and our language, and some overhead for the code produced by Haskell compiler. The tiny speedup over C++ in one case was determined to be a product of complete inlining of the functional code; it was no longer measurable after forcibly inlining the C++ code by hand to a form with similar structure as the assembly produced by `tlc`

Next, thunk overhead was measured on the `foldl` function used to sum elements in a prepared linked-list structure with pseudo-random numbers. The function was run repeatedly on a list that fitted in the CPU L3 cache. Fully inlined, specialized `foldl` was able to process one list element on average in around 4.152ns, `foldl` that was passed a thunk that contained the adding function took 4.548ns. Several (less than 10) extra 8-byte arguments passed with the thunk increased the processing time of one element by around 0.1ns each.

[8] The nice example given on Haskell Wiki [9] exploits the properties of lazy implementation of `scanl`: Code `print $ last $ scanl (+) [0..1000000]` is killed in older GHC implementations as the stack gets filled by the back-references to unevaluated list elements.

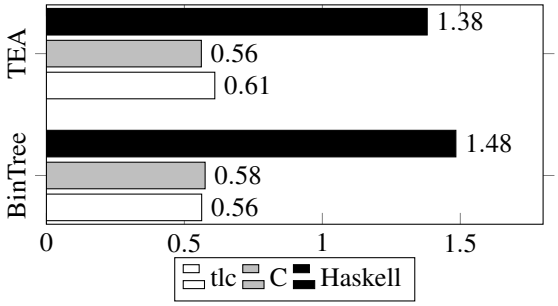

Figure 2: Performance results for the test programs, lower is better. Values are average run time in seconds over 100 runs.

We consider the overhead of indirection less than 10% in a pathological stress-test a good result, and do not expect real applications to suffer serious performance problems. Overhead of passing reasonably-sized thunk contents around seems similarly inconsequential.

All tests were measured on Intel®Core™i5-4200M CPU at 2.50GHz, used C compiler was standard Debian GCC of version 5.3.1-19, Haskell compiler was GHC version 7.10.3.

3.1 Drawbacks and unsolved problems

The main drawbacks of the new language originate in the new combination of system-level problems with purely functional syntax. Suitable solutions for some problems listed below are not yet established, and are slightly more interesting as a question of language design than of actual compiler functionality. We present the most interesting problems as open questions:

Destructors. Finding a good spot for automatic release of resources held by local variables is not very straightforward in a functional language, as the concept of imperative code blocks that correspond to variable validity in C-like languages is not extensible to the monad environment.

A related approach that merely prevents the programmer from forgetting to deallocate a structure (thus creating a memory leak) is the bracketing pattern, known for example from Python as the `with` construction, or as `bracket` from Haskell `Control.Exception` library. For our purpose, bracketing is composable with monads [8] to form an illusion of variable-holding program-blocks that resemble well-accepted syntax from procedural programming.

Similar problems arise with temporary data structures required for sub-results of certain operations: For instance,

the multiplication of 3 matrix objects (`a*b*c`) usually implicitly allocates space for sub-result (`a*b`). In a functional environment, `*` has to be replaced by some monadic construction to allow access to impure primitives (e.g. heap memory operations) required to allocate the space for sub-results.

Pointers to stack. A common method to allocate and deallocate a small structure needed for communication with a system- or a library-call is to allocate it dynamically on the stack and only pass a pointer; a great example of such structure is `struct timeval`. In a functional language, careless usage of such pointer may lead to a program crash — because the stack management does not necessarily correspond to variable scope as in C-like languages, no dependency that would hold the structure in place until is implicitly materialized and the pointer may easily become dangling.

List processing. Easy list traversal and recursive structure comprehension is one of the main features of the functional programming languages. Similar functionality may also exist in a low-level language that forbids the (required) recursive types: Pattern matching on list structures, as known from Haskell, can be also applied to pointers. Common functional allocation of list structures (e.g. the well-known function pattern that returns list tail with a new prepended head) can be either replaced by allocation-free generators similar to `Data.Traversable`, or redesigned to work on STL-like list structures that are implicitly allocated by well-hidden language construction.

4 Conclusion

We have presented a new method to compile a pure functional language to low-level code suitable for system-programming. Main improvement, described in section 2.2, is the inference algorithm that allows to transform lazily-evaluated functional code to a form that does not require automatic memory management for run-time allocation of thunks. Resulting compiler is comparably simple, fast, and produces code that does not require any run-time support, shows no significant performance drawbacks when compared to code generated by current C compilers, and allows basic usage of high-level constructions known from current functional languages.

While the language is not yet very pleasant to work with, mainly because of the drawbacks that are mentioned in section 3.1 and lack of well-developed standard library, the code of the test programs is concise, shows no unnecessary complexity, are is readily comprehensible by any functional-aware programmer. Moreover, as described in section 2, the language allows direct linking with many existing libraries through standardized calling conventions, which may be easily exploited for further applications.

4.1 Future research

Apart from the demonstration of the original goal, the new language opens possibilities to apply high-level optimization methods to programs that intend to run on bare hardware, possibly yielding better results than the optimizers of Haskell that are encased in a pre-defined scheme of memory management, or the optimizers of C++ which are constrained by aliasing problems arising from impurity of the language.

An example of approaches that aggressively modify and restructure the subroutine structure of the program is the work of Danvy and Schultz [6] that shows beneficial impact of possibly non-deterministic combination of argument dropping, lifting and inlining. Similarly, purity of the code allows for a more efficient elimination of common sub-expressions and repeated code. Practical impact of both techniques is yet to be measured.

Newly added program control possibilities, mainly the memory-related primitives described in section 2.3, create many new chances for programmer errors. While current type checkers can discover many errors on compilation, more complicated type systems or verification approaches could be able to check e.g. actual memory safety or related constraint satisfaction. We hope to implement and test the ideas from the Sage language [16] that allows automatic addition of runtime checks for constraints that the compiler was not able to satisfy by static checking; or some of the work of Roorda [21], which allows to use a quite powerful concept of pure type systems in a practical environment.

References

[1] Richard Bird, Geraint Jones, and Oege De Moor. More haste, less speed: lazy versus eager evaluation. *Journal of Functional Programming*, 7(05):541–547, 1997.

[2] Guy E Blelloch and Robert Harper. Cache efficient functional algorithms. *Communications of the ACM*, 58(7):101–108, 2015.

[3] Hans Boehm, Alan Demers, and Mark Weiser. A garbage collector for C and C++, 2002.

[4] Manuel MT Chakravarty, Sigbjorn Finne, F Henderson, Marcin Kowalczyk, Daan Leijen, Simon Marlow, Erik Meijer, Sven Panne, S Peyton Jones, Alastair Reid, et al. The Haskell 98 foreign function interface 1.0, 2002. *Available onlin e: http://www. cse. unsw. edu. au/ chak/haskell/ffi.*

[5] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 207–212. ACM, 1982.

[6] Olivier Danvy and Ulrik P Schultz. *Lambda-dropping: transforming recursive equations into programs with block structure*, volume 32. ACM, 1997.

[7] Damien Doligez and Xavier Leroy. A concurrent, generational garbage collector for a multithreaded implementation of ML. In *Proceedings of the 20th ACM SIGPLAN-SIGACT*

symposium on Principles of programming languages, pages 113–123. ACM, 1993.

[8] Cale Gibbard. Bracket pattern. `https://wiki.haskell.org/Bracket_pattern`, 2008. Accessed: 2016-05-10.

[9] Stack Overflow - Haskell Wiki. `https://wiki.haskell.org/Stack_overflow`. Accessed: 2015-12-21.

[10] The High Assurance Systems Programming Project HASP. *The Habit Programming Language: The Revised Preliminary Report*. Department of Computer Science, Portland State University Portland, Oregon 97207, USA, November 2010.

[11] Gérard Huet. The zipper. *Journal of functional programming*, 7(05):549–554, 1997.

[12] Kristian Damm Jensen, Peter Hjæresen, and Mads Rosendahl. Efficient strictness analysis of Haskell. In *Static Analysis*, pages 346–362. Springer, 1994.

[13] SL Peyton Jones, Cordy Hall, Kevin Hammond, Will Partain, and Philip Wadler. The Glasgow Haskell compiler: a technical overview. In *Proc. UK Joint Framework for Information Technology (JFIT) Technical Conference*, volume 93, 1993.

[14] Stefan Kaes. Parametric overloading in polymorphic programming languages. In *ESOP'88*, pages 131–144. Springer, 1988.

[15] Richard A Kelsey. Pre-Scheme: A Scheme dialect for systems programming, 1997.

[16] Kenneth Knowles, Aaron Tomb, Jessica Gronski, Stephen N Freund, and Cormac Flanagan. SAGE: Unified hybrid checking for first-class types, general refinement types, and dynamic (extended report), 2006.

[17] Miroslav Kratochvíl. Low-level functional programming language. Master's thesis, Charles University in Prague, 2015.

[18] Xavier Leroy. Unboxed objects and polymorphic typing. In *Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 177–188. ACM, 1992.

[19] Avery Pennarun, Bill Allombert, and Petter Reinholdtsen. Debian popularity contest, 2012.

[20] Simon Peyton Jones and Simon Marlow. Secrets of the glasgow haskell compiler inliner. *Journal of Functional Programming*, 12(4-5):393–434, 2002.

[21] J-W Roorda and JT Jeuring. Pure type systems for functional programming. 2007.

[22] Michel Steuwer, Christian Fensch, Sam Lindley, and Christophe Dubach. Generating performance portable code using rewrite rules. 2015.

[23] David J Wheeler and Roger M Needham. TEA, a tiny encryption algorithm. In *Fast Software Encryption*, pages 363–366. Springer, 1994.

[24] Hirofumi Yokouchi. Strictness analysis algorithms based on an inequality system for lazy types. In *Functional and Logic Programming*, pages 255–271. Springer, 2008.