Static closures for functional languages

Miroslav Kratochvíl* and David Bednárek

Department of Software Engineering,
Faculty of Mathematics and Physics, Charles University, Prague
{kratochvil,bednarek}@ksi.mff.cuni.cz

Abstract. In current functional programming environments, the first-class functions with implicitly passed closures are usually represented as structures called thunks. Typical methods of thunk processing require non-trivial run-time support in form of allocation and garbage collection, which may negatively impact performance and restricts applicability of the language in low-level environments, such as embedded devices, operating systems and kernels. We present an algorithm that statically converts all implicitly passed function scopes (that would otherwise be converted to thunks) to explicitly defined static closures. The approach can be used to simplify translation of purely functional programs to highly-performant lower-level languages. Performance impact of the conversion and subsequent compilation to low-level code is demonstrated on several programs.

1 Introduction

Functional programming languages have been proven extremely useful for efficiently expressing complicated tasks: Layering of abstractions is easy, purity of the language reduces negative impacts of code complexity, compilers make good use of absent side-effects to support high-performance optimization targets (e. g. parallelization [14] or cache obliviousness [3]), and static type inference benefits type-dependent expressivity and prevents many sources of errors in the code.

To support the levels of abstraction, such programming environment needs to manipulate objects that are not directly representable on the target hardware platform, including generics, infinite data structures, first-class functions, and lazy values. This article is focused on the handling of the lazy values and functions.

In most current functional-programming environments, lazy values and partially applied functions are represented as dynamically allocated *thunks* (e. g. the STG nodes in Haskell [10]) that carry the information about function implementation, its environment and parameters. In most cases, the thunks exhibit a degree of variability that induces frequent use of small individually allocated memory blocks for each different thunk, which, together with the complexity of

* This work was supported by project SVV 260451.

their mutual references, in turn implies the need for garbage-collection mechanism. Heavy reliance on dynamic allocation degrades performance and limits the applicability of functional programming in environments where resulting unpredictable memory accesses are undesirable.

As the main contribution of the article, we present a code transformation that rearranges the use of first-class function values in a Haskell-like functional programming language so that dynamic allocation is not required for thunk processing, and must be used only in cases which intrinsically require it, i.e. for handling of recursive structures. The functional values are replaced by statically defined low-level closure-representing structures. Knowledge of the static definition then allows the compiler to use standard machine stack (or even registers) for passing the structures among functions. In the cases when thunk structures are recursive or infinite and thus not expressible statically, the programmer is given direct control of their allocation and deallocation.

Transformed programs have the same semantics and complexity as the original, but can be evaluated by a simpler run-time — the evaluator can be completely eager and does not need to support operations on first-class functions and closures or partial applications, since all functions are specialized for all contexts of their occurrences in code and their representations are static structures defined from machine-level types. Resulting code can therefore be almost directly translated to systems as simple as C or LLVM, gaining performance and eliminating large parts of language run-time.

1.1 Approach

A common way to represent a first-class function-value in a low-level language is a function pointer that references machine code that evaluates the intended function. Take the functional-language representation of combinator `k' = \x y -> y`: If the programmer needs to store it in a variable `f = k'`, we would like the compiler to materialize a function that takes two arguments and returns the latter (thus evaluates `k'`), and use the memory address `a` of that function's code to represent the value of `f`.

To store the environment of the closure, we wrap the function pointer in a simple structure: If the programmer writes `t = f p` to apply an argument to the function and store the result, we store `t` as a tuple `(a, p)`. We will call this tuple a *static closure*, `a` will be called a pointer to the *evaluating function*, other entries of the tuple will be called *environment*.

Such structures can be evaluated lazily or eagerly. After applying next argument, `t q` is represented by tuple `(a, p, q)`, which can be either stored lazily with no modification, or *forced* on the spot by calling the pointed function with the two arguments: `a p q` should evaluate to `q`.

Term `t` could be even evaluated partially: If the function pointed by `a` was curried to take only one argument and return some other static closure, we could force `t` on the spot. In this case, we would likely receive a closure with pointer to an identity function, formed from the lambda `\y -> y` from `k'`.

Figure 1 shows basic examples of the input and output of the proposed code

transformation algorithm: A simple program in fig. 1a is compared to a program with a slight modification of strictness in fig. 1b, using keywords `strict` and `lazy` that the algorithm recognizes for this purpose. Results in figs. 1c and 1d show programs converted to use the static closures — first-class function closures are replaced by tuples; function without any environment are represented by the function pointers. Language primitives `makeFPtr` and `fPtrCall` serve for introducing and using the pointer values.

At each point in a program, the occurring closures may have different evaluating functions and different environments. Main concern of the algorithm is to statically eliminate those differences, so that no extra processing or overhead is necessary at the runtime, except for storing the derived of the static closure, and actual computation. This is achieved by introducing *universal* forms of the closures: The algorithm detects sets of closures that are required to be identical at some point in the program (e. g. they must be stored in a single variable) and unifies their environment to a single statically defined form. Two distinct cases when such unification happens are shown in figs. 1e and 1f. Similarly, partial evaluation of the function object requires that its evaluating function is (partially) curried; which in turn requires that the whole program must be prepared to force the function object using this curried function, as in fig. 1g.

Algorithm setting We present the algorithm as a compiler step for a generic functional programming language: The code it receives is given in binding group of type-checked explicit bindings as defined and used by Jones [9, section 11.6.1] for the presentation of Haskell type system. We expect the conditionals and lambdas to be desugared to pattern matching on binding alternatives [9, section 11.4], all bindings lifted to a single top-level group [5, 10] and all functions specialized to monomorphic form [8, section 4.7].

The algorithm recognizes two extra keywords that modify the semantic meaning of the program: `strict` and `lazy` can be used to explicitly mark code that should get (partially) evaluated on that place, resp. code that should not get evaluated until it is explicitly required. We expect that previous strictness analysis has produced `strict` keywords automatically.

The analysis of required closure contents and properties is performed by graph-rewriting-based method described closer in section 2. Final graph structure contains information about fully universal forms of static closures and is used in section 3 to reconstruct the original binding group. Language primitives `makeFPtr` and `fPtrCall` are added to the code to allow usage of function pointers in functional language.

Output The main property of the generated code is that its evaluation does not implicitly require dynamic allocation or garbage collection. In fact, because handling of first-class functions is completely removed from the result, it is possible to execute it on a low-level stack-based eager evaluator (much simpler that e. g. the STG of Haskell) — in section 4 we provide an example of direct one-to-one translation of resulting code to C language.

```
f a = (+) (a*2)
g a b = strict (f 3 (a+b))
h a = g a a
```

(a) Simple example to be converted.

```
f a = lazy ((+) (a*2))
g a b = f 3 (a+b)
h a = strict (g a a)
```

(b) Same code with delayed computation.

```
f' a v = (+) (a*2) v
f a = (makeFPtr f', a)
g a b = let (f1,v1)=f 3
        in (fPtrCall f1) v1 (a+b)
h a = g a a
```

(c) Original code with a simple static closure.

```
f' a v = (+) (a*2) v
f a = (makeFPtr f', a)
g a b = (f 3, (a+b))
h a = let ((f1,v1),v2) = g a a
      in (fPtrCall f1) v1 v2
```

(d) Evaluation of the integer is delayed until explicitly requested.

```
f fn = (g fn, h 3 fn)
g fn = fn id
h n fn = fn ((+) n)
test = f (\a -> a (a 0))

-- evaluator for the lambda:
lam' (f1,v1) = fPtrCall f1 v1
               (fPtrCall f1 v1 0)
```

(e) Copying a closure. Evaluating function of the lambda in `test` must be prepared to handle both cases: It might receive parameter `a` that requires no environment (evaluating function of `g` discards `v1`) or one that must remember an integer.

```
f1 :: Char -> Int -> Int
f2 :: Int -> Int -> Int
f 0 = f1 'a'
f n = f2 n

-- converted code:
f1' a b c = f1 a c
f2' a b c = f2 b c
f 0 = (makeFPtr f1', 'a', 0)
f n = (makeFPtr f2', 0, n)
```

(f) Using two different closures of the same type interchangeably. The environment must then be able to contain different scopes of both evaluating functions, which are modified to simply ignore unused variables in the environment.

```
g True f a = (strict (f a)) a
g False f a = strict (f a a)
```

(g) Function `f` will be requested to partially evaluate at two distinct points: first alternative of `g` partially evaluates `(f a)` to receive a closure to which `a` is then attached. Second alternative must run the partial evaluation twice — the contained evaluating function is first used to get the same closure as in `g1`, resulting closure is then called with a new parameter to obtain the final result.

Fig. 1: Examples of handling of static closures.

The simplicity is granted by conversion of all function objects to previously described form of static closures: Closures with derivable static form are converted to low-level types, recursive closures that can not be represented using a regular statically-sized data structure[1] are broken to representable parts using indirection. Since the language run-time must not cause any implicit overhead, the compiler identifies all places where the indirection is inevitable and gives control to the programmer, who provides any customized implementation of the indirection.

1.2 Related work

Presented approach is related to defunctionalization [2, 4] in the way of preventing run-time creation of thunk structures, but differs in the treatment of laziness that is more suited for a call-by-need language, and in the structure of the closures that do not require additional matching logic at the call site. Also, our tuple-based structure of closures with plain function pointers vastly simplifies cross-module defunctionalization [7].

Compilation to low-level code is a viable target for performance-centric functional programming, e. g. Shaikhha et al. [13] present a small, Turing-incomplete array-centric subset of F language that is compiled into high-performance low-level code, in conceptually similar way as the prototype `tlc` language described in our previous work [11]. Note that closure handling in `tlc` is also limited — its closures can only contain very simple user-defined data types and fail in many recursive cases, and also does not support explicit strictness annotations.

The rest of the paper is organized as follows: Detailed description of algorithm that infers exact content of static closures is given in section 2. Process of using the information gathered by the algorithm to generate code of a new program is detailed in section 3. Examples of the algorithm result and measurements of performance impact are available in section 4. We discuss the results and conclude in section 5.

2 Inference of static closure structures

This section describes the derivation of a *closure graph*[2] that specifies environments of each closure and result of each evaluation in the code. The graph represents flow of the closures in the program: It contains *data nodes* that represent values of expressions in the program together with their type, *data edges* that connect data nodes to their position in containing structures, *application edges* that represent the parameter-applying operation (for reasons of recursion explained later, those are marked either as direct or indirect), and *forcing edges* that represent evaluation of the closures. The components are visualized in fig. 2.

[1] Recursive lookup functions that pack up undetermined number of key-value pairs provide a great example:
`addLookup f key val = \s -> if key==s then val else f s`

[2] Technically, the structure should be called a tagged hypergraph.

Each syntax element of the program is assigned an associated structure in the graph: Identifiers and literals 'remember' their associated data nodes, function applications are associated with an application edge with a resulting connected data node, and `strict` keywords are associated with forcing edges. This association is later used to look up exact form of the closure that occurs at that point in the code: Data node that is a result of application (i. e. has an incoming application edge) is a static closure that must store the closure from the left side of application together with the parameter from the right. Data node without incoming application edge is either a function pointer if its type is functional, or plain data otherwise. Forcing edges represent evaluation of the closure: their generated code takes the closure structure apart and calls the pointed evaluating function with all arguments. Conditions that allow static and unambiguous derivation of such closure-processing code are then viewed as prohibited graph patterns that must be 'repaired', as described in section 2.1.

Preprocessing To simplify the treatment of strictness later in the process, we add explicit forcing to places where it is semantically required: For all externally linked functions (that cannot process closures), built-ins (e. g. `(+)` can only work with plain integers) and pattern matching on alternatives we create a wrapper that explicitly forces all required arguments using `strict` keyword.

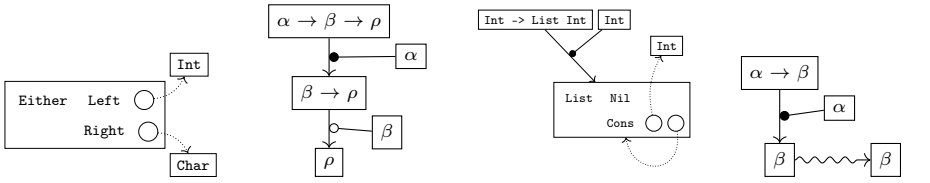
Strictness annotations are processed as follows: `lazy` keyword must ensure that inner structure will not be evaluated until the result is required; we therefore lift [5] the parameter of `lazy` to form a new function that will serve as an evaluating function for the delayed computation, and `lazy` keyword itself is erased. Similarly, parameter of each `strict` keyword is lifted to form a new function that will be called upon partial evaluation. `strict` keyword is inserted to the top of the lifted function to explicitly mark that its result must be forced.[3]

2.1 Graph rewriting algorithm

Closure graph is derived iteratively: First, functions that are known to be certainly used by the program (e. g. `main` or module-exported functions) are *annotated* — their syntax tree is connected to corresponding newly created graph structures. Graph structure is then corrected by the *rewriting rules* to obtain a new graph with closures that are universal for all their uses. If some new, not-yet-annotated functions are referenced from the rewritten graph, algorithm adds them and reiterates from the first step; otherwise it finishes. Termination of the process is assured by limiting possible added variants of each function.

Function annotation Each object in the function syntax tree is assigned a corresponding object in a newly created graph structure: Identifiers and literals are mapped to data nodes. Types of the values are stored in data nodes for future reference. Data nodes of structured types do not contain the types directly, but

[3] That also prevents the inner function from returning another closure, as in fig. 4.



(a) Data nodes and edges that form a value of type `Either Int Char`.

(b) Successive direct and indirect application edges.

(c) The complete set of nodes that represent a function of type `Int -> List Int`.

(d) Forcing edge between data nodes.

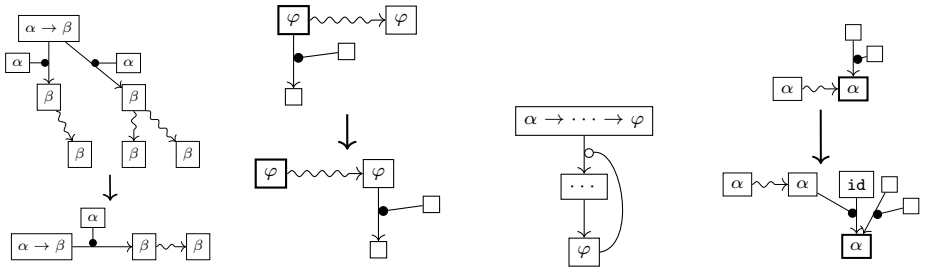
Fig. 2: Basic structures in the closure graph.

are connected to data nodes that describe their fields using data edges (see fig. 2a). Applications and their results are assigned direct application edges that form a connection of two corresponding data nodes to form a new, 'result' data node (see fig. 2b). Finally, `strict` keywords are annotated: If the function is strict, its resulting data node is copied and connected to the original using a forcing edge, as shown on fig. 2d.

Graph rewriting Target of the rewriting is to find a finite number of closure structures that will be universal for all their possible usages in the program. The process is expressed by following rules that may be used to direct pattern matching and rewriting of the graph structure.[4]

- *Statically defined data.* If two data nodes represent the same piece of data in the program (i. e. two occurrences of the same variable, variable used as a function call parameter and corresponding function-internal variable, or data constructor argument and corresponding data field), their closure form must be equal. Their nodes are therefore *merged* into one, all references (edges and references from code) that pointed to the old nodes are moved to the new one. As a result of merging, a field of a data type may have multiple outgoing data edges which must be merged as well. As a special case, a non-functional value may be required to be used as a closure; in that case we convert it as a closure evaluated by `id` function, as on fig. 3d.
- *Deterministic closure processing.* A closure must be applied to a deterministic set of arguments and evaluated with exactly defined environment. To enforce a single way of processing a given closure, we require that it has at most one outgoing edge that describes its modification (i. e. 'left' application or a forcing edge). Duplicate edges are fixed by recursive merging, as shown on fig. 3a. If both types of edges originate in one data node, parameter application is delayed after forcing, as in fig. 3b. Similarly, unnecessary repeated

[4] The extended version of this paper, available at `http://e-x-a.org/mff/papers/2017-scopes.pdf`, provides more details and examples of the rewriting process.



(a) Closure processing must be deterministic, branches on evaluation paths and corresponding arguments are merged.

(b) Ambiguity of evaluation path for the highlighted closure is resolved by moving the application edge.

(c) Recursive environment is made finite by indirecting the application edge.

(d) Result of forcing (highlighted) that is required to become a closure is lifted to an argument of `id`.

Fig. 3: Graph corrections used in rewriting process.

forcing is removed.

– *Finite closures.* A simplified program that produces a closure that may grow non-deterministically is shown in fig. 6. Such cases can be easily detected as cycles in the graph of (direct) application edges, and solved by breaking the loop by marking at least one of the application edges in the cycle indirect, as shown on fig. 3c. The choice of a good loop-breaker may be viewed as optimization of various trade-offs between static size of the closure and amount of indirection in the program; rigorous methods to make the choice are out of the scope of this paper.

Spawning of evaluating functions We now need to prepare bodies of the functions that will be called by the forcing edges in the graph. Complete list of required functions can be constructed by examining originating data nodes of all forcing edges: Functions with identifiers referenced from the left-application chains that lead to these data nodes are exactly the functions that will be required to evaluate the corresponding closures of the data nodes. New variants of the functions are created simply by copying the original function definitions and marking them strict. If the forcing requires more arguments than the function definition has, extra arguments are added by η-conversion.

To prevent infinite loops on recursive functions, the algorithm keeps track of function variants that have been already spawned and merges some of them to keep the count of variants finite. Choice of a good finite set again introduces an

optimization trade-off between large environments of overly generic evaluating functions and too many different evaluating function variants. For purposes of this paper, we simply restrict the environments of evaluating functions to not contain closures evaluated by their own variants.

3 Generating code with static closures

At this step, we use the associations between graph nodes and code elements to rewrite the program code for handling the exact environments of the closures. Remaining concerns are to choose a fitting low-level representation for the resulting closures, and to produce code that handles the indirected environments.

Closure representation. To allow both output to functional code and easy compilation to low-level languages, we construct the closure structures as simple tuples: A data node without environment is represented either by function pointer or actual value (depending on whether its type is functional or not). If it has environment, i. e. is an incoming result of a direct application edge, it is represented by a tuple `(l,r)` where `l` and `r` are recursively generated representations of left and right data nodes. Indirect application edges differ by containing a pointer to `r` instead, their manipulation is detailed later.

Representation of a data node with multiple incoming application edges is required to store any one of incoming closures, which is best represented by union type[5]. In the code examples (this case arises e. g. in fig. 1f) we instead use tuples that contain just enough arguments of each type to store any of the closures — that approach produces slightly larger environments, but removes the need to store the union tag along with the union, and simplifies disposal of the structure in case it contains linearly restricted types.

Generating code. The binding group is updated to reflect handling of the static closures: For each function, the updated type can be derived from environments of incoming and resulting data nodes. Function-referencing data nodes create the corresponding function pointer using `makeFPtr`. Applications are converted to construct the closures that match the environment in the graph. `strict` keywords are converted to function calls using `fPtrCall`.

Environment indirections. Figure 6 shows code that triggers indirect application together with its conversion to static closures. When an indirect argument is added to the closure, it is converted to a reference by a call `storeCl`, and symmetrically, when the arguments are popped from the closure structure for evaluation, the indirected closure is fetched by `loadCl`.

Functions `storeCl` and `loadCl` must be prepared by the programmer in advance. Both are expected to be monadic [15] to be able to store the closure externally, which presents a semantic problem if the affected function is not a monad that would support the offloading. For simplicity, we currently require

[5] Functional languages contain unions implicitly as variants of ADTs.

the programmer to rewrite corresponding functions to return monads, which allows composition with compiler-added code. That approach is not extremely inconvenient, but certainly removes some of the functional simplicity of the code — lifting the affected functions to monads automatically is possible using e. g. an approach similar to `each` package of Template Haskell [1]; details of such conversion are out of scope of this article.

4 Results

We have previously measured the positive impact of explicit, statically defined thunks — indirection of the call address and copying of environment has been shown to have inconsiderable impact on the speed of low-level function calls (especially when combined with some trivial low-level optimization), but provided clear advantage when compared with thunks allocated on heap by the optimized run-time of GHC compiler [11].

To supplement the previously provided examples with a practical usage of static closures, we add an excerpt of a common exploitation of first-class functions converted to low-level code — a generated infinite list in fig. 5.

		`-O0`	`-O3`
`sum`	`ghc`	5.071	0.038
	s. c.	4.135	0.206
	`tlc`	0.731	0.047
	`gcc`	0.155	**0.008**
`lookup`	`ghc`	0.831	0.168
	s. c.	0.499	0.302
	`tlc`	0.316	**0.089**
	`gcc`	0.184	0.094

Table 1: Average execution time of the example programs in seconds, with all optimizations turned off (`-O0`) resp. on (`-O3`).

Benchmarks To obtain an estimation of the performance impact of conversion, we have measured the examples from figs. 4 and 6 that were compiled by standard GHC compiler, both before and after conversion to static closures. Haskell does not specify function pointers, we have therefore replaced `fPtrCall` and `makeFPtr` by `id` to obtain the same functionality.

Test cases[6] were selected to put stress on the closure-handling mechanism: Program `sum` computes sums of successive integers up to 10^4 using lazy evaluation for each computation step and checks the result; program `lookup` uses the method from the second figure to store 10^4 integers and then look up all of them. Test cases show programs compiled by GHC compared to programs converted to static closures and then compiled by GHC (in rows marked 's. c.'). Measurements in table 1 show somewhat surprising performance gain of our method when GHC is run only with default options (i. e. without optimizations, in column `-O0`). 'Native' closures are faster when optimizations are enabled, which is explained by difficulties of optimizing function application 'hidden' in data structures. Further experiments are required to determine whether the performance difference would disappear if the transformation was applied later in the

[6] We have used standard GHC version 8.0.1, GCC version 6.3.0 (also used to process `tlc` output), tests were running on Linux 4.9.30 on Intel i5-4200M CPU.

```
--original sum is lazy by default:
sum 0 a = a
sum n a = sum (n-1) (a+n)
test = strict (sum 10 0)

--converted to static scopes:
id' _ b = b
sum' a b =  --run while loop
  let ((f1,v1), v2) = sum a b
  in fPtrCall f1 v1 v2
sum n a = ((makeFPtr sum', n-1), a+n)
sum 0 a = ((makeFPtr id',0),a)
testStrict =  --start recursion
  let ((f1, v1), v2) = sum 10 0
  in fPtrCall f1 v1 v2
test = testStrict
```

Fig. 4: Successive conversion of a lazy recursive function. Explicit forcing is added to the lifted and evaluating function code so that it can handle the recursion without external assistance, using an efficient tail call.

```
--original:
fibs = fibsL 0 1
fibsL a b = Cons a $
            lazy (fibsL b $ a+b)
(Cons a _) !! 0 = a
(Cons _ b) !! n = b !! (n-1)

--converted code:
fibsL a b = Cons a
  ((makeFPtr fibsL',b) a+b)
fibsL' p1 p2 = fibsL p1 p2

--specialization of (!!) for fibs:
(Cons a _) !! 0 = a
(Cons _ ((f1,v1),v2)) !! n =
     ptrCall f1 v1 v2 !! (n-1)
```

Fig. 5: Simulation of infinite data structures and generators is a prime benefit of lazy values.

```
--original:
addLookup f v =
  return (\s -> if v == s
                then True
                else f v)
emptyLookup = return (\_ -> False)
test = do
  l <- addLookup emptyLookup 1
  l <- addLookup l 2
  return l 3

--converted to static scopes:
addLookup f v = do
  v1 <- storeCl f
  return
    ((makeFPtr lambda1', v), v1)
emptyLookup = return
  ((makeFPtr lambda2', 0),
   nullCl)
lambda1' p1 p2 p3 =
  if p1 == p3 then return True
  else do
    ((f1,v1),v2) <- loadCl p2
    return (fPtrCall f1 v1 v2 p3)
lambda2' _ _ _ = return False
test = do
  l <- addLookup emptyLookup 1
  l <- addLookup l 2
  return let ((f1, v1), v2) = l
         in fPtrCall f1 v1 v2 3
```

Fig. 6: A closure directly contains itself: Lookup function shown here is a monadic version of the example from introduction. Non-monadic lookup function would cause compile error; monadic one can be joined with side-effect indirection of closure data. The converted source is shown re-sugared for brevity. Programmer can provide any implementation of `storeCl`, `loadCl`. `nullCl` is required only as a default value for empty fields specific to our implementation of unions.

```
struct thunk {
  bool (*f)(int, struct thunk*, int);
  int a;
  struct thunk* b;
};
bool lam1(int a, struct thunk*b,
          int c) {
  if (a==c) return true;
  return b->f(b->a, b->b, c);
}
bool lam2(int a, struct thunk*b,
          int c) {
  return false;
}
```

```
struct thunk addLookup(
    struct thunk t, int v) {
  struct thunk *store
    = malloc(sizeof(struct thunk));
  *store = t;
  return {lam1, v, store};
}

struct thunk emptyLookup() {
  return {lam2, 0, 0};
}
```

Fig. 7: Lifted lambdas and the `addLookup` and `emptyLookup` function translated to C from the converted source of fig. 6. Call to `malloc` and assignment in `addLookup` is one (unsafe) possibility of implementation of `storeCl`.

compiler chain.

For completeness, we also include timings of the same programs re-implemented in plain C compiled by `gcc`, and interesting results from an experimental version of the aforementioned `tlc` compiler: `tlc` sacrifices automated memory management and memory safety, but translates the functional program to eagerly evaluated C-style code (excerpt of the result is shown in fig. 7).

5 Conclusion and future research

We have presented an algorithm that converts implicitly-passed function scopes in a purely functional program to explicit structures that store function arguments. The approach removes the need to dynamically allocate thunks during code evaluation; in cases where dynamic allocation is unavoidable, it provides a way to control it, allowing the programmer to use allocation independently on language run-time. Most importantly, because resulting code does not contain any partial application or first-class function objects, a completely eager evaluator (similar to C or LLVM) is sufficient to run the resulting program.

Compared to the method of evaluation used in Haskell, our approach tries to dynamically allocate as few thunks as possible — all closures with derivable static form can be safely stored on program stack. Reduction of the allocator pressure and laziness-related processing in program run-time has an interesting impact on evaluation performance.

Passing the resulting structures entirely through the stack might be fast, but is counter-weighted by stack growth. Still, the increase is only linear, as size of thunks is effectively limited by the size of the program. We expect that putting a limit on a maximal stack size is a next requirement for practical low-level functional programming — one possible approach is a removal of unbounded recursion (e. g. by CPS conversion [6] and offloading the closures to heap); stack usage of such programs can be restricted statically.

Safe memory management in the new environment is still a concern. Morris [12] explores linear-type-based approach to memory safety of functional programming languages based on `Un` typeclass that represents unrestricted types. The fact that static closures are held together by simple, "built-in" types makes them easily adaptable to be an input linear type checking; removal of the function objects from the program also further simplifies linearity analysis by completely omitting the treatment of function behavior using `Fun` typeclass and related entailment. As a result, both general data and closure content can be easily managed by auto-generated code that calls methods of `Un`, which we expect to insert to the code similarly as functions `storeCl` and `loadCl`.

References

[1] The `each` package. `http://hackage.haskell.org/package/each` (2017), accessed: 2017-06-29

[2] Bell, J.M., Bellegarde, F., Hook, J.: Type-driven defunctionalization. In: ACM SIGPLAN Notices. vol. 32, pp. 25–37. ACM (1997)
[3] Blelloch, G.E., Harper, R.: Cache efficient functional algorithms. Communications of the ACM 58(7), 101–108 (2015)
[4] Danvy, O., Nielsen, L.R.: Defunctionalization at work. In: Proceedings of the 3rd ACM SIGPLAN international conference on Principles and practice of declarative programming. pp. 162–174. ACM (2001)
[5] Danvy, O., Schultz, U.P.: Lambda-dropping: transforming recursive equations into programs with block structure, vol. 32. ACM (1997)
[6] Davis, M., Meehan, W., Shivers, O.: No-brainer cps conversion (functional pearl). Proceedings of the ACM on Programming Languages 1(ICFP), 23 (2017)
[7] Fourtounis, G., Papaspyrou, N.S.: Supporting separate compilation in a defunctionalizing compiler. In: OASIcs-OpenAccess Series in Informatics. vol. 29. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik (2013)
[8] Hall, C.V., Hammond, K., Peyton Jones, S.L., Wadler, P.L.: Type classes in haskell. ACM Transactions on Programming Languages and Systems (TOPLAS) 18(2), 109–138 (1996)
[9] Jones, M.P.: Typing haskell in haskell. In: Haskell workshop. vol. 7 (1999)
[10] Jones, S.P., Hall, C., Hammond, K., Partain, W., Wadler, P.: The Glasgow Haskell compiler: A technical overview. In: Proc. UK Joint Framework for Information Technology (JFIT) Technical Conference. vol. 93 (1993)
[11] Kratochvíl, M.: Compiling functional code for system-level environment. In: Proceedings of the 16th ITAT Conference Information Technologies - Applications and Theory, Tatranské Matliare, Slovakia, September 15-19, 2016. pp. 3–10 (2016), http://ceur-ws.org/Vol-1649/3.pdf
[12] Morris, J.G.: The best of both worlds: linear functional programming without compromise. In: Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming. pp. 448–461. ACM (2016)
[13] Shaikhha, A., Fitzgibbon, A., Peyton-Jones, S., Vytiniotis, D.: Using destination-passing style to compile a functional language into efficient low-level code (2017), in submission
[14] Steuwer, M., Fensch, C., Lindley, S., Dubach, C.: Generating performance portable code using rewrite rules: From high-level functional expressions to high-performance opencl code. ACM SIGPLAN Notices 50(9), 205–217 (2015)
[15] Wadler, P.: Monads for functional programming. Advanced Functional Programming pp. 24–52 (1995)