Lucrative Late Lambda Lifting

Sebastian Graf

October 8, 2018

1 Introduction

2 Transformation

Lambda lifting is a well-known technique [2]. Although Johnsson's original algorithm runs in wort-case cubic time relative to the size of the input program, Morazán and Schultz [5] gave an algorithm that runs in $\mathcal{O}(n^2)$.

Our lambda lifting transformation is unique in that it operates on terms of the *spineless tagless G-machine* (STG) [1] as currently implemented [3] in GHC. This means we can assume that the nesting structure of bindings corresponds to the condensation (the directed acyclic graph of strongly connected components) of the dependency graph. TODO: less detail? less language? Additionally, every binding in a (recursive) **let** expression is annotated with the free variables it closes over. The combination of both properties allows efficient construction of the set of *required* TODO: any better names? former free variables, abstraction variables... variables for a total complexity of $\mathcal{O}(n^2)$, as we shall see.

2.1 Syntax

Although STG is but a tiny language compared to typical surface languages such as Haskell, its definition [3] still contains much detail irrelevant to lambda lifting. As can be seen in fig. 1, we therefore adopt a simple lambda calculus with **let** bindings as in Johnsson [2], with a few STG-inspired features:

1. **let** bindings are annotated with the non-top-level free variables they close over
2. Every lambda abstraction is the right-hand side of a **let** binding
3. Arguments and heads in an application expression are all atomic (e.g., variable references)

$$
\begin{array}{lrll}
\text{Variables} & f, x, y & & \\
\text{Expressions} & e ::= & x & \text{Variable} \\
& \mid & f\ x_1 \ldots x_n & \text{Function call} \\
& \mid & \textsf{\textbf{let}}\ b\ \textsf{\textbf{in}}\ e & \text{Recursive } \textsf{\textbf{let}} \\
\text{Bindings} & b ::= & \overline{f_i = [x_{i,1} \ldots x_{i,n_i}]\lambda y_{i,1} \ldots y_{i,m_i} \to e_i} &
\end{array}
$$

Figure 1: An STG-like untyped lambda calculus

2.2 Algorithm

Our implementation extends the original formulation of Johnsson [2] to STG terms, by exploiting and maintaining closure annotations. We will recap our variant of the algorithm in its whole here. It is assumed that all variables have unique names and that there is a sufficient supply of fresh names from which to draw.

We'll define a side-effecting function, lift, recursively over the term structure. This is its signature:

TODO: Why not use plain Haskell? TODO: Why not formulate this as inference rules?

$$\mathsf{lift}_{\_}(\_) \colon \mathsf{Expander} \to \mathsf{Expr} \to \mathcal{W}_{\mathsf{Bind}}\,\mathsf{Expr}$$

As its first argument, lift takes an Expander, which is a partial function from lifted binders to their sets of required variables. These are the additional variables we have to pass at call sites after lifting. The expander is extended every time we decide to lambda lift a binding. It plays a similar role as the E_f set in Johnsson [2]. We write $\mathsf{dom}\,\alpha$ for the domain of the expander α and $\alpha[\mathsf{x} \mapsto S]$ to denote extension of the expander function, so that the result maps x to S and all other identifiers by delegating to α.

The second argument is the expression that is to be lambda lifted. A call to lift results in an expression that no longer contains any bindings that were lifted. The lifted bindings are emitted as a side-effect of the *writer monad*, denoted by $\mathcal{W}_{\mathsf{Bind}}\ \_$.

2.2.1 Side-effects

TODO: Properly define the structure? Or is this 'obvious'? The following syntax, inspired by *idiom brackets* [4] and *bang notation*[1], will allow concise notation while hiding sprawling state threading:

$$[\![E[\langle e_1\rangle, ..., \langle e_n\rangle]]\!]$$

[1] http://docs.idris-lang.org/en/v1.3.0/tutorial/interfaces.html

This denotes a side-effecting computation that, when executed, will perform the side-effecting subcomputations e_i in order (any fixed order will do for us). After that, it will lift the otherwise pure context E over the results of the subcomputations.

In addition, we make use of the monadic bind operators $\ggg\!=$ and $\ggg$, defined in the usual way. The primitive operation note takes as argument a binding group and merges its bindings into the contextual binding group tracked by the writer monad.

2.2.2 Variables

Let's begin with the variable case.

$$\mathsf{lift}_\alpha(\mathsf{x}) = \begin{cases} [\![\mathsf{x}]\!], & \mathsf{x} \notin \mathsf{dom}\,\alpha \\ [\![\mathsf{x}\ \mathsf{y}_1 \ldots \mathsf{y}_n]\!], & \alpha(\mathsf{x}) = \{\mathsf{y}_1, \ldots, \mathsf{y}_n\} \end{cases}$$

We check if the variable was lifted to top-level by looking it up in the supplied expander mapping α and if so, we apply it to its newly required variables. There are no bindings occuring that could be lambda lifted, hence the function performs no actual side-effects.

2.2.3 Applications

Handling function application correctly is a little subtle, because only variables are allowed in argument position. When such an argument variable's binding is lifted to top-level, it turns into a non-atomic application expression, violating the STG invariants. Each such application must be bound to an enclosing **let** binding [2]:

TODO: The application rule is unnecessarily complicated because we support occurrences of lifted binders in argument position. Lifting such binders isn't worthwhile anyway (see section 3). Maybe just say that we don't allow it?

$$\mathsf{lift}_\alpha(\mathsf{f}\ \mathsf{x}_1 \ldots \mathsf{x}_n) = [\![(\mathsf{wrap}_\alpha(\mathsf{x}_n) \circ \ldots \circ \mathsf{wrap}_\alpha(\mathsf{x}_1))(\langle \mathsf{lift}_\alpha(\mathsf{f}) \rangle\ \ \mathsf{x}'_1 \ldots \mathsf{x}'_n)]\!]$$

The notation x' chooses a fresh name for x in a consistent fashion. The application head f is handled by an effectful recursive call to lift. Syntactically heavy **let** wrapping is outsourced into a helper function wrap:

$$\mathsf{wrap}_\alpha(\mathsf{x})(\mathsf{e}) = \begin{cases} \textbf{let}\ \mathsf{x}' = [\mathsf{x}]\lambda \to \mathsf{x}\ \textbf{in}\ \mathsf{e}, & \mathsf{x} \notin \mathsf{dom}\,\alpha \\ \textbf{let}\ \mathsf{x}' = [\,]\lambda \mathsf{y}_1 \ldots \mathsf{y}_n \to \mathsf{x}\ \mathsf{y}_1 \ldots \mathsf{y}_n\ \textbf{in}\ \mathsf{e}, & \alpha(\mathsf{x}) = \{\mathsf{y}_1, \ldots, \mathsf{y}_n\} \end{cases}$$

[2] To keep the specification reasonably simple, we also do so for non-lifted identifiers and assuming that the compiler can do the trivial rewrite $\textbf{let}\ \mathsf{y} = [\mathsf{x}]\lambda \to \mathsf{x}\ \textbf{in}\ E[\mathsf{y}] \Longrightarrow E[\mathsf{x}]$ for us.

2.2.4 Let Bindings

Hardly surprising, the meat of the transformation hides in the handling of **let** bindings. This can be broken down into three separate functions:

$$\mathsf{lift}_\alpha(\textbf{let}\ \mathit{bs}\ \textbf{in}\ e) = (\mathsf{recurse}(e) \circ \mathsf{decide\text{-}lift}_\alpha \circ \mathsf{expand\text{-}closures}_\alpha)(\mathit{bs})$$

1. The first step is to expand closures mentioned in bs with the help of α.
2. Second, a heuristic (that of section 3, for example) decides whether to lift the binding group bs to top-level or not.
3. Depending on that decision, the binding group is noted to be lifted to top-level and syntactic subentities of the **let** binding are traversed with the updated expander.

$$\mathsf{expand\text{-}closures}_\alpha(\overline{f_i = [x_1 \dots x_{n_i}]\lambda z_1 \dots z_{m_i} \to e_i}) = \overline{f_i = [y_1 \dots y_{n_i'}]\lambda z_1 \dots z_{m_i} \to e_i}$$

where

$$\{y_1 \dots y_{n_i'}\} = \bigcup_{j=1}^{n_i} \begin{cases} x_j, & x_j \notin \mathsf{dom}\,\alpha \\ \alpha(x_j), & \text{otherwise} \end{cases}$$

$\mathsf{expand\text{-}closures}$ substitutes all occurences of lifted binders (those that are in $\mathsf{dom}\,\alpha$) in closures of a given binding group by their required set.

$$\mathsf{decide\text{-}lift}_\alpha(\mathit{bs}) = \begin{cases} (\varepsilon, \alpha', \mathsf{lambda\text{-}lift}_{\alpha'}(\mathit{bs})), & \text{if } \mathit{bs} \text{ should be lifted} \\ (\mathit{bs}, \alpha, \varepsilon), & \text{otherwise} \end{cases}$$

where

$$\alpha' = \alpha\left[\overline{f_i \mapsto \mathsf{fvs}(\mathit{bs})}\right] \text{ for } \overline{f_i = [\_]\lambda\_ \to \_} = \mathit{bs}$$

$\mathsf{decide\text{-}lift}$ returns a triple of a binding group that remains with the local **let** binding, an updated expander and a binding group prepared to be lifted to top-level. Depending on whether the argument bs is decided to be lifted or not, either the returned local binding group or the $\mathsf{lambda\text{-}lift}$ed binding group is empty. In case the binding is to be lifted, the expander is updated to map the newly lifted bindings to their required set.

$$\mathsf{fvs}(\overline{f_i = [x_1 \dots x_{n_i}]\lambda y_1 \dots y_{m_i} \to e_i}) = \bigcup_i \{x_1, \dots, x_{n_i}\} \setminus \overline{f_i}$$

The required set consists of the free variables of each binding's RHS, conveniently available in syntax, minus the defined binders themselves. Note that the required set of each binder of the same binding group will be identical.

$$\mathsf{lambda\text{-}lift}_\alpha(\overline{f_i = [x_1 \dots x_{n_i}]\lambda y_1 \dots y_{m_i} \to e_i}) = \overline{f_i = [\,]\lambda \alpha(f_i)\ y_1 \dots y_{m_i} \to e_i}$$

The syntactic lambda lifting is performed in lambda-lift, where closure variables are removed in favor of a number of parameters, one for each element of the respective binding's required set.

$$\mathsf{recurse}(e)(bs, \alpha, lbs) = \mathsf{lift\text{-}bind}_\alpha(lbs) \ggg \mathsf{note} \gg [\![\mathbf{let}\ \langle \mathsf{lift\text{-}bind}_\alpha(bs)\rangle\ \mathbf{in}\ \langle \mathsf{lift}_\alpha(e)\rangle]\!]$$

In the final step of the **let** "pipeline", the algorithm recurses into every subexpression of the **let** binding. The binding group to be lifted is transformed first, after which it is added to the contextual top-level binding group of the writer monad. Finally, the binding group that remains locally bound is traversed, as well as the original **let** body. The result is again wrapped up in a **let** and returned[3].

What remains is the trivial, but noisy definition of the lift-bind traversal:

$$\mathsf{lift\text{-}bind}_\alpha(\overline{f_i = [x_1 \dots x_{n_i}]\lambda y_1 \dots y_{m_i} \to e_i}) = \left[\!\!\left[\overline{f_i = [x_1 \dots x_{n_i}]\lambda y_1 \dots y_{m_i} \to \langle \mathsf{lift}_\alpha(e_i)\rangle}\right]\!\!\right]$$

3 When to lift

Lambda lifting a binding to top-level is always a sound transformation. The challenge is in identifying *when* it is beneficial to do so. This section will discuss operational consequences of lambda lifting, introducing multiple criteria based on a cost model for estimating impact on heap allocations.

We'll take a somewhat loose approach to following the STG invariants in our examples (regarding giving all complex subexpressions a name, in particular), but will point out the details if need be. TODO: I hope this is OK?

3.1 Syntactic consequences

Deciding to lift a binding $\mathbf{let}\ f = [x\ y\ z]\lambda a\ b\ c \to e_1\ \mathbf{in}\ e_2$ to top-level has the following consequences:

(S1) It eliminates the **let** binding.

(S2) It creates a new top-level definition.

(S3) It replaces all occurrences of f in e_2 by an application of the lifted top-level binding to its former free variables, replacing the whole **let** binding by the term $[f \mapsto f_\uparrow\ x\ y\ z]\ e_2$. TODO: Modulo binding non-atomic argument expressions TODO: Maybe less detail here

[3] Similar to the application case, we assume that the compiler performs the obvious rewrite $\mathbf{let}\ \varepsilon\ \mathbf{in}\ e \Longrightarrow e$.

(S4) All non-top-level variables that occurred in the **let** binding's right-hand side become parameter occurrences.

Naming seemingly obvious things this way means we can precisely talk about *why* we are suffering from one of the operational symptoms discussed next.

3.2 Operational consequences

We now ascribe operational symptoms to combinations of syntactic effects. These symptoms justify the derivation of heuristics which will decide when *not* to lift.

Argument occurrences. Consider what happens if f occurred in the **let** body e_2 as an argument in an application, as in $g\ 5\ x\ f$. (S3) demands that the argument occurrence of f is replaced by an application expression. This, however, would yield a syntactically invalid expression, because the STG language only allows trivial arguments in an application.

The transformation from section 2 will immediately wrap the application in a **let** binding for the complex argument expression: $g\ 5\ x\ f \Longrightarrow$ **let** $f = f_{\uparrow}\ x\ y\ z$ **in** $g\ 5\ x\ f$. But this just reintroduces at every call site the very allocation we wanted to eliminate through lambda lifting! Therefore, we can identify a first criterion for non-beneficial lambda lifts:

TODO: Measure that this is actually non-beneficial. The closure growth heuristic will probably catch all bad cases. Also possible code growth.

(C1) Don't lift binders that occur as arguments

Undersaturated calls. When GHC spots an undersaturated call, it arranges allocation of a partial application that closes over the supplied arguments. Pay attention to the call to f in the following example:

TODO: Think about this some mroe and measure. I suspect the allocation heuristic would catch unbeneficial cases. PAPs are let bindings after all TODO: This isn't actually valid STG, as are all the other examples taking a list, etc.

let $f = [x]\lambda y\ z \to x + y + z;$
in $map\ (f\ x)\ [1, 2, 3]$

Here, the undersaturated (e.g., curried) call to f leads to the allocation of a partial application, carrying two pointers, to f and x, respectively. What happens when f is lambda lifted?

$f_{\uparrow} = \lambda x\ y\ z \to x + y + z;$
$map\ (f_{\uparrow}\ x\ x)\ [1, 2, 3]$

The call to $f_{\uparrow}$ will still allocate a partial application, with the only difference that it now also closes over f's free variable x TODO: But no more f!, canceling out the beneficial effects of (S1). Hence

(C2) Don't lift a binding that has undersaturated calls TODO: Measure

Closure growth. (S1) means we don't allocate a closure on the heap for the **let** binding. On the other hand, (S3) might increase or decrease heap allocation. Consider this example:

$$\begin{aligned}&\textbf{let } f = [x\ y]\lambda a\ b \to ...\\ &\quad g = [f\ x]\lambda d \to f\ d\ d + x\\ &\textbf{in } g\ 5\end{aligned}$$

Should f be lifted? It's hard to say without actually seeing the lifted version:

$$\begin{aligned}&f_{\uparrow} = \lambda x\ y\ a\ b \to ...;\\ &\textbf{let } g = [x\ y]\lambda d \to f_{\uparrow}\ x\ y\ d\ d + x\\ &\textbf{in } g\ 5\end{aligned}$$

Just counting the number of variables occurring in closures, the effect of (S1) saved us two slots. At the same time, (S3) removes f from g's closure (no need to close over the top-level $f_{\uparrow}$), while simultaneously enlarging it with f's former free variable y. The new occurrence of x doesn't contribute to closure growth, because it already occurred in g prior to lifting. The net result is a reduction of two slots, so lifting f seems worthwhile. In general:

(C3) Don't lift a binding when doing so would increase closure allocation

Estimation of closure growth is crucial to identifying beneficial lifting opportunities. We discuss this further in section 3.3.

Calling Convention. (S4) means that more arguments have to be passed. Depending on the target architecture, this means more stack accesses and/or higher register pressure. Thus

(C4) Don't lift a binding when the arity of the resulting top-level definition exceeds the number of available hardware registers (e.g., 5 arguments for GHC on x86_64)

Turning known calls into unknown calls. There's another aspect related to (S4), relevant in programs with higher-order functions:

$$\begin{aligned}&\textbf{let } f = [\,]\lambda x \to 2 * x\\ &\quad \mathit{mapF} = [f]\lambda xs \to ...f\ x\ ...\\ &\textbf{in } \mathit{mapF}\ [1,2,3]\end{aligned}$$

Here, there is a *known call* to f in mapF that can be lowered as a direct jump to a static address [3]. Lifting mapF (but not f) yields the following program:

$$\begin{aligned}&\mathit{mapF}_{\uparrow} = \lambda f\ xs \to ...f\ x...;\\ &\textbf{let } f = [\,]\lambda x \to 2 * x\\ &\textbf{in } \mathit{mapF}_{\uparrow}\ f\ [1,2,3]\end{aligned}$$

(C5) Don't lift a binding when doing so would turn known calls into unknown calls

Sharing. Let's finish with a no-brainer: Lambda lifting updatable bindings (e.g., thunks) or constructor bindings is a bad idea, because it destroys sharing, thus possibly duplicating work in each call to the lifted binding.

(C6) Don't lift a binding that is updatable or a constructor application

3.3 Estimating Closure Growth

Of the criterions above, (C3) is the most important for reliable performance gains. It's also the most sophisticated, because it entails estimating closure growth.

3.3.1 Motivation

Let's revisit the example from above:

$$\begin{array}{l} \textbf{let}\ f = [x\ y]\lambda a\ b \to ... \\ \quad g = [f\ x]\lambda d \to f\ d\ d + x \\ \textbf{in}\ g\ 5 \end{array}$$

We concluded that lifting f would be beneficial, saving us allocation of one free variable slot. There are two effects at play here. Not having to allocate the closure of f due to (S1) always leads to a one-time benefit. Simultaneously, each closure occurrence of f would be replaced by its referenced free variables. Removing f leads to a saving of one slot per closure, but the free variables x and y each occupy a closure slots in turn. Of these, only y really contributes to closure growth, because x already occurred in the single remaining closure of g.

This phenomenon is amplified whenever allocation happens under a multi-shot lambda, as the following example demonstrates:

$$\begin{array}{l} \textbf{let}\ f = [x\ y]\lambda a\ b \to ... \\ \quad g = [f\ x]\lambda d \to \\ \qquad \textbf{let}\ h = [f]\lambda e \to f\ e\ e \\ \qquad \textbf{in}\ h\ d \\ \textbf{in}\ g\ 1 + g\ 2 + g\ 3 \end{array}$$

Is it still beneficial to lift f? Following our reasoning, we still save two slots from f's closure, the closure of g doesn't grow and the closure h grows by one. We conclude that lifting f saves us one closure slot. But that's nonsense! Since g is called thrice, the closure for h also gets allocated three times relative to single allocations for the closures of f and g.

In general, h might be occuring inside a recursive function, for which we can't reliably estimate how many times its closure will be allocated. Disallowing to lift any binding which is called inside a closure under such a multi-shot lambda is conservative, but rules out worthwhile cases like this:

$$\begin{array}{l} \textbf{let}\ f = [x\ y]\lambda a\ b \to ... \\ \quad g = [f\ x\ y]\lambda d \to \\ \qquad \textbf{let}\ h_1 = [f]\lambda e \to f\ e\ e \\ \qquad \quad h_2 = [f\ x\ y]\lambda e \to f\ e\ e + x + y \end{array}$$

$\mathbf{in}\ h_1\ d + h_2\ d$

$\mathbf{in}\ g\ 1 + g\ 2 + g\ 3$

Here, the closure of h_1 grows by one, whereas that of h_2 shrinks by one, cancelling each other out. Hence there is no actual closure growth happening under the multi-shot binding g and f is good to lift.

The solution is to denote closure growth in the min-plus algebra $\mathbb{Z}_\infty = \mathbb{Z} \cup \{\infty\}$ and denote positive closure growth under a multi-shot lambda by ∞.

3.3.2 Design

Applied to our simple STG language, we can define a function closure-growth with the following signature:

$$\mathsf{closure\text{-}growth}_{\_\,\_}(\_) \colon \mathcal{P}(\mathsf{Var}) \to \mathcal{P}(\mathsf{Var}) \to \mathsf{Expr} \to \mathbb{Z}_\infty$$

Given two sets of variables for added and removed closure variables, respectively, it maps expressions to the closure growth resulting from

- adding variables from the first set everywhere a variable from the second set is referenced
- and removing all closure variables mentioned in the second set.

In the lifting algorithm from section 2, closure-growth would be consulted as part of the lifting decision to estimate the total effect on allocations like this,

$$\mathsf{closure\text{-}growth}_{\alpha'(f_1)\,\{\overline{f_i}\}}(\mathbf{let}\ \overline{f_i = [x_1 \ldots x_{n_i}]\lambda y_1 \ldots y_{m_i} \to e_i}\ \mathbf{in}\ e) - \sum_i n_i$$

TODO: Written as above, closure-growth will also take the closures we are about to lift into the equation, which is wrong. Think of an elegant workaround

with the *required* set $\alpha'(f_1)$ as the first argument and with $\{\overline{f_i}\}$ for the second set (i.e. the binders for which lifting is to be decided). The expression for which lifting is decided would be the whole **let** expression of the binding in question. The resulting closure growth is entirely due to (S3), so we include the beneficial effect of (S1) into the equation by discounting the sizes of the closures we would no longer allocate. This expression conservatively estimates the effects on heap allocation. In practice, we require that this metric is non-positive to allow the lambda lift.

3.3.3 Implementation

The cases for variables and applications are trivial, because they don't allocate:

$$\mathsf{closure\text{-}growth}_{\varphi^+\varphi^-}(x) = 0$$
$$\mathsf{closure\text{-}growth}_{\varphi^+\varphi^-}(f\ x_1 \ldots x_n) = 0$$

As before, the complexity hides in **let** bindings.

$$\mathsf{closure\text{-}growth}_{\varphi^+\varphi^-}(\mathbf{let}\ \overline{f_i = [x_1 \ldots x_{n_i}]\lambda y_1 \ldots y_{m_i} \to e_i}\ \mathbf{in}\ e) = \mathsf{growth}$$

With the declarations out of the way, here comes the main act:

$$\mathsf{growth} = \mathsf{closure\text{-}growth}_{\varphi^+\varphi^-}(e) + \sum_i \mathsf{local}_i + \sum_i \max 0\, \mathsf{closure\text{-}growth}_{\varphi^+\varphi^-}(e_i) * \mu_i$$

$$\mathsf{local}_i = \begin{cases} |\varphi^+ \setminus \{x_1, \ldots, x_{n_i}\}| - \nu_i, & \text{if } \nu_i > 0 \\ 0, & \text{otherwise} \end{cases}$$

$$\nu_i = \left|\{x_1, \ldots, x_{n_i}\} \cap \varphi^-\right|$$

$$\mu_i = \begin{cases} 1, & f_i \text{ binds a one-shot lambda or thunk} \\ \infty, & \text{otherwise} \end{cases}$$

The local component of growth accounts for allocating each closure of the **let** binding. Whenever a closure mentions one of the variables to be removed (i.e. φ^-, the bindings to be lifted), we count the number of variables that are removed in ν and subtract them from the number of variables from φ^+ (i.e. the required set of the binding group to lift) that didn't occur in the closure before.

The right-hand sides of the **let** binding might or might not be entered, so we cannot rely on a beneficial negative closure growth to occur in all cases. Therefore, we bound closure growth from right-hand sides to be non-negative for a conservative estimate. TODO: We could look at the strictness info to see if it was always called Additionally, we handle multi-shot lambdas by multiplying with ∞.

One final remark regarding analysis performance: closure-growth operates directly on STG expressions. This means the cost function has to traverse whole syntax trees *for every lifting decision.*

We remedy this by first abstracting the syntax tree into a *skeleton*, retaining only the information necessary for our analysis. In particular, this includes allocated closures and their free variables, but also occurrences of multi-shot lambda abstractions. Additionally, there are the usual "glue operators", such as sequence (e.g., the case scrutinee is evaluated whenever one of the case alternatives is), choice (e.g., one of the case alternatives is evaluated *mutually exclusively*) and an identity (i.e. literals don't allocate). This also helps to split the complex **let** case into more manageable chunks.

4 Evaluation

References

[1] Olivier Danvy and Ulrik P. Schultz. "Lambda-lifting in quadratic time". In: *Lecture Notes in Computer Science (including subseries Lecture Notes*

in Artificial Intelligence and Lecture Notes in Bioinformatics). Vol. 2441. 2002, pp. 134–151. ISBN: 3540442332. DOI: 10.1007/3-540-45788-7.

[2] Thomas Johnsson. "Lambda lifting: Transforming programs to recursive equations". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Vol. 201 LNCS. 1985, pp. 190–203. ISBN: 9783540159759. DOI: 10.1007/3-540-15975-4_37.

[3] Simon Marlow and Simon Peyton Jones. "Making a fast curry". In: *Proceedings of the ninth ACM SIGPLAN international conference on Functional programming - ICFP '04*. 2004, p. 4. ISBN: 1581139055. DOI: 10.1145/1016850.1016856. URL: http://portal.acm.org/citation.cfm?doid=1016850.1016856.

[4] CONOR MCBRIDE and ROSS PATERSON. *Applicative programming with effects*. 2008. DOI: 10.1017/S0956796807006326.

[5] Marco T. Morazán and Ulrik P. Schultz. "Optimal lambda lifting in quadratic time". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Vol. 5083 LNCS. 2008, pp. 37–56. ISBN: 3540853723. DOI: 10.1007/978-3-540-85373-2_3.