A fast instruction language for functional programs

ERIK OOSTING, University of Twente, The Netherlands

Within computer science, there is a variety of ways in which algorithms can be expressed. One of these ways is functional programming, in which functions and recursion are used to express programs and control flow. In this paper, an intermediate language is proposed that marries the functional programming style to the imperative nature of native assembly code that is found running in most compiled applications

Additional Key Words and Phrases: Haskell, Compilers, LLVM, Intermediate Languages, ANF

1 INTRODUCTION

When creating a program for a computer, one tends to do that via a programming language. These programming languages make it easier to grasp the logic that the computer program is executing on, but it is not a one-to-one translation of how the computer actually executes this logic. The execution of this logic happens in a local machine code, which is often unreadable at a casual glance. To bridge this gap between developer-readable program code and machine-readable machine code, compilers are used. However, not all compilers are created equal.

Internally, a computer runs on a set of registers, a stack and a heap. These are all increasingly larger pieces of of computer memory that are being used for different purposes. Simple arithmetic can be done using the registers, variables can be fetched on the stack, and large data structures can be put on, and read from the heap. Special attention needs to be paid when a programmer decides to create a function. In that case, arguments that are passed to the function when it is called need to be put on the stack, along with additional information for when the function returns.

As described in the abstract, there are various paradigms in which programmers can write programs. This research will be focusing on the functional programming. The main problem with functional programming is that it uses a lot of functions. For advanced programs, this can be an issue as the stack fills up with many nested function calls if the functional programming language compiler is created naively. Moreover, this research looks into whether the compiler framework LLVM [10] can help optimize these functional programming languages, as if they are imperative (for what it's worth: not functional) languages.

One of the things that distinguishes functional programming languages from LLVM bytecode is the appearance of nested functions. This means that functions can be defined inside of the bodies of other functions. Since LLVM is incapable of doing that. We will have to properly organize our code to make the transformation from nested function to non-nested function happen. One of the most important parts here that can help us is to make all variable names in the program different, while not changing the semantics of the program.

1.1 RQ1

Can we make a intermediate language, designed for functional programming, that enjoys the optimizations of LLVM? [11]

1.2 RQ2

How do we prevent nested functions from appearing in our target program?

1.3 RQ3

How do we correctly substitute variables in our programming language?

2 RELATED WORK

Functional programming languages are usually made without much regards to a von Neumann architecture [2]. This has as consequence that compiling a functional programming language is quite difficult, since you have to translate the program to an entirely new programming paradigm. Boquist & Johnsson [3] point out that this difference can incur a performance penalty to functional languages, due to it having to bridge that gap . A lot of implementations of a function IL have been made before [6, 9, 16], mostly for their own programming languages. Other, smaller languages (such as Idris 2 [4] and unison [8]) have resorted to using Chez Scheme [5] as their back-end language. Chez is a scheme compiler with many optimizations, therefore making it a pretty suitable high-level language to use as a compiler backend. A big codebase to also look towards for inspiration is MLton [6, 11], a version of Standard ML that focuses on whole-program-optimization. It produces very fast programs, and the LLVM backend produced by Leibig [11] can serve as an example on how to compile a high-level language directly to LLVM.

3 METHODOLOGY

The goal is to make an intermediate language that compiles to LLVM. The idea is that the language is high-level enough for it to be easy to make a high-level functional language in it, but low-level enough that it can be agnostic towards all kinds of functional programming languages. The GRIN project [3, 14] is an example of this. However, the implementation of it is seemingly still missing some to the steps described in Boquist and Johnson's original paper.

3.1 Features

Functional programming languages rely a lot on functions. This means that function handling, or at least the closures of functions, need special attention. Normally, every time a function is called, the argument and local variables of that function are put on the call stack. This can blow up the call stack with arguments that are potentially not even needed anymore. Along with that, functional languages often work with the assumption that the values it works

TScIT 40, February 2, 2024, Enschede, The Netherlands

Aexp a	::=	n	:	`number`
	|	x	:	*variable*
	|	" s "	:	`string`
	|	$a + a$	:	*addition*
	|	$a - a$	:	*subtraction*
	|	...		*other simple arithmetic operators*
	|	$\lambda x.c$	:	*lambda expression*

Table 1. BNF grammar for atomic expressions

Cexp c	::=	a	:	*atomic expression*
	|	$f(a, ...)$	:	*function call*
	|	let $x = a$	:	*let binding (atomic expression)*
	|	let $x = f(a, ...)$	:	*let binding (function call)*
	|	if a then c else c	:	*if statement*

Table 2. BNF grammar for complex expressions

with are immutable. For this, the intermediate language should perform closure conversion [1], so that the IL can support nested closures and anonymous lambda functions.

3.2 ANF

The input of a program is done in a language called *Administrative Normal Form* [15]. ANF is useful for directly translating to a intermediate language. The general gist of ANF is that there are no nested function calls. This allows for a clear program flow that tells the consumer of the ANF program when to execute each function along the way. To formalize this, ANF makes a distinction between 2 types of expressions, *atomic* and *complex*. Atomic expressions are expressions that are guaranteed to terminate at all times. As noted in 1, lambda expressions are also considered atomic, since you cannot compute a lambda expression without an argument, so computation is guaranteed to end there. (Do note that table 1 is just an example and not representative of the full grammar

Complex expressions are everything else. This can include let bindings and control flow statements. Note how the grammar in figure table 2 only allows for function calls to be either in a let binding or at the end of a chain of if-statements/let-bindings (tail position).

Languages such as Idris [4] offer an ANF back-end which you can use to let the language compile to a back-end of your choosing. This means that in the end, we could potentially make a new backend for Idris.

While ANF is a good start towards enforcing an order of operations in which functions will be expanded and evaluated, we still allow different variables being given the same name. In order to make our variables unambiguous in which value they point towards, we will need to do variable substitution. There are a variety of ways of doing that, but this paper will explore a novel approach: variable substitution using generalized recursion schemes. For that, we will first need to learn a bit more about recursion schemes themselves.

3.3 Recursion schemes

To get a better, more consistent hold over the recursive nature of the AST, we will use recursion schemes [12]. Recursion schemes are formalisms that help make correct traversals through recursive data structures, where recursive data structures are structures that are (partly) defined in terms of themselves (such as the grammars defined in table 1 and 2). They can be subdivided into three types:

Catamorphisms (folds) are recursion types that take a recursive datatype, and turn it into a final result. These are by far the most common type of recursion, as most traversals of data structures seek to get a single result out of that traversal. To get the result for a term in the data type tree, you need the results of the underlying terms. For this reason, catamorphisms represent bottom-up recursion. The functions we pass to catamorphisms are called algebras, as they take results and perform an operation on them based on a given operator.

Anamorphisms (unfolds) are the opposite of catamorphisms. They generate a recursive data structure from iterating on a sample value. They take in co-algebras as functions. Co-algebras are algebras, but in reverse: they take a value, and based on the type of value they produce a set of different values along with an accompanying operator that binds the values together. As one layer defines more values for lower layers, anamorphisms allow for a top-down recursion.

When we take a value, unfold it into a data structure, and then *refold* it into a into a different value using a separate algebra, that is called a **hylomorphism**. Hylomorphisms take in a sample value, an algebra and a co-algebra to change the supplied value with the help of a recursive datastructure. However, we won't use this combination of algebras and co-algebras as such for we don't really deal with final values all that much. Note also how both anamorphisms and catamorphisms can be represented as hylmorphisms, by replacing the supplied algebra or co-algebra with the identity function $\lambda x.x$, respectively.

Another important recursive action is a change of representation. This means that while the exact value of something may change, it's internal recursive structure will not. A good example of this is mapping a list of values to a list of different values. The resulting list is different from the original one, but the structure of the two lists is identical. For this we can use either a fold or an unfold, whichever works best for the specific change in representation we'd want to accomplish.

3.4 Variable Substitution

At times, a language might want to re-bind a variable to a new value. This means that in the input program there will be multiple values that are bound to the same variable name, and likely multiple instances where this variable is used. To make it unambiguous as to whcih variable refers to which value, we perform variable substitution. At the end of this, each variable assignment contains a unique name. This guarantee means that later transformations don't have to investigate which in assignment a variable is bound.

4 RESULTS

As it stands right now, the project defines algebraic datatypes for ANF. We will first define these datatypes, then use the `recursion-schemes` library to derive some helper datatypes, and finally define a set of functions to help perform variable substitution.

4.1 Haskell Parser

Writing parsers for programming languages can be a tedious exercise, luckily Haskell has a way to automatically generate a parser in the form of `deriving Read`. Then, the whole AST of the program can be expressed as an algebraic datatype, and an AST can be serialized and deserialized in a lisp-style format

```
data AExp
    = LitTrue | LitFalse
    | Ident String | Number Integer | LitStr String
    | AAdd AExp AExp | ASub AExp AExp
    | AMul AExp AExp | ADiv AExp AExp
    | AGt AExp AExp | ALt AExp AExp | AEq AExp AExp
    | ABsl AExp AExp | ABsr AExp AExp
    | AAnd AExp AExp | AOr AExp AExp | AXor AExp AExp
    | Lam [String] CExp
    deriving (Show, Read, Generic)

data Funcall
    = Call String [AExp]
    | Atom AExp
    deriving (Show, Read, Generic)

data CExp
    = Let String Funcall CExp
    | If AExp CExp CExp
    | FC Funcall
    deriving (Show, Read, Generic)
```

An example factorial function can then be described as follows

```
factorial :: AExp
factorial =
    ( Lam
        ["n"]
        ( Let
            "m"
        (Call "factorial" [(ASub (Ident "n") (Number 1))])
            ( If
                (ALt (Ident "n") (Number 2))
                (FC (Atom (Number 1)))
             (FC (Atom (AMul (Ident "m") (Ident "n"))))
            )
        )
    )
```

4.2 Recursion Schemes

The `recursion-schemes`[1] Haskell library does a lot of of heavy lifting when it comes to traversing AST's, as also shown by Tielen [17]. We can generate so-called "base functors" for the AST types with some Template Haskell splices

```
makeBaseFunctor ''CExp
makeBaseFunctor ''AExp
```

A splice for `Funcall` isn't needed, as the base functor of that is structurally identical to itself' This base functor allows for a layer-by-layer analysis of the recursive algorithms that we can use to do some of the more tricky parts of AST analysis. This may also be a good time to explaining which types of recursion schemes we are going to use. As explained in section 3.3, there are 3 important types of recursion schemes we use: folds, unfolds and changes of representation. We'll go into each function we use and describe why we use it

cata: `cata`[2] is a function for straight-forward folds. Whenever this function is used we don't need anything other than the arguments provided in the sub-values.

hoist: `hoist`[3] is a change of representation. The change in representation that we supply must be a pure function (e.g. a mapping from base functor to base functor), and therefore cannot be wrapped in side effects. one important note is because this is a change of representation, we only get access to information available at that exact layer of the recursive data structure. We can't influence underlying layers, even though they are available as arguments. (This is due to a type-level restriction made in the `recursion-schemes` library).

transverse: `transverse`[4] is like `hoist`, except we can put side effects in the resulting representation. Because our recursive structure is now wrapped ina side effect, the result of `transverse` is strictly speaking no longer a recursive structure, so technically (and implementation-wise), `transverse` is a fold.

cotransverse: `cotransverse` is `transverse` in reverse. It takes a recursive structure, wrapped in a side effect, and handles that side effect when changing representation. We can use this in order to decide what kind of side effect we want to envelop the lower layers of our recursive structure in. It is important to note that we are still working with a `hoist`, so we can only affect one recursive layer at a time.

4.3 Variable Substitution

Variable substitution is divided into 2 parts: replacing variables and identifying variables that need to be replaced

For part 1, all variables that are identified will be replaced with the same variable, appended with the character "_"

For atomic expressions, the process is relatively simple. Replace identifiers if need be, and arguments as well. There is no need to look into the deeper structure, so we can use a `hoist` to change the AST

```
-- | replace bound variables in AExps
replaceVarsAExp :: String -> AExpF a -> AExpF a
```

[1] https://hackage.haskell.org/package/recursion-schemes

[2] https://hackage.haskell.org/package/recursion-schemes-5.2.2.5/docs/Data-Functor-Foldable.html#v:cata

[3] https://hackage.haskell.org/package/recursion-schemes-5.2.2.5/docs/Data-Functor-Foldable.html#v:hoist

[4] https://hackage.haskell.org/package/recursion-schemes-5.2.2.5/docs/Data-Functor-Foldable.html#v:transverse

```
replaceVarsAExp n (IdentF m) = IdentF $ compareNames m n
replaceVarsAExp n (LamF args body) =
    LamF
       (fmap (\x -> if x == n then x ++ "_" else x) args)
        (cata replaceVarsCExp body n)
replaceVarsAExp _ rest = rest
```

For complex expressions, we use an simple algebra, but also use the fact that `String ->` forms a monad that provides the given string argument when trying to get a result from the `id` function

```
replaceVarsCExp :: CExpF (String -> CExp) -> String -
> CExp
replaceVarsCExp (LetF name fc restf) = do
    env <- id
    rest <- restf
    let newName = compareNames name env
    return $ Let newName (replaceVarsFC env fc) rest
replaceVarsCExp (IfF cond thenF elseF) = do
    thenPart <- thenF
    elsePart <- elseF
    env <- id
    return $
        If
            (hoist (replaceVarsAExp env) cond)
            thenPart
            elsePart
replaceVarsCExp (FCF fc) = do
    env <- id
    return $ FC (replaceVarsFC env fc)
```

Finally, the `Funcall` datatype can just be a normal function, as it is not primitively recursive

```
-- | replace bound variables in Function calls
replaceVarsFC :: String -> Funcall -> Funcall
replaceVarsFC n (Atom aexp) =
    Atom $ hoist (replaceVarsAExp n) aexp
replaceVarsFC n (Call name args) =
    Call
        (compareNames name n)
        $ fmap (hoist (replaceVarsAExp n)) args
```

Part 2 is the actual checking which variables get substituted, this is done with `cotransverse`[5] algebras. As said in section 3.3, cotransverse algebras are taking a layer of recursion, wrapped in a side effect, and turning that into a pure layer with side-effected constituent parts. In our case, the side effect is a constantly updating list of variable names we can call the "environment" (env for short).

When we substitute variables, we try to find places where values are being bound to values. This can be either in the arguments of functions, or in let-statements. This means that in case of atomic expressions, we'll have to examine the lambda expressions, and in the case that an argument is already in our environment (and therefore, bound to a different value), we'll append a _ to the variable name, and then change the variable occurrences in the function body accordingly with the `replaceVarsCExp` function.

```
subVarsAExp :: ([String], AExpF a) -> AExpF ([String], a)
subVarsAExp (env, LamF args body) =
    let
        toReplace = intersect env args
        newArgs =
            fmap
           (\x -> if x `elem` toReplace then x ++ "_" else x)
                args
     newBody = foldl (cata replaceVarsCExp) body toReplace
     in
        LamF
            newArgs
            ( cotransverse
                subVarsCExp
                (newArgs ++ env, [], newBody)
            )
subVarsAExp (env, rest) = fmap (env,) rest
```

For complex expressions, the identification of variable bindings is similarly easy. However, transformation of bound variables is a bit more tricky. As the type of `cotransverse ((Corecursive s, Recursive t, Functor f) => (forall x. f (Base s x)->Base t (f x)) f s -> t)` doesn't allow us to access the lower levels of a complex expressions. While this limitation wasn't an issue for atomic expressions, as `Lam` is a leaf node in atomic expression trees, it is a problem here, as we may need to edit variables occuring lower in the `CExp` expression tree. To solve this, we add a new variable to our side-effects called queue. This is a list of variables that were found to be previously bound in the current complex expression tree, but haven't changed their variable names in later occurences deeper in the tree. At each later of the `CExp`, check if a variable in an assignment exists in the queue, and re-assign it accordingly if so.

```
{- | cotransverse of a complex expression.
   | The first string list represents
   | bound variables, the second one a queue of
   | variables to be replaced
-}
subVarsCExp ::
    ([String], [String], CExpF a) ->
    CExpF ([String], [String], a)
subVarsCExp (env, queue, LetF name fc rest) =
    let
        (newName, newFC) =
            foldr
        -- repeatedly replace variables in the function call
                ( \m (n, f) ->
                    if n == m
                     then (n ++ "_", replaceVarsFC n f)
                        else (n, f)
                )
```

[5] https://hackage.haskell.org/package/recursion-schemes-5.2.2.5/docs/Data-Functor-Foldable.html#v:cotransverse

```
                (name, fc)
                queue -- make sure to do oldest first!
    in
       LetF
           newName
           newFC
           ( if name `elem` env
               then (env, newName : queue, rest)
               else (name : env, name : queue, rest)
           )
subVarsCExp (env, queue, IfF cond thenPart elsePart) =
    IfF
     (foldr (\x c -> hoist (replaceVarsAExp x) c) cond queue)
       (env, queue, thenPart)
       (env, queue, elsePart)
subVarsCExp (env, queue, FCF fc) =
    FCF $
        foldr replaceVarsFC fc queue
```

5 FUTURE WORK

The project as it stands is far from done. Originally the plan was to write an LLVM frontend, that would output LLVM bytecode that could then be used by `lld`. In order to get to this point, there are a few things left to do. In summary, first the in-line lambdas need to be put into the top scope. Then, the in-haskell IR could be converted to LLVM bytecode. Furthermore, the languages as it s right now doesn't have I/O capabilities, making it not very useful.

5.1 Defunctionalization

In LLVM, one cannot define a function inside of another function. Therefore, in order to translate all the nested functions to LLVM, we need to lift them to the global scope. This can be done using a technique called "lambda lifting" [7]. This can be a three-step process:

(1) Identify all free variables in a nested function
(2) Move these free variables to the function parameters
(3) Identify all call-sites of the nested functions and expand the arguments with the arguments that were added

We don't have to care about variables overlapping here, since we already have used variable substitution to make all variables be assigned only one time.

5.2 LLVM translation

The original goal of this project was to make a full LLVM front-end. To this end, the plan was to use the `llvm-codegen`[6] Haskell library. This library provides a typed haskell interface for LLVM bytecode so that possible errors in code generation can be caught earlier than if codegen was done in a language like python [7]. At this point the efficacy of LLVM could also be analyzed to see how well LLVM optimizes the programs by itself vs. manual optimizations we may have to do ourselves.

[6]https://github.com/luc-tielen/llvm-codegen
[7]https://llvmlite.readthedocs.io/en/latest/

5.3 Optimizations

Because of the lack of an LLVM implementation, we could also not identify if the implementation actually optimizes our language. More research has to go into how well this language optimizes functional programs in general.

5.4 I/O

As said before, a language that doesn't have an effect on it's surroundings is limited to running a single computation per program. In theory, an ultra-optimized compiler could take a language that doesn't have side effects, and compile the final answer that these programs would compute immediately, thus basically creating an interpreter. In order to have the created programs be interactive, the intermediate language could be amended to support I/O

5.5 Example language

This paper discusses an intermediate language, and that means that the language is not meant to be used by programmers by hand. It is instead meant for there to be a front-end language for this intermediate language. Such a language could for example be a lisp. The most important step that should be considered here is that the current intermediate language only supports strings, numbers and functions. However, any custom data structures (such as lists) can be converted to functions using a Mogensen-Scott encoding [13]. This encoding uses the fact that algebraic datatypes are in actuality a set of functions that get you to a data representation, and that this data representation is an intermediate step towards getting a final result. This means that algebraic datatypes can be represented as functions that can be filled in later when the algebraic datatype gets consumed to produce a useful value.

6 DISCUSSION

A lot of design and implementation decisions were made in the process of developing this language. We will go through some of them in a similar order as we did in the rest of the paper

6.1 ANF

ANF has some important design limitations. Most importantly, because function calls only allow atomic variables in their arguments, the language is forced to be eagerly evaluated.

6.2 Recursion schemes & variable substitution

Because of the aforementioned limitations of hoisting, the current implementation of variable substitution on complex expressions 4.3 is sketchy to say the least. Due to time constraints this implementation has not be properly tested for issues in variable substitution, but if there are issues to be found somewhere, it's in the substitution. This could potentially be solved by changing the type of `cotransverse` to a more lenient version: `(Corecursive t, Recursive s, Functor f) => (f (Base s s) -> Base t (f s)) -> f s -> t`. This gives the representation-changing function information about what the underlying building blocks of each layer are.

On a more minor note, the use of do-notation in `replaceVarsAExp` may be unnecessary, as we don't really use the advantage of functions being monads in haskell to it's fullest extent in the function

definition. Also since the value of the argument doesn't change at any time, we could've also used the type `String -> AExpF (AExp) -> AExp`, and forgo function usage in the definition altogether.

7 CONCLUSION

In this paper we have defined a new intermediate language, designed to ease the implementation of functional programming languages through the use of administrative normal form.

We have also shown an alternative way of doing variable substitution using recursion schemes. This implementation has been informally justified to be correct.

Furthermore, we have discussed possible future avenues this project can take, such as implementing the new IR in LLVM and adding I/O. This language has a lot of potential in it, which could be explored in later research

8 ACKNOWLEDGEMENTS

I would like to thank Peter Lammich for supervising this project, and also the many people who helped me make a planning for it.

REFERENCES

[1] Andrew W. Appel. 1992. *Compiling with Continuations.* Cambridge University Press.

[2] John Backus. 1978. Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs. *Commun. ACM* 21, 8 (aug 1978), 613–641. https://doi.org/10.1145/359576.359579

[3] Urban Boquist and Thomas Johnsson. 1996. The GRIN Project: A Highly Optimising Back End for Lazy Functional Languages. In *Implementation of Functional Languages, 8th International Workshop, IFL'96, Bad Godesberg, Germany, September 16-18, 1996, Selected Papers (Lecture Notes in Computer Science, Vol. 1268)*, Werner E. Kluge (Ed.). Springer, 58–84. https://doi.org/10.1007/3-540-63237-9_19

[4] Edwin C. Brady. 2021. Idris 2: Quantitative Type Theory in Practice. 194 (2021), 9:1–9:26. https://doi.org/10.4230/LIPICS.ECOOP.2021.9

[5] R Kent Dybvig. 2006. The development of chez scheme. *ACM SIGPLAN Notices* 41, 9 (2006), 1–12.

[6] Kavon Farvardin and John Reppy. 2018. Compiling with Continuations and LLVM. *arXiv preprint arXiv:1805.08842* (2018).

[7] Sebastian Graf and Simon Peyton Jones. 2019. Selective Lambda Lifting. *CoRR* abs/1910.11717 (2019). arXiv:1910.11717 http://arxiv.org/abs/1910.11717

[8] Simon Højberg, Paul Chiusano, Stew O'Conner, John Ericson, Mitchell Rosen, Travis Staton, Vladislav Zavialov, and Noah Haasis. 2023. unison. https://github.com/unisonweb/unison.

[9] Simon L. Peyton Jones. 1992. Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine. *Journal of Functional Programming* 2, 2 (1992), 127–202. https://doi.org/10.1017/S0956796800000319

[10] Chris Lattner and Vikram S. Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. (2004), 75–88. https://doi.org/10.1109/CGO.2004.1281665

[11] Brian Andrew Leibig. 2013. An LLVM Back-end for MLton. *Department of Computer Science, B. Thomas Golisano College of Computing and Information Sciences, Tech. Rep.* https://api.semanticscholar.org/CorpusID:62345781

[12] Erik Meijer, Maarten M. Fokkinga, and Ross Paterson. 1991. Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire. In *Functional Programming Languages and Computer Architecture, 5th ACM Conference, Cambridge, MA, USA, August 26-30, 1991, Proceedings (Lecture Notes in Computer Science, Vol. 523)*, John Hughes (Ed.). Springer, 124–144. https://doi.org/10.1007/3540543961_7

[13] Torben Æ. Mogensen. 1992. Efficient Self-Interpretations in lambda Calculus. *J. Funct. Program.* 2, 3 (1992), 345–363. https://doi.org/10.1017/S0956796800000423

[14] Peter Podlovics, Csaba Hruska, and Andor Pénzes. 2021. A Modern Look at GRIN, an Optimizing Functional Language Back End. *Acta Cybern.* 25, 4 (2021), 847–876. https://doi.org/10.14232/ACTACYB.282969

[15] Amr Sabry and Matthias Felleisen. 1993. Reasoning about Programs in Continuation-Passing Style. *LISP Symb. Comput.* 6, 3-4 (1993), 289–360.

[16] Camil Staps, John van Groningen, and Rinus Plasmeijer. 2021. Lazy Interworking of Compiled and Interpreted Code for Sandboxing and Distributed Systems. In *Proceedings of the 31st Symposium on Implementation and Application of Functional Languages* (Singapore, Singapore) *(IFL '19)*. Association for Computing Machinery, New York, NY, USA, Article 9, 12 pages. https://doi.org/10.1145/3412932.3412941

[17] Luc Tielen. 2022. How to lower an ir? https://luctielen.com/posts/how-to-lower-an-ir/