Structured Combinators for Efficient Graph Reduction

Cecil Accetti, Rendong Ying, and Peilin Liu

Abstract—Combinators have a long history in mathematics, logic and computer science, as simple primitive symbols with which complex relationships can be described. In practice, this simplicity comes with a cost, impacting the performance of combinator-based languages and computers. We propose a generalized representation for combinators, not as primitives, but as self-contained definitions, where the structure of their graph-reduction semantics is explicit. For a sample of 798 unique lambda-terms from common benchmark programs, we show that *structured* combinators can improve the quality of compiled code, generating smaller and more efficient graphs, while translating seamlessly to a machine-friendly encoding, as part of the open-source *fun* instruction-set architecture.

Index Terms—applicative (functional) programming, instruction set design, compilers

1 Introduction

When looking for ways to minimize the number of fundamental notions of logic and mathematics, Schönfinkel discovered a calculus of primitive higher-order functions, or combinators [1]. With this calculus, he was able to simplify computable functions by abstracting their variables, replacing them for combinators. Applied to practical computing [2], [3], [4], a language of combinators makes for an interesting purely-functional instruction-set architecture (ISA), devoid of variable-handling and control-flow instructions such as *push, load, store, move, branches* and *jumps*. Furthermore, as a formal model, it also inherits beneficial meta-programming properties of the λ-calculus [5]. At a time of growing concerns on computer safety and security [6], it is worth considering these and other offerings of combinatory logic and of functional programming as a whole, as a more solid foundation for computing – or at least as an option for the applications in need [7], [8], [9].

The problem of combinators, however, lies in their implementation. If we employ the same *minimal* combinator sets from the theory (Table 1) on a graph-reduction machine, as in [2], [3], [4], [8], [9], the results are programs of "disastrous size" [10], "often much larger" [11] than the original λ-forms. Multiple solutions to this problem have been proposed, ranging from extended combinator sets and term-rewrite rules, to alternative combinatory systems and abstraction algorithms. Still, most of these works have preserved the idea of minimal sets of primitive combinators [8], [9], [10], [12], [13], [14], [15].

Contributions. In this letter we take a fresh look at combinatory growth from a hardware design perspective, by defining a combinatorial representation that reflects the structure of program-graph transformations. We encode these *structured* combinators as instructions of the open *fun* instruction-set architecture, alongside an optimizing compiler flow (Section 2). We investigate the ISA-dependent performance of graph reduction – the size and shape of program-graphs, and the number of reduction steps and allocated graph nodes during evaluation [16] – using an FPGA-based *fun* graph-reducer CPU (Section 3). For a sample of 21 programs from [17], [18] and 10 common higher-order functions, we show that *structured* combinators enable consistent improvements over primitive SK-combinators, improving the quality of compiled code with smaller and more efficient graphs.

- *All authors are with the School of Electronic Information and Electrical Engineering, Shanghai Jiao Tong University, Shanghai, P.R.C. E-mail: {cecaccetti, rdying}@sjtu.edu.cn*

Manuscript received April 29, 2022; revised July 19, 2022.

2 A New Combinatorial Representation

We start our discussion with an analogy to imperative programming. The primitive notion of any imperative ISA is the storage cell. Instructions can operate on storage cells, copying or overwriting their contents until a desired state is met, but are just derivations from the idea that values are bound to specific memory or register locations. Likewise, for a purely-functional graph reducer [8], [9], we can expect the primitive notion to be the graph node and not a combinator (the instruction). Nodes make up the graph, while combinators merely modify the structure of the graph according to some fixed rule. Despite this, previous works have prioritized the language side, first defining a combinator set and abstraction algorithm to then derive a tailored abstract (physical) machine. Here, we take the opposite path and derive a language of combinators starting from the graph transformations they represent at machine level.

2.1 Structured Combinators

To a graph reducer, a combinator is a recipe for a pre-defined graph transformation. Our approach is to break this recipe down to three structural elements, namely (1) the arity, (2) the reduction pattern, and (3) the contents of the new graph nodes.

Definition 1. *A structured combinator* $\boldsymbol{C}$ *is a 3-tuple*

$$\boldsymbol{C} = (t,a,[idx]) = C^{t}_{a[idx]}$$

where a *is the arity of the combinator,* t *is the structural pattern of its reduction;* [idx] *is a list of de Bruijn indices of the arguments, as they appear in order leftmost-outermost traversal of the reduction pattern.*

The arity of a combinator is the number of arguments it consumes from the program graph during reduction, and is easily inferred from its λ-form (Table 1). A reduction pattern is an empty sub-graph, the result of lifting the structure of a function body from its actual content, e.g. from λabc.+(+ab)c to x(xxx)x, where x denote an empty graph node. While the content of these sub-graphs change from program to program, their structures (*patterns*, on Table 1) are often recurrent, especially on higher-order functions [19].

Structured combinators (SC) can be seen as a bounded-form of program-defined super-combinators [20], limited in *grain* [11] by the reduction patterns implemented in a target machine. SCs are equivalent to the untyped λ-calculus with de Bruijn indices, and can be used to represent SK-combinators, categorical combinators [21] and other equivalent combinatory systems (Table 1). In operational terms, SCs are not different

TABLE 1
Some SK- and Categorical Combinators in structured form [2], [21]

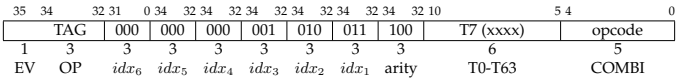
SK	λ-form	Pattern	Structured Form
S	λabc. ac(bc)	xx(xx)	$C^{xx(xx)}_{3[0,2,1,2]}$
K	λab. a	x	$C^{x}_{2[0]}$
I	λa. a	x	$C^{x}_{1[0]}$
B	λabc. a(bc)	x(xx)	$C^{x(xx)}_{3[0,1,2]}$
C	λabc. acb	xxx	$C^{xxx}_{3[0,2,1]}$
S′	λabcd. a(bd)(cd)	x(xx)(xx)	$C^{x(xx)(xx)}_{4[0,1,3,2,3]}$
C′	λabcd. a(bd)c	x(xx)x	$C^{x(xx)x}_{4[0,1,3,2]}$
B′	λabcd. ab(cd)	xx(xx)	$C^{xx(xx)}_{4[0,1,2,3]}$
CCC			
pair	λabf. f a b	xxx	$C^{xxx}_{3[2,0,1]}$
∘	λfga. f(ga)	x(xx)	$C^{x(xx)}_{3[0,1,2]}$
△	λfga. (fa,ga)	x(xx)(xx)	$C^{x(xx)(xx)}_{4[3,0,2,1,2]}$
curry	λfab. f (a,b)	x(xxx)	$C^{x(xxx)}_{4[1,0,2,3]}(pair)$
uncurry	λfp. p f	xx	$C^{xx}_{2[1,0]}$

from primitive combinators, and pose no limitations regarding evaluation order, be it strict, normal or lazy.

2.2 Encoding

The elements of a SC map to RISC-like instruction fields in a straightforward manner, as shown below.

$$\lambda xyzt.\ t\ z\ y\ x \Rightarrow \lambda^4.3210 \Rightarrow C^{xxxx}_{4[3,2,1,0]}$$



Starting from a typical a 32-bit word, we allocate 6 bits for a *pattern identifier* field, allowing the support of 64 reduction patterns with up to 6 nodes, from x (Type 0) to xxxxxx (Type 63). The *arity* field sets a maximum of 8 arguments (3 bits), and 6 3-bit *index* fields encode the list of deBruijn indices of a given reduction rule. As instructions for *fun*, a 5-bit *opcode* field completes the 32-bit word, with an additional 3-bit *tag* for identification of the type of the graph node: operation (OP), literal (lit) or indirection node (link). The EV bit marks the leftmost-outermost node of a subgraph.

2.3 Compilation

The translation between functions in (de Bruijn) λ-form and SCs is done in four phases: atom lifting, pattern matching, abstraction and unification (Fig.1 and Fig.3A).

Atom lifting (`liftAtoms`) removes atoms (links to other functions, constants, and built-in operations such as +, -, etc.) from already λ-lifted functions, resulting in a pair consisting in a higher-order super-combinator and a list of atoms.

During the pattern matching phase (`match`), the structure of the lifted super-combinator is compared to the reduction patterns supported by the target ISA (64 patterns for 36-bit *fun* cells). If pattern-matching succeeds, the newly-defined structured combinator is integrated in the original expression, alongside the lifted constants (`apply`). If pattern-matching fails, an abstraction algorithm is employed, followed by an additional step of *unification*. In this event, the compiler decomposes the super-combinator into standard SK-combinators (or CCCs, SF-combinators, or any other equivalent system) via variable

```
data Exp = Ap Exp Exp          -- curried application
         | Lam Int Exp  -- Lambda in de Bruijn notation
         | C Arity Type [Int] -- structured combinator
         | Idx Int            -- de Bruijn index
         | Const Cts       --constants (Int,+,-,*...)
--split expression into supercombinator and [atoms]
--liftAtoms (\1.+00) -> (\2.011, [+])
liftAtoms :: Int -> Exp -> (Exp, [Exp])

--Match Exp body to a list of supported patterns
--returning either a structured combinator, or
--Nothing if pattern match fails
match :: Exp -> Maybe Exp

-- Re-apply lifted atoms to supercombinator
-->>apply e [+,1,2] -> Ap (Ap (Ap e +)1)2)
apply :: Exp -> [Exp] -> Exp

--Abstraction:bracket[2],semantic[15],categorical[21]...
abstract :: Int ->  Exp -> Exp

--Recursively merge combinators
unify :: Exp -> Exp

-- Convert lambda-form to structured combinators
lam2combi :: Exp -> Exp
lam2combi (Lam n e)= case(match super)of
                Just x -> apply x constants
                Nothing -> unify(abstract n e)
              where (super,constants) = liftAtoms n e
lam2combi (Ap e1 e2)= Ap(lam2combi e1)(lam2combi e2)
lam2combi e = e
```

Fig. 1. A simple compiler from a λ-form to structured combinators.

abstraction, and then merge these components into a maximal structured form supported by the ISA (`unify`). For brevity, the following example illustrates the unification of the term BKK into a single combinator $C^{x}_{3[0]}$ in structured form. The terms in the right side of the | are arbitrary arguments in applicative order (de Bruijn indices).

B K K ⇒ $C^{x(xx)}_{3[0,1,2]}\ C^{x}_{2[0]}\ C^{x}_{2[0]}$	–SK to structured
$C^{x(xx)}_{3[0,1,2]}\ C^{x}_{2[0]}\ C^{x}_{2[0]}$ | 0 1 2 3...n	–Apply n arguments
$C^{x}_{2[0]}\ (C^{x}_{2[0]}\ 0)$ | 1 2 3...n	–1st reduction $C^{x(xx)}_{3[0,1,2]}$
$C^{x}_{2[0]}\ 0$ | 2 3 4...n	–2nd reduction $C^{x}_{2[0]}$
(0) | 3 4...n	–3rd reduction $C^{x}_{2[0]}$
⇒ $(C^{x}_{3[0]})$	–Match: a=3,t=x,[idx]=[0]

Pattern matching and unification effectively define new combinators without the use of pre-defined term-rewrite rules. Their effects can be seen on Fig.2, which compares the compiled graphs for a list-catamorphism (foldr) [19], expressed in traditional symbolic (Fig.2a) and structured forms (Fig.2b). In this example, a term $(C^{xx(xxx)}_{4[3,2,0,1,2]}(C^{xx(xxxx)}_{5[1,3,0,1,2,4]}x))$ replaces a much larger term (B′(S′C)(CI)(S′(B′C)(Bx))).

2.4 Artifacts

The specification for the *fun* ISA, programming examples, complete source-code for the compiler from Fig.1 and benchmarking data can be found at http://wiki.fun-arch.org.

3 Evaluation

3.1 Methodology

We apply a selection of 31 benchmark programs to the compiler flow of Fig.3. These programs range from simple higher-order functions of the Haskell standard Prelude library (foldr, map, zip, etc.) to programs from the *imaginary*,*spectral* and *real* subsets of the nofib suite and other sources [17], [18], [22], and reflect typical programming patterns that handle tuples, lists, trees, grammars and other algebraic data structures.

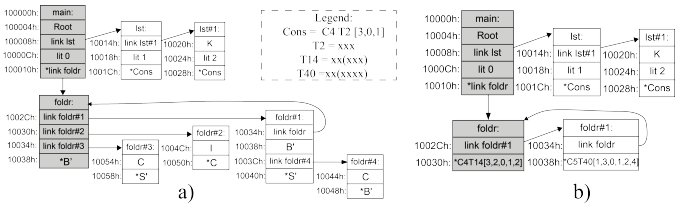


Fig. 2. Combinator graphs for `foldr + 0 [1,2]` in primitive (a) and structured (b) forms. The graph spine is highlighted in gray;

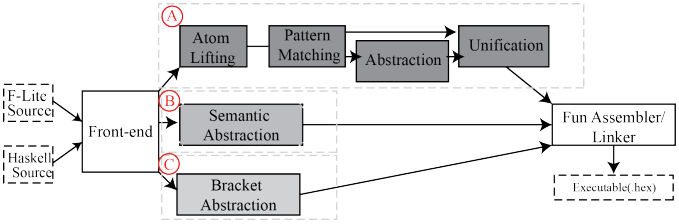


Fig. 3. Experimental compiler flow. Source code in a subset of Haskell or F-lite [18] is translated to *fun* via three user-selectable back-ends: (A) structured, (B) semantic [15] and (C) bracket abstraction [12]

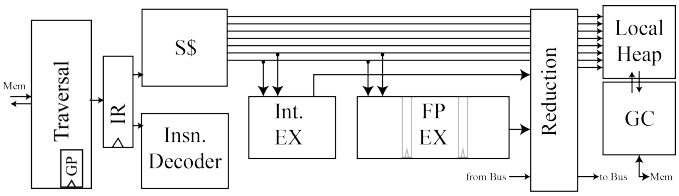


Fig. 4. The Blackbird-II microarchitecture: Traversal unit with a Graph-Pointer (GP) register; Instruction Register (IR); A graph spine cache (S$) stores the 32 most-recently accessed nodes of the graph spine; integer and floating-point execution datapaths (EX); reduction unit and two-space copying garbage collector (GC)

After desugaring and λ-lifting at the compiler front-end [20], each program is converted to a λ-form with de Bruijn indices before translation into structured combinators. Our experimental compiler embeds three user-selectable back-ends (Fig.3). Back-end C implements the bracket abstraction algorithm from [2], using the set of SK-combinators from Table 1, while back-end B implements the semantic abstraction algorithm of [15], limited to combinators SKIBC as described in the original paper. Back-end A fully explores the structured combinator notation, via pattern matching and unification.

Runtime measurements were performed via a cycle-accurate emulator for the Blackbird-II microarchitecture (Fig.4), double-checked against the RTL model of the CPU, written in VHDL and synthesized to an Altera/Intel Cyclone IV FPGA. Blackbird-II is a variant of the Blackbird CPU of [8], with its datapath adapted to structured combinators. It is a 3-stage pipeline implementation that issues one graph node (instruction or data) per clock cycle during traversal (search steps) and takes one clock cycle to execute a combinator reduction (reduction steps), whenever a reducible combinator is found on the graph.

3.2 Static Analysis: Program Size

We first compare the sizes of compiled program-graphs against the sizes of the original source program, in λ-form. The size of a graph is measured in terms of the number of nodes, in accordance with prior work by Hartel [16]. Fig.5a shows that for this set of benchmarks, graphs generated by back-end A are strictly smaller than the source graphs (dotted line). The same does not occur with graphs generated by back-ends B and C, which can be 2.10x larger than the source programs (back-end B, While). Another aspect to consider from the data at Fig.5a is the loose correlation between source and compiled sizes for back-ends B and C. While the sizes of source graphs are presented in ascending order, the outputs of back-ends B and C are less predictable, with longer programs (e.g. Sudoku) generating smaller graphs than those generated by shorter programs (e.g. Sorting), due to inefficient abstraction.

Fig.5b provides a second metric for the quality of static object code generated by back-ends A,B and C: the ratio between graph depth and graph size (depth / size). This value quantifies how "well-behaved" is a program-graph, in terms of its shape. A well-behaved expression [12] is of the form KE_1E_2, where K is a term composed only of combinators. In an ideal translation, K is of the form $K_1K_2K_3...K_n$, with K_n a subterm composed of a single combinator. The higher the depth/size ratio, the easier it is to optimize, during compile-time, and to traverse, during runtime, as all Ks are allocated on the main spine of the graph. As illustrated on Fig.2, less efficient translations result in graphs that spread far from the main spine.

A summary of the static code evaluation is shown on Table 2. From a total of 798 unique lifted functions in λ-form, we find that 60.5% of these functions fit one of the 64 patterns supported by our ISA encoding, and are pattern-matched without additional abstraction and unification steps. The worst conversions are shown relative to the size of the source function.

3.3 Dynamic Analysis

We measure three aspects of program evaluation: reduction steps, search steps and node allocations. Reduction and search steps have direct influence on execution time, as consequences of graph size. Node allocation impacts both time and memory requirements, due to potential pauses for garbage collection.

Fig.6 shows the relative runtime improvement of back-end A (structured) over back-end C (bracket abstraction) in terms of reduction steps, search steps and node allocations. Although single-number statistics for nofib should be taken with a grain of salt [17], we consider the average for illustrative purposes. For this set of programs, those compiled through back-end A require 3.78x less reduction steps, while allocating 2.32x less graph nodes, in average, than those from back-end C. Fig.6 also shows the relative performance for a selection of common higher-order functions, operating on lists of 100 integer values, with a maximum gain of 7.95x for zip3.

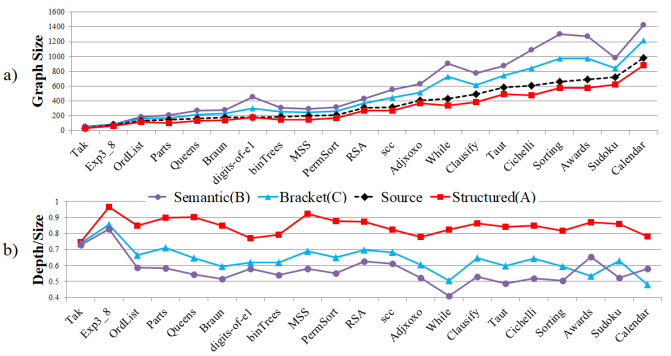


Fig. 5. a) Size of programs in λ-form and after compilation by back-ends A, B and C. b) Depth:size ratio for back-ends A, B and C

TABLE 2
Statistics for static-code evaluation

	#Function	Max.Var.	Avg.Var.	Matched	
	798	8	2.31	60.5%	
Sizes	**Source**	**A-Semantic**	**A-Bracket**	**B**	**C**
Largest	79	78	78	133	78
Mean	6.72	5.85	5.82	11.6	9.11
Median	5	3	3	7	6
σ	6.96	7.67	7.54	13.93	9.49
Worst	-	36/21	35/21	21/3	16/3

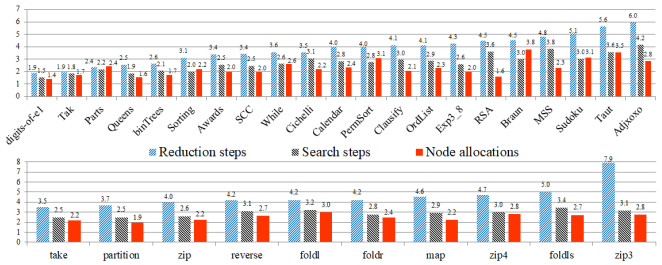


Fig. 6. Relative improvements on reduction, search and node count for back-end A over back-end C.

4 Limitations and Future Directions

On a graph reducer CPU, evaluation time is a result of the combination of the time spent on search, reduction and garbage collection. From these three metrics, only reduction count is immune to microarchitecture-level design decisions, and better represent ISA-dependent performance, the focus of our discussion. Specific mechanisms for instruction fetch, lazy evaluation and memory management [3], [9] can have considerable impacts on overall time, but are beyond the scope of this letter.

Execution time is also subject to programming styles and methods. Fig.7 shows the impact on reduction count of porting some programs to a style of explicit morphisms [19], [23], in the order of 155x (!) for exp3_8 and 33.3x for quickSort, when compared to a recursive SK implementation. While this is an example of how structured combinators can efficiently represent higher-order functions such as *catamorphisms* and *hylomorphisms*, further work is needed to define an adequate benchmark suite in this style and investigate its potential benefits. Since it is difficult to compare our work with experimental results from 30 [16] or 40 years ago [3], [4], we hope our measurements could set a new reference point for further research on combinators and architectural support for functional programming.

5 Conclusions

Combinators are still interesting, a hundred and one years after their discovery. In this letter we reformulated their structure as a way to improve the performance and memory requirements of a combinator-based ISA. For a sample of 798 unique λ-terms obtained from common functional benchmarks we have shown that structured combinators enable a consistent performance improvement over traditional symbolic representations. In the context of a renewed interest in hardware-level support for functional programming to improve the safety and security of computer systems, we encode structured combinators as part of *fun*: a purely-functional instruction set. With an open-source architecture like *fun*, the door is now open for further improvements at microarchitecture and compiler levels.

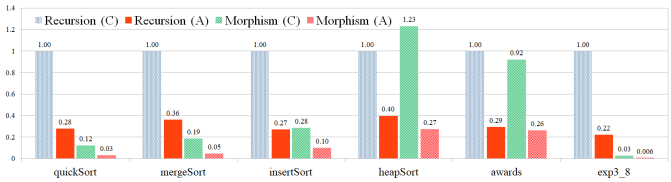


Fig. 7. Performance comparison of programs written as explicit recursive functions and as morphisms, in terms of reduction steps (normalized)

Acknowledgments

The authors would like to thank the anonymous reviewers, whose suggestions greatly improved our discussion.

References

[1] M. Schönfinkel, "Uber der bausteine der mathematischen logic," *Math. Annalen*, vol. 92, no. 3, 1924.
[2] D. A. Turner, "A new implementation technique for applicative languages," *Software: Practice and Experience*, vol. 9, no. 1, 1979.
[3] W. R. Stoye, T. J. W. Clarke, and A. C. Norman, "Some practical methods for rapid combinator reduction," in *Proc. 1984 ACM Symp.on LISP and Functional Programming*, 1984.
[4] M. Scheevel, "Norma: A graph reduction processor," in *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*.
[5] N. Heintze and J. G. Riecke, "The slam calculus: Programming with secrecy and integrity," in *Proc. 25th ACM SIGPLAN-SIGACT Symp. POPL*. ACM, 1998.
[6] D. Chisnall, "C is not a low-level language," *Commun. ACM*, vol. 61, no. 7, Jun. 2018.
[7] J. McMahan, M. Christensen, L. Nichols, J. Roesch, S.-Y. Guo, B. Hardekopf, and T. Sherwood, "An architecture supporting formal and compositional binary analysis," in *Proc. 22nd ASPLOS*. ACM, 2017.
[8] C. Melo, P. Liu, and R. Ying, "A platform for full-stack functional programming," in *2020 IEEE Int.Symp.Circuits and Systems*, 2020.
[9] J. Pope, J. Saget, and C.-J. H. Seger, "Cephalopode: A custom processor aimed at functional language execution for iot devices." in *2020 18th ACM-IEEE Int.Conf.Formal Methods and Models for System Design (MEMOCODE)*, 2020.
[10] S. Broda and L. Damas, "Compact bracket abstraction in combinatory logic," *The Journal of Symbolic Logic*, vol. 62, 1997.
[11] S. Peyton Jones, *The Implementation of Functional Programming Languages*. Prentice Hall, January 1987.
[12] D. Turner, "Another algorithm for bracket abstraction," *J. Symb. Log.*, vol. 44, 1979.
[13] M.W.Bunder, "Some improvements to turner's algorithm for bracket abstraction," *J.Symb.Log.*, vol. 55, 1990.
[14] R. M. F. Lima, R. D. Lins, and A. L. M. Santos, "A back-end for ghc based on categorical multi-combinators," in *Proceedings of the 2004 ACM Symposium on Applied Computing*, ser. SAC '04. ACM, 2004.
[15] O. Kiselyov, "λ to ski, semantically - declarative pearl," in *Proc.14th Int.Symp.Functional and Logic Programming*, 2018.
[16] P. H. Hartel and A. H. Veen, "Statistics on graph reduction of sasl programs," *Softw. Pract. Exper.*, vol. 18, no. 3, 1988.
[17] "Nofib haskell benchmark suite," https://gitlab.haskell.org/ghc/nofib, accessed: 2022-6-10.
[18] M. Naylor and C. Runciman, "The reduceron reconfigured," in *Proceedings of the 15th ACM SIGPLAN Int.Conf.on Functional Programming*, 2010.
[19] L. Augusteijn, "Sorting morphisms," in *Advanced Functional Programming*. Springer, 1999.
[20] R. J. M. Hughes, "Super-combinators a new implementation method for applicative languages," in *Proc.1982 ACM Symp.LISP and Functional Programming*. ACM, 1982.
[21] C. Elliott, "Compiling to categories," *Proc. ACM Program. Lang.*, 2017.
[22] A. Boeijink, P. K. F. Hölzenspies, and J. Kuper, "Introducing the pilgrim: A processor for executing lazy functional languages," in *Implementation and Application of Functional Languages*. Springer, 2011.
[23] E. Meijer, M. Fokkinga, and R. Paterson, "Functional programming with bananas, lenses, envelopes and barbed wire," in *Functional Programming Languages and Computer Architecture*. Springer, 1991.