KappaMutor: A Compact Structured Combinator Processor for Haskell

Yukang Xie[1[0009−0001−4707−7203]], Craig Ramsay[1[0000−0002−8198−0746]], Robert Stewart[1[0000−0003−0365−693X]], and Hans-Wolfgang Loidl[1[0000−0001−6318−1732]]

Heriot-Watt University, Mathematical and Computer Sciences, Edinburgh, UK
{yx3007,Craig.Ramsay,R.Stewart,H.W.Loidl}@hw.ac.uk

Abstract. This paper presents KappaMutor, a new hardware graph reduction architecture. It is based on structured combinators, a recently proposed, efficient 27-bit encoding of millions of combinators. More flexible than fine-grained SKI combinators, they enable a more direct translation of functions to combinators. KappaMutor employs parallel memories to execute structured combinators in a single clock cycle. The architecture uses pipelining for heap writes, hides the latency of heap updates and enables combinators to self-optimise. Based on our measurements, runtimes are 11% to 58% shorter than running equivalent SKI combinator programs in our KappaMutor architecture.

Keywords: Graph reduction · Structured combinators · Hardware design · Haskell.

1 Introduction

For the implementation of functional languages, the main research direction in the 1980s was to develop custom hardware. Many graph reduction processors were developed to support lazy evaluation in hardware, including [6,18,7,21,11] (see Section 6). In the 1990s, with hardware design processes unable to keep pace with RISC processor advances, efforts shifted to compiler techniques[1] [12]. Recent projects have re-explored the original architectural ideas in light of stagnant clock frequencies in general purpose CPUs, compounded by continued advances in hardware design tooling. In particular, FPGAs have continued to enjoy exponential growth of logic density, adoption of multi-ported memories, and wide availability via cloud services.

Early graph reduction systems supported a small set of *fixed* SKI combinators, relying on compilers to split user-defined functions into these fine-grained computation steps. Later systems supported *user-defined* supercombinators, allowing more direct execution of user-defined functions (see Section 6). Accetti recently proposed *structured combinators* [1] to get the benefits of both approaches: (1) a *minimal runtime system* similar to that of a fixed combinator set; (2) *smaller generated programs* enabled by deriving the shape of combinators directly from user functions.

Accetti developed the *Fun* instruction set to encode structured combinators, and the *Blackbird-II* processor to evaluate *Fun* programs [1]. Its input language is formed of lambda terms, and its compiler linearises these into instruction sequences that correspond to the evaluation of structured combinators. In the general case, the compiler converts lambda abstractions first to fixed-set combinators using standard methods, followed by a traversal of the combinator graph, merging small combinators into larger structured combinators.

[1] The authors provide a curated history of functional architectures at: https://haflang.github.io/history.html

This paper presents *KappaMutor*, a new structured combinator architecture. It differs from Fun in three ways:

Language: KappaMutor's input language is Haskell 2010 [13], providing the user a range of convenient modern functional language features. This is thanks to MicroHs [3], which we have extended with a structured combinator backend.

Compiler: Our structured combinator backend takes a bracket abstraction styled approach with three expression combination strategies, merging combinators on-the-fly to reduce program size and runtime reduction steps.

Architecture: The architecture has two forms of memory parallelism: (1) independent memories for the heap, the reduction stack and the update stack, and (2) parallel access to the 8 top-of-stack elements. The architecture uses pipelining for heap writes to achieve *one-combinator-per-cycle* reductions and hides the latency of heap updates.

This paper makes the following **contributions**:

- An optimising structured combinators compiler for full Haskell 2010. Its three strategies achieve two goals: (1) minimising the combinator count of generated programs and (2) meeting the architectural constraints (i.e., the number of holes in a reduction pattern) imposed by the structured combinators architecture (see Section 3).
- KappaMutor, an optimised structured combinator architecture with parallel memories to achieve one-combinator-per-cycle reductions (see Section 4).
- An experimental evaluation showing that our compiler generates smaller combinator programs than MicroHs's SKI backend in all 13 of our benchmarks. When comparing both backends on our KappaMutor architecture, the structured combinators approach reduces runtimes by 11% to 58% (see Section 5).

2 Structured Combinators

2.1 Background

In the context of combinatory logic systems, combinators are symbols that represent simple primitive functions. Designers of hardware architectures for functional languages in the 1980s chose combinator schemes such as SKI [19] as the intermediary between user code and hardware execution, due to their simplicity and known approaches to convert the lambda calculus to combinators. These systems typically supported tens of combinators. Even though custom hardware meant each reduction could be executed efficiently, the downside of small combinator sets was large compiler-generated SKI programs, resulting in more reduction steps and slow runtime performance (Section 6).

Accetti's recent work on structured combinators [1] addresses these limitations. By encoding millions of combinators in a compact 27-bit format, structured combinators offer a more expressive and efficient representation. The hardware overhead of this approach is primarily limited to a 6-bit decoder (Section 2.2), a relatively inexpensive component. The flexibility of structured combinators makes it possible to compile functional programs to far fewer combinators. Fig. 1 illustrates how the Fibonacci function can be represented using both classic SKI combinators and structured combinators. Notably, the SKI representation requires 8 combinators, while the structured combinator representation needs only 3. Fewer combinators reduce the reduction steps required to execute programs, thus improving runtime performance versus SKI-style architectures. We validate this result in Section 5.2 with 13 Haskell benchmarks.

Haskell source

```haskell
fib n = if n <= 1
          then 1
          else fib (n - 2) + fib (n - 1)
```

SKI combinators

$fib : \mathbf{C}(\mathbf{S}(\mathbf{P}(\leq)1)(\mathbf{S}(\mathbf{C}' fib(\mathbf{P}(-)2)(+))(\mathbf{B} fib(\mathbf{P}(-)1))))1$

Structured combinators

$fib : C^{\mathrm{XXX(XX)X}}_{5[4,0,1,2,4,3]}(\leq)1(C^{\mathrm{XX(X(XXX))}}_{5[0,4,1,4,2,3]}(C^{\mathrm{X(XXX)X}}_{5[0,4,1,2,3]} fib(-)2(+))fib(-)1)1$

Fig. 1: The Fibonacci function, compiled to classic SKI combinators and structured combinators.

2.2 Definition and Encoding

Structured combinators are machine-friendly encodings, which are self-contained definitions, where the structure of their graph-reduction semantics is explicit. They are predefined graph transformations that explicitly capture the arity, the reduction pattern and the contents of the new graph nodes. We follow [1] to present the definition of structured combinators.

Definition 1. *A structured combinator $C^{p}_{a\,i}$ is a 3-tuple:*

- ***Arity*** *a: The number of arguments the combinator expects.*
- ***Structural pattern*** *p: The shape of the lambda abstraction's body of the combinator, independent of specific variable names.*
- ***Index list*** *i: A list of de Bruijn indices indicating the order of argument usage in the reduction pattern.*

This 3-tuple representation is equivalent to the corresponding λ-form, providing sufficient information for reduction. For instance, expression $C^{\mathrm{XX(X(XX))}}_{4[0,3,2,1,2]}\ e_1e_2e_3e_4$ will be reduced to $e_1e_4(e_3(e_2e_3))$. Table 1 is borrowed from [1] to show how the common SKI combinators can be represented as structured combinators.

Table 1: Common SKI combinators and their structured combinator representation

SKI combinator	λ-form	Pattern	Structured form
S	$\lambda abc.ac(bc)$	XX(XX)	$C^{\mathrm{XX(XX)}}_{3[0,2,1,2]}$
K	$\lambda ab.a$	X	$C^{\mathrm{X}}_{2[0]}$
I	$\lambda a.a$	X	$C^{\mathrm{X}}_{1[0]}$
B	$\lambda abc.a(bc)$	X(XX)	$C^{\mathrm{X(XX)}}_{3[0,1,2]}$
C	$\lambda abc.acb$	XXX	$C^{\mathrm{XXX}}_{3[0,2,1]}$

We employ the same encoding scheme as [1] to represent structured combinators in our KappaMutor architecture, as depicted in Fig. 2. Each combinator requires 27 bits: 6 bits for 64 possible

reduction patterns (with 1 to 6 holes), 3 bits for arity (up to 6), and 3 bits for each de Bruijn index in the index list.

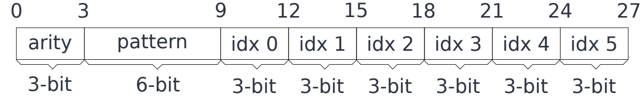


Fig. 2: The 27-bit structured combinator encoding.

To calculate the number of possible structured combinators, we first define $h(n)$ as the number of reduction patterns with n holes. This function can be recursively defined as:

$$h(n) = \begin{cases} 1, & \text{if } n = 1 \\ \sum_{i=1}^{n-1} h(i) \times h(n-i), & \text{otherwise} \end{cases}$$

Using this definition, the total number of combinators supported by KappaMutor can be calculated:

$$\sum_{i=1}^{6}(\sum_{j=1}^{6} h(j) \times i^j) = 3,004,365$$

Therefore, with the 27-bit encoding scheme, KappaMutor can support over 3 million unique combinators.

3 Compiler Design

3.1 Overview

Augustsson's MicroHs compiler [3] compiles Haskell 2010 down to a fixed set of 19 combinators. Our compiler (Fig. 3) extends MicroHs to support over 3 million structured combinators.

KappaMutor's compiler accepts Haskell 2010 as its input language, leveraging the MicroHs frontend. As KappaMutor currently lacks support for floating-point operations and I/O (Section 4), we employ a limited prelude that includes definitions for `Int` and `Bool`. Lists, tuples and user-defined data types are supported. The frontend of MicroHs is reused to transform Haskell code to an intermediate representation, which is essentially the λ-calculus with literals and primitive operators. Our compiler then removes all bound variables from the λ-calculus representation to construct a structured combinator graph for KappaMutor. Our binary generator then transforms the resulting graph into a format compatible with our Chisel-based KappaMutor implementation (Section 4).

The rest of this section focuses on our work on code generation, from the λ-calculus to structured combinators. We first present our baseline code generation scheme (Section 3.2), then show two optimisations that exploit the advantages of structured combinators to reduce combinators in generated code (Section 3.3).

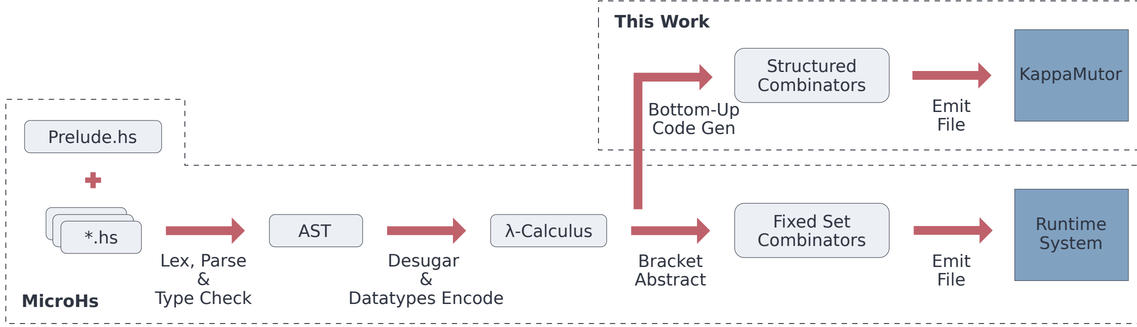


Fig. 3: Compilation workflow for KappaMutor.

3.2 Bottom-Up Structured Combinator Code Generation

In [1], Accetti proposed a compilation scheme to transform the λ-calculus to structured combinators. The scheme first attempts pattern matching on the shape of lambda bodies. If pattern matching succeeds (i.e., the pattern is supported by structured combinators), the lambda abstraction can be directly converted to an applied combinator. If the pattern matching fails, the compiler will transform the lambda abstraction to fixed-set combinators using standard methods (e.g., bracket abstraction [10]), then walk through the combinator graph to merge small combinators into bigger structured combinators.

We propose a different approach in this paper. Our scheme resembles bracket abstraction, building the combinator graph bottom-up in one go.

Bracket Abstraction Styled Algorithm The presentation of Fig 4 is an extension of [10] to contrast the classic bracket abstraction algorithm with our code generation scheme. The compilation function, $\mathcal{C}[e]$, recursively removes all lambdas inside e. When a lambda abstraction is encountered, $\mathcal{A}\,x[e]$ is used to abstract out bounded variable x from e. For basic bracket abstraction, we use the notation $\mathcal{A}_{\mathrm{b}}$. In contrast, our scheme employs $\mathcal{A}_{\mathrm{s}}$. Note that the input expression of $\mathcal{A}\,x$ already does not contain any lambdas, since $\mathcal{C}$ is first applied to the lambda body. We annotate variables and primitives, including combinators, as c.

$\mathcal{C}[e]$ compiles e to combinators	$\mathcal{A}_{\mathrm{b}}\,x[e]$ abstracts x from e	$\mathcal{A}_{\mathrm{s}}\,x[e]$ abstracts x from e
$\mathcal{C}[e_1\ e_2] = \mathcal{C}[e_1]\ \mathcal{C}[e_2]$	$\mathcal{A}_{\mathrm{b}}\,x[e_1\ e_2] = \mathbf{S}\ (\mathcal{A}_{\mathrm{b}}\,x[e_1])\ (\mathcal{A}_{\mathrm{b}}\,x[e_2])$	$\mathcal{A}_{\mathrm{s}}\,x[e_1\ e_2] = \mathcal{M}[\mathcal{A}_{\mathrm{s}}\,x[e_1]]\ [\mathcal{A}_{\mathrm{s}}\,x[e_2]]$
$\mathcal{C}[\lambda x.e] = \mathcal{A}\,x[\mathcal{C}[e]]$	$\mathcal{A}_{\mathrm{b}}\,x[x] = \mathbf{I}$	$\mathcal{A}_{\mathrm{s}}\,x[x] = C^{\mathrm{X}}_{1[0]}$
$\mathcal{C}[c] = c$	$\mathcal{A}_{\mathrm{b}}\,x[c] = \mathbf{K}\ c$, if $c \neq x$	$\mathcal{A}_{\mathrm{s}}\,x[c] = C^{\mathrm{X}}_{2[0]}\ c$, if $c \neq x$

Fig. 4: Basic bracket abstraction and the structured combinator version.

The MicroHs compiler generates code using bracket abstraction with optimisation rewrites, producing a fixed set of combinators. Our code generation algorithm follows the similar structure of bracket abstraction to build structured combinators. We adopt the same main translation as $\mathcal{C}$. For

the two base cases of the abstraction function, structured combinators $C^{\mathrm{X}}_{1[0]}$ and $C^{\mathrm{X}}_{2[0]}$ are produced, instead of the **I** and **K** combinators. For the compound case, we require more delicately designed rules, annotated as $\mathcal{M}[e_1][e_2]$, to merge sub-expressions on-the-fly while minimising the number of combinators generated in the final compilation result.

Sub-Expression Combination To explain the design of $\mathcal{M}$, we first give the definition of *unary form.*

Definition 2. *Expression $e_0 e_1 \cdots e_n$ is in unary form if e_0 is a structured combinator of arity $n+1$.*

We claim that if the output of $\mathcal{M}$ is always a unary-form expression, then its input expressions must also be in unary form.

The combination function $\mathcal{M}$ employs three strategies to merge sub-expressions, namely *absorption*, *extension* and *addition*. *Absorption*|/ is the ideal strategy because it merges two structured combinators into one, thus the reduction of an expression takes a single cycle rather than two in our KappaMutor architecture. It combines the leading combinators of the two sub-expressions by merging their reduction patterns and index lists. This strategy assumes the two input expressions are in unary form. It can only be applied when the resulting arity and reduction pattern are within the constraints of KappaMutor's architecture. Specifically, a structured combinator's arity should not exceed 6, and its pattern can contain up to 6 holes (Section 2.2). If absorption fails, the compiler attempts to apply the *extension* strategy, which preserves one input expression, and extends the other's leading combinator to 'send' the abstracted variable down to the preserved expression. This also requires the arity and reduction pattern of the structured combinator to fit within KappaMutor's architectural constraints. If extension also fails then the compiler falls back to the *addition* strategy, which is equivalent to bracket abstraction. A $C^{\mathrm{XX(XX)}}_{3[0,2,1,2]}$ (i.e., the **S** combinator) is added on top of the input expressions.

Fig. 5 illustrates examples of the three strategies, showing how absorption reduces combinators, extension preserves them, and addition increases them. Note that the extension strategy can be applied to either the left or the right input expression. Additionally, all of the strategies will only produce unary-form expressions.

$$\mathcal{M}\ [C^{\mathrm{XXX}}_{3[2,0,1]} e_1 e_2]\ [C^{\mathrm{X(XX)}}_{3[1,2,0]} e_3 e_4] \ \rightsquigarrow_{\text{Absorption}}\ C^{\mathrm{XXX(X(XX))}}_{5[4,0,1,3,4,2]} e_1 e_2 e_3 e_4$$

$$\mathcal{M}\ [C^{\mathrm{XX}}_{2[1,0]} e_1]\ [C^{\mathrm{X(XXX)X}}_{4[0,2,3,0,1]} e_2 e_3 e_4] \ \rightsquigarrow_{\text{Extension}}\ C^{\mathrm{XX(XX)}}_{3[2,0,1,2]} e_1 (C^{\mathrm{X(XXX)X}}_{4[0,2,3,0,1]} e_2 e_3 e_4)$$

$$\mathcal{M}\ [C^{\mathrm{X(XXX)X}}_{4[0,2,3,0,1]} e_1 e_2 e_3]\ [C^{\mathrm{XX}}_{2[1,0]} e_4] \ \rightsquigarrow_{\text{Extension}}\ C^{\mathrm{XX(XX)}}_{3[0,2,2,1]} (C^{\mathrm{X(XXX)X}}_{4[0,2,3,0,1]} e_1 e_2 e_3) e_4$$

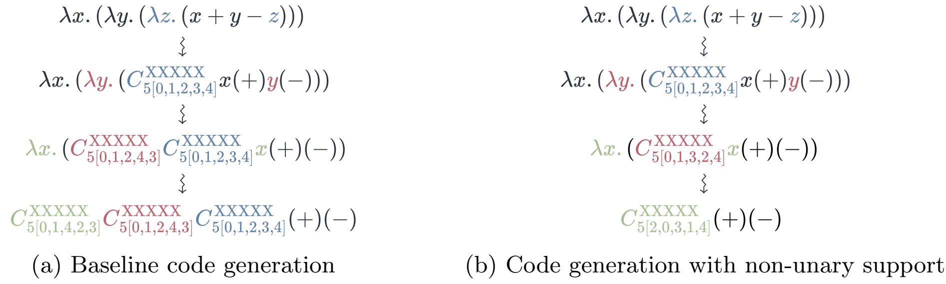
$$\mathcal{M}\ [C^{\mathrm{X(XXX)X}}_{4[0,2,3,0,1]} e_1 e_2 e_3]\ [C^{\mathrm{XXX(XX)}}_{5[0,1,3,4,2]} e_4 e_5 e_6 e_7] \ \rightsquigarrow_{\text{Addition}}\ C^{\mathrm{XX(XX)}}_{3[0,2,1,2]}\ (C^{\mathrm{X(XXX)X}}_{4[0,2,3,0,1]} e_1 e_2 e_3)\ (C^{\mathrm{XXX(XX)}}_{5[0,1,3,4,2]} e_4 e_5 e_6 e_7)$$

Fig. 5: Examples of the absorption, extension and addition strategies.

3.3 Optimisations with Structured Combinator

We now present two optimisations to our code generation scheme. We begin by supporting the combination of non-unary formed expressions. This improvement is designed to optimise the results of nested calls to $\mathcal{A}_\mathrm{s}$, which are commonly introduced by curried functions and inner lambda abstractions. We then show how classic SKI combinator optimisation rewrites can be generalised and applied to structured combinators.

Combining Non-Unary Formed Expressions: The scheme described in Section 3.2 generates functionally correct code. However, it can be inefficient when applied to curried functions or functions with inner lambda abstractions, which lead to nested calls to $\mathcal{A}_\mathrm{s}$. For instance, consider the example in Fig. 6a. The compilation process produces three successively applied combinators which takes three clock cycle to execute. An obviously more efficient representation would be $C^{\mathrm{XXXXX}}_{5[2,0,3,1,4]}(+)(-)$, which can be reduced in one clock cycle.



(a) Baseline code generation (b) Code generation with non-unary support

Fig. 6: Abstracting bound variables from curried functions.

To improve the scheme, we first need to give combinators special treatment in the base case of $\mathcal{A}_\mathrm{s}\,x[e]$. Specifically:

- For $\mathcal{A}_\mathrm{s}\,x[C^{p}_{a\,i}]$, attempt to create $C^{p}_{a'\,i'}$ as result, where $a' = a + 1$, and $i' = \mathrm{map}\,(+1)\,i$.
- If the result exceeds the architectural constraints of KappaMutor, revert to the less efficient $C^{\mathrm{X}}_{2[0]}\;C^{p}_{n\,i}$.

The revised base cases for $\mathcal{A}_\mathrm{s}\,x[e]$ may yield non-unary expressions, which contradicts the underlying assumption of the absorption and extension strategies presented in Section 3.2. Consequently, $\mathcal{M}$ requires new rules to accommodate these non-unary expressions. The addition strategy remains applicable, while the absorption and extension strategies necessitate modified combination logic. Fig. 7 provides examples illustrating the non-unary versions of these two strategies.

The compiler checks the unary nature of input expressions and selects appropriate strategies accordingly. The non-unary version of the absorption strategy is applicable when the left input expression is non-unary and the right input expression is unary. Otherwise, the extension strategy is employed. These enhancements effectively address the issue of stacked combinators resulting from nested calls to $\mathcal{A}_\mathrm{s}$. Fig. 6b shows how the improved code generation handles the same example.

$$\mathcal{M}\ [C^{\mathrm{XXX}}_{3[2,0,2]} e_1]\ [C^{\mathrm{XX}}_{2[1,0]} e_2] \quad \rightsquigarrow \text{Absorption} \quad C^{(\mathrm{XX})\mathrm{X}(\mathrm{XX})}_{3[2,1,0,2,1]} e_1 e_2$$

$$\mathcal{M}\ [C^{\mathrm{XXX}}_{3[2,0,2]} e_1]\ [C^{\mathrm{XX(XXX)}}_{5[1,0,3,4,2]} e_2 e_3] \quad \rightsquigarrow \text{Extension} \quad C^{(\mathrm{XX})\mathrm{X}(\mathrm{XX})}_{4[1,2,0,1,2]} e_1 (C^{\mathrm{XX(XXX)}}_{5[1,0,3,4,2]} e_2 e_3)$$

Fig. 7: Examples of the absorption the extension strategies, applied to non-unary-form expressions.

Generic Rewrites: Classic SKI compilation schemes rely on predefined rewrite rules to optimise code by reducing the number of combinators in generated programs. We propose a similar approach for structured combinator compilers, but instead of hardcoding rewrite rules, we advocate for a more generic optimisation strategy based on the potential for optimisation within structured combinators.

Classic SKI rewrites	Addition strategy with rewrites
$\mathbf{S}\ (\mathbf{K}\ e_1)\ (\mathbf{K}\ e_2) = \mathbf{K}\ (e_1 e_2)$	$\mathcal{M}\ [C^{\mathrm{XX(XX)X}}_{4[0,2,3,2,0]}\ e_1][C^{\mathrm{XXXXX}}_{4[1,3,0,1,0]}\ e_2 e_3]$ $\rightsquigarrow$ Addition $C^{\mathrm{X}}_{2[0]}\ ((C^{\mathrm{XX(XX)X}}_{3[0,1,2,1,0]}\ e_1)(C^{\mathrm{XXXXX}}_{3[1,2,0,1,0]}\ e_2 e_3))$
$\mathbf{S}\ (\mathbf{K}\ e_1)\ e_2 = \mathbf{B}\ e_1 e_2$	$\mathcal{M}\ [C^{\mathrm{XX(XX)X}}_{4[0,2,3,2,0]}\ e_1][C^{\mathrm{XXXXX}}_{4[1,3,2,1,2]}\ e_2 e_3]$ $\rightsquigarrow$ Addition $C^{\mathrm{X(XX)}}_{3[0,1,2]}\ (C^{\mathrm{XX(XX)X}}_{3[0,1,2,1,0]}\ e_1)\ (C^{\mathrm{XXXXX}}_{4[1,3,0,1,0]}\ e_2 e_3)$
$\mathbf{S}\ e_1\ (\mathbf{K}\ e_2) = \mathbf{C}\ e_1 e_2$	$\mathcal{M}\ [C^{\mathrm{XX(XX)X}}_{4[0,2,3,1,0]}\ e_1][C^{\mathrm{XXXXX}}_{4[1,3,0,1,0]}\ e_2 e_3]$ $\rightsquigarrow$ Addition $C^{\mathrm{XXX}}_{3[0,2,1]}\ (C^{\mathrm{XX(XX)X}}_{4[0,2,3,1,0]}\ e_1)\ (C^{\mathrm{XXXXX}}_{3[1,2,0,1,0]}\ e_2 e_3)$

Fig. 8: Classic SKI optimisation rewrites and examples of the improved addition strategy.

Among the numerous SKI rewrite rules, three are particularly fundamental, as illustrated in Fig. 8. These rewrites occur during the compound case of bracket abstraction, where two sub-expressions are merged. The underlying principle of these rules is to avoid unnecessary transmission of the abstracted bound variable when it is not used in a sub-expression [10]. Traditional rewrites achieve this by examining the structure of the sub-expression: the bound variable is deemed unused if the sub-expression takes the form $(\mathbf{K}\ e)$. The three SKI rewrites in Fig. 8 correspond to the following scenarios:

– **Mutual disuse**: If the bound variable is not used in either sub-expression, it is omitted from both.
– **Right-side usage**: If the bound variable is only used in the right sub-expression, the **B** combinator is employed to transmit the variable.

– **Left-side usage**: If the bound variable is only used in the left sub-expression, the **C** combinator is used to transmit the variable.

With structured combinators, we can more generically determine whether a bound variable is unused within a sub-expression: for an expression $C^{p}_{a\,i}\ e_1 \cdots e_n$, the bound variable is unused if n is absent from i. Fig. 8 illustrates how similar rewrites can be applied to the addition strategy in our compiler. When executing the addition strategy, the compiler prioritises these rewrites over the introduction of the $C^{\mathrm{XX(XX)}}_{3[0,2,1,2]}$ combinator. Similar optimisations can be applied to the extension strategy. For instance, $\mathcal{M}\ [C^{\mathrm{XX}}_{2[1,0]}e_1]\ [C^{\mathrm{X(XXX)X}}_{4[0,3,1,0,1]}e_2e_3]$ will yield $C^{\mathrm{XXX}}_{2[2,0,1]}e_1(C^{\mathrm{X(XXX)X}}_{4[0,3,1,0,1]}e_2e_3)$, where the extension strategy is enhanced by the SKI-styled rewrites.

Another category of generic rewrites applicable to structured combinators is based on η-reduction (i.e., $\lambda x.(f\ x) = f$). For an expression $C^{p}_{a\,i}\ e_1 \cdots e_n$, η-reduction is possible if n appears solely as the final element of i, and p conforms to the pattern $p'\mathrm{X}$, where p' represents an arbitrary pattern. In the extension and addition strategies, the compiler consistently attempts the η-reduction styled rewrites on preserved input sub-expressions. For example, $\mathcal{M}\ [C^{\mathrm{XXX}}_{3[1,2,0]}e_1e_2]\ [C^{\mathrm{X(XX)X}}_{4[0,2,1,3]}e_3e_4e_5]$ will produce $C^{\mathrm{XXX(XX)}}_{4[1,3,0,2,3]}e_1e_2(e_3e_5e_4)$, where the extension strategy is optimised through η-reduction-style rewrites.

4 The KappaMutor Architecture

KappaMutor combines the foundation of Reduceron's [14] hardware architecture with a graph reduction model based on Accetti's structured [1] combinators. Together, they exploit parallel memory access across dedicated memory units, bypassing costly node searching and allocation operations inherent in heap-centric architectures. We implement KappaMutor in Chisel [4], a Scala-embedded hardware description language. The implementation currently supports Accetti's 27-bit structured combinator encoding (Section 2.2), integers, and basic arithmetic primitives. This section details KappaMutor's design, beginning with the program representation, followed by the machine's reduction model, and concluding with a discussion of key features and optimisations.

4.1 Program Representation

We adopt the style of [14] to present the program representation in our architecture, using Haskell code (prefixed with '>') for clarity.

In KappaMutor, the source program is compiled into a sequence of top-level functions, each represented as a list of applications. This list comprises the spine application and nested applications of the function. The compiled program is a concatenation of these top-level function lists, forming a single list of applications.

```
> type Func = [App]
> type Prog = [App]
```

This definition diverges from template instantiation-based designs like Reduceron and Heron [16], which define functions as 3-tuples comprising arity, spine application, and a list of nested applications.

Applications in the architecture are flattened, meaning they have no nested applications and can be represented as a list of atomic elements.

```
> type App = [Atom]
```

Atomic elements in KappaMutor can be represented as a sum of products type.

```
> data Atom =
>    PTR Int |                -- Pointer to an application
>    COM Arity Pat [Idx] |    -- Structured combinator
>    INT Int |                -- Integer literal
>    PRM Int |                -- Primitive operator with an opcode
>    Y                        -- Y-combinator
```

Our definition of `Atom` deviates from Reduceron's, omitting function and argument pointers since β-reduction is not used in KappaMutor. In our design, a `PTR` pointing to the spine application of a top-level function effectively serves as a function pointer. Additionally, we exclude data constructors and pattern matching case tables, as they are eliminated by MicroHs's Scott encoding scheme. We introduce structured combinators as a new atomic category. As described in Section 2.2, a structured combinator is a 3-tuple comprising arity, reduction pattern (encoded as a 6-bit field in hardware), and a list of de Bruijn indices.

```
> type Arity = Int
> data Pat = X | At Pat Pat
> type Idx = Int
```

In KappaMutor, each `atom` is represented using 32 bits, including its tag. As introduced in Section 2.2, the arity upper bound is 6, and the 6-bit pattern field supports reduction patterns with up to 6 holes. We limit the maximum application length to 6, as a combinator consumes at most 6 arguments, and compiled applications are typically under-applied. For longer applications, the compiler splits them into multiple smaller in-sized ones.

4.2 Reduction Model

KappaMutor's execution is centered around a reduction stack and its associated control logic. Fig. 9 illustrates the architecture's datapath. The system incorporates three specialised memory units: a dual-port heap for storing `App` structures, an update stack for handling `UpdateRecord`s (Section 4.3), and the reduction stack for storing `Atom` elements.

We adopt Reduceron's stack implementation, enabling asynchronous reads and synchronous writes to the top 8 stack elements. Additionally, we employ Reduceron's infix primitive application scheme to maintain strict evaluation semantics for primitive operators.

At startup, the pointer to the entry function is placed on the top of the reduction stack. At each clock cycle, the type of the top `Atom` on the stack determines the next reduction rule to be applied:

- `PTR`: The pointed application is fetched from the heap and pushed onto the reduction stack.
- `COM`: The combinator and its arguments, as specified by the arity field, are popped from the reduction stack. The combinator's pattern guides the reduction logic in modifying the stack and creating nested applications on the heap.
- `INT`: The type of the `Atom` at position `(top-2)` will be checked. If it's an `INT`, the ALU result is pushed onto the reduction stack; otherwise, a swap operation is performed to evaluate the second ALU input.
- `PRM`: The top two elements of the reduction stack are swapped to preserve the infix order of primitive operators.

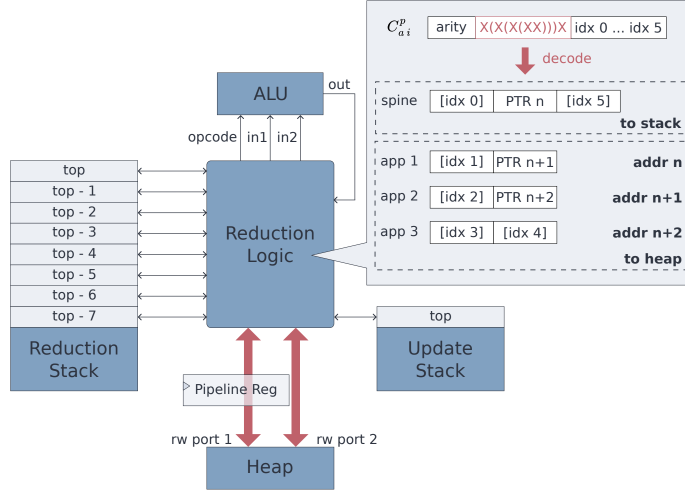


Fig. 9: Datapath diagram of the prototype architecture. In the example, `[a]` refers to the reduction stack element at position `(top-a-1)`.

– `Y`: Following the reduction rule $\mathbf{Y}R = R(\mathbf{Y}R)$, the second-top element of the reduction stack is consumed to create a self-referential application on the heap.

Most reduction rules are similar to those in [14], with the exception of structured combinator (`COM`) reduction. Fig. 9 illustrates an example of structured combinator application. The 6-bit reduction pattern is decoded to guide the construction of the spine and nested applications of the result, while the index list selects the appropriate arguments from the reduction stack.

Execution terminates when the stack depth falls below the required number of arguments for a reduction rule.

4.3 Architecture Features

We now present three features in the KappaMutor design, which differentiate it from template instantiation based architectures like Reduceron and Heron. We first explain the pipeline design in KappaMutor, which improves the performance when multiple nested applications are created on the heap. Secondly, we introduce the heap update mechanism in KappaMutor, and our optimisation to minimise its runtime overhead. Finally, we show how the memory layout of KappaMutor allows top-level functions to be automatically improved during runtime.

Pipelined Heap Writes In Reduceron [14], the author enforces a one-reduction-per-clock-cycle constraint by requiring function applications with more than two nested levels to be split across multiple operations. This ensures that the design maintains its strict single-cycle reduction property. KappaMutor faces a similar challenge: certain supported reduction patterns can generate up to three nested applications within a single reduction, which exceed the capabilities of a dual-port heap to process in one clock cycle.

Instead of splitting, we relaxed the one-reduction-per-clock-cycle requirement, allowing an additional cycle for heap writing when necessary. To mitigate potential performance impacts, we implemented a pipelined heap writing mechanism by introducing a pipeline register to the architecture (Fig. 9). This design offers two advantages. First, the extra heap writing cycle integrates seamlessly with the dual-port heap for structured combinators in KappaMutor. Second, the pipeline architecture ensures minimal performance degradation: execution stalls only when there is direct contention for heap access between consecutive reductions. In most scenarios, the additional writing cycle remains hidden, effectively improving overall throughput.

Hiding Heap Updates Heap updates are necessary in avoiding duplicated computation in lazy evaluation. When an application is loaded from the heap onto the reduction stack, its evaluation progress is tracked. Upon reaching normal form, the application is written back to the same heap address.

In [14], Naylor designed an efficient heap update mechanism which utilises an update stack (Fig. 9) to manage heap updates. When a `PTR` is dereferenced during reduction, an `UpdateRecord` is pushed onto the update stack, recording the address and current stack depth. Before applying any reduction rule, the architecture checks the update condition based on the current stack depth and the top `UpdateRecord`. If a heap update is necessary, the update is performed, and the top `UpdateRecord` is removed from the stack at that cycle.

KappaMutor adopts Reduceron's heap update mechanism but introduces an optimisation. Instead of dedicating separate clock cycles for heap updates, KappaMutor can often perform heap updates and reductions in parallel, provided the reduction doesn't require heap access. This optimisation is possible for specific `Atom` types:

- `PTR`: Never appears as the top element in a normal form.
- `COM`: If the combinator does not create any nested applications (e.g., $C^{\mathrm{XXXX}}_{4[3,0,1,2]}$), it can be reduced in parallel with a heap update.
- `INT` and `PRM`: These atom types can always be reduced in parallel with a heap update.
- `Y`: The **Y** combinator cannot be reduced in parallel with a heap update, as it requires heap access.

Another scenario that prevents parallelised heap updates occurs when a chain of updates is required. To detect and prevent incorrect parallel reductions in such cases, we extend the `UpdateRecord` with an additional field to track the stack depth of the previous update record:

```
> type StackDepth = Int
> type HeapAddr = Int
> type UpdateRecord = (StackDepth, HeapAddr, StackDepth)
```

By incorporating this optimisation, over half of the heap updates can be hidden in runtime, leading to a reduction in overall clock cycles.

Self-Optimising Property of Combinators Turner [20] highlighted the self-optimising property of combinator graph reduction machines, where top-level functions can be dynamically reduced to their normal form after their first execution. For example, in the following definition:

```
add x y = x + y
addOne = add 1
```

The `addOne` function can be optimised to `addOne y = 1 + y` after its initial evaluation during runtime. This property is particularly beneficial when the source code is in a combinatory programming style, as seen in functions built from simpler components like `foldr` [20]. Augustsson [3] further emphasised the importance of this property for handling overloaded functions in Haskell type classes.

In KappaMutor, both top-level functions and runtime applications reside within the same heap, which inherently enables self-optimising through heap updates. Fig 10 illustrates the first call to the `addOne` function. After this initial call, `addOne` is updated to its normal form. Subsequent calls to `addOne` can then bypass the pointer resolution of `add`, resulting in a one-cycle performance improvement for each call.

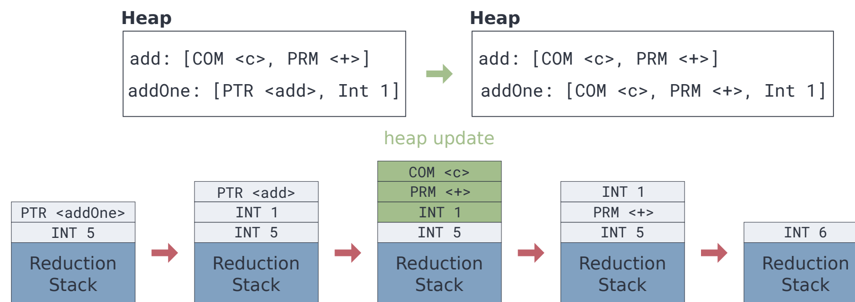


Fig. 10: Example of self-optimisation through heap updates, where `<c>` is the combinator $C^{\mathrm{XXX}}_{3[1,0,2]}$.

Turner [20] suggested that self-optimisation should also be achievable in other types of graph reduction systems, such as a template instantiation machine. However, since Reduceron places programs in a separate memory, and all the argument pointers are replaced by actual arguments during template instantiation, it is less straightforward to achieve the self-optimising property.

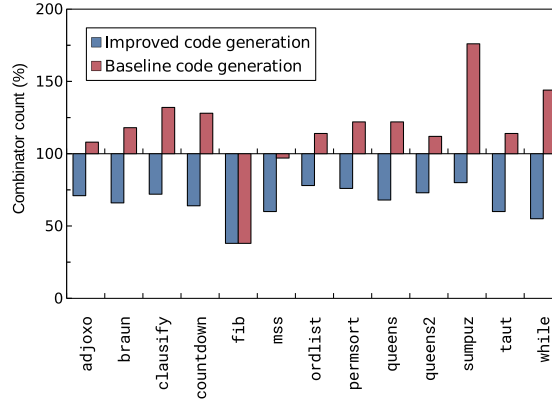
5 Evaluation

In this section, we evaluate structured combinators generated by our code generation scheme (Section 3), analysing both static code and runtime performance to highlight the advantages of structured combinators over classic SKI combinators. We also present the hardware implementation result of the KappaMutor architecture (Section 4), focusing on resource utilisation and maximum clock frequency to demonstrate the feasibility and compactness of our design.

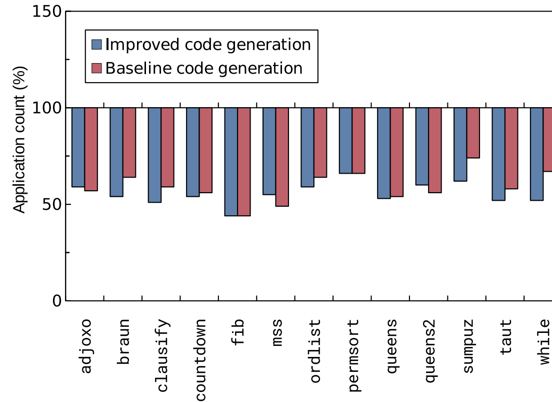
5.1 Compiler-Generated Code Analysis

Two metrics are employed for the analysis of compiler-generated programs: combinator count and application count. While not a direct indicator of runtime, combinator count can reflect the potential reduction steps within a program, which links to the running time, given the one-combinator-per-cycle nature of the KappaMutor core. Application count, on the other hand, is equivalent to program size according to KappaMutor's program representation (Section 4.1).

We compare three *code generation schemes:* the baseline structured combinator scheme from Section 3.2, the optimised scheme from Section 3.3, and the original SKI code generation from MicroHs. For comparison, we added a post-processing pass to the original MicroHs workflow, encoding its 19 combinators into the structured combinator representation for KappaMutor, without further modifications. The benchmark programs used in [14] and [16] were adopted for our evaluation. We translated the f-lite [14] source code to Haskell, as expected by the MicroHs frontend, and used a minimal prelude file for simplicity.



(a) Combinator count, normalised by MicroHs SKI code generation.



(b) Application count, normalised by MicroHs SKI code generation.

Fig. 11: Static code comparison. Improved code generation versus baseline code generation.

Fig. 11a and 11b present the combinator counts and application counts for the two structured combinator schemes, normalised against the MicroHs SKI scheme. The results demonstrate that our optimised scheme can reduce combinator count by 20% to 62% compared to the SKI scheme. Considering MicroHs already applies extensive optimisation rewrites to its SKI code generation,

this significant reduction underscores the power of structured combinators, aligning with Accetti's claims in [1].

It is unsurprising that the baseline scheme is outperformed by the SKI scheme in most benchmarks. This is due to the prevalence of curried functions and inner lambda abstractions which can significantly degrade code efficiency of the baseline scheme as discussed in Section 3.3. The results also validate the necessity of the optimisations we implemented. An interesting exception is the `fib` benchmark, where both structured combinator schemes yield identical results. This is because `fib` lacks curried functions and is the smallest benchmark, with the compiled code consisting of only three combinators.

In terms of application count (i.e., program size), both structured combinator schemes exhibit similar results, reducing the number by 30% to 55% compared to the SKI scheme. A possible explanation for the close similarity between the two structured combinator schemes lies in the shared decision-making when the absorption strategy fails (Section 3.2). This leads to similar graph layouts and application counts, despite the baseline scheme introducing a significantly higher number of combinators per application.

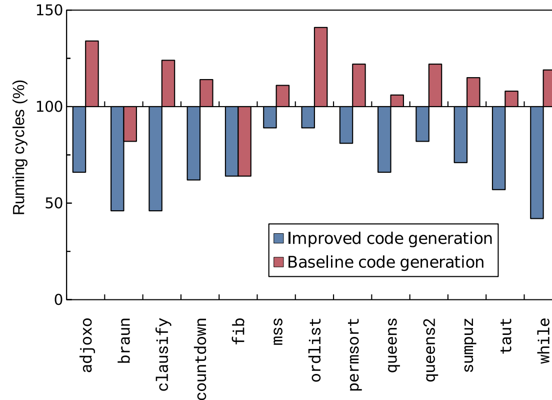
One might argue that the SKI scheme's use of 19 combinators could potentially reduce the required bit width for encoding, thus it is unfair to compare program sizes in terms of application count in a specific architecture like KappaMutor. However, other atomic elements, such as integers and pointers, necessitate wider bit widths. Thus, it is challenging to exclusively optimise the bit width for combinators in a hardware implementation of the SKI approach to achieve smaller program sizes.

5.2 Runtime Performance Analysis

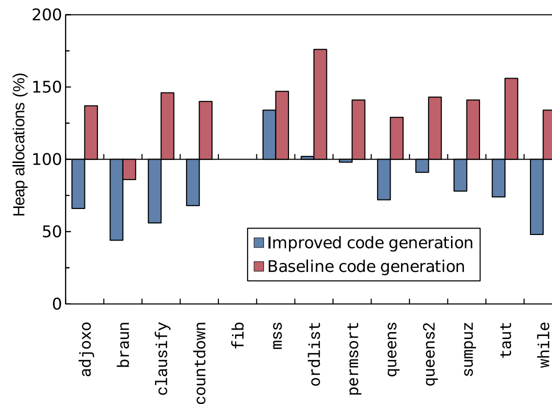
For runtime performance analysis, we focus on clock cycles and heap allocations during program execution on KappaMutor. The comparison is still made between the two structured combinator code generation schemes and MicroHs's SKI code generation. As discussed in Section 5.1, the Haskell benchmark programs are compiled using the three different code generation implementations. We then execute these compiled programs on our KappaMutor architecture with Chisel's simulation framework, collecting runtime statistics. Given the exact same underlying architecture, clock cycles directly correspond with wall-clock execution time. Additionally, heap allocation numbers provide insights into memory consumption and potential garbage collection overhead, although at present we have not yet implemented a garbage collector for KappaMutor.

Fig. 12a and 12b present the running cycle and heap allocation results for our structured combinator code generation, normalised against the MicroHs SKI code generation. The results indicate that our optimised structured combinator scheme can reduce running cycles by 11% to 58% compared to the SKI scheme. While the improvement is significant, the pattern of performance gains for structured combinators is not entirely predictable. We did not observe clear correlations between program size, the proportion of different reduction types, and performance across benchmarks like `braun`, `clausify`, `while`, `mss`, and `ordlist`. As discussed in Section 3.3, the baseline code generation's incompatibility with nested lambdas often leads to inferior performance compared to the SKI approach.

Regarding heap allocations, the results exhibit a similar pattern to running cycles, with two notable exceptions. First, the `fib` benchmark shows identical heap allocation results for all three schemes, likely due to its simplicity. Second, the optimised scheme generates a non-negligible increase in heap allocations (35%) for the `mss` benchmark compared to the SKI scheme. This suggests that



(a) Running clock cycles, normalised by MicroHs SKI code generation.



(b) Heap allocations, normalised by MicroHs SKI code generation.

Fig. 12: Runtime performance. Improved code generation versus baseline code generation.

reduced combinator counts and running cycles may not directly translate to lower heap allocations. We hypothesise that compile-time combinator selection can significantly influence heap allocation behaviour. As our primary focus in this paper is on reducing combinator count through compiler design, the relationship between structured combinators and runtime heap allocations warrants further investigation.

5.3 Hardware Implementation

The KappaMutor architecture is implemented on an Alveo U280 [2] UltraScale+ FPGA using Xilinx Vivado 2023.1 for synthesis and implementation. Table 2 outlines the parameter settings for the FPGA implementation. The reduction and update stacks are mapped to BlockRAM resources, while the dual-port heap is mapped to UltraRAM resources provided by the UltraScale+ device.

Table 2: Memory type and size of the implemented KappaMutor instance.

Memory unit	Element	Width	Depth
Heap	`App`	192-bit	64 k
Reduction stack	`Atom`	32-bit	4 k
Update stack	`UpdateRecord`	42-bit	512

Table 3 summarises the resource utilisation of KappaMutor. Look-up-table (LUT) and register resources are primarily consumed by rotation logic in the reduction stack and the overall reduction logic. BlockRAM resources are utilised by the two stacks. The utilisation of these resources is less than 1%, highlighting the compact nature of our combinator-based machine. UltraRAM utilisation is 5%, contributed by the 64k heap. These results suggest the potential for future scaling to a larger parallel design with tens of KappaMutor cores. The maximum clock frequency of the design is 165 MHz, a respectable result for an FPGA implementation. The implementation result demonstrates structured combinator's hardware-friendly nature.

Table 3: KappaMutor resource consumption on Alveo U280.

Resource type	Consumption	Available	Utilisation%
LUTs	7224	1303680	0.55
Registers	965	2607360	0.04
BlockRAMs	5	2016	0.25
UltraRAMs	48	960	5.00

6 Related Work

Turner demonstrated that combinators [19] had a practical application for implementing functional languages [20]. This insight led to the development of early hardware architectures that supported SKI combinators e.g. SKIM [6] and NORMA [18]. These architectures demonstrated the viability of combinator architectures with an extremely simple run-time systems, even inspiring recent hardware [15] and software (MicroHs) [3] implementations over 40 years later. SKI combinators, in particular, are a small *fixed* set of operations, and their compiler's job is to break down user functions into graphs built from many of these fine-grained operations.

Diverging from the minimalism of fixed combinator sets, Hughes showed how user applications could instead be compiled into coarser-grained supercombinators [8]. This insight led to supercombinator architectures e.g. Tim [7] and Flagship [21]. Because they supported *user defined* supercombinators derived directly from user functions, fewer combinators are executed, often resulting in much shorter runtimes [21]. The abstract G-machine [9] showed how to further specialise on user functions by compiling supercombinators to instructions for ISA-based processors (both special-purpose architectures e.g. GRIP [11] and PilGRIM [5], and general purpose CPUs e.g. GHC). Each top-level function is compiled to a sequence of G-machine instructions.

Similar to supercombinator hardware, recent architectures Reduceron [14] and Heron [16] support a near direct execution of user-defined functions using template instantiation, without the compiler doing too much heavy lifting (e.g., without splitting user functions into fine-grained SKI combinators). These two projects have also demonstrated that simple, non-pipelined, stack-based FPGA architectures are sufficient for reasonable performance — an approach that could be reused to rapidly prototype other graph reduction architectures.

There is a spectrum of design choices spanning fixed SKI combinators and supercombinator techniques. Structured combinators from [1] strike an interesting balance between a fixed set of SKI combinators and user-defined supercombinators. They enable the representation and execution of supercombinator-like structures, while retaining much of the run-time simplicity and self-optimisation properties of fixed set combinators.

7 Conclusion

This paper presents KappaMutor, a new graph reduction processor for Haskell. Its parallel memories allows single-cycle execution of structured combinators. Its compiler generates less machine code (fewer combinators) to execute versus fixed SKI-style combinators. The generated machine code is smaller across all of our 13 benchmarks when compared with restricting the MicroHs compiler to the SKI set. This results in reduced runtimes of between 11% and 58%. The simplicity of the KappaMutor architecture uses a fraction of available resources on a modern Alveo U280 UltraScale+ FPGA, using 1% of BRAMs and 5% of UltraRAMs. It also clocks at a relatively high 165MHz. The ample remaining hardware resources allows us to explore the design space for a multi-core KappaMutor architecture. We also plan on adding garbage collection with the available resources. Our recent work [17] gives us confidence that the latency of hardware-based concurrent collection on modern FPGAs can be almost entirely hidden.

Acknowledgments. This work is supported by the HAFLANG project, funded by the Engineering and Physical Sciences Research Council (EP/W009447/1).

References

1. Accetti, C., Ying, R., Liu, P.: Structured combinators for efficient graph reduction. IEEE Comput. Archit. Lett. **21**(2), 73–76 (2022). https://doi.org/10.1109/LCA.2022.3198844, https://doi.org/10.1109/LCA.2022.3198844
2. AMD: Alveo U280 data center accelerator card data sheet (ds963). https://docs.amd.com/r/en-US/ds963-u280 (2023)
3. Augustsson, L.: MicroHs: A small compiler for haskell. In: Vazou, N., Morris, J.G. (eds.) Proceedings of the 17th ACM SIGPLAN International Haskell Symposium, Haskell 2024, Milan, Italy, September 6-7, 2024. pp. 120–124. ACM (2024). https://doi.org/10.1145/3677999.3678280, https://doi.org/10.1145/3677999.3678280
4. Bachrach, J., Vo, H., Richards, B.C., Lee, Y., Waterman, A., Avizienis, R., Wawrzynek, J., Asanovic, K.: Chisel: constructing hardware in a scala embedded language. In: Groeneveld, P., Sciuto, D., Hassoun, S. (eds.) The 49th Annual Design Automation Conference 2012, DAC '12, San Francisco, CA, USA, June 3-7, 2012. pp. 1216–1225. ACM (2012). https://doi.org/10.1145/2228360.2228584, https://doi.org/10.1145/2228360.2228584

5. Boeijink, A., Hölzenspies, P.K.F., Kuper, J.: Introducing the PilGRIM: A processor for executing lazy functional languages. In: Hage, J., Morazán, M.T. (eds.) Implementation and Application of Functional Languages - 22nd International Symposium, IFL 2010, Alphen aan den Rijn, The Netherlands, September 1-3, 2010, Revised Selected Papers. Lecture Notes in Computer Science, vol. 6647, pp. 54–71. Springer (2010). https://doi.org/10.1007/978-3-642-24276-2_4, https://doi.org/10.1007/978-3-642-24276-2_4
6. Clarke, T.J.W., Gladstone, P., MacLean, C., Norman, A.C.: SKIM - the s, k, I reduction machine. In: Proceedings of the 1980 LISP Conference, Stanford, California, USA, August 25-27, 1980. pp. 128–135. ACM (1980). https://doi.org/10.1145/800087.802798, https://doi.org/10.1145/800087.802798
7. Fairbairn, J., Wray, S.: TIM: A simple, lazy abstract machine to execute supercombinatorics. In: Kahn, G. (ed.) Functional Programming Languages and Computer Architecture, Portland, Oregon, USA, September 14-16, 1987, Proceedings. Lecture Notes in Computer Science, vol. 274, pp. 34–45. Springer (1987). https://doi.org/10.1007/3-540-18317-5_3, https://doi.org/10.1007/3-540-18317-5_3
8. Hughes, R.J.M.: Super combinators: A new implementation method for applicative languages. In: Park, D.M.R., Friedman, D.P., Wise, D.S., Jr., G.L.S. (eds.) Proceedings of the 1982 ACM Symposium on LISP and Functional Programming, LFP 1980, Pittsburgh, PA, USA, August 15-18, 1982. pp. 1–10. ACM (1982). https://doi.org/10.1145/800068.802129, https://doi.org/10.1145/800068.802129
9. Johnsson, T.: Efficient compilation of lazy evaluation. In: Deusen, M.S.V., Graham, S.L. (eds.) Proceedings of the 1984 SIGPLAN Symposium on Compiler Construction, Montreal, Canada, June 17-22, 1984. pp. 58–69. ACM (1984). https://doi.org/10.1145/502874.502880, https://doi.org/10.1145/502874.502880
10. Jones, S.L.P.: The Implementation of Functional Programming Languages. Prentice-Hall (1987)
11. Jones, S.L.P., Clack, C.D., Salkild, J., Hardie, M.: GRIP - A high-performance architecture for parallel graph reduction. In: Kahn, G. (ed.) Functional Programming Languages and Computer Architecture, Portland, Oregon, USA, September 14-16, 1987, Proceedings. Lecture Notes in Computer Science, vol. 274, pp. 98–112. Springer (1987). https://doi.org/10.1007/3-540-18317-5_7, https://doi.org/10.1007/3-540-18317-5_7
12. Loidl, H.W., Hammond, K.: Graphing the future. In: International Workshop on the Implementation of Functional Languages. Norwich, England (Sep 1994)
13. Marlow, S.: Haskell 2010 Language Report. Tech. rep. (2010), https://www.haskell.org/onlinereport/haskell2010/
14. Naylor, M., Runciman, C.: The reduceron reconfigured and re-evaluated. J. Funct. Program. **22**(4-5), 574–613 (2012). https://doi.org/10.1017/S0956796812000214, https://doi.org/10.1017/S0956796812000214
15. Pope, J., Seger, C.H., Valter, H.: Higher-order hardware: Implementation and evaluation of the cephalopode graph reduction processor. In: 22nd ACM-IEEE International Symposium on Formal Methods and Models for System Design, MEMOCODE 2024, Raleigh, NC, USA, October 3-4, 2024. pp. 87–97. IEEE (2024). https://doi.org/10.1109/MEMOCODE63347.2024.00015, https://doi.org/10.1109/MEMOCODE63347.2024.00015
16. Ramsay, C., Stewart, R.J.: Heron: Modern hardware graph reduction. In: The 35th Symposium on Implementation and Application of Functional Languages, IFL 2023, Braga, Portugal, August 29-31, 2023. pp. 3:1–3:12. ACM (2023). https://doi.org/10.1145/3652561.3652564, https://doi.org/10.1145/3652561.3652564
17. Ramsay, C., Stewart, R.J.: Cloaca: A concurrent hardware garbage collector for non-strict functional languages. In: Vazou, N., Morris, J.G. (eds.) Proceedings of the 17th ACM SIGPLAN International Haskell Symposium, Haskell 2024, Milan, Italy, September 6-7, 2024. pp. 41–54. ACM (2024). https://doi.org/10.1145/3677999.3678277, https://doi.org/10.1145/3677999.3678277
18. Scheevel, M.: NORMA: A graph reduction processor. In: Scherlis, W.L., Williams, J.H., Gabriel, R.P. (eds.) Proceedings of the 1986 ACM Conference on LISP and Functional Programming, LFP 1986, Cambridge, Massachusetts, USA, August 4-6, 1986. pp. 212–219. ACM (1986). https://doi.org/10.1145/319838.319864, https://doi.org/10.1145/319838.319864

19. Schönfinkel, M.: Über die bausteine der mathematischen logik. Mathematische Annalen **92**, 305–316 (1924), http://eudml.org/doc/159074
20. Turner, D.A.: A new implementation technique for applicative languages. Softw. Pract. Exp. **9**(1), 31–49 (1979). https://doi.org/10.1002/SPE.4380090105, https://doi.org/10.1002/spe.4380090105
21. Watson, P., Watson, I.: Evaluating functional programs on the FLAGSHIP machine. In: Kahn, G. (ed.) Functional Programming Languages and Computer Architecture, Portland, Oregon, USA, September 14-16, 1987, Proceedings. Lecture Notes in Computer Science, vol. 274, pp. 80–97. Springer (1987). https://doi.org/10.1007/3-540-18317-5_6, https://doi.org/10.1007/3-540-18317-5_6