author2hash=ACfamily=Accetti, familyi=A., given=Cecil, giveni=C., hash=LPfamily=Liu, familyi=L., given=Peilin, giveni=P.,
author3hash=ACfamily=Accetti, familyi=A., given=Cecil, giveni=C., hash=YRfamily=Ying, familyi=Y., given=Rendong, giveni=R., hash=LPfamily=Liu, familyi=L., given=Peilin, giveni=P.,
author1hash=BHfamily=Barendregt, familyi=B., given=Henk, giveni=H.,
author1hash=BKfamily=Bimbò, familyi=B., given=Katalin, giveni=K.,
author2hash=BSfamily=Broda, familyi=B., given=Sabine, giveni=S., hash=DLfamily=Damas, familyi=D., given=Luis, giveni=L.,
author4hash=CTJWfamily=Clarke, familyi=C., given=Thomas J. W., giveni=T. J. W., hash=GPfamily=Gladstone, familyi=G., given=P., giveni=P., hash=MCfamily=MacLean, familyi=M., given=C., giveni=C., hash=NACfamily=Norman, familyi=N., given=Arthur C., giveni=A. C.,
author1hash=HRfamily=Hindley, familyi=H., given=R., giveni=R.,
author1hash=JJMfamily=Jannsen, familyi=J., given=Jan Martin, giveni=J. M.,
author3hash=JJMfamily=Jansen, familyi=J., given=Jan Martin, giveni=J. M., hash=KPfamily=Koopman, familyi=K., given=Pieter, giveni=P., hash=PRfamily=Plasmeijer, familyi=P., given=Rinus, giveni=R.,
author1hash=KRBfamily=Kieburtz, familyi=K., given=Richard B., giveni=R. B.,
author1hash=MRfamily=Milner, familyi=M., given=R., giveni=R.,
author1hash=MRfamily=Milner, familyi=M., given=Robin, giveni=R.,
author2hash=NMfamily=Naylor, familyi=N., given=Matthew, giveni=M., hash=RCfamily=Runciman, familyi=R., given=Colin, giveni=C.,
author2hash=NMfamily=Naylor, familyi=N., given=Matthew, giveni=M., hash=RCfamily=Runciman, familyi=R., given=Colin, giveni=C.,
author1hash=SRfamily=Smullyan, familyi=S., given=Raymond, giveni=R.,

Massimult: A Novel Parallel CPU Architecture Based on Combinator Reduction

Jürgen Nicklisch-Franken
juergen.nicklisch@symbolian.net

Ruslan Feizerakhmanov
me@russoul.me

***Abstract*—The Massimult project aims to design and implement an innovative CPU architecture based on combinator reduction with a novel combinator base and a new abstract machine. The evaluation of programs within this architecture is inherently highly parallel and localized, allowing for faster computation, reduced energy consumption, improved scalability, enhanced reliability, and increased resistance to attacks.**

In this paper, we introduce the machine language LambdaM, detail its compilation into KVY assembler code, and describe the abstract machine Matrima. The best part of Matrima is its ability to exploit inherent parallelism and locality in combinator reduction, leading to significantly faster computations with lower energy consumption, scalability across multiple processors, and enhanced security against various types of attacks. Matrima can be simulated as a software virtual machine and is intended for future hardware implementation.

I. Introduction

VON NEUMANN architecture relies on the sequential execution of operations. As current CPUs and computers are predominantly built on this architecture, considerable effort has been invested in achieving parallelism within a single core. Modern CPUs are equipped with multiple cores to facilitate parallel processing. However, this approach comes with the trade-off that it requires software specifically designed to leverage these multiple cores effectively.

Combinator reduction [**Arch**][**Graph**][**ParComb**] represents a fundamentally different approach to computation, providing intrinsic parallel execution without requiring special considerations from the programmer. In this model, a computation can be visualized as a tree, where all branches can evolve concurrently, driven by the leaves, which are composed of combinators and primitive operations. The tree dynamically grows and shrinks as the computation progresses, ultimately reaching completion when no further reductions are possible. However, this tree is effectively a graph, as expressions are shared whenever possible, guaranteed to produce the same result within a functional context.

Another key distinction between combinator reduction and the Von Neumann architecture is that, in graph reduction, both program and data are treated uniformly, whereas a Von Neumann architecture separates code and data into different memory regions and handles them differently. In the Von Neumann model, global reads and writes are the typical access patterns, necessitating complex cache hierarchies in modern CPUs to mitigate the inefficiencies associated with global memory access. In contrast, graph reduction operates through the local application of combinators, inherently reducing the need for such extensive caching mechanisms.

The Von Neumann architecture and combinator reduction are rooted in distinct theoretical models: the Von Neumann architecture is based on Turing machines, while Combinator Reduction is founded on Lambda Calculus [**Barend**] and its closely related counterpart, Combinatory Logic [**Bimbo**]. These underlying models are reflected in different programming paradigms. Imperative programming, which focuses on state changes over time, aligns with the Turing machine model. In contrast, functional programming, centered on the transformation of streams of information, aligns with the principles of Lambda Calculus.

Turing Machine	Lambda Calculus
Von Neumann Architecture	Massimult Architecture
Imperative Programming	Functional Programming

It is widely recognized that any program can be written in a purely functional style, as exemplified by languages like Haskell or Idris, in a purely imperative style, or, as is often the case, in a hybrid style that combines both paradigms. Traditionally, functional languages are compiled down to imperative assembly languages, reflecting the dominance of Von Neumann architecture in most hardware. However, it is equally feasible to compile imperative languages into a functional assembly language, such as **KVY** — section IV-A.

Because functional programs avoid side effects and mutable state, they are naturally more amenable to parallel execution, resulting in faster execution times and more efficient use of computational resources compared to their imperative counterparts.

Since combinator reduction is based on local operations, it allows for the selective activation of only the parts of a chip that are actively needed at a very fine-grained level. This approach can significantly reduce energy consumption compared to conventional CPUs, which often require larger portions of the chip to be active simultaneously. Additionally, the intrinsic parallelism of combinator reduction reduces the need to maximize clock frequency, further conserving energy.

The functional programming paradigm offers a significant advantage due to its strong theoretical foundations, which facilitate reasoning about program behavior. This advantage extends not only to human developers but also to software tools that can automatically analyze and verify code. Notably, dependently typed functional languages provide a powerful means of encoding software requirements directly within the

type system. This enables developers to formally prove that their software adheres to these requirements, paving the way for the development of bug-free, secure, and highly reliable software of unprecedented quality.

In this project, we combine deep research with strong engineering skills. We have successfully defined a new combinator code and developed a novel reduction mechanism for parallel and speculative evaluation. Our work includes the implementation of the Matrima virtual machine on a CPU, as well as the development of a **LambdaM** compiler and interpreter written in `Idris2`. Our next objective is to achieve competitive performance through GPU emulation and to better understand code optimization in the context of speculative evalation.

While the theoretical benefits of our approach are promising, the project will truly gain traction when we demonstrate that this methodology can achieve high effectiveness and speed in practical applications.

In this paper, we will first discuss the machine language **LambdaM** and its compilation into pure lambda calculus. We will then explore the **KVY** assembler code and, finally, describe the architecture and functioning of the Matrima machine. Readers only interested in novelties can start reading **??**

II. Related Work

The concept of graph reduction as a model for computation has its roots in the 1970s and 1980s, with early research exploring the potential of functional programming languages and graph reduction techniques to enable parallel computation. During this period, a variety of experimental architectures were developed to investigate these ideas, such as the SKIM (S, K, I Machine) [**SKIM**] and the G-machine [**GMachine**], which were specifically designed to execute combinatory logic and lambda calculus efficiently. These early efforts laid the groundwork for the exploration of parallelism in functional programming.

In more recent years, projects like the Reduceron [**Reduceron2008**] [**Reduceron2010**] and the HVM (Higher-Order Virtual Machine) have expanded on these foundational concepts to explore more efficient implementations of graph reduction for functional programming languages. The Reduceron, for instance, is a hardware accelerator that uses FPGA technology to execute graph reduction more efficiently, optimizing the execution of functional languages like Haskell by mapping combinator graph reductions directly to hardware operations. This approach significantly improves execution speed compared to traditional software implementations.

The HVM (Higher-Order Virtual Machine) represents a contemporary effort to optimize the performance of functional languages using an innovative runtime system. Unlike traditional virtual machines, HVM leverages a combination of beta-optimal graph reduction techniques and garbage-collection-free memory management to provide a high-performance environment for executing functional programs. HVM is designed to exploit modern hardware capabilities, aiming to deliver a scalable and energy-efficient runtime that can outperform conventional interpreters and virtual machines for functional languages.

Our project builds on these prior efforts by proposing a new combinator code and reduction machinery specifically designed for parallel and speculative evaluation. We aim to advance the state-of-the-art in functional programming and combinator reduction by leveraging GPU emulation and exploring novel hardware implementations. By drawing from both historical and contemporary work in the field, we seek to create a competitive and energy-efficient architecture that enhances the capabilities of functional programming languages.

III. Machine language and compilation

A. The Typed Machine Language LambdaM

LambdaM is a functional machine language specifically designed to facilitate the compilation of fully-featured functional languages, such as Haskell, Idris, or Agda. Unlike traditional imperative assembly languages, LambdaM is less technical in nature; instead, it serves as a highly abstract core language that retains the expressive power and functional characteristics of higher-level functional programming paradigms.

The primary goal of LambdaM is to provide a minimalistic yet expressive language that preserves the semantics of functional programming while allowing for efficient execution on current and future hardware. LambdaM is typed, ensuring that programs written in the language adhere to strict type rules, which aids in both correctness and optimization during compilation and execution. This typed approach also allows for advanced type-checking mechanisms that can prevent common runtime errors and enforce invariants at the machine level.

Technically, LambdaM is based on the Hindley-Milner typed lambda calculus [**Hindley**][**Milner**], enriched with features such as algebraic data types, `letrec` expressions, `case` expressions, `if-then-else` expressions, and pattern matching. Additionally, LambdaM includes primitive types, literal representations for certain types, primitive functions, and type annotations. The syntax is designed to be similar to functional languages such as Haskell, Idris, or Agda. Readers primarily interested in the novel contributions may choose to skip this section.

The use of a typed language in LambdaM is intentional, as it eliminates a wide range of potential run-time errors by enforcing type safety. The decision to adopt a restricted and straightforward Hindley-Milner type system provides the advantage of automatic type inference, which is important for a machine language.

A top-level binding in LambdaM defines either a data type or a function. Top-level data type definitions specify the structure and constructors of algebraic data types, providing a foundation for constructing complex data structures. Function bindings define the computational logic, consisting of function names, parameters, type annotations, and bodies composed of expressions and patterns.

1) Data Types: A data type definition in LambdaM begins with the `data` keyword, followed by an uppercase identifier that names the newly defined type. This identifier is optionally followed by a sequence of lowercase identifiers, which

represent type variables for parametric polymorphism. For example, the definition `List a` introduces a generic type `List`, which can be instantiated as `List Int` or `List String`, depending on the specific type used in place of `a`.

The use of type variables allows for the creation of highly versatile and reusable data structures, supporting a wide range of applications while maintaining type safety. This approach enables developers to define generic data types that can be used consistently across different contexts without sacrificing the benefits of a statically-typed system.

After the equality sign =, the definition proceeds with an uppercase type constructor, followed by a potentially empty sequence of type expressions. These type expressions can be uppercase type names, type variables, or function (arrow) types. This sequence effectively defines a record with types, where the record elements are accessed by their position rather than by name.

Furthermore, since the newly defined type name can be referenced within its own definition on the right-hand side, recursive types can be defined. This capability is essential for defining complex data structures such as linked lists, trees, and other recursive constructs that are fundamental in functional programming.

It is possible to define multiple constructors for a data type in LambdaM. When doing so, the vertical bar `|` is used as a separator between constructors. This allows instances of the data type to take on different forms, making the type more flexible and expressive. The combination of parametric polymorphism, records, unions (multiple constructors), and recursion forms what is known as an algebraic data type (ADT), a fundamental concept in the functional programming community.

Below are three example data type definitions demonstrating these concepts:

```
data List a = Nil | Cons a (List a)
data IORes a = MkIORes a World
data IO a = MkIO (World -> IORes a)
```

LambdaM includes built-in types that are typically constructed from literals. Additionally, it provides a set of Prelude types that are immutable and cannot be modified by the user. These Prelude types are fundamental to the language for several reasons: they may have literal representations, they are integral to language constructs, or they are employed by primitive functions, or a combination of these reasons.

Furthermore, LambdaM includes standard library types designed to offer efficient implementations of commonly used data structures, such as sequences and maps. These library types are optimized for performance and are essential for developing robust and efficient functional programs. For a detailed overview of these types and their implementations, see Appendix 1.

2) Functions: In LambdaM, function definitions begin with a lowercase name, followed by a potentially empty sequence of patterns, an equality sign, and the function's body. Patterns in function definitions serve as the mechanism for destructuring inputs and can take several forms: lowercase variable names for binding values, underscores _ for matching any value without binding it, or type constructors, each followed by an optional sequence of additional patterns. When the sequence of patterns following a constructor is not empty, it must be enclosed in parentheses along with the constructor to ensure proper grouping and clarity.

This pattern-matching capability allows for concise and expressive function definitions, facilitating the handling of complex data structures directly within the function signature. The combination of destructuring with algebraic data types enhances both readability and correctness in functional programs by enabling direct manipulation of data forms.

When type constructors are used, multiple lines can be utilized to pattern match on the input of a function. It is essential that all possible input cases are covered; otherwise, an error will be raised when loading the program due to incomplete pattern matching. Pattern matching follows a top-down approach where lines are evaluated in order from top to bottom, with the first matching pattern being selected.

The following example demonstrates a function that removes the first occurrence of an element from a list. Note that LambdaM does not support overloaded functions or typeclasses; therefore, the equality function (`eqFunc`) must be explicitly passed to this polymorphic function:

```
remove eqFunc ele (Cons hd tl) =
  if eqFunc ele hd
    then tl
    else Cons hd (remove eqFunc ele tl)
remove _ _ Nil = Nil
```

In this example, the `remove` function takes three arguments: an equality function `eqFunc`, the element to be removed `ele`, and a list. The function pattern matches on the list input. If the list is non-empty, it checks if the head element `hd` matches the element `ele` using `eqFunc`. If they are equal, the function returns the tail `tl`; otherwise, it recursively constructs a new list, retaining `hd` and removing `ele` from `tl`. The second line handles the base case of an empty list, returning `Nil`.

The terms of the language that constitute the right-hand side of function definitions in LambdaM are standard lambda calculus terms, including bound variables, function applications, and abstractions. These terms allow for the definition of functions in a manner consistent with the foundational principles of lambda calculus.

LambdaM also supports `let` expressions, which enable the definition of local functions that are scoped exclusively within the body term of the `let` and any other functions defined within the same `let` expression. This scoping mechanism facilitates modular function definitions and helps manage function visibility.

Any function in LambdaM can be recursive simply by invoking its own name within its body. Furthermore, functions defined at the top level and within `let` expressions can be mutually recursive, achieved by referencing each other by name within their respective definitions.

The language includes the conditional `if-then-else` construct, where the condition must be of type `Bool`, and the types of the expressions for both the true and false branches must be unifiable. This ensures type consistency across different branches of the conditional.

LambdaM also provides `case` expressions for pattern matching. In a `case` expression, the type of the match term must be unifiable with the types of all specified patterns, and the types of the resulting expressions for each pattern must also be unifiable. The patterns available in `case` expressions are identical to those used in function definitions. Additionally, all possible patterns must be covered; otherwise, the coverage checker will raise an error, ensuring exhaustive pattern matching and preventing runtime failures due to incomplete matches.

The formal language definition of LambdaM is provided below, illustrating the various terms (Tm) that make up the language:

$$
\begin{array}{lll}
Tm = \mathit{var} & & \text{Variable} \\
\quad = Tm_l\, Tm_r & & \text{Application} \\
\quad = \lambda\, \mathit{var}\,.\, Tm & & \text{Abstraction} \\
\quad = \textbf{let}\, \mathit{var}_1\ pat_{1,1}\ \ldots\ pat_{1,n} = Tm_1 & & \text{Letrec} \\
\qquad \ldots & & \\
\qquad \mathit{var}_m\ pat_{m,1}\ \ldots\ pat_{m,n} = Tm_m & & \\
\quad\ \textbf{in}\, Tm & & \\
\quad = \textbf{if}\, Tm_c\, \textbf{then}\, Tm_p\, \textbf{else}\, Tm_n & & \text{If-Then-Else} \\
\quad = \textbf{case}\, Tm\, \textbf{of} & & \text{Case} \\
\qquad Pat_1 \rightarrow Tm_1 & & \\
\qquad \ldots & & \\
\qquad Pat_n \rightarrow Tm_n & & \\
\quad = \mathit{TypeConstr} & & \text{Type Constructor} \\
\quad = \mathit{prim} & & \text{Primitive} \\
\quad = \mathit{literal} & & \text{Literal}
\end{array}
$$

Fig. 1. LambdaM Terms

Immediately preceding a function definition at the top level or within a `let` expression, a type annotation can be provided in the form `functionId : typeExpr`. Type annotations are particularly useful in cases involving literal constants, such as `maxSize = 3`, where the type checker may not be able to infer the type on its own.

At the beginning of a file, one or more import statements can be included. These statements take the form of `import StdLib`. LambdaM does not support namespaces, so when a simple import statement is used, all top-level names must be distinct to avoid naming conflicts, or you have to qualify the name with the module name, such as `StdLib.Map.Map` or `StdLib.Map.mapInsert`.

Finally, there must be a special function defined at the top level called `main`, which serves as the entry point of the program. The `main` function is mandatory for program execution and defines the initial action to be performed when the program starts.

B. Compilation to Pure Untyped Lambda Calculus

In this section, we provide a brief overview of the compilation process utilized by the LambdaM compiler, although the foundational concepts are well established within the functional programming research community. Our novel contributions begin in the subsequent sections.

The LambdaM language is compiled into an intermediate representation, referred to as **LambdaBase**. This intermediate language is based on the pure untyped lambda calculus, augmented with primitive operations and a special construct for recursion. This approach facilitates the efficient translation of LambdaM programs while preserving the core functional properties of the original source code.

When loading a source file, the compiler first parses only the import statements. It searches for all required files in the current directory, within the `Massimult/LambdaM` path, and in any directory specified by the `LAMBDAM` environment variable. If a required file cannot be found, or if a cyclic dependency is detected, the compiler throws an error. Otherwise, all files are successfully loaded and processed as if they were a single, unified file with all names fully expanded.

It is worth noting that future versions of the compiler may implement caching mechanisms to store type-checking and compilation results for source files, thereby avoiding redundant type-checking and compilation for files that have not changed.

When a program is loaded, it must first be syntactically correct; otherwise, a lexing error will be raised. Following this, the program must conform to the rules that define valid expressions within the language. If it does not, a parsing error will be generated. Next, all patterns in function definitions and `case` expressions must be exhaustive; failure to do so will result in a coverage checker error. Finally, the program must pass type checking to ensure that all expressions conform to the expected types; if not, a type error will be thrown.

1) Type Checking: Type checking in LambdaM ensures that a program is free from type-related errors, thereby guaranteeing that certain classes of runtime errors cannot occur. While the type checker does not guarantee program termination or prevent out-of-memory errors, it does provide assurances that the program adheres to the specified type constraints. This means that, aside from potential errors arising from primitive functions (such as division by zero) or explicit error handling within the program, the execution is free from type-related failures. Thus, type checking provides a foundational level of safety, ensuring that well-typed programs cannot ”go wrong” in terms of type consistency.

To achieve type safety, the type checker employs the Hindley-Milner type inference algorithm to automatically infer the types of all expressions in a program. The type system utilized strikes a balance between expressiveness and simplicity, providing types that are as expressive as possible without requiring explicit type annotations in the program code. This characteristic is particularly advantageous for a machine language, as it simplifies programming while maintaining robust type safety.

Consequently, a program written in LambdaM does not require explicit type annotations. The type checker infers the most general type for every expression, ensuring that all components fit together correctly, thereby preventing a wide range of potential runtime errors. A program that successfully typechecks is also guaranteed to compile without issues.

After loading a program into the LambdaM interpreter, the inferred types of top-level expressions can be displayed using the command `:display t` or its shorthand `:d t`. Additionally, to view the type of a specific expression, the `:type <expr>` command or its shorthand `:t <expr>` can be used.

2) Unused Code Removal: Although the type checker requires all loaded code to be consistent, the subsequent compilation step involves a dependency analysis that begins from the `main` function. This analysis identifies and removes any functions and data constructors that are not referenced, effectively eliminating unused code from the program. This process ensures that only the necessary code is included in the final compiled output, optimizing the program's size and performance.

3) Literal Replacement: In this step, literals are replaced with their internal representations. For example, the literal number `3` could represent different types, such as `Nat`, `BinNat`, `Int`, or `Float`. The type checker determines the appropriate type based on the context, and the literal is subsequently replaced by its corresponding internal representation. For instance, if the literal is determined to be of type `Nat`, the number `3` would be represented internally as `S (S (S Z))`. As previously mentioned, this process may necessitate a type annotation in cases where the type cannot be inferred, such as in a definition like `size = 3`.

4) Replacing Patterns and General Case Expressions with Simple Case Expressions: Complex patterns in LambdaM can appear in various contexts, such as top-level function definitions, `let` expressions, lambda expressions, and `case` expressions. These patterns are capable of matching data constructors to an arbitrary depth, allowing for expressive and powerful pattern matching capabilities.

In contrast, simple `case` expressions restrict pattern matching to a depth of 1. This means that they can only match the constructors of a single data type and must include all possible constructors of that data type as cases. The compilation process involves transforming complex patterns into equivalent simple `case` expressions, ensuring that each simple `case` matches only at the first level of the data type's constructors.

Below is an example illustrating a complex pattern and its transformation into a series of simple `case` expressions:

```
-- Complex pattern example
exampleFunc (Cons  a (Cons b bs)) = result
exampleFunc _ = default

-- Transformed to simple case expressions
exampleFunc xs = case xs of
  Cons y ys => case ys of
    Cons z zs => result
    Nil => default
  Nil => default
```

While the concept of transforming complex patterns into simple `case` expressions is straightforward, devising an efficient algorithm for this transformation is considerably more complex. For a detailed discussion of the algorithm and its optimization strategies, we refer the reader to the existing literature [**Patterns**]. With this transformation, the code is now prepared for the subsequent step, where data types and simple `case` expressions are further converted into functions.

5) Encoding Data Types: Data types in LambdaM are encoded using the Scott encoding [**Jansen2013**]. In this encoding scheme, each data constructor and simple `case` expression is transformed into an ordinary function. This approach allows for a uniform treatment of data types and case expressions as first-class functions within the language.

Now we show how a simple enumeration (or sum) type can be transformed. For example, consider the following data type definition for a Boolean type and a function that uses a `case` expression to handle Boolean values:

```
data Bool = False | True
cond b = case b of
           False => S Z
           True  => Z
```

Using the Scott encoding, each data constructor and the `case` expression are transformed into ordinary functions. The Boolean data type Bool and the function cond would be encoded as follows:

```
False = \ f1 f2 . f1
True  = \ f1 f2 . f2
cond  = \ b . b (S Z) Z
```

In this transformation:

- The constructor False is encoded as a function that takes two arguments and returns the first argument. This corresponds to the behavior of selecting the first branch in a `case` expression when the value is False.
- The constructor True is encoded as a function that takes two arguments and returns the second argument. This reflects the behavior of selecting the second branch in a `case` expression when the value is True.
- The function cond is transformed into a function that applies the Boolean argument b to two possible outcomes, S Z and Z, effectively encoding the `case` expression as a function application. This transformation demonstrates how conditional branching is handled in a functional setting through the use of first-class functions.

For a simple record (or product) type, the transformation using Scott encoding proceeds as follows:

```
data Tuple a b = MkTuple a b
fst (MkTuple a b) = a
snd (MkTuple a b) = b
=>
MkTuple = \ a b f . f a b
fst = \ f . f (\ a b . a)
snd = \ f . f (\ a b . b)
```

In this transformation:

- The constructor MkTuple is encoded as a function that takes two arguments, a and b, and a function f, and applies f to a and b. This encoding effectively turns the tuple constructor into a higher-order function.
- The function fst, which retrieves the first element of the tuple, is transformed into a function that takes a function f and applies f to a lambda expression that returns the first element a.

- Similarly, the function snd, which retrieves the second element of the tuple, is encoded as a function that takes a function f and applies f to a lambda expression that returns the second element b.

This transformation demonstrates how product types, like tuples, can be represented in a purely functional setting using functions. By encoding tuples as functions, LambdaM maintains uniformity in its treatment of data types, supporting a higher level of abstraction and functional manipulation.

The following example defines a `List` type, which exhibits both an enumeration (sum) type and a record (product) type. The `List` type is an enumeration type because it can be either Nil (representing an empty list) or Cons (representing a non-empty list). It is also a record type because the Cons constructor includes both an element and a list of elements as its members:

```
data List a = (Cons a (List a)) | Nil
head (Cons a rest) = Just a
head Nil = Nothing
tail (Cons a rest) = Just rest
tail Nil = Nothing
=>
Cons  = \ a r f1 f2 . f1 a r
Nil   = \ f1 f2 . f2
head  = \ l . l (\ a r . Just a) Nothing
tail  = \ l . l (\ a r . Just r) Nothing
```

In this transformation:

- The constructor Cons is encoded as a function that takes an element a, a list r (the rest of the list), and two additional function arguments, f1 and f2. It applies f1 to a and r, effectively representing the non-empty list case.
- The constructor Nil is represented as a function that takes two function arguments, f1 and f2, and returns f2, corresponding to the empty list case.
- The function head retrieves the first element of the list. It is transformed into a function that applies the list l to two lambdas: one that returns Just a when the list is non-empty, and Nothing when it is empty.
- Similarly, the function tail retrieves the rest of the list (excluding the first element). It is encoded as a function that applies the list l to two lambdas: one that returns Just r for a non-empty list and Nothing for an empty list.

This transformation illustrates how a combined enumeration and record type, such as a `List`, can be encoded in a purely functional style using Scott encoding. By converting data constructors and functions into first-class functions, LambdaM maintains a consistent and flexible representation for complex data structures.

This encoding closely aligns with the conventional handling of data types while maintaining efficiency. Scott encoding provides a functional representation of data constructors and case expressions, transforming them into first-class functions. The general transformation for a data type and its corresponding `case` expression is formalized as follows:

$$
\begin{aligned}
&\text{data}\, N \;=\; C_1\, v_{1,1} \ldots v_{1,n} \mid \ldots \mid C_m\, v_{m,1} \ldots v_{m,n} \\
&\qquad\qquad \Downarrow \\
&C_1 = \lambda\, v_{1,1} \ldots v_{1,n}\, f_1 \ldots f_m \,.\, f_1\, v_{1,1} \ldots v_{1,n} \\
&\qquad\qquad \ldots \\
&C_m = \lambda\, v_{m,1} \ldots v_{m,n}\, f_1 \ldots f_m \,.\, f_m\, v_{m,1} \ldots v_{m,n} \\
\\
&\text{case}\, x\, \text{of} \\
&\quad C_1\, v_{1,1} \ldots v_{1,n} \Rightarrow b_1 \\
&\qquad\qquad \ldots \\
&\quad C_m\, v_{m,1} \ldots v_{m,n} \Rightarrow b_m \\
&\qquad\qquad \Downarrow \\
&\lambda\, f \,.\, f\, (\lambda\, v_{1,1} \ldots v_{1,n} \,.\, b_1) \\
&\qquad\qquad \ldots \\
&\qquad\qquad (\lambda\, v_{m,1} \ldots v_{m,n} \,.\, b_m)
\end{aligned}
$$

Fig. 2. Scott Encoding

In this general transformation:

- Each data constructor C_i is transformed into a function that takes its arguments $v_{i,1}, \ldots, v_{i,n}$ and a series of functions $f_1, \ldots, f_m$. The constructor function applies the corresponding function f_i to its arguments, encoding the selection logic of case expressions.
- The `case` expression is transformed into a lambda function that takes a function f and applies it to a series of lambda expressions. Each lambda expression represents a branch of the original `case` expression, effectively encapsulating the branching logic in a functional form.

This transformation leverages the power of higher-order functions to represent both data structures and control flow, demonstrating the expressive capabilities of the Scott encoding in a functional programming context.

6) Replacing `let` Expressions with Lambdas: Replacing `let` expressions with lambda expressions is straightforward. A `let` expression of the form `let x = e in t` is translated into the equivalent lambda expression `(\x. t) e`. This transformation leverages lambda abstraction to achieve the same effect as a local binding.

Consequently, in `let` expressions containing multiple definitions, only those definitions declared earlier can be referenced in subsequent expressions. This restriction also applies to top-level function definitions, which can be viewed as a `let` expression without an explicitly written `let`. To enforce this order, the compiler performs a dependency analysis and arranges the defining terms accordingly.

However, when mutually recursive functions are present, additional considerations are required. The compilation of such functions is addressed in the next subsection.

7) Recursion: In LambdaM, recursion is supported both for single functions and for multiple functions that reference each other. A function is considered recursive if it refers to its own name within its definition. Similarly, functions are

considered mutually recursive if two or more functions refer to each other. Both forms of recursion are fully supported in LambdaM, allowing for the definition of complex recursive algorithms.

We begin by addressing simple recursion. There are multiple strategies for compiling recursive functions, such as introducing cycles in the evaluation graph or passing the function itself as an extra argument to enable self-reference. In our approach, we utilize a fixpoint combinator. Specifically, we employ an applicative order fixpoint combinator, which we denote as `Y`. The `Y` combinator is defined as `Y f x = f (Y f) x`, which allows the function `f` to recursively call itself in a purely functional manner.

The factorial function is used as an illustrative example to demonstrate the compilation process through multiple transformations. The first transformation involves encoding data types, patterns, and `case` expressions, as previously described. The second transformation demonstrates how recursion is handled using the `Y` combinator.

```
-- Original recursive factorial function
fac Z = S Z
fac (S n) = mul (fac n) (S n)

-- Step 1: Scott Encoding
fac = \ sn .
    sn (\ n . mul (fac n) (S n)) (S Z)

-- Step 2: Using the fixpoint combinator
fac = Y (\ fac' sn .
    sn (\ n . mul (fac' n) (S n)) (S Z))
```

In these transformations:

- **Original Function Definition:** The factorial function, fac, is defined recursively with two cases: one for the base case (Z) and one for the recursive case (S n).
- **Step 1: Encoding:** The original recursive function is transformed to a lambda expression that encodes pattern matching using function application. The function fac takes an argument sn representing the structure of the natural number (either Z or S n) and applies appropriate logic for each case.
- **Step 2: Applying the Fixpoint Combinator:** To handle recursion, we introduce the `Y` combinator. This transformation replaces the direct recursive call to fac with a reference to the function itself (fac') using the fixpoint combinator. This allows recursion to be expressed without explicit self-reference, enabling the factorial calculation to proceed in a purely functional context.

These steps illustrate the transformation of a recursive function into a form that is compatible with the functional paradigm of LambdaM. By utilizing the `Y` combinator, we ensure that recursion is managed efficiently within the functional framework, preserving both the expressiveness and correctness of the original program.

For mutually recursive functions, we extend the use of the `Y` combinator by passing a tuple of functions rather than a single function. This approach enables the handling of multiple functions that recursively reference each other, allowing them to be defined together in a cohesive manner. To illustrate this, we consider the classic example of mutually recursive functions, `even` and `odd`. Although this example is straightforward and primarily pedagogical, it effectively demonstrates the transformation process involved in managing mutual recursion using the `Y` combinator:

```
-- Original mutually recursive functions
even Z = True
even (S n) = odd n
odd Z = False
odd (S n) = even n

-- Step 1: Scott Encoding
even = \ sn .
  sn (\ n . odd n) True
odd = \ sn .
  sn (\ n . even n) False

-- Step 2: Handling mutual Recursion
evenodd = Y (\ eo .
  (Tuple (\ sn .
    sn (\ n . (snd eo) n) True)
   (\ sn .
    sn (\ n . (fst eo) n) False)))

even = fst evenodd
odd  = snd evenodd
```

In this transformation:

- **Original Function Definitions:** The functions even and odd are defined with two cases each: one for the base case (Z) and another for the recursive case (S n).
- **Step 1: Encoding:** The original recursive functions are transformed into lambda expressions that handle pattern matching through function application. Here, even and odd are encoded to apply the appropriate logic based on the structure of the input.
- **Step 2: Using the Fixpoint Combinator:** To handle mutual recursion, we construct a tuple of functions (`Tuple e o`) representing even and odd. The `Y` combinator is applied to this tuple, allowing both functions to recursively reference each other through the tuple. The functions fst and snd are then used to extract the individual functions from the tuple.

8) LambdaBase Language: With these transformations, the compilation of LambdaM to a representation based on pure untyped lambda calculus, augmented with primitives and recursion, is complete. The resulting intermediate language, which we call **LambdaBase**, consists of a minimal set of constructs that capture the essence of functional computation while supporting recursion and primitive operations. Below is the compact definition of the **LambdaBase** language:

$$
\begin{array}{ll}
Tm = \mathit{var} & \text{Bound Variable} \\
\quad = Tm_l\, Tm_r & \text{Application} \\
\quad = \lambda\, \mathit{var}\, .\, Tm & \text{Abstraction} \\
\quad = \underline{Y} & \text{Recursion via Fixpoint Combinator} \\
\quad = \mathit{prim}\ldots & \text{Primitives}
\end{array}
$$

Fig. 3. LambdaBase Language

In this definition:

- **Bound Variable (*var*):** Represents a variable that is bound within a lambda abstraction.
- **Application ($Tm_l\, Tm_r$):** Represents the application of one term to another, capturing function application in the calculus.
- **Abstraction ($\lambda\, \mathit{var}\, .\, Tm$):** Represents the definition of an anonymous function with a bound variable.
- **Recursion via Fixpoint Combinator ($\underline{Y}$):** Introduces recursion into the language by using a fixpoint combinator, allowing for the definition of recursive functions.
- **Primitives (*prim* ...):** Represents a set of primitive operations, such as arithmetic functions, that are treated as basic operations in the language.

The **LambdaBase** language provides a foundational core for further compilation to combinatory machine code, while preserving the essential features of functional computation.

9) And Back Again: When using LambdaM as an interpreter, the compilation process is identical to that of using LambdaM as a compiler. However, in an interpretive context, the output is expected to be presented in the form of LambdaM expressions. Reconstructing LambdaM terms from **LambdaBase** terms is feasible, but only when the type of the expression is known.

The reason for this constraint is that different constructors in LambdaM can be compiled into indistinguishable functions in **LambdaBase**. Consequently, without type information, it would be impossible to accurately reconstruct the original constructors. Fortunately, for every expression interpreted in LambdaM, the result type is always known, enabling the correct reconstruction of both constructors and literals. Functions are also displayed correctly, although their presentation may not always be optimal at this stage.

This ensures that, despite the compilation to a lower-level representation, the interpreter can effectively present results in the original, more readable LambdaM format, preserving both the semantic clarity and usability of the interpreted output.

IV. KVY Combinator Code

The simplest combinator base requires only two combinators, `K` and `S`, as demonstrated by Smullyan [**Smullyan**]. The reduction rules for these combinators are defined as follows:

$$
\begin{aligned}
K\, x\, y &\Rightarrow x \\
S\, x\, y\, z &\Rightarrow x\, z\, (y\, z)
\end{aligned}
$$

Fig. 4. KS reduction rules

A computer that implements only the `K` and `S` combinators with the aforementioned reduction rules is computationally universal, or Turing complete. This means it possesses the capability to perform any computation that can be executed by more complex computing machines. This result is remarkable, highlighting the power of a system that is even simpler than the Lambda calculus.

Despite their simplicity, the `K` and `S` combinators provide a sufficient foundation for constructing any computable function. This minimalistic approach not only facilitates a deeper understanding of the fundamental principles of computation but also allows for optimizations that can exploit the inherent parallelism in combinator-based systems. Such properties make them highly suitable for theoretical investigations into the nature of computation and for practical applications where simplicity and elegance are desired.

The equivalence in expressiveness between combinators and the Lambda calculus can be demonstrated by defining a transformation from one formalism to the other. The transformation from Lambda calculus to combinators is known as *abstraction elimination*, with its core mechanism referred to as *bracket abstraction*. This transformation can be defined as follows, where the transformation is indicated by square brackets:

$$
\begin{aligned}
\langle Tm \rangle_x &\mapsto K\, Tm \quad \text{if}\, x \notin Tm \\
\langle x \rangle_x &\mapsto S\, K\, K \\
\langle Tm_l\, Tm_r \rangle_x &\mapsto S\, \langle Tm_l \rangle_x\, \langle Tm_r \rangle_x
\end{aligned}
$$

$$
\begin{aligned}
[\lambda\, x\, .\, Tm] &\mapsto \langle [Tm] \rangle_x \\
[x] &\mapsto x \\
[Tm_l\, Tm_r] &\mapsto [Tm_l]\, [Tm_r] \\
[Y] &\mapsto Y
\end{aligned}
$$

Fig. 5. KS Abstraction Elimination

The reverse transformation is straightforward and involves the application of the reduction rules defined for each combinator.

A notable issue with using only the `K` and `S` combinators is the significant increase in the number of combinators compared to the number of lambda expressions, leading to a proliferation of fine-grained operations, which result in inefficient computation performance. Enhancing the combinator base by including additional combinators greatly improves efficiency. A more effective base includes the combinators `I`, `K`, `S`, `B`, and `C`, each with its respective reduction rules:

$$
\begin{aligned}
I\,x &\Rightarrow x \\
K\,x\,y &\Rightarrow x \\
S\,x\,y\,z &\Rightarrow x\,z\,(y\,z) \\
B\,x\,y\,z &\Rightarrow x\,(y\,z) \\
C\,x\,y\,z &\Rightarrow x\,z\,y
\end{aligned}
$$

Fig. 6. IKSBC reduction rules

Early implementations of functional languages that leveraged combinatory logic relied on a fixed set of pre-defined combinators. To enhance code compactness and minimize the number of fine-grained reductions, these implementations introduced additional combinators, e.g. Turner. These combinators served to minimize the growth of the number of combinators with respect to the number of lambda expressions. Our approach advances as high as mapping one lambda to exactly one combinator.

A. V: A Family of Combinators Indexed by a Multipath

The concept is to introduce a family of combinators, each indexed by a multi-path within a binary tree, which can be interpreted as a binary tree lacking any content. These combinators operate by inserting an element into the specific positions designated by the multi-path within the tree structure. Due to the nature of the multi-path, the element can be inserted one or more times at various locations, thereby allowing for flexible and dynamic reconstruction of expressions during computation.

This family of combinators, which we denote as **V**, specifies the positions within a tree where an element (the last argument) will be inserted. The multi-path is described using the following elements: $<$ for *Left*, $>$ for *Right*, and $\langle , \rangle$ for *Bifurcation*.

$$
\begin{aligned}
Path &= < \; Path \\
&= > \; Path \\
&= \langle Path_l,\, Path_r \rangle \\
&= \varnothing
\end{aligned}
$$

We refer to the count of left $<$ and right $>$ elements as the *degree* of a multi-path. Thus, the multi-path is effectively characterized by its degree, which represents the number of times each directional element appears within it.

To illustrate the concept of multi-paths and their degrees, consider the following examples:

- **Example 1:** Multi-path $<><$
  The multi-path $<><$ has a degree of 3, 2 for $(<)$ and 1 for $(>)$.
- **Example 2:** Multi-path $\langle\langle <, > \rangle, < \rangle$
  This multi-path has a degree of 3, 2 for $<$ and 1 for $>$. The bifurcation $\langle l, r \rangle$ indicates a branching, where the degree counts the total occurrences of $<$ and $>$ including those in the branches.

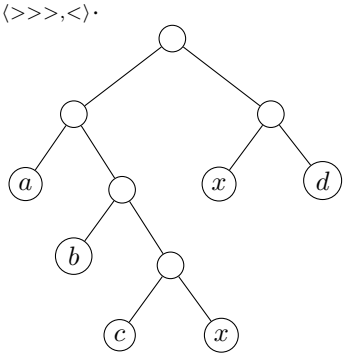
The intuition behind this family of combinators lies in its connection to the process of abstraction, particularly in the context of $(\lambda x.\, t)$, which abstracts the variable x from all its occurrences in a term t. During the compilation process—specifically, in the abstraction elimination phase—each lambda abstraction is substituted by a corresponding combinator from this family. When this combinator is applied to a term from which all instances of x have been removed, along with a specific value, it effectively reinserts that value into all the positions where x was originally located prior to the abstraction.

The implementation of the **V** combinator presents a particular challenge, specifically in the layout of the tree argument. A naive approach would involve placing the tree as the first argument and the object to be inserted as the second argument of the **V** combinator. The expected result would be a tree where the object is inserted at all positions indicated by the multi-path directed by the **V** combinator.

However, such a combinator would not exhibit the optimal parallelism properties that standard combinators possess. In particular, it would require a more sequential evaluation, reducing the potential for simultaneous execution across multiple parts of the expression. This lack of inherent parallelism would limit the performance and scalability that our architecture is designed to achieve.

To solve this problem the tree gets laid out at the spine guided by the multipath. The degree of the multi-path equals the number of arguments for the tree. Consequently, the arity of the **V** combinator is equal to the degree of the multi-path plus one (the additional argument for the element to be inserted).

For example, consider the lambda expression $\lambda x.\, a\,(b\,(c\,x))\,(x\,d)$. This expression would compile to the **V** combinator with a multi-path representation of $\mathbf{V}_{\langle >>>, < \rangle}$.



The compiler arranges the tree along the spine by following the specified path and treating the opposite side of the path as a spine element. For any bifurcation, the general approach is to traverse the left side first, followed by the right. This realized layout of arguments we call the residual.

In the given example, to write this tree as argument, we follow the path while we collect the arguments for the spine of the tree. Looking at the **V** combinator path we have an initial split, and we proceed down the left-hand branch, because we always do it first. Then we encounter a and right, then b and right, and finally c and right. So the current residual looks: $a\ b\ c$.

But we have to do the right side as well, where d is encountered. So the spine residual is: $a\ b\ c\ d$ and consequently,

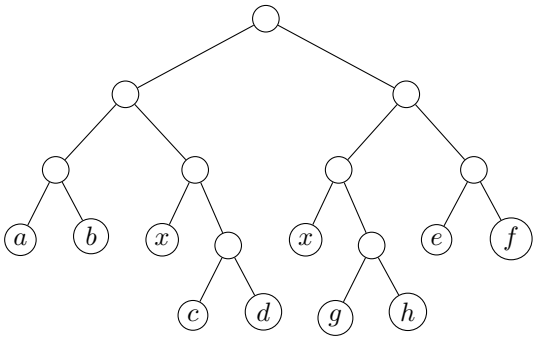
the compiler translates the expression $\lambda x.\, a\,(b\,(c\,x))\,(x\,d)$ into the combinator $V_{\langle >>>,<\rangle}\, abcdx$, which evaluates to $a\,(b\,(c\,x))\,(x\,d)$.

We observe that the tree, into which the elements are to be inserted, is rearranged by the compiler such that the parts that will change at reduction are laid out along the spine. This provides the **V** combinator with optimal conditions for parallelism.

The **V** combinator, due to its complexity, may not exhibit a constant execution time. However, the architecture presented in the following section is designed to handle such variability efficiently. Unlike traditional reduction models that require one reduction per cycle, our machine is capable of accommodating the non-constant execution times of the **V** combinator without compromising performance.

Let us consider another example to further demonstrate the compilation process. Take the lambda expression:

$$\lambda x.(a\,b\,(x\,(c\,d)))\,(x\,(g\,h)\,(e\,f))$$



To determine the multi-path for the expression $(a\,b\,(x\,(c\,d)))\,(x\,(g\,h)\,(e\,f))$, we analyze the positions where the variable x is applied. As observed, the multi-path for this expression is $\langle ><, << \rangle$.

In the given example, the initial steps follow the multi-path to arrange the residual. The first path, $><$, proceeds by moving right and then left: first encountering $(a\,b)$, followed by $(c\,d)$, and finally reaching x. The second path, $<<$, moves left twice encountering the subtrees $(e\,f)$ and $(g\,h)$.

Consequently, the compiler translates the expression $x.(a\,b\,(x\,(c\,d)))\,(x\,(g\,h)\,(e\,f))$ into the combinator $V_{\langle >\!<,<<\rangle}\,(a\,b)(c\,d)(e\,f)(g\,h)x$. This transformation ensures that the expression maintains its original structure, allowing the combinator to evaluate to the intended expression $(a\,b\,(x\,(c\,d)))\,(x\,(g\,h)\,(e\,f))$.

B. The KVY Assembler Code

The combinator base **KVY** is designed to provide a minimal yet powerful set of combinators that can represent any computation expressible in the lambda calculus. The base consists of three fundamental combinators: **K**, **V**, and **Y**.

As an example we take a program to calculate "one plus one". The numbers are represented as natural numbers. The program consists of the definition of the type of Nat, the definition of the function plus and the computation **plus 1 1**:

```
data Nat = S Nat  | Z
plus Z n = n
plus (S m) n = S (plus m n)
main = plus 1 1
```

Compiled to KVY assembler, it looks this way:

```
V<{{<>>>>>,},} (K V) V<< V<{>>,>}
Y (V<> V{><>,}) V<<> V><> (V>< K)
```

KVY has the following abstract reduction rules:

$$\begin{aligned}
K\,x\,y &\Rightarrow x\\
V\,\varnothing\,w &\Rightarrow w\\
V\,(<\,Path)\,x\,\overline{x}\,w &\Rightarrow (V\,Path\,\overline{x}\,w)\,x\\
V\,(>\,Path)\,x\,\overline{x}\,w &\Rightarrow x\,(V\,Path\,\overline{x}\,w)\\
V\,\{Path_l,\,Path_r\}\,\overline{x}_l\,\overline{x}_r\,w &\Rightarrow (V\,Path_l\,\overline{x}_l\,w)\,(V\,Path_r\,\overline{x}_r\,w)\\
Y\,f\,x &\Rightarrow f\,(Y\,f)\,x
\end{aligned}$$

And this is the abstraction elimination algorithm for the KVY language:

$$\begin{aligned}
\langle x\rangle_x &\mapsto \varnothing\\
\langle Tm_l\,Tm_r\rangle_x &\mapsto < \langle Tm_l\rangle_x \quad \text{if } x \notin Tm_r\\
\langle Tm_l\,Tm_r\rangle_x &\mapsto > \langle Tm_r\rangle_x \quad \text{if } x \notin Tm_l\\
\langle Tm_l\,Tm_r\rangle_x &\mapsto \{\langle Tm_l\rangle_x\,,\,\langle Tm_r\rangle_x\}
\end{aligned}$$

$$\begin{aligned}
(x)_x &\mapsto \cdot\\
(Tm_l\,Tm_r)_x &\mapsto Tm_r\,(Tm_l)_x \quad \text{if } x \notin Tm_r\\
(Tm_l\,Tm_r)_x &\mapsto Tm_l\,(Tm_r)_x \quad \text{if } x \notin Tm_l\\
(Tm_l\,Tm_r)_x &\mapsto (Tm_l)_x\,(Tm_r)_x
\end{aligned}$$

$$\begin{aligned}
[\lambda\,x\,.\,Tm] &\mapsto K\,[Tm] \quad \text{if } x \notin Tm\\
[\lambda\,x\,.\,Tm] &\mapsto V\,\langle[Tm]\rangle_x\,([Tm])_x\\
[x] &\mapsto x\\
[Tm_l\,Tm_r] &\mapsto [Tm_l]\,[Tm_r]\\
[Y] &\mapsto Y
\end{aligned}$$

Fig. 7. KVY Abstraction Elimination

The original work, on which the V combinator is based on, is from **[ClK]**. Together, these combinators form a robust foundation for computation. The **KVY** base is minimal yet expressive, providing the necessary tools to represent complex computational constructs while ensuring that the resulting expressions are optimized for parallel evaluation.

V. The Matrima Machine

The combinator code outlined in the last section necessitates a dedicated machine for its evaluation. In this section, we describe the architecture of such a machine. The ultimate

objective of this project is to develop this machine as a specialized silicon chip, meticulously designed to execute combinator code with high performance. Although this goal has yet to be realized, we are currently conducting experiments using virtual machines that simulate the target architecture on contemporary CPUs. Additionally, we are exploring further steps to simulate the target architecture on GPUs and FPGAs, which may provide valuable insights into optimizing the machine's performance before its physical implementation.

The architecture must be meticulously designed to leverage the inherent advantages of functional programming in general and combinator code in particular. The success of this project hinges on the architecture's ability to deliver high-speed performance for functional programs, coupled with greater energy efficiency compared to current CPUs. Furthermore, the architecture must facilitate the scalable deployment of these chips, ensuring that the system can be expanded easily to meet growing computational demands. Achieving these goals could position the Matrima Machine as a significant disruptive force within the current IT landscape, which is predominantly based on the Von Neumann architecture

A. The Basics of the Matrima Machine

At its core, the Matrima Machine represents a significant departure from traditional Von Neumann-based designs, which are characterized by their reliance on sequential instruction execution and shared memory models. In contrast, the Matrima Machine adopts a non-sequential, stateless computational model, which is inherently better aligned with the parallelism and immutability that are foundational to functional programming. This shift enables the architecture to fully exploit the advantages of functional code, particularly in terms of parallel processing efficiency and scalability.

The evaluation of combinator code necessitates two fundamental activities. The first activity involves determining whether an expression can be reduced, assessing whether the expression is in head normal form or in normal form. The second activity is the reduction process itself, wherein the expression is systematically simplified according to the rules of combinator logic.

One of the most significant advantages of combinator code lies in its inherent properties regarding parallelism. Specifically, any expression within combinator code can be reduced in parallel without any conflicts, which starkly contrasts with the sequential nature of the Von Neumann architecture. This characteristic of combinator code is the cornerstone of the Massimult architecture, and the design of the actual machine is meticulously crafted to exploit this advantage to its fullest extent. By maximizing parallel processing capabilities, the Matrima machine seeks to achieve superior performance and efficiency, setting it apart from traditional computational models.

The primary concept behind the Matrima machine is to assign a separate thread to every expression and subexpression, enabling all these threads to operate in parallel. Each thread is responsible for evaluating whether its respective expression can be reduced; if reduction is possible, the thread will carry out the reduction. To determine the reducibility of an expression, a thread needs only to inspect the relevant subexpressions.

As reductions occur, new expressions are generated, and corresponding threads are initiated to evaluate them. These threads also check whether the expression has reached head normal form, in which case it will not be further evaluated, or whether it has achieved normal form, in which case the expression, along with all its subexpressions, will no longer undergo any reductions.

Once an expression reaches normal form, no further threads are required for that expression. If the root cell of the computation reaches normal form, the overall computation is complete, and the result is ready for use. This approach ensures that the architecture efficiently leverages the parallelism inherent in combinator code, driving optimal performance.

While this concept of code evaluation is straightforward and simple in principle, it necessitates careful consideration of its implications. In this architecture, we are dealing with an extraordinarily large number of parallel threads. Each of these threads operates independently, without any communication or synchronization with other threads, yet collectively they contribute to producing a single correct result.

However, it is important to note that the order of these reductions can vary from one execution to another. As a result, a computation in this model does not consist of a predetermined sequence of operations; rather, it is the sum of all possible reduction orders, with a specific order being selected during each individual run. Consequently, when executing the identical program with identical inputs, two separate runs are highly likely to diverge in both the number of reductions performed and the amount of memory utilized. This variability introduces a degree of nondeterminism in resource usage, which must be carefully managed to ensure optimal performance and reliability.

A second, and perhaps even more unexpected, observation is that at any given moment during program execution, the global expression on which the threads operate may become inconsistent. This arises from the lack of synchronization, allowing reads and writes to occur in any arbitrary order. In our prototype implementation, we ultimately rely on operating system-level threads, which can be paused and resumed at times largely beyond our control, leading to certain writes being executed much later than anticipated. Given the absence of synchronization, we must acknowledge and accept this inherent inconsistency.

We anticipate that a similar situation will arise in the silicon-based implementation, as the architecture is deliberately designed to avoid synchronization or communication between threads, which would otherwise introduce latency and slow down the machine. This design choice, while potentially leading to temporary inconsistencies, is crucial for maintaining the high-speed performance and parallel efficiency that the Matrima Machine seeks to achieve.

This phenomenon of unpredictable timing manifests in two critical aspects of our prototype implementation. We employ simple reference counting for memory recycling. However, due to the asynchronous nature of the system, increments and

decrements to reference counts can be written in any arbitrary order. This leads to the peculiar situation where a cell with a reference count of zero might later be referenced again, and in some cases, we even encounter negative reference counts.

We will revisit the issue of reference counting in more detail in the section dedicated to memory recycling. However, we can already anticipate that, as a consequence of this phenomenon, it is necessary to halt all reductions before initiating the recycling of cells. This precaution is essential to prevent the erroneous reclamation of memory that may still be in use, thereby ensuring the integrity of the computation.

The second unexpected phenomenon we encountered was related to normal forms. Initially, we assumed that when both the left and right subexpressions of any given expression are in normal form, and the expression itself cannot be further reduced, the expression must also be in normal form. However, due to the unpredictable timing of operations, we observed cases where reductions were written after the expression had already been tagged as being in normal form. This discrepancy led to inconsistencies in our evaluations.

To resolve this issue, we implemented an additional verification step, ensuring that the expression must also be in head normal form before it is conclusively tagged as being in normal form. This additional check mitigates the effects of timing unpredictability, thereby maintaining the accuracy and consistency of our normal form evaluations.

Despite the timing unpredictability inherent in our system, it remains true that the data is consistent at any moment when the evaluation is paused and all changes have been fully written. Additionally, it is important to emphasize that, regardless of the variations in timing, the correct result will always be computed. This consistency is a critical property of the Matrima Machine, ensuring that, even in the face of asynchronous operations and non-deterministic execution paths, the integrity of the computational process is preserved and reliable outcomes are guaranteed.

To summarize our approach: A vast number of threads operate in parallel, without any form of communication, synchronization, locking, or waiting on one another, yet they collaborate seamlessly to achieve a unified outcome. This is a rather remarkable achievement, demonstrating the robustness of the Matrima Machine's design. The ability to maintain coherence and correctness in the absence of traditional coordination mechanisms is both surprising and a testament to the power and efficiency of this parallel processing model.

However, there is one critical point of concurrency that remains: the allocation of fresh memory, which is a shared resource among all threads. To maintain the efficiency of the system, it is essential that memory allocation be implemented in such a way that it does not become a bottleneck. Specifically, we must avoid a scenario where there is a single point of allocation for all cells, which could result in threads being blocked while waiting for others to complete their allocations. Instead, memory allocation must be distributed or managed in a manner that allows threads to allocate memory independently, without being impeded by the activities of other threads.

Finally, in our prototype implementation, we have found it necessary to use atomic reads and writes of 128-bit chunks. This measure is essential to prevent inconsistencies that could arise, which are not merely due to timing unpredictabilities but could lead to critical errors in data integrity. By ensuring that these operations are atomic, we maintain consistency within the system, even in the absence of traditional synchronization mechanisms.

We will now provide a more detailed overview of the Matrima machine, a key component of the Massimult architecture, as illustrated in Figure 11. This section will delve into the specific components that underlies the Matrima machine, highlighting how it integrates within the broader architectural framework to achieve the desired performance and efficiency.

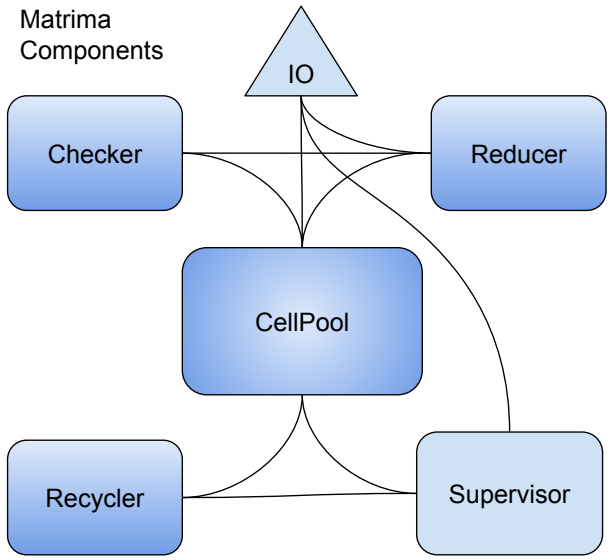


Fig. 8. Matrima Components

VI. Key Components of the Matrima Machine

A. The CellPool

The **CellPool** represents expressions in memory.

The **CellPool** functions as an array of cells, specifically designed to facilitate the binary representation of combinator expressions within memory. In this context, a **cell** denotes the representation of either a node or a leaf within a combinator expression. The compiler is responsible for generating binary code, structured as an array of cells, which can be directly transferred into the CellPool without requiring any modifications.

For each Matrima process, the cells within the CellPool that are accessible from the process's root cell are maintained. Each cell in the CellPool is assigned a unique index, which serves as its reference point. As previously discussed, a computation is considered complete when the root cell reaches normal form. At this point, the result is produced as a binary array of cells, which is derived by compressing the cells, starting from the root cell, into a contiguous array of cells.

Each cell is currently represented as a 16-byte or 128-bit binary expression. This configuration provides an address space of 2^{32} cells, allowing a CellPool to have a maximum

size of 2^{32} cells in this representation, which equates to approximately 69 gigabytes of memory. For applications requiring larger CellPools, it would be necessary to modify the current representation to accommodate the increased memory demands.

A cell consists of 64 bytes of content, a 16-byte reference count, 16-bit flags, 16-bit checker data, and 16 bits reserved for future use. A cell can either be *alive* or *garbage*. If a cell is classified as garbage, it does not represent any meaningful data but remains available for future allocations. Conversely, if a cell is alive, it can be a node or a leaf.

In the case where a cell is a node, its content part consists of 32-bit left and right components. These typically serve as indices within the CellPool, representing combinator graphs. As an optimization, the content part can also contain embedded combinators or primitives, a configuration specified by the cell's flags. Additional flags for nodes store information regarding whether a node is in head normal form or normal form. These flags are crucial for the evaluation process, as they help determine the state of a node's reducibility and guide the reduction strategy accordingly.

If a cell is classified as a leaf, it may represent either combinators or primitives, where primitives can include both primitive operations and primitive data. The flags associated with leaves provide a description of all primitive data types, ensuring that each leaf is properly identified and processed according to its specific type.

It is important to note that the reference count may become negative due to the unpredictable timing issues previously described. To accommodate this, the reference count is represented as `refcount + 16`, allowing for the necessary range to account for potential negative values. The 16 bits allocated for checker arity will be discussed in detail in the subsequent section.

Byte	Node	Leaf
0	Left child	Contents
1		
2		
3		
4	Right child	
5		
6		
7		
8	Flags	Flags
9		
10	Reference count + 16	Reference count + 16
11		
13	Checker arity	Reserved
12		
13	Reserved	Reserved
15		

The CellPool can be conceptualized as the memory of the processor. However, as will be discussed in the next section, it functions as active memory, in that every cell can concurrently execute the checker task. This parallel capability allows the architecture to efficiently manage computations and optimize performance.

B. The Checker

The **Checker** identifies reducible expressions, head normal forms, and normal forms.

The checker is responsible for evaluating three key aspects of a node: (1) determining whether the node can be reduced, (2) assessing if the node is in head normal form, and (3) verifying if the node is in normal form. The checker should be implemented in hardware to enable simultaneous operation on all active node cells in parallel.

To determine if a cell can be reduced, the checker traverses down the left path of the expression until a combinator or primitive operator is encountered. A cell is deemed reducible if the arity of the identified combinator or primitive operator matches the number of steps taken during this traversal.

However, we employ an alternative implementation that utilizes the 16-bit checker arity of each node cell to determine its reducibility. This approach works by propagating the arity upwards step by step while decrementing it by 1 at each step. In practice, each cell reads the arity value of its left child. If the left child is a leaf containing a combinator or primitive operation, the checker arity is set to the arity of the combinator or primitive minus 1. If the left child is another node, the checker arity is set to the checker arity of that node minus 1. A cell is considered reducible when its checker arity reaches 0.

To determine if a cell is in head normal form, it is sufficient to check whether the checker arity is greater than zero. In this case this cell will never reduce.

The third task of the checker is to determine whether a node cell is in normal form. A node cell is considered to be in normal form if it is in head normal form and both its left and right children are also in normal form. When a cell is in normal form, neither the cell itself nor any of its subterms can be further reduced in the future. This concludes the description of the checker's functionality.

C. The Reducer

The **Reducer** performs reductions on single combinators or primitives.

The reducer performs a destructive update on the top cell of a reducible expression to ensure that the cell correctly represents the result of the reduction. During this process, the reducer may allocate new cells, which subsequently initiate new checker tasks. Additionally, the reducer adjusts the reference counts, which may lead to certain cells becoming unreferenced and, therefore, no longer participating in representing any expression. Finally, the reducer sets the checker arities for the newly allocated cells to facilitate subsequent evaluations.

The reducer may also perform primitive operations, such as the division of floating-point numbers or input-output operations. Both types of operations involve interaction with the Supervisor component: the former may trigger a "division by zero" error, while the latter may cause the process to block while waiting for input.

The reducer serves as the core component of the Matrima machine. The description provided here is concise, as the details of combinators have already been covered in Part 1, and the specifics of primitive operations fall outside the scope of this paper.

D. The Recycler

The **Recycler** handles memory recycling for reuse, ensuring efficient memory management.

As previously discussed, the order of updates in the system is non-deterministic. This characteristic has implications for reference counting: it can result in negative reference counts, or reference counts that have reached zero may become positive again. To manage this, we halt the evaluation process prior to recycling and resume it afterward. Recycling involves returning a cell with a reference count of zero to the pool of allocable memory. If the cell is a node, the reference counts of its left and right subcells are decremented accordingly. This is a recursive process, as the reference counts of these subcells may also reach zero, necessitating further recycling. Once this process is complete, the reduction can be resumed.

E. The Supervisor

The **Supervisor** manages computational processes, overseeing the coordination and execution of tasks within the architecture.

A single CellPool can be utilized to perform multiple computations concurrently by simply designating different root cells for each computation. There is no sharing of cells between different processes. Each process is managed by a dedicated supervisor, which is responsible for loading the code into the CellPool, maintaining a record of the root cell, detecting when the computation has concluded, and subsequently compressing and returning the result.

The supervisor is also responsible for handling runtime errors, such as division by zero or out-of-memory conditions. Additionally, it collaborates with input-output primitives to monitor the state of the process, which may, for example, be waiting for a specific input. The supervisor's introspective capabilities will further encompass data metrics such as the time and memory utilized by the process.

VII. Future Work and Conclusion

The Massimult architecture presents a promising new direction for building highly parallel, energy-efficient, and scalable computing systems based on combinator reduction. While this work has demonstrated the theoretical foundations and implemented a virtual machine simulation, several key areas require further exploration and development to fully unlock the potential of this approach.

A. Future Work

Several key areas for future research and development are outlined below:

- **GPU and FPGA Implementation**: To gain traction and showcase the potential of this architecture, it is crucial to demonstrate that it can execute certain programs quicker and with reduced memory consumption on current hardware. The immediate goal is to approach a GPU implementation, followed by targeting FPGAs. These platforms offer the opportunity to scale the architecture and evaluate its performance in real-world high-performance computing environments.
- **Optimization in Speculative Evaluation (Compile Time)**: One of the critical areas for improvement lies in optimizing the speculative evaluation strategy at compile time. By enhancing the compiler's ability to predict which parts of a program will likely need evaluation, unnecessary computations can be minimized, further enhancing the system's performance.
- **Seamless Integration of Array Structures (Compile Time and Runtime)**: Initial research on integrating array structures into the combinator model has yielded promising results. Future work will focus on seamlessly supporting arrays at both compile time and runtime, ensuring efficient handling of indexed data while preserving the benefits of combinator reduction.
- **Compiling Functional Languages to LambdaB and LambdaM**: To execute a wider range of real-world functional programs using the Massimult architecture, one of our intermediate languages shall serve as a backend language for existing functional languages such as Idris. Compiling these languages to an intermediate representation will expand the usability and adoption of the Massimult architecture, fostering a broader and more diverse ecosystem.
- **Localization in Cell Pools (Runtime)**: Enhancing memory allocation and management is another essential aspect of future work. By localizing the allocation of cells in cell pools, locality can be optimized, leading to improved performance during runtime. Additionally, for long-running processes, an automatic reordering of cells should be implemented to further enhance performance over time.
- **Hardware Implementation and Development**: The ultimate objective is the design and fabrication of a physical chip based on the Massimult architecture. This hardware will fully exploit the inherent parallelism and energy efficiency of combinator reduction. After initial validation, the project will progress toward developing a full-scale hardware prototype, which will be essential for benchmarking performance and energy efficiency gains. This step holds the potential to fundamentally transform modern CPU design.

B. Conclusion

The Massimult architecture represents a fundamental departure from traditional Von Neumann-based computing systems. By leveraging the parallelism inherent in lambda calculus and combinator reduction, Massimult offers the potential for more efficient computation, reduced energy consumption, and improved scalability. Although our current implementation remains in its early stages, it has demonstrated the feasibility of this approach and laid a strong foundation for future advancements in both software and hardware.

Ultimately, the success of the Massimult architecture will depend on demonstrating its advantages in real-world applications, particularly in terms of performance and energy efficiency. If these benefits are realized, the Massimult architecture could disrupt the current landscape of computing, providing a foundation for future systems that are faster, more efficient, and more scalable than ever before.

VIII. Appendix