上海交通大學

SHANGHAI JIAO TONG UNIVERSITY

学士学位论文

BACHELOR'S THESIS

论文题目：基于 Blackbird CPU 的测试程序设计

学生姓名: 石达

学生学号: 516021910169

专　　业: 信息工程

指导教师: 刘佩林

学院(系): 电子信息与电气工程学院

基于 Blackbird CPU 的测试程序设计

摘要

Blackbird CPU 是基于函数式编程理念设计的一款处理器，具有名为 fun 的组合子集架构、函数式处理器的 RTL 模型以及支持函数式编程语言 Wu 的编译器。Wu 是针对该硬件架构而设计的一种低抽象级的、惰性的、严格的、函数式编程语言。

本课题对 Blackbird CPU 进行综合测评，即为 Blackbird CPU 的 fun 组合子集架构定义并设计一套适配的测试程序。最终设计测试集包括基本核函数、小型程序和真实应用。另外，在编写测试集的过程中采用多种优化方式，包括组合子集选择的优化，利用数据结构定义的编写方式优化以及利用态射（morphism）原理的编写方式的优化。评估运算性能采用的度量标准包括程序的规约次数、运行周期数、分配存储大小。该硬件结构的测试集设计与测试环境为 Ubuntu。结果表明，采用几种优化方式对不同程序的运算量有不同程度的降低，其中优化效果最为明显的是采用优化后组合子集并利用态射原理编写的程序。最后，课题比较了 GRIN 编译器与 GHC 在运行函数式程序上的性能差异，得出前者更加具有优势的结论。

由于利用不同态射设计不同类型程序需要一定数学基础，因此仅完成利用 catamorphism 对测试程序的优化，大多数测试程序仍然使用数据结构定义的优化编写方式，仍然具有优化的空间。除此以外，该函数式运算架构在面向神经网络等大型并行程序方面的运算性能的优势在实验方面尚待验证。架构中对于文件导入、浮点数处理的模块也有待完善。

关键词：函数式编程，组合子集架构，图规约编译器，组合子合并，态射原理

BENCHMARK PROGRAM DESIGN OF BLACKBIRD CPU

ABSTRACT

Blackbird CPU is a processor designed based on the concept of functional programming. It has a combinator set architecture called fun, an RTL model of a functional processor, and a compiler that supports the functional programming language Wu. Wu is a low-abstraction level, strict, functional system programming language designed for this architecture.

This subject conducts a comprehensive evaluation of the Blackbird CPU, that is, to define and design a set of suitable benchmark programs for the architecture of the Blackbird processor and fun combinator set architecture. The final design benchmark programs include basic kernel programs, toy programs and real applications. In addition, a variety of optimization methods are used in the process of writing benchmark programs, including the optimization of the selection of the combinator set, the optimization of the writing method using the definition of the data structure, and the optimization of the writing method using the morphism principle. Metrics used to evaluate computing performance include the reductions counts, the total cycles, and the allocated cells in the program. The benchmark programs design and testing environment of the hardware structure is Ubuntu. The results show that several optimization methods are used to reduce the computational load of different programs to different degrees. Among them, the most obvious optimization effect is the program written by combining the optimized subsets and using the morphism principle. Finally, the subject compares the performance differences between GRIN compiler and GHC in running functional programs, and draws the conclusion that the former has more advantages.

Because the design of different types of programs using different morphisms requires a certain mathematical foundation, only the optimization of test programs using catamorphism is completed. Most test programs still use the method of writing data structure definitions and still have room for optimization. In addition to this, the operational performance advantages of this functional computing architecture for large parallel programs such as neural networks have yet to be verified in experiments. The module for file import and floating point processing in the architecture also needs to be improved.

Key words: Functional programming, combinator set architecture, graph reduction compiler, combinator fusion, morphism principle

目 录

第一章 绪论

1.1 本课题的研究目的与意义

现代计算机的主流架构为冯·诺依曼架构。该架构中存储程序主要为由处理单元顺序执行的指令。该架构中具有不同级别的存储区，包括寄存器（程序计数器、指令寄存器、寄存器文件）、缓存区、主存等。起初的命令式架构工作效率并不高，流水线的引入通过实现指令级的并行提高了它的工作效率，乱序执行技术和推测执行技术的引入又进一步提高了流水线的执行效率。为了支持这一特征，编译器需要扩展成可以生成可被乱序执行和实现分支预测的代码。这种性能上的提升伴随着日益复杂的软件和微架构。命令式编程语言要求程序按照顺序执行，因为执行的顺序将会影响到变量以及存储空间的状态[3]。因此，在执行乱序指令时，处理器需要跟踪操作原来的执行顺序进而保证该程序的状态与顺序执行机中程序的状态相同。在推测执行过程中状态十分重要，尤其是当程序走向违背预测的分支时计算机必须退回到原来的状态并舍弃该乱序操作结果的情况发生时。而主流架构中的缓存区并不在这一状态退回的操作范围内，因此给计算机带来安全隐患，如发生侧信道攻击等[1,2]。现代计算机架构的劣势可被归纳为：（1）架构过于复杂，大量存储单元主要用于存储推测执行中所需数据的副本[4]；（2）即使C语言仍然是当今世界主要使用的语言之一，该架构与C语言不再形成映射关系[3]；（3）恶意代理可轻易使用访问和修改状态的指令，进而对计算机执行过程进行直接控制。

在保证将程序存储用以计算的形式所带来的优势和可编程性的前提下，一种无状态的基于函数式编程理念设计的处理器 Blackbird CPU 被提出。它主要通过限制状态的方式来避免在现代计算机中存在的安全问题，如数据泄露、侧信道攻击等。该处理器具有名为 fun 的组合子集架构以及支持函数式编程语言 Wu 的编译器[5]。由于语言 Wu 是在设计该处理器中提出的一种新的函数式编程语言，所以并没有相应的采用 Wu 进行编程的基准测试程序集来测试该处理器的实际性能。

1.2 函数式编程的发展现状

函数式编程作为一种编程范式，经常被认为是命令式编程的替代选项。相比较命令式编程而言，函数式编程更加侧重于函数的执行，其主函数的基本形式即是对于输入实参应用嵌套的函数。

函式编程的理论根源是组合逻辑的思想，即利用组合逻辑符号与自然数表示形式系统中的一切公式。在此基础上，逐渐发展出输入单参数的 λ-演算、利用嵌套形式实现的可输入多参数的 λ-演算、组合子逻辑、超级组合子逻辑等等逐渐优化的理论，在理论优化过程中也出现了例如 FORTRAN、LISP、Haskell 等典型的函数式编程语言的发展，以及例如图规约编译器、SKI 机、G 机等硬件架构层面的发展。

由于函数式编程语言与代数理论密切相关，在编写和优化的理解上较有难度，因此并不被许多人理解和学习。但更加重要的原因是，主流的现代计算机架构大多采用在冯诺依曼架构的基础上改进的命令式架构，在此架构下命令式编程语言的编译代码的执行效率优于函数式编程语言。

然而事实证明，函数式编程在发展方面仍然有它的可取之处。一方面，函数式编程的无

状态、无副作用、存储的即用即删、实参的惰性调用的特征为其在安全性方面的发展提供了理论依据（而相比之下，乱序执行的命令式架构并不能很好地应对一些现代安全性问题，如侧信道攻击）；另一方面，函数式编程在并发并行编程上的优势，使得其在大型并行应用上的发展潜力得到关注，一些新的函数式语言（例如 Sun 的 Fortress、Microsoft 的 F#）已被相继开发出来[6]。除此以外，函数式程序相比命令式程序更加简洁且高效；函数式硬件架构的开发也将促使函数式编程语言的进一步发展。

1.3 本文的主要工作

由于函数式编程语言 Wu 是一种新开发的函数式编程语言，因此尚未有以其编写的基准测试程序集出现，为了综合测评 Blackbird CPU 的运行性能，本课题的主要内容即为编写相应的基准测试集并作相应的测试分析。

本文的主要工作包括：

- 对函数式编程语言的特性、组合子的基本定义以及图规约编译器的基本实现原理作系统阐述；
- 提供关于优化方面的组合子合并理论、态射理论的介绍；
- 简要介绍函数式编程语言 Wu 并简要介绍组合子集架构 fun；
- 给出基准测试程序集，包括基本核函数、玩具函数、真实应用；
- 比对测试集测试结果并给出相应分析；
- 对课题成果作归纳总结并讨论课题可能的发展潜力。

第二章 函数式编程的理论介绍

2.1 函数式编程的基本理论

2.1.1 函数式编程语言的特性

函数式编程得名于它的基本操作，即对于实参的函数调用。在函数式程序中，主函数会被写作输入实参、输出结果的函数的形式，其中的函数可以是简单的逻辑函数，也可以是嵌套或组合应用的多个高阶函数，而高阶函数的定义或者是有简单的高阶函数嵌套组合定义、或者是有简单的逻辑函数嵌套组合定义，基本的逻辑数学函数则是处在一切函数的最底层的基本函数，即函数式编程语言的数学原语。

函数式编程语言的最显著的特性为不包含赋值语句，即程序中使用的变量仅有初始的赋值，而在函数调用的过程中不会被重新赋值或改变。这一特性充分说明函数式程序中函数的调用仅用于计算结果，而没有其它的副作用，即任一函数表达式在确定输入时输出就已经确定了，任一函数都是引用透明的；同时函数的调用将不会对函数以外的部分造成任何影响。这一特性一方面避免了副作用带来的不必要的调试错误，另一方面不同函数调用的执行之间没有相互影响因此可以并发并行而不用考虑彼此之间的执行顺序。

函数式编程语言的更为重要的特性是结构良好[7]。结构良好的编程语言在设计函数时往往先设计编写一组通用的模块。这种模块化的编写方式主要优势包括：（1）模块因其编写量较小而更容易实现；（2）任一通用模块都将应用到一组同类函数中，大大提高了代码重用率，减少了未来的编程成本；（3）程序中所使用的各模块之间相互独立，因此可以分开调试，减少调试时间。函数式编程语言在实现模块化设计的方面主要通过两种手段：高阶函数和惰性计算。

高阶函数是连接函数的方式。通过将同一类简单函数模块化，我们可以得到一些通用的模块，并进一步将这些通用模块应用到更多的同类函数中，简化该类函数的编写。在新的数据类型被定义时，仅需要增添少量模块的定义，即可较为容易地扩展该类型的相关函数。举例如图 2-1 所示。在实现一列表类型的元素求和运算时，首先通过图样匹配（pattern match）和迭代的方式简单定义；在此基础上抽象出函数 foldr，该函数可将列表映射到单个值；利用函数 foldr 可定义元素求和、元素求积、求元素是否有真值等同一类型的函数。

要求：有数列 list：[1, 2, 3]，现需要设计函数对该数列进行求和。

$$[1,2,3] = (1:2:3:[\,]) = (:1\,(:2\,(:3\,[\,]))) = (Cons\ 1\ (Cons\ 2\ (Cons\ 3\ [\,])))$$

定义：

$$sum\ [\,] = 0$$
$$sum\ (Cons\ n\ rest) = +n\ (sum\ rest)$$

将 sum 模块化为通用的迭代的形式：

$$sum = foldr\ (+)\ 0$$

其中，foldr 函数定义为：

$$foldr\ f\ x\ [\,] = x$$
$$foldr\ f\ x\ (Cons\ a\ l) = f\ a\ (foldr\ f\ x\ rest)$$

验证：

$$foldr\ (+)\ 0\ [1,2,3] = +1\ (foldr\ (+)\ 0\ [2,3]) = +1\ (+2\ (+3\ 0)) = 6$$

应用：

$$product = foldr\ (*)\ 1$$
$$anytrue = foldr\ (\cup)\ False$$

图 2-1 高阶函数举例

惰性计算是连接程序的方式。当程序由嵌套或组合的高阶函数构成时，高阶函数仅在被要求提供其返回值时被调用，而当其返回值未被需要时将不会被调用。这一优势减少了不必要的函数调用所带来的时间消耗。相较函数式编程而言，命令式编程范式因为需要考虑副作用，在编写程序时需要预先考虑函数流的执行顺序，因此难以实现惰性计算。具体案例如在执行程序“or (true) (A)”（A 为一函数调用）时，函数式程序在编译时将根据 true 直接返回 true，并直接丢弃 B 段的程序内容，相比之下命令式程序则需要先调用函数 B 并给出其返回值，最终再返回 true。

2.1.2 组合子理论

组合子是没有自由变量的函数。在经典的 SKI 机理论中，Turner 给出了最基本的组合子定义，以此作为图规约的符号表示形式[8]。组合子定义为等式的形式，等式左侧为组合子的名称以及它需要的输入实参，等式右侧为组合子的在规约后的返回值。

最起始的组合逻辑仅采用 S 和 K 两种符号（机器的整个指令集所需要的仅仅是 S 和 K），它们满足：

$$K\ x\ y = x \tag{2-1}$$

$$S\ f\ g\ x = f\ x\ (g\ x) \tag{2-2}$$

在 S 和 K 的基础上定义 I（恒等函数），I 满足：

$$I = S\ K\ K \tag{2-3}$$

$$I\ x = S\ K\ K\ x = K\ x\ (K\ x) = x \tag{2-4}$$

任何 λ-演算都可以表示成基本组合子的形式：

$$\lambda\ a.a \rightarrow I \tag{2-5}$$

$$\lambda\ a.b \rightarrow K\ b\ (a \neq b) \tag{2-6}$$

$$\lambda\ a.p\ q \rightarrow S\ (\lambda\ a.p)(\lambda\ a.q)\ (p,q\ is\ any\ funtion\ application) \tag{2-7}$$

当需要采用列表类型和整型数作为其支持的数据类型时，对于组合子模型则需要添加定义少量相应去处理该数据类型的高阶函数，如：

$$plus\ m\ n = (the\ sum\ of\ m\ and\ n) \tag{2-8}$$

$$cons\ a\ b = (a\ list\ node\ with\ refs.to\ a\ and\ b) \tag{2-9}$$

相比 λ-演算模型而言，组合子模型不需要涉及该程序外的内容，即不需要将变量保存为类似于环境的东西，因此其寄存器和堆栈中的内容对于终端使用者不可见（更具有保密性），而 λ-演算模型则可以通过数据和地址的关联获取内容。同时，因为组合子模型的惰性计算的性质，它的对标准指令应用规约过程的运算量会远远小于 λ-演算。

在 SKI 机中定义的组合子集所包含的除部分算术逻辑运算符和部分内部用于调试的组合子以外的组合子如表 2-1 所示：

表2-1 SKI机中的主要组合子

Combinator	Reduction rule
S	S f g x = f x (g x)
K	K x y = x
I	I x = x
B	B f g x = f (g x)
C	C f x y = f y x
S’	S’ k f g x = k (f x) (g x)
B’	B’ k f g x = k f (g x)
C’	C’ k f x y = k (f x) y
Y	Y f = f (Y f)

续表 2-1

Combinator	Reduction rule
P	P a b = a.b (a list)
U’	U’ f g a = f a (if a is not a list)
	U’ f g (p.q) = g p q (where p.q is a list)
A	A b1 b2 = b1 and b2 (boolean)
O	O b1 b2 = b1 or b2 (boolean)
U	U f x car cdr = f (car x) (cdr x)

2.1.3 图规约编译器

图规约编译器 GRIN（Graph Reduction Intermediate Notation） Compiler 是一个面向函数式编程的编译器框架，可以支持函数式程序的惰性计算、运行使用逻辑变量的程序[10]。

在编译函数式程序时，编译程序将原程序编译为一个有根图（rooted graph）存储在专门设置的存储区内（可以是堆栈等）。该有根图的节点为架构定义的数据结构，它的弧为指针，它的每个节点可以仅拥有一个存储 cell 用来存储单一的组合子或实参和指针的组合，也可以设置为两个 cells 分别存储组合子、实参或指针。

规约是指根据组合子的规约规则对有根图进行有方向性的重写，即对表达式进行替换、匹配图样、重组形式等操作。而对于该有根图的不断应用规约规则进行规约的过程即是程序运行的过程。

在运行函数式程序方面，对于组合子图的直接规约是一种很有吸引力的计算范式，因为这一过程无需另外的程序加以控制。因为组合子图的直接规约是根据组合子在子图中的位置根据规约规则推断出下一步规约，其过程控制是自发的。这使得程序员在解决指定的约束系统的解决方案时，无需详细确定该解决方案各独立部分产生的顺序，尤其在使用到回溯控制时，惰性计算可完全透明地实现而不借助程序员的控制。另外，编译过程可以检测未被调用的子表达式中的大多数，而当编译器检测到需要调用的变量而未被获取时，才会调用相应的函数获得其返回值。

举例程序“main = ((+ 1 2) == 3) 1 2”来说明图规约的基本过程，如图 2-2 所示。该程序的实现内容为：计算 1 与 2 的和值，将结果与 3 进行比较，如果相等则返回 1，否则返回 2。首先编译器在存储器中将程序存储为 2-2(a)的形式，然后从最左侧开始寻找组合子，当遇到最左侧的组合子将依据组合子的规约规则进行规约，当输入实参未完全准备好时，则调用相应函数要求输入实参，然后再进行接下来的规约。当规约过程完成时，该有根图内仅包含最终的返回值 1。

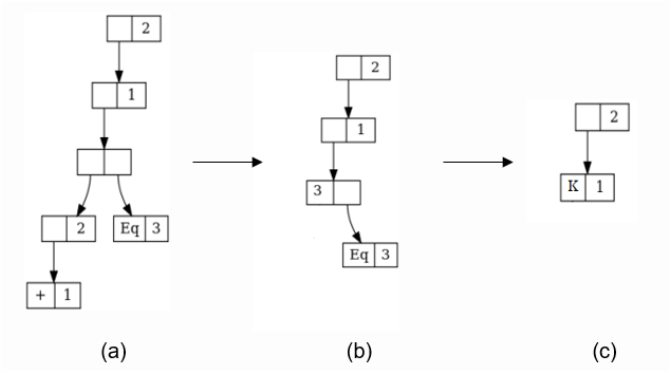

图 2-2 图规约示例图

2.2 函数式编程的优化理论

利用组合逻辑的理念去设计函数式硬件架构的有效途径之一。这种方法最大的优势在于精简，一方面在程序编译为组合逻辑的过程中去除了变量绑定语句（let 语句），另一方面在函数定义和运行过程中不需要考虑执行顺序，因此省略了冗杂的控制操作。

然而，最基本的组合逻辑设计也具有其自身的劣势[9]。这种劣势主要体现为：（1）组合逻辑在进行图规约的过程中粒度较小，即单次规约所改变的存储内容、实现的运算量较小；（2）程序在编译为组合逻辑后，采用组合子与实参的组合形式所构成的组合逻辑表达式较长。在考虑到实际硬件的运行效率方面，这两者都是需要被直面的问题。

在组合逻辑的发展过程中，组合子合并理论和态射理论在一定程度上优化了组合逻辑的劣势，使得利用组合逻辑实现的硬件执行效率大大提高。

2.2.1 组合子合并理论

组合子是函数式程序中一切函数的最底层函数，即函数式程序在编译后会成为采用组合子与实参的组合形式所构成的组合逻辑表达式。组合子合并理论针对单次规约运算量较小的问题，提出将原来使用的最底层的组合子进行合并形成超级组合子[11]。在实现单次运算量提高的同时，该方法还减少了组合逻辑表达式的大小，并且只需要对底层的组合子与编译器进行修改，而不需要改变上层的函数式程序。

此处我们使用 foldr 函数的组合逻辑表达式规约过程作为示例[9]。在使用基本组合子集对程序进行编译后，foldr 将编译为：

$$Y(B'(B'(S'(C)))(C(I))(S'(B'(C))(B'))) \tag{2-8}$$

编译之后形成的树图如图 2-3 所示。

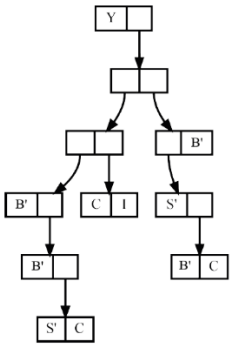

图 2-3 使用基本组合子编译的 foldr 的树图

此时，对于函数 foldr，我们引入四个利用组合子合并定义的超级组合子：

$$B1 = B'(B\ a) \tag{2-9}$$

$$S^* = S'(C) \tag{2-10}$$

$$T = C(I) \tag{2-11}$$

$$G = B'(C) \tag{2-12}$$

利用加入的组合子编译函数 foldr 后所得组合逻辑表达式为：

$$Y(B1(S^*(T))(S'(G)(B'))) \tag{2-13}$$

编译之后形成的树图如图 2-4 所示。

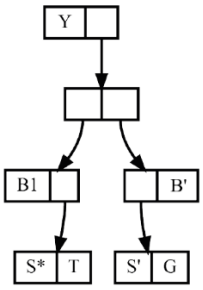

图 2-4 组合子合并后编译的 foldr 树图

结合图 2-3 与图 2-4，我们可以看到，在经过组合子合并后，编译的组合逻辑表达式更加精简，对应的树图所占用的存储空间也相应减少。而在规约过程中，使用新的组合子所带来的单次规约存储空间内容的变化量——运算量提高。

至此我们仍然可以对现有的组合子进行进一步的合并。但我们必须考虑组合子合并带来的影响：（1）组合子合并意味着组合子的通用性降低，即新的组合子往往使用在某一类函数的编译中，进一步的合并所带来的编译效率上的提高有待验证；（2）组合子合并意味着单词规约运算量的进一步提高，相应地，单次规约所增加、舍弃地存储空间数量也会提高，这种合并提高了单次规约的时间（在存储空间读写速度不足时），同时也要求垃圾收集器的性能的跟进。具体是否采用新的组合子，需要根据实际测试集的测试结果分析来确定。

本课题中所采用的用于对照的基本组合子为 Turner 组合子集架构，其内容（不包括算术逻辑组合子）为如表 2-2 所示，内容为：

表2-2 基本组合子

Combinator	Reduction rule
S	S f g x = f x (g x)
K	K x y = x
KI	KI x y = y
I	I x = x
B	B f g x = f (g x)
C	C f x y = f y x
S1	S' k f g x = k (f x) (g x)
D	B' k f g x = k f (g x)
C1	C' k f x y = k (f x) y
Y	Y f = f (Y f)

2.2.2 态射理论

观察并归纳出函数的模式可以实现更加直接的优化，这种优化方式也更加易于编写和检测，同时也能相应地定义更加通用的组合子集。在分类理论中，有一种可以采用函子（functors）与态射（morphisms）来归纳出集合、函数以及数据结构的工具——态射理论[12]。

从映射关系来看，态射是一种高阶函数，它可以将一种种类转换为另一种。此处我们仍然采用 foldr 作为案例[9]。首先，我们观察函数 foldr，它的作用是将一个元素类型为 a 的 list 转换为一个类型为 b 的值（类型 a 与类型 b 可以是同一类型）。这种形式的转换在态射理论中对应于 catamorphism 的功能，它实现转换的方式是对一种数据结构进行迭代操作。许多具有同样模式的函数可由 catamorphism 来构造，但在利用 catamorphism 构造出函数 foldr 函数之后，这一类函数几乎都可以由 foldr 来构造，例如对 lists 内的所有元素求和、求积、求 lists 的长度等等，具体实现见图 2-1。catamorphism 不仅可以用于 lists，还可以应用于 binary-trees、quad-trees、tuples 等其它数据结构。

利用 catamorphism 构造 foldr 首先我们需要定义一个组合子 cata，并构造一个与数据类型 lists 相关的函子 functorList：

$$fun\ functorList\ f\ xs\ =\ xs\ []\ (\backslash h\ ->\backslash t\ ->\ (h:\ (f\ t))) \tag{2-14}$$

$$funrec\ cata\ functor\ f\ =\ f\ .\big(functor\ (self\ functor\ f)\big) \tag{2-15}$$

利用组合子 cata 与函子 functorList 构造函数 foldr：

$$fun\ foldr\ f\ z\ =\ cata\ functorList\ (\backslash x\ ->\ x\ z\ f) \tag{2-16}$$

由于 cata 可以应用于不同的数据类型中的同类转换函数，所以在将其应用于其它数据类型时，只需相应地构造对应该数据类型的函子，而函数 foldr 的构造形式完全相同。

利用态射理论构造函数的优势在于：（1）使得函数的构造与函数的模式特征更加直接地贴合，在函数执行时减少了不必要的控制过程；（2）使得新的数据类型的扩展更加方便，仅需构造对应的函子即可实现大量函数，提高了代码重用率；（3）利用态射自身的性质可以进一步简化代码，优化存储空间使用和运行时间，具体案例如函数 foldmap 的构造[9]。

函数 foldmap 的功能为对 lists 类型数据通过某一函数进行映射（map），对映射后的 lists 再进行 foldr 操作，使之返回一个数值。具体定义如下：

$$fun\ map\ f\ =\ cata\ functorList\ (\backslash xs\ ->\ xs\ []\ (\backslash h\ ->\backslash t\ ->\ ((f\ h):\ t))) \tag{2-17}$$

$$fun\ foldmap\ f\ g\ z\ =\ (foldr\ f\ z)\ .\ (map\ g) \tag{2-18}$$

此时，利用 catamorphism 的性质：

$$(cata\ functorA\ f)\ .\ (cata\ functorA\ g)\ =\ cata\ functorA\ (f\ .\ g) \tag{2-19}$$

由此构造函数 foldmap 为：

$$\begin{aligned} fun\ foldmap\ &=\ cata\ functorList\ (f\ .\ g) \\ &where\ f\ =\ (\backslash x\ ->\ x\ z\ f) \\ &\quad g\ =\ (\backslash xs\ ->\ xs\ []\ (\backslash h\ ->\backslash t\ ->\ ((f\ h):\ t))) \end{aligned} \tag{2-20}$$

通过这种方式构造函数 foldmap，我们不需要设置额外的存储空间来放置在映射之后形成的暂态的 lists 类型数据，同时也会减少函数测试时的运行时间。

利用态射理论构造函数的缺点在于，当我们使用这种迭代模式的方法时，函数的可读性将会下降，不利于表达式的理解。在态射理论中，除去 catamorphism 的方式以外，还有 anamorphism、paramorphism、hylomorphism 等工具，它们分别对应其它的模式转换类型的函数构造的优化。本课题中主要使用的优化方法是 catamorphism。

2.3 本章小结

本章内容围绕函数式编程进行阐述。

本章首先系统地归纳了函数式编程语言所具有的优势——无状态、无副作用、易于模块化、具有惰性计算特性等等；然后对函数式编程中最底层的函数组合子进行介绍，这里使用的案例是较为经典的 SKI 机模型；在介绍完组合子之后，再对具有函数式编程特性的经典编译器模型——图规约编译器进行原理的阐述。由此从函数式编程的程序设计到编译行为的基本原理的部分都进行了介绍。

在介绍完基本原理之后，本章再对函数式编程中两种行之有效的优化方式的原理作举例阐明。首先是组合子合并形成超级组合子的优化方式，该途径直接与图规约编译过程关联，易于理解和实现，但合并后的组合子的通用性降低，因此是否采用新的组合子需要根据基准测试程序的测试分析结果进行取舍；其次是利用态射理论对函数的构造进行优化，这种优化方式取自于分类理论，它与函数功能模式直接关联，构造时需要结合数据类型的函子，在优化时需要结合数学原理，有一定复杂性，但利用该途径构造的态射组合子更加具有通用性。

第三章 Blackbird CPU 与基准测试程序集

3.1 函数式编程语言 Wu

编程语言 Wu 是应用于组合子集架构 fun 的、低抽象级的、静态类型的（数据类型是在编译其间检查的，即在写程序时要声明所有变量的数据类型）、多态的（抽象类型可以泛化为具有不同行为的具体类型；在运行时根据类型做出不同的行为解释）、惰性运行的、纯函数式编程语言。相比较 Haskell、Erlang 等主流的高抽象级的函数式编程语言而言，它增加了内置的算术逻辑运算组合子、输入输出组合子以及用于异常处理、图配置、内务处理（house-keeping）的组合子[5]。

Wu 的数据类型包括 literals（32-bit number, Int, Float; 8-bit characters, Char; Boolean values, Bool）、lists（最后一个元素是空列表组合子[]）、Tuples。Wu 的类型检查机制要求，lists 的元素仅能为同一种数据类型，而 tuples 的元素可为不同的数据类型。在程序中可以使用组合子 IN 和 OUT 来模拟数据在堆与外存之间的读写操作。

Wu 的语句结构包括：（1）let 语句用于对实参的初始定义，该实参在程序运行过程中以 let 语句的返回值的形式被引用（let 语句仅运行一次，之后所有的引用共享同一个返回值），因此该语句所对应的存储仅在对于该语句的所有引用都有效后才被舍弃，实参在程序过程中不能被再次赋值；（2）case 语句用于条件语句；（3）where 语句用于简明化函数中过于复杂的或者重复使用的变量。

Wu 编写的程序要求定义一个主函数 main，这是编译器所理解的表达式规约图的起始。Wu 中的所有循环都需要用迭代的形式实现，在迭代函数的定义前需标记其类型为 funrec（非迭代函数标记 fun），在定义中调用的函数自身用 self 表示。

Wu 是一个垃圾收集（garbage-collected）语言，即被舍弃的存储空间将在总存储空间占用达到一定程度时被自动清理，因此不同于命令式语言的允许存储分配；垃圾收集过程是硬件内实现的，并不属于编译器编译、程序执行时间的一部分。

3.2 组合子集架构 fun

3.2.1 组合子集架构 fun

组合子集架构 fun 中所使用的组合子集包含：基本组合子（Turner 组合子集的扩展）、内置算术逻辑组合子、输入输出组合子[5]。

组合子集架构 fun 实现顺序执行的图规约操作，规约过程符合惰性计算的特性。无论是实参还是组合子函数都存储在同一个堆内。该堆的最小存储单位为 cell，cell 的大小为 72 bits。每个 cell 由两个大小为 36 bits 的部分组成，如图 3-1 所示，具体组成为：大小为 32 bits 的信息字段（组合子或实参）、3 bits 的类型字段以及 1 bit 的垃圾收集标志字段（用于存储管理）。其中，类型字段共有 7 种可能，分别是：CTRL（控制指令）、PTR（指针）、LIT（Literal）、COMBI（组合子）、LPTR（列表指针）、TLPTR（终端列表指针）、RPTR（返回指针）。后两种类型用于图规约的内部操作，因此不可访问。

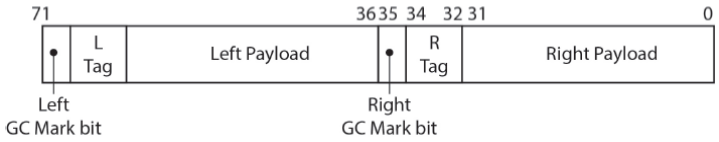

图 3-1 存储单元 cell 的具体构成

共有两种工作模式：用户模式和管理员模式。输入输出组合子仅在管理员模式下被允许。权限管理由操作系统（纯函数操作系统核 funk 尚未完成）完成。

CSR（特殊控制寄存器）是一种用于控制的寄存器，仅在管理员模式下可被使用。其它寄存器包括：RA（根地址）寄存器、HS（堆尺寸）寄存器和 FH（空闲列表头）寄存器。其中 RA 用于设置将要运行的图的全局的根，它可以被操作系统控制从而实现多进程（多个图规约）；HS 和 FH 由垃圾收集器使用，来管理每个进程执行时可使用的存储空间。FH 在新的 cell 被创造时动态更新。垃圾收集器将利用这 3 个寄存器来确定哪些 cells 将不再被使用，即没有指针指向这些 cells，并回收这些 cells 让应用程序能够重新使用。

在图规约处理方面需要采用利用到 PRTR（返回指针）类型的 cell 作为规约的中间步骤所需的存储空间，PRTR 指针仅用于处理严格组合子，即数学运算和 I/O 组合子。图规约的具体步骤举例程序“main = + (- 9 4) 3”，其图规约的具体过程如图 3-2 所示。当编译器从根处寻找最左侧的组合子时，找到 add，因为 add 所需要实参并未都准备好，所以不能执行，如图 3-2(a)所示；于是编译器创造一个新的 cell，其左侧为指向原图根 cell 的 PRTR 指针，右侧为指向子图的指针，如图 3-2(b)所示；此时子图的函数 sub 所需的 2 个实参已准备好，于是执行完成后，将返回值 5 保存至新 cell 的右侧空间内，如图 3-2(c)所示；之后将新 cell 右侧的 5 存储至原图中连接的 cell 的右侧空间，并重新从根 cell 进行规约，如图 3-2(d)所示；最终执行函数 add 并返回 8。

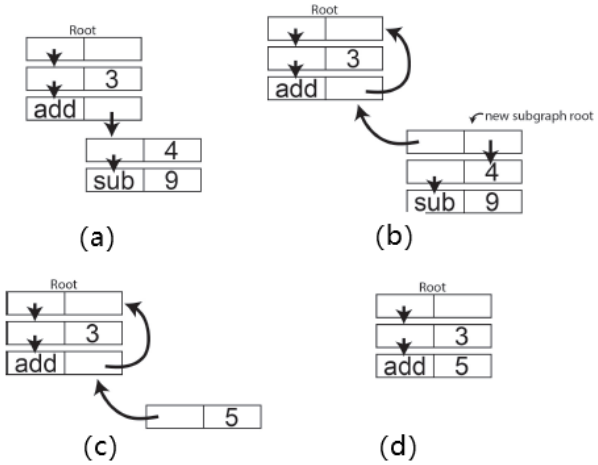

图 3-2 图规约过程实现步骤

由于在真实应用中输入输出是最为基础且具有副作用的功能，该副作用在理论上不会对安全性造成影响。组合子集架构中输入输出功能是直接通过定义应用于不同长度字符的输入输出组合子实现的。输入输出组合子仅在管理员模式下可被访问。具体的组合子包括 IN、OUT、HALT，分别具有不同字符长度的组合子，分别实现输入、输出、中断。具体的定义见附录 2。

3.2.2 微架构

Blackbird 处理器的微架构的最基础单元包括：GRU（图规约单元）、HMU（对处理单元）与 heap subsystem（堆子系统）。另外，该架构还具有 WISHBONE 外设单元用于支持输入输出信号和外界接口。[5]

搜索单元遵循 Church-Rossor 定理实现图遍历。图从最左侧节点开始遍历，直到找到一个组合子函数。在遍历到组合子后，搜索单元暂停。在遍历过程中，每个节点都被解码并且推入 GPW（graph-spine window，图脊窗口）中。GPW 是 SGR 处理器（stackless processer，无状态处理器）中唯一的内置存储空间。该架构采用 5 个链式寄存器（chained register）构造 GPW，既节省空间，又利于并行的读写操作。这些链式寄存器在遍历时以移位寄存器（shift-register）使用，在组合子规约时以并行形式更新。

解码单元持续地读入最左侧节点的内容，并生成用于规约时堆内更新的控制信号与用于存储管理即垃圾收集的控制信号。

规约单元依照组合子规约规则执行组合子规约，在一个时钟周期内并行修改多达 4 个堆单元和所有 GPW 的内容。

堆以存储块网络形式组织，每个有 72bits 宽和 15cells 深，用于存储编译器生成的规约树图。该堆允许在一个时钟周期内对 4 个存储位置同时进行读写操作。存储管理即在图规约过程中动态重用废弃的 cells 的功能由垃圾处理器实现。该堆还支持外部存储，但在使用外存时获取任一图节点时将会产生延迟。

3.3 基准测试程序集

该套基准测试集的设计，一方面是为 Blackbird 处理器提供一套最基本的核函数，为在此基础上的编程变得更加便捷和简洁；另一方面，由于 Blackbird 处理器具有新的组合子集架构 fun 和新的函数式编程语言 Wu，因此需要通过一组基准测试函数真实反映 Blackbird 处理器的性能，本课题主要侧重于运行时间以及存储空间占用上的性能测试。

该套基准测试集包含三种级别的测试程序：基础核函数、小型程序和真实应用，详细内容见附录 1。测试集使用编程语言 Wu 编写，简明易用，面向数据类型为 literals、lists 和 tuples。在编写任一测试程序时，需要编写一个 main 函数语句作为图规约编译器编译的入口。

3.3.1 基本核函数

基本核函数处于应用层面，它们构成应用程序中短小且关键的部分。本课题设计所定位的基本核函数主要完成各种最基本数据类型的定义构造、各数据类型的基本操作以及输入输出等辅助操作。

基本核函数的设计目的主要为：（1）验证处理器基本功能实现的正确性；（2）作为小型程序以及真实应用程序的组成部分；（3）作为测试程序，验证优化理论对于处理器运行性能提升的改善效果，本课题中主要使用它们评估采用态射理论后的优化效果。

基本核函数的详细内容见表 3-1。

表 3-1 基本核函数

Application domain	Name	Description
Bool	and	逻辑与
	or	逻辑或
	not	逻辑否
	xor	逻辑异或
Numeric	abs	取绝对值
	max	取较大值
	min	取较小值
	pow	求整数幂
	sqr	求平方
	cube	求立方

续表 3-1

Application domain	Name	Description
Numeric	gcd	取两数绝对值并求最大公约数
	succ	加一操作
Lists	isnull	判断列表是否空
	head	取第一个元素
	tail	取第一个元素以外的列表
	init	取最后一个元素以外的列表
	last	取最后一个元素
	foldr	从列表右侧开始的折叠
	foldl	从列表左侧开始的折叠
	map	列表映射生成新列表
	filter	列表过滤生成新列表
	scanl	见下文
	zipWith	见下文
	iterate	见下文
	ones	生成元素为 1 的无限长列表
	ones_pn	生成元素为+1、-1 相间的无限长列表
	from	生成从 n 开始、步长为 1 的顺序列表
	range	生成从 m 到 n、步长为 1 的顺序列表
	length	求列表长度
	sum	列表元素求和
	prod	列表元素求积
	reverse	列表倒置
	average	列表元素求平均值（返回值四舍五入）
	foldmap	映射列表并作折叠操作
	zip	将两个 literal 列表合并为一个 tuple 列表
	concat	拼接两列表
	take	以列表前 n 个元素生成新列表
	drop	不包括列表前 n 个元素生成新列表
	partition	见下文
	split	见下文
	dropWhile	见下文
	takeWhile	见下文
	andList	列表元素与操作
	orList	列表元素或操作
	elem	判断是否属于列表
	repeat	生成仅含有同一数值元素的列表
	all	判断列表元素是否任意都符合一条件
	any	判断列表元素是否有符合一条件

续表 3-1

Application domain	Name	Description
Vector	index	取向量第 n 个元素（从 0 计数）
	first	取向量第一个元素
	second	取向量第一个元素
	third	取向量第三个元素
	vecdotprod	向量点乘
	veccrossprod	向量叉乘
	scale	向量缩放
	vecmodulo	向量取模的平方
	vecplus	向量求和
	vecminus	向量求差
Matrix	matsize	求矩阵大小
	isSingle	判断矩阵是否仅有一个元素
	matprod_line	向量与矩阵乘积
	matprod	矩阵乘积
Sort	quicksort	快速排序
	insertSort	插入排序
	mergeSort	合并排序
Tuple	fst	取元组第一个元素
	snd	取元组第二个元素
	tupleswap	交换元组的两个元素
	curry	将两个数值合并为元组并输入一函数
	uncurry	取出元组内两元素并输入一函数
ASCII	itoa	将整数转换为 ascii 码
	toUpper	小写字母转大写字母
	toLower	大写字母转小写字母
	isAlpha	判断是否为字母
I/O	printbool	输出布尔类型
	printMaybe	输出 monad 类型值
	err	输出异常信息
	put	GPIO 输出 literal
	putc	UART 输出 literal
	print	UART 可控输出 literal
	printlist	GPIO 输出 list
	printl	UART 输出 list

表格中补充函数功能描述：

- 函数 scanl：列表操作。输入为需要两个输入实参的函数 f、起始值 z 和待操作列表 xs。从左侧开始，生成新列表的第一个元素为 z，第二个元素为输入为新列表第一个元素 z 与 xs 的第一个元素的函数 f 的返回值，第三个元素为输入为新列表第二个元素 z 与 xs 的第二个元素的函数 f 的返回值，依此类推。例如程序“main＝scanl

+ 0 [1,3,5]”将返回“[0,1,4,9]”。

- 函数 zipWith：列表操作。输入为需要两个输入实参的函数 f 和两个列表 xs1、xs2。生成新列表的第一个元素为输入为 xs1 的第一个元素与 xs2 的第一个元素的函数 f 的返回值，第二个元素为输入为 xs1 的第二个元素与 xs2 的第二个元素的函数 f 的返回值，依此类推。当其中任一列表没有元素时，则新列表末尾添加空符号并返回。例如程序“main = zipWith + [1,2,3] [2,3]”将返回“[3,5]”。
- 函数 iterate：输入为单输入函数 f 与 literal 类型数 x。生成新的列表内容为“[x, f x, f (f x), f (f (f x)), …]”。
- 函数 partition：输入条件 cond 与列表 xs，返回一个元组（tuple）。该元组内容为“（符合条件的元素生成的列表，符合条件的元素生成的列表）”。
- 函数 split：输入为整数 n 与列表 xs，返回一个元组。该元组内容为“（列表的 n 个元素组成的列表，列表的剩余元素组成的列表）”。
- 函数 dropWhile：输入条件 cond 与列表 xs，返回一个列表。从列表 xs 起始处依次判断元素是否符合条件 cond，若当前元素符合条件，则直接丢弃；若当前元素不符合条件，则保留当前元素以及后续元素，生成新的列表返回。
- 函数 takeWhile：输入条件 cond 与列表 xs，返回一个列表。从列表 xs 起始处依次判断元素是否符合条件 cond，若当前元素不符合条件，则直接丢弃；若当前元素符合条件，则保留当前元素以及后续元素，生成新的列表返回。

3.3.2 小型程序

小型基准测试程序指的是一些不具有实际应用、但具有一定运算量的测试程序。它们通过组合、调用基本核函数的方式构造。小型基准测试程序的设计目的与基本核函数的目的相似，即在更大运算量情况下在处理器功能性以及优化理论对处理器的性能提升方面进行验证。本课题中主要使用它们评估采用组合子合并理论后的优化效果。

本课题在选取小型基准测试程序方面，采用了 guthub 用户 csabahruska 设计的、使用语言 Haskell 编写的、用于测试图规约编译器的测试程序集[13]，主要包括 fact、fib、linfib、sieve、tak、tsumupto、foldfrom、hamdist 这 8 个测试程序。它们的功能简介如下（具体源码见附录 1）：

- 函数 fact：利用递归的形式实现对自然数 n 的阶乘。
- 函数 fib：利用递归的形式生成斐波那契数列中的第 n 个元素。
- 函数 linfib：利用递归的形式生成以任一整数为起始的类斐波那契数列的第 n 个元素。
- 函数 sieve：输入数列，要求返回数列中没有任一元素数值是另一元素数值的整数倍，如果原数列中有，则舍弃较大数值的元素。例如程序“main = sieve [2,4,6,7,9]”将返回“[2,7,9]”。
- 函数 tak：该迭代函数主要用于基准测试。其具体内容为：输入三个数值 x、y、z。若 y 小于 x，则返回 tak (tak (x-1) y z) (tak (y-1) z x) (tak (z-1) x y)；否则直接返回 z。
- 函数 tsumupto：输入三个数值 z、n、m。首先构造从整数 n 到整数 m、步长为 1 的顺序数列，求和后与初始值 z 相加，返回和值。
- 函数 foldfrom：输入数值 n、m，返回从整数 n 到整数 m、步长为 1 的顺序数列的元素求和值。与 tsumupto 的编写方式不同。
- 函数 hamdist：求取输入两列表（包括字符串）之间的汉明距离

3.3.3 真实应用

真实应用程序是模拟实际应用特征及行为设计的基准测试程序。它们相比小型基准测

试程序而言，具有更高的运算量级以及编写程序的复杂度。

在本课题中，真实应用程序的设计目的在于：在验证功能性的基础上，用于评估 GRIN（图规约）编译器与 GHC 编译器的性能比对结果。进而验证该处理器架构在运行函数式程序方面的优越性。

本课题中所设计的真实应用程序选取了 3 个较为常见的函数，分别为 detmat、queens、rsa。它们的功能简介如下（具体源码见附录 1）：

- 函数 detmat：输入方阵，返回方阵的行列式值。
- 函数 queens：n-皇后问题，即：将 n 个皇后按照一定规则放置在大小为 n×n 的棋盘上，规则要求不能在棋盘的任一行、任一列、任一斜线上同时出现两个皇后。函数输入 n，将返回 n-皇后问题的解的数目。
- 函数 rsa：RSA 加密算法。首先利用质数列表生成加密密钥，利用加密密钥对输入字符串进行 RSA 加密，即输入字符串，输出加密后的数字列表。

3.4 本章小结

本章对课题的编程语言 Wu、测试对象 Blackbird 处理器的组合子集架构 fun 以及设计的基准测试程序集作详细阐述。

对于函数式编程语言 Wu，本章分别给出它的基本特征、数据类型、语句结构、垃圾收集的介绍。对于组合子集架构 fun，本章不仅描述了数据在存储堆内的存储形式、不同存储单元的连接方式、图规约在架构内的实现过程，还对该处理器的微架构各部分组成与功能作了整体的说明。

对于本课题所采用的基准测试程序集，本章展示了它的按照运算量级别的划分，以及各自的设计目的、组成，同时对基本核函数、小型程序、真实应用中各测试程序的功能进行详细的描述。在此基础上我们将利用设计的测试集对该处理器进行具体的评估。

第四章 基准测试集性能评估

4.1 评估方法

4.1.1 评估内容

本课题的基准测试程序集源码在附录 1 中有详细描述，编写语言为 Wu。评估环境为操作系统 Ubuntu，具有 2048MB 大小的内存与 15GB 大小的外存。

本课题首先对所设计的所有程序进行功能性的验证，即保证所设计程序在 Blackbird 处理器下能够运行并产生正确的结果。这一部分的实验结果不在评估结果中作详细的阐述，读者可以使用附录 1 中的测试程序进行实验，实验结果理应与理论上的输出结果一致。

在功能性验证完成的基础上，本课题的主要评估目的包括两方面：一方面通过比较两种优化理论——组合子合并理论与态射理论采用前后的处理器运行效率，来证明优化理论对于函数式的 Blackbird 处理器在性能方面的提升；另一方面通过比较在 Blackbird 处理器与命令式处理器上分别运行采用同一算法的函数式编程程序，来证明该处理器在性能上的优势。

在优化理论的验证上，本课题首先对处理器采用组合子合并理论前后的性能进行评估。该评估过程将选取基准测试程序 fact、fib、sieve、tak、foldfrom、hamdist 作为评估对象，因为小型程序具有一定的运算量，可以看出性能提升的幅度。其次，本课题将对处理器采用态射理论前后的性能进行评估。评估过程将选取测试程序 foldr、map、filter、length、scale、partition，前 3 个函数为通用模块化函数，后 3 个函数分别为利用这些模块化函数进一步定义的高阶函数。对比项为采用 case 语句的定义、采用列表数据结构特征的定义与采用态射理论的定义，具体见附录 1。

最后，本课题会对比两种处理器下运行同一函数式编程算法的性能，他们各自实现的具体说明如下：

- 利用编程语言 Wu 编写程序，在 Blackbird 处理器的图规约编译器上运行，理论上使用时钟频率为 100MHz 的 Altera Cyclone IV FPGA，但实际测试在虚拟机上进行；
- 利用编程语言 Haskell 编写程序，在命令式处理器电脑的 GHC（Glasgow Haskell Compiler，为 Haskell 设计的编译器）上运行，使用时钟频率为 1.8GHz 的 Intel Core i5-8250U 处理器。

该评估过程将选取测试程序 detmat、queens、rsa 作为评估对象。

4.1.2 评估度量

对于前两种优化理论对于性能提升的测试，将采用与运行时间、存储占用相关的度量：

- 规约次数（Reuction counts）：运行程序时图规约编译器进行图规约的次数；
- 总周期数（Total cycles）：运行程序时规约、搜索以及其他操作的总运行周期数；
- 分配存储单元数（Allocated cells）：运行程序时图规约编译器分配的存储单元 cell 的数量。

我们将使用优化率来衡量优化效果的提升，优化率的计算公式为：

$$\text{优化率} = \text{度量差值}/\text{优化前度量值} \tag{4-1}$$

在比较不同处理器上的函数式程序的运行性能方面，我们将采用运行时间作为度量标准。其中，对于 GHC 利用其指令“:set +s”即可显示程序的运行时间；对于本课题的图规约编译器将使用公式进行计算：

运行时间 = 总周期数/FPGA 理论频率 (4-2)

4.2 评估结果

4.2.1 组合子合并前后的评估结果

使用组合子合并前后的评估结果如表 4-1 所示。经过整理的优化率柱状图如图 4-1 所示。程序 A、B、C、D、E、F 的具体描述分别为：A: fact 1000；B: fib 15；C: sieve [2..1000]；D: tak 15 10 5；E: foldfrom 1 1000；F: hamdist [2..1000] [1..1001] 0。

由表 4-1 所示评估结果可知，在规约次数、总周期数、分配存储单元数这 3 个指标的考量下，除了 tak 函数在分配存储单元数这一指标上的表现，通过合并组合子所取得运行性能几乎都优于采用基本组合子的情况。综合考量可得出以下结论：组合子合并可以从整体上（1）减少规约机的规约次数；（2）减少程序运行的总体时间；（3）减少存储空间的占用。

比对观察各程序在 3 种度量下的优化率，可以看出优化的程度各有不同，甚至还出现了负优化的情况。这是因为，在进行组合子合并时，部分合并后的组合子具有较强的函数依赖性，相应地，在这些函数上的优化程度则更为明显，而对于另一些程序则会产生较低的、甚至负值的优化率。从整体考量，该组合并组合子取得了比基本组合子更好的性能，但仍然具有进一步优化的空间。

比对规约次数和总运行时间的优化率可知，这两种指标在评估结果上趋于一致，并且后者的优化率总是高于前者，这说明：相比较加速规约行为，组合子合并对于加速规约以外的处理器行为（如搜索）具有更好的效果。另外，存储单元占用的优化率与前两种指标的优化率也具有一致性。

表 4-1 使用组合子合并前后的评估结果表

Func	Reduction counts			Total cycles			Allocated cells		
	Before	After	Rate	Before	After	Rate	Before	After	Rate
A	10095	8994	18.20%	40987	31976	21.99%	7001	6998	0.04%
B	21701	17753	18.19%	90775	70023	22.86%	16775	15785	5.90%
C	22	15	31.82%	88	44	50.00%	28	20	28.57%
D	199156	183634	7.79%	659353	599844	9.03%	219844	225010	-2.35%
E	21015	14006	33.35%	89063	45017	49.45%	19018	15009	21.08%
F	55032	45019	18.19%	213143	156035	26.79%	80026	69016	13.76%

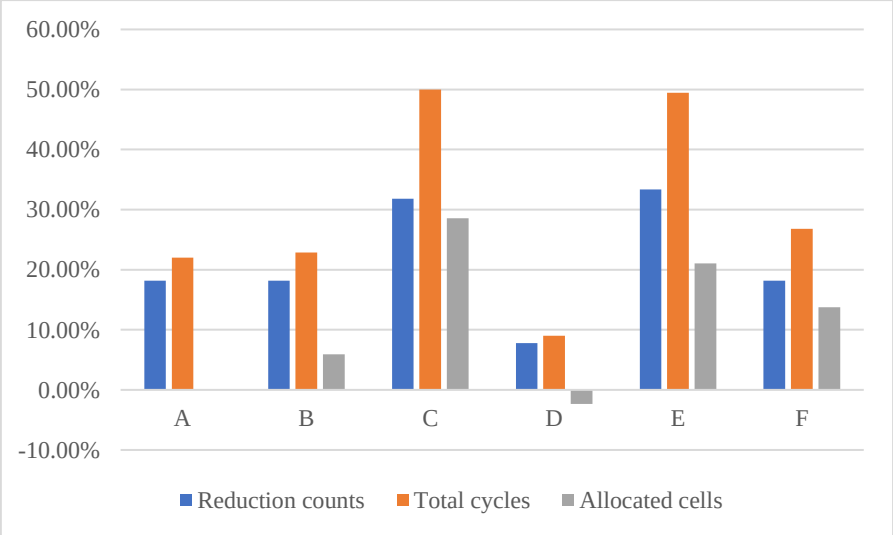

图 4-1 使用组合子合并的优化率比对图

4.2.2 不同编写方式的评估结果

以不同方式编写函数的评估结果如表 4-2 和图 4-2 所示，利用态射获得的优化率比对图如图 4-3 所示。其中 version1 代表使用 case 语句编写的函数，version2 代表使用列表数据结构的定义构造的函数，version3 代表使用 catamorphism 编写的函数。程序 A、B、C、D、E、F 的具体描述分别为：A: foldr * [1..100]；B: printlist (map succ [1..100])；C: filter (< 50) [1..100]；D: length [1..100]；E: printlist (scale [1..100] 5)；F: printlist ((partiton (<50) [1..100]) fst)。

观察评估结果表明，无论是哪个测试函数，其在三种测量指标下的 version1 版本的测试结果总是最不理想的；相比之下，它的 version2 和 version3 相差较小，且都能反映较好的优化效果。

比较 version2 和 version3，两者虽然相差较小，但在函数 foldr 以及以函数 foldr 构造的函数 length 方面，version3 的函数运行都将花费更少的时间，并且占用更少的存储；而在其余函数运行时，version2 均具有更优的评估结果。这一评估结果可以由态射理论的原理进行解释，即因为 version3 是使用以 catamorphism 形式编写的函数，而 catamorphism 在分类理论的定义即是将某一数据结构类型转换为某一类型的数值，这与函数 foldr、length 的作用是一致的，因此在这两者上可以取得更好的效果；而其余函数的功能是将一个数据结构类型转换为另一个数据结构类型（两结构类型可以相同），这与 catamorphism 并不一致，在态射理论中与其一致的是 hylomorphism，因此效果略差。可以预料的是，当使用 hylomorphism 时，其性能应优于此处的 version2。观察图 4-3 可知，各基准测试函数均能通过态射理论的优化方式实现很高的优化率。

表 4-2 使用不同编写方式的评估结果表

Func	Reduction counts			Total cycles			Allocated cells		
	Version1	Version2	Version3	Version1	Version2	Version3	Version1	Version2	Version3
A	3413	1911	1811	11237	7836	6232	3822	2515	2115
B	3514	2014	2315	11937	7035	7839	3619	2515	2717
C	1679	973	1069	5385	3475	3658	1792	1335	1327
D	2915	2211	2111	9842	7839	6235	3622	2515	2115
E	3514	2014	2315	11937	7035	7839	3619	2515	2717
F	3920	2568	2622	12454	8251	8660	4080	3075	3180

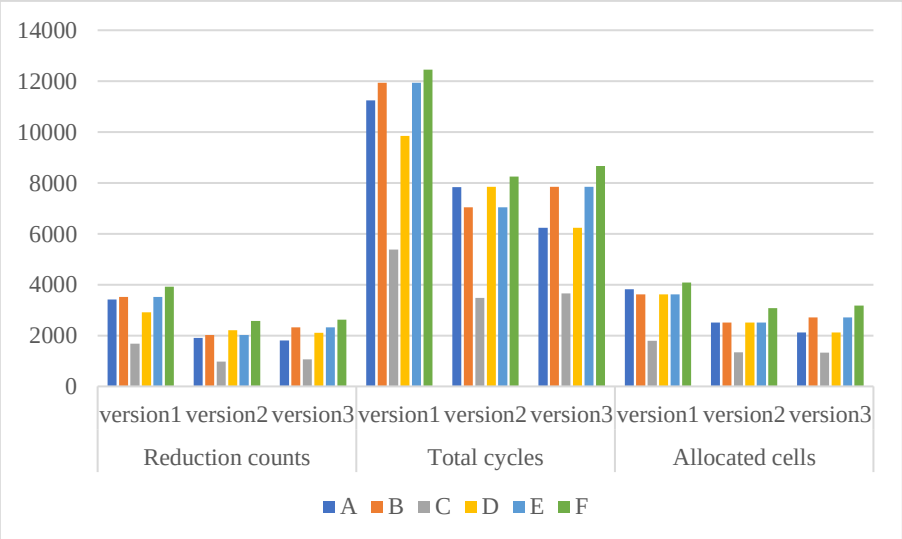

图 4-2 使用不同方式编写方式的评估结果图

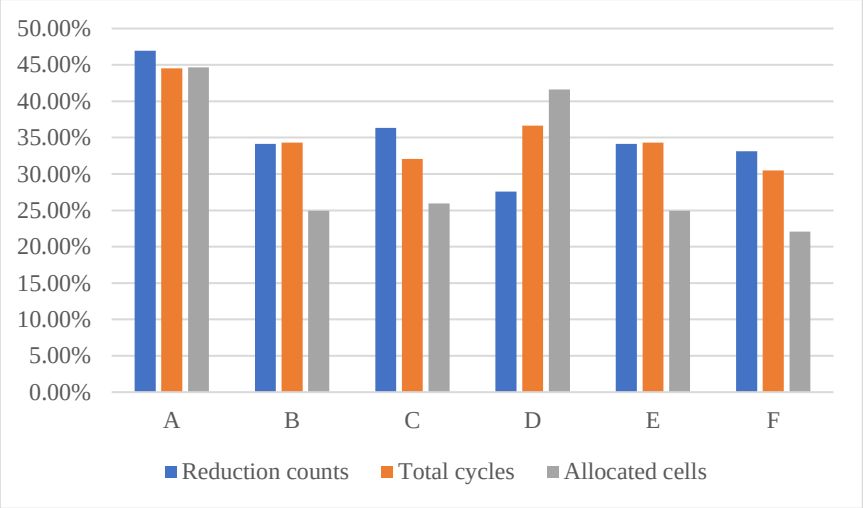

图 4-3 态射理论的优化率比对图

4.2.3 图规约编译器与 GHC 的评估结果

使用图规约编译器与 GHC 的评估结果如表 4-3 所示。

观察表 4-3 我们可以看出，在函数式程序的运行时间方面，比较采用 GHC 的命令式架构执行函数式程序而言，本课题所采用函数式架构在运行同一算法时，在运行时间方面的所实现的优化率分别为 90.98%、55.34%、93.40%。由此可见，函数时架构下的函数式程序执行速度将快于命令式架构下的函数式程序执行速度。

在讨论该结论时，我们也需要考虑该优化率的误差因素：（1）实际的操作系统的运行会在一定程度上使得后者的实验数据偏大于理想情况；（2）GHC 编译器内置的读取函数运行时间命令所产生的结果的精度。

表 4-3 使用图规约编译器与 GHC 的评估结果表

Name	GRIN	GHC	Optimization rate
detmat(5×5)	0.001805s	0.02s	90.98%
8-queens	0.084862s	0.19s	55.34%
rsa	0.0033s	0.05s	93.40%

4.3 本章小结

本章对根据课题的目的所设计测试程序集进行评估，首先介绍了评估内容，即对两种优化方式的综合评估以及与命令式架构下的 GHC 进行的比较式评估，同时适当地选取了测试程序集中的部分函数以获取测试结果。之后，对评估所采用的度量进行介绍。

在完成评估方法的说明后，本课题对评估所得结果进行了展示，并作出解释。从结果可知，组合子合并优化与态射优化都将较为明显地提高 Blackbird 处理器的运行效率；GRIN 编译器能够比 GHC 更快速地运行函数式程序。

第五章 结论

本文系统阐述了函数式编程语言的优势，并对与它密切相关的组合子理论、图规约编译器作了详细的说明。同时，我们展示了两种在函数式编程领域中可行的优化理论——组合子合并理论与态射理论，为后续的评估过程提供了理论基础。在此基础上，我们又对本课题的评估对象——Blackbird 处理器作了深入的介绍。Blackbird 处理器是一款由 Cecil Accetti 设计的、基于函数式编程理念搭建的函数式架构，其具有组合子集架构 fun 与适配的函数式编程语言 Wu。

在介绍完以上背景知识后，我们根据本课题的需求——评估 Blackbird 处理器的功能性这一目的，给出本课题所设计的基准测试程序集，它主要包括基本核函数、玩具函数、真实应用三种类型，其划分标准为应用范围和运算量。之后我们便使用测试程序集对处理器进行实际的评估。评估主要验证：（1）该处理器是否能够正确运行程序并给出正确的结果；（2）该处理器在优化理论下是否能够真正实现优化以及相应的优化程度；（3）在函数式程序的运行性能方面，该处理器是否能取得比现行的、在命令式架构下实现的 GHC 编译器更好的效果。

本课题最终所得的评估结果表明，采用组合子合并理论和态射理论对原有组合子集以及函数编写方面的优化是显而易见的，两种方法不仅能够实现该处理器在程序程序时间方面的缩减，同时也能够减少对处理器存储空间的占用。对比在本课题的图规约编译器与在 GHC 的运行结果表明，在函数式程序的运行速度方面，函数式架构比命令式架构更加具有竞争力。

在利用态射理论对该测试集进行的优化方面，仍然具有较大的优化潜力。本课题仅应用 catamorphism 对测试程序集中的 foldr 类函数进行优化，其效果已经十分显著；如果在此基础上利用其它态射对其余不同的类型转换类的函数进行优化，理论上也会取得相当的性能上的提升。

在测试集的优化方面，当前测试集中的大部分函数仍然采用利用数据结构定义的方式进行编写，并没有完全发挥态射理论的优化能力。除此以外，该处理器的核尚未完备，例如对于文件的处理、对于浮点数的处理还没有相应的配置和定义，这些优化需要在编译器、微架构层面有所改动，例如对于浮点数编译方式的设计、对于存储空间单元长度的调整等等，因此该处理器仍然具有完善的空间。

在函数式编程的前景方面，函数式架构的优势在于并发并行，但当前主流使用的函数式编程在命令式架构下执行，因此最终在计算机底层的执行也仍然是命令式的，所以与命令式架构下的命令式编程仍然难以在运算性能上相提并论。在函数式架构的理论和框架设计得以完备后，在该架构下实现的函数式编程理应获取比命令式架构下实现的函数式编程更好的性能，而在某些需要并行处理的应用或许能实现比命令式架构下的命令式编程更好的性能。

参考文献

[1] Kocher P, Genkin D, Gruss D, et al. Spectre Attacks: Exploiting Speculative Execution[J]. 2018.

[2] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space[R]. 2018.

[3] Chisnall, David. C Is Not a Low-level Language[J]. Queue, 2018, 16(2):18-30.

[4] Hammarlund P, Martinez A J, Bajwa A A, et al. Haswell: The Fourth-Generation Intel Core Processor[J]. IEEE Micro, 2014, 34(2):6-20.

[5] Cecil Accetti. Instructions Considered Harmful: Design of a Lazy, Functional Programming Architecture. (unpubl. MS).

[6] Hinsen K . The Promises of Functional Programming[J]. “Research Topics in Functional Programming” ed. 1990: 17–42.

[7] John Hughes. Why Functional Programming Matters[J]. The Computer Journal(2):2.

[8] Clarke T J W , Gladstone P J S , Maclean C D , et al. [ACM Press the 1980 ACM conference - Stanford University, California, United States (1980.08.25-1980.08.27)] Proceedings of the 1980 ACM conference on LISP and functional programming - LFP "80 - SKIM - The S, K, I reduction machine[J]. 1980:128-135.

[9] Cecil Accetti. Making an ISA with Morphisms. (unpubl. MS).

[10] Boquist, U. Code optimization techniques for lazy functional languages[D]. Doktorsavhandlingar vid Chalmers Tekniska Hogskola 1999 (1495): 1-331

[11] J.Hughes. Supercombinators - A new implementation method for applicative languages[J]. LFP '82: Proceedings of the 1982 ACM symposium on LISP and functional programmingAugust 1982: 1–10.

[12] E. Meijer, M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire[J]. Springer-Verlag, 1991: 124–144.

[13] csabahruska. grin-compiler/ghc-grin[EB/OL]. https://github.com/grin-compiler/ghc-grin/tree/master/ghc-grin-benchmark/boq-custom.

附录 1：基准测试程序集

```
// prelude.fn
// INFIX OPERATIONS :    +          BUILT-IN ADD
//                       -          BUILT-IN SUB
//                       *          BUILT-IN MUL
//                       /          BUILT-IN DIV
//                       %          BUILT-IN REM
//                       >          BUILT-IN GREATER-THAN
//                       <          BUILT-IN LESS-THAN
//                       ==         BUILT-IN EQUAL
//                       ||         BUILT-IN LOGIC OR
//                       &&         BUILT-IN LOGIC AND
//                      :          BUILT-IN CONS
//                       >>         BUILT-IN SRL
//                       <<         BUILT-IN SLL
//                       .          compose
//                       ++         concat
//                       ( _ , _ ) CONS (TUPLE)

// CONSTANTS :   true
//               false
//               []

// COMBINATORS
fun I a = a;
fun K a b = a;
fun S a b c = a c (b c);
fun KI a b = b;
fun C1 a b c d = a ( b d ) c;
fun S1 a b c d = a ( b d ) (c d);
fun D a b c d = a b ( c d );
fun C a b c = (a c) b;
fun B a b c = a (b c);

//BASIC COMBINATORS
fun id x = x;                    // I COMBINATOR
fun const x y = x;               // K COMBINATOR
fun skip x y z = x;              // B5
fun skip2 x y z = y;             // KK
```

```
fun distribute x y z = (x z) ( y z);      // S
fun flip f x y = (f y) x;          // C
fun compose f g x = f (g x);    // B, CAN BE CALLED INFIX AS . -> foo . bar 2 = foo (bar 2)
fun double f x = f (f x );        // W
fun apply x y =    y x ;              // T
fun unconsf = Ds; // -> a b (d c);    D*
fun composeflip a b c = a(c b); // Q1
fun tri a b c d = d a b c; // Vp

//---------------
//      Bool data
//---------------
//Constructors
fun true = K;
fun false = KI;
//Operations
fun and a b = a b a;
fun or a b    = a a b;
fun not x = x false true;
fun xor x y = x (not y) y;

//--------------------
// Numeric Operations
//--------------------
//Operations
fun abs x = (> 0 x) (- 0 x) x;
fun max x y = (< x y) y x;
fun min x y = (< x y) x y;
funrec pow x n = (== 0 n) 1 (* x (self x (- n 1)));
fun sqr x = pow x 2;
fun cube x = pow x 3;
fun gcd x y = gcd_abs (abs x) (abs y);
funrec gcd_abs x y = (== 0 y) x (self y (% x y));
fun succ = + 1;

//---------------
//    List data
//---------------
//Constructors
fun [] = \f -> \g -> f;
fun cons a b = \f -> \g -> g a b;
//Deconstructors
fun isnull xs = xs true (\h -> \t -> false);
fun head xs = xs (err "head []") true;
```

```
fun tail xs = xs (err "tail []") false;
funrec init xs = xs (err "init []") (\h -> \t -> (isnull t) [] (h : (self t)));
funrec last xs = xs (err "last []") (\h -> \t -> (isnull t) h (self t));
//fmap and morphisms
//l_fmap: functorList
fun l_fmap f xs = xs [] (\h -> \t -> h : (f t));
//l_cata: CATA functorList
fun l_cata g = CATA l_fmap g;
//foldr, foldl
funrec foldr f z xs = case
                        | isnull xs @ z
                        | _ @ f (head xs) (self f z (tail xs))
                        ;
funrec foldr2 f z xs = xs z (\h -> \t -> f h (self f z t));
fun foldr3 f z = l_cata (\x -> x z f);
fun foldr4 f z = FIX (\n -> \x -> x z (\ a -> \b -> f a (n b)));
funrec foldl f z xs = case
                        | isnull xs @ z
                        | _ @ self f (f z (head xs)) (tail xs)
                        ;
funrec foldl2 f z xs = xs z (\h -> \t -> self f (f z h) t);
fun foldl3 k b mu = foldr3 (\a -> \f -> (. f (C k a))) I mu b;
//map
funrec map f xs = case
                    | isnull xs @ []
                    | _ @ ((f ( head xs)) : (self f (tail xs)))
                    ;

funrec map2 f xs = xs [] (\h -> \t -> ((f h) : (self f t)));
fun map3 f = l_cata (\x -> x [] (\h -> \t -> (f h) : t));
//filter
funrec filter cond xs = case
                          | isnull xs @ []
                          | _ @ (cond (head xs)) ((head xs) : (self cond (tail xs))) (self cond (tail
xs))
                          ;
funrec filter2 cond xs = xs [] (\h -> \t -> (cond h) (h : (self cond t)) (self cond t));
fun filter3 cond = l_cata (\x -> x [] (\h -> \t -> (cond h) (h : t) t));
fun filter4 cond = FIX (\n -> \x -> x [] (\h -> \t -> (cond h) (h: (n t)) ((n t))));
//scanl
funrec scanl f z xs = case
                        | isnull xs @ (z : [])
                        | _ @ (z : (self f (f z (head xs)) (tail xs)))
                        ;
```

```
funrec scanl2 f z xs = xs (z : []) (\h -> \t -> (z : (self f (f z h) t)));

//zipWith
funrec zipWith f xs1 xs2 = case
                           | (or (isnull xs1) (isnull xs2)) @ []
                           | _ @ ((f (head xs1) (head xs2)) : (self f (tail xs1) (tail xs2)))
                           ;

funrec zipWith2 f xs1 xs2 = (or (isnull xs1) (isnull xs2)) [] ((f (head xs1) (head xs2)) : (self f (tail xs1) (tail xs2)));

//iterate
funrec iterate f x = x : (self f (f x));

//Operations
funrec ones = 1 : self;
funrec ones_pn = 1 : (- 0 1) : self; //[1,-1,1,-1,...]
funrec from n = n : (self (succ n));
funrec range n m = (> n m) [] (n : (self (succ n) m));
fun length = foldr4 (\x -> \n -> succ n) 0;
fun sum = foldr4 + 0;
fun prod = foldr4 * 1;
fun reverse xs    = (((foldl3 (flip (:))) []) xs);
fun average = \xs -> / (sum xs) (length xs);
fun foldmap f g z xs = foldr4 f z (map3 g xs);
fun zip = zipWith2 tuple;
fun concat xs ys = foldr4 (:) ys xs;
funrec take xs n = xs [] (\h -> \t -> (or (== 0 n) (> 0 n)) [] (h : (self t (- n 1))));
funrec drop xs n = xs [] (\h -> \t -> (or (== 0 n) (> 0 n)) xs (self t (- n 1)));
fun partition cond xs = ((filter4 cond xs) , (filter4 (. not cond) xs));
fun split xs n = xs [] (\h -> \t -> ((take xs n) , (drop xs n)));
funrec dropWhile cond xs = xs [] (\h -> \t -> (cond h) (self cond t) xs);
funrec takeWhile cond xs = xs [] (\h -> \t -> (cond h) (h : (self cond t)) []);
fun andList = foldr4 (&&) true;
fun orList = foldr4 (||) false;
fun elem x = any (== x);
funrec repeat x = x : (self x);
fun all cond xs = andList (map3 cond xs);
fun any cond xs = or (map3 cond xs);

//-------------------
// Vector Operations
//-------------------
funrec index xs n = xs (err "out of range") (\h -> \t -> (== 0 n) h (self t (- n 1)));
```

```
fun first = head;
fun second xs = index xs 1;
fun third xs = index xs 2;
fun vecdotprod xs1 xs2 = foldr4 + 0 (zipWith2 * xs1 xs2);
fun veccrossprod xs1 xs2 = (\a -> \b -> \c -> \e -> \f ->\g -> (- (* b g) (* c f)) : (- (* c e) (* a g)) : (- (* a f) (* b e)) : []) (first xs1) (second xs1) (third xs1) (first xs2) (second xs2) (third xs2);
fun scale xs n = map3 (* n) xs;
fun vecmodulo = foldmap + sqr 0 ;
fun vecplus = zipWith2 +;
fun vecminus = zipWith -;

//-------------------
// Matrix Operations
//-------------------
//Operations
fun matsize xss = ((length xss) : (length (head xss)) : []);
fun isSingle xss = (\size -> (and (== 1 (first size)) (== 1 (second size)))) (matsize xss);
funrec transpose xss = (isnull (head xss)) [] ((map3 (\xs -> head xs) xss) : (self (map3 (\xs -> tail xs) xss)));
funrec matprod_line xs xss = map3 (vecdotprod xs) (transpose xss);
funrec matprod xss1 xss2 = xss1 [] (\h -> \t -> (map3 (vecdotprod h) (transpose xss2)) : (self t xss2));
funrec matremoveline xss n = xss [] (\h -> \t -> (== 0 n) t (h : (self t (- n 1))));

//----------
//      Sort
//----------
funrec quickSort xs = xs [] (\h -> \t -> ++ (self ((partition (\x -> < x h) t) fst)) (h : (self ((partition (\x -> < x h) t) snd))));

funrec trins rev xs1 xs2 = xs1 (self [] (++ (reverse rev) ((head xs2):[])) (tail xs2)) (\h1 -> \t1 -> xs2 (++ (reverse rev) xs1) (\h2 -> \t2 -> (< h1 h2) (self (h1 : rev) t1 xs2) (self [] ( (++ (reverse rev) (h2 : xs1))) t2)));
fun insertSort xs = trins [] ((head xs) : []) (tail xs);

funrec merge xs1 xs2 = xs1 xs2 (\h1 -> \t1 -> xs2 xs1 (\h2 -> \t2 -> (or (< h1 h2) (== h1 h2)) (h1 : (self t1 xs2)) (h2 : (self xs1 t2))));
funrec mergeSort xs = xs [] (\h -> \t -> (isnull t) xs ((\sp -> merge (self (sp fst)) (self (sp snd))) (split xs (/ (length xs) 2))));

//---------------
//      Tuple data
//---------------
//Constructors
fun tuple = \a -> \b -> \f -> f a b;
```

```
//Deconstructors
fun fst = K;
fun snd = KI;
//Operations
fun tupleswap t = ((t snd) , (t fst));
fun curry f x y = f (x , y);
fun uncurry f t = f (t fst) (t snd);

//---------
//    ASCII
//---------
funrec itoa_ n = (> 1 n) [] ((+ 48 (% n 10)) : (self (/ n 10)));
fun itoa n = (> 0 n) (++ (45 , []) (reverse itoa_ (- 0 n))) (reverse (itoa_ n));
fun toUpper ch = + ch 32;
fun toLower ch = - ch 32;
fun isAlpha x = (or (and (< 65 x) (> 90 x)) (and (< 97 x) (> 122 x)));

//---------------------------
// maybe monad and functions
//---------------------------
//Constructors
fun Nothing = \nothing -> \just -> nothing;
fun Just a = \nothing -> \just -> just a;
//Operations
fun fmapMaybe f a = a (Nothing)(Just);
fun maybeC n f = CATA fmapMaybe (\a -> a (n) (f));
fun pure = Just;
fun applyM f x = maybeC Nothing x f;
//TODO TESTAR applyM
fun >>= m f = m (Nothing) f;
fun => a1 a2 = >>= a1 (\x -> a2);

//---------------
//      Basic IO
//---------------
let UART = #00001000;
let GPIO = 0;
let stdout = GPIO;
//Operation
fun printbool x = x (print "True") (print "False")
fun printMaybe x = x (print "Nothing") (>>=    (print "Just") (OUT GPIO));
fun err xs = EXC (print xs );
fun put x = (OUT GPIO) x;
fun putc x = (OUT UART) x;
```

```
fun print    = foldr3    (OUT UART) (I);
fun printlist = FIX (\n -> \x -> x I (\ a -> \b -> (OUT GPIO a) (n b) ));
fun printl = foldr3 (OUT UART) (OUT UART 10);

end


// fact.fn
funrec fact n =    (< n 2) 1 (* n (self (- n 1)));

main = put (fact 10) true


// fib.fn
funrec fib n = (> 2 n) 1 (+ 1 (+ (self (- n 1)) (self (- n 2))));
funrec linfib x y n = (== 0 n) y (self y (+ x y) (- n 1));

main = put (fib 5) true


// foldfrom.fn
fun foldfrom n m = HYLO l_fmap (\x -> x 0 +) (\a -> (< m a) [] ( a : (+ 1 a)) ) n;

main = put (foldfrom 1 1000) true


// hamdist.fn
funrec hamdist xs1 xs2 d = xs1 (xs2 d (\h -> \t -> + d (length xs2))) (\h1 -> \t1 -> xs2 (+ d (length
xs1)) (\h2 -> \t2 -> (== h1 h2) (self t1 t2 d) (self t1 t2 (+ 1 d))));

let str1 = "astronomy";
let str2 = "astrology";
main = put (hamdist str1    str2 0) true


// sieve.fn
fun upto n m = FIX (\ana -> \a -> (< m a) [] (a : (ana (+ 1 a)))) n;
funrec xfilter x xs = xs [] (\h -> \t -> (== 0 (% h x)) (self x t) (h : (self x t)));
funrec sieve xs = xs [] (\h -> \t -> (h : (self (xfilter h t))));

let l1 = (2:3:4:5:6:[]);
main = printlist (sieve l1)
```

```
// tak.fn
funrec tak x y z = (< y x) (self (self (- x 1) y z) (self (- y 1) z x) (self (- z 1) x y)) z;

main = put (tak 18 12 6) true


// tsumupto.fn
fun upto n m = FIX (\ana -> \a -> (< m a) [] (a : (ana (+ 1 a)))) n;
funrec xsum n xs = + n (sum xs);
fun tsumupto z n m = xsum z (upto n m);

main = put (tsumupto 0 1 100) true


// detmat.fn
funrec matremoveline xss n = xss [] (\h -> \t -> (== 0 n) t (h : (self t (- n 1))));
funrec submat_unremove xss = xss [] (\h -> \t -> ((tail h) : (self t)));
fun submat xss n = matremoveline (submat_unremove xss) n;
fun submatvec xss = foldmap (:) (\x -> submat xss x) [] (range 0 (- (length xss) 1));
funrec detmat xss = (isnull xss) 0 ((isSingle xss) (head (head xss)) (sum (zipWith2 (*) (zipWith2 (*)
(map3 (\xs -> head xs) xss) (take ones_pn (length xss))) (map3 self (submatvec xss)))));

let m1 = ((1:2:3:4:5:[]):(2:3:4:5:6:[]):(3:4:5:6:7:[]):(4:5:6:7:8:[]):(5:6:7:8:9:[]):[]);
main = detmat m1


// queens.fn
fun solve nq = length (generate nq nq);
funrec generate nq n = (== 0 n) ([]:[]) (concatMapAddOne nq (self nq (- n 1)));
funrec concatMapAddOne nq xs = xs [] (\a -> \b -> ++ (addOne nq a) (self nq b) );
fun addOne nq xs = filterOk (mapCons xs (upto 1 nq));
funrec filterOk xs = xs [] (\a -> \b -> (ok a) (a : self b) (self b));
funrec mapCons xs ys = ys [] (\a -> \b -> (a : xs) : (self xs b ) );
fun ok xs = xs true (\a -> \b -> safe a 1 b );
funrec safe x d ys = ys true (\a -> \b -> and (and (and (not(== x a)) (not(== x (+ a d)))) (not(== x (-
a d)))) (self x (+ d 1) b));
fun upto n m = FIX (\ana -> \a -> (< m a) [] (a : (ana (+ 1 a)))) n;

main = put (solve 8) true


// rsa.fn
fun upto n m = FIX (\ana -> \a -> (< m a) [] (a : (ana (+ 1 a)))) n;
funrec xfilter x xs = xs [] (\h -> \t -> (== 0 (% h x)) (self x t) (h : (self x t)));
```

```
funrec sieve xs = xs [] (\h -> \t -> (h : (self (xfilter h t))));
fun totient pr1 pr2 = * (pr1 - 1) (pr2 - 1);

fun myE tot = head (filter3 (\n -> (== 1 (gcd n tot))) (range 2 (tot - 1)));
fun myD e n phi = head (filter3 (\d -> (== 1 (% (* e d) phi))) (range 1 n));

fun rsa_encode n e numbers = map3 (\nm -> (% (pow nm e) n)) numbers;
// fun rsa_decode d n ciphers = map3 (\c -> % (pow c d) n) ciphers;

let primes = take (sieve (from 2)) 90;
let str = "abcd efg 1234567 hijk lmn";

let p1 = last primes;
let p2 = (last (init primes));
let tots = totient p1 p2;

let E = myE tots;
let N = * p1 p2;
let rsa_encoded = rsa_encode N E str;
// let D = myD E N tots;
// let rsa_decoded = rsa_decode D N rsa_encoded;

main = printlist rsa_encoded
// main = printl rsa_decoded


// detmat.hs
sPermutations :: [a] -> [([a], Int)]
sPermutations = flip zip (cycle [1, -1]) . foldl aux [[]]
  where
    aux items x = do
      (f, item) <- zip (cycle [reverse, id]) items
      f (insertEv x item)
    insertEv x [] = [[x]]
    insertEv x l@(y:ys) = (x : l) : ((y :) <$>) (insertEv x ys)

elemPos :: [[a]] -> Int -> Int -> a
elemPos ms i j = (ms !! i) !! j

prod
  :: Num a
  => ([[a]] -> Int -> Int -> a) -> [[a]] -> [Int] -> a
prod f ms = product . zipWith (f ms) [0 ..]
```

```
sDeterminant
   :: Num a
   => ([[a]] -> Int -> Int -> a) -> [[a]] -> [([Int], Int)] -> a
sDeterminant f ms = sum . fmap (\(is, s) -> fromIntegral s * prod f ms is)

determinant
   :: Num a
   => [[a]] -> a
determinant ms =
   sDeterminant elemPos ms . sPermutations $ [0 .. pred . length $ ms]

permanent
   :: Num a
   => [[a]] -> a
permanent ms =
   sum . fmap (prod elemPos ms . fst) . sPermutations $ [0 .. pred . length $ ms]

-- TEST ------------------------------------------------------------------------
result
   :: (Num a, Show a)
   => [[a]] -> String
result ms = unlines [ show (determinant ms)]

main :: IO ()
main = (putStrLn . result) [[1, 2, 3, 4, 5], [2, 3, 4, 5, 6], [3, 4, 5, 6, 7], [4, 5, 6, 7, 8], [5, 6, 7, 8, 9]]


// queens.hs
--module Main(main) where

main = print (solve 8)

solve :: Int -> Int
solve nq = length (generate nq nq)

generate :: Int -> Int -> [[Int]]
generate nq 0 = [[]]
generate nq n = concatMap (add_one nq) (generate nq (n-1))

add_one :: Int -> [Int] -> [[Int]]
add_one nq xs = filter ok (map (:xs) (upto 1 nq))

ok :: [Int] -> Bool
ok []         = True
```

```
ok (x:xs) = safe x 1 xs

safe :: Int -> Int -> [Int] -> Bool
safe x d []         = True
safe x d (y:ys) = (x /= y) && (x /= y+d) && (x /= y-d) && safe x (d+1) ys

upto :: Int -> Int -> [Int]
upto m n = if m > n then
                    []
                else
                    m : upto (m+1) n

// rsa.hs
module RSAMaker where
import Data.Char ( chr )

encode :: String -> [Integer]
encode s = map (toInteger . fromEnum ) s

rsa_encode :: Integer -> Integer -> [Integer] -> [Integer]
rsa_encode n e numbers = map (\num -> mod ( num ^ e ) n ) numbers

divisors :: Integer -> [Integer]
divisors n = [m | m <- [1..n] , mod n m == 0 ]

isPrime :: Integer -> Bool
isPrime n = divisors n == [1,n]

totient :: Integer -> Integer -> Integer
totient prime1 prime2 = (prime1 - 1 ) * ( prime2 - 1 )

myE :: Integer -> Integer
myE tot = head [n | n <- [2..tot - 1] , gcd n tot == 1]

main = do
    let primes = take 90 $ filter isPrime [2..]
        p1         = last primes
        p2         = last $ init primes
        tot       = totient p1 p2
        e          =    myE tot
        n      = p1    * p2
        rsa_encoded    =    rsa_encode n e $ encode "abcd efg 1234567 hijk lmn"
        encrypted = concatMap show rsa_encoded
    putStrLn ("Encrypted: " ++ encrypted )
```

谢辞

关于毕业设计，首先必须要感谢的是我的指导老师刘佩林教授，感谢她在毕设选题与过程中的悉心帮助和富有价值的建议和指导。还需要十分感谢在毕业设计过程中负责与我对接的学长 Cecil Accetti，感谢他所搭建的 Blackbird 处理器、所实现的图规约编译器、所设计的函数式编程语言 Wu；感谢他在我需要帮助的时候及时给予，尽管在疫情期间，他也愿意抽出夜晚和周末的时间通过视频的方式面对面向我讲解和解答；感谢他为我提供需要参考的文献以及可以加以利用的 Haskell 代码的网址；感谢他的鼓励与幽默，缓解了我在毕业设计过程中的担心，促使我更有信心。

另外，还要感谢肖邦的夜曲、赫尔曼的《荒原狼》、伍尔夫的《到灯塔去》、《Rick & Morty》第四季以及英雄联盟的 2020 年春季赛等等，感谢它们为我的宅家毕设之旅提供了众多可以暂时停靠休憩的港湾。最后的最后，还要感谢我的女友，感谢她使得毕设中每一个分享与交心的夜晚都变得如此独特和美妙。

BENCHMARK PROGRAM DESIGN OF BLACKBIRD CPU

The mainstream architecture of modern computers is the von Neumann architecture. The stored programs in this architecture are mainly instructions that are sequentially executed by the processing unit. The architecture has different levels of storage areas, including registers (program counters, instruction registers, register files), cache areas, and main memory. At first, the work efficiency of the imperative architecture was not high. The introduction of the pipeline improved its work efficiency by implementing instruction-level parallelism. The introduction of out-of-order execution technology and speculative execution technology further improved the execution efficiency of the pipeline. To support this feature, the compiler needs to be extended to generate code that can be executed out of order and implement branch prediction. This increase in performance is accompanied by increasingly complex software and microarchitecture. Imperative programming languages require programs to be executed in order, because the order of execution will affect the state of variables and storage space. Therefore, when executing out-of-order instructions, the processor needs to track the original execution order of the operation to ensure that the state of the program is the same as the state of the program in the sequential execution machine. State is very important during speculative execution, especially when the computer must retreat to the original state and discard the result of the out-of-order operation when the program goes to a branch that violates the prediction. However, the cache area in the mainstream architecture is not within the operating range of returning to this state, so it brings security risks to the computer, such as side channel attacks. The disadvantages of modern computer architecture can be summarized as: (a) The architecture is too complicated, and a large number of storage units are mainly used to store copies of data required for speculative execution; (b) Even if the C language is still the main world today One of the languages used, the architecture and the C language no longer form a mapping relationship; (c) malicious agents can easily use instructions to access and modify the state, and thus directly control the computer execution process.

Compared to instructive programming, functional programming has its merits in terms of development. On the one hand, the features of functional programming such as statelessness, no side effects, store-and-use and delete, and lazy call of actual parameters provide a theoretical basis for its development in security (in contrast, out-of-order execution The imperative architecture does not respond well to some modern security issues, such as side-channel attacks); on the other hand, the advantages of functional programming in concurrent parallel programming have paid attention to its development potential in large-scale parallel applications. New functional languages (such as Sun's Fortress and Microsoft's F#) have been developed in succession. In addition, functional programs are more concise and efficient than imperative programs; the development of functional hardware architecture will also promote the further development of functional programming languages.

On the premise of ensuring the advantages and programmability of storing programs for

calculation, a stateless processor Blackbird CPU designed based on the functional programming concept was proposed exploratively. It mainly avoids the security problems in modern computers by restricting the state, such as data leakage and side channel attacks. Blackbird CPU is a processor designed based on the concept of functional programming. It has a combinator set architecture called fun, an RTL model of a functional processor, and a compiler that supports the functional programming language Wu. Wu is a low-abstraction level, strict, functional system programming language designed for this architecture.

Since the functional programming language Wu is a newly developed functional programming language, there is no benchmark test program written in it. In order to comprehensively evaluate the running performance of the Blackbird CPU, the main content of this subject is to write the corresponding benchmark Test set and make corresponding test analysis.

This subject conducts a comprehensive evaluation of the Blackbird CPU, that is, to define and design a set of suitable benchmark programs for the architecture of the Blackbird processor and fun combinator set architecture. The final design benchmark programs include basic kernel programs, toy programs and real applications. In addition, a variety of optimization methods are used in the process of writing benchmark programs, including the optimization of the selection of the combinator set, the optimization of the writing method using the definition of the data structure, and the optimization of the writing method using the morphism principle. Metrics used to evaluate computing performance include the reductions counts, the total cycles, and the allocated cells in the program. The benchmark programs design and testing environment of the hardware structure is Ubuntu. The results show that several optimization methods are used to reduce the computational load of different programs to different degrees. Among them, the most obvious optimization effect is the program written by combining the optimized subsets and using the morphism principle. Finally, the subject compares the performance differences between GRIN compiler and GHC in running functional programs, and draws the conclusion that the former has more advantages.

The main work of this article includes: Systematic description of the features of the functional programming language, the basic definition of combinators, and the basic implementation principles of the graphical protocol compiler; Introduction to combinatorial merger theory and morphism theory in optimization; Introduction to the functional programming language Wu and a brief introduction to the combined subset architecture fun; Design of benchmark programs suite, including basic kernel function, small function, real application; Results of benchmark programs testing and corresponding analysis.

Because the design of different types of programs using different morphisms requires a certain mathematical foundation, only the optimization of test programs using catamorphism is completed. Most test programs still use the method of writing data structure definitions and still have room for optimization. In addition to this, the operational performance advantages of this functional computing architecture for large parallel programs such as neural networks have yet to be verified in experiments. The module for file import and floating point processing in the architecture also needs to be improved.