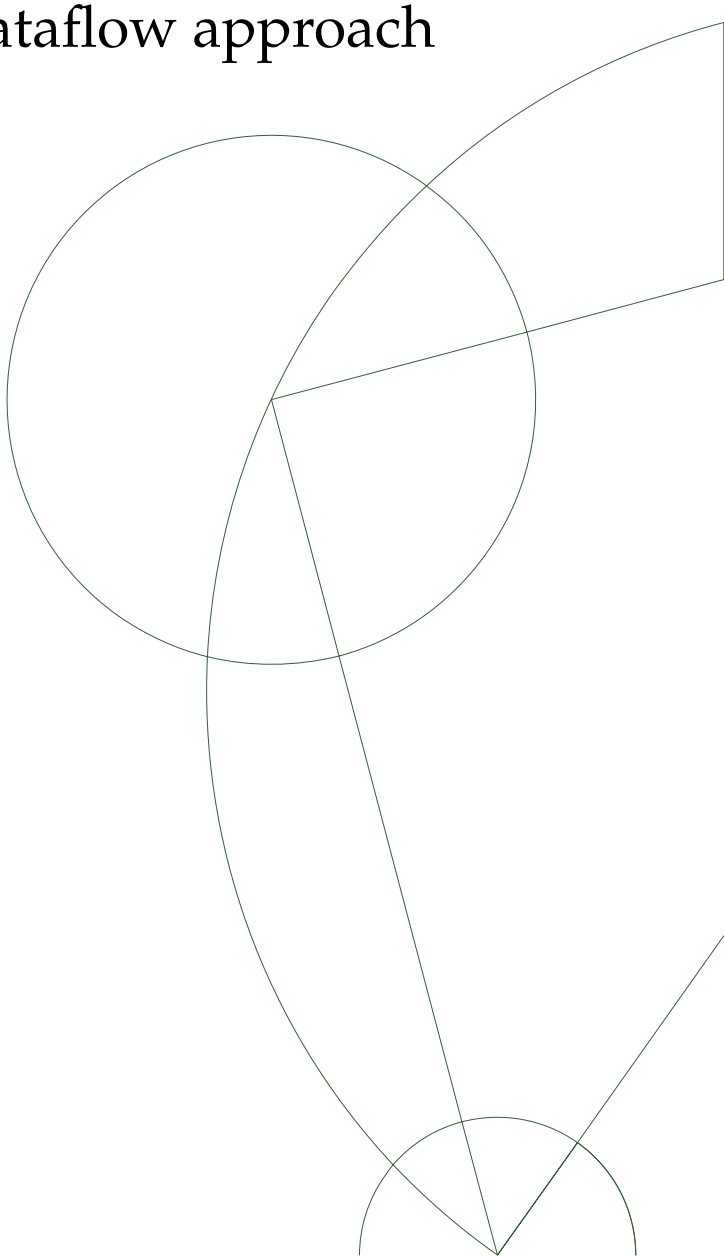
FACULTY OF SCIENCE
UNIVERSITY OF COPENHAGEN

Master's Thesis

Troels Henriksen – athas@sigkill.dk

Exploiting functional invariants to optimise parallelism: a dataflow approach



Supervisors: Cosmin Eugen Oancea and Fritz Henglein

February 2014

Abstract

We present $\mathcal{L}_0$, a purely functional programing language supporting nested regular data parallelism and targeting massively parallel SIMD hardware such as modern graphics processing units (GPUs).

$\mathcal{L}_0$ incorporates the following novel features:

- A type system for in-place modification and aliasing of arrays and array slices that ensures referential transparency, which in turn supports equational reasoning.
- An assertion language for expressing bounds checks on dynamically allocated arrays, which can often be checked statically to eliminate dynamic bounds checks.
- Compiler optimisations for hoisting bounds checks out of inner loops and performing loop fusion based on structural transformations.

We show that:

- The type system is simpler than existing linear and unique typing systems such Clean [4], and more expressive than libraries such as DPH, Repa and Accelerate [12, 24, 13], for efficient array processing.
- Our fusion transformation is capable of fusing loops whose output is used in multiple places, when possible without duplicating computation, a feature not found in other implementations of fusion [23].
- The effectiveness of our optimisations is demonstrated on three real-world benchmark problems from quantitative finance, based on empirical run-time measurements and manual inspection of the optimised programs. In particular, hoisting and fusion yield a sequential speed-up of up to 71% compared to the unoptimised source code, even without any parallel execution.

The results suggest that the language design, expressiveness and optimisation techniques of $\mathcal{L}_0$ can be realized across a range of SIMD-architectures, notably existing GPUs and manycore-chips, to simultaneously achieve portability and eventually performance competitive with hand-coding.

The results reported are based on joint work with Cosmin Oancea, DIKU.

Contents

Preface

This dissertation is submitted in fulfillment of the graduate education in computer science (*Datalogi*) at the University of Copenhagen, for Troels Henriksen.

Chapter 1

Introduction

For practically the entire lifetime of the electronic computer, programmers have been used to an exponential growth in commonly available computing power. Until around 2006, this directly manifested itself as improvements to sequential performance, although physical limits made it uneconomical (or impossible) for this trend to continue. These days, hardware designers are making their machines increasingly *parallel*: rather than speeding up the individual processors, as happened previously, *more* processors, or more *specialised* processors, are added. Thus, while computing power is still growing, it has become increasingly necessary to write programs that are parallel in order to take full advantage of modern advancements in hardware.

One interesting development is the commoditisation of massively parallel vector processors in the form of graphics cards. While hardware acceleration of graphics became commonplace in the 90s, it was not until the rise of CUDA and OpenCL in 2006 that *General-Purpose computing on Graphics Processing Units* (GPGPU) began to move into the mainstream. The kind of parallelism supported by GPGPU is *data parallelism*, wherein each processor performs the same task on different pieces of the data. This is also called Single Instruction Multiple Data (SIMD). Today, there are three main ways to take advantage of this data parallel processing power:

Low-level interfaces: CUDA and OpenCL implementations are supplied by the GPU vendors.[1] These are very low-level, and provide a C-like programming interface. Furthermore, GPU hardware has very complicated performance characteristics, and it can be hard to achieve optimal, or even good performance. Nevertheless, the full supported power of the devices is available at this level, and optimal performance is theoretically achievable, although in practice, specialist knowledge is required to achieve good results at this level.

At the most basic level, all other approaches eventually boil down to talking to a low-level interface.

[1] Strictly. OpenCL has a broader focus, and seeks to provide an interface to heterogenous computation in general, but for the purpose of this thesis, we will consider OpenCL and CUDA to be GPU-oriented.

Libraries: Some programming libraries aimed at high-peformance computing have been rewritten to take advantage of GPU acceleration. For example, Nvidia provides CUBLAS [30], an implementation of the well-known BLAS array operations API. These libraries are typically written by experts, and come close to peak potential performance on the target hardware. It is generally easy to use these libraries from any language offering a good foreign function interface, and it is thus an efficient way to reach a large number of potential users. Usage of these libraries requires little in the way of GPU knowledge, or indeed knowledge about parallel programming at all.

On the downside, although each discrete function may be well-optimised in isolation, the library approach does not permit optimisation of composed operations. For example, if a library exports a function `mult` to multiply two matrices, and we use it in two invocations to multiply three matrices, as in `mult(x,mult(y,z))`, the library will likely not be as fast as if we used a specialised ternary multiplication function. Although particularly clever libraries may use a variant of lazy evaluation to delay computation and optimise some composed operations [26], the optimisation potential is still limited as long as the program cannot be inspected directly.

The library approach is very popular in practice, with many high-performance computing libraries now possessing GPGPU backends.

Data-parallel programming languages: The final way to perform GPGPU is to integrate GPU support directly into a programming language, with full compiler support. This permits code generation based on a global view of the entire computation, at least in theory, and optimise with full knowledge of the program. There appears to be two main paths within the programming language approach:

Embedded languages: Somewhat similar to the library approach, this integrates GPGPU support in an existing language as an *Embedded Domain Specific Language* (EDSL) [16]. The distinction between an EDSL and a library is often fuzzy, with the distinction typically being about the level of composability offered, and whether the EDSL follows the same evaluation rules as the host language. Further blurring the issue, some EDSLs use syntactical extensions – for example through macros [25] or quasiquotation [27] – while others take advantage of the host languages existing syntactical facilities.

The limitations for EDSLs are similar to the ones for libraries. For example, the embedded language must be expressible in the type system of the host language. It also varies how much support the host language provides for hooking into the compiler, in order to perform optimisation. On the other hand, much of the infrastructure of the host language will be inherited by the EDSL, leading to a much simpler implementation, compared to writing a full compiler. Furthermore, due to the integration with the host language, EDSL usage can be very seamless. On the other hand, it is extremely hard to access an EDSL from outside of the host language.

DPH, Repa and Accelerate [12, 24, 13] are examples of data-parallel EDSLs for Haskell, with Accelerate suppors OpenCL and CUDA backends.

Independent languages: The final approach is to write an entire compiler targeting GPGPU execution. This provides total control, at the cost of greatly increased implementation complexity. Furthermore, it can be difficult to integrate components written in a these new languages into existing code-bases written in mainstream languages. Nevertheless, the language can be designed from the bottom up for efficient parallel execution, without compromises due to host language integration. The NESL [8] language is an early ('96) example of a programming language designed entirely for data-parallel execution. Although designed before the proliferation of GPUs, a GPU backend has recently been developed [5]. Another example of such a language is Single-Assignment C (SAC) [18].

In a way, we could also consider the OpenCL and CUDA kernel languages themselves to be in this category, but we only consider high-level languages to be proper members of this group.

The library approach is effective if a library exists for the specific problem the programmer is attempting to solve, but will often be neither sufficiently fast nor expressive for new domains. EDSLs suffer from a similar problem – in particular, nested parallelism and similar complex control flow is generally poorly supported. The NESL and SAC languages are more expressive, but their implementation does not perform many advanced optimisations. Clearly, there is still great uncertainty about the best way to program these massively parallel machines.

To investigate possible solutions, we have examined several real-world financial kernels originally implemented in languages such as OCaml, C++, and C, and measuring in the range of hundreds of lines of compact code, with two main objectives in mind:

1. What is the simplest language that permits a relatively straightforward translation of the financial programs, while still expressing algorithm invariants that enable the generation of efficient parallel code?

2. What compiler optimizations would result in efficiency comparable to code hand-tuned for the specific hardware?

To answer the first question, we will present $\mathcal{L}_0$, an independent language designed for parallel execution. We have chosen to implement $\mathcal{L}_0$ as a non-embedded language in order to have more design freedom, as we do not need to, e.g., fit $\mathcal{L}_0$ into the type system of an existing language. Our language supports *nested* parallelism on *regular* arrays, i.e., arrays where all rows of the array have the same size. This restriction is due to regular arrays being more amenable to compiler optimizations, in particular they allow transposition and simplified size analysis.

Our language supports nested parallelism because many programs exhibits several layers of parallelism that cannot be exploited by flat parallelism in the style of REPA [24]. For example, the examined financial kernels contain several innermost `scan` or `reduce` operations, and at least one semantically sequential loop per benchmark.

Our language is also *functional*, because we would rather invest compiler effort in exploiting high-level program invariants rather than in proving them. The common example here is parallelism: `map-reduce` constructs are inherently parallel, while Fortran-style `do` loops require sophisticated analysis to decide parallelism. Furthermore, such analyses [11, 19, 32, 31] have not yet been integrated in the repertoire of commercial compilers, likely due to "heroic effort" concerns, albeit their effectiveness was demonstrated on comprehensive suites, and some of them were developed more than a decade ago.

The answer to the second question seems to be that a common ground needs to be found between functional and imperative optimizations and, to a lesser extent, between functional and imperative language constructs. Much in the same way in which data parallelism seems to be generated by a combination of `map`, `reduce`, and `scan` operations, the optimization opportunities seem solvable via a combination of *transposition*, loop *fusion*, loop *interchange* and loop *distribution* [2].

It follows that classic index-based loops are necessary in the intermediate representation, regardless of whether they are provided as a language construct or are derived from tail-recursive functions via a code transformation.

Loop fusion is one of the most important code transformations, as it has the potential for substantially optimising both memory hierarchy overhead and, sometimes asymptotically, space requirements. In imperative languages, fusing producer-consumer loops requires dependency analysis on arrays applied at loop-nest level. Such analysis, however, has often been labeled as "heroic effort" and, if at all, is supported only in its simplest and most conservative form in industrial compilers. In functional languages however, fusion is naturally and relatively easily derived from the producer-consumer relation between program constructs that expose a rich, higher-order algebra of program invariants, such as the `map-reduce` list homomorphisms.

Finally, an indirect consequence of having to deal with sequential dependent loops is that $\mathcal{L}_0$ provides support for *in-place updates* of array elements. The observable semantics still respect referential transparency, i.e., a deep copy of the original array but with the corresponding element replaced, *intersected* with the imperative one, i.e., referential transparency cannot be guaranteed, a compile-time error is signaled. This approach enables the intuitive cost model that the user likely assumes, while preserving the functional semantics.

Throughout this thesis, we will often refer to a vaguely defined "programmer", as well as ascribe various motives and expectations to this nebulous being. While $\mathcal{L}_0$ is intended as an intermediate language, and in the end is intended as a target language by compilers for higher-level languages, it has a well-defined human-readable (and writable) syntax, and can be programmed directly. Indeed, all extant $\mathcal{L}_0$ programs have been written by hand. Thus, when "the programmer" is referenced, we can refer to either an actual human, or a compiler generating

$\mathcal{L}_0$ code. For our purposes, these will have identical motives, although a human programmer may complain somewhat more vocally about the lack of syntactical niceties in the language.

1.1 Contributions

We present a purely functional data-parallel programming language, $\mathcal{L}_0$, with support for nested parallelism. The language supports a method for safely performing in-place updates of array data through a type system concept called *uniqueness types*. Through the translation of real-world financial programs to $\mathcal{L}_0$, we demonstrate the practical usefulness of this language feature.

We describe the design and implementation of several optimisations, notably hoisting bounds checks out of inner loops, and loop fusion based on a structural transformation. The fusion transformation is capable of fusing loops whose output is used in multiple places, when possible without duplicating computation. Optimising bounds checks is an example of a general principle of removing checks statically when possible, and dynamically when necessary.

The benefits of our optimisations are demonstrated on three real-world financial benchmarks. It is shown that the compiler is able to hoist bounds checks and other assertions outside of loops.

The effectiveness of fusion is demonstrated via compiler instrumentation and quantitative and qualitative measurements on the three benchmarks, in the form of inspecting the changes in program dataflow. This shows that always refusing to duplicate computation is too conservative on parallel hardware, and discuss potential directions for further improvement.

The implementation of the $\mathcal{L}_0$ compiler consists of roughly ten thousand lines of Haskell (ignoring comments and blank lines), and it is hosted and publicly browsable at `https://github.com/HIPERFIT/L0Language`.

Parts of this thesis, in particular the core of the fusion algorithm in Chapter 7, has been previously published as

> Henriksen, Troels and Cosmin Eugen Oancea. "A T2 Graph-reduction Approach to Fusion". In: *Proceedings of the 2Nd ACM SIGPLAN Workshop on Functional High-performance Computing*. FHPC '13. Boston, Massachusetts, USA: ACM, 2013, pp. 47–58. ISBN: 978-1-4503-2381-9. DOI: `http://dx.doi.org/10.1145/2502323.2502328`. URL: `http://doi.acm.org/10.1145/2502323.2502328`

1.2 Report outline

The remainder of the report is structured as follows. Chapters 2 and 3 will introduce the programmer-visible part of $\mathcal{L}_0$ and serves as a language reference. Chapter 4 presents a slight modification of the external language, that makes it more amenable to transformation and optimisation. Chapter 5 discusses simple classical optimisations in the context of $\mathcal{L}_0$, while Chapter 6 discusses slightly more advanced classical optimisations. Chapter 7 covers *loop fusion*, an

important structural optimisation, while Chapters 8 and 9 cover transformations that enable other optimisations (although particularly fusion).

1.3 Notation

In various places, I will use an $\overline{\text{overline}}$ to indicate a comma-separated sequence of terms. For example, when describing a function call, rather than writing:

$$f(e_1, \ldots, e_n)$$

I may instead write:

$$f(\overline{es})$$

I may also use this in conjunction with expliclt arguments, as in:

$$f(e_{start}, \overline{es}, e_{end})$$

Which is a shortcut for

$$f(e_{start}, e_1, \ldots, e_n, e_{end})$$

Part I

Language Design

Chapter 2

The $\mathcal{L}_0$ language

The $\mathcal{L}_0$ programming language is a purely functional, call-by-value, mostly first-order language that permits bulk operations on arrays using *second-order array combinators* (SOACs).

The primary idea behind $\mathcal{L}_0$ is to design a language that has enough expressive power to conveniently express complex programs, yet is also amenable to aggressive optimisation and parallelisation. Unfortunately, as the expressive power of a language grows, the difficulty of optimisation often rises likewise. For example, we support nested parallelism, despite the complexities of efficiently mapping to the flat parallelism supported by hardware, as a great many programs depend on this feature. On the other hand, we do not support non-regular arrays, as they complicate size analysis a great deal. The fact that $\mathcal{L}_0$ is purely functional is intended to give an optimising compiler more leeway in rearranging the code and performing high-level optimisations. It is also the plan to eventually design a rigorous cost model for $\mathcal{L}_0$, although this work has not yet been completed.

This chapter serves as a reference and basic introduction to the $\mathcal{L}_0$ language, while Chapters 3 and 4 describes more subtle design issues. Sections 2.1 and 2.2 will present $\mathcal{L}_0$ through informal walkthrough of the major language concepts, whilst a complete reference of language constructs is given in Section 2.3.

2.1 First-order $\mathcal{L}_0$

The syntax of $\mathcal{L}_0$, as seen on Figure 1 and Figure 2, is heavily inspired by Haskell and Standard ML. An identifier starts with a letter, followed by any number of letters, digits and underscores. Numeric, string and character literals use the same notation as Haskell (which is very similar to C), including all escape characters. Comments are indicated with `//` and span to end of line.

An $\mathcal{L}_0$ program consists of a sequence of *function definitions*, of the following form.

```
fun return-type name(params...) = body
```

A function must declare both its return type and the types of all its parameters. All functions (except for inline anonymous functions; see below) are

t	::=	`int`	(Integers)
	|	`real`	(Floats)
	|	`bool`	(Booleans)
	|	`char`	(Characters)
	|	`{`t_1`, ..., `t_n`}`	(Tuples)
	|	`[`t`]`	(Arrays)
	|	`*[`t`]`	(Unique arrays)
k	::=	n	(Integer)
	|	x	(Decimal number)
	|	b	(Boolean)
	|	c	(Character)
	|	`{`$v_1, \ldots, v_n$`}`	(Tuple)
	|	`[`$v_1, \ldots, v_n$`]`	(Array)
p	::=	`id`	(Name pattern)
	|	`{`p_1`, ..., `p_n`}`	(Tuple pattern)

Figure 1: $\mathcal{L}_0$ syntax

defined globally. $\mathcal{L}_0$ does not use type inference. Symbolic constants are not supported, although 0-ary functions can be defined. As a concrete example, here is the recursive definition of the factorial function in $\mathcal{L}_0$.

```
fun int fact(int n) =
  if n = 0 then 1
           else n * fact(n-1)
```

Indentation has no syntactical significance in $\mathcal{L}_0$, but recommended for readability.

The syntax for tuple types is a comma-separated list of types or values enclosed in braces, so `{int, real}` is a pair of an integer and a floating-point number. Both single-element and empty tuples are permitted. Array types are written as the element type surrounded by brackets, meaning that `[int]` is a one-dimensional array of integers, and `[[[{int, real}]]]` is a three-dimensional array of tuples of integers and floats. An immediate array is written as a sequence of elements enclosed by brackets.

```
[1, 2, 3]       // Array of type [int].
[[1], [2], [3]] // Array of type [[int]].
```

All arrays must be *regular* (often termed *full*) - for example, all rows of a two-dimensional array must have the same number of elements.

```
[[1, 2], [3]]      // Compile-time error.
[iota(1), iota(2)] // A run-time error if reached.
```

The restriction to regular arrays simplifies size analysis and optimisation.

Arrays are indexed using the common row-major notation, e.g., `a[i1, i2, i3...]`. An indexing is said to be *full* if the number of given indexes is equal to the dimensionality of the array.

e	::=	k	(Constant)
	|	v	(Variable)
	|	`{`e_1`,...,`e_n`}`	(Tuple expression)
	|	`[`e_1`,...,`e_n`]`	(Array expression)
	|	$e_1 \odot e_2$	(Binary operator)
	|	`-`e	(Prefix minus)
	|	`not` e	(Logical negation)
	|	`if` e_1 `then` e_2 `else` e_3	(Branching)
	|	v`[`e_1`, ..., `e_n`]`	(Indexing)
	|	v`(`e_1`, ..., `e_n`)`	(Function call)
	|	`let` p `=` e_1 `in` e_2	(Pattern binding)
	|	`zip(`e_1`, ..., `e_n`)`	(Zipping)
	|	`unzip(`e`)`	(Unzipping)
	|	`iota(`e`)`	(Range)
	|	`replicate(`e_n`, `e_v`)`	(Replication)
	|	`size(`e`)`	(Array length)
	|	`reshape((`e_1`,...,`e_n`), `e`)`	(Array reshape)
	|	`transpose(`e`)`	(Transposition)
	|	`split(`e_1`, `e_2`)`	(Split e_2 at index e_1)
	|	`concat(`e_1`, `e_2`)`	(Concatenation)
	|	`let` v_1 `=` v_2 `with` `[`e_1`,...,`e_n`]` `<-` e_v `in` e_b	(In-place update)
	|	`loop (`p `=` e_1`) =` `for` v `<` e_2 `do` e_3 `in` e_4	(Loop)
fun	::=	`fun` t v`(`t_1 v_1`,...`t_n v_n`)` `=` e	
$prog$	::=	ϵ	
	|	fun $prog$	

Figure 2: $\mathcal{L}_0$ syntax, continued

A `let`-expression can be used to refer to the result of a subexpression:

```
let z = x + y in ...
```

Recall that $\mathcal{L}_0$ is eagerly evaluated, so the right-hand side of the `let` is evaluated exactly once, at the time it is first encountered.

Two-way `if-then-else` is the only branching construct in $\mathcal{L}_0$. Pattern matching is supported in a limited way for taking apart tuples, but this can only be done in `let`-bindings, and not directly in a function argument list. Specifically, the following function definition is not valid.

```
fun int sumpair({int, int} {x, y}) = x + y // WRONG!
```

Instead, we must use a let-binding explicitly, as follows.

```
fun int sumpair({int, int} t) =
  let {x,y} = t in x + y
```

Pattern-matching in a binding is the only way to access the components of a tuple.

Function calls are written as the function name followed by the arguments enclosed in parentheses. All function calls must be fully saturated - currying is only permitted in SOACs (see Section 2.2).

2.1.1 Sequential loops

$\mathcal{L}_0$ has a built-in syntax for expressing certain tail-recursive functions. Consider the following tail-recursive formulation of a function for computing the Fibonacci numbers.

```
fun int fib(int n) = fibhelper(1,1,n)

fun int fibhelper(int x, int y, int n) =
  if n = 1 then x else fibhelper(y, x+y, n-1)
```

We can rewrite this using the `loop` construct.

```
fun int fib(int n) =
  loop ({x, y} = {1,1}) = for i < n do
                            {y, x+y}
  in x
```

The semantics of this is precisely as in the tail-recursive function formulation. In general, a loop

```
loop (pat = initial) = for i < bound do loopbody
in body
```

has the following semantics:

1. Bind *pat* to the initial values given in *initial*.
2. While $i <$ *bound*, evaluate *loopbody*, rebinding *pat* to be the value returned by the body. At the end of each iteration, increment i by one.
3. Evaluate *body* with *pat* bound to its final value.

Semantically, a `loop` expression is completely equivalent to a call to its corresponding tail-recursive function. For example, denoting by `t` the type of `x`, the loop in Figure 3 has the semantics of a call to the tail-recursive function on the right-hand side.

The purpose of `loop` is partly to render some sequential computations slightly more convenient, but primarily to express certain very specific forms of recursive functions, specifically those with a fixed iteration count. This property can eventually be used for analysis and optimisation, although the current $\mathcal{L}_0$ compiler does not yet exploit this.

```
loop (x = a) =                    fun t f(int i, int n, t x) =
  for i < n do                      if i >= n then x
    g(x)                       ⇒        else f(i+1, n, g(x))
in body                           let x = f(i, n, a)
                                  in body
```

Figure 3: Equivalence between loops and recursive functions

2.1.2 In-place updates

In an array-oriented programming language, a common task is to modify some elements of an array. In a pure language, we cannot permit free mutation, but we can permit the creation of a duplicate array, where some elements have been changed. General modification of array elements is done using the `let-with` construct. In its most general form, it looks as follows.

```
let dest = src with [indexes] <- value
in body
```

This evaluates *body* with *dest* bound to the value of *src*, except that the element(s) at the position given by *indexes* take on the new value *value*.[1] The given indexes need not be complete, but in that case, *value* must be an array of the proper size. As an example, here's how we could replace the third row of an $n \times 3$ array.

```
let b = a with [2] <- [1,2,3] in b
```

Whenever $dest = src$, we can write

```
let dest[indexes] = value in body
```

as a shortcut. Note that this has no special semantic meaning, but is simply a case of normal name shadowing.

For example, the loop given below implements the "imperative" version of matrix multiplication of two $N \times N$ matrices.

```
fun *[[int]] matmultImp(int N, [[int]] a, [[int]] b) =
    let res = replicate(N, iota(N)) in
    loop (res) = for i < N do
        loop (res) = for j < N do
            let partsum =
                let res = 0 in
                loop (res) = for k < N do
                    let res = res + a[i,k] * b[k,j]
                    in  res
                in res
            in let res[i,j] = partsum in res
        in res
    in res
```

[1] Yes, this is the *third* binding construct in the language, ignoring function abstraction!

$$
\begin{array}{rcll}
l & ::= & \texttt{fn}\ t\ (t_1\ v_1, \ldots, t_n\ v_n)\ \texttt{=>}\ e & \text{(Anonymous function)} \\
 & | & \texttt{id}\ (e_1, \ldots, e_n) & \text{(Curried function)} \\
 & | & \texttt{op}\ \odot\ (e_1, \ldots, e_n) & \text{(Curried operator)} \\
 & & & \\
e & ::= & \texttt{map}(l, e) & \\
 & | & \texttt{filter}(l, e) & \\
 & | & \texttt{reduce}(l, x, e) & \\
 & | & \texttt{scan}(l, x, e) & \\
 & | & \texttt{redomap}(l_r, l_m, x, e) &
\end{array}
$$

Figure 4: Second-order array combinators

With the naive implementation based on copying the source array, executing the `let-with` expression would require memory proportional to the entire source array, rather than proportional to the slice we are changing. This is not ideal. Therefore, the `let-with` construct has some unusual restrictions to permit in-place modification of the *src* array, as described in Chapter 3. Simply put, we track that *src* is never used again. The consequence is that we can guarantee that the execution of a `let-with` expression does not involve any copying of the source array in order to create the newly bound array, and therefore the time required for the update is proportional to the section of the array we are updating, not the entire array. We can think of this as similar to array modification in an imperative language.

2.2 SOACs

The language presented in the previous section is in some sense "sufficient", in that it is Turing-complete, and can express imperative-style loops in a natural way with `do`-loops. However, $\mathcal{L}_0$ is not intended to be used in such a way - bulk operations on arrays should be expressed via the four *second-order array combinators* (SOACs) shown in Figure 4, as the optimisations covered in later chapters are expressed as transformations on these.

The semantics of the SOACs is identical to the similarly-named higher-order functions found in many functional languages, but we reproduce it here for completeness. The types given are not $\mathcal{L}_0$ types, but a Haskell-inspired notation, since the SOACs cannot be typed in $\mathcal{L}_0$ itself.

$$
\begin{aligned}
\texttt{map}(f,a) &:: (\alpha \to \beta) \to [\alpha] \to [\beta] \\
&\equiv \{f(a[0]), \ldots, f(a[n])\}
\end{aligned}
$$

$$
\begin{aligned}
\texttt{filter} &:: (\alpha \to \texttt{bool}) \to [\alpha] \to [\alpha] \\
\texttt{filter}(f,a) &\equiv \{a[i] \mid f(a[i]) = \texttt{True}\}
\end{aligned}
$$

$$
\begin{aligned}
\texttt{reduce} &:: (\alpha \to \alpha \to \alpha) \to \alpha \to [\alpha] \to \alpha \\
\texttt{reduce}(f,x,a) &\equiv f(\ldots(f(f(x,a[0]), a[1])\ldots), a[n])
\end{aligned}
$$

$$\texttt{scan} :: (\alpha \to \alpha \to \alpha) \to \alpha \to [\alpha] \to [\alpha]$$
$$\texttt{scan}(f,x,a) \equiv \{f(x,a[0]),\ f(f(x,a[0]),a[1]),\ldots\}$$

`scan` is an inclusive prefix scan, and returns an array of the same outer size as the original array. The functions given to `reduce` and `scan` must be binary associative operators, and the value given as the initial value of the accumulator must be the neutral element for the function. These properties are not checked by the $\mathcal{L}_0$ compiler, and are the responsibility of the programmer.

`redomap` is a special case – it is not intended for use by the programmer, but used internally for fusing `reduce` and `map`. Its semantics is as follows.

$$\texttt{redomap} :: (\alpha \to \alpha \to \alpha) \to (\alpha \to \beta \to \alpha) \to \alpha \to [\beta] \to \alpha$$
$$\texttt{redomap}(\odot,g,x,v) \equiv \texttt{foldl}(g,\ x,\ v)$$

Note that the runtime semantics is a left-fold, not a normal $\mathcal{L}_0$ `reduce`. In particular, g need not be associative. We use a Haskell-like syntax to explain the rationale behind `redomap`:
(`reduce` $\odot$ e) $\circ$ (`map` f) can be formally transformed, via the list homomorphism (LH) promotion lemma [6], to an equivalent form:
(`reduce` $\odot$ e) $\circ$ (`map` f) $\equiv$ `reduce` $\odot$ e $\circ$ `map` (`reduce` $\odot$ e $\circ$ `map` f) $\circ$ `split`$_p$
where the original list is distributed to p parallel processors, each of which execute the original map-reduce computation sequentially and, at the end, reduce in parallel the per-processor result using the operator $\odot$. Hence, the *inner* map-reduce can be rewritten as a left-fold:
(`reduce` $\odot$ e) $\circ$ (`map` f) $\equiv$ `reduce` $\odot$ e $\circ$`map` (`foldl` g e) $\circ$ `split`$_p$
Where g is a function generated from the composition of f and $\odot$. It follows that in order to generate parallel code for
`(reduce` $\odot$ e`)` $\circ$ `(map` f`)` it is sufficient to record either $\odot$ and f, or $\odot$ and g. We choose the latter, i.e., `redomap(`$\odot$`,` g`,` e`)`, because it allows a richer compositional algebra for fusion. In particular, it allows us to fuse `reduce` $\circ$ `map` $\circ$ `filter` into a `redomap` without duplicating computation, as described in Chapter 7.

2.3 Language reference

The builtin types in $\mathcal{L}_0$ are `int`, `real`, `bool` and `char`, as well as their combination in tuples and arrays.

The following list describes every syntactical language construct in the language.

constant
: Evaluates to itself.

var
: Evaluates to its value in the environment.

`x arithop y`
: Evaluate the binary operator on its operands, which must both be of

either type `int` or `real`. The following operators are supported: `+`, `*`, `-`, `/`, `%`, `=`, `<`, `<=`, `pow`.

`x bitop y`
: Evaluate the binary operator on its operands, which must both be of type `int`. The following operators are supported: `^`, `&`, `|`, `>>`, `<<`, i.e., bitwise xor, and, or, and arithmetic shift right and left.

`x && y`
: Logical conjunction; both operands must be of type `bool`. Not short-circuiting, as this complicates program transformation. If short-circuiting behaviour is desired, the programmer can use `if` explicitly.

`x || y`
: Logical disjunction; both operands must be of type `bool`. As with `&&`, not short-circuiting.

`not x`
: Logical negation of x, which must be of type `bool`.

`-x`
: Numerical negation of x, which must be of type `real` or `int`.

`a[i]`
: Return the element at the given position in the array. The index may be a comma-separated list of indexes.

`zip(`$a_1, \ldots, a_n$`)`
: Zips together the elements of the outer dimensions of arrays $a_1, \ldots, a_n$. Static or runtime check is required to check that the sizes of the outermost dimension of arrays $a_1, \ldots, a_n$ are the same. If this invariant does not hold, program execution stops with an error.

`unzip(`a`)`
: If the type of a is $[\{t_1, \ldots, t_n\}]$, the result is a tuple of n arrays, i.e., $\{[t_1], \ldots, [t_n]\}$, otherwise it is a type error.

`iota(n)`
: An array of the integers from 0 to n.

`replicate(n, a)`
: An array consisting of n copies of a.

`size(`k`, a)`
: The size of dimension k of array a. k must be a static integral constant.

`split(n, a)`
: Partitions the given array into two disjoint arrays $a[0 \ldots n]$, $a[n+1 \ldots]$, returned as a tuple.

`concat(a, b)`
: Concatenate the rows/elements of one array with another. The shape of the two arrays must be identical in all but the first dimension.

`copy(x)`
: Return a deep copy of the argument. Semantically, this is just the identity function, but it has special semantics related to uniqueness types as described in Chapter 3.

`reshape((`$\texttt{dim}_1$`, ..., `$\texttt{dim}_n$`), a)`
: Reshape the elements of the given array into the specified shape. The number of elements in a must be equal to $\texttt{dim}_1 \times \ldots \times \texttt{dim}_n$.

`transpose(a)`
: Return the transpose of a.

`transpose(k,n,a)`
: Return the generalised transpose of a. If b=`transpose(`k`,`n`,`a`)`, then

$$\texttt{a}[i_1, \ldots, i_k, i_{k+1}, \ldots, i_{k+n}, \ldots, i_q] = \texttt{b}[i_1, \ldots, i_{k+1}, \ldots, i_{k+n}, i_k, \ldots, i_q].$$

  We will call this an operation an (k, n)-*transposition*. Note that `transpose(0,1,a)` is the common two-dimensional transpose.

  Be aware that k and n must be static integer literals, and $k + n$ must be non-negative and smaller than the rank of `a`, or it is considered a type error.

`let pat = e in body`
: While evaluating *body*, bind the names mentioned in *pat* to the components in the corresponding positions of the value of e. We will refer to the expression e as the "right-hand side" (or RHS).

`let dest = src with [index] <- v in body`
: Evaluate *body* with *dest* bound to the value of *src*, except that the element(s) at the position given by the index take on the value of v. The given index need not be complete, but in that case, the value of v must be an array of the proper size.

`if c then a else b`
: If c evaluates to *True*, evaluate a, else evaluate b.

`loop (pat = initial) = for i < bound do loopbody in body`

1. Bind *pat* to the initial values given in *initial*.
2. While $i <$ *bound*, evaluate *loopbody*, rebinding *pat* to be the value returned by the body.
3. Evaluate *body* with *pat* bound to its final value.

`map(f, a)`
: Apply f to every element of a and return the resulting array.

`reduce(f, x, a)`
: Left-reduction with f across the elements of a, with x as the neutral element for f. f must be associative, as the evaluation order is not otherwise specified.

`scan(f, x, a)`
: Inclusive prefix-scan.

`filter(f, a)`
: Remove all those elements of a that do not satisfy the predicate f.

2.3.1 Tuple shimming

In a SOAC, if the given function expects n arguments of types $t_1, \ldots, t_n$, but the SOAC will call the function with a single argument of type `{`$t_1, \ldots, t_n$`}` (that is, a tuple), the $\mathcal{L}_0$ compiler will automatically generate an anonymous unwrapping function. This allows the following expression to type-check (and run):

```
map(op +, zip(as, bs))
```

Without the tuple shimming, the above would cause an error, as `op +` is a function that takes two arguments, but is passed a two-element tuple by `map`.

2.3.2 Arrays of tuples

For reasons that will be explained in Chapter 7, arrays of tuples are in a sense merely syntactic sugar for tuples of arrays. The type `[{int, real}]` is transformed to `{[int], [real]}` during the compilation process, and all code interacting with arrays of tuples is likewise transformed. In most cases, this is fully transparent to the programmer, but there are edge cases where the transformation is not trivially an isomorphism.

Consider the type `[{[int], [real]}]`, which is transformed into `{[[int]], [[real]]}`. These two types are not isomorphic, as the latter has more stringent demands as to the fullness of arrays. For example,

```
[
 {[1],   [1.0]},
 {[2,3], [2.0]}
]
```

is a value of the former, but the first element of the corresponding transformed tuple

```
{
 [[1],   [2, 3]],
 [[1.0], [2.0]]
}
```

is not a full array. Hence, when determining whether a program generates full arrays, we must hence look at the *transformed* values - in a sense, the fullness requirement "transcends" the tuples.

Section 4.1 contains more information on the transformation of arrays of tuples.

Chapter 3

Uniqueness Types

While $\mathcal{L}_0$ is through and through a pure functional language, it may occasionally prove useful to express certain algorithms in an imperative style. Consider a function for computing the n first Fibonacci numbers:

```
fun [int] fib(int n) =
  // Create "empty" array.
  let arr = iota(n) in
  // Fill array with Fibonacci numbers.
  loop (arr) = for i < n-2 do
                 let arr[i+2] = arr[i] + arr[i+1]
                 in arr
  in arr
```

If the array `arr` is copied for each iteration of the loop, we are going to put enormous pressure on memory, and spend a lot of time moving around data, even though it is clear in this case that the "old" value of `arr` will never be used again. Precisely, what should be an algorithm with complexity $O(n)$ becomes $O(n^2)$, due to copying the size n array (an $O(n)$ operation) for each of the n iterations of the loop.

To prevent this, we will want to update the array *in-place*, that is, with a static guarantee that the operation will not require any additional memory allocation, such as copying the array. With an in-place modification, a `let-with` can modify the array in time proportional to the slice being updated ($O(1)$ in the case of the Fibonacci function), rather than time proportional to the size of the final array, as would the case if we perform a copy. In order to perform the update without violating referential transparency, we need to know that no other references to the array exists, or at least that such references will not be used on any execution path following the in-place update.

In $\mathcal{L}_0$, this is done through a type system feature called *uniqueness types*, similar to, although simpler, than the uniqueness types of Clean [3, 4]. Alongside a (relatively) simple aliasing analysis in the type checker, this is sufficient to determine at compile time whether an in-place modification is safe, and signal a compile time error if `let-with` is used in way where safety cannot be guaranteed. This means that `let-with` must *always* be efficient, and its use is not permitted otherwise.

The simplest way to introduce uniqueness types is through examples. To that end, let us consider the following function definition.

```
fun *[int] modify(*[int] a, int i, int x) =
  let b = a with [i] <- a[i] + x in
  b
```

The function call `modify(`a`,`i`,`x`)` returns a, but where the element at index `i` has been increased by x. Note the asterisks: in the parameter declaration `*[int] a`, this means that the function `modify` has been given "ownership" of the array a, meaning that any caller of `modify` will never reference array a after the call again. In particular, `modify` can change the element at index `i` without first copying the array, i.e. `modify` is free to do an in-place modification. Furthermore, the return value of `modify` is also unique - this means that the result of the call to `modify` does not share elements with any other visible variables.

Let us consider a call to `modify`, which might look as follows.

```
let b = modify(a, i, x) in
...
```

Under which circumstances is this call valid? Two things must hold:

1. The type of a must be `*[int]`, of course.
2. Neither a or any variable that *aliases* a may be used on any execution path following the call to `modify`.

In general, when a value is passed as a unique-typed argument in a function call, we consider that value to be *consumed*, and neither it nor any of its aliases can be used again. Otherwise, we would break the contract that gives the function liberty to manipulate the argument however it wants. Note that it is the type in the argument declaration that must be unique - it is permissible to pass a unique-typed variable as a non-unique argument (that is, a unique type is a subtype of the corresponding nonunique type).

A variable v aliases a if they may share some elements, i.e. overlap in memory. As the most trivial case, after evaluating the binding `let b = a`, the variable `b` will alias a. As another example, if we extract a row from a two-dimensional array, the row will alias its source:

```
let b = a[0] in
... // b is aliased to a (assuming a is not one-dimensional)
```

Section 3.1 will cover sharing and sharing analysis in greater detail.

Let us consider the definition of a function returning a unique array:

```
fun *[int] f([int] a) = e
```

Note that the argument, a, is non-unique, and hence we cannot modify it. There is another restriction as well: a must not be aliased to our return value, as the uniqueness contract requires us to ensure that there are no other

```
let b = a with [i] <- 2 in
f(b,a) // Error: a used after being source in a let-with
```

Figure 5: Violation of Uniqueness Rule 1

```
fun *[int] broken([[int]] a, int i) =
  a[i] // Return value aliased with 'a'.
```

Figure 6: Violation of Uniqueness Rule 2

references to the unique return value. This requirement would be violated if we permitted the return value in a unique-returning function to alias its (non-unique) parameters.

To summarise: *values are consumed by being the source in a* `let-with`*, or by being passed as a unique parameter in a function call.* We can crystallise valid usage in the form of three principal rules:

Uniqueness Rule 1 When a value is passed in the place of a unique parameter in a function call, or used as the source in a `let-with` expression, neither that value, nor any value that aliases it, may be used on any execution path following the function call. A violation of this rule is illustrated on Figure 5.

Uniqueness Rule 2 If a function definition is declared to return a unique value, the return value (that is, the result of the body of the function) must not share memory with any non-unique arguments to the function. As a consequence, at the time of execution, the result of a call to the function is the only reference to that value. A violation of this rule is illustrated on Figure 6.

Uniqueness Rule 3 If a function call yields a unique return value, the caller has exclusive access to that value. At *the point the call returns*, the return value may not share memory with any variable used in any execution path following the function call. This rule is particularly subtle, but can be considered a rephrasing of Uniqueness Rule 2 from the "calling side".

Finally, it is worth emphasising that everything in this chapter is used as part of a static analysis. *All* violations of the uniqueness rules will be discovered at compile time (in fact, during type-checking), thus leaving the code generator and runtime system at liberty to exploit them for low-level optimisation.

3.1 Sharing analysis

Whenever the memory regions for two values overlap, we say that they are *aliased*, or that *sharing* is present. As an example, if you have a two-dimensional array `a` and extract its first row as the one-dimensional array `b`, we say that `a`

and `b` are aliased. While the $\mathcal{L}_0$ compiler may do a deep copy if it wishes[1], it is not required, and this operation thus holds the potential for sharing memory. Sharing analysis is necessarily conservative, and merely imposes an upper bound on the amount of sharing happening at runtime. The sharing analysis in $\mathcal{L}_0$ has been carefully designed to make the bound as tight as possible, but still easily computable.

In $\mathcal{L}_0$, the only values that can have any sharing are arrays - everything else is considered "primitive". Tuples are special, in that they are not considered to have any identity beyond their elements. Therefore, when we store sharing information for a tuple-typed expression, we do it for each of its element types, rather than the tuple value as a whole.

To be precise, sharing information for an expression e, written aliases(e), can take one of two forms:

1. l, where l is a subset of the variables in scope at e. This means that e may share data with some of the variables in l. This is the sharing information when the type of e is not a tuple.
2. $\langle l_1, \ldots, l_n \rangle$, which requires that the type of e is a tuple $\{t_1, \ldots, t_n\}$, and denotes that the sharing of the ith component is l_i. This is the shape of the sharing information when the type of e is a tuple.

We need a way to combine sharing information. The typical case is computing sharing information for the expression `if c then e1 else e2`, where the sharing of the resulting value is the "combination" of the sharing in both `e1` and `e2`. We make this combination precise by the associative, commutative operation $s_1 \oplus s_2$, which is defined by the following equation.

$$l_1 \oplus l_2 = l_1 \cup l_2$$
$$\langle l_1, \ldots, l_n \rangle \oplus \langle l_{n+1}, \ldots, l_{2n} \rangle = \langle l_1 \oplus l_{n+1}, \ldots, l_n \oplus l_{2n} \rangle$$

Now we can define

$$\text{aliases}(\texttt{if c then e1 else e2}) = \text{aliases}(\texttt{e1}) \oplus \text{aliases}(\texttt{e2}).$$

We will often treat sharing information as a set and write things such as $\forall v \in \text{aliases}(e).p(v)$ – in these cases, the set elements are all variables contained anywhere in the sharing information.

Aliasing is transitive – if $v \in \text{aliases}(e)$ and $v' \in \text{aliases}(e)$, then $v \in \text{aliases}(v')$. Aliasing is mostly intuitive - during type-checking, the symbol table contains not only the type of each variable, but also which other variables it may alias. Hence, we can define an aliasing rule for variables:

$$\text{aliases}(v) = \{v\} \cup \{\text{Any variable in scope that aliases } v\}$$

[1] At some point, a proper cost model for $\mathcal{L}_0$ will be developed, and it is very likely that we require such indexing to be $O(1)$.

The aliasing rules for other expressions are mostly intuitive, but a few interesting cases are presented here:

$$\begin{aligned}
\text{aliases}(e) &= \emptyset \text{ (Whenever } e \text{ has a basic type}^2\text{)} \\
\text{aliases}(a\texttt{[i]}) &= \text{aliases}(a) \\
\text{aliases}(\texttt{copy(}e\texttt{)}) &= \emptyset \\
\text{aliases}(\texttt{if } c \texttt{ then } e_1 \texttt{ else } e_2) &= \text{aliases}(e_1) \oplus \text{aliases}(e_2) \\
\text{aliases}(\texttt{transpose(}e\texttt{)}) &= \text{aliases}(()e)
\end{aligned}$$

Note that `transpose` introduces aliasing - this is to permit an implementation where the transposed array is never actually manifested in memory, but is merely an index space transformation of the underlying array resolved at compile-time. The operations `reshape`, `split`, etc. have a similar rule.

The rule for function application is more complicated. To begin with, and this was indeed the original rule in $\mathcal{L}_0$, we can state that the return value of a function call aliases all of its arguments.

$$\text{aliases}(f(e_1, \ldots, e_n)) = \bigcup_{1 \leq i \leq n} \text{aliases}(e_i) \qquad \textit{(–Too restrictive!)}$$

However, it turns out that this is far too restrictive. Consider a call `f1(`a`)` to the function `f1` whose type is shown on Figure 7 - if the return value aliased the argument a, then we could never use the return value at all, as it would alias something that has been consumed, namely the parameter a:

```
let x = f1(a) in // Now 'x' would alias 'a'.
x                // Violates Uniqueness Rule 1,
                 // as something aliasing 'a' is accessed
```

Hence, a first elaboration is that the return value should only alias those function arguments that are not consumed:

$$\text{aliases}(f(e_1, \ldots, e_n)) = \bigcup_{1 \leq i \leq n, e_i \text{ is not consumed}} \text{aliases}(e_i) \quad \textit{(–Still too restrictive!)}$$

The argument for the soundness of this rule is as follows: even if the return value may at runtime alias a consumed argument, we do not need to record it, as that argument will never be accessed elsewhere.

Unfortunately, the above rule is still too restrictive, as can be illustrated by function `f2` from Figure 7. Consider a call `f2(`a`)` - by the above rule, the return value would be aliased to a, which would violate Uniqueness Rule 3, as a may be used again.

Hence, we add another elaboration, wherein the alias set is empty if the return value is unique.

```
fun [int] f1(*[int] a) = ...

fun *[int] f2([int] a) = ...
```

Figure 7: Unique arguments

$$\text{aliases}(f(e_1, \ldots, e_n)) = \begin{cases} \emptyset & \text{If } f \text{ returns an unique value} \\ \bigcup_{\substack{1 \leq i \leq n \\ e_i \text{ is not consumed}}} \text{aliases}(e_i) & \text{Otherwise} \end{cases}$$

The final rule is essentially correct, except that it ignores tuples. As mentioned earlier, sharing information for tuples is represented element-wise. Hence, we can simply apply the above rule piecewise for each element in the tuple.

Although the current aliasing rules for function calls have proven sufficient for now, there are cases where it is too conservative. Consider the following function.

```
fun [int] contrived([[int]] src, [int] indexes, int i) =
  src[indexes[i]]
```

In a call `contrived(src,indexes,i)`, by the above rules, we would consider the return value to be aliased to *both* `src` and `indexes`, as both are non-consumed parameters. Yet, it is clear by inspecting the actual function definition that the return value will only index the `src` parameter.

This problem is not solvable merely through refinement of the aliasing rules - either the user must annotate each function with information about which of the parameters may be aliased by the return value, or the compiler could deduce it using some sharing inference algorithm. As the latter would add a great deal of complexity, and the former require a language change, we have postponed tackling this problem until it becomes a problem in practice.

3.2 Tracking uniqueness

Let us summarise:

- If the type of an array parameter is preceded by a single asterisk, it denotes that the array is unique, i.e., that it will never be reused outside of the current function.
- The source operand to a `let-with` *must* be unique. If it is not, it is reported as a type error.

Let-with and function calls are the only places in which variable consumption can happen. As a first example, let us consider a function that replaces the value at a given position in an integer array.

```
fun *[int] replace(*[int] arr, int i, int x) =
  let arr[i] = x in arr
```

```
let b = a in                // Now b ∈ aliases(a).
let c = a with [i] <- x in // ∀v ∈ aliases(a) ⇒ Mark v as consumed.
b                           // Error, because b ∈ aliases(a)!
```

Figure 8: Example of array consumption

The type of this function expresses the fact that it consumes its array argument, and also returns a unique array. This permits composition - `replace(replace(a, i1, x), i2, y)` is a valid application. Defining `replace` as

```
fun [int] replace2(*[int] arr, int i, int x) =
  let arr[i] = x in arr
```

would still be type correct (a unique array can be used anywhere a nonunique is expected), but the composition `replace2(replace2(a, i1, x), i2, y)` would no longer be well typed.

Checking that uniqueness invariants are being upheld is far subtler than normal type checking. In particular, detailed sharing analysis has to be performed, in order to ensure that after an array a is consumed, it becomes an error to use any value that may refer to (parts of) the old value of the array. Whenever we consume a variable a, we mark as inaccessible all of its aliases, as illustrated on Figure 8.

A key principle is that of *sequence points* that lexically checkpoint the use of variables. As an example, assume that we are given a function `f` of type `*[int] -> int`. That is, `f` consumes an array and returns an integer. The expression

```
f(a) + a[i]
```

is invalid because a consumption and observation of the same variable happens within the same *sequence*. It is valid for a sequence to contain multiple observations of the same variable, but if a variable is consumed, that must be the only occurrence of the variable (or any of its aliases) within the sequence. Binding constructs (lets, let-withs and loops) create sequence points that delimit sequences. If we rewrite the expression to coordinate the consumption into its own sequence, all will be well.

```
let c = a[i] // Since a[i] is of primitive type,
             // c does not alias a.
in f(a) + c
```

The reason for this rule is to enable simpler code generation, as any necessary order of operations is evident in the code. It does require a certain amount of care when doing program transformations, as for example expression reordering may result in an invalid program, as shown on Figure 9 and discussed in further detail in Chapter 7.

In the previous examples, function arguments that were consumed were all simple variables, making it easy to describe what was being consumed. But in general, we might have an expression

```
let x = a[0] in          let b = a with [i] <- y in
let b = a with      ⇒    let x = a[0] in // Error:
  [i] <- y in                              // violates Uniqueness Rule 1
x + b[1]                 x + b[1]
```

Figure 9: Expression reordering causing violation of uniqueness rules

```
replace(e, i, x)
```

where `e` is some arbitrary expression. In this case, we mark as consumed all variables in aliases(`e`).

Constant, literal arrays are not considered unique, as the compiler may put them in read-only memory and return the same reference every time they are accessed. For example, the following program is invalid.

```
fun [int] fibs(int i, int x) =
  let a = [1, 1, 2, 3, 5, 8, 13] in
  let a[i] = x in a
```

Since `a` is not unique, its use in the let-with is a type error. However, we can use `copy` to create a unique duplicate of the array.

```
fun [int] fibs(int i, int x) =
  let a = copy([1, 1, 2, 3, 5, 8, 13]) in
  let a[i] = x in a
```

If we have a function such as

```
fun int f(*[int] a, int x) = x
```

then it is not valid to curry it in such a way that we provide values for the consumed parameters. For example, `map(f (`a`), b)` would be an error. The reason for this is that `f` may be called an arbitrary number of times during the mapping, but a can only be consumed once.

Chapter 4

Internal Representation

It is a common compilation technique to transform the externally visible language into a simpler intermediate language, on which all optimisation and further compilation is performed. This is usually necessary, because languages written for human consumption have a large amount of bells and whistles that make programming more convenient, but offer no significant avenues for optimisation, but rather just leave more cases for the optimiser to handle.

We do not suffer as badly from that problem with $\mathcal{L}_0$, as it was designed as an intermediate language itself. Nevertheless, before embarking on any optimisation, we do transform the input program to an internal dialect of $\mathcal{L}_0$. In most compilers, it is usually not possible to transform the intermediate language to the external language, at least not without losing much of the structure of the original program. In comparison, the transformation from external to internal $\mathcal{L}_0$ is comparatively simple and reversible (with some loss of information; see Section 4.3). Internal $\mathcal{L}_0$ also supports a mechanism for making usually implicit checks explicit, such as bounds checks when indexing arrays, in some cases permitting their optimisation through standard optimistaions. This is described in Section 4.2.

A very important first notice: All names in internal $\mathcal{L}_0$ are *globally unique*. This means that we will never have name shadowing, and we can uniquely identify, say, a `let`-binding by one of the names it binds.

4.1 Tuple Transformation

The principal difference between external and internal $\mathcal{L}_0$ is that the latter does not permit arrays of tuples. There are two reasons for this:

- Dataflow analysis is simplified by removing `zip`/`unzip` expressions - the fusion algorithm in Chapter 7 benefits greatly from this.
- Arrays of tuples cannot in general be efficiently represented in memory due to alignment constraints, and so we would have to remove them at some stage anyway.

An array of tuples type is converted to a tuple containing arrays of the original tuple components, for which the process is then repeated recursively.

Futhermore, tuples are flattened. All other types are left unchanged. A few examples:

```
                [int] ⇒ [int]
          [{int,real}] ⇒ {[int],[real]}
      [{[int], real}] ⇒ {[[int]], [real]}
{{int, bool}, real} ⇒ {int, bool, real}      (Flattening)
```

`zip` and `unzip` are not allowed in internal $\mathcal{L}_0$. Both are removed during the conversion process, as they perform a transformation between representations that is not necessary in internal $\mathcal{L}_0$, although `zip` has some additional complications that are discussed in Section 4.2.

Most expressions operating on arrays of tuples have to be modified to operate on the components of the replacement tuples-of-arrays instead, but the transformation is relatively trivial, so we will not describe it in detail.

Internal $\mathcal{L}_0$ also does not permit tuple-typed variables. Hence, every pattern that binds a variable to a tuple must be converted to explicitly name the elements of the tuple, e.g:

```
let x = {f(a), g(b)} in
...
  ⇓
let {x1,x2} = {f(a), g(b)} in
...
```

4.1.1 Tupleless SOACs

As internal $\mathcal{L}_0$ does not permit tuples of arrays, we need to find a way to convert an expression such as:

```
map(fn int (int,int t) =>
      let x,y = t in
      f(x,y),
    zip(a, b))
```

The solution is tupleless SOACs: variants of the normal SOACs in which several arrays are traversed in parallel. For example, the previous expression will be converted to:

```
mapT(fn {int} (int x, int y) =>
       f(x,y),
     a, b)
```

Conceptually, we can imagine that the tupleless SOACs all have a built-in `zip`. Furthermore, initial values for tuple-typed accumulators (for `reduceT`, `scanT` and `redomapT`) are given element-wise, and also passed element-wise to the function. This, combined with the ban on arrays of tuples, implies that no parameter of a function in a tupleless SOAC is ever tuple-typed.

```
e  ::=  <c>mapT(fn t (t1 v1, ..., tn vn) => ef, e1, ..., en)
      | <c>filterT(fn t (t1 v1, ..., tn vn) => ef, e1, ..., en)
      | <c>reduceT(fn t (t1 v1, ..., tn vn) => ef, {x1, ..., xn}, e1, ..., en)
      | <c>scanT(fn t (t1 v1, ..., tn vn) => ef, {x1, ..., xn}, e1, ..., en)
      | <c>redomapT(fn to (to1 vo1, ..., ton von) => eo,
                    fn ti (ti1 vi1, ..., tin vin) => ei,
                    {x1, ..., xn}, e1, ..., en)
```

Figure 10: Tupleless Second-order array combinators

As a minor detail, curried functions are also not permitted in tupleless SOACs, although we will still use them for notational convenience. The notation is summarised on Figure 10 - the `<c>` notation is explained in the next section.

4.2 Assertions

Removal of `zip` carries with it an additional complication. Recall that `zip` is responsible for checking that the input arrays are of the same (outer) size, and if they are not, terminate the program with an error. The obvious solution is to make each tupleless SOAC do the same check, but that would be wasteful of computer resources. Even worse, it's not a full solution to the problem. Consider the following expression.

```
let c = zip(a,b) in
c[0]
```

How should this be transformed? There is an obvious solution:

```
{a[0], b[0]}
```

But it is of course wrong - there is no checking that `a` and `b` are of the same length, hence the original expression may fail, while the transformed expression may evaluate successfully. The final solution is to *predicate* the expression on the fact that `a` and `b` must match.

```
if a and b have the same length
then {a[0], b[0]}
else fail
```

Using branches to accomplish this somewhat complicates further transformation and optimisation however, so an approach based on *assertions* was chosen. First, we introduce a new type, `cert`, which is inhabited by only one value `Checked`. The idea is that a value of type `cert` acts as a *certificate*, certifying that some property has been checked. We then add three new language constructs:

`assert(`e`)`
: Returns `Checked` if the boolean expression e returns `True`, otherwise terminates the program with an error.

`conjoin(`$e_1, \ldots, e_n$`)`
: All of $e_1, \ldots, e_n$ must be expressions of type `cert`. Always returns `Checked`. The purpose of `conjoin` is to combine several `cert` values into one.

`<`$v_1, \ldots, v_n$`>`e

A predicated expression. Evaluate e and return its value. The variables $v_1, \ldots, v_n$ are certificates: variables that must each be of type `cert`. They exist solely to express a dependency between the expression e and whichever assertions it is predicated upon, which implies that on any execution path leading to e, the assertions computing the certificates will have been executed first.

Now the index expression given above can be transformed into the following.

```
let c = assert(size(0,a) = size(0,b)) in
{<c>a[0], <c>b[0]}
```

An advantage of this approach is that the association between assertions and predicated expressions is given by the same variable-usage rules that govern all other expressions, and it is therefore not necessary to handle `assert` specially in the various transformations. It is, however, necessary to maintain the list of certificates when transforming a predicated expression, but it turns out that this is simple in practice. Essentially, the rule of thumb is that every expression derived one or more predicated expressions must be predicated likewise.

The assertion design also has the particularly convenient property that even if later transformations pick apart the tuple literal, the individual components will still be predicated on the original property. Even more importantly, the condition `size(0,a) = size(0,b)` is seen as a perfectly ordinary expression by the rest of the compiler, and can be simplified (or removed altogether) by size analysis, copy propagation, constant folding, common-subexpression elimination and other standard optimisations.

One thing is worth noting: The assertion expression, or the value that it computes, has no recognisable meaning by itself. It is not a proposition in the sense of formal logics, and we can only determine which property it guarantees by the way in which it is used. This limits what kinds of information the compiler can deduce from an assertion, compared to using a real theorem prover or dependent type system. In particular, we can perform only very limited static checking, where through simplification we are able to obtain the expression `assert(False)`, although even then we cannot be certain that the assertion is not dead code. Hence, the assertion system functions solely at run-time. Nevertheless, the power-to-weight ratio is very high for this design, and it suits our purpose well.

Having introduced an assertion mechanism to solve the problem of tupleless SOACs, it is of course worth to consider whether it can be used for other purposes as well. As it turns out, we can use assertions to express bounds-checking (using a somewhat ugly syntax):

```
a[i]    ⇒ let c = assert(0 <= i && i < size(0,a)) in
          a[<c> | i]
```

Again, what we gain is the ability to exploit our standard expression-optimisation machinery to simplify and perhaps even remove the bounds check. In particular, range analysis can be used to hoist such expressions out of inner loops - all the while staying within the (internal) $\mathcal{L}_0$ language. This will be covered in greater detail in Section 6.1.

t	::=	`cert`	(Certificate)
k	::=	`Checked`	(Always-true certificate)
c	::=	$v_1, \ldots, v_n$	(Sequence of variables)
e	::=	`assert(`e`)`	(Assertion)
	|	`conjoin(`e_1`, ..., `e_n`)`	(Conjoin assertions)
	|	`<`c`>id[`e_1`, ..., `e_n`]`	(Indexing)
	|	`<`c`>size(`k`, `e`)`	(Array length)
	|	`<`c`>reshape((`e_1`,...,`e_n`), `e`)`	(Array reshape)
	|	`<`c`>transpose(`e`)`	(Transposition)
	|	`<`c`>split(`e_1`, `e_2`)`	(Split e_2 at index e_1)
	|	`<`c`>concat(`e_1`, `e_2`)`	(Concatenation)
	|	`let <`c`>`v_1 `= `v_2 `with`	(In-place update)
		`  [<`c`>|`e_1`,...,`e_n`] <- `e_v	
		`in `e_b	
	|	v`[<`c`>|`e_1`, ..., `e_n`]`	(Indexing)

Figure 11: Assertions

4.2.1 Function Calls

One question is left - namely how functions mesh with the assertion system. One solution is to simply require function calls to be predicated. That is, the program

```
fun [int] f([{int,int}] input) =
  map(op+, input)

f(zip(a,b))
```

is converted into

```
fun [int] f([int] input1, [int] input2) =
  mapT(op +, input1, input2)

let c = assert(size(0,a) = size(0,b)) in
f<c>(a,b)
```

This is satisfactory from the point of view of the caller - the function `f` (after undergoing transformation) has the precondition that `a` and `b` are of the same size, and this is indeed checked by the above call. The body of the function is slightly more dubious, as `mapT` is invoked without a certificate that its inputs are of the same size, but as long as the caller takes care to only invoke the function when this precondition is satisfied (which we do in the above case), all will be well.

The real problem occurs if we inline `f`. If we do it naïvely, we get this program:

```
let c = assert(size(0,a) = size(0,b)) in
mapT(op +, a, b)
```

And *now* we have a problem – `c` is not used anywhere, and may thus be moved to after the `mapT` expression, or maybe even removed as dead code! This is bad. A possible solution would be to make the inliner smarter, and modify the inlined function body such that every leaf of its syntax tree is predicated on the same certificates as the original function call. This is a rather clumsy solution however, and may cause unnecessary predication on branches of the function that do not depend on the precondition, which would inhibit some transformations, like hoisting. A more fine-grained solution is needed.

That solution is to embed the certificates directly into the parameter list of the function. A function that originally accepted a single parameter of type `[{int,int}]` will, after conversion to internal $\mathcal{L}_0$, accept three parameters of types `cert`, `[int]`, `[int]`. The example given at the beginning of the section will become:

```
fun [int] f(cert input_c, [int] input1, [int] input2) =
  mapT<input_c>(op +, input1, input2)

let c = assert(size(0,a) = size(0,b)) in
f(c, a,b)
```

Naïve inlining will produce:

```
let c = assert(size(0,a) = size(0,b)) in
mapT<c>(op +, a, b)
```

Which is the desired result.

Return values can be handled in a similar fashion. The program

```
fun [{int,int}] f(int x) =
    zip(iota(x), iota(x))

let a = f(10) in
map(g,a)
```

is converted into

```
fun {cert, [int], [int]} f(int x) =
  let v1 = iota(x) in
  let v2 = iota(x) in
  let c = assert(size(0, v1) = size(0, v2)) in
  {c, v1, v2}

let {c, a1, a2} = f(10) in
mapT<c>(g, a1, a2)
```

The return type of `f` has been modified to include a certificate for the postcondition that the two return arrays have the same outer size.[1]

This solution requires a large amount of tedious tracking of certificates in the module that converts external to internal $\mathcal{L}_0$, but as a tradeoff, the rest

[1] In this specific case, later simplification will eventually result in the assertion being removed and replaced with the literal `Checked`, but it has been retained for clarity in this example.

of the compiler can be kept simpler, as all dependencies between predicated expressions and assertions are explicit.

4.3 Converting from Internal to External $\mathcal{L}_0$

The internal language matches the external quite closely; nevertheless, the transformation from external to internal language does not have a fully defined inverse. That is, given a program in internal $\mathcal{L}_0$, we can compute an equivalent program in external $\mathcal{L}_0$, but it might not be the original program. The steps are simple in principle:

1. Convert tupleless SOACs to ordinary SOACs, `zip`ping the arguments and `unzip`ping the return value.
2. Remove all expressions and bindings of type `cert`, including function parameters.
3. Remove predicates from all predicated expressions.
4. Remove all `assert`s.

The problem is that we cannot reconstruct the original arrays of tuples from the tuples of arrays of internal $\mathcal{L}_0$. We cannot know whether any tuple of arrays we encounter was originally an array of tuples or not. However, as long as the entry point of the program, the `main` function, does not use arrays of tuples as argument or return value, there should be no observable difference. If necessary, we could handle `main` on an ad-hoc basis to preserve its original type.

Part II

Optimisations

Chapter 5

First Order Optimisations

As a data-parallel programming language, most of the interesting optimisations for $\mathcal{L}_0$ naturally revolve around SOACs. Yet, classical optimisations such as copy propagation, constant folding, hoisting and common subexpression elimination (CSE) remain important. For example, they are part of optimising the delayed representation of some constructs, as demonstrated on Chapter 5.

This chapter will cover the implementation of copy propagation and constant folding for $\mathcal{L}_0$. Hoisting and CSE will be covered in Chapter 6.

5.1 Inlining

One property holds for all optimisations performed by the $\mathcal{L}_0$ compiler: They are all strictly intraprocedural. Thus, we rely on aggressive inlining as the first step of the optimisation pipeline, wherein we inline every non-recursive function call. Inlining a large function at multiple call sites can of course result in a tremendous amount of code bloat, but as function calls are in any case usually always inlined on current GPU hardware, due to very little (or no) stack being available, this is perhaps excusable.

After inlining, most functions will be dead, and are summarily removed.

5.2 Let- and tuple-normalisation

At its core, program optimisation is about recognising code patterns, and rewriting them to a more efficient form that retains the meaning of the original code. To make this process simpler, we pre-process the program to give it a more regular structure. The use of internal $\mathcal{L}_0$ as presented in Chapter 4 is an

```
let a = iota(n) in
let b = replicate(m, x) in
let c = transpose(arr) in
a[4] + b[1] + c[i,j]
```

(a) Unoptimised

```
4 + x + arr[j,i]
```

(b) Optimised

Figure 12: Optimising indexing into `replicate`, `iota` and `transpose`

important step in this process, but it is not sufficient by itself. To this end, we use a transformation pass that rewrites needlessly complex program structure into a simpler form. The mechanics behind the transformation are tedious and unimportant (basically a recursive traversal through the syntax tree), and it is best understood by the invariants guaranteed of the resulting program:

- Tuple expressions can appear only as the final result of a function, SOAC, or `if` expression, and similarly for the tuple pattern of a let binding, e.g., a formal argument cannot be a tuple,
- Consecutive `let`, `let-with` and `loop` expressions are at the same nesting level, e.g., e_1 cannot be a `let` expression when used in `let` p `=` e_1 `in` e_2,
- Each `if` is bound to a corresponding `let` expression, and an `if`'s condition cannot be in itself an `if` expression, e.g.,

```
a + if (if c1 then e1 else e2)
    then e3
    else e4
  ⇓
let c2 = if c1 then e1 else e2 in
let b = if c2 then e3 else e4 in a+b
```

- Function calls, including SOACs, have their own `let` binding, e.g., `g(reduceT(f,a))` ⇒ `let y = reduceT(f,e,a) in g(y)`,
- All actual arguments in a function call are vars, e.g., `f(a+b)`⇒`let x=a+b in f(x)`.

Note that we consider "function-like" constructs such as `transpose`, `reshape` and `replicate` to be functions as far as the above invariants are considerned.

5.3 Copy/constant propagation and constant folding

Copy propagation is the mechanism by which we eliminate bindings that are merely copies of existing variables. Constant propagation is the inlining of constant bindings where the bindings are used. Constant-folding is the process of evaluating a constant expression at compile time, for example an addition where both operands are statically known. Figure 13 illustrates the difference between the three processes.

In imperative compilers, these optimisations are usually performed on a program after it has been converted to a basic block graph. However, after undergoing the let/tuple-normalisation described in the previous section, it is easy to perform all three optimisations in tandem directly on the syntax tree of an $\mathcal{L}_0$ program.

The central idea is that we consider some expressions to be *inlineable*. Whenever the RHS of a `let`-expression is inlineable, we substitute any occurrences of the name bound by the binding within the body of the `let`-expression by the RHS (we ignore bindings where the pattern is a tuple-pattern for now). For example, consider this expression:

```
let x = 2 in
let y = 3 in
let z = x in
z + y
  ⇓
let x = 2 in
let y = 3 in
x + y
```

(a) Copy propagation

```
let x = 2 in
let y = 3 in
x + y
  ⇓
2 + 3
```

(b) Constant propagation

```
2 + 3
  ⇓
5
```

(c) Constant folding

Figure 13: Examples of copy/constant propagation and constant folding

```
let b = transpose(a) in
b[i,j]
  ⇓
a[j,i]
```

```
let b = reshape((n,m), a) in
b[i,j]
  ⇓
a[i*m+j] // Assuming 'a' has rank 1.
```

Figure 14: Removing `reshape` and `transpose`

.....

```
let a = 2 in
e
```

We consider the expression `2` (a constant number) to be inlinable, hence we substitute `2` for `a` within e and remove the binding of `a` entirely.

A big question is which expressions to inline. As inlining may duplicate the inlined expression, we should only inline where such duplication will not result in additional computation at run-time. As a start, we can certainly inline variables and non-array constants. Incidentally, this by itself provides copy- and constant-propagation.

As for other expressions, we note that `reshape` and `transpose` operations are entirely index-space transformations, and can thus be handled at compile-time wherever the result of the operation is used. Although uninplemented in the current $\mathcal{L}_0$ compiler, it is envisioned that a transformation similar to the one on Figure 14 will be used by the code generator. Therefore, we freely inline `reshape` and `transpose`. As `iota` and `replicate` can be removed in a similar manner, they are therefore also considered inlineable.

Bindings with tuple-patterns can be inlined under some circumstances, specifically if the RHS is itself a tuple literal, where every component is an inlineable expression.

However, even if an expression is in principle inlineable, there are still three cases that prevent inlining:

- If an array-typed variable is indexed, we need to keep it in the program, unless the replacement expression is itself a variable (that is, copy propagation). This is because $\mathcal{L}_0$ only permits indexing of variables, not arbitrary array-typed expressions.
- If an array-typed variable is used as the source in a `let-with`, we again need to keep its binding in the program.

- If a variable cannot be substituted with an expression for some other reason (notably, because it would violate the `let`-normalisation properties from Section 5.2), we also cannot remove its binding.

Apart from substituting variables, we also look at other expressions to determine whether constant folding is possible. This is done by a bottom-up traversal of the syntax tree, where each expression is processed as follows:

`x binop y`
: If x and y are literals, compute and substitute with the result.

`unop x`
: If x and y is a literal, compute and substitute with the result.

`size(`k`, a)`
: Depending on k and how a was bound, we may be able to replace the `size` expression.

  - If $k = 0$ and `a` is the result of `iota(n)` or `replicate(n,e)`, we can substitute with simply `n`, as that is the size of `a`.
  - If `a` is a literal constant, we can substitute with the exact size.

`let pat = e in body`
: If, after transforming *body*, none of the names in *pat* are used, remove the binding.

`if c then a else b`
: If c can be constant-folded to either `True` or `False`, replace with the corresponding branch.

`f(...)`
: If all parameters to a function call are literal values, we use the interpreter to evaluate the function and insert its return value. At the moment, we assume that the function will terminate, although this assumption is not really justifiable. Instead, we should probably only evaluate non-recursive functions.

`assert(True)`
: Replace with `Checked`.

`<`c_1`,...,`c_n`>e`
: Remove any certificate c_i that is bound to `Checked` (i.e. a certificate that is always true).

`conjoin(`c_1`,...,`c_n`)` hfill
: Remove any certificate c_i that is bound to `Checked` (i.e. a certificate that is always true).

`a[i]`
: There are several potential avenues for constant-folding index operations, depending on how `a` was bound:

`a` is bound to a variable `b`: Replace with `b[i]`.

`a` is `iota(n)`: Replace with `i` – note that this may make an invalid program valid, as we remove the bounds check $\texttt{i} < \texttt{n}$. We could use the assertion mechanism from Section 4.2 to bring it back, but this is not done in the current compiler.

`a` is `b[j]`: Replace with `b[j,i]`.

`a` is an array literal: Replace with the corresponding element in the array literal.

`a` is `replicate(n, v)`: Replace with `v`.

`a[i,j]`

If more than one index is given, we can handle the same cases as above (although indexing into e.g. an `iota` would of course be a type error), as well as a few more:[1]

`a` **is** `transpose(b)`**:** Replace with `b[j,i]`.

`a` **is** `replicate(n,iota(m))`**:** Replace with `j`.

[1] For simplicity, we treat only the case where two indices are given. The implementation in the $\mathcal{L}_0$ compiler supports an arbitrary number of indices.

Chapter 6

The Rebinder

In the compiler literature, *hoisting* (also known as *loop-invariant code motion*) is the movement of loop-invariant expressions out of a loop. This has a clear benefit: rather than executing once per iteration of the loop, the expression is executed once before the loop begins. For $\mathcal{L}_0$, hoisting can be an important optimisation, as it holds the potential for moving bounds checks and other assertions out of inner loops. We will see an example of this in the next section.

Common Subexpression Elimination (henceforth referred to as CSE) is a popular compiler optimisation that identifies identical expressions (i.e. expressions that always evaluate to the same value), and replaces them with a variable holding the computed value. If the common expressions are expensive or computed very frequently, e.g. by being part of an inner loop, this can result in significant speedup.

It turns out that hoisting and common subexpression elimination can be unified in a single framework, which in the $\mathcal{L}_0$ compiler is termed the *Rebinder*.

This chapter will start out by describing basic principles of hoisting and CSE in Sections 6.1 and 6.2. In Section 6.3, we will describe their implementation in the $\mathcal{L}_0$ compiler.

6.1 Hoisting

When compiling an imperative language, we must be careful not to move any code with side effects, but in a pure language such as $\mathcal{L}_0$, we can hoist freely (with a few restrictions that I'll get to in Section 6.1.1). A simple example of hoisting in action is shown on Figure 15.

At first glance, hoisting may seem to apply too rarely to be of much benefit, since most programmers would put loop-invariant code outside of the loop in

```
map(fn int (int x) =>                  let k = y + z in
      let k = y + z in                 map(fn int (int x) =>
      x + k,                                 x + k,
    a)                          ⇒          a)
```

Figure 15: Hoisting in action

the first place. However, there are two important use cases that do not involve programmer-written code:

- Much $\mathcal{L}_0$ code is not written by the programmer, but is rather the result of program transformation by the compiler. Inlining and constant folding may easily result in the creation of loop-invariant expressions within a loop.
- Explicit bounds checks, as introduced in Section 4.2, can sometimes be hoisted out of inner loops.

The latter case merits futher elaboration. Consider the following program:

```
map(fn int (int i) =>
      a[i] + a[i*2],
    iota(n))
```

Here, `a` is a free variable. Once the compiler has turned the implicit bound checks explicit, the program will look like this:

```
map(fn int (int i) =>
      let c1 = assert(i   >= 0 && i   < size(0,a)) in
      let c2 = assert(i*2 >= 0 && i*2 < size(0,a)) in
      a[<c1>|i] + a[<c2>|i*2],
    iota(n))
```

Now, the assertions are not loop-invariant, as they depend on `i`. If we assume a sufficiently smart compiler, for example by employing some extension of *symbolic range propagation*[10], we can deduce that the variable i will always be in the range $[0, n-1]$. This allows us to rewrite the assertions – the checks for non-negativity goes away, as it is always true, and we only have to check the upper bound for the maximum values that `i` and `i*2` may attain:

```
map(fn int (int i) =>
      let c1 = assert(n   < size(0,a)) in
      let c2 = assert(n*2 < size(0,a)) in
      a[<c1>|i] + a[<c2>|i*2],
    iota(n))
```

Now `c1` and `c2` are loop-invariant, and we can move them out of the loop body, and perform bounds checking just once, before entering the loop:

```
let c1 = assert(n   < size(0,a)) in
let c2 = assert(n*2 < size(0,a)) in
map(fn int (int i) =>
      a[<c1>|i] + a[<c2>|i*2],
    iota(n))
```

For a simple loop such as the above, the potential benefits are great, as most of the instructions of the original loop body was devoted to bounds checkings.

The $\mathcal{L}_0$ compiler does not yet support the range analysis that enables the critical rewrite of the `assert` expressions. An unpublished bachelors thesis by

Jonas Brunsgaard and Rasmus Wriedt Larsen suggests that the technique works in practice, but their work has not yet been merged with the main compiler code base.

Hoisting assertions such as these is useful not only when the program uses explicit array indexing. While the array accesses performed by SOACs are by construction always in-bounds, and therefore do not need dynamic checks, the `assert` expressions we generate when transforming from external SOACs to tupleless SOACs are conceptually identical to bounds checks, and similarly important to hoist. For example, consider this program:

```
map(fn [int] ([int] r) =>
      map(op+, zip(r, b)),
    a)
```

After transformation to internal $\mathcal{L}_0$, we get the following:

```
mapT(fn {[int]} ([int] r) =>
       let c = assert(size(0,r) = size(0,b)) in
       <c>mapT(op+, r, b),
    a)
```

The assertion is not loop-invariant, and range analysis is no help. For this case, structural size analysis (described in depth in Section 9.1.1) reveals that since `r` is a row of `a`, the outer size of `r` (`size(0,r)`) is equal to the inner size of `a` (`size(1,a)`). Thus, the compiler rewrites to:

```
mapT(fn {[int]} ([int] r) =>
       let c = assert(size(1,a) = size(0,b)) in
       <c>mapT(op+, r, b),
    a)
```

The assertion is now loop-invariant and can be hoisted:

```
let c = assert(size(1,a) = size(0,b)) in
mapT(fn {[int]} ([int] r) =>
       <c>mapT(op+, r, b),
    a)
```

The details of how hoisting is implemented in the $\mathcal{L}_0$ compiler is covered in Section 6.3.

6.1.1 What Not to Hoist

Clearly, we can hoist only loop-invariant expressions. Unfortunately, not all loop-invariant expressions are hoistable, and as is often the case when seemingly valid transformations become problematic, constraints imposed by uniqueness types are at fault. Consider the following program:

```
map(fn (int i) =>
      let a = iota(10) in
      f(a, i),
    b)
```

It seems that we should be able to hoist `a` out of the loop like this:

```
let a = iota(10) in
map(fn (int i) =>
      f(a, i),
    b)
```

However, if the function call `f(a,i)` consumes the `a` argument, hoisting would result in an invalid program, as `a` would be consumed multiple times. We need a "freshly allocated" `a` for each iteration of the loop. Hence, we must not hoist a binding out of a loop in which it is consumed.

As a minor, but important point, it is strictly not permitted to hoist out of loops unless it can be proven that the loop will always execute at least one iteration. The $\mathcal{L}_0$ compiler currently ignores this restriction, implicitly assuming that all arrays are non-empty.

Hoisting out of branches

Most compilers generally do not hoist out of branches, as branching is often used to prevent expensive execution of expensive expressions whose value is not needed. On many GPUs however, execution happens in lock-step across all processors. This implies that unless the branch condition computes the same value in all threads, both sides of the branch will have to be executed [20]. This implies that in some cases, hoisting out of a branch does not cause more instructions to be executed, and hoisting might expose the possibility of other optimisations, particularly common subexpression elimination (see Section 6.2).

We should still be careful when hoisting expressions out of the branches of a conditional, as it is possible that the expression may result in an error unless the condition checked by the conditional is true. For example, consider this expression:

```
if c then y / x
     else if p then y / x
               else 0
```

If `y / x` was hoisted out of the branch, the resulting program might end up dividing by zero. Assertions and array indexing are other operations that are not safe to hoist out of a branch. Most expressions are safe however, and the $\mathcal{L}_0$ compiler aggressively hoists these out of branches. Depending on improvements in hardware, or the targeting of $\mathcal{L}_0$ for non-GPU systems, it is likely that this strategy will need to be revised.

Performance Considerations

There is another potential case where hoisting, while not resulting in an invalid program, proves detrimental rather than beneficial to performance. This occurs when retrieving the hoisted value from memory would be more expensive than re-computing it for each loop iteration, which is particularly likely to occur on GPUs, as global memory accesses are enormously expensive. Balancing this problem is not currently tackled by the $\mathcal{L}_0$ compiler, which instead hoists as

aggressively as possible. Note that this is not a problem when hoisting assertions, such as bounds checks, as the resulting values are not actually accessed from within the loop.

6.2 Common Subexpression Elimination

Common Subexpression Elimination (henceforth referred to as CSE) is a popular compiler optimisation that identifies identical expressions (i.e. expressions that always evaluate to the same value), and replaces them with a variable holding the value. For example, this program:

```
2 * x + 2 * x
```

Can be transformed through CSE into:

```
let tmp = 2 * x in
tmp + tmp
```

This saves us a multiplication. As with hoisting, CSE can potentially be detrimental to performance if it increases memory pressure, but again like hoisting, this is not something the $\mathcal{L}_0$ compiler currently takes into consideration.

We must be careful not to perform CSE such that we end with a violation of the uniqueness rules. Consider this program:

```
let a = iota(10) in
let b = iota(10) in
let a[2] = 5 in
f(a,b)
```

Since `iota(10)` appears in two places, we might be tempted to factor it out:

```
let tmp = iota(10) in
let a = tmp in
let b = tmp in
let a[2] = 5 in
f(a,b)
```

However, this violates Uniqueness Rule 1, as `b` is aliased to `a`, yet is used after `a` is consumed in a `let-with` expression. The solution is to not perform CSE on expressions whose result is eventually consumed – or more conservatively, never perform CSE on an expression of type `*[`α`]`. The latter is easier to implement, although too conservative, but is what the $\mathcal{L}_0$ compiler currently does.

6.3 Rebinder Implementation

Conceptually, hoisting and CSE are rather different transformations. However, they both depend on identifying subexpressions that can be moved (in the case of hoisting) or replaced (in the case of CSE), but where the actual expression rewriting is quite simple. In the $\mathcal{L}_0$ compiler, the observation was made that a lot of the machinery used to implement hoisting could be easily extended to also

perform CSE, and thus was born a compiler pass with the rather idiosyncratic name *the Rebinder*.

The central idea is to assume a program in a slightly modified A-normal form [35], a format similar to *three address code*, where the definition of each `let`-binding is a *simple expression*, and the body of a `let`-binding is either another binding or a variable. A simple expression is either a branch[1], or an expression where all subexpressions are variables or constants (except for the bodies of SOAC functions), which implies that their execution terminates immediately. For example, the following program:

```
fun real solve(real a, real b, real c) =
  (-b + sqrt(b*b - 4.0*a*c)) / (2.0*a)
```

Would look like this in A-normal form. Note that it is also `let`-normalised (Section 5.2):

```
fun real solve(real a_0, real b_1, real c_2) =
  let negate_3  = -b_1 in
  let times_4   = b_1 * b_1 in
  let times_5   = 4.0 * a_0 in
  let times_6   = times_5 * c_2 in
  let minus_7   = times_4 - times_6 in
  let norm_8    = sqrt(minus_7) in
  let plus_9    = negate_3 + norm_8 in
  let times_10  = 2.0 * a_0 in
  let divide_11 = plus_9 / times_10 in
  divide_11
```

This simplifies hoisting and CSE significantly, as the problem is now reduced to moving nodes in the syntax tree (for hoisting) and substituting definitions of `let`-bindings (for CSE). A-normalisation is performed by a separate pass before entering the Rebinder, and is generally trivial, but there are a few difficulties that I will cover in Section 6.3.3.

In order to keep the exposition simpler, `loop` and `let-with` will be ignored (except with respect to upholding uniqueness constraints), and only discuss hoisting of normal `let`-bindings.

To try to give an intuition of the Rebinder, let us consider the following contrived program:

```
fun int main([int] a, int i, int v) =
  let res =
    reduceT(fn {int} (int sum, int x) =>
              sum + x + v*a[i],
            {v*a[i]}, a) in
  res
```

The goal is to hoist the loop-invariant expression `v*a[i]` out of the loop, and use CSE to combine it with the initial value of the accumulator. To this end, it is first transformed to A-normal form:

[1] Not permitted in "standard" A-normal form.

```
fun int main([int] a_0, int i_1, int v_2) =
  let index_10 = a_0[i_1] in
  let times_11 = v_2 * index_10 in
  let {res_5} =
    reduceT(fn {int} (int sum_3, int x_4) =>
              let plus_6 = sum_3 + x_4 in
              let index_7 = a_0[i_1] in
              let times_8 = v_2 * index_7 in
              let plus_9 = plus_6 + times_8 in
              plus_9,
            {times_11}, a_0) in
  res_5
```

The intuition we will use is to strip an expression e of any enclosing bindings, resulting in a "core" expression e', and a set of bindings. There may be free variables in e' that are bound by the bindings in the set. At some point, we will have to insert the bindings in the program, but we will try to put them as far up the syntax tree as possible. For example, stripping the body of `main` above would result in the core expression `res_5` and the bindings `index_10`, `times_11`, and `res_5`.

The set of bindings, which we will term the *potentially hoistable set*, is a partially ordered set of binding pairs (p_i, e_i). A binding pair (p_i, e_i) corresponds to the $\mathcal{L}_0$ binding `let` p_i `=`e_i. The partial order is $\preceq$: for two bindings b_i, b_j, $b_i \preceq b_j$ if b_j uses any variables bound by b_i, or if $b_i = b_j$. The potentially hoistable set thus represents an acyclic data-dependency graph.

The Rebinder proceeds with a recursive walk down the syntax tree, collecting bindings into the potentially hoistable set. Additionally, for each binding, we recurse down its right-hand side in order to determine whether it contains any *hoistable subexpressions*. This is only the case if the RHS is a SOAC – where we can hoist out of the body – or `if` – where we can hoist out of the branches. For all other expressions, due to the program being in A-normal form, the subexpressions will be variables or constants, which are not hoisted.

A traversal of the example program listed above follows:

- First, we encounter the `index_10` and `times_11` bindings, and their right-hand sides are inspected. Neither of these inspections yield hoistable subexpressions, but we remove `index_10` and `index_11` themselves and insert them into the potentially hoistable set.
- Next, we encounter the `res_5` binding and we descend recursively into the scope of the SOAC function body:
  - We inspect the right-hand sides of `plus_6`, `index_7`, `times_8` and `plus_9`, none of which yield hoistable subexpressions. We collect these bindings into a potentially hoistable set and remove them from the function body, leaving just the expression `{plus}`.
  - We are now done inspecting the function, and we need to decide which of the bindings in the potentially hoistable set can in fact be hoisted out. The function parameters are `sum_3` and `x_4`, and any

bindings that depend on these cannot be hoisted (the details are given in Section 6.3.1). This leaves the bindings for `index_7` and `times_8` as hoistable; while `plus_6` and `plus_9` are re-inserted into the program.

This yields the `index_7` and `times_8` bindings as new elements in the potentially hoistable set. We also add the modified `res_5` binding itself.

- Since there are no more bindings left, and we are at the top level of a function, we insert the bindings in the potentially hoistable set (`index_10`, `times_11`, `res_5`, `index_7` and `times_8`) in the program, yielding:

```
fun int main([int] a_0, int i_1, int v_2) =
  let index_10 = a_0[i_1] in
  let times_11 = v_2 * index_10 in
  let index_7 = a_0[i_1] in
  let times_8 = v_2 * index_7 in
  let res_5 =
    reduceT(fn {int} (int sum_3, int x_4) =>
              let plus_6 = sum_3 + x_4 in
              let plus_9 = plus_6 + times_8 in
              {plus_9},
            {times_11}, a_0) in
  res_5
```

We could now do a separate CSE pass over the entire program, but there may be a more efficient strategy. The Rebinder is able to perform the CSE optimisation whenever we insert the non-hoistable bindings into the syntax tree. We will try to provide an intuition for the approach, using the above example:

When, at the end, we have to insert bindings for `index_10`, `times_11`, `res_5`, `index_7` and `times_8`, we have to determine an order that does not result in a variable being used before it is defined. This is easy, since the potentially hoistable set is already dependency-ordered and thus defines a data dependency graph (shown on Figure 16). The following insertion order is obtained by performing a depth-first traversal of the graph:

1. `index_10`
2. `times_11`
3. `index_7`
4. `times_8`
5. `res_5`.

None of these bindings can be removed, as the names they bind may be used in subexpressions, but their right-hand sides (RHS) can be changed freely. This is what is exploited to perform CSE. The following steps are performed:

`index_10`**:** Inserted unchanged, but we record its RHS, in case we end up seeing an identical expression later on.

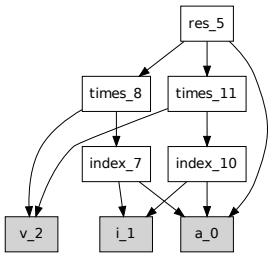


Figure 16: Data dependency graph for example program

`times_11`**:** Likewise inserted unchanged, as its RHS does not correspond to any previously seen.

`index_7`**:** Since the RHS of this binding is identical to the RHS of `index_10`, we replace the RHS of `index_7` with the variable `index_10`. We insert the resulting binding `let times_11 = index_7` and record the substitution `index_y` $\rightarrow$ `index_10`.

`times_8`**:** The substitution `index_y` $\rightarrow$ `index_10` is performed on the immediate subexpressions of the RHS, obtaining `v_2 * index_10`, then check whether the resulting expression has been seen before. This turns out to be identical to the RHS of `times_11`, and therefore we insert the binding `let times_8 = times_11`.

`res_5`**:** The substitution `index_y` $\rightarrow$ `index_10`, `times_8` $\rightarrow$ `times_11` is performed, although no changes are made. The binding is then inserted.

The result is the following program:

```
fun int main([int] a_0, int i_1, int v_2) =
  let index_10 = a_0[i_1] in
  let times_11 = v_2 * index_10 in
  let index_7 = index_10 in
  let times_8 = times_11 in
  let res_5 =
    reduceT(fn {int} (int sum_3, int x_4) =>
              let plus_6 = sum_3 + x_4 in
              let plus_9 = plus_6 + times_8 in
              {plus_9},
            {times_11}, a_0) in
  res_5
```

Copy propagation can then be used to remove the `index_7` and `times_8` bindings.

6.3.1 Hoisting Bindings

The problem is as follows: We are given a potentially hoistable set, $\{(p_i, e_i)\}$, of patterns p_i and e_i. Each pattern p_i defines several names that may be used by other expressions in the set.

We need to split the potentially hoistable set into a *hoistable set*, and a set of the bindings that cannot be hoisted further, the *unhoistable set*. To this end, we are given a unary relation $\mathcal{B}$, that given a binding (p_i, e_i), tells us whether e_i is blocked from being hoisted. As the most archetypical reason, $\mathcal{B}(p_i, e_i)$ whenever e_i uses a parameter of the function that we are trying to hoist out of, but the relation also checks the other cases mentioned in Section 6.1.1. We will consider as unhoistable any binding that is unhoistable according to $\mathcal{B}$, as well as any binding that depends on an unhoistable binding.

Each of the expressions e_i may use variables from any of the patterns p_j (except its own), but there may also be free variables that are not contained in any pattern in the potentially hoistable set. These correspond to names bound higher up in the program.

Since the potentially hoistable set is partially ordered, we can process its elements in dependency order. We keep track of the names bound by unhoistable bindings in an *unhoisted name set*, initially empty. For each binding (p_i, e_i), we check whether $\mathcal{B}(p_i, e_i)$ or e_i uses a name in the unhoisted name set:

- If so, the binding is put into the unhoistable set, and the names in p_i are added to the unhoisted name set.
- Otherwise, the binding is put into the hoistable set.

The end result is a set of hoistable bindings, and a set of nonhoistable bindings. Because of the ordering, we are guaranteed that no binding in the former uses a name bound in the latter.

6.3.2 Inserting Bindings

After dividing the potentially hoistable set into hoistable and non-hoistable bindings, we have to insert all non-hoistable bindings around the core expression. Concretely, we are given a core expression e_c and an ordered set of expressions $\{(p_i,\ e_i)\}$, where binding $(p_i,\ e_i)$ must precede binding $(p_{i+1},\ e_{i+1})$. We can take this opportunity to perform dead code elimination by removing any binding $(p_i,\ e_i)$ that is not used in either e_c or any enclosed binding. The idea is to track which names are actually used in the lexical scope of the binding, and remove bindings that are never referenced.

The algorithm is as follows. We will track two pieces of data: A set $\mathcal{F}$, which is initialised to the free variables of e_c [2]. Then we proceed *backwards* through the list of bindings, i.e., we first process the binding that should be innermost. For each binding $(p_i,\ e_i)$, we check whether any of the names bound by p_i are present in $\mathcal{F}$. If so, we add the free variables of e_i to $\mathcal{F}$ and insert the binding. If the binding is not used, we skip it.

[2] This can be done efficiently if the rebinder constantly tracks the free variables of the core expression.

```
let a = x * y in
let b = a * z in
b
```

(a) `(x * y) * z`

```
let a = y * z in
let b * x * a in
b
```

(b) `x * (y * z)`

```
let a = z * y in
let b * x * a in
b
```

(c) `x * (z * y)`

Figure 17: Normalisation of syntactically different expressions

6.3.3 Simplification of Expressions

The CSE technique described in the previous sections depends heavily on semantically identical code also having (α-)equivalent bindings. However, in many cases, two expressions may have different syntax trees, yet semantically be the same. For example, consider the expressions `(x*y)*z` versus `x*(y*z)`. If we assume that `*` is associative[3], these expressions will always compute the same value. With respect to A-normal form, the two different normalised expressions are shown on Figure 17.

Despite the syntactical difference between these two expressions, they are clearly semantically identical, and we would like for CSE to remove one of them. Given the way our CSE optimisation works, the only solution is to ensure that equivalent expessions A-normalise to α-equivalent bindings. In general, this is a very hard-problem - in fact, determining whether two expressions will always evaluate to the same result is undecideable, as it reduces to the halting problem. Fortunately, the problem becomes quite tractable with restricted to arithmetic operations, through the use of *simplification* before employing the normaliser.

One solution, which was implemented in an unpublished bachelors thesis by Jonas Brunsgaard and Rasmus Wriedt Larsen, is to simplify arithmetic expressions into a form known as *sum-of-products*. In this form, the top node of the syntax tree for an arithmetic expression is always an addition node with n multiplication children. Each of these multiplication nodes may have m children, which can be arbitary numeric expressions. By ordering the children of each node according to some criterion (say, lexicographically), we obtain a unique tree structure for arithmetic expressions that ignores superficial syntactical differences, such as one shown on Figure 17. This unique tree structure can then be A-normalised into an equally unique set of bindings.

[3]Strictly not the case for floating-point multiplication.

Chapter 7

Fusion

This chapter will outline the principles behind *producer-consumer* loop fusion, describe their implementation in the $\mathcal{L}_0$ compiler, and discuss possible complications and restrictions of our handling of loop fusion.

In producer-consumer fusion, the aim is to merge (or *fuse*) two loops, where the output of the first loop – the producer – is used as input to the second – the consumer. We currently only fuse loops that are expressed via SOACs, not the `loop`-notation. The reason for this is to simplify analysis, as it can be hard to determine in which cases arbitrary `do`-loops can be combined, whereas it is possible to define simple rules for how and when SOACs can be fused.

As a simple case, we can fuse the two loops in

$$\texttt{(map } f\texttt{)} \circ \texttt{(map } g\texttt{)}$$

and get

$$\texttt{map } (f \circ g),$$

thus removing the need to construct an intermediary array for the result of `map` g, and in the context of GPGPU, reducing the likelyhood of global memory accesses, which exhibit high memory latency. We will write "c_1-c_2 fusion" for the case where a fusion is formed with c_1 as the producer and c_2 as the consumer. Therefore, the previous example would be "`map-map`"-fusion.

The rules by which we combine SOACs through fusion is called our *fusion algebra*. We aim at preserving the parallelism of the resulting expressions.

Most fusion algorithms in the literature are unable to handle fusion across `zip`/`unzip`, and more generally the case where the output of a producer is used by several consumers. The algorithm presented in this chapter is capable of fusing such cases whenever possible without duplicating computation, as demonstrated on Chapter 7.

This chapter covers two main themes: Section 7.1.1 describes informally which producer-consumer we can fuse, as well as the form of the resulting SOAC. Section 7.4 describes our aggressive fusion algorithm, in particular when a producer result may be used by multiple consumers, without duplicating computation

```
let b = map(op+(1), a) in
let c = map(op+(2), b) in
let d = map(op+(3), b) in
map(op+, zip(c,d))
```

(a) Unfused

```
map(fn int (int x) =>
      let c = x + 2 in
      let d = x + 3 in
      c + d,
    a)
```

(b) Fused

Figure 18: Fusing multiple consumers without duplicating computation

Producers	Consumers
`map`	`map`
	`reduce`
	`scan`
`filter`	`filter`
	`redomap`

Figure 19: Producers and consumers in $\mathcal{L}_0$

7.1 Fusion in $\mathcal{L}_0$

The language used to describe the fusion algorithm in this chapter is `let`-normalised, internal $\mathcal{L}_0$, as described in Chapter 4 and Section 5.2. We will also assume that all instances of `replicate(n,x)` have been rewritten as `map(fn t (int i) => x, iota(n))`[1]. For clarity, expository examples will use the external $\mathcal{L}_0$.

Figure 19 lists which $\mathcal{L}_0$ SOACs are producers, which are consumers, and which are both. In particular, note that even if we have a `reduce`-expression returning an array, this does not mean that the `reduce`-expression is a producer - because, in our algebra, it cannot be fused into another SOAC expression. The reason is that the output of the reduction is only fully known after the final input array element has been processed. Consider the following program:

```
let b = reduce(fn [int] ([int] acc, int x) =>
                 map(op + (x), acc),
               iota(10), a) in
map(f, b)
```

The contents of the array `b` is not determined until the very last element of `a` has been processed, and thus fusion with the `map`-expression cannot take place. While it is possible to use `reduce` in a way that could theoretically be fused with a consumer (for example by using it to simulate `map`), the analysis necessary to determine whether a given reduction is fusible would be quite onerous, and likely not useful in any but contrived examples, such as the above-mentioned simulation of `map`.

In this way, `reduce` differs from `map`, in which each element of the output is calculated from one element of the input — a classic case of data-parallelism.

[1] In the actual implementation, we convert these back into `replicate` expressions if they are not fused, but for clarity the bookkeeping necessary is elided in this presentation.

Even if we have a producer-consumer-pair, not all such pairs *can* be fused, and not all are *desirable* to fuse. For instance, `filter-map` fusion is not possible, although `filter-reduce` is. The reason is that, with the former, the size of the `map`-output is the same as the size of its input, yet the size of the output of `filter` cannot be known in advance, which precludes an efficient fused form.

7.1.1 Fusion algebra

In this section, we will give an informal introduction to which SOACs can be fused, as well as the form of the result. In order to preserve clarity, we will not go into great detail until Section 7.3.

`map-map` fusion

The quintessential example of fusion is composing two consecutive `map` operations into a single `map`, as follows:

```
let {x1, x2} = mapT(f, a1)
in  mapT(g, x1, y)
    ⇓
mapT(fn β (α1 a1_i, α2 y_i) =>
      let {x1_i, x2_i} = f(a1_i)
      in  g(x1_i, y_i)
    , a1, y )
```

`replicate` fusion

`replicate` is an interesting case. We wish to always be able to fuse replicate into a consumer, like this:

```
let x = replicate(N,a) in
mapT(f, x) in
    ⇓
mapT(fn β1 (int i) =>
       f(a), b)
```

And indeed, this can be done through ordinary `map`-fusion if `replicate(N,a)` is first rewritten to `map` as described in Section 7.1.

`map-reduce` and `map-scan` fusion

The result of `map-reduce`-fusion is normally `redomap`.

```
let {x1, x2} = mapT(f, a1)
in  reduceT(⊕,e1,e2, x1,y)
    ⇓
redomapT(⊕
, fn (β1,β2) ( β1 e1, β2 e2
             , α1 a1_i,α2 y_i)
   => let {x1_i, x2_i} = f(a1_i)
      in  ⊕(e1,e2,x1_i,y_i)
, (e1, e2), a1, y )
```

```
let {c} = mapT(fn {int} (int x, int y) => {x+y},
                a, b) in
reduceT(op +, {0}, c)
  ⇓
reduceT(fn int (int acc, int x, int y) => acc + x + y,
        {0}, a, b) // Type error, as accumulator
                   // type must match array input type
```

Figure 20: Cannot fuse to `reduce` (but `redomap` would be valid)

In general, we cannot fuse `map` and `reduce` to another `reduce`, as the accumulator type of a reduction must match the element type of the input array. Adding the input of the `map` to the input of the `reduce` may violate this requirement, as demonstrated on Figure 20.

The solution is to first rewrite `reduceT(`$\oplus$`,`x`,`a`)` to `redomapT(`$\oplus$`,`$\oplus$`,`x`,`a`)`, since we can always fuse `map` with `redomap`:

```
let {x1, x2} = mapT(f, a1)
in  redomapT(⊕, g, e, x1, y)
    ⇓
redomapT(⊕
, fn β (β e, α1 a1ᵢ, α2 yᵢ)
   => let {x1ᵢ, x2ᵢ} = f(a1ᵢ)
      in  g(e, x1ᵢ, yᵢ)
, e, a1, y )
```

However, there are a few rare cases where we can fuse `map` and `reduce` to `reduce`. This only happens when the *input* to the map has the same count and types as the outputs of the `map` that are being used as input to the `reduce`.

```
let {x₁, x₂} = mapT(f, a₁, a₂)
in  reduceT(⊕,{e₁, e₂, e₃}, x₁, x₂, y)
    ⇓
reduceT(fn (α₁,α₂) ( α₁ x₁, α₂ x₂, α₃ x₃,
            α₁ ai₁, α₂ ai₂, α₃ ye) =>
          let {x₁, x₂} = f(ae₁, ae₂)
          in  ⊕(x₁, x₂, x₃, x₁, x₂, ye)
        , {e₁, e₂, e₃}, a₁, a₂, y)
```

In fact, under these circumstances we can also fuse `map` with `scan`:

```
let {x₁, x₂} = mapT(f, a₁, a₂)
in  scanT(⊕,{e₁, e₂, e₃}, x₁, x₂, y)
    ⇓
scanT(fn (α₁,α₂) ( α₁ x₁, α₂ x₂, α₃ x₃,
          α₁ ai₁, α₂ ai₂, α₃ ye) =>
        let {x₁, x₂} = f(ae₁, ae₂)
        in  ⊕(x₁, x₂, x₃, x₁, x₂, ye),
      {e₁, e₂, e₃}, a₁, a₂, y)
```

It should be clear that the composed function is still associative, as required by `scan` and `reduce`.

`filter-filter` fusion

Fusing `filter-filter` is quite simple - it's a new `filter` SOAC where both of the filter functions must return true for each element. However, we can only perform the fusion if the input set of the consumer is a subset of the output set of the producer. Put another way, the consumer must accept input from no other source than the producer involved in the fusion.[2]

```
let {x1,x2}=filterT(c1,a1,a2) in
let {y} = filterT(c2, x1) in
...
    ⇓
let {y, _} = filterT(fn bool (α1 ai1,α2 ai2) =>
                       if   c1(ai1, ai2)
                       then c2(ai1)
                       else False,
                     a1, a2 ) in
...
```

As a bit of a technical curiosity, we are forced to use an `if`-expression, as the `&&` operator in $\mathcal{L}_0$ is not short-circuiting.

`filter-reduce` fusion

We can fuse `filter` with `reduce` and obtain `reduce` *only* in the case where the input set of the `reduce` is equal to the output set of the `filter`.

```
let {x} = filterT(c, a) in
reduceT(⊕, e, x)
    ⇓
reduceT(fn β (β e, β ai) =>
          if c(ai) then ⊕(e,ai) else e,
        {e}, a)
```

We can fuse `filter` with `reduce` and obtain `redomapT` if the input set of the `reduce` is included in the output set of the `filter`.

```
let {x1,x2} = filterT(c, a1, a2)
in  reduceT(⊕, {e}, x1)
    ⇓
redomapT(⊕,
         fn β (β e, α1 ai1, α2 ai2) =>
           if c(ai1, ai2)
           then ⊕(e, ai1)
           else e,
         {e}, a1, a2)
```

[2]The implementation in the $\mathcal{L}_0$ compiler is currently even more restrictive, in that the input set of the consumer must match the output set of the producer *exactly*. Fixing this oversight is left as an exercise for the author.

`filter-redomap` fusion

Similarly, we can fuse `filter` with `redomapT` and obtain `redomapT` if the input set of the `reduce` is included in the output set of the `filter`.

```
let {x1,x2}=filterT(c, a1, a2)
in  redomapT(⊕, g, {e}, x1)
    ⇓
redomapT(⊕,
         fn β (β e, α1 ai1, α2 ai2) =>
           if c(ai1, ai2)
           then g(e, ai1)
           else e,
         {e}, a1, a2 )
```

7.1.2 Invalid fusion

We must be careful not to violate the uniqueness rules when performing fusion. For example, consider the following program.

```
let b = map(f, a) in
let c = a with [i] <- x in
map(g, b)
```

Without the constraints imposed upon us by the semantics of in-place modification, we could fuse to the following program.

```
let c = a with [i] <- x in
map(g ∘ f, a)
```

However, this results in a violation of Uniqueness Rule 1, and the resulting program is thus invalid. In general, we must track the possible execution paths from the producer-SOAC to the consumer-SOAC, and only fuse if none of the inputs of the producer have been consumed (in the uniqueness type sense of the word) by a `let-with` or function call on any possible execution paths. This is easier than it may appear at first glance, as the fusion algorithm will only fuse when the consumer is within the lexical scope of the producer anyway.

7.1.3 When to fuse

Even when fusion is possible, it may not be beneficial, and may be harmful to overall performance in the following cases.

Computation may be duplicated.

In the program

```
let x = map(f, a) in
{map(g, x), map(h, x)}
```

fusing the `x`-producer into the two consumers will double the number of calls to the function `f`, which might be expensive. The implementation in the $\mathcal{L}_0$ compiler will currently only fuse if absolutely no computation

is duplicated, although this is likely too conservative. Duplicating cheap work, for example functions that use only primitive operations on scalars, is probably not harmful to overall performance, although we have not investigated this fully. In Section 9.2, we present a transformation that, in some cases, duplicates computation in order to enhance fusibility.

In general, in the context of GPU, the tradeoff between duplicating computation and increasing communication is not an easy problem to solve. Accessing global memory can be more than a hundred times slower than accessing local (register) memory, hence duplicating computation may in some cases be preferable.

Can reduce memory locality.
Consider a simple case of fusing `(map` f`)` $\circ$ `(map` g`)`. When g is executed for an element of the input array, neighboring elements will be put into the cache, making them faster to access. This exhibits good data locality. In contrast, the composed function $f \circ g$ will perform more work after accessing a given input element, increasing the risk that the input array may be evicted from the cache before the next element is to be used. On GPUs, there is the added risk of the kernel function exercising additional register pressure, which may reduce hardware occupancy (thus reducing latency hiding) by having fewer computational cores active. In this case, it may be better to execute each of the two `map`s as separate kernels.

The $\mathcal{L}_0$ compiler does not currently handle this problem, as it is envisioned that a later (and as-of-yet unimplemented) transformation will perform *loop distribution* (sometimes called *loop fission*). This step is necessary in any case, as it can be used to improve the degree of parallelism, compared to the original program. Figure 21 demonstrates a fully fused `map` where the degree of parallelism can be improved by distributing the inner reductions out of the loop. In the original program, the inner map had to wait for the two reductions to finish computing `x` and `y` before executing its inner loop, whereas the distributed program consists of three parallel loop nests.

The fusion algorithm is currently designed to fuse as much as possible, although without duplicating computation.

7.2 Composition

To begin the exposition of the precise mechanics of fusion, we will present the mechanics behind composing the functions involved in a fusion operation. For example, consider the trivial example of `map`-`map`-fusion. In principle, the equation seems simple enough:

$$\texttt{map}\ f \circ \texttt{map}\ g = \texttt{map}\ (f \circ g).$$

However, while the intuition behind the above equation is correct, it is woefully imprecise. Fusion in $\mathcal{L}_0$ is not performed on simple `map`s that take input from

```
map(fn int ([int] r) =>
      let x = reduce(f, 0, r) in
      let y = reduce(g, 0, r) in
      map(h(x,y), r),
    a)
                  ⇓
let xs = map(reduce(f,0),a) in
let ys = map(reduce(g,0),a) in
map(fn [int] ({int,int,[int]} t) =>
      let {r,x,y} = t in
      map(h(x,y), r),
    zip(a,xs,ys))
```

Figure 21: Loop distribution

```
let {x, y, z} = mapT(f, a) in
mapT(fn int (int a, int b, int c) => e, x, y, x)
                  ⇓
      let {x, y, z} = mapT(f, a) in
      mapT(fn int (int a, int b, int d) =>
             let c = a in e,
           x, y, z)
```

Figure 22: Single-input transformation

only one other `map`, but complex `mapT`s that may take input from several sources, where only some may be fusible. Hence, a more detailed elaboration is necessary.

This section will describe various ways of combining the functions used in SOACs, as appropriate for different cases of fusion. In Section 7.3, we will describe the rules used for fusion of full SOAC expressions.

In this section, we will assume that each output of a producer is used exactly once in every relevant `redomap`- and `map`-consumer. This assumption can be provided through trivial rewriting prior to performing function composition, as illustrated on Figure 22. We gain the property that each output of the producer is bound to exactly one parameter of the consumer's function, making it easier to describe the relationship between producer and consumer.[3]

The presentation will be of the form of *judgements*. To skip ahead a bit, we will write the `map`-composition of two functions as

$$\boxed{(l_b, e_{b_1}, \ldots, e_{b_m}) \overset{o_1,\ldots,o_k}{\underset{\texttt{map}}{\circ}} (l_a, e_{a_1}, \ldots, e_{a_n}) \Rightarrow (l_r, e_{r_1}, \ldots, e_{r_l})}.$$

This judgement is said to *hold* if the preconditions specified for the judgement are upheld. The preconditions for a given judgement, if any, will be listed when the judgement is defined below.

[3] In the actual implementation, this transformation is not done. Instead, the composition uses more complicated bookkeeping, but presenting all details would obscure the exposition of the central mechanism.

7.2.1 `map-map` composition

We are given two functions:

$$l_a \equiv \texttt{fn}\ t_{a_r}\ \texttt{(}p_{a_1}\texttt{, ..., }p_{a_n}\texttt{) => }e_a$$

whose inputs are $e_{a_1}\texttt{,...,}e_{a_m}$ and whose outputs are $o_1\texttt{,...,}o_k$; and

$$l_b \equiv \texttt{fn}\ t_{b_r}\ \texttt{(}p_{b_1}\texttt{, ..., }p_{b_m}\texttt{) => }e_b,$$

whose inputs are $e_{b_1}\texttt{,...,}e_{b_m}$.

The goal is to compute a function

$$l_r \equiv \texttt{fn}\ t_{b_r}\ \texttt{(}p_{r_1}\texttt{, ..., }p_{r_l}\texttt{) => }e_r$$

that corresponds to the intuitive notion of the composition $l_b \circ l_a$.

For notational convenience, we define the following sets of parameters of the two functions.

$$\text{params}(l_a) = \{p_{a_1}, \ldots, p_{a_n}\}$$
$$\text{params}(l_b) = \{p_{b_1}, \ldots, p_{b_m}\}$$

If the inputs of l_b are disjoint from the outputs of l_a, then we are done, and $l_r = l_b$. Otherwise, there is a non-empty mapping

$$\mathcal{I}(o_i) = p_{b_j} \quad \text{when } o_i = e_{b_j}$$

of outputs of l_a to the corresponding parameters of l_b. The parameters (and corresponding inputs) to the desired function l_r are the parameters of l_b, except those in $\mathcal{I}$, concatenated with the parameters of l_a:

$$\{e_{r_1}, \ldots, e_{r_l}\} = \text{params}(l_r) = (\text{params}(l_b) \backslash \delta) \oplus \text{params}(l_a)$$

where δ are the parameters p_{b_j} in the range of $\mathcal{I}$.

The body of l_r is then defined as follows:

$$e_r \equiv \texttt{let \{}\mathcal{I}(o_1)\texttt{,...,}\mathcal{I}(o_k)\texttt{\} = }e_a\ \texttt{in}\ e_b$$

We will refer to this entire operation as:

$$(l_b, e_{b_1}, \ldots, e_{b_m}) \overset{o_1,\ldots,o_k}{\underset{\texttt{map}}{\circ}} (l_a, e_{a_1}, \ldots, e_{a_n}) \Rightarrow (l_r, e_{r_1}, \ldots, e_{r_l})$$

7.2.2 `filter-filter` composition

We do not have `fold` per se in $\mathcal{L}_0$, but this method of composition is used for both `reduce` and the `fold`-like semantics of `redomap`, hence the name.

We are given two functions:

$$l_a \equiv \texttt{fn \{bool\} (}p_{a_1}\texttt{, ..., }p_{a_n}\texttt{) => }e_a$$

which take as inputs $e_{a_1}\texttt{,...,}i_{e_n}$, and whose outputs are $o_1\texttt{,...,}o_k$; and

$$l_b \equiv \texttt{fn \{bool\} (}p_{b_1}\texttt{, ..., }p_{b_n}\texttt{) => }e_b,$$

whose inputs are $e_{b_1}, \ldots, e_{b_n}$.

Precondition: Every input e_{b_i} must correspond to some output o_j, and every output o_i must correspond to some input e_{b_i}. That is, the producer set of l_a is equal to the input set of l_b. Or to put it another way, l_b takes input from no other source.

The goal is to compute a function

$$l_r \equiv \texttt{fn \{bool\} (}p_{a_1}\texttt{, ..., }p_{a_n}\texttt{) => }e_r$$

whose inputs are $i_{r_1}, \ldots, i_{r_l}$, that corresponds to the intuitive composition of $l_a \wedge l_b$. Note that the parameters are the same as for l_a, which means that we have to explicitly create a `let`-binding for the names of the parameters of l_b or they will be free in e_b. To this end, define the mapping

$$\mathcal{I}(o_i) = p_{b_j} \quad \text{when } o_i = e_{b_j}.$$

The body of l_r is now definable as

$$\begin{aligned} e_r \equiv\ & \texttt{let \{}ok\texttt{\} = }e_a\texttt{ in }ok\texttt{ \&\&} \\ & \texttt{let \{}\mathcal{I}(o_1)\texttt{,...,}\mathcal{I}(o_k)\texttt{\} = \{}p_{a_i}\texttt{,...,}p_{a_n}\texttt{\} in }e_b \end{aligned}$$

where ok is some fresh variable.

We will refer to this entire operation as:

$$(l_b, e_{b_1}, \ldots, e_{b_n}) \overset{o_1,\ldots,o_k}{\underset{\texttt{filter}}{\circ}} (l_a, e_{a_1}, \ldots, e_{a_n}) \Rightarrow (l_r, e_{a_1}, \ldots, e_{a_n})$$

7.2.3 `filter-fold` composition

We are given two functions:

$$l_a \equiv \texttt{fn \{bool\} (}p_{a_1}\texttt{, ..., }p_{a_n}\texttt{) => }e_a$$

which take as inputs $e_{a_1}, \ldots, i_{e_n}$, and whose outputs are $o_1, \ldots, o_k$; and

$$l_b \equiv \texttt{fn }t_{b_r}\texttt{ (}u_{b_1}\texttt{, ..., }u_{b_m}\texttt{, }p_{b_1}\texttt{, ..., }p_{b_n}\texttt{) => }e_b,$$

whose inputs are $e_{b_1}, \ldots, e_{b_n}$.

Precondition: Every input e_{b_i} corresponds to some output o_j, and every output o_i corresponds to some input e_{b_i}. That is, the producer set of l_a is equal to the input set of l_b. Or to put it another way, l_b takes input from no other source. The u_bs are accumulator parameters that do not correspond to an array input.

The goal is to compute a function

$$l_r \equiv \texttt{fn }t_{b_r}\texttt{ (}p_{r_1}\texttt{, ..., }p_{r_l}\texttt{) => }e_r.$$

Note that the parameters are the same as for l_a, which means that we have to explicitly create a `let`-binding for the names of the parameters of l_b before e_b makes sense. To this end, define the mapping

$$\mathcal{I}(o_i) = p_{b_j} \quad \text{when } o_i = e_{b_j}.$$

The body of l_r is now definable as

$$\begin{aligned} e_r \equiv\ & \texttt{let } \{ok\} \texttt{ = } e_a \texttt{ in if } ok \\ & \texttt{then let } \{\mathcal{I}(o_1),\ldots,\mathcal{I}(o_k)\} \texttt{ = } \{p_{a_i},\ldots,p_{a_n}\} \texttt{ in } e_b \\ & \texttt{else } \{u_{b_1}, \ldots, u_{b_m}\} \end{aligned}$$

where ok is some fresh variable.

We will refer to this entire operation as:

$$(l_b, e_{b_1}, \ldots, e_{b_n}) \overset{o_1,\ldots,o_k}{\underset{\texttt{fold}}{\circ}} (l_a, e_{a_1}, \ldots, e_{a_n}) \Rightarrow (l_r, e_{a_1}, \ldots, e_{a_n})$$

7.3 Fusion rules

With function composition defined, we can define fusion rules for SOACs. We present fusion as a judgement

$$\boxed{\begin{array}{c} producer \overset{os}{\rightsquigarrow} consumer \\ \Rightarrow result \end{array}}.$$

This means that *producer*, which produces outputs *os*, can be fused with *consumer*, yielding *result* as the combined SOAC. Valid judgements of this form are given by the following inference rules, which should mostly be intuitive.

$$\frac{(l_b, \overline{es_b}) \overset{\overline{os}}{\underset{\texttt{map}}{\circ}} (l_a, \overline{es_a}) \Rightarrow (l_r, \overline{es_r})}{\begin{array}{c}\texttt{mapT}(l_a, \overline{es_a}) \overset{os}{\rightsquigarrow} \texttt{mapT}(l_b, \overline{es_b}) \\ \Rightarrow \texttt{mapT}(l_r, \overline{es_r})\end{array}} \qquad \text{(Fuse-Map-Map)}$$

$$\frac{(l_b, \overline{es_b}) \overset{\overline{os}}{\underset{\texttt{map}}{\circ}} (l_a, \overline{es_a}) \Rightarrow (l_r, \overline{es_r})}{\begin{array}{c}\texttt{mapT}(l_a, \overline{es_a}) \overset{os}{\rightsquigarrow} \texttt{scanT}(l_b, \{\overline{us}\}, \overline{es_b}) \\ \Rightarrow \texttt{scanT}(l_r, \{\overline{us}\}, \overline{es_r})\end{array}} \quad (\overline{es_a} = \overline{es_b}) \qquad \text{(Fuse-Map-Scan)}$$

Fusing `map-reduce` and `filter-reduce` is usually done by first rewriting `reduce` to `redomap`, although when the producer-output and consumer-input match exactly, `filter-reduce` can fuse to `reduce`.

$$\frac{\begin{array}{c}\texttt{filterT}(l_a, \overline{es_a}) \overset{\overline{os}}{\rightsquigarrow} \texttt{redomapT}(l_b, l_b, \{\overline{us}\}, \overline{es_b}) \\ \Rightarrow \texttt{redomapT}(l_b, l_r, \{\overline{us}\}, \overline{es_r})\end{array}}{\begin{array}{c}\texttt{filterT}(l_a, \overline{es_a}) \overset{os}{\rightsquigarrow} \texttt{reduceT}(l_b, \{\overline{us}\}, \overline{es_b}) \\ \Rightarrow \texttt{reduceT}(l_r, \{\overline{us}\}, \overline{es_a})\end{array}} \quad (\overline{es_b} \subseteq \overline{os})$$

(Fuse-Filter-Reduce-1)

Note that Fuse-Filter-Reduce-1 has a side condition that implies that the types of $\overline{es_a}$ are equal to the types of $\overline{es_b}$. This permits us to keep the result as a `reduceT` rather than a `redomapT`.

$$\frac{(\texttt{fn } t_{b_r} \ (\overline{ps_b}) \texttt{ => } e_b, \overline{es_b}) \overset{\overline{os}}{\underset{\texttt{map}}{\circ}} (l_a, \overline{es_a}) \Rightarrow (\texttt{fn } t_{b_r} \ (\overline{ps_r}) \texttt{ => } e_r, \overline{es_r})}{\begin{array}{c}\texttt{mapT}(l_a, \overline{es_a}) \overset{os}{\rightsquigarrow} \texttt{redomapT}(\oplus, \texttt{fn } t_b \ (\overline{us_b}, \ \overline{ps_b}) \texttt{ => } e_b, \{\overline{vs}\}, \overline{es_b}) \\ \Rightarrow \texttt{redomapT}(\oplus, \texttt{fn } t_b \ (\overline{us_b}, \ \overline{ps_r}) \texttt{ => } e_r, \{\overline{vs}\}, \overline{es_r})\end{array}}$$

(Fuse-Map-Redomap)

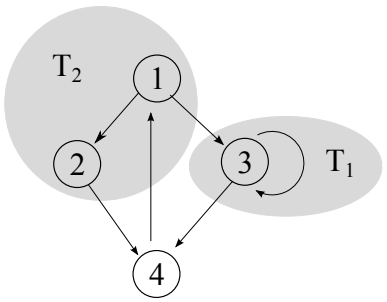


Figure 23: T_1-T_2-reduction

$$\frac{(\texttt{fn}\ t_{b_r}\ (\overline{ps_b})\ \texttt{=>}\ e_b, \overline{es_b}) \overset{\overline{os}}{\underset{\texttt{fold}}{\circ}} (l_a, \overline{es_a}) \Rightarrow (\texttt{fn}\ t_{b_r}\ (\overline{ps_r})\ \texttt{=>}\ e_r, \overline{es_a})}{\begin{array}{c}\texttt{filterT}(l_a, \overline{es_a}) \overset{os}{\rightsquigarrow} \texttt{redomapT}(\oplus, \texttt{fn}\ t_b\ (\overline{us_b},\ \overline{ps_b})\ \texttt{=>}\ e_b, \{\overline{vs}\}, \overline{es_b}) \\ \Rightarrow \texttt{redomapT}(\oplus, \texttt{fn}\ t_b\ (\overline{us_b},\ \overline{ps_r})\ \texttt{=>}\ e_r, \{\overline{vs}\}, \overline{es_a})\end{array}}$$

(Fuse-Filter-Redomap)

$$\frac{\begin{array}{c}\texttt{filterT}(l_a, \overline{es_a}) \overset{\{os\}}{\rightsquigarrow} \texttt{redomapT}(l_b, l_b, \{\overline{us}\}, \overline{es_b}) \\ \Rightarrow \texttt{redomapT}(\oplus, l_r, \{\overline{us}\}, \overline{es_r})\end{array}}{\begin{array}{c}\texttt{filterT}(l_a, \overline{es_a}) \overset{os}{\rightsquigarrow} \texttt{reduceT}(l_b, \{\overline{us}\}, \overline{es_b}) \\ \Rightarrow \texttt{redomapT}(\oplus, l_r, \{\overline{us}\}, \overline{es_r})\end{array}}$$

(Fuse-Filter-Reduce-2)

7.4 The fusion algorithm

The entire algorithm consists of two distinct stages:

1. Traverse the program, collecting SOAC-expressions and fusing producers into consumers where possible. The end result is a mapping from SOACs in the original program, to replacement SOAC expressions (the result of fusion operations). This is called the *gathering* phase.

2. Traverse the program again, using the result of the gathering phase to replace SOAC expressions with their fully fused forms. This may lead to dead code, as the output variables of producers that have been fused into their consumers are no longer used. These can be removed using standard dead code removal.

The replacement stage is trivial, hence the rest of this section will be concerned solely with the gathering stage.

$\mathcal{L}_0$, as a block-structured language, is suited to region-based analysis, and the fusion algorithm is indeed designed as a reduction of a dataflow graph.

Our structural analysis is inspired by the T_1-T_2-reduction [1]. We say that a flow graph is reducible if it can be reduced to a single node by the following two transformations:

T_1: Remove an edge from a node to itself.

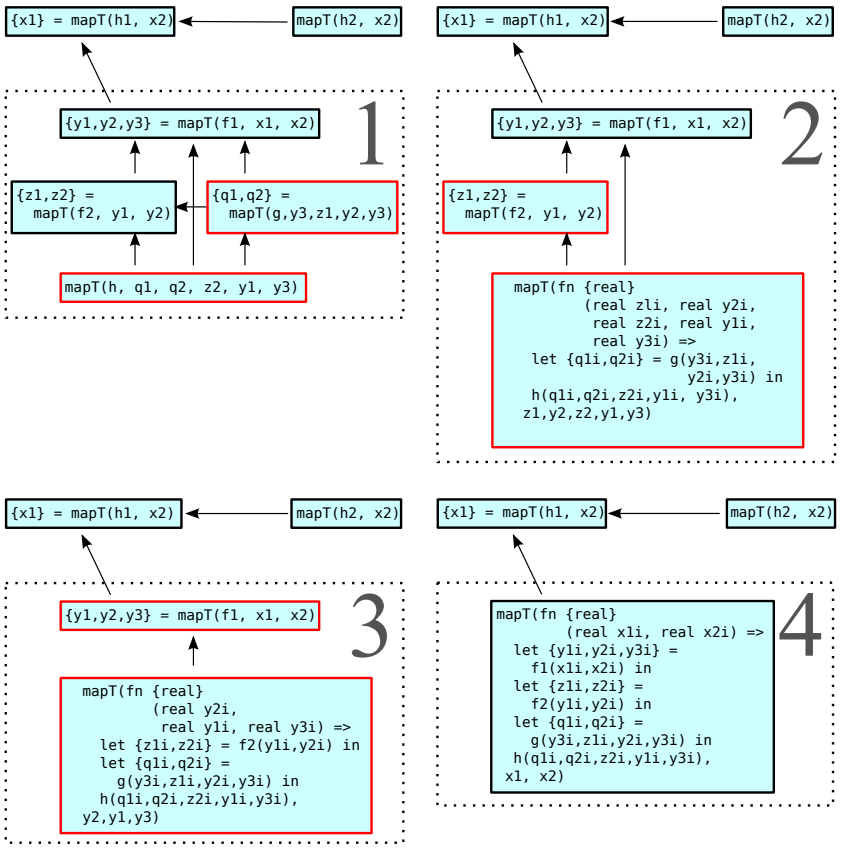


Figure 24: Fusion by T_2 transformation on the dependency graph

T_2: Combine two nodes m and n, where m is the single predecessor of n, and n is not the entry of the flow graph.

On Figure 23 is shown a small flow graph and highlights instances where the two reductions could apply. The overall idea is to construct a flow graph of the $\mathcal{L}_0$ program, reduce it to a single point, and at each reduction step combine the information stored at the nodes being combined.

$\mathcal{L}_0$ always produces a reducible graph. Each node corresponds to an expression, with the successors of the node being its subexpressions. This means that we can implement the reduction simply as a bottom-up traversal of the $\mathcal{L}_0$ syntax tree.

Figure 24 depicts the intuitive idea on which our fusion transformation is based. The top-left figure shows the dependency graph of a simple program, where an arrow points from the consumer to the producer.

The main point is that all SOACs that appear inside the box dashed box can be fused without duplicating any computation, even if several of the to-be-fused arrays are used in different SOACs. For example, `y1` is used to compute both `{z1,z2}` and `{q1,q2}`[4]. This is accomplished by means of T_2 reduction on the

[4]Note also that not all input arrays of a SOAC need be produced by the same SOAC.

dependency graph:

The rightmost child, i.e., `mapT(g,..)`, of the root SOAC (`mapT(f1,...)`) has only one incoming edge, hence it can be fused. This is achieved by:

1. Replacing in the root SOAC the child's output with the child's input arrays
2. Inserting a call to the child's function in the root's function, which computes the per-element output of the child,
3. Removing duplicate input arrays of the resulting SOAC.

This operation is exactly what the fusion rules in Section 7.3 formalise.

The top-right part of Figure 24 shows the (optimised) result of the first fusion, where the copy statements have been eliminated by copy propagation. In the new graph, the leftmost child of the root, i.e., the one computing `{z1,z2}`, has only one incoming edge and can be fused. The resulting graph, shown in the bottom-left figure can be fused again resulting in the bottom-right graph of Figure 24. At this point no further T_2 reduction is possible, because the SOAC computing `x1` has two incoming edges. This example demonstrates a key benefit of removing `zip`/`unzip` and using the tupleless SOACs representation: There are no intermediate nodes in the data-dependency graph between fusable producer and consumer.

7.4.1 Dataflow rules

During reduction, we will track the following pieces of information.

SOACs : $Exp \to (Label \times Pat \times Exp)Set$**.** The set of SOACs that appears in an expression, modelled as a mapping from a (unique) label to a pair of a SOAC expression and its output pattern. We shall say SOACs(e) to refer to this mapping, and SOACs(e)[l] to refer to the SOAC with label l. For example,

$$\text{SOACs}(\texttt{let \{}a\texttt{,}b\texttt{,}c\texttt{\} = mapT(}f\texttt{,}x\texttt{,}y\texttt{,}z\texttt{) in \{}a\texttt{,}b\texttt{,}c\texttt{\}}) = \{(\ell, \texttt{\{}a\texttt{,}b\texttt{,}c\texttt{\}}, \texttt{mapT(}f\texttt{,}x\texttt{,}y\texttt{,}z\texttt{)})\},$$

where ℓ is a fresh label. After the SOACs set has been computed, we can use SOACs(e_b), where e_b is the body of a function to refer to the set of all SOACS in that function. Since the fusion transformation is strictly intraprocedural, this is sufficient for our needs.

This mapping may not necessarily contain all SOACs that appear *syntactically* in the program. A core idea behind the fusion algorithm is that whenever we would add a SOAC to this mapping, we instead check whether it can be fused with the SOACs already present.

unfusible : $Exp \to NameSet$**.** The *infusible set*, a set of variable names, is key to preventing unwanted fusion, as it indicates which SOACs should never be fused. The infusible set prevents both undesired and invalid fusion, as outlined in sections Sections 7.1.2 and 7.1.3 respectively. Given an

$\mathcal{L}_0$ expression e, we shall say UNFUSIBLE(e) to refer to the infusible set produced by e. For example:

$$\text{UNFUSIBLE}(\texttt{let x = mapT(f,a) in}\\ \texttt{let y = mapT(g,a) in \{x,y\}}) = \{a\},$$

because a is used twice, and hence fusing its producer into `f` and `g` would cause work duplication. To simplify the example, f and g are ignored when computing the infusible set, although as we shall see below, this is not the case in practice.

arrInputs : $Exp \to Name \to Labels$**.** A mapping from arrays to a set of the SOACs that use the array as input. This is modelled as a set of pairs, each pair consisting of an array name and a SOAC name. We shall refer to the mapping generated by a given expression e as ARRINPUTS(e). For example,

$$\text{ARRINPUTS}(\texttt{mapT(}f\texttt{, }x\texttt{, }y\texttt{, }z\texttt{)}) = \{(x, \{\ell\}), (y, \{\ell\}), (z, \{\ell\})\},$$

where ℓ is the label of the `mapT`-SOAC.

We define an associative and commutative operation $\sqcup$ to combine multiset mappings by taking the union of values (in the case of ARRINPUTS, sets of labels, s) of corresponding keys (for ARRINPUTS, variable names, v), as follows.

$$\{(v_1, s_1), \ldots, (v_n, s_n)\} \sqcup \{(v_{n+1}, s_{n+1}), \ldots, (v_{n+m}, s_{n+m})\} =\\ \{(v_i, \bigcup_{(v_i, s_j), 0 \leq j \leq n+m} s_j)\},$$

Intuitively, $x \sqcup y$ is a mapping that contains the union of the keys in x and y, with the value for a key v being the union of the values for v in x and y (or just an untouched value, if v was only present in one of the mappings).

Similarly, we use $\sqcap$ to denote a similar mapping, except taking the intersection of values.

$$\{(v_1, s_1), \ldots, (v_n, s_n)\} \sqcap \{(v_{n+1}, s_{n+1}), \ldots, (v_{n+m}, s_{n+m})\} =\\ \{(v_i, \bigcap_{(v_i, s_j), 0 \leq j \leq n+m} s_j)\},$$

consumed : $Exp \to Label \to NameSet$**.** A mapping from the labels of SOACs in an expression, to a set of the names that are consumed on the path to that SOAC. The purpose of this mapping is to ensure that we do not fuse in violation of the uniqueness rules. For example, if

$$\mathit{consumed}(e)[\ell] = \{\texttt{a}\}$$

then we cannot fuse any producer taking `a` as input with the SOAC labelled ℓ, as `a` is consumed on the execution path to ℓ.

If more specific rules are not given, the data flows default to the following.

$$\textsc{unfusible}(e) = \bigcup_{e' \in \textsc{childExps}(e)} \textsc{unfusible}(e')$$

$$\textsc{arrInputs}(e) = \bigsqcup_{e' \in \textsc{childExps}(e)} e'$$

$$\textsc{SOACs}(e) = \bigcup_{e' \in \textsc{childExps}(e)} \textsc{SOACs}(e')$$

$$\textsc{consumed}(e) = \bigsqcup_{e' \in \textsc{childExps}(e)} \textsc{consumed}(e')$$

Where $\textsc{childExps}(e)$ are the *immediate* children of e, e.g.

$$\textsc{childExps}(\texttt{if p(x) then t(y) else f(z)}) = \{\texttt{p(x)}, \texttt{t(y)}, \texttt{f(z)}\}.$$

Now for specific rules, based on the shape of the given expression.

Case $e \equiv v$ (v is a variable)
This rule is only applied when v is not an array input to a SOAC. This implies that the producer of v cannot be fused, as its output v is used here.

$$\textsc{unfusible}(e) = \{v\}$$

Case $e \equiv v\texttt{[}e_1\texttt{, ..., }e_n\texttt{]}$
If an element is retrieved from an array through indexing, we have no choice but to manifest that array, thus forcing us to avoid fusion due to our principle of avoiding duplication of computation. In many cases, for example if the array v is the result of a `map` operation, it might be beneficial to replace the index operation by an inlined copy of the `map` function, and let the original `map` be fused. Duplicating computation of a single element is likely acceptable, but not done in the general case by the current implementation, and it is hard to determine the optimal choice as long as $\mathcal{L}_0$ does not yet have a well-defined cost model. Nevertheless, Section 9.2 will describe how we inline particularly simple cases.

$$\textsc{unfusible}(e) = \{v\} \cup \bigcup_{1 \leq i \leq n} \textsc{unfusible}(e_i)$$

Case $e \equiv$ `if` e_c `then` e_t `else` e_f
The UNFUSIBLE of a conditional consists of whatever is in the UNFUSIBLE

```
let b = map(f, a) in
if p(x) then map(g,b)
        else map(h,b)
```

Figure 25: Fusion into branches acceptable

```
let b = map(f, a) in
if p(x) then concat(map(g,b),map(v,b))
        else map(h,b)
```

Figure 26: Duplicating computation in one branch

```
let b = map(f, a) in
if p(map(v,b)) then map(g,b)
               else map(h,b)
```

Figure 27: Duplicating computation in conditional

sets of its branches, plus any SOAC outputs that may be used multiple times. Note that an output can be used in both the true and the false branch, and it will still only have been considered to be used once, becase only one of the true and false branch will be executed, never both.

$$\begin{aligned}&\textsc{arrInputs}(e) = \\ &\quad \{(v, s') \mid (v, s) \in \textsc{arrInputs}(e_t) \cup \textsc{arrInputs}(e_f)\} \\ &\qquad \text{Where } s' \text{ is the set of all SOAC consumers of } v \text{ in both } e_t \text{ and } e_f.\end{aligned}$$

$$\begin{aligned}&\textsc{unfusible}(e) = \\ &\quad \textsc{unfusible}(e_c) \cup \textsc{unfusible}(e_t) \cup \textsc{unfusible}(e_f) \\ &\quad \cup (\textsc{arrInputs}(e_c) \sqcap \textsc{arrInputs}(e_t)) \\ &\quad \cup (\textsc{arrInputs}(e_c) \sqcap \textsc{arrInputs}(e_f))\end{aligned}$$

The reason for these rules can be illustrated by the following examples. In Figure 25, it is clear that fusing computation of `b` with both `map(g,b)` and `map(h,b)` will not cause duplicated computation, as the two consumers are on separate control-flow paths. On the other hand, if even one branch contains multiple uses, as in Figure 26, we should not fuse. Additionally, if both the conditional expression and a branch consumes the same array, as on Figure 27, then we should also not fuse.

Case $e \equiv$ `loop (`p `=` e_1`) = for` v `<` e_2 `do` e_3 `in` e_4

For loops, we add any arrays used as SOAC inputs in the loop body to the infusible set. This is because fusing into the loop would duplicate computation by re-evaluating the function in the consumer for every

```
let b = map(f, a) in
loop (v) = for i < n do
             let c = map(g, b) in
             h(v,c) in
...
```

Figure 28: Fusing the producer into the consumer in the loop body would duplicate computation

iteration of the loop – see Figure 28 for an example of this. This is similar to how we ban fusing into SOAC-functions.

Note that any use of an array for another purpose than as SOAC input results in that array name being present in $\textsc{unfusible}(e_3)$ already, thus banning fusion.

$$\begin{aligned}&\textsc{unfusible}(e) = \\ &\quad \textsc{unfusible}(e_1) \cup \textsc{unfusible}(e_2) \cup \textsc{unfusible}(e_3) \cup \textsc{unfusible}(e_4) \\ &\quad \cup \{v \mid (v, s) \in \textsc{arrInputs}(e_3)\}\end{aligned}$$

Case $e \equiv \texttt{let } \{\overline{vs}\} \texttt{ = } soac \texttt{ in } e_b$

The big question here is whether *soac* can be fused as producer with something in $\textsc{SOACs}(e_b)$. In the following, ℓ_p is a fresh name. Let

$$\zeta = \bigcup_{v \in vs} \textsc{arrInputs}(e_b, v)$$

be the set of the labels of all SOACs that contain at least one of our outputs in their input set.

For all $\ell_i^c \in \zeta$, we find the corresponding triple $(\ell_i^c, vs_i^c, soac_i^c)$ in $\textsc{SOACs}(e_b)$. We can check whether fusion is possible by determining whether the following judgement is derivable.

$$\begin{aligned}soac &\overset{vs}{\rightsquigarrow} soac_i^c \\ &\Rightarrow soac_i^r\end{aligned}$$

In total, the following four conditions must all be upheld before we can fuse:

1. There must be at least one consumer $soac_i^c$ with which we can fuse. Note that even if there are more than one, we do not end up duplicating computation, as they would belong to different branches.
2. For each consumer $soac_i^c$, we must be able to fuse and and get some $soac_i^r$. If we could only fuse with some, we would be unable to remove the producer from the resulting program, thus duplicating computation.

3. None of $\overline{vs}$ are in the infusible set. That is,

$$\overline{vs} \cap \textsc{unfusible}(e_b) =$$

   This rule also helps avoiding duplicate computation.

4. Fusion must not bring an array past a point where it is consumed. Formally, we must have that,

$$\textsc{consumed}(e_b)[\ell_i^c] \cap (\text{The array inputs of } soac) = \emptyset$$

   for all ℓ_i^c. Violating this rule would create an invalid program.

If the four conditions are true: In this case, we are fusing *soac* with several SOACs $soac_i^c$, each with a corresponding label ℓ_i^c, and fused as $soac_i^r$. In the following, let e_f be the body of the function in *soac*.

$$\begin{aligned}
&\text{SOACs}(e) = \\
&\quad (\text{SOACs}(e_b) \backslash \zeta) \\
&\quad \cup\, \text{SOACs}(e_f) \\
&\quad \cup\, \{(\ell_i^c, (vs, soac_i^r)) \mid \text{for each } soac_i^c\}
\end{aligned}$$

$$\begin{aligned}
&\textsc{arrInputs}(e) = \\
&\quad (\textsc{arrInputs}(e_b) \text{ with all mappings to each } \ell_i^c \text{ removed}) \\
&\quad \sqcup\, \{(v, \ell_i^c) \mid \text{for all array inputs } v \text{ in each } soac_i^c\}
\end{aligned}$$

$$\begin{aligned}
&\textsc{unfusible}(e) = \\
&\quad \textsc{unfusible}(e_b) \\
&\quad \cup\, \{v \mid (v, s) \in \textsc{arrInputs}(e_f)\}
\end{aligned}$$

If they are not: We cannot fuse with *soac* as a producer, and we must add it as a kernel by itself. It may be fused as the consumer at some later stage of the algorithm, however. To the UNFUSIBLE set, we first add every array variable used as a SOAC input in the body of *soac*. We also also insert every array variable used both as input to *soac*, but also to some SOAC in e_b. The rationale is that fusing whichever SOAC (if any) outputs this variable would duplicate computation, as we have at least two consumers.

$$
\begin{aligned}
&\textsc{unfusible}(e) = \\
&\quad \textsc{unfusible}(e_b) \\
&\quad \cup \{v \mid (v, s) \in \textsc{arrInputs}(e_f)\} \\
&\quad \cup \{v \mid v \text{ is used as input to } soac \text{ but is also in } \textsc{arrInputs}(e_b)\}
\end{aligned}
$$

$$
\begin{aligned}
&\textsc{arrInputs}(e) = \\
&\quad \textsc{arrInputs}(e_b) \\
&\quad \sqcup\, \textsc{arrInputs}(e_f) \\
&\quad \sqcup\, \{(e_1, \{\ell_p\}), \ldots, (e_n, \{\ell_p\})\}
\end{aligned}
$$

$$
\textsc{SOACs}(e) = \textsc{SOACs}(e_b) \cup \{(\ell_p, (vs, soac))\}
$$

Case $e \equiv$ `let` v_1 `=` v_2 `with [`e_1`,...,`e_n`] <-` e_v `in` e_b

For any soac ℓ in the body e_b, we note that the aliases of v_2 are consumed on the path to ℓ.

$$
\begin{aligned}
&\textsc{consumed}(e) = \\
&\quad \{(\ell, \text{aliases}(v_2) \cup \textsc{consumed}(e_b)[\ell]) \mid (\ell, p_\ell, e_\ell) \in \textsc{consumed}(e_b)\}
\end{aligned}
$$

Chapter 8

Fusion-enabling SOAC Transformations

The fusion rules in Section 7.3 cover only simple cases, where the output of the producer is used directly by the consumer, without any intermediary steps. This means that the following program, where the output of the producer is first passed through `transpose`, cannot be fused.

```
let {b} = mapT(f, a) in
mapT(fn [int] ([int] r) => mapT(g, r), transpose(b))
```

However, it is actually possible to fuse this case by first moving the transposition to after the consumer instead:

```
let {b} = mapT(f, a) in
transpose(mapT(fn [int] ([int] r) => mapT(g, r), b))
```

After this transformation, the simple `map`-`map` fusion rule applies. When moving around transformations such as `transpose` (and, later, `reshape`), remember that we think of them as having a delayed representation, and hence moving them will not influence the cost model of the program.

In many cases, such rewriting of a producer-consumer pair is necessary before the simple fusion rules can apply. Indeed, one might consider the Fuse-Map-Redomap rule a particularly simple example of such a rewriting. Fortunately, these rewriting schemes can be incorporated into the existing fusion framework simply by considering them as additional inference rules. The rewriting above can be defined as follows (using the nested map notation from Figure 29):

$$\frac{\texttt{mapT}(l_a, e_a) \overset{e_b}{\rightsquigarrow} \texttt{mapT}^2(l_b, e_b) \Rightarrow soac}{\texttt{mapT}(l_a, e_a) \overset{e_b}{\rightsquigarrow} \texttt{mapT}^2(l_b, \texttt{transpose}(e_b)) \Rightarrow \texttt{transpose}(soac)} \text{ (Fuse-Map-Transpose-Map-Single)}$$

Of course, this rule is far too specific - it covers only `map`-producers and -consumers, and with a single output and input respectively. In the next section, we will see a more general treatment of when we can fuse across arrays.

Furthermore, we will need to extend the SOAC notation we use for expressing fusion judgements. The Fuse-Map-Transpose-Map-Single rule uses

```
mapT^1(f, a_1, ... , a_k) ≡
mapT(f, a_1, ..., a_k)

mapT^{n+1}(f, a_1, ..., a_k) ≡
mapT(fn {[β_1], ..., [β_t]} ([α_1] x_1, ..., [α_k] x_k)) =>
       mapT^n(f, x_1, ..., x_k)
```

Figure 29: Nested map notation

```
let {b} = mapT(f, a) in
let {c} = transpose(b) in
mapT(g, c)
```

(a) Before inlining

```
let {b} = mapT(f, a) in
mapT(g, transpose(b))
```

(b) After inlining

Figure 30: Inlining transposition

`transpose` in places where we have previously considered only plain variables and SOAC expressions. Their meaning is as follows:

- Whenever we use `transpose` (or, later, `reshape`) around a SOAC, the intent is that we transpose every output of the SOAC. Thus, even though `transpose(mapT(...))` is technically not type-correct $\mathcal{L}_0$, since `mapT` returns a tuple, the intended meaning is that every output is transposed.
- When we enclose inputs to a SOAC in `transpose` or `reshape`, as in for example `mapT(`f`,transpose(`$\overline{es}$`))`, the intended meaning is to apply `transpose` to every input, i.e. `mapT(`f`, transpose(`e_1`), ..., transpose(`e_n`))`.

Conceptually, we integrate `transpose` and `reshape` into the consumers by inlining them in the input positions prior to fusion. This is illustrated on Figure 30

We add the following two ancillary fusion rules for fusing with a consumer whose output is transposed or reshaped.

$$\frac{soac_p \overset{os}{\rightsquigarrow} soac_c \Rightarrow soac_r}{soac_p \overset{os}{\rightsquigarrow} \texttt{transpose}(soac_c) \Rightarrow \texttt{transpose}(soac_r)} \quad \text{(Fuse-Transposed-Consumer)}$$

$$\frac{soac_p \overset{os}{\rightsquigarrow} soac_c \Rightarrow soac_r}{soac_p \overset{os}{\rightsquigarrow} \texttt{reshape}(shape,\ soac_c) \Rightarrow \texttt{reshape}(shape,\ soac_r)} \quad \text{(Fuse-Reshaped-Consumer)}$$

The intuition behind these rules is that fusion is not sensitive to what happens with the output of the consumer.

We will need a judgement to determine which SOAC inputs are simply transformations of an origin array. The judgement

$$\boxed{\Re(e) = v}$$

```
let {b} = map(f, a) in
map²(g, transpose(b))
```

(a) Unfusible

```
let {b} = map(f, a) in
transpose(map²(g, b))
```

(b) Fusible

Figure 31: Pushing transposition past consumer

```
let b = map²(f, a) in
map(g, transpose(b))
```

(a) Unfusible

```
let b = map²(f, transpose(a)) in
map(g, b)
```

(b) Fusible

Figure 32: Pulling transposition before producer

means that e is an application of `transpose` or `reshape` (possibly both, or several in sequence) of the array-typed variable v. For example e, may be `transpose`(v). Valid judgements are defined by the following inference rules.

$$\frac{\Re(e) = v}{\Re(\texttt{transpose}(k, n, e)) = v}$$

$$\frac{\Re(e) = v}{\Re(\texttt{reshape}(\mathit{shape}, e)) = v}$$

$$\frac{\text{If } v \text{ is a variable}}{\Re(v) = v}$$

When $\Re(e) = v$, we will say that the *source array* of e is v.

8.1 Fusing across `transpose`

In the general case, `transpose` acts like a fusion blocker - if the output of a producer is transformed before being fed to the consumer, then most likely fusion cannot take place. In some instances, we can perform a local transformation, either *pushing* the transposition past the consumer, or *pulling* it to before the producer. An example is seen on Figure 31 - on Figure 31a, the transposition blocks fusion, but by pushing the transpose operation to the return value of the consumer, as on Figure 31b, we expose `map`-`map`-fusibility. Figure 32 demonstrates same concept, but by pulling instead of pushing the transposition.

To get an intution for the validity of these transformations, we employ the concept of *transpose depth*. A standard textbook transposition interchanges the outer two dimensions - hence we say that it has a depth of 2, as those are the dimensions that are affected. If a SOAC does not directly access the two outermost dimensions, as for example a `map`^{2} does not, we can interchange them without modifying the values that are seen by the body of the SOAC. This can also be generalised to support the generalised (k, n)-transposition described in Section 2.3.

$$\mathcal{D}(k,n) = \begin{cases} k+n & n \geq 0 \\ k & n < 0 \end{cases}$$

Figure 33: Transposition depth

$$\texttt{transpose}^{-1}\texttt{(k,n,e)} \equiv \texttt{transpose(k+n,-n,e)}$$

Figure 34: Inverse transpose

Specifically, $\texttt{mapT}^n$ accesses the element at index $[i_1, \ldots, i_n]$, but we are free to transpose the indices $i_1, \ldots, i_{n-1}$. Of course, the order of the results will be different, which is why we need to perform an inverse transposition on the result. To this end, Figure 33 defines a function for determining the transpose depth of a (k,n)-transpose, and Figure 34 defines a notation for the inverse of a transposition.

Note the crucial property $\texttt{transpose}^{-1}\texttt{(}k\texttt{, }n\texttt{, transpose(}k\texttt{, }n\texttt{, }e\texttt{)) = }e$.

Our presentation will assume that transpositions are inlined as part of the inputs to the consumer, as on Figure 31.

8.1.1 Pushing `transpose`

To get an intution for how the transformation works, let us look at a slightly more complicated example:

```
let {b} = mapT(f, a) in
map^D(k,n)(g, transpose(k, n, b), c)
```

The consumer takes two inputs, and only one of them is from the producer. Our goal is to remove the `transpose` from b - we do not care about whether the algorithm will eventually try to fuse c as well. Observe that by applying an inverse transposition to both inputs, we would remove the transposition surrounding `b`, thus obtaining inputs `b` and $\texttt{transpose}^{-1}\texttt{(k, n, c)}$, and thereby exposing `map`-`map`-fusibility between the producer and consumer. Of course, we still have to transpose the output of the new consumer. This solution also works *only* when all inputs coming from the producer are transposed in the *exact same* way, as otherwise applying the inverse transposition would not cause the transpositions to go away.

To define fusion rules that fit into the framework established in Chapter 7, we will need a bit of machinery. First, we define a judgement for determining whether the outputs of a producer are transposed by the consumer:

$$\boxed{\text{transposed}(os, \overline{es}) \Rightarrow (k,n)}$$

Here, os must be the outputs of a producer, and $\overline{es}$ are the inputs of the consumer. The judgement produces (k,n) if one of the outputs are transposed in $\overline{es}$, defined by the following inference rules:

$$\frac{\Re(e) = v \quad (v \in os)}{\text{transposed}(os, \texttt{transpose(}k\texttt{,}n\texttt{,}e\texttt{)}, \overline{es}) \Rightarrow (k, n)}$$

$$\frac{\Re(e) = v \quad \text{transposed}(os, \overline{es}) \Rightarrow (k, n) \quad (v \notin os)}{\text{transposed}(os, e, \overline{es}) \Rightarrow (k, n)}$$

We also need a judgement for checking whether all inputs coming from the producer are transposed the exact same way, and if so, produce a modified input list with inverse transpositions applied:

$$\boxed{\text{transpose}^{-1}(k, n, os, \overline{es}) \Rightarrow \overline{es'}}$$

This judgement states that all inputs in $\overline{es}$ whose underlying array is in os is (k, n)-transposed. Furthermore, the result of applying an inverse (k, n)-transposition to every input in es is $\overline{es'}$. The judgement is defined by the following inference rules:

$$\frac{}{\text{transpose}^{-1}(k, n, os, \texttt{transpose(}k\texttt{,}n\texttt{,}e\texttt{)}) \Rightarrow e}$$

$$\frac{(e \text{ is not a transposition})}{\text{transpose}^{-1}(k, n, os, e) \Rightarrow e}$$

$$\frac{\Re(e) = v \quad \text{transpose}^{-1}(k, n, os, \overline{es}) \Rightarrow \overline{es'} \quad (v \in os)}{\text{transpose}^{-1}(k, n, os, \texttt{transpose}(k, n, e), \overline{es}) \Rightarrow e, \overline{es'}}$$

$$\frac{\Re(e) = v \quad \text{transpose}^{-1}(k, n, os, \overline{es}) \Rightarrow \overline{es'} \quad (v \in os)}{\text{transpose}^{-1}(k, n, os, \texttt{transpose}(k, n, e), \overline{es}) \Rightarrow e, \overline{es'}}$$

$$\frac{\Re(e) = v \quad \text{transpose}^{-1}(k, n, os, \overline{es}) \Rightarrow \overline{es'} \quad (v \notin os)}{\text{transpose}^{-1}(k, n, os, e, \overline{es}) \Rightarrow \texttt{transpose}^{-1}\texttt{(}k\texttt{, }n\texttt{, }e\texttt{)}, \overline{es'}}$$

We can now define a fusion rule for pushing `transpose` past `mapT` operations of sufficient nesting. The general outline is as follows: First, determine whether an output of the producer is (k, n)-transposed by the consumer. Then, check whether *all* of the producer outputs used by the consumer are (k, n)-transposed, and if so, apply the inverse transposition to every input, obtaining $\overline{es'}$. Finally, attempt to fuse the result.

$$\frac{\begin{array}{c}\text{transposed}(os, \overline{es}) \Rightarrow (k, n) \\ \text{transpose}^{-1}(k, n, os, es) \Rightarrow es' \\ soac_p \overset{os}{\rightsquigarrow} \texttt{mapT}^{\mathcal{D}(k,n)}\texttt{(}f\texttt{,}\overline{es'}\texttt{)} \\ \Rightarrow soac_r\end{array}}{\begin{array}{c}soac_p \overset{os}{\rightsquigarrow} \texttt{mapT}^{\mathcal{D}(k,n)}\texttt{(}f\texttt{,}\overline{es}\texttt{)} \\ \Rightarrow \texttt{transpose(}k\texttt{, }n\texttt{, }soac_r\texttt{)}\end{array}} \quad \text{(PUSH-TRANSPOSE)}$$

8.1.2 Pulling `transpose`

Consider the following program:

```
let {b1, b2} = mapT^D(k,n)(f, a1, a2) in
reduceT(g, transpose(k, n, b1), transpose(k, n, b2), c)
```

We can only push transpositions past sufficiently deeply nested `mapT`s, and in this case the consumer is a `reduceT`. However, since all of the inputs derived from outputs of the consumer are (k, n)-transposed, we can instead transform the producer itself by (k, n)-transposing its inputs. This produces the following program, where `mapT`-`reduceT` fusion is possible:

```
let {b1, b2} = mapT^D(k,n)(f, transpose(k, n, a1), transpose(k, n, a2)) in
reduceT(g, b1, b2, c)
```

Again, this relies on the fact that the body of the producer is invariant with respect to the outer $\mathcal{D}(k, n)$ dimensions.

Again, we will need an ancillary judgement.

$$\boxed{\text{untranspose}(k, n, os, \overline{es}) \Rightarrow \overline{es'}}$$

This judgement checks that all inputs in $\overline{es}$ that use an input from os are (k, n)-transposed, and produces a new sequence of inputs $\overline{es'}$ where such transposes are removed. It is defined by the following inference rules.

$$\frac{}{\text{untranspose}(k, n, os, \texttt{transpose}(k, n, e)) \Rightarrow e}$$

$$\frac{(e \text{ is not a transposition})}{\text{untranspose}(k, n, os, e) \Rightarrow e}$$

$$\frac{\Re(e) = v \qquad \text{untranspose}(k, n, os, \overline{es}) \Rightarrow \overline{es'} \qquad (v \in os)}{\text{untranspose}(k, n, os, \texttt{transpose}(k, n, e), \overline{es}) \Rightarrow e, \overline{es'}}$$

$$\frac{\Re(e) = v \qquad \text{untranspose}(k, n, os, \overline{es}) \Rightarrow \overline{es'} \qquad (v \notin os)}{\text{untranspose}(k, n, os, e, \overline{es}) \Rightarrow e, \overline{es'}}$$

We can now define a fusion rule for pulling transpositions past `mapT` producers of sufficient nesting. The general outline is as follows: First, determine whether an output of the producer is (k, n)-transposed by the consumer. Then, check whether *all* of the producer outputs used by the consumer are (k, n)-transposed, and if so, strip those transpositions and (k, n)-transpose the inputs to the producer instead. Finally, attempt to fuse the result.

$$\frac{\text{transposed}(os, \overline{es_c}) \Rightarrow (k, n) \qquad \text{untranspose}(k, n, os, \overline{es_c}) \Rightarrow \overline{es'_c} \qquad \texttt{mapT}^{\mathcal{D}(k,n)}(f, \texttt{transpose}(k, n, \overline{es_p})) \overset{os}{\rightsquigarrow} \texttt{mapT}(g,\ \overline{es'_c}) \Rightarrow soac_r}{\texttt{mapT}^{\mathcal{D}(k,n)}(f, \overline{es_p}) \overset{os}{\rightsquigarrow} \texttt{mapT}(g,\ \overline{es_c}) \Rightarrow soac_r}$$

(Pull-Transpose)

8.2 Fusing across `reshape`

The idea behind fusing across `reshape` operations is similar to the one covering `transpose`, although less well-developed. We support solely *pulling* reshape prior to `mapT` producers taking single-dimensional arrays as input. For example, we can fuse the following program:

```
let {b} = mapT(f, a) in // a is one-dimensional
reduceT(g, reshape((e_1, ..., e_n), b))
```

Conceptually, the producer applies the function `f` to every element in `a`. We reshape `a` to be n-dimensional and change the producer to be a depth-n nest:

```
let {b} = mapT^n(f, reshape((e_1, ..., e_n) a)) in
reduceT(g, b)
```

In the resulting program, `mapT`-`reduceT`-fusability is exposed.

Again, we need ancillary judgements. These are entirely analogous to the "transposed" and "untransposed" judgements.

$$\boxed{\text{reshaped}(os, \overline{es}) \Rightarrow (k, n)}$$

$$\frac{\Re(e) = v \qquad (v \in os)}{\text{reshaped}(os, \texttt{reshape(}shape\texttt{,}e\texttt{)}) \Rightarrow shape}$$

$$\frac{\Re(e) = v \qquad \text{reshaped}(os, \overline{es}) \Rightarrow shape \qquad (v \notin os)}{\text{reshaped}(os, e, \overline{es}) \Rightarrow shape}$$

$$\boxed{\text{unreshape}(shape, os, \overline{es}) \Rightarrow \overline{es'}}$$

$$\frac{}{\text{unreshape}(shape, os, \texttt{reshape(}shape\texttt{,}e\texttt{)}) \Rightarrow e}$$

$$\frac{(e \text{ is not a reshaping})}{\text{unreshape}(shape, os, e) \Rightarrow e}$$

$$\frac{\Re(e) = v \qquad \text{unreshape}(shape, os, \overline{es}) \Rightarrow \overline{es'} \qquad (v \in os)}{\text{unreshape}(k, n, os, \texttt{reshape}(shape, e), \overline{es}) \Rightarrow e, \overline{es'}}$$

$$\frac{\Re(e) = v \qquad \text{unreshape}(shape, os, \overline{es}) \Rightarrow \overline{es'} \qquad (v \notin os)}{\text{unreshape}(shape, os, e, \overline{es}) \Rightarrow e, \overline{es'}}$$

The fusion rule is also extremely similar to Pull-Transpose.

$$\frac{\text{reshaped}(os, \overline{es_c}) \Rightarrow (e_1^s, \ldots, e_n^s) \qquad \text{unreshaped}((e_1^s, \ldots, e_n^s), os, \overline{es_c}) \Rightarrow \overline{es'_c} \quad \texttt{mapT}^n\texttt{(}f\texttt{, reshape((}e_1^s, \ldots, e_n^s\texttt{),} \overline{es_p}\texttt{))} \overset{os}{\rightsquigarrow}_{\Rightarrow soac_r} \texttt{mapT(}g\texttt{,} \overline{es'_c}\texttt{)} \quad (\text{All of } \overline{es_p} \text{ have rank 1})}{\texttt{mapT(}f\texttt{,} \overline{es_p}\texttt{)} \overset{os}{\rightsquigarrow}_{\Rightarrow soac_r} \texttt{mapT(}g\texttt{,} \overline{es_c}\texttt{)}}$$

(Pull-Reshape)

```
scanT(fn [real] ([real] x, [real] y) =>
        mapT(op +, x, y),
      {0.0, ..., 0.0}, a)
```

(a) Unfusible

```
transpose(
  mapT (fn [real] ([real] x) =>
          scanT(op +, 0.0, x),
                transpose(a)))
```

(b) Potentially fusible

Figure 35: Interchange Scan With Inner Maps

8.3 ISWIM - Interchange Scan With Inner Maps

The fusion algebra for `scanT` is quite poor – in particular, it can never be fused as a producer. In some cases however, we can rewrite `scanT` to expose producer-fusibility. This section presents a high-level transformation that may enable fusion of `scanT`. Specifically, when the body of a `scanT` operation consists of a nested `mapT`, we can interchange the two loops and transpose both input and output. A simple example to demonstrate the intuitive idea is illustrated on Figure 35. Using Haskell-like notation, a `scan` operation on a matrix in which the binary associative operator is `zipWith` $\odot$ has the same semantics as transposing the matrix, mapping each of the rows, i.e., former columns, with `scan` $\odot$ and transposing back the result.

In principle, this transformation interchanges the `scanT` with the inner `mapT`, hence ISWIM, with the result that the transformed code can be executed as a segmented scan [9], i.e., exploiting both levels of parallelism. With `scanT` on the outside, we would have to choose between the parallel `scanT` and the parallel `mapT`. Furthermore, pushing the least parallel construct, i.e., `scanT`, at the innermost position might reveal a deeper `mapT`-nest, e.g., if the original `scanT` was inside a `mapT` itself, thus increasing the depth of parallelism. Finally, if the created `mapT` nest exhibits enough parallelism, then the `scanT` can be executed sequentially rather than in parallel. In this way, the ISWIM transformation is not solely about enabling fusion, but is worthwhile on its own.

Two fusion rules are defined. One where ISWIM is applied to the producer, and one where it is applied to the consumer.

$$\frac{soac_p \overset{os}{\rightsquigarrow} \texttt{transpose(mapT(scanT(}f\texttt{),}\ \{e_1^v,\ldots,e_k^v\},e_1^a,\ldots,e_k^a\texttt{))} \Rightarrow soac_r}{soac_p \overset{os}{\rightsquigarrow} \texttt{scanT(mapT(}f\texttt{),}\{e_1^v,\ldots,e_k^v\},e_1^a,\ldots,e_k^a\texttt{)} \Rightarrow soac_r}$$

(Fuse-ISWIM-Consumer)

$$\frac{\texttt{transpose(mapT(scanT(}f\texttt{),}\ \{e_1^v,\ldots,e_k^v\},e_1^a,\ldots,e_k^a\texttt{))} \overset{os}{\rightsquigarrow} soac_c \Rightarrow soac_r}{\texttt{scanT(mapT(}f\texttt{),}\{e_1^v,\ldots,e_k^v\},e_1^a,\ldots,e_k^a\texttt{)} \overset{os}{\rightsquigarrow} soac_c \Rightarrow soac_r}$$

(Fuse-ISWIM-Producer)

ISWIM depends critically on the fusion algorithm being able to fuse through `transpose`. There is also a further generalisation for ISWIM, illustrated on Figure 36, which permits the inner `mapT` to be arbitrarily nested, but it has not yet been implemented in the $\mathcal{L}_0$ compiler.

```
scanT( fn ( [1[...n+1α1]], ..., [1[...n+1αk]] )
          ( [1[...n+1α1]] x1^1, ..., [1[...n+1αk]] xk^1,
            [1[...n+1α1]] x1^2, ..., [1[...n+1αk]] xk^2 ) =>
              mapT^(n+1)(⊕, x1^1,..., xk^1, x1^2,..., xk^2),
     (ne1, ..., nek), a1, ..., ak)
              ≡
let (..., ret, ...) = (..., map_{n+1}( replicate(1), net ), ...)
// Replicate dimension n+1 of neutral elements so mapT sizes match
let ( y1, ..., yk ) =
  mapT(fn ( [1[...n+1α1]],    ..., [1[...n+1αk]] )
          ( [1[...n+1α1]] x1, ..., [1[...n+1αk]] xk ) =>
          mapT^n(fn (α1,  ..., αk)
                    (α1 e1,...,αk ek,
                     α1 x1,...,αk xk)
                      =>scanT(⊕, e1[0],...,ek[0], x1,...,xk),
                re1, ..., rek, x1, ..., xk ),
       transpose(1,n+1,a1), ..., transpose(1,n+1,ak))
in (transpose(n+1, q1-(n+1), y1), ..., transpose(n+1, qk-(n+1), yk))
// transpose back the result; qt is the dimension of αt
```

Figure 36: Arbitrary-depth generalisation of ISWIM

8.4 Fusing A Transposed Producer

While an input program will never contain a producer of the form `transpose(`k`,`n`,`$soac$`)`, the ISWIM transformation may create them. To fuse these, we first move the transpositions to the inputs of the consumer.

We need yet another ancillary judgement:

$$\boxed{\text{transpose}(k, n, os, \overline{es}) \Rightarrow \overline{es'}}$$

This judgement wraps every input in $\overline{es}$ whose source array is in os in an (k, n)-transposition, producing $\overline{es'}$. It is defined by the following inference rules:

$$\frac{\Re(e) = v \qquad (v \in os)}{\text{transpose}(k, n, os, e) \Rightarrow \texttt{transpose(}k\texttt{,}n\texttt{,}e\texttt{)}}$$

$$\frac{\Re(e) = v \qquad (v \notin os)}{\text{transpose}(k, n, os, e) \Rightarrow e}$$

$$\frac{\Re(e) = v \qquad (v \in os) \qquad \text{transpose}(k, n, os, e, \overline{es}) \Rightarrow \overline{es'}}{\text{transpose}(k, n, os, e, \overline{es}) \Rightarrow \texttt{transpose(}k\texttt{,}n\texttt{,}e\texttt{)}, \overline{es'}}$$

$$\frac{\Re(e) = v \qquad (v \notin os) \qquad \text{transpose}(k, n, os, e, \overline{es}) \Rightarrow \overline{es'}}{\text{transpose}(k, n, os, e, \overline{es}) \Rightarrow e, \overline{es'}}$$

We can now define a fusion rule. First, we extract the array inputs of the consumer (not formalised), then transpose those whose source arrays come

from the producer, producing $\overline{es'}$. Finally, we attempt fusion where we have substituted the original array inputs in the consumer with $\overline{es'}$.

$$\frac{\text{Inputs of } soac_c \text{ is } \overline{es} \qquad \text{transpose}(k, n, os, \overline{es}) \Rightarrow \overline{es'} \qquad soac_p \overset{os}{\rightsquigarrow} soac_c \text{ with inputs } \overline{es'} \Rightarrow soac_r}{\texttt{transpose}(k,\ n,\ soac_p) \overset{os}{\rightsquigarrow} soac_c \Rightarrow soac_r}$$

(Fuse-Transposed-Producer)

Chapter 9

Hindrance Removal

The fusion algorithm presented in previous chapters assumed an input program with a structure that made any possibilities for fusion as explicit as possible. The most obvious such structure is the normalised input program and the use of tupleless SOACs, but there are other transformations we can do in order to enable more possibilities for fusion.

In particular, recall that the fusion algorithm is very strict about never duplicating computation, and hence multiple uses of the output of a SOAC may easily block any fusion of the SOAC. We call such a use a *hindrance*, with an example shown on Figure 37a. In some cases, the hindrances are unavoidable, but in other cases, a pre-fusion transformation of the program can remove some unnecessary hindrances.

Section 9.1 will cover cases where we can rewrite `size`-expressions that reference a SOAC output. Although primarily concerned with increasing fusibility, Section 9.1.3 will describe how removing hindrances can also enable hoisting, particularly of bounds checks. Section 9.2 will describe inlining of index expressions where the index array is the result of a `map` operation. This inlining may duplicate a small amount of computation.

An important detail is that neither of the presented transformations should be considered optimisations *per se*. Rather, their purpose is enable the fusion optimisation to apply more often.

Both transformations are run completely independently (and in advance) of fusion. This results in greater conceptual and technical simplicity, but at some cost in precision. In particular, the rewriting of `size`-expressions can in fact inhibit fusion in some cases.

```
let b = map(f, a) in
size(0,b) + reduce(op +, 0, b)
```

(a) Hindrance blocking fusion

```
let b = map(f, a) in
size(0,a) + reduce(op +, 0, b)
```

(b) Hindrance removed

Figure 37: Typical case of `size`-hindrance

9.1 Size Hindrance Removal

Consider the program shown on Figure 37. The output of the `map` producer, `b`, is used in two places - as input to a consumer `reduce`, and as argument to a `size` expression. This means that fusing the two SOACs would duplicate computation, as we be unable to remove the original `map` expression.

Fortunately, in this case, we can exploit a property of `map` to rewrite the `size` expression. Specifically, the outer size of the array output from a `map` expression is equal to the outer size of its array inputs. In the context of Figure 37, this means that the expression `size(0,b)` will always give the same value as `size(0,a)`. Hence, we can rewrite the program as shown on Figure 37(b), which has now become fusible.

At first glance, removing `size` hindrances may appear to rarely be useful, but in fact, it is crucial to making the fusion algorithm perform well in practice. The reason is that the assertions described in Section 4.2 check the dimensions of various arrays via `size` expressions. Hence, after the transformation from external to internal $\mathcal{L}_0$, we will have generated several `size` expressions, many of which may act as hindrances to fusion. As an example, consider this simple program:

```
fun [int] main([int] a, [int] b) =
  let a2 = map(op+(1), a) in
  let b2 = map(op+(2), b) in
  map(op+, zip(a2, b2))
```

After transformation to internal $\mathcal{L}_0$, we obtain the following (slightly denormalised for readability):

```
fun [int] main([int] a, [int] b) =
  let {a2} = mapT(op + (1), a) in
  let {b2} = mapT(op + (2), b) in
  let a2_sz = size(0, a2) in
  let b2_sz = size(0, b2)) in
  let zip_assert = assert(a2_sz = b2_sz) in
  let {res} = <zip_assert>mapT(op + a2, b2) in
  res
```

Two `size`-hindrances are present. Since `a2` and `b2` are the outputs of mapping over `a` and `b` respectively, we can rewrite `size(0,a2)` to `size(0,a)` and `size(0,b2)` to `size(0,b)`, thus removing the nuisances and turning the program fusible.

Our chosen approach is quite simple: Traverse the program, and whenever an expression of the form `size(`k`,`v`)` is encountered, see if it can be rewritten to a "better" form. In most cases, we will have several alternative expressions to choose from, and hence we need a way to determine the best replacement.

For our purposes, we will want the expression that has the least chance of being a hindrance to fusion. As noted in the introduction to this chapter, hindrance removal is done outside of the fusion module, and hence we do not have access to precise information about whether a candidate replacement expression

```
fun [int] main([int] a) =
  let b = map(op + (1), a) in
  let c = map(op + (2), b) in
  let n = size(0, c) in
  let d = map(op + (n), c) in
  d
```

Figure 38: Multiple potential hindrance replacements

removes a potential hindrance, or perhaps even moves it. Section 9.1.2 will describe cases in which moving `size`-expressions may cause new hindrances to appear.

It is conceptually simple to generate alternatives to the expression `size(`k`,`v`)`. During traversal of the program, we track the binding of all array-typed variables in a symbol table mapping variable names to static size information. For example, after seeing the binding `let a = iota(`e`)`, we know that `size(0,a)` can be rewritten to e. The details of size analysis are described in Section 9.1.1. However, in some cases there may be several possible replacement expressions, and we need a way to select the best one.

We define a heuristic determining the *quality* of a candidate expression e as follows: For every free variable v_i in e, determine the data-flow path from v_i to either a constant or a function parameter. The quality of the expression is inversely proportional to the number of nodes in this path, exluding nodes that are simply copies or indexing. That is, the expression with the lowest number is best. The idea behind this heuristic is to choose the expression that we can move the furthest up the program, ideally preceding all SOACs.

For example, for the program shown on Figure 38, the hindrance `size(0, c)` can be replaced with either `size(0, b)` or `size(0, a)`. We pick the latter, because its single free variable can be traced directly to a function parameter (`a`), whereas the free variable in the former can only reach a function parameter through the binding for `b`.

In the $\mathcal{L}_0$ compiler, size hindrance removal is implemented as part of the Rebinder introduced in Section 6.3. Array sizes are tracked by inspecting bindings during the traversal of the syntax tree, as described in the next section. Whenever we encounter a binding of a `size`-expression, we use the size information to obtain candidate replacements, then use the quality heuristic to determine the best replacement.

9.1.1 Size analysis

To aid in rewriting `size` expressions, the Rebinder maintains a symbol table mapping variables to size information. The size analysis presented in this section is an entirely *ad hoc* mechanism focused solely on the removal of `size` nuisances. More sophisticated size analysis, which could be used to optimise memory allocation in the code generator, is left as a future project.

The size information we store takes the form of a sequence of sets of expressions, with the set at index i representing the various expressions that evaluate to the size of that dimension. For example, we may have the following

mapping in the symbol table:

$$a \mapsto \langle \{10\}, \emptyset, \{\texttt{size(0,}b\texttt{)}, \texttt{size(1,}c\texttt{)}\}]$$

This indicates that the the first dimension of a has size 10, we know nothing of the second dimension, and the size of the third dimension is equal to the first dimension of b or the second dimension of c. If the symbol table contains bindings for b and c, we can look them up recursively and find even more accurate size information.

If a mapping refers to an array variable with n dimensions, the mapping may contain less than n sets. This implicitly means that we know nothing (i.e. $\emptyset$) about the excess dimensions.

The mappings generated by different bindings are given below. Note that not all bindings generate a mapping; this means that the symbol table does not necessarily include size information for all variables in scope.

`let` a `=` `iota(`e`)`
$a \mapsto \langle \{e\} \rangle$

`let` a `=` `replicate(`e_n`,` e_v`)`
We know that the number of rows in a is e_n, but we also know that the size of dimension d of a will be the size of dimension $d+1$ in e_v. This is reflected in the size binding:
$a \mapsto \langle \{e_n\}, \{\texttt{size(0,}e_v\texttt{)}\}, \ldots, \{\texttt{size(}n\texttt{,}e_v\texttt{)}\} \rangle$, where n is the rank of e_v.

`let {`a`,`b`} = split(`e_n`,` e_v`)`
The semantics of `split(`e_n`,` e_v`)` is that the first returned array contains the initial e_n elements, while the remaining `size(0,`e_v`)-`e_n are in the the second returned array. This leads to the following size bindings:
$a \mapsto \langle \{e_n\}, \{\texttt{size(1,}e_v\texttt{)}\}, \ldots, \{\texttt{size(n,}e_v\texttt{)}\rangle$
$b \mapsto \langle \{\texttt{size(0,}e_v\texttt{)} - e_n, e_n\}, \{\texttt{size(1,}e_v\texttt{)}\}, \ldots, \{\texttt{size(n,}e_v\texttt{)}\rangle$
Where n is the rank of e_v.

`let` a `=` `concat(`e_x`,` e_y`)`
$a \mapsto \langle \{\texttt{size(0,}e_x\texttt{)+size(0,}e_y\texttt{)}\}, \{\texttt{size(1,}e_x\texttt{)}, \texttt{size(1,}e_y\texttt{)}\}, \ldots, \{\texttt{size(n,}e_x\texttt{)}, \texttt{size(n,}e_y\texttt{)}\rangle$, where n is the rank of e_x and e_y (the same, according to the type rules of $\mathcal{L}_0$).

`let` a `=` b`[`$e_0, \ldots, e_n$`]`
$a \mapsto \langle \{\texttt{size(}n+1\texttt{,}b\texttt{)}\}, \ldots, \{\texttt{size(}m\texttt{,}b\texttt{)}\} \rangle$, where m is the rank of b.

`let` a `=` `transpose(`e`)`
$a \mapsto \langle \{\texttt{size(1,}e\texttt{)}\}, \{\texttt{size(0,}e\texttt{)}\}, \{\texttt{size(2,}e\texttt{)}\}, \ldots, \{\texttt{size(}n\texttt{,}b\texttt{)}\} \rangle$, where n is the rank of b.

`let` a `=` b `with [...] <-` e
$a \mapsto \langle \{\texttt{size(0,}b\texttt{)}\}, \ldots, \{\texttt{size(}n\texttt{,}b\texttt{)}\} \rangle$, where n is the rank of b.

`let {`$a_1, \ldots, a_k$`} = mapT(fn` t `(`$p_1, \ldots, p_n$`) =>` e`,` e_1`, ...,` e_n`)`
Within the body of the SOAC function (e), the symbol table will map the parameters to row slices of their corresponding arrays:

$p_1 \mapsto \langle\{\texttt{size(1, } e_1\texttt{)}\}, \ldots, \{\texttt{size(m, } e_1\texttt{)}\}\rangle$, where m is the rank of e_1.

$\vdots$

$p_n \mapsto \langle\{\texttt{size(1, } e_n\texttt{)}\}, \ldots, \{\texttt{size(}m\texttt{, } e_n\texttt{)}\}\rangle$, where m is the rank of e_n.

Additionally, we know by the semantics of `mapT` that the outer size of any a_i must match the outer size of any e_j. This gives rise to the following mappings:

$a_1 \mapsto \langle\{\texttt{size(0, } e_j\texttt{)} \mid 1 \leq j \leq n\}\rangle$.

$\vdots$

$a_k \mapsto \langle\{\texttt{size(0, } e_j\texttt{)} \mid 1 \leq j \leq n\}\rangle$.

Similar rules apply to the other SOACs, but a few are interesting enough to be mentioned explicitly.

`let` $\{a_1, \ldots, a_k\}$ `=` $\texttt{mapT}^d\texttt{(fn } t \texttt{ (}p_1, \ldots, p_n\texttt{) => } e\texttt{, } e_1\texttt{, ..., } e_n\texttt{)}$

We create the same bindings as above for the function parameters, but the interesting fact is that since we are dealing with a d-deep mapping, the outer d dimensions of the output correspond to the outer d dimensions of the input. This gives rise to the following mappings:

$a_1 \mapsto \langle\{\texttt{size(0, } e_j\texttt{)} \mid 1 \leq j \leq n\}, \ldots, \{\texttt{size(}d\texttt{, } e_j\texttt{)} \mid 1 \leq j \leq n\}\rangle$.

$\vdots$

$a_k \mapsto \langle\{\texttt{size(0, } e_j\texttt{)} \mid 1 \leq j \leq n\}, \ldots, \{\texttt{size(}d\texttt{, } e_j\texttt{)} \mid 1 \leq j \leq n\}\rangle$.

`let` $\{a_1, \ldots, a_n\}$ `=`
$\texttt{scanT(fn } t \texttt{ (}p^v_1, \ldots, p^v_n\texttt{, } p^a_1, \ldots, p^a_n\texttt{) => } e\texttt{, } \{v_1, \ldots, v_n\}\texttt{, } e_1\texttt{, ..., } e_n\texttt{)}$

Within the body of the SOAC function (e), the symbol table will map the parameters to row slices of their corresponding arrays. Additionally, by the semantics of `scanT`, we know that the inner size of e_i must be equal to the outer size of v_i:

$p^a_1 \mapsto \langle\{size(1, e_1), \texttt{size(0, } v_1\texttt{)}\}, \ldots, \{size(m, e_1), \texttt{size(}m - 1\texttt{, } v_1\texttt{)}\}\rangle$, where m is the rank of e_1.

$\vdots$

$p^a_n \mapsto \langle\{size(1, e_n), \texttt{size(0, } v_n\texttt{)}\}, \ldots, \{size(m, e_n), \texttt{size(}m - 1\texttt{, } v_n\texttt{)}\}\rangle$, where m is the rank of e_n.

Additionally, we also know by the semantics of `scanT` that the outer size of any a_i must match the outer size of any e_j. This gives rise to the following mappings:

$a_1 \mapsto \langle\{\texttt{size(0, } e_j\texttt{)} \mid 1 \leq j \leq n\}\rangle$.

$\vdots$

$a_n \mapsto \langle\{\texttt{size(0, } e_j\texttt{)} \mid 1 \leq j \leq n\}\rangle$.

There is no rule for the binding of array literals - this is handled by the constant folder presented in Section 5.3. Furthermore, a `size` expression depending on an array literal binding will not ever inhibit fusion.

9.1.2 Accidentally Adding Hindrances

In some cases, our aggressive rewriting of `size` expressions may in fact *create*, rather than *remove* hindrances. Consider the following program:

```
let b = map(f, a) in
let c = map²(g, b) in
let k = size(1,c) in
h(k,c)
```

Here, there is a clear opportunity for fusing the two `map`s. Note that the function `f` is opaque, and we cannot know the size of the arrays it returns. Since `c` is the result of a `map`*myindu*2 operation, the Rebinder can change `size(1,c)` to `size(1,b)`, resulting in the following program:

```
let b = map(f, a) in
let k = size(1,b) in
let c = map²(g, b) in
h(k,c)
```

The `size` expression is now a nuisance preventing fusion. One possible solution, and the one taken in the current $\mathcal{L}_0$ compiler, is a preliminary fusion stage, prior to executing the Rebinder, then run the Rebinder and re-run fusion. A more precise solution would be to integrate nuisance removal into the fusion algorithm, but this requires careful engineering in order to keep the resulting compiler code complexity under control. Alternatively, it may be possible to tweak the rules in the Rebinder to remove the possibility of creating fusion hindrances, although this has not been investigated in depth.

9.1.3 Size Hindrance Removal as a Hoisting Enabler

This section has been solely concerned with `size` nuisance removal as a transformation to enable fusion. However, it is equally useful in enabling hoisting. Consider the following program:

```
mapT(fn {[int]} ([int] ar, [int] br) =>
       let c = assert(size(0,ar) == size(0,br)) in
       mapT<c>(op +, ar, br),
     a, b)
```

Size analysis will reveal that `size(0,ar)` can be rewritten to `size(1,a)`, and `size(0,br)` to `size(1,b)`, which can then be hoisted out of the loop. This results in the following program:

```
let c = assert(size(1,ar) == size(1,br)) in
mapT(fn {[int]} ([int] ar, [int] br) =>
       mapT<c>(op +, ar, br),
     a, b)
```

Not only does this result in removing the assertion from the inner loop, but the result is a perfect map nest, potentially permitting fusion across transpose.

9.2 Inlining of indexing

The fusion algorithm presented in Chapter 7 is very strict about never duplicating computation, to the point where otherwise beneficial fusion is prevented. For example, consider this program:

```
let b = map(f, a) in
reduce(min, b[0], b)
```

Fusion is not possible, as `b` is used in multiple places. If duplication of computation were acceptable, we could rewrite `b[0])` as `f(a[0])`, and get the following program:

```
let b = map(f, a) in
reduce(min, f(a[0]), b)
```

We could then perform `map-reduce` fusion and obtain a fully fused `redomap` expression. If `f` is cheap, it is very likely that the small duplication of computation is worthwhile.

Similarly to size hindrance removal, we can implement this as a transformation performed before running the fusion algorithm. Specifically, when we find an expression of form `b[i]`, where `b` is the result of an expression `map(f, a)` and `f` is *cheap* (see below), we rewrite `b[i]` to `f(a[i])`, essentially inlining part of the `map` operation.

A function is considered cheap if its body executes in constant time - notably, no SOACs. We must also be careful not to inline into a loop, as this would duplicate more than a constant amount of computation. This implies that every such instance of inlining at most results in duplicating a constant amount of work.

Part III

Evaluation

Chapter 10

Optimisation Results

It is difficult at this point to quantitatively report the impact of fusion and our other optimisations, because the compiler does not yet produce quality parallel code. There are three main problems:

1. The optimisation is "incomplete", in the sense that we are still too conservative about duplicating trivial computation. In addition, we have no heuristics for avoiding fusion in cases where the added memory traffic becomes a detriment (as outline Section 7.1.3).

2. There is not yet a way to execute $\mathcal{L}_0$ code in an efficient manner. The $\mathcal{L}_0$ compiler has an interpreter, but its performance characteristics are very different from parallel hardware – for example, variable bindings carry great overhead.

   The compiler also has a code generator which generates strictly sequential C code. The resulting C code uses very naïve memory management, however. In particular it copies arrays very often when executing SOACs, although this might put the fusion optimisation in a better light, as it will reduce the number of distinct SOAC expressions in the program.

3. Finally, fusion, which is our primary optimisation, does not really reduce the number of discrete computation steps necessary to execute the program. The purpose of our fusion optimisation is to increase parallelism and reduce the number of discrete GPU kernels, which is not something that will benefit the sequential code generated by our code generator.

Nevertheless, this chapter presents an evaluating the impact of the fusion optimisation. This will primarily be in the form of manual inspection of program structure before and after optimisation, with comments on the quality of the result. The reader can be assured that said inspection of hundreds of lines of machine-generated code was enormously tedious.

Six programs will be used for evaluation: three relatively simple, artificial benchmarks, and three real-world financial programs that have been manually

translated from C++ to what we consider "idiomatic" $\mathcal{L}_0$[1]. We present run-time statistics for these programs in Section 10.1.

The code for the artificial benchmarks can be found in Part IV, as well as the programs resulting from optimisation, but are summarised here:

P0 Black-Scholes[7] pricing computation. 34 SLOC (Source Lines Of Code - ignoring comments and blank lines).

P1 Matrix multiplication written in a functional style (i.e, no use of `loop` and `let-with`). 13 SLOC.

P2 Shortest path algorithm written in a functional style. 27 SLOC.

The real world benchmarks are as follows.

R0 A stochastic option pricing engine. The optimisation of this program has previously been studied in the literature[33]. 344 SLOC.

R1 A program for doing stochastic volatility calibration, i.e., given a set of (observed) prices of contracts, we identify the parameters of a model of such prices, as a function of volatility (unknown), time and strikes (known), and unobserved parameters like alpha, beta, nu, etc.

In this program, the volatility is modelled as a system of continuous partial differential equations, which are solved via Crank-Nicolson's finite differences method[15].

172 SLOC.

R2 A dynamic evolution model method, i.e., genetic algorithm, for calibrating the interest rate based on a known history of swaption prices.

Briefly, the interest rate is modelled as a sum of two stochastic processes, which gives four unknown (real) parameters, and in addition the two processes are assumed correlated as well, i.e., a fifth parameter.

These five (unknown) parameters appear in the formula that computes the swaption's price, i.e., numerical integration via hermitian-polynomials approximation.

The genetic algorithm is used to find the five parameters that best fit the (known) history of swaption prices.

798 SLOC.

The structure of the artificial benchmarks are shown on Figure 39. P0, being a straightforward sequence of four `map`s, fuses well. P1 also fuses well - the main loop becomes a two-dimensional `tmap`, with the dot product at each location being computed in a `redomap`. Although not visible in the data flow diagram, it is worth remarking that hoisting has moved all `assert` expressions (originating in the use of `zip`) out of the main loop, which can thus be evaluated with no bounds checking - or indeed, any branching at all.

[1] Or at least as much as it makes sense to talk about an "idiomatic" style for a language whose sole users are also its designers.

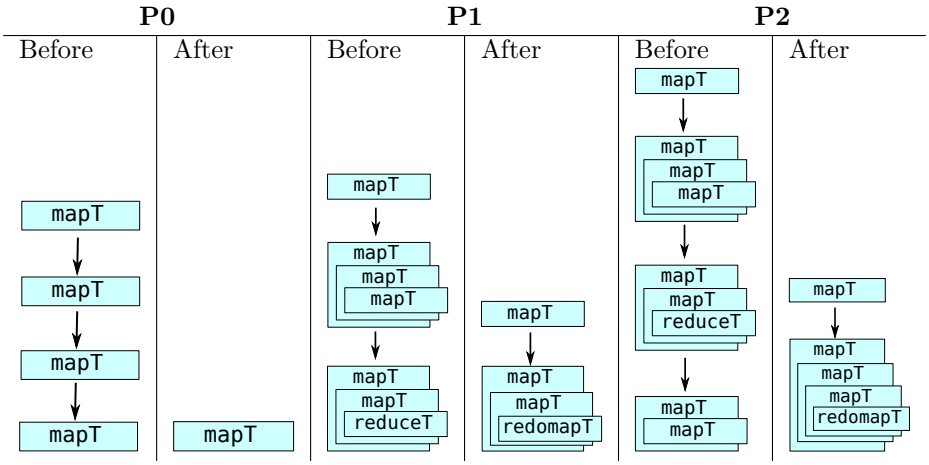


Figure 39: Artificial benchmark dataflows, before and after optimisation

There is clearly a missed opportunity for fusion, though, the reason for which becomes clear when we inspect the code around the unfused `map`:

```
...
let untuple_13 =
  mapT(fn {[[int]]} ([int] param_0_8) =>
         // tmp_repl_11 aliases param_0_8
         let tmp_repl_11 = replicate(N_2, param_0_8) in
         {tmp_repl_11},
       x_0) in
let tmp_size_14 = size(2, untuple_13) in
... // untuple_13 is eventually input to main loop.
```

The size analyser is not smart enough to rewrite the `size` expression, and `untuple_13` is thus used several times, blocking fusion. The most reasonable solution is to improve the size analyser, for which a potential approach is outlined in Section 11.2. P2 suffers from the same problem, although again the main loop is fully fused.

When illustrating the dataflow for the real-world benchmarks, I performed some minor simplifications. Specifically, prologue and epilogue code has been removed in order to emphasise the main loop.

Of the real-world benchmarks, R0, whose dataflow is illustrated on Figure 40, benefits the most from optimisations. The program is turned into a big `redomapT` that runs over an array of a thousand elements. The body of the `redomapT` runs three loops in sequence. The two first could in principle be fused, but we are again foiled by limitations of the size analyser. In this case, the use of an explicit `loop` prevents the size analyser from determining the column size of the two-dimensional array returned by the `mapT`. The fusion of R0 also requires fusing across `transpose` and `reshape`.

For R1, the gains are more muted. The overall structure is a sequential, iterative main loop, which of course limits what we can do, but the body of this loop can in principle be parallelised. The unoptimised and optimised loop

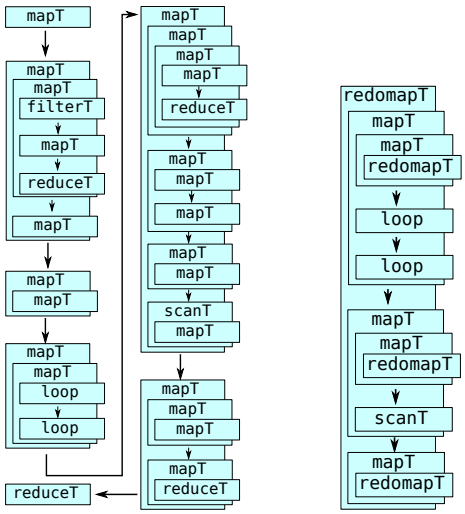


Figure 40: R0 benchmark dataflow, before and after optimisation

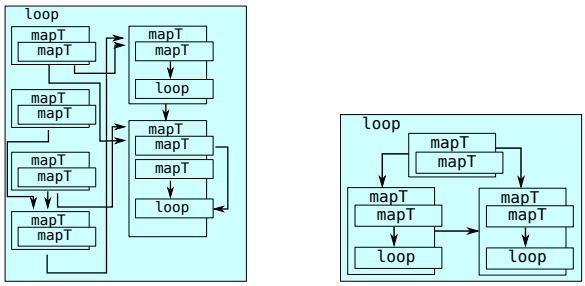


Figure 41: R1 benchmark dataflow, before and after optimisation

bodies can be seen on Figure 41. At first sight, two possible avenues for further fusion are possible:

1. The first `mapT` could be fused into its two consumers. While this would surely duplicate computation, perhaps it is worthwhile in this case. Inspecting the code, which is shown in Figure 42, we find that the computation that would be duplicated for each element is approximately four primitive arithmetic operations, and two calls to exponent and logarithm functions. Such duplication would likely be acceptable in this case, as it enables an instance of fusion.

2. The reason for why the two latter `mapT`s are not fused is more tricky. Although not expressed in the diagram, the input to the rightmost `mapT` is *transposed*, and we have no fusion rule capable of handling a transposition in this case, as neither consumer nor producer is a map nest.

   It is not immediately clear how this could be solved.

```
mapT(fn {*[real], *[real], *[real], *[real]} (real xi_481) =>
       let tmp_call_488 = log(xi_481) in
       let bop_493 = 0.5 * tmp_call_488 in
       let {soac_v_506, soac_v_507, soac_v_508, soac_v_509} =
         mapT(fn {real, real, real, real} (real yj_495) =>
                let bop_496 = bop_493 + yj_495 in
                let bop_498 = bop_496 - bop_477 in
                let val_504 = 2.0 * bop_498 in
                let tmp_call_505 = exp(val_504) in
                {0.0, tmp_call_505, 0.0, 0.36},
              untuple_247) in
       {soac_v_506, soac_v_507, soac_v_508, soac_v_509},
     untuple_130)
```

Figure 42: Unfused map in R1

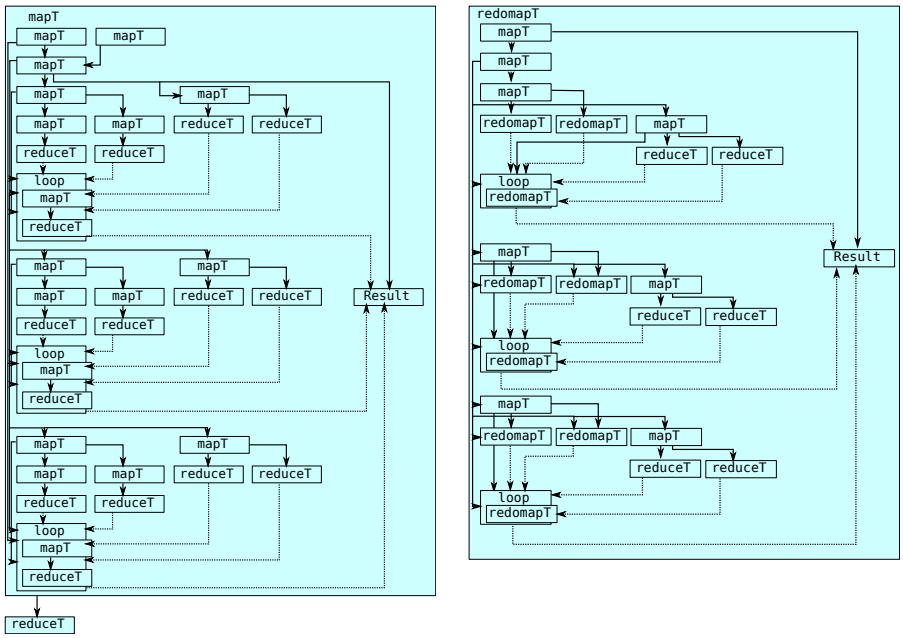


Figure 43: R2 benchmark dataflow, before and after optimisation

```
...
let {soac_v_685, soac_v_686} =
  mapT(fn {real, real} (real arg_675, real arg_676, real arg_677, real arg_678) =>
         let baix_679 = arg_675 * mux_239 in
         {arg_677 * exp(-baix_679),
           (arg_678 - baix_679) / arg_676},
       soac_v_669, soac_v_670, soac_v_671, soac_v_672) in
let {untuple_690} =
  reduceT(fn {real} (real x_687, real y_688) =>
            {x_687 + y_688},
          {0.0}, soac_v_685) in
let {untuple_695} =
  reduceT(fn {real} (real param_0_691, real param_1_692) =>
            if param_0_691 < param_1_692
            then {param_1_692}
            else {param_0_691},
          {-10000000000000000000000000000000000000000000000000.0},
          soac_v_686) in
...
```

Figure 44: Unfused loops in R2

R2 is easily the most complex benchmark, and also the one for which fusion has the smallest impact on the data flow graph. As shown on Figure 43, the body of the main loop contains three independent (but near-identical) loops whose results are combined using a non-fusible series of reductions (summarised as a single node). The optimised structure is virtually identical: the only optimisation is a few instances of `map-map` and `map-reduce` fusion. However, it is worth noting that there are instances where we take advantage of our fusion algorithms ability to fuse just part of the input to a SOAC.

As in R1, each of the three inner loops have a case where multiple uses of the output of a `mapT` SOAC prevents us from fusing it into a `reduceT`. Again, it is worth inspecting the code to see whether our reluctance to duplicate computation is again too conservative. The code in question (slightly simplified for readability) is shown on Figure 44.

Again we see that that a relatively cheap `mapT` operation cannot be fused without duplicating computation. This case is particularly interesting, because the `mapT` conceptually computes two distinct arrays, with only the computation of `baix_679` (a single multiplication per element) being shared. This structure was also present in the original program. In a future elaboration of the fusion algorithm, it may be worthwhile to split SOACs apart to make the representation of shared computation even more precise.

10.1 Runtime results

The real-world benchmarks were repeatedly passed through the $\mathcal{L}_0$ optimisations until no further optimisation was achieved, then compiled with a code generator generating sequential C code.

The resulting programs were compiled with GCC 4.8.2 using maximum

	Unoptimised	**Optimised**	**Speedup**
R0	0.430s	0.292s	46%
R1	0.098s	0.057s	71%
R2	0.061s	0.047s	31%

Figure 45: Benchmark runtimes

optimisation (`-O3`) on an Intel Core i7-2630QM CPU running at 2.00GHz. Each program was executed one thousand times and the run-times averaged. The results are shown on Figure 45

It is hard to determine how much of the speedup is due to fusion in isolation and how much is due to other optimisations, as they all interact to enable each other. However, given that the C programs were compiled with full optimisation, it is likely that the C compiler performed much hoisting and most of our simpler optimisations itself.

Again, it must be emphasised that most of the speedup is likely due to the code generator copying all input arrays upon executing a SOAC. With this behaviour, fusion reducing the number of discrete SOAC expressions will likewise reduce the number of expensive memory copies.

Chapter 11

Conclusions

This chapter summarises the result of our work. Section 11.1 compares our fusion algorithm with fusion in other data-parallel programming languages, as well as looking at other approaches to uniqueness typing. Section 11.2 outlines a number of possible future improvements to $\mathcal{L}_0$ and the compiler. Section 11.3 provides a final summary of the results of this thesis.

11.1 Related Work

Our approach to performining fusion, via rewrite rules, is not unique by itself, as this is the approach used in e.g. Data-Parallel Haskell [12] (DPH). What sets us apart is the fact that our rewriting rules are defined on the dataflow graph, and not the program itself. While DPH obtains good results, its rewrite rules are quite limited – they are an inherently local view of the program, and would be unable to cope with limitations in the presence of in-place array updates, and whether the result of an array operation is used multiple times. The Glasgow Haskell Compiler itself also bases its list fusion on rewrite rules and cross-module inlining [23, 17].

The Repa [24] approach to fusion is based on a delayed representation of arrays, which models an array as a function from index to value. With this representation, fusion happens automatically through function composition, although this can cause duplication of work in many cases. To counteract this, Repa lets the user *force* an array, by which it is converted from the delayed representation to a traditional sequence of values. The pull arrays of Obsidian [14] use a similar mechanism. This approach puts the onus on the programmer to specify points where the manifestation of arrays is beneficial, even though this may be a low-level consideration that depends on details of the target hardware. We therefore consider this a job better suited for the compiler.

Accelerate [29] uses an elaboration of the delayed arrays representation from Repa, and in particular manages to avoid duplicating work. All array operations have a uniform representation as constructors for delayed arrays, on which fusion is performed by tree contraction. Accelerate supports multiple arrays as input to the same array operation (using a `zipWith` construct). Although arrays are usually used at least twice (once for getting the size, once for the data), it does

not seem that they can handle the difficult case where the output of an array operation is used as input to two other array operations.

NESL has been extended with a GPU backend [5], for which the authors note that fusion is critical to the performance of the flattened program. The NESL approach is to use a form of copy-propagation on the intermediary code, and lift the resulting functions to work on entire arrays. This approach only works for what we would term `map-map` fusion, however.

Our uniqueness attributes have some similarities to the "owning pointers" found in the impure language Rust [22], albeit there are deep differences. In Rust, owning pointers are used to manage memory – when an owning pointer goes out of scope, the memory it points to is deallocated – while we use uniqueness attributes to handle side effects. In addition, we allow function calls to consume arrays passed as unique-type parameters, whereas in Rust this causes a deep copy of the object referenced by the owning pointer.

A closer similarity is found in the pure functional language Clean, which contains a sophisticated system of uniqueness typing [4]. Clean employs uniqueness typing to re-use memory in cases where a function receives a unique argument, but also (and perhaps more importantly) to control side effects including arbitrary I/O. As in $\mathcal{L}_0$, alias analysis is used to ensure that uniqueness properties are not violated. A notable difference is that the Clean language itself does not have any facilities for consuming unique objects, apart from specifying a function parameter as unique, but delegate this to (unsafe) internal functions, that are exposed safely via the type system. Furthermore, a unique return value in Clean may alias some of the parameters to the function, which is forbidden in $\mathcal{L}_0$. We have found that this greatly simplifies analysis, and allows it to be fully intraprocedural.

11.2 Future Work

A very important immediate goal is the implementation of a code generator targeting GPU execution. Furthermore, a number of other avenues for further research and development of $\mathcal{L}_0$ ae available.

11.2.1 Size Information in Type System

The size analysis presented in Chapter 9 is quite restricted, and was designed and extended on an ad-hoc basis in order to enable fusion of the real-world benchmarks. Considering the great importance of accurate size information in not only doing high-level optimisations, but also generating efficient low-level code, it appears very worthwhile to integrate tracking of array sizes into the language itself.

We suggest a type system extension inspired by *dependent types*, although much simpler. As an example, let us look at how we would like to be able to define matrix multiplication:

```
fun [[int,N],P] matMult([[int,N],M] a, [[int,M],P] b ) =
  ...
```

This function declares that it takes two `int` array arguments, the first of size $N \times M$ and the second of size $M \times P$, and returns an integer array of size $N \times P$. Any caller of `matMult` must first prove to the type system that the arguments have the correct size, while `matMult` itself must prove that its body always returns an array of the appropriate size.

Such a proof could be provided through a mechanism much like the current `assert`, which allows us a sort of "escape hatch" for when we cannot statically guarantee the size of our data - for example, when it is given to us as input from the outside world, or the result of a `filter`. As the current $\mathcal{L}_0$ compiler is already able to optimise and hoist many assertions away, this would be useful by itself.

However, this would not solve the problem encountered in Chapter 10, when the inability to transform a `size` expression prevented fusion in program R0. The problematic part of R0 has this essential structure (where `N` is some variable in scope):

```
let b = map(fn [real] (int x) =>
              let xa = replicate(N,x) in
              loop (xa) = for i < N do
                let xa = f(xa) // Does not change size of xa
                in
              xa,
            a) in
let n = size(1,b) in
map(g(n), b)
```

The current ad-hoc size analyser is not smart enough to figure out the inner size (`N`) of the array `b`. Integrating size information into the type system would allow us to annotate the return type of the anonymous function as follows:

```
let b = map(fn [real,N] (int x) =>
              ...,
            a) in
...
```

We now statically promise that the inner size of `b` will always be `N`, possibly backed by an assertion within the body of the map. The intent is that this promise can be checked by the type-checker. Presumably, for the body of the function to be type-correct, the function `f` would have been defined to return an array of the same size as its input.

It is not yet clear exactly how we should deal with cases where the size of an array dimension cannot be statically known, or where it is the result of a complex expression. It is not desirable to support the full power of dependent types, nor to include a full theorem prover in $\mathcal{L}_0$, as this could make it very cumbersome to use $\mathcal{L}_0$ as a compiler target language. In the end, it is important to remember that our primary motivation is to improve size tracking for the benefit of optimisation and code generation.

11.2.2 Improved Aliasing Analysis

The system of uniqueness types presented in Chapter 3 hinges crucially on tracking potential sharing between arrays. However, the current model of aliasing is very coarse-grained, as it cannot describe sharing at a more precise level than entire arrays. For example, assume that we have the following function:

```
fun *[int] replace(*[int] arr, int i, int x) =
  let arr[i] = x in arr
```

The result of `replace(a,i,x)` is the array `a` with the element at index `i` replaced by `x`. We may want to use this function to replace an element within a slice of an array, like so[1]:

```
let b = a with [j] <- replace(a[j], j, x) in
...
```

Unfortunately, this code will be refused by the compiler: The call to `replace` consumes the array `a`, because `a[j]` is aliased with `a`, yet `a` is used as the source in a let-binding. Making a separate binding for the call to `replace` may make things more clear:

```
let r = replace(a[j], j, x) in // Consumes a
let b = a with [j] <- r in
...
```

As far as the type system is concerned, both the call to `replace` and the `let-with` expression consume the entirety of `a`, hence causing a compile-time double-consumption error. The only solution is to use `copy`:

```
let r = replace(copy(a[j]), j, x) in // Consumes a
let b = a with [j] <- r in
...
```

It is clear to us however, that the call to `replace` only modifies the memory associated with the `j`th row of `a`. Furthermore, when `a` is next accessed (and consumed), the `j`th row is replaced anyway. Thus, it should be possible to perform this entire operation in $O(1)$ space.

We could of course make a specialised variant of `replace` for two-dimensional arrays, but this leads to unnecessary code bloat. Thus, we believe that more precising tracking of sharing would be worthwhile. The problem is not easy, however, as the expression for the array index may be arbitrarily complicated.

11.2.3 Array Views

Array indexing in $\mathcal{L}_0$ is quite limited - only entire dimensions can be extracted. With `split`, we can get slightly more control, but extracting e.g. the inner

```
fun [[int]] inner([[int]] a) =
  let n = size(0,a) in
  let m = size(1,a) in
  let {_,a2}   = split(1,a) in
  let {rows,_} = split(n-2,a2) in
  map(fn [int] ([int] row) =>
        let {_, row2} =
          split(1,row) in
        let {res, _}  =
          split(m-2,row2) in
        res,
      rows)
```

(a) $\mathcal{L}_0$

```
def inner(a):
  a[1:-1, 1:-1]
```

(b) Numpy

Figure 46: Removing the outer elements of a 2-dimensional array

elements of a 2-dimensional array is an enormously clumsy affair, as illustrated on Figure 46a.

The primary reason for this is that `split` only slices the outer dimension. In other systems, such as the Python library Numpy [34], it is comparatively much simpler to simultaneously slice on every dimension of an array, as illustrated on Figure 46b.

As an array-oriented programming language, $\mathcal{L}_0$ should have similar convenient support for slicing arrays. It is not only practical when writing code, but the resulting slice expressions are much easier to analyse than a dense expression using `size` and `split`.

More radically, many operations in $\mathcal{L}_0$ are operationally just transformations of the index space of an underlying array. Transpositions, `reshape`, `replicate`, and even array indexing, merely provide different views of underlying data. Thus, perhaps the best solution would be to provide a language construct that can express this mapping directly. We can envision a syntax like the following:

```
fun [[int]] transpose([[int]] a) =
  arrange a as
    size (n,m) => (m,n) // Determine size of output array
    elem [i,j] => [j,i] // Map position (i,j) in output to position (j,i) in input
```

A crucial property is that every element of the output array maps directly to some element in the input array, which means that at compile time, we can remove the `arrange` intermediary and access the original array directly. This design is very similar to the delayed arrays of Repa [24], which represent arrays as a function from the index space to the value space. The built-in `replicate` could be reformulated as follows:

```
fun [[[int]]] replicate(int k, [[int]] r) =
  arrange r as
```

[1] Of course, in this contrived example, we could just use `let-with` rather than a separate function.

```
    size (n,m)   => (k,n,m)
    elem [i,j,p] => [j,p]
```

We can also express entirely novel transformations, such as a rearrangement that repeats every element of its input array twice:

```
fun [int] dupElem([int] a) =
  arrange a as
    size (n)     => (n)
    elem (2*i)   => [i]
    elem (2*i+1) => [i]
```

A large potential problem with a construct such as `arrange` is whether the compiler will be able to recognise "known" transformations. For example, we demonstrated in Chapter 8 that the fusion algorithm depends on being able to recognise and rewrite `transpose` expressions, which it must be able to do, even if they are formulated in terms of `arrange`. Hence, it would be important that every `arrange` can be reduced to a canonical form that uniquely represents the transformation it performs.

11.2.4 Software Engineering

The world already has plenty of papers and theses stuffed with long listings of Haskell code, and we have therefore tried to shy away from talking too much about the software architecture of the $\mathcal{L}_0$ compiler. While the overall code base is healthy and well-structured, there are still several instances of technical debt that should be paid off:

- While the compiler is nicely divided into discrete passes, the order in which said passes should be invoked is a bit unclear. As it stands, programs are passed through every pass several times, simply to ensure that they get optimised fully. This requires some bit of re-engineering, probably also involving changing some passes (particularly the fusion module) to be less sensitive as to the shape of the input program.

- For this thesis, $\mathcal{L}_0$ has been divided into an external and internal language. In the compiler, both of these are included in the same abstract syntax tree definition, with most passes either silently ignoring or loudly crashing if they encounter a construct that belongs to the external language. This creates undue complexity, and should be resolved by splitting the language more clearly, even if the cost is some code duplication (for example, we might need separate but very similar parsers).

- Somewhat related to the previous issue, the $\mathcal{L}_0$ syntax tree definition does not statically enforce normalisation. Again, compiler passes either ignore them or crash when an un-normalised term is encountered.

- Many optimisations depend on every variable in the program posessing a unique name. This property is ensured by tagging each input name with a unique integer, then passing around a counter that can be used to generate

fresh, globally unique integers. Unfortunately, many transformations (e.g. inlining) end up duplicating bits of code, which then have to be entirely renamed in order to preserve uniqueness. Furthermore, passing the counter around is cumbersome, even if packaged in a state monad. An alternative approach to handling name binding, based on de Bruijn indices [28], is being considered. Such an approach would allow us to get rid of the counter, while still being able to cheaply avoid unwanted name capture.

11.3 Conclusion

In this master's thesis, we have presented the design of a pure functional data-parallel language, with a design that enables both (i) a degree low-level imperative programming, as well as (ii) supporting high-level structural transformations such as loop fusion. The language contains a type system for in-place modification and aliasing of arrays and array slices that ensures referential transparency, which in turn supports equational reasoning.

Previous work on fusion has taken two main directions: Either fusion is performed aggressively, and the programmer is provided primitives to inhibit fusion, for example by forcing array to materialise, or fusion is performed via rewriting rules on the syntax tree. The latter approach relies tightly on the inliner engine, and its applicability is limited to the case when each fused array is consumed by one array combinator.

This thesis has presented a program-level, structural-analysis approach to fusion that handles the difficult case in which an array produced by a second-order array combinator (SOAC), such as `map`, is consumed by several other SOACs (if the SOAC producer-consumer dependency graph is reducible). This essentially allows fusion to operates across `zip`/`unzip`.

Furthermore, we have shown a compositional algebra for fusion that includes array combinators, such as `map`, `reduce`, `filter`, `scan`, and `redomap`, and other built-in functions that would otherwise hinder fusion applicability, such as `size`, `transpose`, and `reshape`. This algebra also includes transformations that in come cases allow fusion with `scan` as both consumer and producer.

Part IV

Closing Credits

Bibliography

[1] Aho, Alfred V. et al. *Compilers, Principles, Techniques, and Tools*. Pearson Addison Wesley, 2007. isbn: 0-321-49169-6.

[2] Allen, Randy and Ken Kennedy. *Optimizing Compilers for Modern Architectures*. Morgan Kaufmann, 2002, p. 790. isbn: 1-55860-286-0.

[3] Barendsen, Erik and Sjaak Smetsers. "Conventional and Uniqueness Typing in Graph Rewrite Systems". In: *Found. of Soft. Tech. and Theoretical Comp. Sci. (FSTTCS)*. Vol. 761. LNCS. 1993, pp. 41–51.

[4] Barendsen, Erik and Sjaak Smetsers. "Uniqueness Typing for Functional Languages with Graph Rewriting Semantics". In: *Mathematical Structures in Computer Science* 6.6 (1996), pp. 579–612.

[5] Bergstrom, Lars and John Reppy. "Nested Data-Parallelism on the GPU". In: *Procs. of Int. Conf. Funct. Prog. (ICFP)*. ACM. 2012, pp. 247–258.

[6] Bird, R. S. "An Introduction to the Theory of Lists". In: *NATO Inst. on Logic of Progr. and Calculi of Discrete Design*. 1987, pp. 5–42.

[7] Black, F. and M. Scholes. "The Pricing of Options and Corporate Liabilities". In: *The Journal of Political Economy* (1973), pp. 637–654.

[8] Blelloch, Guy. "Programming Parallel Algorithms". In: *Communications of the ACM (CACM)* 39.3 (1996), pp. 85–97.

[9] Blelloch, Guy E. "Scans as Primitive Parallel Operations". In: *Computers, IEEE Transactions* 38.11 (1989), pp. 1526–1538.

[10] Blume, William and Rudolf Eigenmann. "Symbolic range propagation". In: *Parallel Processing Symposium, 1995. Proceedings., 9th International*. IEEE. 1995, pp. 357–363.

[11] Blume, William and Rudolf Eigenmann. "The Range Test: A Dependence Test for Symbolic, Non-Linear Expressions". In: *Procs. Int. Conf. on Supercomp. (ICS)*. 1994, pp. 528–537.

[12] Chakravarty, Manuel M. T. et al. "Data Parallel Haskell: A Status Report". In: *Int. Work. on Decl. Aspects of Multicore Prog. (DAMP)*. 2007, pp. 10–18.

[13] Chakravarty, Manuel M.T. et al. "Accelerating Haskell Array Codes with Multicore GPUs". In: *Int. Work. on Declarative Aspects of Multicore Prog. (DAMP)*. 2011, pp. 3–14.

[14] Claessen, Koen, Mary Sheeran, and Bo Joel Svensson. "Expressive Array Constructs in an Embedded GPU Kernel Programming Language". In: *Work. on Decl. Aspects of Multicore Prog DAMP*. 2012, pp. 21–30.

[15] Crank, J. and P. Nicolson. "A practical method for numerical evaluation of solutions of partial differential equations of the heat-conduction type". English. In: *Advances in Computational Mathematics* 6.1 (1996), pp. 207–226. issn: 1019-7168. doi: `http://dx.doi.org/10.1007/BF02127704`. url: `http://dx.doi.org/10.1007/BF02127704`.

[16] Czarnecki, Krzysztof et al. "DSL implementation in MetaOCaml, Template Haskell, and C++". In: *Domain-Specific Program Generation*. Springer, 2004, pp. 51–72.

[17] Gill, Andrew, John Launchbury, and Simon L Peyton Jones. "A Short Cut to Deforestation". In: *Procs. of Int. Conf. on Functional Prog. Lang. and Computer Arch.* ACM. 1993, pp. 223–232.

[18] Grelck, Clemens and Sven-Bodo Scholz. "SAC - A Functional Array Language for Efficient Multi-Threaded Execution". In: *International Journal of Parallel Programming* 34.4 (2006), pp. 383–427.

[19] Hall, Mary W. et al. "Interprocedural Parallelization Analysis in SUIF". In: *Trans. on Prog. Lang. and Sys. (TOPLAS)* 27(4) (2005), pp. 662–731.

[20] Han, Tianyi David and Tarek S. Abdelrahman. "Reducing Branch Divergence in GPU Programs". In: *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*. GPGPU-4. Newport Beach, California: ACM, 2011, 3:1–3:8. isbn: 978-1-4503-0569-3. doi: http://dx.doi.org/10.1145/1964179.1964184. url: http://doi.acm.org/10.1145/1964179.1964184.

[21] Henriksen, Troels and Cosmin Eugen Oancea. "A T2 Graph-reduction Approach to Fusion". In: *Proceedings of the 2Nd ACM SIGPLAN Workshop on Functional High-performance Computing*. FHPC '13. Boston, Massachusetts, USA: ACM, 2013, pp. 47–58. isbn: 978-1-4503-2381-9. doi: http://dx.doi.org/10.1145/2502323.2502328. url: http://doi.acm.org/10.1145/2502323.2502328.

[22] Hoare, Graydon. *The Rust Programming Language*. June 2013. url: http://www.rust-lang.org/.

[23] Jones, Simon Peyton, Andrew Tolmach, and Tony Hoare. "Playing by the Rules: Rewriting as a Practical Optimisation Technique in GHC". In: *Haskell Workshop*. Vol. 1. 2001, pp. 203–233.

[24] Keller, Gabriele et al. "Regular, Shape-Polymorphic, Parallel Arrays in Haskell". In: *ACM Sigplan Notices* 45.9 (2010), pp. 261–272.

[25] Kohlbecker, Eugene et al. "Hygienic macro expansion". In: *Proceedings of the 1986 ACM conference on LISP and functional programming*. ACM. 1986, pp. 151–161.

[26] Kristensen, Mads RB et al. "Bohrium: Unmodified NumPy Code on CPU, GPU, and Cluster". In: ().

[27] Mainland, Geoffrey. "Why it's nice to be quoted: quasiquoting for haskell". In: *Proceedings of the ACM SIGPLAN workshop on Haskell workshop*. ACM. 2007, pp. 73–82.

[28] McBride, Conor and James McKinna. "Functional Pearl: I Am Not a Number–I Am a Free Variable". In: *Proceedings of the 2004 ACM SIGPLAN Workshop on Haskell*. Haskell '04. Snowbird, Utah, USA: ACM, 2004, pp. 1–9. isbn: 1-58113-850-4. doi: http://dx.doi.org/10.1145/1017472.1017477. url: http://doi.acm.org/10.1145/1017472.1017477.

[29] McDonell, Trevor L. et al. "Optimising Purely Functional GPU Programs". In: *Procs. of Int. Conf. Funct. Prog. (ICFP)*. 2013.

[30] Nvidia, CUDA. "Cublas library". In: *NVIDIA Corporation, Santa Clara, California* 15 (2008).

[31] Oancea, Cosmin E. and Lawrence Rauchwerger. "A Hybrid Approach to Proving Memory Reference Monotonicity". In: *Int. Lang. Comp. Par. Comp. (LCPC'11)*. Vol. 7146. LNCS. 2013, pp. 61–75.

[32] Oancea, Cosmin E. and Lawrence Rauchwerger. "Logical Inference Techniques for Loop Parallelization". In: *Procs. of Int. Conf. Prog. Lang. Design and Impl. (PLDI)*. 2012, pp. 509–520.

[33] Oancea, Cosmin et al. "Financial Software on GPUs: between Haskell and Fortran". In: *Funct. High-Perf. Comp. (FHPC'12)*. 2012.

[34] Oliphant, Travis E. *A Guide to NumPy*. Vol. 1. Trelgol Publishing USA, 2006.

[35] Sabry, Amr and Matthias Felleisen. "Reasoning About Programs in Continuation-passing Style." In: *SIGPLAN Lisp Pointers* V.1 (Jan. 1992), pp. 288–298. issn: 1045-3563. doi: http://dx.doi.org/10.1145/141478.141563. url: http://doi.acm.org/10.1145/141478.141563.

[36] Uffe, et al. “Almene Teoridannelser om generisk programmering af numeriske løsninger til sædvanlige og partielle differentialligninger, med specielt henblik på anvendelse af Standard Fortran 76, og dennes henvisninger til Turings artikel, ”Über die Wesen des Primtalalgorimus des kvadratischer Bubblesort anno 1943” - bilag F. Før, nu og i fremtiden!” In: *DIKURevy* (2007).

Artificial Benchmark Programs

P0

```
fun real horner (real x) =
   let {c1,c2,c3,c4,c5} =
     {0.31938153,-0.356563782,1.781477937,-1.821255978,1.330274429}
   in x * (c1 + x * (c2 + x * (c3 + x * (c4 + x * c5))))

fun real abs (real x) = if x < 0.0 then -x else x

fun real cnd0 (real d) =
   let k        = 1.0 / (1.0 + 0.2316419 * abs(d)) in
   let p        = horner(k) in
   let rsqrt2pi = 0.39894228040143267793994605993438 in
   rsqrt2pi * exp(-0.5*d*d) * p

fun real cnd (real d) =
   let c = cnd0(d)
   in if 0.0 < d then 1.0 - c else c

fun real go ({bool,real,real,real} x) =
   let {call, price, strike, years} = x in
   let r       = 0.08 in  // riskfree
   let v       = 0.30 in  // volatility
   let v_sqrtT = v * sqrt(years) in
   let d1      = (log (price / strike) + (r + 0.5 * v * v) * years) / v_sqrtT in
   let d2      = d1 - v_sqrtT in
   let cndD1   = cnd(d1) in
   let cndD2   = cnd(d2) in
   let x_expRT = strike * exp (-r * years) in
   if call then
     price * cndD1 - x_expRT * cndD2
   else
     x_expRT * (1.0 - cndD2) - price * (1.0 - cndD1)

fun [real] blackscholes ([{bool,real,real,real}] xs) =
   map (go, xs)
```

```
fun [real] main () =
  let days = 5*365 in
  let a = map(op+(1), iota(days)) in
  let a = map(toReal, a) in
  let a = map(fn {bool,real,real,real} (real x) =>
               {True, 58.0 + 4.0 * x / toReal(days), 65.0, x / 365.0},
             a) in
  blackscholes(a)
```

P0 – optimised

```
fun [real] main() =
  let {untuple_141} =
    mapT(fn {real} (int y_0) =>
           let val_1 = 1 + y_0 in
           let val_2 = toReal(val_1) in
           let bop_9 = val_2 / 365.0 in
           let bop_10 = 0.08 * bop_9 in
           let val_11 = -bop_10 in
           let tmp_call_12 = exp(val_11) in
           let x_expRT_13 = 65.0 * tmp_call_12 in
           let bop_14 = 0.125 * bop_9 in
           let tmp_call_15 = sqrt(bop_9) in
           let v_sqrtT_16 = 0.3 * tmp_call_15 in
           let bop_17 = 4.0 * val_2 in
           let bop_60 = bop_17 / 1825.0 in
           let bop_61 = 58.0 + bop_60 in
           let val_62 = bop_61 / 65.0 in
           let tmp_call_63 = log(val_62) in
           let bop_64 = tmp_call_63 + bop_14 in
           let d1_65 = bop_64 / v_sqrtT_16 in
           let bop_66 = d1_65 < 0.0 in
           let negate_67 = -d1_65 in
           let bop_68 = 0.5 * d1_65 in
           let bop_69 = bop_68 * d1_65 in
           let val_70 = -bop_69 in
           let tmp_call_75 = exp(val_70) in
           let bop_76 = 0.3989422804014327 * tmp_call_75 in
           let bop_77 = 0.0 < d1_65 in
           let d2_78 = d1_65 - v_sqrtT_16 in
           let bop_83 = d2_78 < 0.0 in
           let negate_84 = -d2_78 in
           let bop_85 = 0.5 * d2_78 in
           let bop_86 = bop_85 * d2_78 in
           let val_87 = -bop_86 in
           let tmp_call_88 = exp(val_87) in
           let bop_90 = 0.3989422804014327 * tmp_call_88 in
```

```
let bop_91 = 0.0 < d2_78 in
let tmp_bop_94 =
  if bop_83
  then negate_84
  else d2_78 in
let bop_96 = 0.2316419 * tmp_bop_94 in
let bop_97 = 1.0 + bop_96 in
let k_103 = 1.0 / bop_97 in
let bop_108 = k_103 * 1.330274429 in
let bop_109 = -1.821255978 + bop_108 in
let bop_110 = k_103 * bop_109 in
let bop_111 = 1.781477937 + bop_110 in
let bop_112 = k_103 * bop_111 in
let bop_113 = -0.356563782 + bop_112 in
let bop_114 = k_103 * bop_113 in
let bop_115 = 0.31938153 + bop_114 in
let p_116 = k_103 * bop_115 in
let c_117 = bop_90 * p_116 in
let bop_118 = 1.0 - c_117 in
let cndD2_119 =
  if bop_91
  then bop_118
  else c_117 in
let bop_120 = x_expRT_13 * cndD2_119 in
let tmp_bop_121 =
  if bop_66
  then negate_67
  else d1_65 in
let bop_124 = 0.2316419 * tmp_bop_121 in
let bop_125 = 1.0 + bop_124 in
let k_126 = 1.0 / bop_125 in
let bop_127 = k_126 * 1.330274429 in
let bop_128 = -1.821255978 + bop_127 in
let bop_129 = k_126 * bop_128 in
let bop_130 = 1.781477937 + bop_129 in
let bop_131 = k_126 * bop_130 in
let bop_132 = -0.356563782 + bop_131 in
let bop_133 = k_126 * bop_132 in
let bop_134 = 0.31938153 + bop_133 in
let p_135 = k_126 * bop_134 in
let c_136 = bop_76 * p_135 in
let bop_137 = 1.0 - c_136 in
let cndD1_138 =
  if bop_77
  then bop_137
  else c_136 in
let bop_139 = bop_61 * cndD1_138 in
```

```
          let bop_140 = bop_139 - bop_120 in
          {bop_140},
        iota(1825)) in
  untuple_141
```

P1

```
fun   int    redplus1( [int]  a) = reduce(op +, 0, a)
fun  [int]   redplus2([[int]] a) = map   (redplus1, a)

fun  [int]   mul1( [int]  a,  [int]  b) = map(op *, zip(a, b))
fun [[int]]  mul2([[int]] a, [[int]] b) = map(mul1, zip(a, b))

fun [[int]]  replin(int N, [int] a) = replicate(N, a)

fun [[int]] matmultFun([[int]] a, [[int]] b ) =
    let N   = size(0, a)                   in
    let br  = replicate( N, transpose(b) ) in
    let ar  = map      ( replin(N),    a ) in
    let abr = map  (mul2, zip(ar, br))     in
        map(redplus2, abr)

fun [[int]] main([[int]] x, [[int]] y) =
  matmultFun(x, y)
```

P1 – optimised

```
fun [[int]] main([[int]] x_0, [[int]] y_1) =
  let tmp_size_2 = size(1, x_0) in
  let tmp_size_3 = size(0, y_1) in
  let tmp_e_4 = tmp_size_2 = tmp_size_3 in
  let zip_assert_5 = assert(tmp_e_4) in
  let tmp_size_6 = size(1, y_1) in
  let N_13 = size(0, x_0) in
  let tmp_e_19 = N_13 = tmp_size_6 in
  let zip_assert_27 = assert(tmp_e_19) in
  // untuple_45 aliases x_0
  let {untuple_45} =
    mapT(fn {[int]} ([int] param_0_33) =>
           let {untuple_44} =
             <zip_assert_27>
             mapT(fn {int} ([int] arg_34) =>
                    let {untuple_43} =
                      <zip_assert_5>
                      redomapT(fn {int} (int x_35, int y_36) =>
                                 let val_37 = x_35 + y_36 in
                                 {val_37},
```

```
                                fn {int} (int x_38, int arg_39, int arg_40) =>
                                  let val_41 = arg_40 * arg_39 in
                                  let val_42 = x_38 + val_41 in
                                  {val_42},
                                {0}, arg_34, param_0_33) in
                    {untuple_43},
                  transpose(y_1)) in
          {untuple_44},
        x_0) in
  untuple_45
```

P2

```
fun int MIN(int a, int b) = if(a<b) then a else b

fun [int] min1([int] a, [int] b) = map(MIN, zip(a, b))


fun   int    redmin1( [int]  a) = reduce(MIN, 1200, a)
fun  [int]   redmin2([[int]] a) = map   (redmin1, a)

fun  [int]   plus1( [int]  a,  [int]  b) = map(op +, zip(a, b))
fun [[int]]  plus2([[int]] a, [[int]] b) = map(plus1, zip(a, b))

fun [[int]]  replin(int len, [int] a) = replicate(len, a)

fun [[int]] floydSbsFun(int N, [[int]] D ) =
    let D3  = replicate( N, transpose(D) ) in
    let D2  = map      ( replin(N),   D  ) in
    let abr = map(plus2, zip(D3, D2))       in
    let partial = map(redmin2, abr)        in
        map(min1, zip(partial, D) )

fun [[int]] main() =
    let arr = [[2,4,5], [1,1000,3], [3,7,1]] in
    floydSbsFun(3, arr)
```

P2 – optimised

```
fun [[int]] main() =
  let arr_0 = [[2, 4, 5],
               [1, 1000, 3],
               [3, 7, 1]] in
  // untuple_37 aliases arr_0
  let {untuple_37} =
    mapT(fn {[int]} ([int] param_0_1) =>
           let {untuple_36} =
```

```
            mapT(fn {int} (int arg_2, [int] arg_4) =>
                   let {untuple_33} =
                     redomapT(fn {int} (int param_0_5, int param_1_24) =>
                                let bop_25 = param_0_5 < param_1_24 in
                                let val_26 =
                                  if bop_25
                                  then param_0_5
                                  else param_1_24 in
                                {val_26},
                              fn {int} (int param_0_27, int arg_28,
                                        int arg_29) =>
                                let val_30 = arg_28 + arg_29 in
                                let bop_31 = param_0_27 < val_30 in
                                let val_32 =
                                  if bop_31
                                  then param_0_27
                                  else val_30 in
                                {val_32},
                              {1200}, arg_4, param_0_1) in
                   let bop_34 = untuple_33 < arg_2 in
                   let val_35 =
                     if bop_34
                     then untuple_33
                     else arg_2 in
                   {val_35},
                 param_0_1, transpose(arr_0)) in
           {untuple_36},
         arr_0) in
untuple_37
```