CHALMERS

A Parallel Intermediate Representation for Embedded Languages

Master of Science Thesis in the Programme
Computer Science – Algorithms, Languages and Logic

Ivar Lång

Department of Computer Science and Engineering
Chalmers University of Technology
Göteborg, Sweden 2013

A Parallel Intermediate Representation for Embedded Languages

Ivar Lång

Examiner: Mary Sheeran

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden July 2013

Abstract

This thesis presents a parallel intermediate representation for embedded languages called PIRE, and its incorporation into the Feldspar language. The original Feldspar backend translates the parallel loops of Feldspar to ordinary for loops, meaning that they are not actually parallel in the generated code. We create an alternate backend for the Feldspar project, where the parallel loops of Feldspar are translated as OpenCL kernels that run on the GPU. We show that we gain performance using our new backend for big input sizes compared to the original backend.

Keywords: Data-Parallelism, Feldspar, Intermediate Representation, OpenCL

Acknowledgements

First and foremost I'd like to extend my gratitude to my supervisor Anders Persson for discussions, insight, desk-borrowing, and the heroic late-night hacking efforts. Likewise I extend my gratitude to Mary Sheeran for discussions, advice, paper suggestions, and all things writing.

This thesis was sponsored by Ericsson AB. Thanks to the members of the Baseband Research group at Ericsson Lindholmen. You made me feel welcome. Thank you for making my stay most pleasant.

Thanks to Joel Svensson for advice on Obsidian and GPUs, Koen Claessen for providing me with code, Nikita Frolov and Josef Svenningsson for paper suggestions, and Emil Axelsson for pointing my work in the right direction, and for providing me with the Sklansky implementation I used in one of the case studies.

This thesis was essentially kickstarted with a workshop at Lindholmen, thanks to all of those who attended said workshop.

This work is part of the RAW FP Project, funded by the Swedish Foundation for Strategic Research (Stiftelsen för Strategisk Forskning).

Contents

Chapter 1

Introduction

As single-core processors are ceasing to scale, multi-core processors have over the last decade or so made their way into the mainstream. These kinds of processors provide programmers with a high degree of support for parallelism. While few would argue that having more computing power is a bad thing, it does put more burden on the programmer. High-level languages, particularly functional languages like Haskell (Haskell.org, 2011), attempt to relieve some of this burden.

An embedded domain-specific language (EDSL) is a particular kind of domain-specific language (DSL). Such a language exists only as a library within another language, the *host* language. One popular host language is Haskell. Throughout this thesis we will see numerous examples of embedded languages using Haskell as host.

When performance is crucial, EDSLs often end up compiled rather than interpreted. The compiler for the EDSL, like traditional compilers, uses intermediate representations (IRs) to represent a program internally in an abstract fashion. It is intermediate in the sense that it represents the program while it goes from one representation (the input string) to another (the output string). The IR is an important piece of the puzzle that is compilation, for it aids in analysis and transformation of the source code.

Recent trends in EDSLs (e.g. Obsidian (Svensson, 2011), Nikola (Mainland and Morrisett, 2010) and Accelerate (Chakravarty et al., 2011)) indicate that the graphical processing unit (GPU) is hardware worth exploring. It provides a massively parallel machinery that is appropriate for data-parallel computations. With the introduction of programming languages such as CUDA (Nvidia, 2013) and OpenCL (Khronos Group, 2013), programmers can use GPUs for general purpose (GPGPU) programming.

Other EDSLs than those for GPGPU programming could also benefit from increased parallelism support. For instance, incorporating support for parallelism into Feldspar (Axelsson et al., 2010), an EDSL for digital signal processing, has been a longtime goal.

A survey of the field of current (compiled) EDSLs indicates that many of the IRs used are tailored to the domain of the language and influenced by the target language. As an example, languages for GPU programming often have their IR designed to fit the CUDA/OpenCL programming model. Moreover, many EDSLs attempt to exploit the opportunities for parallelism provided by current hardware, therefore making parallel intermediate representations an interesting area to explore.

In response to the lack of generality in existing parallel IRs, this thesis presents a more general kind of parallel IR. It can be targeted by several kinds of EDSLs and in turn it can target different kinds of backends.

1.1 Background

The need for special-purpose programming languages aimed at particular domains is ever-present. We call such languages domain-specific languages (DSLs). They are characterised by their functionality in a particular domain, and are often unsuitable for other domains. Some common examples of DSLs include OpenGL for 3D graphics, LaTeX for typesetting, HTML for document markup, YACC for parsing, and SQL for database queries.

DSLs implemented as libraries in existing (general-purpose) languages we call embedded domain-specific languages (EDSLs). The benefits of this approach to DSL development are many (Kosar et al., 2008; Hudak, 1998), but primarily we can bypass the need to develop front-end components like the lexer and the parser. One language that is particularly well-suited for acting as host language is the functional language Haskell. The features of Haskell, such as laziness and higher-order functions (Hughes, 1990), polymorphism and type classes lend themselves nicely for this application (Hudak, 1998).

To give a flavour of the idea of EDSLs, here follows some examples of domains and languages therein. For (functional) image manipulation and synthesis there is Pan (Elliott, 2003), for software testing there is the property-based random testing language QuickCheck (Claessen and Hughes, 2000) and for circuit design there are Lava (Bjesse et al., 1998) and Wired (Axelsson et al., 2005). For GPGPU programming, there are several examples including Nikola (Mainland and Morrisett, 2010) and Obsidian (Svensson, 2011). For DSP applications there is the aforementioned Feldspar (Axelsson et al., 2010). There are even languages for expressing music notation; Haskore (Hudak et al., 1996) is such a language.

EDSL programs can easily be interpreted in the host language. While this is often enough, some applications require performance or predictability better than what can be achieved by interpretation. Alternatively, applications may contain expressions that are not feasible or possible to compute in the host language. In such cases, we turn to compilation. By developing a code-generating backend we can target a language other than the host language and hope to gain benefits. For instance, the Feldspar backend (Dévai et al., 2010) currently does exactly this with C99 as target language.

1.1.1 Parallelism

This section presents some of the fundamental concepts of parallelism. First and foremost, we emphasise the difference between concurrency and parallelism. While concurrency is concerned with a lot of things such as threads, synchronisation and critical sections, parallelism is concerned with one thing only - to make things go faster by adding more computing elements (e.g. CPU cores).

Arguably the two most common forms of parallelism are task-parallelism and data-parallelism (Subhlok et al., 1993). Task-parallelism is parallelisation of tasks (of arbitrary size or complexity) over available computing elements. There are many issues involved with task-parallelism. Because of the arbitrary complexity of tasks, scheduling becomes an issue. We don't always know how long a task takes to complete. Moreover, tasks may share data or rely on each other for control flow, which further complicates things.

Data-parallelism is where we exploit the independence of data-set members (Subhlok et al., 1993; Hillis and Steele, 1986; Blelloch et al., 1993b). Data is distributed over available computing elements, and we achieve parallelism by applying operations in parallel to each member. Typically, this is done in a SIMD (single instruction multiple data (Flynn, 1972)) fashion.

We distinguish between *flat* and *nested* data-parallelism. An intuitive way of thinking about the distinction is to imagine the parallelism in levels. Flat can only provide us with one level of parallelism (see Figure 1.1), whereas nested can provide several (see Figure 1.2). For nested data-parallelism, operations that we apply in parallel to the data can themselves spark addi-

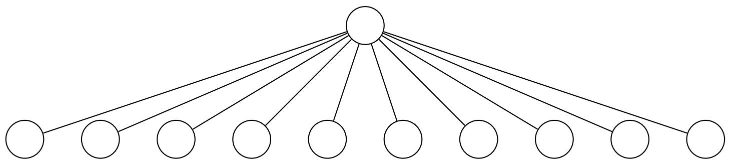

Figure 1.1: Flat data-parallelism.

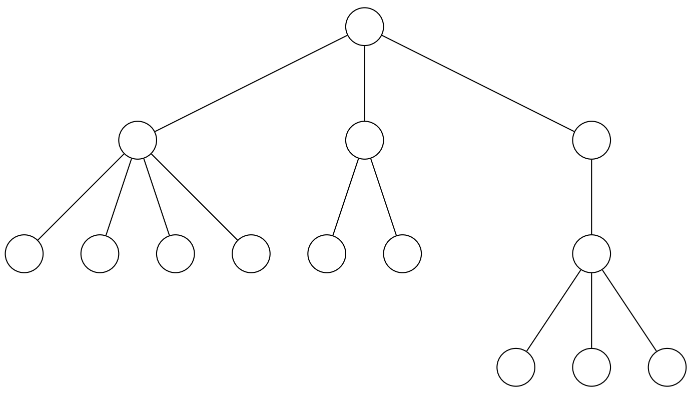
Figure 1.2: Unbalanced, nested data-parallelism.

tional parallelism. This is not the case for the flat kind. Languages support different nesting depth. For instance, the library Repa (Keller et al., 2010) for Haskell supports only flat (depth zero) data-parallelism, the language NESL (Blelloch et al., 1993b) supports nesting of arbitrary depth and EmbArBB (Svensson and Sheeran, 2012) (an EDSL for Intel's Array Building Blocks) supports one level of nesting.

The CUDA/OpenCL model is classified as SPMD (single program multiple data) (Pharr and Mark, 2012). In SPMD, we run instances of the same program simultaneously by spreading them over multiple (available) computing elements. The instances all work on different points in the data (that is, they run with different input).

We briefly note that there is also another quite common form of parallelism known as pipeline parallelism. We consider pipeline parallelism to be out of scope for this thesis. However, the interested reader is encouraged to look at (Trinder et al., 1998, sec. 4.4) and (Gordon et al., 2006).

In Haskell there are several ways of introducing parallelism, and the following is a selection of approaches. Trinder et al. (1998) presents a way of introducing parallelism via strategies, which builds upon existing parallel primitives of Haskell (`par` and `pseq`). The approach was later refined by Marlow et al. (2010). Strategies can be used to achieve both task-parallelism and data-parallelism (among other forms). The Par Monad (Marlow et al., 2011) gives us a way of modelling data-flow networks and a way of expressing task-parallelism. The Repa library (Keller et al., 2010) allows us to express (non-nested) data-parallel computations over regular (i.e. dense), multi-dimensional arrays. It is related to Data-Parallel Haskell (DPH) (Jones et al., 2008), which is an extension to the Glasgow Haskell Compiler (GHC) for nested data-parallelism over irregular arrays.

One might wonder why there is need for two such (seemingly) similar ways of achieving parallelism. The answer is given by (Keller et al., 2010, p. 11):

> Both provide a way of writing high-performance parallel programs but DPH supports irregular, arbitrarily nested parallelism which requires it to sacrifice performance when it comes to purely regular computations.

The Repa library is meant to complement DPH, and the intention is to integrate Repa into DPH, thus providing support for both regular and irregular arrays.

Adding to this, the first language to *fully* support nested data-parallelism was NESL by Blelloch et al. (1993b). It is a functional language for specifying data-parallel programs. One typically uses a set-like notation, similar to that of list-comprehensions in Haskell, to apply functions to a sequence of elements. For instance, the NESL-program (taken from Blelloch et al. (1993b)) that negates all elements in a sequence that are less than four can be written as:

```
{ negate(a): a in [3, -4, -9, 5] | a < 4 }
```

The program returns the sequence `[-3, 4, 9]`.

1.2 Objective

This thesis aims to design and implement a parallel intermediate representation for EDSLs. The intermediate representation will encompass a code-generating backend for C, where parallel loops are offloaded to the GPU using OpenCL.

Moreover, the IR will be incorporated into the compiled EDSL Feldspar. The parallel loop construction of Feldspar will be a focal point throughout this thesis. In the current Feldspar compiler, the parallel loop compiles as an ordinary C for loop, meaning there is no actual parallelism in the generated code. We will incorporate our parallel IR into the Feldspar compiler and use the OpenCL backend to achieve parallelism.

1.2.1 Scope

The parallel IR aims to target single-core, multi-core and GPU systems, but only one at a time. Heterogeneous systems we consider future work. In addition, there are several kinds of parallelism that are of interest. This thesis will only focus on data-parallelism. Task-parallelism and pipeline parallelism, however interesting, we also consider future work. Furthermore, time does not allow for a full-scale implementation of Feldspar. Instead, a slightly reduced version of Feldspar that excludes monadic constructs and mutable arrays will be used.

1.3 Thesis Outline

This thesis is structured as follows. In Chapter 2 we give an overview of EDSLs, and give a case study (in Haskell) that explores the difference between different kinds of embeddings. Chapter 3 introduces Feldspar, an EDSL for digital signal processing. Chapter 4 presents the parallel IR for embedded languages that we call PIRE. Its implementation and backend are described, with a focus on how parallel loops are compiled. Chapter 5 describes how PIRE replaces the current low-level IR of the Feldspar compiler. Chapters 6 and 7 discuss the evaluation of the code-generating capabilities of PIRE. Chapter 8 presents related work, and Chapter 9 aims to discuss the results from the evaluation. The thesis is concluded with Chapter 10.

Chapter 2

Embedded Domain-Specific Languages

This chapter presents the concept of embedded domain-specific languages (EDSLs). For readers not familiar with embeddings, this chapter will serve as an introduction to the area. Section 2.1 attempts to make clear the differences between different kinds of embeddings. In Section 2.2 we further explore the concept of compiling embedded languages. The chapter ends with a description of Feldspar, a language of particular interest for this thesis, in Section 3.

Expanding on what was introduced in the background (Section 1.1), it is generally the view that EDSLs come in two flavours, shallowly embedded and deeply embedded. One can, however, imagine a combination of the two. Axelsson and Svenningson (2012) presents a technique that combines both flavours. This technique is employed in Feldspar, where shallow embeddings are built on top of a deeply embedded core-language (Axelsson and Sheeran, 2012).

A shallow embedding lies closer to the semantics while its counterpart lies further away. In a shallow embedding, we use functions and values in the host language to represent functions and values in the domain language. Typically, a deep embedding is needed when we need to specify computations that cannot easily be expressed in the host language. From a deep embedding, we can begin to target a language other than the host language. This is something we will see throughout this thesis, and it is a fundamental building-block when compiling EDSLs.

According to Hudak (1998), embedding a DSL is nothing new (and certainly not something limited to just Haskell) and has been done for a long time in the language Lisp, where macros have been used to develop embedded languages.

Common to EDSLs of all types is that they share the concepts of constructor-functions, combinators and interpretation functions (known also as run functions). It is fairly straightforward - constructor-functions introduce values into the domain, combinators modify values and interpretation functions take an expression in the domain language and interpret it to a value in the host language.

Generally, a shallow embedding is represented by a single interpretation function, making it difficult to add new interpretations. In a deep embedding, this is less of a problem since interpretation functions are cheap to add. The drawback of deep embeddings is that they are less flexible when adding new constructs to the language. The following case study on a language for vectors aims to make clear these differences.

2.1 Case Study: a Vector Language

This section aims to highlight differences between shallow and deep embeddings by presenting two variations of a small language for vectors. For the sake of conciseness, the language implements only a few common functions on vectors. These are vector append (+++), `length`, `head`, indexing `(!)`, `take` and `drop`. Additionally, the `fromList` and `toList` are the main constructor and interpretation function, respectively.

The following code implements the language in a *deep* fashion. We assume that standard prelude imports as `P` where needed. For instance, the difference between `head` and `P.head` is that `head` is a function in the embedded language (working on the Vector type) while `P.head` is the standard head function of Haskell (working on the regular Haskell list type).

```
data Vector a where
  Append   :: Vector a → Vector a → Vector a
  FromList :: [a] → Vector a

(+++) :: Vector a → Vector a → Vector a
(+++) = Append

infixr 5 +++

fromList :: [a] → Vector a
fromList = FromList

toList :: Vector a → [a]
toList (Append xs ys) = toList xs ++ toList ys
toList (FromList xs)  = xs
```

```
length :: Vector a → Int
length (Append xs ys) = length xs + length ys
length (FromList xs)  = P.length xs

head :: Vector a → a
head (Append xs _) = head xs
head (FromList xs) = P.head xs

(!) :: Vector a → Int → a
(!) (Append xs ys) i | i < xsLen  = xs ! i
                     | otherwise  = ys ! (i-xsLen)
                    where xsLen = length xs
(!) (FromList xs) i = xs !! i

infixr 5 !

take :: Int → Vector a → Vector a
take n (Append xs ys) | n > length xs =
                         take (length xs) xs +++
                         take (n - length xs) ys
                      | otherwise     = take n xs
take n (FromList xs)  = FromList (P.take n xs)

drop :: Int → Vector a → Vector a
drop n (Append xs ys) | n ≤ 0         = xs +++ ys
                      | n ≤ length xs = drop n xs +++ ys
                      | otherwise = drop (n-length xs) ys
drop n (FromList xs)  = FromList (P.drop n xs)
```

An important thing to note is that each constructor in the data type has a corresponding constructor-function. For instance `Append` is introduced by the function `+++`.

Furthermore, a function that works on the data type has to mention all of the constructors. It now becomes clear to us why this approach can be very tedious. Imagine adding another constructor — for instance one that expresses `cons`. Every function has to be modified to accommodate the new constructor.

As a side-note, the problem of adding cases to data types without recompiling, while maintaining type safety, is known as the Expression Problem (coined by Wadler (1998)). For a recent solution to this problem, the interested reader is encouraged to look at the work by Axelsson (2012). It is currently used in Feldspar to increase modularity, in the form of the library Syntactic. The library also supports embedding of monadic constructs (Persson et al., 2012).

We now turn to the *shallow* approach. The following code implements the same language.

```
data Vector a = Vec { vecLength :: Int
                    , runVec :: Int → a
                    }

(+++) :: Vector a → Vector a → Vector a
(Vec l1 ixf1) +++ (Vec l2 ixf2) = Vec (l1 + l2)
    (λi → if i ≥ l1 then ixf2 (i-l1) else ixf1 i)

fromList :: [a] → Vector a
fromList xs = Vec (P.length xs) (λx → xs !! x)

toList :: Vector a → [a]
toList (Vec l ixf) = map ixf [0..l-1]

length :: Vector a → Int
length = vecLength

head :: Vector a → a
head xs = xs ! 0

(!) :: Vector a → Int → a
(!) = runVec

take :: Int → Vector a → Vector a
take n (Vec l ixf) = Vec (min l n) ixf

drop :: Int → Vector a → Vector a
drop n (Vec l ixf) = Vec (max (l-n) 0) (λi → ixf (i+n))
```

In this implementation, a vector is represented by a length and a function from indices to values. The interface remains the same, but the implementation is radically different. The functions are smaller, and arguably simpler. It seems vector libraries indeed benefit from being embedded shallowly rather than deeply.

We conclude this case study with the following observation. When working with deep embeddings, most work is done in the interpretation functions, while constructor-functions and combinators are basically free. In the case of shallow embeddings, the situation is reversed; most of the work is done in constructor-functions and combinators while the interpretation-functions often are derived from the data type.

2.2 Compilers and Embedded Languages

Traditional compilers are made up of many components that together form these very complex pieces of software. Some of these components are called intermediate representations, and are intended to aid in the analysis, optimisation and transformation of programs. Embedded compilers are often simpler, since they use already existing features of the host language.

Broadly speaking, there are two different types of IRs – graph-like representations and linear representations. These can of course come in various flavours. We can for instance imagine representing programs as strings, graphs (Girkar and Polychronopoulos, 1994), trees and in special-purpose intermediate languages like three-address code and abstract machine instructions (Aho et al., 2006; Demange, 2012).

Moreover, we are not limited to just one form of representation, as different compilation phases may have to scrutinise the program in different ways. Indeed, as noted by Aho et al. (2006), abstract syntax trees (ASTs) are often used in early phases of compilation for syntax analysis while abstract machine instructions may be a more appropriate representation in later phases. For later-phase IRs Aho et al. point out two important properties; the representation should be easy to produce, and it should be easily translated to the target machine. An example of a multi-IR compiler is the Feldspar compiler (Axelsson et al., 2010). It uses the purely functional core language early on, and then translates it to abstract imperative code, which is straightforward to translate to the target machine.

Early work on embedded compilers include the circuit-design language Lava (Bjesse et al., 1998). Lava produces symbolic descriptions of circuits that can be used to interface with, for instance, external theorem provers. While this might not be a general-purpose language, the idea is very much the same as in other embedded compilers. Leijen and Meijer (1999) also presents an early embedded compiler in Haskell. The EDSL introduced is one for generating SQL queries called Haskell/DB. Furthermore, Kamin (1996) uses SML/NJ as host language to embed languages producing C++. Kamin gives four examples; a top-down parser generator language, a geometric region server language, a message specification language and a language for pretty-printing.

More recent literature includes Elliott et al. (2003), which present techniques for creating optimising embedded compilers. Furthermore, many current EDSLs employ compilers, for instance Nikola (Mainland and Morrisett, 2010), Obsidian (Svensson, 2011; Claessen et al., 2008), Feldspar (Axelsson and Sheeran, 2012; Axelsson et al., 2010, 2011; Dévai et al., 2010) and Accelerate (Chakravarty et al., 2011).

Chapter 3

The Language Feldspar

Feldspar (Axelsson et al., 2010) is an EDSL in Haskell for digital signal processing (DSP) and high-performance numerical computations. DSP is a domain where algorithms traditionally are implemented in low level, sequential, C code. This is an error-prone and costly approach which offers very limited portability, due to the use of hardware-specific intrinsics. Moreover, given the parallel architectures of today, the Feldspar project questions the aptness of C as the language of choice for DSP applications.

An aim of Feldspar is to raise the level of abstraction. This means that algorithms, primarily over vectors, can be described in a concise manner. Compared to the traditional approach of C programming, the idea of the algorithm becomes clearer since it is shrouded in less boilerplate code. The high-level descriptions of algorithms are currently resulting in individual C-functions (Axelsson and Sheeran, 2012).

The language uses a deeply embedded, purely functional, core language on top of which a collection of shallowly embedded libraries are built, as illustrated in Figure 3.1 (Axelsson and Sheeran, 2012; Axelsson and Svenningson, 2012). The most prominent of these is the Feldspar vector library. There are other libraries, but we leave them out as they do not concern this work.

Feldspar is split up into a language component describing the frontend, and a compiler component describing the backend. We refer to these as *feldspar-language* and *feldspar-compiler* respectively. The work in this thesis uses a reduced version of feldspar-language, where monadic constructs (Persson et al., 2012) and mutable arrays are excluded.

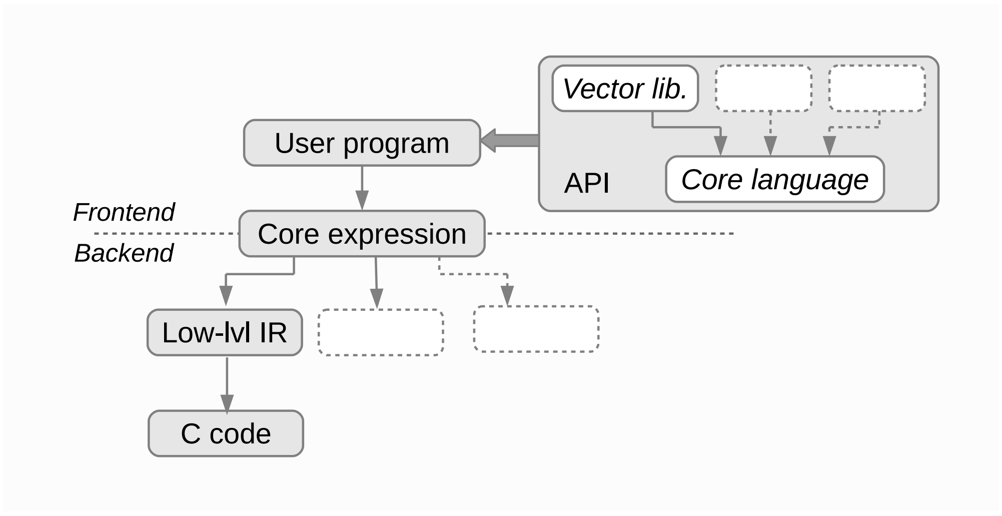

Figure 3.1: The Feldspar architecture.

3.1 The Core Language

The components feldspar-language and feldspar-compiler are essentially tied together by the core language - a purely functional intermediate representation. Core expressions (ASTs in the core language) constructed by the frontend can be passed to the compiler for compilation.

A program in the core language has type `Data a`. For instance, the type signature of the equality operation is:

```
(==) :: Eq a ⇒ Data a → Data a → Data a
```

In the core language there are two types of arrays, sequential and parallel. A parallel core array is created with the function `parallel`, which looks as follows (type constraint omitted):

```
parallel :: Data Length → (Data Index → Data a) → Data [a]
```

This is a fairly common way of representing data-parallel arrays. Obsidian calls this kind of array a *pull* array, in contrast to a push array (Claessen et al., 2012). Feldspar has high-level push *vectors* instead of push arrays. The push vectors of the Vector library are translated as mutable arrays in the core language. Mutable arrays require the `Mutable` module – a module excluded from the version of Feldspar we are working with in this thesis (see Section 1.2.1). It follows that we don't support push vectors.

Using `parallel` we can encode the program that returns the first n even numbers:

```
evens :: Data Index → Data [Index]
evens n = parallel n (*2)
```

Naturally, there are functions we cannot express with just parallel arrays. Therefore we require the existence of sequential arrays, where we rely on a state being passed along. Since we are concerned with parallelism in this thesis, sequential arrays are omitted.

In addition to parallel loops, Feldspar also supports sequential loops. The for loop is straightforward: it takes a length, an initial state value, and a function that transforms the state. The state after the last iteration becomes the result of the for loop. We consider the type of `forLoop`:

```
forLoop :: Data Length
        → a
        → (Data Index → a → a)
        → a
```

Using `forLoop` we can write (the slightly contrived) program that adds 10 to its input:

```
addTen :: Data Index → Data Index
addTen a = forLoop 10 a $ λ_ st → st + 1
```

The function `addTen` runs for ten iterations, and the initial value of the state is the input parameter of the function. We ignore the first argument of the state-transforming function (as indicated by the underscore), and add 1 to the state in each iteration.

3.2 The Vector Library

On top of the core language, there are numerous high-level libraries implemented as shallow embeddings. The benefit of having the libraries as shallow embeddings is that it allows for experimentation and augmentation without any modification to the (deeply embedded) core.

Vectors in Feldspar are called *symbolic* vectors, since they do not necessarily result in actual arrays in the target language (Axelsson and Sheeran, 2012). The Vector is implemented using the following type:

```
data Vector a = Empty
              | Indexed
                  { segmentLength :: Data Length
                  , segmentIndex  :: Data Index → a
                  , continuation  :: Vector a
                  }
```

A vector can either be Empty or non-empty (Indexed). A non-empty vector has a length (the sum of all its segments' lengths), an index-to-value function and possibly also a continuation vector describing the next segment. The library provides vector-variants of many of the common functions on Haskell lists, such and `append`, `drop`, `fold` and `map`.

3.3 Case-Study: Dot Product

In this section we conclude the chapter on Feldspar with a case study on dot product. We will revisit the dot product problem in Section 6, albeit with slightly different, more parallel, implementation. Below we give an implementation which is:

```
dotProd :: Vector1 Word32
        → Vector1 Word32
        → Data Word32
dotProd xs ys = sum (zipWith (*) xs ys)
```

We see that it is a call to `zipWith` using multiplication and the two input vectors, followed by a call to `sum`. The function `sum` is defined in the Feldspar vector library and is just a shorthand for `fold` using the addition operator. We would perhaps expect to generate two loops from this code – one for `zipWith` and one for `sum`. But as we will see, Feldspar will fuse the calls into a single loop. The fusion takes place in the shallow embedding of the vector library, and superfluous arrays will be removed before the creation of the core expression takes place.

When we've loaded Feldspar into GHCI, we can call `icompile` in order to quickly output a function named `test` with the functionality of the program we provide. Below is the generated C-code from the `dotProd` program:

```
void test(struct array * v0,
          struct array * v1,
          uint32_t * out)
{
  uint32_t len0;
  uint32_t v3;

  len0 = min(getLength(v0), getLength(v1));
  *out = 0;
  for (uint32_t v2 = 0; v2 < len0; v2 += 1)
  {
    v3 = (*out +
           (at(uint32_t,v0,v2) *
             at(uint32_t,v1,v2)));
    *out = v3;
  }
}
```

Arrays are represented as a struct `array` that is defined in the Feldspar C files that come with *feldspar-compiler*. We use the `at` macro in the C code to access the data buffer of the `array` struct and type cast the result.

The two input vectors are translated as pointers to arrays, and the output pointer is a scalar, as we would expect when seeing the type signature of `dotProd`. The two loops we would expect to see have been fused to one, as noted earlier. The length of the loop is initialised to the length of the shortest vector (indicated by `len0`). This behaviour follows from the use of `zipWith`, and is mimicking the behaviour of the original Haskell function. The loop does one multiplication and one accumulation into the output parameter per iteration.

Chapter 4

PIRE

This chapter aims to present the idea and design of PIRE – a Parallel Intermediate Representation for Embedded languages[1]. PIRE encompasses a code-generating backend for C with parallel loops using OpenCL kernels. Kernels are functions that are run on the GPU. They produce output using only their input parameters. The kernels generated by PIRE are currently generating *global* kernels (kernels that only use global memory).

Implemented in Haskell, PIRE is intended to be a low-level parallel IR used in compiled embedded languages. It is low-level in the sense that it is intended to be the last in a chain of representations before actual code generation. This means that very few traditional optimisations are done in PIRE – if optimisations are desired, they should take place higher up the chain of IRs.

PIRE is an imperative IR as opposed to, for instance, a functional one. Imperative code lies closer to the machine and thus gives a better opportunities for producing specialised code (still future work) and code for special hardware (e.g. GPUs). Producing sequential C code from a PIRE-AST is essentially done by pretty-printing, assuming that one translates parallel loops to ordinary loops.

Following from the thesis scope (Section 1.2.1), the kind of parallelism supported by PIRE is data-parallelism. There is, for instance, no support for task parallelism; threads, synchronisation and spawn-primitives are left out completely.

Throughout the remainder of this thesis we will use the terms *host* and *kernel*. The C code that runs on the CPU and has access to the ordinary memory is what we in OpenCL terms call the host code. A kernel is code that runs

[1]The repository containing PIRE can be found at `https://github.com/rCEx/PIRE`.

on the device (e.g. GPU) and has access only to specific device memory. Device memory can be both global and local, but this thesis explores only kernels that use global memory. The host program is responsible for both transferring the memory needed by the kernel to the device, and for reading the memory back from the device.

Incorporating PIRE into an embedded language (in Haskell) consists of three steps: (1) identify a mapping between its constructs and the constructs of PIRE, (2) construct a PIRE AST from the surface language expressions using the mapping, and (3) pass the PIRE AST to the PIRE code generator.

4.1 Implementation

PIRE is made up of the two main data types `Expr` and `Program`. The type `Expr` represents things that can go on the right-hand side of assignments, or appear simply as expressions within statements. Things of type `Program` will form the ASTs that are later passed to the code-generator of PIRE.

The `Program` type is built in continuation-passing style (Claessen, 1999). This means that when `Program` trees are constructed, they are written into a function that later will create a `Program`. One benefit of this is that surface languages (such as Feldspar) do not have to care about generating fresh names. Our hope is that this will simplify the porting process, by removing tedious state-carrying (which is typically required to generate fresh names in a functional setting). All actual naming is done within the PIRE backend.

We start by considering the `Expr` data type and related type synonyms:

```
data Expr where
  Num    :: Int → Expr
  Index  :: Memory → Name → [Expr] → Expr
  Call   :: Expr → [Expr] → Expr
  Cond   :: Expr → Expr → Expr → Expr
  BinOp  :: BOp → Expr
  UnOp   :: UOp → Expr

type Name  = String
type Size  = Expr
type Dim   = [Size]
```

In PIRE we are currently limited to the integer type, and the `Num` construct introduces an integer literal. The `Index` constructor allow us to introduce a variable name that can be indexed into. Giving it the empty list:

```
Index Host "x" []
```

gives us just a regular, non-indexed, variable. For this purpose, we can also use the function `var :: Name → Expr`. We note also the `Memory` field, that describes in which kind of memory the variable is placed. Either memory is placed in the host program or in the global memory of the GPU device. Future work would include allowing variables to also be placed in shared device memory. The `Memory` type is defined as:

```
data Memory = Host
            | DevGlobal
```

`Call` is the construct for function call – the list describes the parameters. `Cond` is for inline conditional expressions (like the `?:` operator in C). For the binary and unary operators, we have two additional data types `BOp` and `UOp`. As we consider them quite self-explanatory, we leave them out. For completeness they are given in Appendix A.

We now consider the Program data type:

```
data Program a where
  Skip      :: Program a
  Assign    :: Expr → [Expr] → Expr → Program a
  Statement :: Expr → Program a
  (:>>)     :: Program a → Program a → Program a
  If        :: Expr → Program a → Program a →
               Program a
  For       :: Expr → Expr →
               (Expr → Program a) → Program a
  Par       :: Expr → Expr →
               (Expr → Program a) → Program a
  Alloc     :: Type →
               (Name → Name →
                (Memory → Dim → Program a) →
                Program a) →
               Program a
  Decl      :: Type → (Name → Program a) → Program a

  BasicProc :: Program a → Program a
  OutParam  :: Type → (Name → Program a) → Program a
  InParam   :: Type → (Name → Program a) → Program a
```

Many of the constructs of this data type are quite similar, and thus, self-explanatory once the reader knows how a similar construct works. However, we recognise that some do require a more thorough explanation. The `Skip` construct is simply the empty statement and we can use it when doing AST analysis in order to remove nodes that are unnecessary. Some of the constructs of `Program` also have corresponding smart constructors that, among other things, remove `Skip` nodes from the AST.

The Assign construct is a fundamental one, since it allows us to assign expressions to left-hand side locations (Section 4.1.1).

```
Assign (var "x") [Num 4] (Num 5)
```

assigns 5 to the expression x[4]. Each element in the list parameter corresponds to an indexing into the name x. The program

```
x[4][y] = 5;
```

can thus be written as:

```
Assign (var "x") [Num 4, var "y"] (Num 5)
```

The constructor :>> simply means program sequencing, which means doing

```
Assign (var "x") [Num 4] (Num 5) :>>
  Assign (var "x") [Num 5] (Num 6)
```

yields:

```
x[4] = 5;
x[5] = 6;
```

The loops supported by PIRE come in two flavours – parallel and non-parallel. They are both of the for loop kind and allow us to specify a bound on the number of iterations. The regular for loop is introduced by the For construct and the parallel loop by the Par construct. They are very similar and differ only in execution environment. The parallel loop compiles to kernel code (code that runs on the GPU) and host code (in our case C code) which calls the kernel via the OpenCL interface. The regular for loop runs on the host without GPU involvement.

When we need to introduce variable names in a program, we use either Decl or Alloc. Decl is intended to be a stack-based declaration, while Alloc dynamically allocates memory that can be used for arrays. An allocation always takes place in the host code, but depending on the memory parameter of the allocation function the allocation will either be intended for use in the host code, or for being transferred to the device.

The Decl construct we consider to be straight-forward, but we realise that Alloc requires further explanation. Let us first consider its type again:

```
Alloc ::
Type → (Name → Name →
          (Memory → Dim → Program a) → Program a)
      → Program a
```

Intuitively, when given a type and a funny-looking continuation it produces a Program. When breaking down the continuation we notice three parts, two

`Names` and another continuation function which we will refer to as the *allocation function*. The first name is the actual name of the variable and the second name is the corresponding size descriptor. Typically the second name is involved in the resulting allocation. The actual allocation is not performed until the allocation function is called. This has one major benefit compared to outputting the allocation as soon as the `Alloc` node is seen in the AST. Namely, it allows us to gather more information about the allocation before outputting code. The information in question is the `Dim` parameter. The drawback is that it is easy to forget to call the function at all, and thus end up with no memory allocation in the final code. A way of forcing the user to call the allocation function (possibly only once) we consider important future work.

Depending on the `Memory` parameter of the allocation function, memory is allocated either in the host program via `malloc`, or allocated as a device memory buffer in the host program. The device memory buffer can later be written to and transferred to the device when a kernel call is performed.

Encoding a simple program that allocates and initialises an array of size 10 can be done in the following fashion:

```
Alloc (TPointer TInt) $ λv vc af →
  af Host [Num 10] :>>
  (For (Num 0) (var vc) $ λe → Assign (var v) [e] e)
```

The generated code is what we expect:

```
int mem0c;
mem0c = 10;
int* mem0 = (int*) malloc(sizeof(int) * mem0c);
for(int j = 0; j < mem0c; j++) {
  mem0[j] = j;
}
```

If we now instead remove the call to the allocation function `af`:

```
Alloc (TPointer TInt) $ λv vc af →
  (For (Num 0) (var vc) $ λe → Assign (var v) [e] e)
```

The generated code is broken since the allocation is missing:

```
int mem0c;
for(int j = 0; j < mem0c; j++) {
  mem0[j] = j;
}
```

At times, it might not even be obvious that the allocation *is* missing. This is especially true for big programs where information easily gets obfuscated. Since `af` can be passed around inside the code generator of the EDSL it may be called from anywhere and thus place the allocation at an unexpected place (that is still valid in the final code). Locating such a missing allocation can be tricky.

Three of the constructs have to do with procedures and their parameters. Without these constructs, we would only be able to write constant programs. `BasicProc` introduces an empty procedure without any parameters. `OutParam` adds an out parameter, and `InParam` adds an in parameter to the parameter list of the procedure. The program that takes one in parameter (of integer type) and returns it as the out parameter can be written:

```
BasicProc $
 OutParam TInt $ λout →
   InParam TInt $ λn →
     Assign (var out) [] (var n)
```

If we apply the function `showProg . gen` to the above program we get:

```
void f0(int out1, int arg2) {
  out1 = arg2;
}
```

4.1.1 Locations

To model a location, the left-hand side of an assignment, we can use the type-synonym `Loc a b`. It is defined as `a → Program b`, and gives us a simple way of expressing locations. The type parameter `a` is the type of the right-hand side (typically `Expr`), and `b` is what `Program` is parameterised on.

Simply put, a location is a partially applied `Assign` constructor. In order to have some use for the type `Loc`, PIRE comes with a collection of functions to introduce locations. To introduce a location that is a simple variable, we use the function

```
loc :: Name → Loc Expr a
loc v = λx → Assign (var v) [] x
```

Here, `loc` is a function that, when given a name, produces a function that expects an expression. If we proceed to give this function an expression, the result is an assignment where `v` is assigned `x`.

We can also create a location that is an index into an array. We use `locArray` for this:

```
locArray :: Name → Index → Loc Expr a
locArray v i = λx → Assign (var v) [i] x
```

In order to allow for easy array copying, PIRE also provides us with a `memcpy` location:

```
memcpy :: Expr → Size → Type → Loc Expr a
memcpy dst s t = ...
```

It writes data to the destination `dst` from the expression that we eventually feed the location. It is inspired by the `memcpy` function of C, but has a twist. It can transfer memory not just within the host program, but also between host program and device, and *locally* on the device. Local transfer means that memory does not have to be read back to the host program when performing a sequence of kernel calls that all require the output of the previous call as input. We simply leave the memory buffers sitting on the device and use them as input parameters to the next kernel call.

4.1.2 Motivations and Limitations

One of the early decisions that had to be made was whether or not to support explicit arrays (i.e. push- and pull-arrays (Claessen et al., 2012)). PIRE does not currently do this, due to the focus of making it a low-level, concise, IR. However, this doesn't mean that there wouldn't be benefits of having them. Translating a language like Obsidian, that relies heavily on these arrays, might be more straightforward if they were to also exist in the intermediate representation.

A big limitation currently in PIRE is the small set of types. PIRE supports only integers (type `int` in C) and pointers to integers. While being an active decision in order to keep implementation time down, it does make it difficult to express things that require floating point types or similar.

Another limitation has to do with the local work item size of OpenCL. This number is set when calling a kernel on the device. It is, however, dependant on both problem and device type. The local work item size is constant in the backend and intended to be tweaked by hand. In OpenCL terms, local item size is the number of threads within a work group. However, since we are not using any local memory in kernels, the work group concept does not concern us much. Lastly, we note that it is generally the case that the *global* work item size (the total input size) has to be a multiple of the local work item size.

4.2 Parallel Loops Using OpenCL

Parallel loops are introduced with the `Par` construct of the `Program` data type. The resulting host program carries a lot of boilerplate-style code for each kernel call, and thus we will elide some of the details throughout this chapter. We will, however, make clear when doing so.

We recall the program for vector initialisation that we saw earlier:

```
Alloc (TPointer TInt) $ λv vc af →
  af Host [Num 10] :>>
  (For (Num 0) (var vc) $ λe → Assign (var v) [e] e)
```

This program is easily parallelized since it has no form of state that gets updated between iterations. We substitute `Par` for `For` and change `Host` to `DevGlobal`:

```
Alloc (TPointer TInt) $ λv vc af →
  af DevGlobal [Num 10] :>>
  (Par (Num 0) (var vc) $ λe → Assign (var v) [e] e)
```

Running it in the PIRE compiler yields the following kernel:

```
__kernel void k1( __global int* mem0 ) {
  int tid = get_global_id(0);
  mem0[tid] = tid;
}
```

We recognise that it looks very similar to an ordinary function definition in any C-like language. Instead of using a regular loop variable, we use the *global* thread identifier (`tid` in the code) of the device threads. In this case, global means that the work group identifier is already factored into the thread identifier, thus making the thread globally identifiable. The number of threads that runs on the device is decided by the `global_item_size` parameter in the host code (see below).

We use the keyword `__global` on kernel parameters to denote the use of global memory (i.e. memory that is accessible by all threads). This is the only kind of memory currently used by kernels generated by PIRE. Global memory is much slower than local memory (memory within to a work group), but has the (very minor) benefit that we don't have to split the problem into sufficiently small chunks (i.e. work groups) to match local memory restrictions. Work group sizes are decided by the `local_item_size` parameter in the host code. Ideally, work group size and use of local memory should be derived automatically. We consider this to be an important piece of future work, and discuss it further in Section 9.1.

Moreover, all parameters have the possibility of acting as both input and output parameters. Through AST analysis it is automatically decided which of the parameters are read back to the host program at the end of a kernel call. This is currently done in a naïve way – all parameters that appear in the left-hand side of `Assignment` nodes are read back.

With some details elided, the host code that calls the kernel looks as follows:

```
int mem0c;
mem0c = 10;
cl_mem mem0 = clCreateBuffer(context,
                             CL_MEM_READ_WRITE,
                             (mem0c * sizeof(int)),
                             NULL,
                             NULL);
clSetKernelArg(k1, 0, sizeof(cl_mem), &mem0);
size_t global_item_size = mem0c;
size_t local_item_size = 1;
clEnqueueNDRangeKernel(command_queue,
                       k1,
                       1,
                       NULL,
                       &global_item_size,
                       &local_item_size,
                       0,
                       NULL,
                       NULL);
```

As we see, a kernel call carries a fair bit of boilerplate code. What is of interest to us is the call to `clSetKernelArg`, where the input data is mapped to a kernel parameter. Also worth noting is that when we change the `Host` argument of the allocation function `af` to `DevGlobal`, `mem0` becomes an OpenCL memory buffer instead of a regular variable. The actual kernel invocation is done via
`clEnqueueNDRangeKernel`.

Chapter 5

Connecting Feldspar

This chapter describes the incorporation of PIRE into the Feldspar compiler[1]. We substitute PIRE for the original low-level Feldspar IR known as abstract imperative code (the AST type which is used to output C code). The Feldspar compiler will instead produce PIRE ASTs, which we pass to the PIRE backend for OpenCL and C code-generation. Figure 5.1 illustrates the updated Feldspar architecture after the incorporation of PIRE. The result is an alternate backend for the Feldspar language.

Due to the size of the *feldspar-compiler* component, we cannot cover its whole implementation here. Instead, we try and present the key points and make clear what separates our new feldspar compiler from the original. We examine in detail the compilation of some specific core language symbols such as parallel loops.

The original Feldspar compiler is highly modular and makes good use of type classes. We see no reason to change this and therefore decided to only modify this existing framework where necessary. Some modules are of particular interest to us and we outline their role in the compilation process.

FromCore The module FromCore is a low-level entry-point of the compiler. Traditionally we use the function `compile` to compile a Feldspar program, but the FromCore module is where the actual translation of core-expressions starts. It defines the function `compileProgTop`, which compiles all top-level lambda binders and top-level let-expressions. The top-level lambdas correspond to the input parameters of the program. Top-level let-expressions, on the other hand, are the result of invariant code motion.

[1]The repository of the *feldspar-compiler* fork used in this thesis can be found at `https://github.com/rCEx/feldspar-compiler/`. The (subset) fork of *feldspar-language* can be found at `https://github.com/rCEx/feldspar-lang-small`.

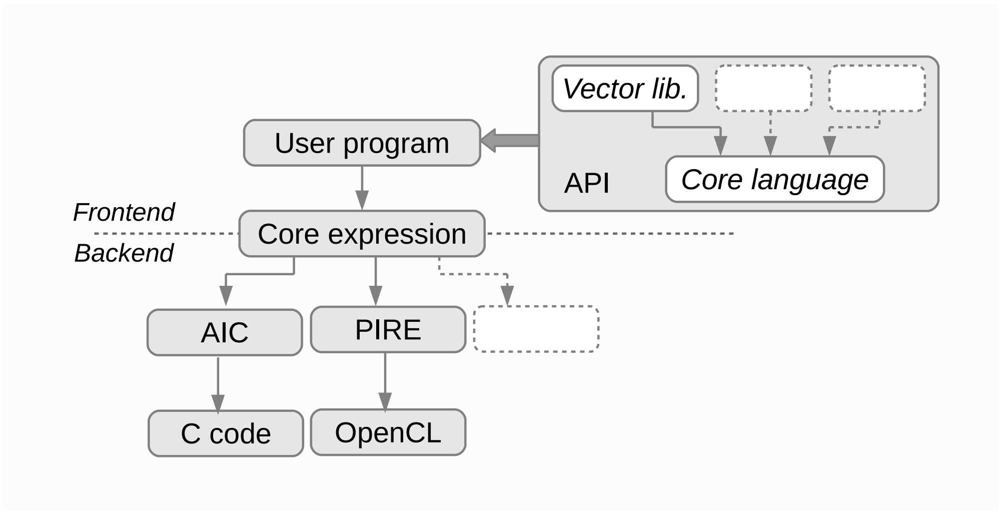

Figure 5.1: The Feldspar architecture after the incorporation of PIRE. PIRE replaces the abstract imperative code (AIC) and the result is a new backend.

Interpretation The Interpretation module defines the type classes for compilation as well as helper functions for various tasks. The base cases for compilation are covered by this module; the more specific cases are covered by a number of other modules. In order to support the continuation-passing style of PIRE, we define a new function `compileProgBasic` for `Programs` in the `Compile` class:

```
compileProgBasic
    :: (Expr, Loc Expr ())
    → Maybe Name
    → AllocFun
    → sub a
    → Info (DenResult a)
    → Args (AST (Decor Info dom)) a
    → CodeWriter ()
compileProgBasic name sub = ...

type CodeWriter a = Alias → Program a
type Alias        = M.Map VarId Expr
type AllocFun     = Maybe (Dim → Program ())
```

We also modify the existing function `compileExprSym` for compilation of expressions to look as follows:

```
compileExprSym
    :: sub a
    → Info (DenResult a)
    → Args (AST (Decor Info dom)) a
    → Alias → [Expr]
compileExprSym = ...
```

To clarify the above, there is one function (`compileProgBasic`) for compilation to the `Program` type of PIRE, and one function (`compileExpr`) for compilation to the `Expr` type of PIRE. The idea of the compiler is to define these two functions for each symbol in the core language (where applicable). It is simply a case of pattern matching.

Some symbols do not make sense to compile as expressions, while some are not sensible to compile as Programs. Generally, simple core-language symbols can be returned as expressions while more complex ones require something more. `compileProgBasic` can result in intermediate memory allocation while `compileExpr` cannot.

To give an example, it may be natural to compile the length of an array as an expression. This can be done by returning the name of the array size parameter. On the other hand, a for loop is tricky to return as a simple expression. Therefore we need to provide a name to which we can write the computation of the for loop. To get access to such a name, we use `compileProgBasic` instead of `compileExpr`.

Array The `Parallel` symbol of the core language is defined in the Array module. Since the main topic of this thesis is parallelism, we consider in detail how parallel loops are compiled to PIRE. Below is the code for the `Parallel` symbol. We've inserted the type signature for `compileProgBasic` to help the reader.

```
compileProgBasic
    :: (Expr, Loc Expr ())
    → Maybe Name
    → AllocFun
    → sub a
    → Info (DenResult a)
    → Args (AST (Decor Info dom)) a
    → CodeWriter ()
compileProgBasic name
                 namec
                 af
                 (C' Parallel)
                 info
                 (len :* (lam :$ ixf) :* Nil)
                 m
```

```
  | Just (SubConstr2 (Lambda v)) ← prjLambda lam
    =  let bound = (head $ compileExpr len m)
           (Index name' _) = fst name
       in maybe Skip (λf → f [bound]) af
      .>> par (Num 0) bound (λe →
           compileProgWithName
              (fst name, locArray ( name') e)
              Nothing
              Nothing
              ixf
              (M.insert v e m))
```

The function `compileProgWithName` is a high-level wrapper for `compileProgBasic` that hides some work related to Syntactic.

5.1 Case-Study: Dot Product Revisited and Expanded

In Section 3.3 we showed how the original Feldspar compiler generates code for an implementation of dot product. In this section we instead show how the code generated from the new Feldspar backend with PIRE will look. We first recall the Feldspar implementation:

```
dotProd' :: Vector1 Word32
         → Vector1 Word32
         → Data Word32
dotProd' xs ys = sum (zipWith (*) xs ys)
```

This will result in code that is quite funny-looking, since we're converting all input parameters to OpenCL memory buffer objects without checking that we actually need them. We thus omit the generated code. In this case we are actually hurt by vector fusion. The `zipWith` should be compiled as a kernel call, but because of fusion we are left with an ordinary for loop that does all the work. This is why the OpenCL memory buffers that we generate from arguments are useless. Instead, we force the resulting vector of `zipWith` to manifest (to explicitly appear in the generated code) via the use of `force`. The following program does what we want:

```
dotProd'' :: Vector1 Word32
          → Vector1 Word32
          → Data Word32
dotProd'' xs ys = sum zs
  where zs = force $ (zipWith (*) xs ys)
```

We are going to take our dot product implementation one step further by implementing a parallel reduction that will compile to a kernel call. We end up with the implementation of dot product that we use in the case study in Section 6.2 (modulo types). We call the parallel reduction `parFold` and implement it as follows:

```
parFold :: (Syntax a, Num a)
        ⇒ (a → a → a)
        → Vector a
        → Vector a
parFold f xs = forLoop (log2 (length xs)) xs $
  λi' acc → let i = i' + 1 in indexed (length acc) $
    λj → condition
    (j `mod` (2^i) == 0)
    (f (acc ! j)
       (acc ! (j+(2^(i-1)))))
    0
```

We are using a for loop (that runs for log_2 times of the length of the input vector) with an inner, parallel loop. The parallel loop follows from the use of `indexed`, which creates a new vector in parallel. In the inner loop we have to start with the loop variable `i'` set to 1, hence the let binding that binds `i` to `i' + 1`. In the case that the conditional is false, we give the arbitrarily chosen value `0`. It does not matter, as it just acts as a padding – the value will never be accessed again. Lastly we note that the `parFold` function only works if the length of the vector is a power of 2.

Our final, parallel, dot product implementation in Feldspar looks as follows (we don't have to call `force` on `zipWith` any longer):

```
dotProd :: Vector1 Word32
        → Vector1 Word32
        → Data Word32
dotProd xs ys = head $ parFold (+) (zipWith (*) xs ys)
```

Chapter 6

Case-Studies

We evaluate the Feldspar version in which we incorporated PIRE, described in Chapter 5, using three example programs and perform measurements against reference implementations. The examples are scan (Section 6.1), dot product (Section 6.2) and bitonic sort (Section 6.3). For each case-study, we implement it in Feldspar and run measurements using both the original Feldspar backend and PIRE. We also compare these to hand-coded OpenCL versions.

6.1 Scan

Scan (prefix sum) is a well-known and fundamental operation that, for an associative operator $\circ$, produces from the input $a_0, a_1 \ldots a_{n-1}$ the output $b_i = a_0 \circ a_1 \circ \ldots \circ a_i$ for $0 \leq i < n$ (Blelloch, 1990). This case-study aims to see how well a parallel scan implementation in Feldspar will perform in PIRE. The parallel scan used here is implemented using Sklansky construction (Sheeran, 2011).

The Sklansky scan implementation is shown below, where the function `sklansky` is our entry-point. In our case study we use addition as the associative operator (argument `f` in the function `sklansky`).

```
sklansky :: Syntax a
         ⇒ (a → a → a)
         → Vector a
         → Vector a
sklansky f a = forLoop (log2 (length a)) a (step f)
```

```
step :: Syntax a
     ⇒ (a → a → a)
     → Level
     → Vector a
     → Vector a
step f l as = indexed (length as) $ λi →
    cond l i
        ? f (as ! leftIx l i) (as!i)
        $ (as!i)
```

The `sklansky` program is a nested loop, where the outer, sequential, loop is the result of `forLoop`. The inner, parallel, loop is the result of `step`. The parallel loop results in the kernel shown in Section 6.1.1.

The helper functions and type aliases are defined below:

```
type Level    = Data Index
type Position = Data Index

leftIx :: Level → Position → Position
leftIx l p = ((p .>>. l) .<<. l) - 1

cond :: Level → Position → Data Bool
cond l p = testBit (p .>>. l) 0

log2 :: Data Length → Data Length
log2 a = bitSize a - 1 - bitScan a
```

6.1.1 Generated Code by PIRE

The PIRE compiler generates a function, complete with in and out parameters. We omit prologue and epilogue code related to OpenCL features and memory management. From the code below, we also elide some arguments from the OpenCL functions that are set to `NULL` or are otherwise uninteresting.

The for loop runs log_2(arg1c) times, which is what we expect after seeing the Feldspar implementation above. It is made concrete in the generated code by calling the function `bitScan_fun_int32_t`, which is defined in the Feldspar header files. After the outer for loop is finished, we read back the result from the device to the out parameter of the function.

We note one particular weakness of the code below. The creation of the `mem4` memory buffer is completely unnecessary. It gets initialised to the contents of the `mem2` memory buffer without any modification whatsoever. A much neater way of writing the same program would be to solely use `mem2` throughout.

The problem is related to the way the *state* of the for loop is currently compiled. Identifying that a memory buffer already exists for the state would solve this problem. We note that this is a general problem of our implementation and not just something isolated to scan. In the case of scan, however, we only pay the cost of *one* extra copying. Having nested for loops could result in significantly more expensive code.

```
void f0(int* arg1, int arg1c, int** out3) {
  int mem2c;
  mem2c = arg1c;
  cl_mem mem2 = clCreateBuffer(context,
                               CL_MEM_READ_WRITE,
                               (mem2c * sizeof(int)));
  clEnqueueWriteBuffer(command_queue,
                       mem2,
                       (mem2c * sizeof(int)),
                       arg1);
  int mem4c;
  mem4c = mem2c;
  cl_mem mem4 = clCreateBuffer(context,
                               CL_MEM_READ_WRITE,
                               (mem4c * sizeof(int)));
  clEnqueueCopyBuffer(command_queue,
                      mem2,
                      mem4,
                      (mem4c * sizeof(int)));
  for(int n = 0;
      n < (31 - bitScan_fun_int32_t(mem2c));
      n++) {
    clSetKernelArg(k6, 0, sizeof(int), &n);
    clSetKernelArg(k6, 1, sizeof(cl_mem), &mem4);
    size_t global_item_size = mem4c;
    size_t local_item_size = 1024;
    clEnqueueNDRangeKernel(command_queue,
                           k6,
                           &global_item_size,
                           &local_item_size);
  }
  clEnqueueReadBuffer(command_queue,
                      mem4,
                      (mem4c * sizeof(int)),
                      (*out3));
  clReleaseMemObject(mem4);
  clReleaseMemObject(mem2);
}
```

Scan Kernel

```
__kernel void k6( int n, __global int* mem4 ) {
  int tid = get_global_id(0);
  int mem7;
  mem7 = (tid >> n);
  int mem8;
  mem8 = mem4[tid];
  mem4[tid] = testBit_fun_int32_t(mem7,0) ?
                (mem4[((mem7 << n) - 1)] + mem8) : mem8;
}
```

6.2 Dot Product

Dot product (scalar product) is a well-known operation that takes two vectors of equal length and reduces them to a single scalar value. The mathematical definition is $a \cdot b = \sum_{i=1}^{n} a_i b_i$ where a and b are vectors of length n. A simple solution in Haskell can be written as:

```
dp :: Num a ⇒ [a] → [a] → a
dp a b = sum $ zipWith (*) a b
```

The Feldspar definition is the very same, modulo types.

```
dp' :: Vector1 Word64 → Vector1 Word64 → Data Word64
dp' a b = sum $ zipWith (*) a b
```

However, the one used in this case study is the following one that uses a parallel fold. Since it returns a vector, we are interested only in the first element.

```
dotProd :: Vector1 Word64
        → Vector1 Word64
        → Vector1 Word64
dotProd xs ys = parFold (+) (zipWith (*) xs ys)
```

The `parFold` function has the drawback of only being applicable to vectors of length 2^n for some n. It is defined as follows:

```
parFold :: (Syntax a, Num a)
        ⇒ (a → a → a)
        → Vector a
        → Vector a
parFold f xs = forLoop (log2 (length xs)) xs $
  λi' acc → let i = i' + 1 in indexed (length acc) $
    λj → condition
    (j `mod` (2^i) == 0)
    (f (acc ! j)
       (acc ! (j+(2^(i-1)))))
    0
```

We note that the 0 that is given in the last `condition` statement is necessary, but the value does not actually matter. It is there simply for padding purposes in order to keep the vector the same length throughout iterations.

6.2.1 Generated Code by PIRE

The code generated by PIRE for the dot product case study is presented in this section. The two kernels `k8` and `k12` are presented at the end of the section. These correspond to the two parallel loops that we would expect to find in the program. One follows from the use of `zipWith` and the other follows from the use of `indexed` in the `parFold` function.

```
void f0(int* arg1,
        int arg1c,
        int* arg3,
        int arg3c,
        int** out6) {
  int mem2c;
  mem2c = arg1c;
  cl_mem mem2 = clCreateBuffer(context,
                               CL_MEM_READ_WRITE,
                               (mem2c * sizeof(int)));
  clEnqueueWriteBuffer(command_queue,
                       mem2,
                       CL_TRUE,
                       (mem2c * sizeof(int)),
                       arg1);
  int mem4c;
  mem4c = arg3c;
  cl_mem mem4 = clCreateBuffer(context,
                               CL_MEM_READ_WRITE,
                               (mem4c * sizeof(int)));
  clEnqueueWriteBuffer(command_queue,
                       mem4,
                       CL_TRUE,
                       (mem4c * sizeof(int)),
                       arg3);
  int mem5;
  mem5 = min(mem2c,mem4c);
  int mem7c;
  mem7c = mem5;
  cl_mem mem7 = clCreateBuffer(context,
                               CL_MEM_READ_WRITE,
                               (mem7c * sizeof(int)));
  clSetKernelArg(k8, 0, sizeof(cl_mem), &mem7);
  clSetKernelArg(k8, 1, sizeof(cl_mem), &mem2);
  clSetKernelArg(k8, 2, sizeof(cl_mem), &mem4);
  size_t global_item_size = mem5;
  size_t local_item_size = 1024;
  clEnqueueNDRangeKernel(command_queue,
                         k8,
                         &global_item_size,
                         &local_item_size);
  for(int r = 0;
      r < (31 - bitScan_fun_int32_t(mem5) - 1);
      r++) {
    int mem10;
    mem10 = pow(2,(r + 1));
    int mem11;
    mem11 = pow(2,r);
    clSetKernelArg(k12, 0, sizeof(cl_mem), &mem7);
    clSetKernelArg(k12, 1, sizeof(int), &mem10);
    clSetKernelArg(k12, 2, sizeof(int), &mem11);
    global_item_size = mem7c;
    local_item_size = 1024;
    clEnqueueNDRangeKernel(command_queue,
                           k12,
                           &global_item_size,
                           &local_item_size);
  }
  clEnqueueReadBuffer(command_queue,
                      mem7,
                      (mem7c * sizeof(int)),
                      (*out6));
  clReleaseMemObject(mem7);
  clReleaseMemObject(mem4);
  clReleaseMemObject(mem2);
}
```

Dot Product Kernels

```
__kernel void k8(__global int* mem7,
                 __global int* mem2,
                 __global int* mem4) {
  int tid = get_global_id(0);
  mem7[tid] = (mem2[tid] * mem4[tid]);
}
__kernel void k12(__global int* mem7,
                  int mem10,
                  int mem11) {
  int tid = get_global_id(0);
  int mem13;
  mem13 = mem7[tid];
  mem7[tid] = ((tid % mem10) == 0) ?
                (mem13 + mem7[(tid + mem11)]) : mem13;
}
```

6.3 Bitonic Sort

Bitonic sort is a sorting algorithm which lends itself nicely for parallelisation. We construct sorting networks that are made of comparators, and we sort by swapping elements. The resulting sorting network will consist of $\mathcal{O}(nlog^2n)$ comparators, where n is the number of elements to sort. The algortihm we employ is a variation of the original, as described by Claessen et al. (2012). We give the Feldspar implementation of this variation below.

```
flipLSBsTo :: Bits a ⇒ Data Index → Data a → Data a
flipLSBsTo i = (‘xor‘ oneBits (i+1))

vee :: Syntax a
    ⇒ (a → a → a)
    → (a → a → a)
    → Data Index → Vector a → Vector a
vee f g s v = indexed (length v) ixf
  where
    ixf i = condition (testBit i s) (g a b) (f a b)
      where
        a = v ! i
        b = v ! flipLSBsTo s i

dee :: Syntax a
    ⇒ (a → a → a)
    → (a → a → a)
    → Data Index
    → Vector a
    → Vector a
dee f g s v = indexed (length v) ixf
  where
    ixf i = condition (testBit i s) (g a b) (f a b)
      where
        a = v ! i
        b = v ! (i ‘xor‘ bit s)

tmerge :: (Type a, Ord a)
       ⇒ Data Index
       → Vector1 a
       → Vector1 a
tmerge n v = share (vee min max (n-1) v) $ λw →
              forLoop (n-1) w $ λi →
                                    dee min max (n-(i+2))

tsort :: (Type a, Ord a)
      ⇒ Data Index
      → Vector1 a
      → Vector1 a
tsort n v = forLoop n v (λi w → tmerge (i+1) w)
```

6.3.1 Generated Code by PIRE

```
void f0(int arg1, int* arg2, int arg2c, int** out4) {
  int mem3c;
  mem3c = arg2c;
  cl_mem mem3 = clCreateBuffer(context,
                               CL_MEM_READ_WRITE,
                               (mem3c * sizeof(int)));
  clEnqueueWriteBuffer(command_queue,
                       mem3,
                       CL_TRUE,(
                       mem3c * sizeof(int)),
                       arg2);
  int mem5c;
  mem5c = mem3c;
  cl_mem mem5 = clCreateBuffer(context,
                               CL_MEM_READ_WRITE,
                               (mem5c * sizeof(int)));
  clEnqueueCopyBuffer(command_queue,
                      mem3,
                      mem5,
                      (mem5c * sizeof(int)));
  for(int o = 0; o < arg1; o++) {
    int mem7;
    mem7 = (~(4294967295 << (o + 1)));
    int mem8;
    mem8 = (o + 1);
    int mem9c;
    mem9c = mem5c;
    cl_mem mem9 = clCreateBuffer(context,
                                 CL_MEM_READ_WRITE,
                                 (mem9c * sizeof(int)));
    clSetKernelArg(k10, 0, sizeof(cl_mem), &mem5);
    clSetKernelArg(k10, 1, sizeof(int), &mem7);
    clSetKernelArg(k10, 2, sizeof(cl_mem), &mem9);
    clSetKernelArg(k10, 3, sizeof(int), &o);
    size_t global_item_size = mem5c;
    size_t local_item_size = 1024;
    clEnqueueNDRangeKernel(command_queue,
                           k10,
                           &global_item_size,
                           &local_item_size);
    clEnqueueCopyBuffer(command_queue,
                        mem9,
                        mem5,
                        (mem5c * sizeof(int)));
    for(int v = 0; v < o; v++) {
      int mem14;
      mem14 = (mem8 - (v + 2));
      int mem15;
      mem15 = (1 << mem14);
      clSetKernelArg(k16, 0, sizeof(cl_mem), &mem5);
      clSetKernelArg(k16, 1, sizeof(int), &mem15);
      clSetKernelArg(k16, 2, sizeof(int), &mem14);
      global_item_size = mem5c;
      local_item_size = 1024;
      clEnqueueNDRangeKernel(command_queue,
                             k16,
                             &global_item_size,
                             &local_item_size);
    }
    clReleaseMemObject(mem9);
  }
  clEnqueueReadBuffer(command_queue,
                      mem5,
                      CL_TRUE,(mem5c * sizeof(int)),
                      (*out4));
  clReleaseMemObject(mem5);
  clReleaseMemObject(mem3);
}
```

Bitonic Sort Kernels

```
__kernel void k10(__global int* mem5,
                  int mem7,
                  __global int* mem9,
                  int o) {
  int tid = get_global_id(0);
  int mem11;
  mem11 = mem5[tid];
  int mem12;
  mem12 = mem5[(tid ^ mem7)];
  mem9[tid] = testBit_fun_int32_t(tid,o) ?
                max(mem11,mem12) : min(mem11,mem12);
}

__kernel void k16(__global int* mem5,
                  int mem15,
                  int mem14) {
  int tid = get_global_id(0);
  int mem17;
  mem17 = mem5[tid];
  int mem18;
  mem18 = mem5[(tid ^ mem15)];
  mem5[tid] = testBit_fun_int32_t(tid,mem14) ?
                max(mem17,mem18) : min(mem17,mem18);
}
```

6.4 Reference Implementations

The reference implementations for dot product, parallel scan and bitonic sort that are used as comparison are taken from the NVIDIA SDK that can be found at `http://developer.download.nvidia.com/compute/cuda/4_2/rel/sdk/website/OpenCL/html/samples.html`

Chapter 7

Evaluation

This chapter describes the results of running the case studies described in Chapter 6. Throughout this chapter we will refer to the original Feldspar compiler as *Feldspar*, our new Feldspar compiler using PIRE as *PIRE*, and the library code from the NVIDIA SDK as *reference*.

7.1 Test Setup

We are generating code from the Feldspar programs using both the original Feldspar compiler and the PIRE compiler. In addition to these two, we are also comparing with OpenCL code from the NVIDIA SDK. Each program was run 10 times with two uncounted warm-up iterations. The average running times are what is presented. The programs were measured using input vectors of length 2^n where $10 \leq n \leq 23$. The vectors were initialised using the repeating sequence $(0, 1, 2, 3)$.

The experiments were performed on an Amazon EC2 GPU instance, with 2 $\times$ Intel Xeon X5770 quad-core CPUs at 2.93 GHz, 22.5 GiB of Memory, and one (of the EC2's two) NVIDIA Tesla M2050 (448 cores @ 1150 MHz). The OpenCL programs were compiled using `nvcc -O3 -std=c99`. The C programs (from the original Feldspar backend) were compiled using `gcc -O3 -std=c99`.

The reference code for Scan and Bitonic Sort outputs running times for specific input sizes, hence the small number of data points. We chose not to modify the reference code to output more data, since such modifications would be too great and thus destroy the purpose of having reference code to begin with. We also note that the NVIDIA SDK does not output any measurements for its dot product implementation.

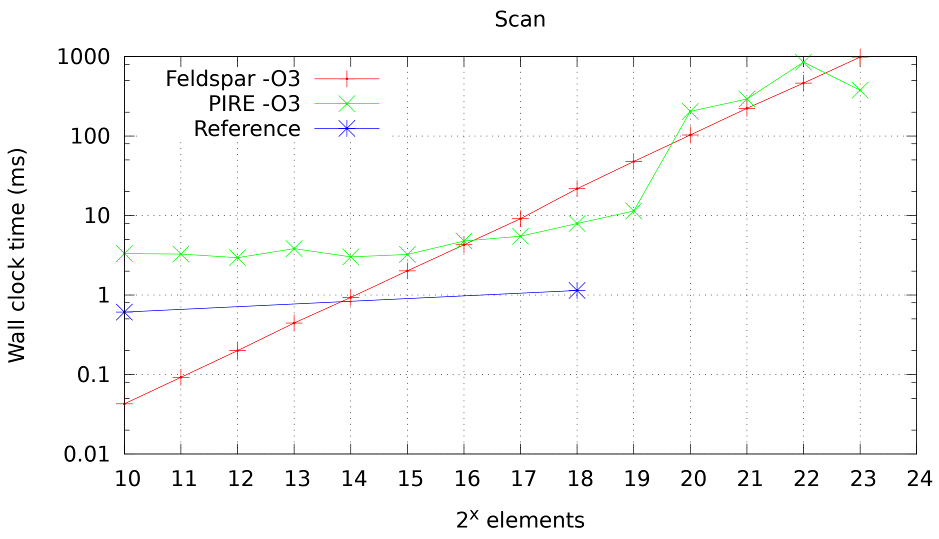

Figure 7.1: Running times for the scan case study.

7.2 Scan

It is not surprising that the sequential C code of Feldspar would outperform PIRE for smaller input sizes, since we have to expect some overhead for kernel calls. What perhaps is surprising is that it takes quite large input sizes ($n \geq 16$) for PIRE to outperform Feldspar, as can be seen in Figure 7.1. The spike for PIRE at 2^{19} is a bit unfortunate, and our guess is that it is related to memory congestion. It does, however, once again go below the Feldspar running time when reaching 2^{23} elements. PIRE is $6.95\times$ slower than the reference for 2^{18} elements, much due to the use of global memory in PIRE. For the same number of elements, PIRE is $2.7\times$ faster than Feldspar. Moreover, we see that for 2^{23} elements the result is very similar: PIRE is $2.6\times$ faster than Feldspar.

For small numbers of inputs, the extra copying that we noted in Section 6.1.1 will have a greater impact than for large inputs. This explains the some of the difference in time between PIRE and Feldspar for input sizes smaller than 2^{16}. Other reasons include memory overhead and use of global memory.

We also note that even the reference is slower than the sequential C code of Feldspar for small inputs. This indicates that there needs to be some form of process for deciding whether a computation is worth offloading or not.

7.3 Dot Product

The running time of our PIRE-generated code becomes (roughly) equal to the generated code of Feldspar for 2^{15} elements, and from that point stays below the Feldspar curve. For 2^{23} elements PIRE is $4.6\times$ faster than Feldspar. We also note the slight increase in PIRE for 2^{20} elements, which we assume is (once again) related to memory congestion on the device. This is similar to the spike we saw in the scan case study in Secion 7.2.

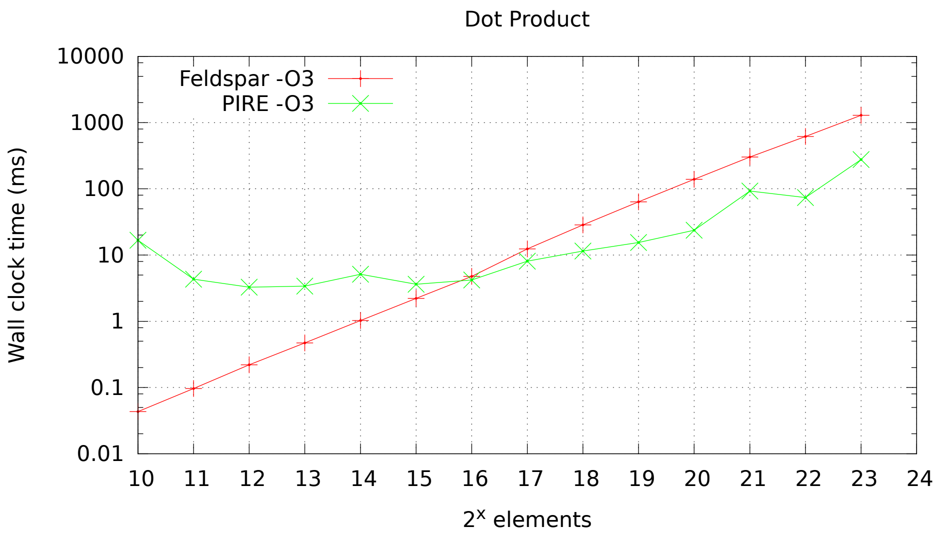

Figure 7.2: Running times for the dot product case study.

7.4 Bitonic Sort

The bitonic sort case study is the largest one in terms of code size, and is perhaps also the one that shows the most difference between Feldspar and PIRE. The point of intersection is at 2^{15}, and PIRE keeps below Feldspar from this point on. For 2^{20} elements PIRE is $5.3\times$ slower than the reference. However, for the same input size PIRE is $11\times$ faster than Feldspar. Furthermore, for 2^{23} elements, PIRE is $15.6\times$ faster than Feldspar. We also note the steep increase at 2^{16} elements, similar to what we've seen in the scan and dot product case studies already.

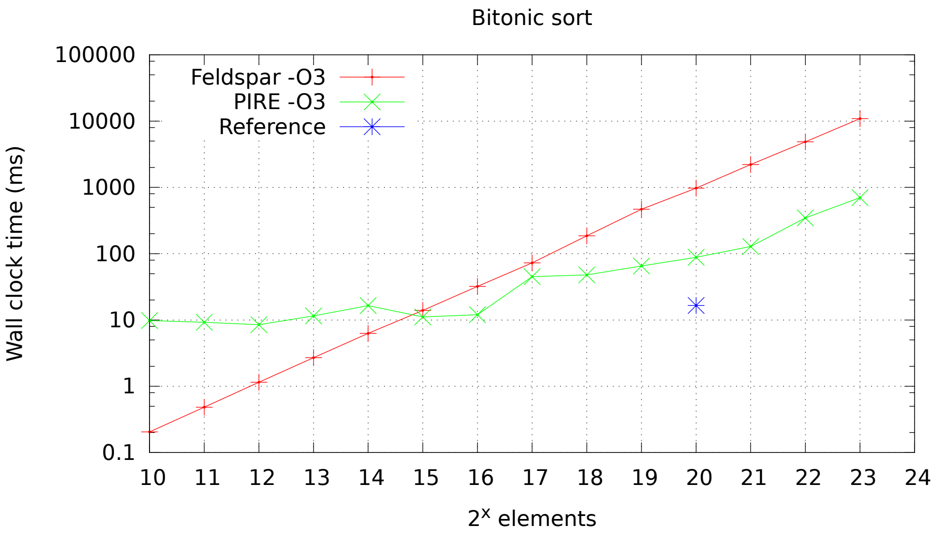

Figure 7.3: Running times for the bitonic sort case study.

Chapter 8

Related Work

This chapter addresses related work in the area of embedded domain-specific languages, parallelism and intermediate representations.

8.1 Nikola

Nikola (Mainland and Morrisett, 2010) is an EDSL in Haskell for array computations on GPUs. It is a high-level language, meaning it abstracts away from many lower-level details that would otherwise concern the programmer — marshalling of data, size inference on buffers, memory manage and automatic loop parallelisation. It targets CUDA and permits the choice of compile-time or run-time code-generation. Both approaches have drawbacks and benefits (mainly flexibility vs. efficiency), and as such allows the programmer to choose whichever is more beneficial.

In an attempt to maintain the syntactic convenience of Haskell, Nikola is implemented using a deep embedding with a higher-order abstract syntax (Pfenning and Elliot, 1988). The HOAS allows Nikola to express let-sharing and λ-sharing, two kinds of optimisations for minimising recomputation of common subexpressions and for reducing the final code-size.

Nikola relies on size inference to guarantee an upper bound on allocated memory buffers. To allow for this, the language has certain limitations. Many functions we expect to be able to write cannot be expressed. As an example, the following program is not accepted:

```
replicate (fold (+) xs) 1
```

The reason for this is that the result of `fold (+) xs` is not decidable ahead of time. Therefore, since all memory needs to be allocated before the call of a function, the solution employed by Nikola is to simply make the above program illegal.

Two benchmarks are run to demonstrate effectiveness of the language; the Black-Scholes call option evaluation and radix sort. The Nikola implementation of Black-Scholes is compared to hand-written CUDA as well as regular Haskell-code using Data.Vector-libraries. The Nikola implementation is evaluated for both compile-time and run-time code generation.

Their implementation of radix sort is a translation of one presented by Blelloch (1990). It is compared to an implementation in regular Haskell, again using Data.Vector, but not to any hand-written CUDA.

For Black-Scholes, compile-time Nikola is on par with hand-written CUDA, and it greatly outperforms regular Haskell. Run-time Nikola has a greater initial cost than all other implementations but eventually outperforms regular Haskell. Given enough data it (seems) to approach hand-written CUDA and compile-time Nikola for data-sets greater than approximately 2MB.

For radix sort, Nikola outperforms regular Haskell code for data-sets larger than approximately 32KB. It is unclear whether this is compile-time or run-time Nikola.

8.2 Accelerate

The Accelerate EDSL (Chakravarty et al., 2011) is aimed at GPGPU array programming, much in the same vein as Nikola (section 8.1) and Obsidian (Svensson, 2011). The languages all generate CUDA kernel programs, but Accelerate is the only one with programs that span several kernels. As we will see later, however, this is not without problem. Chakravarty et al. identify that fusion of adjacent kernels is a future necessity.

Accelerate works on arrays that are shape-polymorphic, a concept used by the Repa library (Keller et al., 2010). The notion is that of heterogeneous *snoc* lists (snoc being the reverse of cons). This is made concrete by introducing two data types:

```
data Z = Z
data tail :. head = tail :. head}
```

Using these types, we can define dimensions of arrays (i.e. array shapes):

```
type DIM0 = Z
type DIM1 = DIM0 :. Int
type DIM2 = DIM1 :. Int
type DIM3 = DIM2 :. Int
(etc)
```

DIM0 represents a scalar value. An array has type `Array sh e`, where sh is the shape and e is the element type. Thus, the type of arrays of three dimensions of Floats can be written as:

```
Array (Z:.Int:.Int:.Int) Float
```

or equivalently by using the shorthand:

```
Array DIM3 Float
```

In traditional EDSL fashion, Accelerate overloads the standard Haskell type classes such as `Num` and `Integral`. Since some Bool operations cannot be overloaded, special functions are introduced to handle this (including one for `if_then_else`). Accelerate also supports functions like `replicate`, something that Nikola does not, due to its size inference (Mainland and Morrisett, 2010).

The Accelerate operations build an abstract syntax tree (AST) of higher order. The AST is higher order, as it embeds function-valued expressions (lambda binders). Furthermore, the tree maintains in its nodes the full type information of the operations, and front-end transformations on the AST are type-preserving.

The AST has explicit constructs for operations like Fold, ZipWith and Scan. The constructs are translated to (hand-tuned) skeleton kernels, instantiated with concrete parameters (e.g. which function to use when zipping together arrays). Programs can be made up of more than one skeleton (ZipWith followed by Fold, for instance). This, however, results in several kernels, one for each such operation. This is often suboptimal, since kernel invocation carries overhead. Chakravarty et al. propose as future work fusion of adjacent kernels.

The front-end, after creating an initial AST, traverses the tree and turns it into a *nameless* representation, using *de Bruijn* indices. The AST becomes nameless in the sense that lambda binders no longer carry named variables. Instead we use a natural number to refer to an occurrence of a variable. This number denotes the number of lambda binders that are in scope between the occurrence and the binder corresponding to the occurrence. For instance, the K combinator λ`x.`λ`y.x` (i.e. `const` in Haskell) is encoded as λ`.`λ`.1`.

The benefit of using de Bruijn indices is that we do not have to rely on α-conversion when checking syntactic equality (because α-equivalence now corresponds to syntactic equality). For more information on de Bruijn indices and terms, see chapter 6 in Pierce (2002).

Accelerate employs memoisation to avoid repeated generation of kernels that are invoked more than once. This is done by hashing the (nameless)

AST and mapping the hash value to the binary code of the compiled kernel. This is a way of dealing with the overhead that comes with dynamic kernel generation. While this approach is still carrying overhead compared to pre-compiled kernels, Chakravarty et al. (2011) argues that "...it is only worthwhile to offload computations to a GPU if they are compute-intensive" and "...the overhead of dynamic kernel compilation is often not problematic in the face of long kernel runtimes and long data-transfer times between host and device memory".

Evaluation is done by benchmarking against hand-written library implementations of dot product, black-scholes and matrix multiplication. In the case of dot product, the library version is almost exactly twice as fast as the Accelerate version, regardless of input size. This is due to Accelerate implementing dot product using two kernels, whereas the library implements it with just one. For the black-scholes benchmark, Accelerate is shown to perform very close to the library implementation. For 9 million options, Accelerate takes 2.94 ms while the library takes 2.217 ms.

8.3 NESL

The first language that fully supported nested data-parallelism was NESL (Blelloch et al., 1993b). Concretely, this means that the NESL language and its implementation support both nested, irregular, data structures and nested data-parallel function calls.

Originally, NESL was run on the vector (SIMD or MIMD) supercomputers of the early 90s, like the Connection Machine CM-2 and the Cray C90. A goal of NESL was to be a portable language, and NESL uses the intermediate language VCODE (Blelloch and Chatterjee, 1990) and a library of vector routines called CVL (Blelloch et al., 1993a) to abstract away from low-level details. Thus, VCODE and CVL are the parts that have to translated when porting to new architectures.

We recall Figure 1.2, where a case of nested, unbalanced, parallelism is illustrated. Supporting such parallelism is generally considered difficult, but interestingly enough this is exactly the sort of parallelism that NESL supports. But just scheduling nested parallel computations as is doesn't work very well. We often end up with unbalanced scheduling, as is illustrated in Figure 8.1. Getting even load balance is a big problem when implementing nested data-parallelism. To solve the problem of unbalanced scheduling, NESL employs a technique called *flattening nested parallelism* Blelloch and Sabot (1990). The technique transforms a nested structure into a flat vector, which is much easier to properly load-balance. This vectorization is accomplished using so called segment descriptors in the underlying VCODE rep-

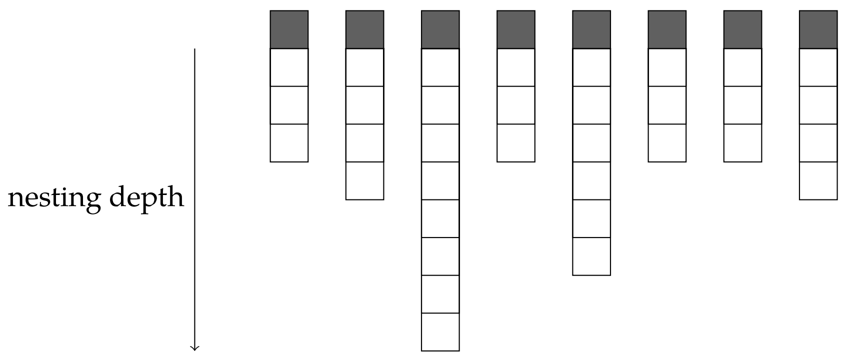

Figure 8.1: Schedulers (grey) with unbalanced workloads that follows from nesting data-parallel operations.

resentation (Blelloch et al., 1993b; Blelloch and Chatterjee, 1990). A segment descriptor describes the segmentation of a vector of values. In VCODE vectors are represented as value vectors coupled with segment descriptors. A one-dimensional vector is a pair:

```
segdes = [6] values = [4, 8, 15, 16, 23, 42]
```

In contrast, the nested vector [[4, 8], [15, 16, 23], [42]] is represented as:

```
segdes1 = [3]
segdes2 = [2, 3, 1]
values  = [4, 8, 15, 16, 23, 42]
```

Here, `segdes1` describes the segmentation of `segdes2`, and `segdes2` describes the segmentation of `values`.

Blelloch et al. (1993b) compares interpreted NESL (using a VCODE interpreter) to native code versions in a number of benchmarks. The benchmarks are performed on a number of machines, but we consider benchmarks run on the Cray C90 (since they are the most thorough). The native code of the C90 is Fortran 77. One of the benchmarks is sparse matrix-vector multiplication – a problem rich with nested parallelism.

In the sparse matrix-vector multiplication benchmark, each (sparse) matrix has its number of non-zero values fixed at 10^6 and the row length is what is variable. The NESL code outperforms the native code by a factor of ten for short row lengths. Blelloch et al. (1993b) notes that the running time of the NESL code is essentially independent of the row length. For row lengths greater than 128, the native code performs roughly equal to the NESL code. For row lengths greater than 256, the NESL code is also slightly slower due to interpretation overhead. Additional benchmarks are carried out, and the curious reader is referred to Blelloch et al. (1993b).

8.4 ispc

The Intel SPMD Program Compiler (ispc) aims to deliver very high performance on CPUs (Pharr and Mark, 2012), by using both scalar units and SIMD units of modern CPUs. They present not only a compiler, but also a language very similar to C/C++. Pharr and Mark take the approach that no single performance-focused language can be a good fit for both CPUs and GPUs. They therefore exclude GPUs altogether and focus solely on attaining high performance on CPUs. The focus is also on providing the programmer with performance transparency (which we also take to mean a lack of abstraction). As with C, the programmer should be able to get a good idea of how the code will perform once compiled.

As indicated by its name, ispc implements SPMD (single program multiple data) execution using the SIMD units found in CPUs. The authors call this model SPMD-on-SIMD. Several instances of a program are instantiated, one for each available SIMD lane. Each instance is running on its own SIMD lane, and is operating on its own data. The group of instances is referred to as a *gang*. As in CUDA and OpenCL, the program instances has access to an id descriptor that, for instance, can be used to index into arrays.

In addition to using SIMD as a means to achieve parallelism, the language also provides a spawn facility for asynchronous task launch. This can be used to start computation on a different kernel using a different hardware thread.

By default, data in ispc is replicated across program instances. This means that a variable like `float x` might contain a different value for different instances. However, some data such as loop variables we might want to share between program instances. The data can in such cases be prefixed with the keyword `uniform`, to indicate that data is shared. The ability to mark data as uniform makes a number of optimisations possible. For instance, uniform data is stored as a scalar, and since scalar and vector instructions can be issued concurrently it can lead to increased performance. Not only that, but it also alleviates pressure on vector registers since uniform data lives in scalar space. Furthermore, sharing data implies a smaller memory footprint.

For performance, ispc implements a way of representing "arrays of structures" (AOS) as "structures of arrays" (SOA). The SIMD units of modern CPUs often perform better when reading and writing data that is contiguous. Having an array of structures does not typically result in a contiguous data layout, but rather a scattered one. Arrays, on the other hand, are contiguous. The difference is illustrated in Figure 8.2. Generally, allowing for SOA leaves us with code that is quite verbose. By introducing a `soa` keyword, ispc achieve SOA layout while still making the rest of the program

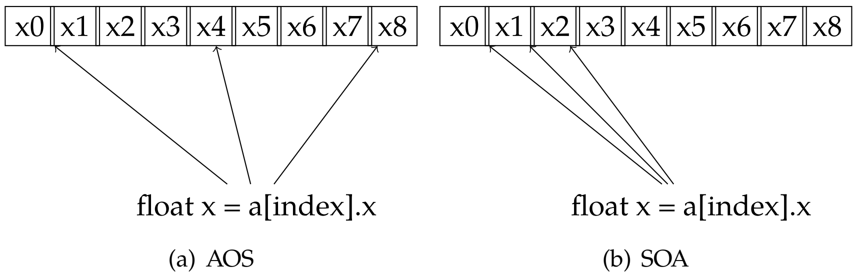

Figure 8.2: Reading data using AOS (left) and SOA (right). SOA results in more efficient code since data is contiguous and permits the use of vector read instructions.

(for arrays of length four) look like it was AOS (code from Pharr and Mark (2012)):

```
struct Foo{float x,y,z;};
soa<4> struct Foo a[...] = {...};
int index = ...;
float x = a[index].x;
```

Without the support for the soa keyword, we might instead have written:

```
struct Foo4{float x[4],y[4],z[4];};
uniform Foo4 a[...] = {...};
int index = ...;
float x = a[index/4].x[index & 3];
```

Pharr and Mark run seven case studies; ambient occlusion, binomial options, black-scholes, mandlebrot set, ray tracing, stencil computation and volume rendering. We outline just some of the results here.

Even on a single core, the speedups using the vector instructions (8-wide SIMD units) are significant. The theoretical maximum speedup one can hope for is 8x. The black-scholes benchmark benefits the most and gains a speedup of 7.43x over sequential C++ code. Ray-tracing is sped up by 6.85x, and the benchmark with the least speedup is volume rendering with a speedup of 3.24x.

Using a 40-core system with 4-wide SIMD units, volume rendering benefits the most with a 243.18x speedup over sequential code. Noteworthy is that the stencil computation only gains 9.40x (interestingly this is worse than when run on just 4 cores, where it gains 12.03x!). Pharr and Mark identify the problem as that the stencil computation is iterative and spawned asynchronous tasks all have to finish before the next iteration can start.

Chapter 9

Discussion

The most general conclusion that can be drawn from the results is that PIRE requires a relatively big input size to matter even a little bit. For both the scan and bitonic sort case study, PIRE outperforms Feldspar only for about 65000 elements. Outperforming Feldspar is, however, only mildly impressive, seeing how it is sequential. The gap between the reference running time and the running time of PIRE is great, and using local kernel memory might give us a way of closing the gap. We also suggest that unnecessary copying of memory needs to be minimised.

There seems to be little hope of beating Feldspar for a small number of elements. The scan case study indicates that the overhead for device offloading is still great. In our case, even the reference is slower than Feldspar for 1024 elements. It follows that one has to make a decision on whether or not a computation is actually worth offloading. This view on this kind of decision-making is shared also by Chakravarty et al. (2011).

Another point brought up by Chakravarty et al. (2011) is the topic of adjacent kernel fusion. It's an interesting concept, and we can expand upon the idea to fit PIRE even better. We often end up with kernel calls within a for loop (e.g. the scan case study, Section 6.1.1). By pushing the for loop into the kernel code, we can eliminate the cost of all but one kernel call.

We've seen in all of the case studies that at some point, there is a steep increase in the running time. The exact point varies between the case studies, however. Since they all show this tendency, we are inclined to believe that it is not just a random spike but rather something related to hardware. Memory congestion and limits on bus-speed are things that we expect to cause this behavior.

The original objective of this master's thesis was to also include the porting of the EDSL Obsidian (Svensson, 2011). This would have demonstrated the potential generality aspect of PIRE, had it been done. This objective was clearly over-ambitious, much due to failing to realise how complex Feldspar is. Furthermore, the original aim was also to make PIRE as independent as possible of the target language and surface EDSL. While more work is required to argue for this kind of generality, PIRE has been designed with generality in mind. The only part of PIRE that mentions anything about the target platform is the `Memory` type.

9.1 Future Work

The work presented in this thesis opens up several interesting possibilities for future work.

9.1.1 Using Local Kernel Memory

Arguably, the most limiting factor with regards to the performance of PIRE-generated code is the use of global memory in kernels. Global memory accesses are ubiquitous in our kernels, and each access very costly. We also guess that the sudden increases in running time that we've seen in in Section 7 have to do with global memory access. Making use of local memory is therefore a definite must for the future.

Since local memory is limited, we need to come up with a way of breaking down a problem into smaller pieces so they fit into local memory. These smaller pieces we generally refer to as work groups in OpenCL terms, and local memory is shared between the work items (threads) in a work group.

9.1.2 Running on Non-GPU Hardware

One of the problems of running OpenCL on a discrete device like the GPU is the overhead of moving data. It would therefore be interesting to explore the use of PIRE-generated programs on hardware where host and kernels share memory. OpenCL runs not just on GPUs, but on a wide variety of hardware. Several manufacturers of hardware like Intel[1] and AMD[2] have released OpenCL SDKs.

[1] http://software.intel.com/en-us/vcsource/tools/opencl-sdk

[2] http://developer.amd.com/resources/heterogeneous-computing/opencl-zone/

9.1.3 Mutable Data and Monadic Constructs in Feldspar

We made the decision to not include mutable data and monadic constructs in the Feldspar subset we are using. While justified from a thesis scope, it still limits us in what algorithms that can actually implement in Feldspar. With access to mutable data constructions in Feldspar, we could for instance implement counting and radix sort using the parallel scan from Section 6.

9.1.4 Additional Languages for Generality

In order to demonstrate the generality aspect of PIRE, it has to be incorporated into more languages. One suitable candidate is Obsidian (Svensson, 2011). We could also argue for the generality aspect if we add more backends. Generating code for OpenMP, Pthreads or CUDA are all viable options.

Chapter 10

Conclusion

This thesis has presented PIRE, a parallel IR for embedded languages, and its incorporation into the Feldspar EDSL. Through experiments, we have evaluated the code generating backend of PIRE. Despite the naïve approach we are taking, the results are looking promising. For fairly big input sizes we are outperforming the original Feldspar backend. Even though we are slower than the reference code, we are still hopeful. Incorporation of local memory and reducing copying will certainly bring us closer to the reference in the future. Finally, we hope that the future can better demonstrate some generality aspects of PIRE – our initial, overly ambitious, goal.

10.1 Looking Back

Working on this thesis has been quite the experience. I set out with a goal that turned out to be over-ambitious – to create what is now known as PIRE, and to port not just one, but two languages (Feldspar and Obsidian). We had to settle for one language – Feldspar – and thus abandon the goal of generality.

As I try to look back, the most valuable lesson learned has to be that it is really easy to overestimate one's self. While trying to grasp the code base of the Feldspar project was difficult and time consuming (future students beware – it *will* take more time than you think), and coming up with PIRE was challenging, nothing was harder than actually trying to stay within the time frame.

Bibliography

Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006. ISBN 0321486811.

Emil Axelsson. A generic abstract syntax model for embedded languages. In *Proceedings of the 17th ACM SIGPLAN international conference on Functional programming*, ICFP '12, pages 323–334, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1054-3. doi: 10.1145/2364527.2364573. URL http://doi.acm.org/10.1145/2364527.2364573.

Emil Axelsson and Mary Sheeran. Feldspar: Application and Implementation. volume 7241, pages 402–439. Springer-Verlag, 2012.

Emil Axelsson and Josef Svenningson. Combining deep and shallow embeddings for EDSL. In *Proceedings of the 13th International Symposium on Trends in Functional Programming (to appear)*, June 2012.

Emil Axelsson, Koen Claessen, and Mary Sheeran. Wired: Wire-aware circuit design. In *Proc. of Conference on Correct Hardware Design and Verification Methods (CHARME)*, pages 5–19. Springer Verlag, 2005.

Emil Axelsson, Koen Claessen, Gergely Dévai, Zoltán Horváth, Karin Keijzer, Anders Persson, Mary Sheeran, Josef Svenningsson, and András Vajda et al. Feldspar: A Domain Specific Language for Digital Signal Processing algorithms. In *Proceedings of the 8th ACM/IEEE international conference on Formal methods and models for codesign. IEEE*, 2010.

Emil Axelsson, Koen Claessen, Mary Sheeran, Josef Svenningsson, David Engdal, and Anders Persson. The design and implementation of Feldspar an embedded language for Digital Signal Processing. In *Proceedings of the 22nd international conference on Implementation and application of functional languages*, IFL'10, pages 121–136, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-24275-5. URL http://dl.acm.org/citation.cfm?id=2050135.2050143.

Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. Lava: hardware design in Haskell. In *Proceedings of the third ACM SIGPLAN interna-*

tional conference on Functional programming, ICFP '98, pages 174–184, New York, NY, USA, 1998. ACM. ISBN 1-58113-024-4. doi: 10.1145/289423.289440. URL http://doi.acm.org/10.1145/289423.289440.

Guy Blelloch and Siddhartha Chatterjee. VCODE: A data-parallel intermediate language. In *Proceedings of the 3rd Symposium on the Frontiers of Massively Parallel Computation*, pages 471–480. IEEE Computer Society Press, 1990.

Guy Blelloch and Gary W. Sabot. Compiling collection-oriented languages onto massively parallel computers. *Journal of Parallel and Distributed Computing*, 8:119–134, 1990.

Guy E. Blelloch. Prefix sums and their applications. Technical Report CMU-CS-90-190, School of Computer Science, Carnegie Mellon University, November 1990. Also appears in Synthesis of Parallel Algorithms, Reif (ed.), Morgan Kaufmann, 1993.

Guy E. Blelloch, Siddhartha Chatterjee, Jonathan C. Hardwick, Margaret Reid-Miller, Jay Sipelstein, and Marco Zagha. CVL: A C vector library – manual version 2, 1993a.

Guy E. Blelloch, Jonathan C. Hardwick, Siddhartha Chatterjee, Jay Sipelstein, and Marco Zagha. Implementation of a portable nested data-parallel language. In *Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPOPP '93, pages 102–111, New York, NY, USA, 1993b. ACM. ISBN 0-89791-589-5. doi: 10.1145/155332.155343. URL http://doi.acm.org/10.1145/155332.155343.

Manuel M.T. Chakravarty, Gabriele Keller, Sean Lee, Trevor L. McDonell, and Vinod Grover. Accelerating Haskell array codes with multicore GPUs. In *Proceedings of the sixth workshop on Declarative aspects of multicore programming*, DAMP '11, pages 3–14, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0486-3. doi: 10.1145/1926354.1926358. URL http://doi.acm.org/10.1145/1926354.1926358.

Koen Claessen. A poor man's concurrency monad. *Journal of Functional Programming*, 9(3):313–323, May 1999. ISSN 0956-7968. doi: 10.1017/S0956796899003342. URL http://dx.doi.org/10.1017/S0956796899003342.

Koen Claessen and John Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, ICFP '00, pages 268–279, New York, NY, USA, 2000. ACM. ISBN 1-58113-202-6. doi: 10.1145/351240.351266. URL http://doi.acm.org/10.1145/351240.351266.

Koen Claessen, Mary Sheeran, and Joel Svensson. Obsidian: GPU programming in Haskell. In *Proc. of 20th International Symposium on the Implementation and Application of Functional Languages (IFL '08)*. Springer-Verlag, 2008.

Koen Claessen, Mary Sheeran, and Bo Joel Svensson. Expressive array constructs in an embedded GPU kernel programming language. In *Proceedings of the 7th workshop on Declarative aspects and applications of multicore programming*, DAMP '12, pages 21–30, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1117-5. doi: 10.1145/2103736.2103740. URL http://doi.acm.org/10.1145/2103736.2103740.

Delphine Demange. *Semantic Foundations of Intermediate Program Representations*. PhD thesis, ENS Cachan - Bretagne, 2012.

Gergely Dévai, Máté Tejfel, Zoltán Gera, Gábor Páli, Gyula Nagy, Zoltán Horváth, Emil Axelsson, Mary Sheeran, András Vajda, Bo Lyckegård, and Anders Persson. Efficient code generation from the high-level domain-specific language Feldspar for DSPs. In *ODES-8: 8th Workshop on Optimizations for DSP and Embedded Systems, workshop associated with IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2010.

Conal Elliott. Functional images. In Jeremy Gibbons and Oege de Moor, editors, *The Fun of Programming*, "Cornerstones of Computing" series. Palgrave, March 2003. URL http://conal.net/papers/functional-images/.

Conal Elliott, Sigbjørn Finne, and Oege De Moor. Compiling embedded languages. *J. Funct. Program.*, 13(3):455–481, May 2003. ISSN 0956-7968. doi: 10.1017/S0956796802004574. URL http://dx.doi.org/10.1017/S0956796802004574.

Michael J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, 21(9):948–960, September 1972. ISSN 0018-9340. doi: 10.1109/TC.1972.5009071. URL http://dx.doi.org/10.1109/TC.1972.5009071.

Milind Girkar and Constantine D. Polychronopoulos. The hierarchical task graph as a universal intermediate representation. *International Journal of Parallel Programming*, 22(5):519–551, 1994.

Michael I. Gordon, William Thies, and Saman Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. *SIGARCH Comput. Archit. News*, 34(5):151–162, October 2006. ISSN 0163-5964. doi: 10.1145/1168919.1168877. URL http://doi.acm.org/10.1145/1168919.1168877.

Haskell.org. The Haskell programming language web page, 2011. The

Haskell web page for information on Haskell, compilers, tutorials and packages. Retrieved on February 04, 2013.

W. Daniel Hillis and Guy L. Steele, Jr. Data parallel algorithms. volume 29, pages 1170–1183, New York, NY, USA, December 1986. ACM. doi: 10.1145/7902.7903. URL http://doi.acm.org/10.1145/7902.7903.

P. Hudak. Modular domain specific languages and tools. In *Proceedings of the 5th International Conference on Software Reuse*, ICSR '98, pages 134–, Washington, DC, USA, 1998. IEEE Computer Society. ISBN 0-8186-8377-5. URL http://dl.acm.org/citation.cfm?id=551789.853532.

Paul Hudak, Tom Makucevich, Syam Gadde, and Bo Whong. Haskore music notation – an algebra of music. *Journal of Functional Programming*, 6(3): 465–483, May 1996.

John Hughes. Research topics in functional programming. chapter Why functional programming matters, pages 17–42. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990. ISBN 0-201-17236-4. URL http://dl.acm.org/citation.cfm?id=119830.119832.

Simon L. Peyton Jones, Roman Leshchinskiy, Gabriele Keller, and Manuel M. T. Chakravarty. Harnessing the multicores: Nested data parallelism in Haskell. In Ramesh Hariharan, Madhavan Mukund, and V. Vinay, editors, *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2008, December 9-11, 2008, Bangalore, India*, volume 08004 of *Dagstuhl Seminar Proceedings*. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2008. doi: http://drops.dagstuhl.de/opus/volltexte/2008/1769.

Samuel Kamin. Standard ML as a meta-programming language. Technical report, University of Illinois, Computer Science Dept., September 1996.

Gabriele Keller, Manuel M.T. Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, and Ben Lippmeier. Regular, shape-polymorphic, parallel arrays in Haskell. In *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming*, ICFP '10, pages 261–272, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-794-3. doi: 10.1145/1863543.1863582. URL http://doi.acm.org/10.1145/1863543.1863582.

Khronos Group. http://www.khronos.org/opencl/, 2013. Homepage for the OpenCL programming language. Retrieved on February 04, 2013.

Toma Kosar, Pablo E. Martinez Lopez, Pablo A. Barrientos, and Marjan Mernik. A preliminary study on various implementation approaches of domain-specific language. *Inf. Softw. Technol.*, 50(5):390–405, April

2008. ISSN 0950-5849. doi: 10.1016/j.infsof.2007.04.002. URL http://dx.doi.org/10.1016/j.infsof.2007.04.002.

Daan Leijen and Erik Meijer. Domain specific embedded compilers. In *Proceedings of the 2nd conference on Domain-specific languages*, DSL '99, pages 109–122, New York, NY, USA, 1999. ACM. ISBN 1-58113-255-7. doi: 10.1145/331960.331977. URL http://doi.acm.org/10.1145/331960.331977.

Geoffrey Mainland and Greg Morrisett. Nikola: embedding compiled GPU functions in Haskell. volume 45, pages 67–78, New York, NY, USA, September 2010. ACM. doi: 10.1145/2088456.1863533. URL http://doi.acm.org/10.1145/2088456.1863533.

Simon Marlow, Patrick Maier, Hans-Wolfgang Loidl, Mustafa K. Aswad, and Phil Trinder. Seq no more: better strategies for parallel Haskell. In *Proceedings of the third ACM Haskell symposium on Haskell*, Haskell '10, pages 91–102, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0252-4. doi: 10.1145/1863523.1863535. URL http://doi.acm.org/10.1145/1863523.1863535.

Simon Marlow, Ryan Newton, and Simon Peyton Jones. A monad for deterministic parallelism. In *Proceedings of the 4th ACM symposium on Haskell*, Haskell '11, pages 71–82, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0860-1. doi: 10.1145/2034675.2034685. URL http://doi.acm.org/10.1145/2034675.2034685.

Nvidia. http://www.nvidia.com/cuda, 2013. Homepage for the CUDA programming language. Retrieved on February 04, 2013.

Anders Persson, Emil Axelsson, and Josef Svenningsson. Generic monadic constructs for embedded languages. In *Proceedings of the 23rd international conference on Implementation and Application of Functional Languages*, IFL'11, pages 85–99, Berlin, Heidelberg, 2012. Springer-Verlag. ISBN 978-3-642-34406-0. doi: 10.1007/978-3-642-34407-7_6. URL http://dx.doi.org/10.1007/978-3-642-34407-7_6.

F. Pfenning and C. Elliot. Higher-order abstract syntax. *SIGPLAN Not.*, 23 (7):199–208, June 1988. ISSN 0362-1340. doi: 10.1145/960116.54010. URL http://doi.acm.org/10.1145/960116.54010.

Matt Pharr and William R. Mark. A SPMD compiler for high-performance CPU programming. In *Proceedings of the 2012 Innovative Parallel Computing: Foundations & Applications of GPU, Manycore, and Heterogeneous Systems*, InPAR '12, 2012.

Benjamin C. Pierce. *Types and programming languages*. MIT Press, Cambridge, MA, USA, 2002. ISBN 0-262-16209-1.

Mary Sheeran. Functional and dynamic programming in the design of parallel prefix networks. *J. Funct. Program.*, 21(1):59–114, January 2011. ISSN 0956-7968. doi: 10.1017/S0956796810000304. URL http://dx.doi.org/10.1017/S0956796810000304.

Jaspal Subhlok, James M. Stichnoth, David R. O'Hallaron, and Thomas Gross. Exploiting task and data parallelism on a multicomputer. In *Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPOPP '93, pages 13–22, New York, NY, USA, 1993. ACM. ISBN 0-89791-589-5. doi: 10.1145/155332.155334. URL http://doi.acm.org/10.1145/155332.155334.

Bo Joel Svensson and Mary Sheeran. Parallel programming in Haskell almost for free: an embedding of Intel's Array Building Blocks. In *Proceedings of the 1st ACM SIGPLAN workshop on Functional high-performance computing*, FHPC '12, pages 3–14, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1577-7. doi: 10.1145/2364474.2364477. URL http://doi.acm.org/10.1145/2364474.2364477.

Joel Svensson. Obsidian: GPU Kernel Programming in Haskell. Technical Report 77L, Computer Science and Enginering, Chalmers University of Technology, Gothenburg, 2011. Thesis for the degree of Licentiate of Philosophy.

P. W. Trinder, K. Hammond, H.-W. Loidl, and S. L. Peyton Jones. Algorithm + strategy = parallelism. *J. Funct. Program.*, 8(1):23–60, January 1998. ISSN 0956-7968. doi: 10.1017/S0956796897002967. URL http://dx.doi.org/10.1017/S0956796897002967.

Philip Wadler. The expression problem, November 12 1998. http://www.daimi.au.dk/~madst/tool/papers/expression.txt.

Appendix A

Additional PIRE Data Types

```
data UOp where
  BWNeg  :: Expr → UOp
  Deref  :: Expr → UOp

data BOp where
  Add     :: Expr → Expr → BOp
  Sub     :: Expr → Expr → BOp
  Mul     :: Expr → Expr → BOp
  Div     :: Expr → Expr → BOp
  Mod     :: Expr → Expr → BOp
  LT      :: Expr → Expr → BOp
  LTE     :: Expr → Expr → BOp
  GT      :: Expr → Expr → BOp
  GTE     :: Expr → Expr → BOp
  EQ      :: Expr → Expr → BOp
  NEQ     :: Expr → Expr → BOp
  And     :: Expr → Expr → BOp
  Or      :: Expr → Expr → BOp
  BWAnd   :: Expr → Expr → BOp
  BWOr    :: Expr → Expr → BOp
  BWXOr   :: Expr → Expr → BOp
  ShiftL  :: Expr → Expr → BOp
  ShiftR  :: Expr → Expr → BOp
```