Design and Implementation of an Array Language for Computational Science on a Heterogeneous Multicore Architecture

Paul Keir

Submitted in fulfilment of the requirements for the degree of Doctor of Philosophy

School of Computing Science
College of Science and Engineering
University of Glasgow

July 8, 2012

Abstract

The packing of multiple processor cores onto a single chip has become a mainstream solution to fundamental physical issues relating to the microscopic scales employed in the manufacture of semiconductor components. Multicore architectures provide lower clock speeds per core, while aggregate floating-point capability continues to increase.

Heterogeneous multicore chips, such as the Cell Broadband Engine (CBE) and modern graphics chips, also address the related issue of an increasing mismatch between high processor speeds, and huge latency to main memory. Such chips tackle this *memory wall* by the provision of addressable caches; increased bandwidth to main memory; and fast thread context switching. An associated cost is often reduced functionality of the individual accelerator cores; and the increased complexity involved in their programming.

This dissertation investigates the application of a programming language supporting the first-class use of arrays; and capable of *automatically parallelising* array expressions; to the heterogeneous multicore domain of the CBE, as found in the Sony PlayStation®3 (PS3). The language is a pre-existing and well-documented proper subset of Fortran, known as the 'F' programming language. A bespoke compiler, referred to as E♯, is developed to support this aim, and written in the Haskell programming language.

The output of the compiler is in an extended C++ dialect known as Offload C++, which targets the PS3. A significant feature of this language is its use of multiple, statically typed, address spaces. By focusing on generic, polymorphic interfaces for both the generated and hand constructed code, a number of interesting design patterns relating to the memory locality are introduced.

A suite of medium-sized (100-700 lines), real-world benchmark programs are used to evaluate the performance, correctness, and scalability of the compiler technology. Absolute speedup values, well in excess of one, are observed for all of the programs.

The work ultimately demonstrates that an array language can significantly reduce the effort expended to utilise a parallel heterogeneous multicore architecture, while retaining high performance. A substantial, related advantage in using standard 'F' is that any Fortran compiler can create debuggable, and competitively performing *serial* programs.

Acknowledgements

I would firstly like to thank my wonderful wife, Steffi, not only for her moral support, patience, and above all, love; but also for her practical advice, albeit through the refreshingly alternative perspective of a doctor of letters.

Many thanks are also due to my supervisor, Dr. Paul Cockshott, who possesses an uncanny, raw intelligence, and knack for constructing the very question one might hope to avoid; to the benefit of all in earshot. Paul was exactly the supervisor I needed, and I greatly appreciate his kindness, candour, wisdom, and tolerance over these last years.

I have seen far more of my second supervisor, Dr. John O' Donnell, than is prescribed by regulation, and I have benefited from our regular meetings more than he may know. Thankyou John for patiently introducing me to the mathematics behind programming languages.

Thankyou to all the academic and administrative staff in the School of Computing Science.

Dr. Alastair Donaldson, my first industrial supervisor, thankyou. You provided a solid operational structure; enthusiasm for the topic; and nothing less than a complete and thorough answer to any question I ever posed. Regrettably, time with my second industrial supervisor, Dr. Andrew Cook, was brief. Your assistance was nevertheless gratefully received. Andrew Richards, my third and final industrial supervisor, please accept my thanks also. Without your early decisions and integrity I would not be writing this. Thankyou Uwe Dolinsky, for your attention to detail, and diligent responses to my technical emails. To all those at Codeplay Software Ltd., many thanks for your assistance.

To Prof. Lenore Mullin, thankyou kindly for being our SICSA distinguished visitor, and introducing the school to the mathematics of arrays. Your wisdom and ideas remain a source of inspiration long after your departure.

To my dear colleague and companion, Youssef Gdura, you have reminded me of the essential quality of a good sense of humour. Your constant humility and equanimity, even in the face of a turbulent period for your homeland, remains a valuable lesson. Thanks also go to Mark

Shannon for being a friend on the road. Your certainty with regard to both technical and collegiate issues provided an essential ying to my doubting yang.

Thankyou to Wikipedia® and the Wikimedia Foundation Inc.

Contents

List of Tables

List of Figures

Listings

Glossary

ABI Application Binary Interface - A low-level interface between applications; including the operating system. An embedded ABI also specifies data type conventions.

absolute speedup The ratio of wall clock time spent on a serial calculation, to the time spent on the same calculation on a parallel machine. The processors should be the same. An ideal speedup is p, where p is the number of processors in the parallel machine. The serial version should be the fastest available; and may use a distinct algorithm.

AST Abstract Syntax Tree - A data structure encoding the essential structure of a program or program fragment. Source level syntactic details are omitted, though may be recovered by a compiler's code generator.

CBE Cell Broadband Engine. A heterogeneous processor designed by STI and used in the first petaflop supercomputer, Roadrunner; and more commonly in the Sony Playstation 3 (PS3). Also known as the Cell processor; Cell B.E.; or CBEA.

combinator A higher-order function with no free, or unbound, variables.

conformance A relation between array expressions denoting their suitability as arguments to a class of functions defined on scalars. Typically this implies that both arguments have the same shape, though commonly one may also remain scalar.

DMA Direct Memory Access - A protocol for the transfer of data between main memory and a device, independently of the CPU, using a dedicated hardware component.

dope vector A low-level representation of arrays used internally by Fortran compilers. For each dimension, a dope vector provides information such as the upper bound; lower bound; extents; and stride. An array's rank *can* be calculated from its dope vector, though rank is more readily available statically from the type system. A dope vector also includes a pointer to the base of the actual array data.

elemental An explicitly-specified attribute of a Fortran procedure, defined for scalar arguments, yet also applicable element-wise to an array expression.

hoisting A source-level compiler transformation wherein a subexpression is evaluated early, and stored in a variable. The variable is then textually substituted at the site of the original subexpression.

HPC High Performance Computing. A field of computing concerned with massive parallelism and synonymous with *supercomputing*. Often used in preference to the more specialised term, *computational science*.

intrinsic A built-in 'F' or Fortran function, or subroutine, is intrinsic; e.g. `print`.

kernel A component of a HPC program responsible for the majority of the calculations in, and duration of, a program's execution.

left recursive A formal grammar production rule whose definition contains itself as the left-most symbol.

overloading A programming language feature allowing multiple definitions of a procedure to use the same name. The version invoked by a call depends upon the type and number of arguments. Also known as ad hoc polymorphism.

PPE PowerPC Processor Element - The Power Architecture based CPU of the CBE; comprising the PPU and the PowerPC Processor Storage Subsystem (PPSS).

PPU PowerPC Processor Unit - The dual-threaded, 64-bit RISC processor of the PPE.

pure A computation free of side-effects. In Fortran a procedure denoted as pure is less strictly defined. For example, arguments to pure subroutines may be modified.

rank The rank of an array is the number of dimensions it contains.

section An array section is a piece of an array; either contiguous or strided.

side-effect A by-product of a computation, such as IO or the modification of state.

SPE Synergistic Processor Element - A coprocessing module of the CBE; comprising an SPU and a Memory Flow Controller (MFC). The CBE includes up to 8 SPEs.

SPU Synergistic Processor Unit - The 128-bit SIMD vector processing component of an SPE; including 256KB of local store memory.

STI Sony Toshiba IBM - The consortium responsible for development of the CBE.

The Cell SDK The IBM SDK for Multicore Acceleration.

vector An array with only one dimension; that is, having a rank of one.

Chapter 1

Introduction

Parallelism in computational science has for six decades applied itself to the simulation of physical processes such as weather systems, galaxy formation, or protein folding. More generally, the field of high performance computing (HPC), or supercomputing, also addresses computing problems from outside the realms of the physical sciences, tackling problems from areas including information security, search engine technology, and computational finance.

Contemporary desktop and mobile computing too has been compelled to embrace parallel hardware, and plays a significant role in what is often referred to as the multicore era. As previously, the central processing unit (CPU) on such devices remains a single silicon die chip, however this will now routinely have more than one processor core. Once again in the context of HPC, comparable symmetric multiprocessors have long existed in servers, workstations, and the nodes of some supercomputers; albeit with multi-socket motherboards. The last five years, however, have seen a transition to a fulsome mainstream acceptance of the shared memory model of parallelism provided by these multicore platforms. Whereas the sales of new computing hardware in recent decades had been driven by the lure of increasing processor clock speeds, more often it is now the number of cores, or low power consumption, which directs consumer demand.

The compulsion for such a significant change in processor architectures originates in fundamental physical factors. Traditionally, by reducing the processor die size, and thus the diameter of each wire, the power consumption falls, so permitting an increase in clock frequency, and hence power density. However, the resistive-capacitive (RC) delays in signal transmission lengthen proportionally with each increase in clock frequency. More significantly, the transistor leakage, a residual current present when a voltage is no longer applied, also becomes an increasingly pronounced consumer of power at such scales. Another major

factor is the high latency of memory access. The speed of memory has not kept pace with that of processors, and memory access times, measured in processor clock cycles, continue to increase. This phenomenon is known as the memory wall.

The individual processing elements of a multicore chip are clocked at a slower rate than the serial chips from a decade earlier. Thus are the pressures described above somewhat alleviated. As is often mentioned, however, there is no such thing as a free lunch. The toll demanded in this case is one of increased software complexity.

The multicore era is frequently referred to as the multicore *revolution*, and this is an apt denotation. For the user of medium to high level programming languages, prior advances in processor design may have required a compiler update. Other factors may have compelled a developer to utilise a different programming language. Nevertheless, that language typically remained within the prevailing paradigm of serial and imperative programming. For these modest efforts, performance of the resulting executable would typically improve by a factor proportionate to the increase in clock speed of the new circuitry.

In contrast, parallel programming, even within the restricted domain of shared memory systems, adds a considerable burden of complexity to the development task. Unlike a concurrent program, a parallel program aims to reduce the runtime of a program or subcomponent relative to a serial equivalent. A popular abstraction is the use of multiple threads of control; each existing within a single operating system process; and each with access to the same region of memory. Sequential dependencies in the order of access between common data structures will constrain the selection of code regions which may be executed in parallel to the main thread. However, even having identified a region suitable for parallel execution, timing measurements must be taken to ensure the overhead to launch and join a thread are not obstructive. Performance should ideally scale with the number of processor cores, and so, ultimately, regular control structures are sought, and leveraged to ensure a steady supply of work units. Consequently, looping constructs are a conventional target, with the iteration space partitioned across multiple threads. The freedom of each thread to access memory common to all, can also result in subtle errors, known as race conditions, relating to simultaneous access of the same memory location. Specialised compare-and-swap (CAS) instructions can facilitate locks and semaphores which may be employed to help avoid such race conditions, at the risk of introducing equally significant execution problems, such as deadlock.

In this instance, hardware design has a profound impact on that of software. Issues regarding the parallel decomposition of a program using conventional threading illustrate only one aspect of this. The more general question for research in parallel computing is substantial: specify a programming language, or language constructs, which can enable the development

of straightforward, correct, parallel programs with strong and scalable performance, and performance portability.

An alternative parallel programming model exists in the paradigm of collection-based, or array languages. Array languages offer a number of advantages over traditional models of explicit threads; message passing; or preprocessing directives. Array languages are implicitly parallel; deterministic; scalable; and have a predictable performance model. As serial and parallel execution models coexist, they are also straightforward to debug. The use of array languages is nevertheless an unfamiliar approach for many; and parallel execution is typically reserved for distributed architectures.

Heterogeneous systems add further complexity to the issues outlined so far. The memory available at each node in a supercomputer may or may not have direct hardware access to the memory of other nodes. In either situation, the different latencies must be accounted for to maximise performance. Similarly, a commodity heterogeneous cluster must account for the distinct capabilities of each node.

More recently, *heterogeneous multicore* systems have emerged. The application of graphics processing units (GPUs) to general purpose computation (GPGPU) marries a conventional *host* CPU to a graphics card via a PCI Express expansion port on the motherboard. Each GPU contains hundreds of low-clock speed, special purpose processors, groups of which share common access to local, addressable, cache-like memory. The Cell Broadband Engine (CBE) is comparable, though providing both the host processor, and the eight accelerators on a single chip. The accelerators of the CBE are, however, clocked at the same high rate as the host. Each accelerator again has its own small, local store memory. Both of these processor designs attempt to address the problem of the memory wall by facilitating the explicit prefetch of data.

1.1 Thesis Statement

Heterogeneous multicore architectures present a pragmatic solution to the physically-based problems associated with the continuation of Moore's law of increasing transistor counts (Moore, 1965). However, the introduction of both parallelism *and* discrete memory spaces places a substantial burden on those tasked with engineering correct and performant software. I assert that collection-based languages, heretofore used in distributed computing contexts, may profitably be applied within the domain of heterogeneous multicore programming. Programs written in such a language should make no explicit reference to parallelism, and hence eliminate a class of problems related to synchronisation. Performance of the result-

ing executable programs should scale in proportion to the number of coprocessors available, and outperform an equivalent serial version. Furthermore, the generated programs should themselves also be capable of serial execution, thereby simplifying initial development and debugging, while performing on a par with peers produced by a serial compiler applied to the same input language.

The claims above are demonstrated by the following:

- the design and implementation of an automatic parallelising compiler to transform the array expressions from a known subset of modern Fortran, into a form capable of parallel execution on the STI Cell Broadband Engine; and
- the verification by experiment of the correctness and performance of a suite of HPC benchmark programs adapted specifically for the proposed array language.

The target language is a novel dialect of C++, with parallel constructs, and a type system augmented to support multiple address spaces. Reference types are thus statically assigned a locality, and the traditional C++ overloading mechanism is extended for the locality of each address parameter. The compiler's code generator and runtime library must then carefully ensure that type correctness is maintained throughout. Hence it is anticipated that our knowledge of this recent parallel programming model will be deepened.

1.2 Contributions

This work contributes to the research into parallel languages for heterogeneous computing in the following ways:

- The feasibility of using a mainstream array language to develop implicitly parallel programs on a heterogeneous multicore architecture, with strong and scalable performance, is demonstrated. The contribution is verified empirically through the measurement of serial performance, absolute speedup, and the memory footprints of a suite of four, medium-size (100-700 lines of code) HPC benchmarks; reported in Chapter 6.
- An extensible source-to-source research compiler, E♯, is designed and developed; see Chapter 4. E♯ is implemented in the pure, non-strict, functional programming language, Haskell, and provides a significant case study in the application of the language to a large ($\sim$10,000 lines of code), industry-oriented, compiler project.

- The simplicity and thread safety of implicit parallelism, derived from a serial array language is demonstrated, alongside competitive serial performance results, in Chapter 6. Distinctively, the base language is a subset of a prominent *serial* HPC language, supported comprehensively by numerous industry and research compilers. Furthermore, no implementation dependencies are created by the introduction of a runtime library.

- A suite of HPC benchmark programs are created, compatible with the advocated approach for structuring array programs, and applied in Chapter 6.

- A C++ array class template, binary compatible with the array formats of all Fortran compilers, and built upon the low-level 'C' Chasm library, is developed, and presented in Section 5.2. The array class also abstracts the memory locality of its underlying data in a form compatible with the Offload C++ compiler. A set of C++ functions, equivalent to many of the built-in operations of Fortran are also developed.

- A problem involving manual replication in the Offload C++ language is outlined, along with the subsequent development of a methodology to reduce or eliminate its impact. The solution is applied within Offload C++ class definitions, parametrised by the locality of underlying data, and is described in Section 5.4.

- An augmented version of the C++ heap memory allocation operator, *new*, is introduced in Section 5.4.2. Distinctively, this allocator can specify the address space, or *locality*, of the requested memory in a form compatible with the use of an integer template argument corresponding to pointer locality.

- A novel smart pointer class is demonstrated in Section 5.4.4, capable of safely encapsulating the locality of a contained memory address, and compatible with the use of a constant integer template parameter to represent that address space.

1.3 Publications

The work discussed in this dissertation led to the following publications:

> Paul Keir, Paul W. Cockshott and Andrew Richards, *Mainstream Parallel Array Programming on Cell*, in Proceedings of the 5th Euro-Par Workshop on Highly Parallel Processing on a Chip (HPPC'11), 2011. (Keir et al., 2011)

Collaborative work on topics directly related to the main topics of the thesis also resulted in the following publications:

George Russell, Paul Keir, Alastair F. Donaldson, Uwe Dolinsky, Andrew Richards and Colin Riley, *Programming Heterogeneous Multicore Systems using Threading Building Blocks*, in Proceedings of the 4th Euro-Par Workshop on Highly Parallel Processing on a Chip (HPPC'10), 2010. (Russell et al., 2010)

Alastair F. Donaldson, Paul Keir and Anton Lokhmotov, *Compile-time and Runtime Issues in an Auto-parallelisation System for the Cell BE Processor*, in Proceedings of the 2nd EuroPar Workshop on Highly Parallel Processing on a Chip (HPPC'08), 2008. (Donaldson et al., 2008)

1.4 Outline

The remainder of the dissertation is structured as follows:

Chapter 2 looks at related work by focusing either on a technology's relevance to parallelism on heterogeneous architectures; or to parallelism through array expressions. The presentation is conducted through the prism of three lenses: the first considers parallel interfaces provided by libraries, starting with those specifically targeting the CBE; the second examines the use of preprocessor directives within parallel programs; the third provides and overview of relevant parallel languages.

Chapter 3 provides an overview of the methodology advanced in support of the thesis goals; respectful of the apparatus available. The chapter begins with a description of the architecture of the CBE, followed by a detailed look at the Offload C++ language and compiler. Thereafter, the 'F' dialect of Fortran 90 is introduced, followed by a presentation on the parallel execution model, as implemented by the project compiler, E♯.

Chapter 4 describes the implementation details of the E♯ source-to-source compiler. Starting with an overview of all the compilers involved in the transformation from 'F' source to parallel executable, the chapter continues with a description of the compiler intermediate forms; followed by the parsing and compiler front-end; C++ code-generation in the back-end; and concluding with a description of the crucial parallelising transformations.

Chapter 5 presents the E♯ runtime library, looking at the C++ class representations for Fortran character strings and arrays; both developed in a form compatible with the dual address space type system of the Offload C++ compiler. The Offload C++ concept of

locality is then applied to the design of a novel smart pointer and memory allocator. C++ template metaprogramming is then applied to emulate the type checking and overload resolution mechanisms otherwise provided by a standalone Fortran compiler. Optimisations for automatic array and character string allocation conclude the chapter.

Chapter 6 begins by introducing the results of some microbenchmarks addressing questions of minimal kernel launch overheads and memory footprint. HPC benchmark programs developed and substantially adapted from existing codes are subsequently described, and their performance measured and analysed through experiment. Further code modifications are then explored to expose additional optimisations performed by the E♯ compiler.

Chapter 7 concludes the dissertation by placing the results of the research in a broader context, and highlights both the successes and failures of the project. Opportunities to extend the work exist aplenty, and the latter portion of the chapter presents a number of interesting directions with which this research may be developed further.

Chapter 2

Related Work

The design of the software interface by which parallel programs are constructed, is an area of intense research. Furthermore, this is likely to remain the case for some decades as the computing industry transitions from multicore to heterogeneous manycore. Programmers will seek new tools to compete in terms of productivity, performance, and reliability, in a computing hardware landscape of increasing complexity. A first recommendation from the well-cited *View from Berkeley* report is uncontroversial:

> "The overarching goal should be to make it easy to write programs that execute efficiently on highly parallel computing systems" Asanovic et al. (2006)

This chapter discusses the state of the art for parallel software interfaces as a background to the thesis. In particular, research relevant to heterogeneous multicore architectures is presented, with some emphasis on the CBE. Discussion begins in Section 2.1 with examples of the use of libraries as an appendage to existing non-parallel languages. The subsequent Section 2.2 considers APIs centred on the use of compiler and preprocessor directives. Section 2.3 illuminates developments in object-oriented parallel language *extensions*, providing context to the integral use of Offload C++ (Cooper et al., 2010; Russell et al., 2010) within the research; itself discussed at length in Section 3.2. Section 2.4 then considers array languages, starting with the seminal APL, before swiftly moving on to *parallel* array languages. Finally, the nascent partitioned global address space (PGAS) programming model is considered in Section 2.5. This interesting paradigm introduces data locality, while retaining the simplicity of a global, distributed view of memory; and targeting the HPC sector, as addressed by the present research.

2.1 Library Support for Parallelism

Parallel libraries often provide the first API for a new parallel architecture, and can appear as a straightforward solution to developers familiar with an existing language. POSIX Threads (The Austin Group, 2010) is a celebrated example: cross-platform, standardised, and well supported; it is even available on the heterogeneous architecture of the PS3. Nevertheless, there is strong evidence, for thread-based libraries at least (Boehm, 2005), to suggest that developers concerned with code correctness or safety should perhaps consider alternative, language-based, solutions.

2.1.1 The Cell SDK

Development under Linux on the PlayStation 3 is typically predicated by the installation of the IBM Software Development Kit for Multicore Acceleration (The Cell SDK) [1]. Custom versions of the GNU Compiler Collection (GCC) toolchain, developed at the Barcelona Supercomputing Centre as part of their *Linux on Cell* program, are included with the SDK. The version of the GCC compiler targeting the PPU (PPU-GCC) included with IBM SDK 3.0 provides a front-end to the C; C++; Fortran; and ADA languages, while an SPU cross-compiler version (SPU-GCC) supports all bar ADA. The IBM XL C/C++ and IBM XL Fortran compilers are also provided. Both compiler vendors provide the option to target either the PPU, or the SPU. SPU compilation in particular is optimised for the restrictive SIMD architecture of the SPU, and includes support for C/C++ language extensions (IBM Corporation, 2007a) such as the `vector` keyword. The very simplest method to execute a program on one SPU is therefore to prepare a small C, C++, or Fortran program, and compile as an executable. As only the PPU has access to the operating system, system calls are merely *initiated* by the SPU, using a *stop-and-signal* instruction (IBM Corporation, 2007b), before subsequent execution by the PPU. Such a program is known as an *spulet*.

Another routine method to invoke the execution of an SPE thread utilises the SPE Runtime Library Management Library, more commonly identified as *libspe*. Developed by IBM, libspe provides a low-level API, providing the PPE host with control over both the execution of SPE programs; and bidirectional DMA data transfer between main memory and SPE local store. From within PPE host code, libspe provides an abstract representation of an SPE; an *SPE context*. Once initialised, an SPE context may be used to synchronously load and run an SPE *main* program from a 'C'-compatible programming language environment.

[1] The tools, libraries, and middleware available from the Sony Computer Entertainment Developer Network (DevNet) (Sony Computer Entertainment) are another routine choice for developers in the industrial sector.

```
#include <libspe2.h>

extern spe_program_handle_t spe_handle;

int main(int argc, char *argv[]) {
  unsigned int entry = SPE_DEFAULT_ENTRY;
  spe_context_ptr_t ctx = spe_context_create(0, NULL);
  spe_program_load(ctx, &spe_handle);
  spe_context_run(ctx, &entry, 0, NULL, NULL, NULL);
  return 0;
}
```

Listing 2.1: Minimal synchronous PPE initiation of an SPE program.

```
#include <stdio.h>

int main(unsigned long long spuid) {
  printf("Hello from SPU %llx\n", spuid);
  return 0;
}
```

Listing 2.2: Simple SPE program suitable for libspe.

Listing 2.1 presents a minimal, synchronous launch of an SPE program, addressed through the externally defined `spe_program_handle_t` variable, `spe_handle`. The symbol and definition of `spe_handle` are generated by the GNU `ppu-embedspu` command-line tool. Given two arguments: the filepath of an executable SPE file; and a symbol name to bind to a fresh `spe_program_handle_t` variable, the `ppu-embedspu` tool will produce a linkable PPU object file.

Assuming files `ppe_main.c` and `spe_main.c` contain the text of Listings 2.1 and 2.2 respectively, a sequence of commands to generate an executable incorporating both of the prescribed SPE and PPE programs is listed in Figure 2.1.

```
$ spu-gcc spe_main.c -o spe_main
$ ppu-embedspu spe_handle spe_main spe.o
$ ppu-gcc ppe_main.c spe.o -lspe2
```

Figure 2.1: Command sequence for PPE binary with embedded SPE program.

2.1.2 Threading Building Blocks

Threading Building Blocks (TBB) (Intel Corporation, 2010) is a cross-platform C++ library for programming homogeneous, shared-memory multicore processors. TBB was instigated, and is currently supported by, Intel, and provided under both a commercial, and a GPL v2-style open-source license. The object-oriented TBB API provides support for multiple programming models: data-parallelism, including loop and user-defined reduction operations; pipeline parallelism; and task parallelism. TBB also provides a selection of concurrent container classes, with synchronisation obtained either using fine-grained locks, or lock-free synchronisation. TBB also includes concurrent memory allocation routines; mutual exclusion primitives; atomic operations; and timing routines.

In our publication, Russell et al. (2010), it is demonstrated that a portion of the data-parallel TBB API is also suitable for execution on the *heterogeneous* parallel architecture of the Cell. In this work, the extended Offload C++ language (Cooper et al., 2010; Russell et al., 2010) is utilised, implementing the `parallel_for` class, via a straightforward library implementation.

2.2 Annotations, Directives and Pragmas

Many parallel programming APIs make significant, and essential use of code annotations. Such annotations, which are also referred to as preprocessor directives, or pragmas[2], though may also be embedded within comments, are parsed either by the compiler, or a symbiotic tool of the compiler; such as a preprocessor. Unlike the handling of simple macro substitutions, however, the program transformations necessary to enact the directives of a parallel annotation API require more substantial transformations, and typically rely heavily on the main compiler. A unifying concept behind the utilisation of code annotations to enable parallelism, is that the semantics of the original, serial, program should not be altered by their presence. Nevertheless, such APIs frequently depend on the use of a complementary runtime library, which may well alter the program's meaning; whether by design or otherwise.

The use of such annotative methods have been particularly well adopted by the HPC community. Although typically far more than compiler *hints*, there *is* a connection with implicit parallelism, as employed by E♯: a traditional serial compiler, albeit provided with a runtime library of stub routines, may simply ignore the annotations to regain a serial execution

[2] An early reference to the use of the word "pragma" in this context is Ichbiah et al. (1979, pages 2-3), where its derivation is attributed to the Greek word for action.

semantics.

2.2.1 High Performance Fortran

High Performance Fortran (HPF) is an influential design for a portable, distributed data-parallel language developed by the HPF Forum (HPFF), and published as a series of informal specifications, starting with version 1.0 in 1993 (High Performance Fortran Forum, 1993), and concluding in 1996 with version 2.0. The approach of HPF continues a vein of research into data-parallel, distributed-memory languages from the early nineties, including CM Fortran (Thinking Machines Corporation, 1991), Fortran D (Fox et al., 1990), and Vienna Fortran (Zima et al., 1992). HPF is an extension of the Fortran 90, and later Fortran 95, languages, and adopts an SPMD execution model; a single thread of control; and a single, *global* address space. Though not described as such at the time, HPF can be seen to adopt the contemporary Partitioned Global Address Space (PGAS) model discussed later.

Many of the HPF constructs which enable parallelism are familiar from Fortran 90/95: the `where` and `forall` statements and constructs; array assignment; elemental functions; and transformational intrinsics will all execute in parallel. Compiler directives may also be used to expose further parallelism. For example, the placement of the `independent` directive immediately prior to a `forall` or `do` loop, expresses the programmer's knowledge that the execution order of loop iterations will not affect the overall result, so permitting a parallel execution. Listing 2.3 illustrates a `do` loop preceded by the `independent` directive:

```
!hpf$ independent
do i = 1,128
  a(perm(i)) = b(i)
end do
```

Listing 2.3: Independent loop iterations in High Performance Fortran.

Of the compiler directives available in HPF, the `independent` directive, shown above, is the only *executable directive*. The other seven directives are *specification directives*, and provide information regarding specific aggregate variable declarations. Specification directives are used to express the intended physical distribution of arrays among participating compute nodes.

While an HPF program without directives is implicitly parallel, performance can be significantly improved by their careful inclusion. Listing 2.4 demonstrates four specification directives. On line 2, the `processors` directive specifies the number, and abstract topology,

```
1 real, dimension(1024,1024) :: a
2 !hpf$ processors p(4,4)
3 !hpf$ template   t(16,16)
4 !hpf$ align      a(i,j)   with t(i,j)
5 !hpf$ distribute t(block,block) onto p
```

Listing 2.4: Distribution directives in High Performance Fortran.

of the processor network participating in the program. The `template` directive on line 3 is akin to a declaration of a virtual array, with no memory allocated, and represents the smallest useful granularity for subsequent parallel decomposition. On line 4 the `align` directive asserts that the elements of `a` will physically reside by the same processor as arrays aligned either with `a`, or the template `t`. Lastly, the `distribute` directive on line 5 uses `block` arguments to indicate that a regular, rectilinear partitioning should be applied, so distributing each array aligned with `t` onto the processor network represented by `p`.

Such directives are specified to be semantically equivalent to comments. Pragmatically this presents the advantage that an algorithm may be quickly and iteratively developed on a serial machine, prior to further testing and performance profiling in parallel; assuming sufficient memory is available. Note that this influential use of compiler directives was employed previously by CM Fortran (Thinking Machines Corporation, 1991), where the Fortran comment character, `C`, placed at column 1, forms a pun with the acronym "CMF". For example: `cmf$ layout x(:news).`

A compliant HPF implementation should include a set of intrinsic and runtime procedures. Alongside archetypal system inquiry functions such as `number_of_processors` and `processors_shape`; the runtime library exposed through the `hpf_library` module includes procedures to perform non-generic array reductions, prefix and suffix scans; scatters; gathers; and sorting operations. In HPF 2.0, reductions can also be obtained by augmenting an `independent` directive with a `reduction` clause.

Although the HPF language itself is today almost extinct, it is noted that the Japanese HPC community has maintained persistent interest and activity relating to HPF (Kennedy et al., 2007). From 2002 to 2004 the Japanese vector supercomputer, the Earth Simulator, was the fastest in the world (Meuer et al.), demonstrating a performance of almost 15 Teraflops on the IMPACT-3D plasma simulation program, using an HPFF-approved extended version of HPF: HPF/JA. The Earth Simulator 2, commissioned in 2009, continues to utilise and develop HPF through extension in the form of HPF/ES.

The lack of more widespread adoption of HPF has been attributed to factors including: high initial expectations combined with immature compiler technology; missing features; poor

performance portability; and the complexity of performance tuning. Nevertheless, the HPF project has in its entirety been a notable success. As an alternative to the complexity of message passing, a significant quantity of HPC language research now pursues the PGAS model popularised by HPF. The use of HPF-style compiler directives for parallelism also remains pervasive, as shown throughout this chapter section. A particularly direct influence of HPF can be seen in the nascent, Japan-centric XcalableMP project (XcalableMP Specification Working Group, 2011), which utilises both directives, and a PGAS model. HPF has also influenced Fortran, as evidenced by the adoption in Fortran 95 of the HPF `forall` statement and construct; and HPF `pure` procedure qualifier. Even the most recent Fortran standard, Fortran 2008, owes a debt to HPF via its inclusion of the HPF bitwise array reductions for the *and*, *or* and *exclusive or* operations: `iall`; `iany`; and `iparity`. Also in Fortran 2008, the transformational function `parity`; along with elemental bit counting functions `trailz`, `popcnt`, and `poppar` originate in HPF.

2.2.2 OpenMP

OpenMP defines a suite of compiler directives, library routines, and environment variables, for use in shared-memory parallelism specified by the C, C++, and Fortran languages. The *OpenMP Application Program Interface* (OpenMP Architecture Review Board, 2011), is an informal standard produced and published by the OpenMP Architecture Review Board (ARB), and follows the work of the Parallel Computing Forum (The Parallel Computing Forum, 1991). The OpenMP ARB is an international body comprising 21 individuals, each representing a prominent industrial or academic institution. The most recent version of the OpenMP specification is version 3.1, and published in July 2011.

OpenMP adopts a fork/join model of parallelism, and supports a single global address space for all participating threads. OpenMP parallelisation annotations are represented lexically using `#pragma` compiler directives when C/C++ is the base language. Fortran has less established support for the use of a preprocessor, and OpenMP directives in free source form Fortran are consequently encoded as a comment starting with the sentinel `!$omp`[3]. That OpenMP directives may be treated as comments is a concept borrowed from HPF (Kennedy et al., 2007). The syntax of the precursor to OpenMP, developed by the Parallel Computing Forum (The Parallel Computing Forum, 1991), used a language-based syntax.

Listings 2.5 and 2.6 demonstrate the OpenMP loop construct, both in Fortran and in 'C'. Assuming that p threads are participating, the iteration space of an OpenMP parallel loop is by default split into p contiguous chunks of approximately equal size. This default corre-

[3]Fixed source form Fortran also permits `c$omp` and `*$omp` as sentinels.

```
subroutine reverse(n, a, b)
  integer i, n
  real(kind=8) a(:), b(:)
  !$omp parallel do
  do i = 1,n
    a(i) = b(n+1-i)
  end do
  !$omp end parallel do
end subroutine reverse
```

Listing 2.5: OpenMP array reversal in Fortran.

```
void reverse(int n, double *a, double *b)
{
  int i;
  #pragma omp parallel for
  for (i = 0; i < n; i++)
    a[i] = b[n-1-i];
}
```

Listing 2.6: OpenMP array reversal in 'C'.

sponds to providing the `static` argument to the optional `schedule` clause, which may also accept `dynamic`, `guided`, `runtime`, or `auto`. The above loop construct is made concise also by the default action of assigning `shared` status to all variable references within its scope. An exception is made for the iterator, `i`, which sensibly defaults to `private` and receives special handling. Without these defaults, the loop directive in the Fortran `reverse` example of Listing 2.5 would become as shown in Listing 2.7.

```
!$omp parallel do schedule(static) shared(n,a,b) private(i)
```

Listing 2.7: Verbose OpenMP parallel do directive in Fortran.

Many aspects of OpenMP are nondeterministic. Regarding the ordering and visibility of updates to shared addresses, OpenMP has a relaxed memory consistency model. More specifically, it implements a variant of *weak ordering* (Hoeflinger and de Supinski, 2005), distinguished both by the provision of an explicit *flush* operation, and by that operation's facility to update a *subset* of the shared memory locations. The order in which updates made as part of a worksharing construct become visible to other threads are similarly affected. For example, were the Fortran `reverse` code of 2.5 to perform the reversal "in-place", using one array instead of two, as shown in Listing 2.8, the outcome would depend on multiple race conditions, and the result is therefore non-deterministic.

The vector reversal example can also illustrate a strength of Fortran array notation. The

```
subroutine reverse_race(n, a)
  integer i, n
  real(kind=8) a(:)
  !$omp parallel do
  do i = 1,n
    a(i) = a(n+1-i)
  end do
  !$omp end parallel do
end subroutine reverse_race
```

Listing 2.8: Non-determinism in OpenMP.

specification of an "in-place" reversal, still in serial, may use a negative-stride array section as shown in listing 2.9. This concise encoding may also be safely parallelised. The OpenMP specification (OpenMP Architecture Review Board, 2011) describes a Fortran-specific *work-share construct* which can parallelise a structured block consisting of statements and constructs chosen from the list below:

- array assignments
- scalar assignments
- **forall** statements
- **forall** constructs
- **where** statements
- **where** constructs
- **atomic** constructs
- **critical** constructs
- **parallel** constructs

```
subroutine reverse_serial(n, a)
  integer n
  real(kind=8) a(:)
  a(:) = a(n:1:-1)
end subroutine reverse_serial
```

Listing 2.9: In-place Fortran array reversal in serial.

The workshare construct therefore enables the parallelisation of array assignment statements. The right hand side of an array assignment, is an array expression, and so the workshare construct may be compared to E♯'s parallel interface. Nevertheless, an implementation of the workshare construct in a compiler targeting heterogeneous parallelism is not know to exist. In fact, while the GNU Fortran compiler has for some time given partial parallel

support to the OpenMP workshare construct, the simple vector reversal code of Listing 2.10 is sufficiently idiomatic as to produce erroneous results, even in its most recent release, version 4.6.

```
subroutine reverse_parallel(n, a)
  integer n
  real(kind=8) a(:)
  !$omp parallel workshare
  a(:) = a(n:1:-1)
  !$omp end parallel workshare
end subroutine reverse_parallel
```

Listing 2.10: In-place OpenMP Fortran array reversal in parallel.

The workshare construct is in fact well known in programming folklore for a combination of both implementation complexity, and scarcity of real world usage. While many issues relating to synchronised variable update in OpenMP are left to the programmer, the distinctive semantics of Fortran array assignment should, in principle, ensure that concise algorithms such as `reverse_parallel` are well specified. This is enshrined in the OpenMP standard by the following quote:

> "An implementation of the `workshare` construct must insert any synchronization that is required to maintain standard Fortran semantics." (OpenMP Architecture Review Board, 2011, page 54)

OpenMP also supports a fixed set of associative, commutative, parallel reduction operations[4], while the runtime library routines, and associated environment variables, allows the query and alteration of the OpenMP runtime environment. Locking procedures too, join other synchronisation constructs such as the `barrier`, `critical` and `atomic` directives. OpenMP 3.0 added support for task parallelism.

OpenMP Implementations

Early implementations of OpenMP came from traditional HPC compiler vendors, and also the research community; with projects such as NanosCompiler (Ayguadé et al., 1999; Balart et al., 2004), OdinMP (Karlsson and Brorsson, 2004), and Omni (Kusano et al., 2000). Meanwhile, over the last decade, OpenMP adoption has evolved to the point where virtually every C/C++ and Fortran compiler includes the technology. Nevertheless, to accompany

[4]Though subtraction is neither associative nor commutative, OpenMP does provide a subtraction reduction. This operation actually involves both addition and subtraction.

the advancing OpenMP standard, new OpenMP research compilers, such as the Rose compiler (Liao et al., 2010), continue to appear. Attempts have even been undertaken to add support for OpenMP to other languages, for example: both JOMP (Bull et al., 2000), and more recently, JaMP (Klemm et al., 2007), use Java; and even an ADA version was proposed in (Stpiczynski, 2003).

Although OpenMP is intended for homogeneous systems, its success has nevertheless spurred implementers towards more challenging architectures. Distributed systems, though not an ideal fit for the OpenMP memory model, which assumes equal memory access latency for all processors, have nevertheless received attention. Distributed OpenMP implementations typically use a software distributed shared memory (DSM) library to provide the OpenMP shared address space abstraction. Treadmarks' software DSM (Amza et al., 1996) was chosen by early adopters such as Lu et al. (1998) at Rice University who propose and implement two modifications to the OpenMP standard when applied to clusters: removal of the `flush` directive; and a default sharing attribute of `private` rather than `shared`. Research at Basumallik et al. (2002) at Purdue University found naïve compilation of realistic benchmark programs were lacking in performance terms, and proposed further compiler optimisations. Contemporaneous research involving Purdue University and others (Eigenmann et al., 2002) asks "Is OpenMP for Grids?". Compilation technology on this project builds on the Polaris Fortran compiler (Blume et al., 1996). Treadmarks software DSM is introduced again, though conscious of its attendant overheads, only *irregular* access to arrays would fall back to this approach, with remaining communications using message passing. Again, some extension of OpenMP is proposed, with directives for data distribution, computation distribution, and communication operations described.

Intel's Cluster OpenMP (Hoeflinger, 2006; Terboven et al., 2008) is a commercial offering in this arena, and another user of the Treadmarks software who finds the OpenMP standard too constraining: Cluster OpenMP incorporates one new directive, `sharable`, to follow the declaration of variables which will be accessed by more than one thread. Only a subset of OpenMP is targeted by the STEP tool proposition (Millot et al., 2008), which compiles to native 'C' with MPI. Balder (Karlsson et al., 2002) is an OpenMP runtime library using its own software distributed shared memory (DSM) library. Integrated within the OdinMP compiler (Karlsson and Brorsson, 2004), Balder was recently re-written (Karlsson, 2008) to support OpenMP 2.0 and multi-processor nodes. The Omni OpenMP compiler (Kusano et al., 2000) has also been adapted for such architectures (Sato et al., 1999). An alternative approach, described in Huang et al. (2003), advocates the use of the Global Arrays Toolkit's (Nieplocha et al., 2006) library implementation of single-sided, shared memory distributed communication primitives, as part of a source-to-source translation strategy from OpenMP to Global Arrays.

As with distributed architectures, implementations of OpenMP for heterogeneous architectures must focus on correct and efficient distribution of data among participating compute units. Local memory may also be severely restricted, and support for paged memory absent. In addition, differing instruction sets, memory alignment constraints, and reduced OS support, conspire to create a greater implementation challenge. The most significant OpenMP compiler for the CBE to utilise the SPEs was originally developed by IBM under the moniker of "Octopiler" (Eichenberger et al., 2006; O'Brien et al., 2008). Structured blocks within OpenMP constructs are here transformed by the compiler using a process known as *outlining*, wherein a new function is generated, corresponding to the action of the structured block. A call to the fresh function then replaces the original code section. The outlined function, along with all functions called therein, are then compiled for both the SPU; and also the PPU, as the PPU master thread also participates in the parallel workload. The PPU master thread uses a runtime library to manage parallel tasks, and signal registers to assign work to, and control, as many SPUs as required. An SPU may contact the PPU using a mailbox, and performs a busy wait loop until further communication arrives from the PPU. The detrimental performance effect of naïve, ad hoc DMA requests to shared memory are minimised by the compiler itself controlling much of the data movement, while a software cache takes care of remaining unoptimised data references. The compiler also supports SIMD parallelism on both SPE and PPE, with automatic SIMD code generated from suitable loop iterations, and handling data alignment problems by the insertion of shift operations on contiguous registers where necessary. Much of the "Octopiler" research was subsequently merged into the IBM XL compiler series, as *XL C/C++ for Multicore Acceleration*. Nevertheless, though IBM also produced the capable *XL Fortran for Multicore Acceleration*, a version using OpenMP to target SPU parallelism was never released. The IBM XL compiler series for CBE is now discontinued. Outside of IBM, comparable research is scarce. Early work on an OpenMP CBE compiler from China was however presented in Wei and Yu (2008).

More recently, OpenMP has also been demonstrated on heterogeneous GPU architectures, sometimes by the same research groups which previously worked on distributed implementations. OMPCUDA (Ohshima et al., 2010) converts some OpenMP codes to Nvidia's GPGPU language, CUDA (NVIDIA Corporation, 2011), and is constructed using the venerable Omni OpenMP compiler (Kusano et al., 2000). An optimising OpenMP to CUDA compiler developed at Purdue University is described in Lee et al. (2009), presenting impressive performance results on four benchmarks. Both regular and irregular loops benefit from compile-time transformations to optimise access to GPU global memory. A two-stage translation scheme is employed, using an intermediate OpenMP representation optimised for GPU architectures. Although the Purdue group claim to be the first such OpenMP to CUDA compiler framework, earlier work is presented in Keir (2007). Here, a subset of OpenMP is extended with the `allocatable` directive, used in two ways: to annotate "serial" pointer

declarations later accessed within a parallel region; and to annotate functions which should be compiled also to target the GPU architecture.

In JCudaMP (Dotzler et al., 2010), the JaMP compiler (Klemm et al., 2007) has also been extended to target GPU architectures supporting CUDA, using a custom Java class loader; CUDA code specific to discovered hardware; and runtime invocation of Nvidia's CUDA compiler. Two new directives are provided: `tiled`, and `managed`. The `tiled` directive is required for arrays too large for GPU memory, thereby avoiding runtime checks, while prohibiting random access. The `managed` directive meanwhile aims to simplify the requirements regarding the use of an optimised array package, and signals a compiler transformation of a standard Java array declaration, and its accesses, to that of the new array class. Systematic restrictions appear, both in the reduced set of OpenMP constructs supported; and also to the Java language features permitted in parallel regions. ClusterJaMP (Veldema et al., 2011) is a recent extension of JCudaMP to target GPU-enabled clusters, using MPI as the communication fabric. ClusterJaMP partitions arrays according to the result of brief bandwidth and performance tests, and dynamically adapts the set of participating compute devices according to a "grow" autotuning heuristic.

2.2.3 Cell Superscalar

The Cell Superscalar (CellSs) framework (Bellens et al., 2006; Perez et al., 2007) is a single-source, directive-based parallelising API targeting the CBE architecture exclusively. The framework, which is built on the Nanos Mercurium compiler (Balart et al., 2004), provides a mechanism for annotating function *definitions*, allowing subsequent function *calls* to be represented internally as a data dependency graph. Having built said graph, the CellSs runtime can then transparently schedule the asynchronous execution of each function call as a *task* on each participating SPU, with attendant data transfers handled automatically. CellSs supports both the C99 language, and a subset of Fortran 95. The example code from Listing 2.11 is now introduced.

CellSs has three types of pragma directives: initialisation and finalisation pragmas; task pragmas; and synchronisation pragmas. A `task` directive, as used above on line 1, identifies a side-effect free procedure, suitable for asynchronous execution on the SPU. The `input`, `output`, and `inout` clauses, must specify the read/write attributes of the procedure parameters, while pointers used as arrays must also include the extents. To launch a task, it is sufficient to call an annotated function within the lexical scope of a block delimited by the `start` and `end` directives. The CellSs runtime is able to analyse data dependencies between SPU tasks, and so identify and schedule those which may safely be executed in parallel with

```
1  #pragma css task input(size) input(data[size]) output(result)
2  void psum(double *result, double *data, int size) {
3    *result = 0.0;
4    for (int i=0; i < size; ++i)
5      *result += data[i];
6  }
7
8  void sum(double *data, int size) {
9    double tmp1, tmp2;
10   int half_size = size / 2;
11 #pragma css start
12   psum(&tmp1, data,           half_size);
13   psum(&tmp2, data+half_size, half_size);
14 #pragma css barrier
15   printf("Result is %g\n", tmp1+tmp2);
16 #pragma css finish
17 }
```

Listing 2.11: Simple Cell Superscalar parallel summation.

one another. The runtime is not, however, able to assist with race conditions arising from data accessed by both PPU and SPU code, and consequently a synchronisation directive, such as the `barrier`, shown above on line 14, may be necessary.

The CellSs framework is somewhat distinct from other directive-based APIs in that it provides *no* user-level runtime library. This emphasises the facility of a directive-based API to provide an observably equivalent serial and parallel program simultaneously.

CellSs compilation proceeds at the level of translation units, followed by an object linking stage. The occurrence of a task *call*, in a translation unit, which does not also contain the task definition, requires further annotation. For such a situation, the `target` directive is provided. The `target` directive is positioned prior to the relevant function's definition, and lists the architectures, from `spu` and `ppu`, for which object files should be generated. The Offload C++ compiler's `__duplicate` function attribute performs a similar role.

2.3 Language Extensions

Offload C++ (Cooper et al., 2010; Russell et al., 2010) is an extension of the C++ language developed by Codeplay Software Ltd., targeting heterogeneous parallel architectures, including the PS3. Notably, Offload C++ is utilised as the back-end language of the project's research compiler: E♯. The Offload language and compiler are discussed fully in Section 3.2. In brief, Offload C++ introduces the *offload block*, which launches a synchronous or

asynchronous SPU thread, and will duplicate the enclosed call-graph wherever required to accommodate both PPU and SPU architectures; and thus provides a single-source language solution. Offload C++ also extends the C++ type system by the static assignment of an address space to each pointer.

In the remainder of this section, parallel object-oriented language extensions, comparable to Offload C++, are considered, starting with the precursor to the Offload C++ language: Sieve C++. Though also included here, OpenCL is both more and less than a language extension, and *often* considered the de facto competitor to Nvidia's dominant GPGPU C/C++ language extension: CUDA (NVIDIA Corporation, 2011).

2.3.1 Sieve C++

Sieve C++ (Lindley, 2007; Lokhmotov et al., 2007, 2008; Donaldson et al., 2008) is an extension to C++ designed and developed for the PS3 by Codeplay Software Ltd. Sieve blocks allow domain-specific knowledge of program data dependencies to be expressed in the form of annotations on compound statements and functions. This additional information is then used by Codeplay's VectorC compiler to aid detection of program regions suitable for parallelisation. These dependency assertions are supplied in the form of a novel semantic concept known as the *sieve construct*.

The syntax of the sieve construct is comprised of the `sieve` keyword, followed by a compound statement; referred to as a *sieve block*. The implicit assertion of the construct is that within a sieve block, no writes are followed by a read from the same location, on data declared outside of its scope.

Code within a sieve block may still access data declared outwith. Writes to such data, however, are *delayed* until execution of the sieve construct has completed. Prior to their execution, each write is stored, in sequential order, as an address-value pair in a data structure known as the *side-effect queue*. Consequently, while both writes following reads (*anti-dependencies*), and writes following writes (*output dependencies*), are preserved, reads following writes (*true dependencies*) are prohibited (Kennedy and Allen, 2002, pages 37-38). With reference to the *call-by-value/result* parameter passing mode of Algol-W (Reynolds, 1981, page 168), and more recently Sequoia (Fatahalian et al., 2006), the mechanism has been referred to (Donaldson et al., 2008) as *call-by-value/delayed result*.

Listing 2.12 illustrates the delayed side-effects of the sieve construct. The variable `a` is declared outside the scope of the sieve block, and written to twice, at line 7 and line 10. However, subsequent reads within the block obtain only the value held by `a` at the *start*

```
1  int main(int argc, char *argv[])
2  {
3    int a = 0;
4
5    sieve {
6      int b = 0;
7      a = a+1;
8      b = b+1;
9      printf("%d %d\n", a, b); // prints 0 1
10     a = a+1;
11     b = b+1;
12     printf("%d %d\n", a, b); // prints 0 2
13   }
14
15   printf("%d\n", a); // prints 1
16   return 0;
17 }
```

Listing 2.12: Delayed side-effects in Sieve C++.

of the construct. Both of the `a+1` assignment expressions therefore resolve to the value 1. An unoptimised side-effect queue will hold both updates of `a`. The queue is first-in/first-out (FIFO), though for this example, the order is not important; any order would result in `a` being assigned to 1.

A block tagged with the `sieve` keyword is potentially suitable for parallel execution. Removing true dependencies on externally declared data means that the compiler's alias analysis need only focus on data declared inside the sieve block, so providing the compiler with more opportunities to reorder and parallelise statements in a sieve block. A sequential ordering protocol applied to the processing of side-effect queues, then ensures that execution is deterministic. Consequently, debugging is simplified, as sequential and parallel results are identical.

Loops are still required to obtain maximum performance. A more realistic example, a simple 1D convolution, follows. In Listing 2.13, another keyword, `splithere`, is introduced. A `splithere` statement signifies a point in a sieve block where execution may be partitioned between available processors.

Were two processors applied to the above example, with a value of 1024 for `size`, 511 loop iterations might be supplied to each processor. Crucially though, there is no race condition on the update of array variable `x`'s 512th element[5].

[5]This convolution example is handled exactly as the Fortran 90 array expression, `x(1:size-1) = (x(0:size-2) + x(2:size)) / 2.0`

```
void conv1d(double *x, int size)
{
  sieve {
    for (intitr i(1); i < size-1; i++) {
      splithere;
      x[i] = (x[i-1] + x[i+1]) / 2.0;
    }
  }
}
```

Listing 2.13: The `splithere` statement of Sieve C++.

2.3.2 OpenCL

Open Compute Language (OpenCL) (Khronos OpenCL Working Group, 2011) is an open, royalty-free, cross-platform, industry standard defining an interface to modern, heterogeneous, parallel architectures. OpenCL defines a platform model including a host, and one or more discrete computing devices, which may include multicore CPUs, GPUs, and other processors such as DSPs, and the CBE (IBM Research, 2009); referred to as OpenCL *devices*. OpenCL provides a high-performance, low-level, C/C++ based API, including support for SIMD operations by the provision, through language extension, for built-in vector types such as `float2`, `ulong3`, or `double16`. OpenCL is maintained by the non-profit consortium, the Khronos Group.

OpenCL provides both a data-parallel, and a task-parallel programming model; though the former is emphasised. Each OpenCL kernel is embodied by a serial function, qualified with the `kernel` keyword, and defined within an associated domain specific kernel language: an extended subset of ISO C99 known as OpenCL C. The OpenCL compiler provided by each OpenCL runtime library is responsible for translating the OpenCL C code into an executable program, and is typically accessed solely through host API calls such as `clCreateProgramWithSource`, which receives an array of character strings as input; `clBuildProgram`; and `clCreateKernel`.

In anticipation of kernel execution, a host creates one or more coordinating *command queues*, each associated with a specific OpenCL device; the commands themselves configure kernel execution, memory operations, and synchronisation. The `clEnqueueNDRangeKernel` host API call can then be used to enqueue the kernel. Data-parallel execution of a kernel instance is subsequently applied to each point within an abstract domain of indices. A single thread corresponds to a *work-item*, assigned a global identifier within a rectilinear framework of up to three dimensions. Work-items are also grouped into *work-groups*, wherein they may share access to fast, *local* memory.

Under the alternative *task parallel* execution model, a task is logically equivalent to a kernel executing in a single work-group, of size one. A task is enqueued by calling `clEnqueueTask`.

An interesting aspect of OpenCL, that also draws comparison with the Offload C++ language, is its model of complex memory hierarchies, wherein four distinct regions of memory are specified: *global*, *constant*, *local*, and *private*. As with Offload C++, the type system of OpenCL C is extended by the incorporation of these discrete address spaces, with pointers qualified using similarly named keywords; and defaulting to `private`. However, as with Embedded C (WG14, 2006), the type system is restrictive in terms of genericity. Each pointer argument to an OpenCL C user function must match *exactly* the type specified by the function declaration; wherein a pointer type incorporates *one* address space. Furthermore, there is no function or operator overloading in OpenCL C.

2.3.3 Cilk++

The Cilk++ language (based on Cilk Blumofe et al. (1995); Frigo et al. (1998)) extends C++ by three keywords: `cilk_spawn`, `cilk_sync` and `cilk_for`. The Cilk++ compiler is available for homogeneous shared-memory x86 architectures and is packaged alongside a work-stealing runtime scheduler, a race detector and parallel profiler. Non-determinism remains an issue for the language, however, type-checked constraints on the launching of threads (strands) and the existence of sequential semantics allows the race detector to identify all non-determinism *introduced* by a Cilk++ construct, albeit assuming well-chosen test data and an absence of locks.

2.3.4 Deterministic Parallel Java

Built upon the *ForkJoinTask* framework of Java 1.7, Deterministic Parallel Java (DPJ) (Bocchino et al., 2009; Adve et al., 2009) adds two significant concepts to the Java programming language: explicit fork-join concurrency control primitives, in the form of the `cobegin` block and `foreach` loop; and a region-based *type and effect system*. Research towards automatic region annotation is ongoing. For now, however, significant effort may be required to convert existing Java codes to DPJ. Nevertheless, a suitably annotated DPJ program can guarantee deterministic behaviour equivalent to a sequential counterpart, along with good performance scaling on homogeneous multicore.

2.4 Array Languages

Many programs utilise libraries, or directives to enable parallelism. Nevertheless, it has been argued that such fragmented approaches possess intrinsic danger. A vibrant example from Boehm (2005) is illustrated by the two 'C' statements in Listing 2.14; each intended for execution by a separate thread.

```
1 if (x == 1) ++y;
2 if (y == 1) ++x;
```

Listing 2.14: One statement per thread, is there a race?

Under a sequential memory consistency model (Hoeflinger and de Supinski, 2005), with `x` and `y` initially set to zero, is there a race? Though the answer appears to be in the negative, a valid, albeit unlikely, compiler transformation could result in Listing 2.15, also from Boehm (2005), wherein a race condition *is* now present.

```
1 ++y; if (x != 1) --y;
2 ++x; if (y != 1) --x;
```

Listing 2.15: One statement per thread, a race is present.

The language and compiler research community are motivated by such considerations, towards the long term prospect of an intrinsically language-centric approach to parallel program specification; with attendant syntax, semantics, and memory model as required.

Array programming languages have existed for almost as long as high-level languages, and have been applied to parallel problems since the 1980s. They are typically defined with deterministic semantics, and avoid the use of locks in favour of implicit parallelism. The divisibility of arrays ensures that even a simple arithmetic expression may be efficiently parallelised. We begin our review at the beginning, historically, with APL, before examining a number of parallel array language implementations.

2.4.1 APL

APL is the seminal array programming language. Originally a system of mathematical array notation devised by Kenneth Iverson from around 1954, it was over ten years later, in 1965, before an interpreter was created for the IBM 7090; using punched cards as input (Falkoff and Iverson, 1978). Even today, APL implementations are often interpreted. An APL interpreter

is usually accompanied with an integrated development environment, allowing the user to save workspaces, or *notebooks*, including current name bindings and function definitions. The APL name is itself taken from the initial letters of Iverson's early book covering the notation of arrays, *A Programming Language* (Iverson, 1962).

Creating an implementation compelled the language designers to linearise the hitherto mathematical notation, producing unexpectedly positive outcomes. One example is that by replacing the use of subscripts and superscripts to denote array ranks, arrays of higher rank could be specified. Another is the use of composite characters, where one character can overstrike another; facilitating mnemonic schemes within groups of similar functions[6]. The distinctive non-ASCII character set of APL did not succumb to change, boasting frequent use of Greek letters and mathematical symbols; often requiring a custom keyboard. As an example of the syntax, the mathematical product $\prod_{i=1}^{100} i$ can be denoted in APL by $\times/\iota 100$; where $/$ is the reduction operator.

APL pioneered early programming language efforts to abstract from low-level machine operations; specifying the order of instructions, and which registers should be loaded. The language eschewed the tradition in mathematics of exploiting function precedence, primarily because this was established for only a handful of classical functions. Instead, a simple evaluation rule was enforced: "Every function takes as its right hand argument the value of an entire expression to the right of it." This permits a frequent omission of parentheses, enabling a concise style. Along with the character set, the familiar comparison to Egyptian hieroglyphics is clear.

APL provides a selection of built-in functions, and a small number of *operators*. An APL operator is a higher-order function, a combinator in fact[7]. Early versions of APL allowed neither user-defined operators; nor the application of operators to user-defined functions; and supported only binary or unary functions, referred to as dyadic and monadic respectively.

More recently, Kenneth Iverson has created the J programming language (Peelle, 2004), which builds on the concepts of APL, with support for *point-free*, or *tacit programming*; along with an ASCII character set. Meanwhile many other implementations of APL, and the APL ideology continue to flourish[8], with parallel implementations also reported; for example in Budd (1984) and Sauermann (1990). The influence of APL is also seen directly and indirectly in more recent languages such as MATLAB (Downey, 2008), Fortran 90, or K (Systems, 2012).

[6]This is particularly relevant given APL's restriction to 88 characters.

[7]The concept nomenclature of an operator in APL inherits from the classic operators such as the derivative or convolution operators.

[8]The APL acronym often now addresses *array programming languages* in general.

2.4.2 Fortran 90

Fortran and F are described comprehensively in Section 3.3, however it is worth remembering that the design of Fortran 90 had data-parallel execution in mind. One of the motivating factors behind the introduction of first class array support in Fortran 90 was the potential for array expressions to be evaluated in parallel. Commercial Fortran compilers such as PGI Visual Fortran (The Portland Group, 2011) provide *automatic* parallelisation on *homogeneous* multicore systems. With the `-Mconcur` option set, loops which are free of cross-iteration data dependencies, and of sufficient size, are potential targets for automatic parallelisation. Due to the possibility of side-effects, loops containing procedure calls are not parallelised.

In the ADAPT project, described in Merlin (1992, 1991), all array expressions appearing in executable statements are automatically transformed for parallel execution on MIMD, message-passing, distributed memory architectures, such as networks of INMOS transputer chips. While the system would automatically parallelise standard Fortran 90 codes, a declaration attribute for arrays, `distribution`, was added to the language, and paired with a new intrinsic function, `distr`. Restrictions on the use of these distributed arrays in declarative contexts also exist. In addition, the size of the distributed processor array is supplied in a separate file. Unlike E♯, the system is therefore a *language extension*. No performance figures were made available from ADAPT, and the project later evolved to focus on HPF, or rather a subset of it (Merlin et al., 1996).

2.4.3 ZPL

ZPL is an imperative, implicitly parallel, array-based programming language developed at the University of Washington (Snyder, 1999). The language primarily targets distributed and serial architectures in a portable manner through the compiler's source-to-source translation into ANSI C, alongside a set of retargetable communications libraries, including crucial support for MPI. Performance and scaling of ZPL programs have been observed to compare favourably to hand-coded 'C' and Fortran with MPI. The name, ZPL, makes a clear reference to the seminal array programming language: APL. Nevertheless, the ZPL acronym is customarily expanded to: Z-level Programming Language. The *Z-level* refers to the highest of three levels of programming abstraction defined in relation to a parallel architecture model known as the Candidate Type Architecture (CTA) (Snyder, 1999, pages 115-116) and defined by Lawrence Snyder in 1986. The initial design of *ZPL* was developed in the early 1990s by Lawrence Snyder and Calvin Lin, based on concepts developed in the Orca C language (Alverson et al., 1998).

Most of the common imperative, scalar operators are provided by ZPL; including arithmetic, relational, logical, bitwise, and assignment operators. Along with a traditional suite of basic scalar types, ZPL also provides derived types including record types and arrays. ZPL has no parallel constructs; instead, all array operations are executed in parallel.

ZPL uses a distinctive form of rectilinear index template, known as a *region*. A region may be used both to define the extent of a new array; and also to specify the set of indices affected by an array operation. In Listing 2.16, the one-dimensional arrays, `X`, `Y`, and `Z`, have their extents defined on line 5 by the region, `R1`, itself declared on line 3; and likewise for the two-dimensional arrays `A`, `B`, and `C`. The `config` keyword denotes a constant value, which may conveniently be set from the command line arguments using a similarly named switch. That the `n` binding is constant results in constant regions, and although this is recommended, it is not necessary.

Line 9 of Listing 2.16 also demonstrates the scoping of regions, whereby one region is active, for each dimensionality, for the scope of the statement it precedes. Hence, the specification of region `R2` on line 9 applies to the two-dimensional assignment on line 13. The increment of `A` on the following line, however, is governed by the preceding, unnamed, literal region `[1..512,1..512]`; and so only one quarter of array `A` is altered. Likewise, the placement of region `R1` on line 9 controls all one-dimensional statements within the scope of the outermost compound statement; and hence affects the assignment statement on line 11.

```
1  program ZPL1;
2  config var n : integer = 1024;
3  region R1 = [1..n];
4  region R2 = [1..n,1..n];
5  var X, Y, Z : [R1] double;
6  var A, B, C : [R2] double;
7
8  procedure ZPL1(dim : integer);
9  [R1] [R2] begin
10             if dim = 1
11               then X := Y + Z
12               else begin
13                      A := B + C
14       [1..512,1..512] A += 1
15                    end;
16           end;
```

Listing 2.16: Using regions in ZPL.

Regions are used throughout ZPL, and amongst other things, may be used to define the halo regions required in relaxation algorithms. This is performed simply, and without modifying indices, using a second region type, *directions*, which modify regions using the `@` and `of`

operators. Such support for the region concept is pervasive, and robust throughout ZPL; for example, no access to individual array elements is permitted.

A second type of array, an *indexed array*, is also provided. Indexed arrays are used for explicitly serial operations. No nesting of the default parallel arrays is possible, though an indexed array of parallel arrays is; and vice versa[9]. ZPL provides many other useful aggregate operations, such as reduce, scan, expand, and arithmetic progression generators. Procedure overloading, user-defined reductions, and first-class status for regions have also recently been introduced to the language.

2.4.4 Single Assignment C

Single Assignment C (SAC) (Scholz, 2003; Grelck and Scholz, 2006) is a strict, purely functional array language initiated in 1994 at the University of Kiel. Development of SAC is motivated by an ambition to provide a high-level array language abstraction, in a functional setting, yet with the performance characteristics of a high-performance imperative language such as Fortran or 'C'. In particular, SAC targets large scale, array-based scientific computation.

The reference to the 'C' language in the appellation is borne out by the syntax. The relationship to the 'C' language is intended both to provide a reduced threshold for the SAC novice; and also to facilitate a simplified mapping from some SAC language components to 'C', which is used by the SAC compiler as an intermediate format.

Despite the concession of a 'C' veneer, SAC is a declarative language, free of side-effects, which borrows only a functional subset of C; hence global variables and pointers are absent. The SAC assignment statement, meanwhile, is mere *syntactic sugar*, implemented by a translation to a nested let expression upon compilation. Loop constructs, similarly, become tail-recursive functions[10]. Type declarations are implicit except for those specifying function parameters, and garbage collection is also provided. Another notable feature of SAC permits multiple function return values.

The essential feature of SAC, however, concerns its first class support for arrays. Distinctively, SAC does not require the rank of an array to be known at compile time, and whereas APL and other array languages commonly provide only a fixed set of operations, SAC allows the user to define fully shape-polymorphic, efficiently *composable*, functions and operators;

[9] An indexed array of arrays of indexed arrays is also possible.

[10] Note that the tail-recursive functions may become loops once again after optimisations on the intermediate representation are completed.

using a bespoke construct known as the *WITH-loop*. Efficient *vertical*, and *horizontal* composition of WITH-loops is handled by the SAC compiler using WITH-loop folding, and WITH-loop fusion respectively. A third crucial step, is WITH-loop scalarisation: an optimisation involving the elimination of temporary structures from nested WITH-loops.

Furthermore, such function definitions may be *specialised*, and hence optimised, by overloading a narrower type specification. Every value in SAC is an array; including scalars, which are simply arrays with a rank of zero. Such foundational concepts reveal the influence of Lenore Mullin's work on array formalism from the late 1980s. The core SAC `sel` function, is essentially the ψ function from her dissertation, "A Mathematics of Arrays" (Mullin, 1988).

The SAC type system is though enriched by a practical application of the `typedef` keyword. Unlike 'C', the result is not an alias, but a new type; so providing additional opportunities for function overloading. A simple example is the alternative ambiguity between a complex number and a short vector. Note that there is no *element* polymorphism in SAC. Specifically, no single, generic definition for a function can be defined. Overloading can compensate somewhat.

From the outset, the SAC language has been designed with parallel processing in mind, though mainly of a shared-memory, homogeneous SMP; and latterly, multicore variety. SAC has though recently targeted heterogeneous GPU architectures via a CUDA back-end (Guo et al., 2011). The absence of a stack in CUDA however necessitates that no function calls are present within the SAC WITH-loops which provide the sites for parallelisation.

2.4.5 Vector Pascal

Vector Pascal is an extension of Niklaus Wirth's strongly typed, block-structured Pascal language (Wirth, 1971), composed of elements from ISO Standard Pascal, ISO Extended Pascal, and Borland Turbo Pascal (Cockshott and Renfrew, 2004). The language incorporates a high-level array semantics similar to APL, J and Fortran 90, and initially was purposed to provide a portable, high-performance, data-parallel interface to target SIMD architectures.

All existing Pascal arithmetic operators, and assignment, are available for use in an element-wise fashion between either two conforming arrays, or an array and a scalar value. Such operators are also valid between arrays of differing rank, when the bounds of the lower rank array match the rightmost ranks of the other. Array sectioning in Vector Pascal adopts solely the contiguous range specification of Algol 68 (Tanenbaum, 1976, pages 170-171); there

referred to as *trimming*, and *slicing*. Reduction operations are also included for each of the binary operators.

Vector Pascal reinstates the conditional expression; previously ousted from Pascal by Wirth (Cockshott and Renfrew, 2004). The expression allows a data-parallel assignment to multiple arbitrary array sections, akin to Fortran 90's `where` statement. For example, the assignment statement on array `arr` in Listing 2.17 avoids a division by zero.

```
arr := IF arr = 0 THEN foo
       ELSE 10 DIV foo
```

Listing 2.17: The Pascal assignment statement.

Array reorganisation permits both element permutation within individual array ranks, and also the permutation of the ranks themselves. In the former case, the language provides a distinctive syntactic form to access otherwise implicit array induction iterators: `iota` *i* returns the *i*th implicit array index[11]. With a rank 2 array, `iota 0` and `iota 1` may then be used in a manner similar to the familiar `i` and `j` loop iterators of Fortran or C. The example in Listing 2.18 declares a zero-based 2D array, `mat2D`, and uses `iota` to populate it with consecutive integers, starting from zero.

```
program mat;
const size = 4;
var mat2D:array[0..size-1,0..size-1] of integer;
begin
  mat2D := iota 0 + size * iota 1;
end.
```

Listing 2.18: The Vector Pascal iota operator.

A compiler for Vector Pascal exists for a variety of x86 architectures and associated SIMD ISAs including MMX; 3D Now!; SSE1&2; and AVX. The MIPS architecture of the PlayStation 2 has also been targeted, as has the Opteron through multicore, and Sandybridge using hyperthreading. Most recently, a version has been developed that targets the array expressions on the PS3 (Gdura and Cockshott, 2011). The PS3 version automatically parallelises array expressions of sufficient size across up to four SPEs using a novel virtual SIMD machine (VSM) model. Through this abstraction, the aggregation of all SPEs corresponds to one or more VSM registers, and leverage the Vector Pascal compiler's existing register allocation apparatus. The VSM register file contains 8 virtual registers, hosted across the local stores of participating SPEs. Each register supports instructions either to load or store data

[11]The indices are implied by the left hand side of the assignment.

by DMA transfer; or perform computation. An interpreter on the PPE then partitions and aligns the array data; and also signals to the SPEs by mailbox which instruction should be performed. The SPE interpreter, meanwhile, runs constantly, checking for such messages from the PPE encoding the actions to be performed.

2.5 Partitioned Global Address Space Languages

Partitioned Global Address Space (PGAS) languages are a relatively recent development in parallel language research. PGAS languages seek to combine the scalability of a message-passing API, together with the simple and direct communication of a shared-memory interface such as OpenMP. A characteristic of such languages is the facility to define a *distributed data structure*, such as an array. While variables of such a type may be accessed from any participating thread, subsections of it are identified as local to subsets of the machine's participating thread team, so providing an opportunity for the compiler to optimise with respect to locality of reference. Reading and writing to PGAS memory is therefore an abstraction, later transformed into, for example, either a remote message transfer; or a local memory access; by the PGAS language compiler. Tasks, or subprograms, may also be given a locality. The programmer participates both in the creation and utilisation of affinity between threads, subprograms, and memory locations.

In common with parallel array languages like E♯, PGAS languages reject explicit message passing, and provide a global-view abstraction across distributed data structures. The *partitioned* aspect of such languages, however, may offer both enhanced programmability; and further information regarding data locality, which should ultimately lead to opportunities for compiler optimisations relating to processor affinity, and data caching.

Co-Array Fortran (CAF) is the first PGAS language examined. Originally an extension of Fortran, CAF has recently been passed for inclusion in the next revision of the Fortran standard (INCITS/J3, 2010), approved in September 2010, and referred to informally as Fortran 2008. Secondly, Unified Parallel C (UPC), is considered. UPC extends the 'C' language with explicit SPMD parallelism. Though highly distinct from Offload C++, UPC may be so compared by focusing on their common use of a dual address space, and respective `shared/private` and `__outer/__inner` qualifiers.

2.5.1 Co-Array Fortran

Co-Array Fortran (CAF) began life as F^{--} (Numrich, 1997), a parallel extension to Cray's Fortran 77 compiler. Despite the name, Cray's Fortran 77 compiler would at this stage support many features from Fortran 90, including array syntax, though not yet array sections. The target architecture for Cray's compiler was the Cray T3D. The F^{--} name was chosen firstly through a desire to avoid F^{++}, and an association with object orientation in general; and presumably C++ in particular. Secondly, the placement of the minus symbols in a superscript position was intended to suggest a relationship to tensor notation.

CAF adopts the SPMD model of parallel execution. Each program is composed of a number of asynchronously executed *images* arranged in a rectilinear grid. Each image has a unique integer identifier, and is consequently able to branch and select independently. A co-array is a distributed array, accessible in its entirety from all program images, and constructed from the aggregate of pieces stored local to each image. Communication between the distinct images of a program is represented as read or write accesses to co-array variables.

Co-arrays add one new piece of syntax to Fortran. In the first F^{--} paper (Numrich, 1997), copying element `y(i,j)` from a remote image identified by coordinate `(p,q)`, into `x(i,j)`, is represented by the code of Listing 2.19.

```
x(i,j) = y(i,j|p,q)
```

Listing 2.19: $F--$ syntax for a remote copy.

The subsequent F^{--} paper (Numrich and Steidel, 1997) modifies the notation for greater compatibility with Fortran 90, and the same assignment statement is as shown in Listing 2.20.

```
x(i,j) = y(i,j)[p,q]
```

Listing 2.20: Modern Fortran syntax for a remote copy.

When accessing the `y` array co-array, the traditional subscripts in parentheses index its dimensions, while the expressions within the square brackets, known as *cosubscripts*, index its *codimensions*. This second, bracketed, syntactic form is maintained in the current notation for co-arrays. A design principle of CAF requires that a section of code visibly free of brackets, is also free of inter-image communication. In line with this principle, pointers may not target remote images.

Co-arrays are now included in the latest version of the Fortran standard (Reid, 2010; INCITS/J3, 2010), informally known as Fortran 2008. Fortran 2008 defines a co-array [12] as a "data entity that has nonzero corank", and it follows that scalar and user-defined type variables may also be co-arrays. When declaring a co-array, the upper-bound of the last codimension is always omitted; with a `*` as placeholder. This is reminiscent of the *assumed size* arrays of Fortran 77, and allows the number of images to be deferred.

As with array indices, care must be taken to avoid coindices exceeding the imits set by the co-array *cobounds*. The declaration in Listing 2.21 is taken from the draft standard (INCITS/J3, 2010) and demonstrates a co-array where, with 16 images, the final absent codimension, induced to be the smallest value for which the product of the *coshape* extents is less than or equal to the number of images, is 4. Therefore `A(:)[1,4]` is a valid reference, as it specifies image 16, while `A(:)[2,4]` is not, as it specifies image 17.

```
REAL :: A(10)[5,*]
```

Listing 2.21: A co-array declaration.

Fortran 2008 also includes a number of new statements and intrinsic functions which support the user of co-arrays. These include functions which query a co-array's upper and lower cobounds; the number of images; and the current image index. There are also new synchronisation statements to provide full and partial barriers; critical sections; and locks, supported by the new derived type `lock_type`.

As co-arrays *are* now included in Fortran 2008, most Fortran compiler developers have now incorporated them into their roadmap. Nevertheless, the g95 compiler has had a network implementation available, minus locks, since 2009. Alan Wallcroft's CAF website [13] has also, for some years, provided a translator to convert a subset of CAF, into Fortran with OpenMP. Other prominent proponents of co-arrays are found in John Mellor-Crummey's team at Rice University, where development continues on a new, more expressive, co-array based language design (Mellor-Crummey et al., 2009) referred to as Co-array Fortran 2.0. A beta version of their translator is available.

2.5.2 Unified Parallel C

Unified Parallel C (UPC) is a PGAS language based on the 'C' language. Development of UPC began in 1999, at the Institute for Defense Analyses Center for Computing Science.

[12] The Fortran 2008 standard has in fact removed the hyphen from "co-array".
[13] See http://www.co-array.org.

Shortly thereafter, participation in UPC research diversified, alongside the formation of the UPC consortium. UPC working groups now exist at a number of American academic and industrial research centres. UPC compilers are available from a number of vendors including Cray; IBM; Hewlett Packard; GNU; and also as a joint project between the Lawrence Berkeley National Laboratory and the University of California at Berkeley.

The language design of UPC is a fusion of earlier research into 'C'-oriented parallel languages such as Split-C (Culler et al., 1993), AC (Carlson and Draper, 1995), and Parallel C Preprocessor (PCP) (Brooks and Warren, 1995). UPC is also influenced by pC++ (Bodin et al., 1993) and CC++ (Chandy and Kesselman, 1993). Like CAF, UPC adopts the SPMD execution model; presents a global view of distributed memory; and develops a notion of *affinity* between threads and data addresses. Each data element has affinity with one thread only. UPC is a superset of C99, however its parallel execution model would result in each thread of the program `main(){ printf("!"); }` printing an exclamation mark.

A new type qualifier, `shared`, is provided to identify data objects that are accessible to all participating threads. The constant integer values `THREADS` and `MYTHREAD` are also provided, and initialised by the UPC compiler. The number of threads participating in a program is provided by `THREADS`, and may be defined either statically or dynamically; for example through the `UPCC_FIXED_THREADS` environment variable. The `MYTHREAD` constant instantiates to a unique integer value in the range `0..THREADS` for each participating thread.

In Listing 2.22, line 4 declares two shared arrays: `a1` and `a2`. We will assume that 4 threads are used, and hence the `THREADS` constant is set to 4 throughout. The two arrays manifest themselves in 4 pieces, each local to a distinct thread. The affinity of each array element is assigned on a round-robin basis by default, and so each thread t has affinity for elements indexed by t and $t + 4$.

Observe that, unlike CAF, it is not syntactically obvious when remote data accesses will occur. The first assignment statement on lines 11 requires the remote transfer of data between portions of the shared arrays `a1` and `a2`. In contrast, the assignment on the following line 12, involves only the privately declared arrays `b1` and `b2`, each stored in its entirety local to each thread. The assignment therefore performs only a local read and write.

In 'C', an integer constant may not be used to define the size of an array. As an integer constant, the compiler-managed UPC constant `THREADS` also cannot define the size of a *private* array; such as `b1` or `b2` above. In contrast, `THREADS` ***must*** be used, once only, to define the size of a shared array. Also, shared variables may only be declared at file scope.

UPC provides two memory consistency models: strict and relaxed. The choice of con-

```
1  #include <upc.h>
2  #include <stdio.h>
3
4  shared int a1[THREADS*2], a2[THREADS*2];
5  int b1[4*2], b2[4*2];
6  shared int x;
7  int y;
8
9  int main(int argc, char *argv[])
10 {
11   a1[MYTHREAD] = a2[MYTHREAD+1];
12   b1[MYTHREAD] = b2[MYTHREAD+1];
13   if (MYTHREAD==3) {
14     x = 1;
15   }
16   upc_barrier;
17   y = x;
18   return 0;
19 }
```

Listing 2.22: Implicit remote access syntax in UPC.

sistency determines when the update of shared resources become visible to other threads. Under the relaxed consistency model, the UPC compiler may reorder the updates to shared memory references, so long as they *appear* to occur in the same order as the program code, from the perspective of the issuing thread. Under the strict consistency model, the same ordering restriction must be observable to *all* threads. If the shared resources of the entire translation unit should have the same memory consistency, the upc_strict.h or upc_relaxed.h header files may simply be provided to a `#include` directive. A finer grain of control may be obtained using the `strict` and `relaxed` type qualifiers alongside a `shared` variable declaration. The synchronisation statement `upc_fence` may also be useful in this context. The `upc_fence` statement is equivalent to a null strict access, and ensures that prior accesses to shared resources are complete before others are issued.

The shared scalar variable `x` is stored with affinity for thread 0. The if statement on line 13 allows thread 3 alone to update this variable, and so avoid a race condition. To guarantee that all threads see the effect of this update, a barrier statement is used at line 16; a barrier statement executes a `upc_fence` statement as its first action. Finally, the private variable `y` may safely be updated by all threads.

The round-robin method of assigning affinity to shared resource elements may be overridden by providing a positive integer block argument to an optional layout qualifier. Once more with 4 threads, the declaration on line 1 of Listing 2.23 will again locate two elements with every thread. On this occasion, however, an affinity is created between each thread t, and the two array elements indexed by $t * 2$ and $t * 2 + 1$.

```
shared [2] int a3[THREADS*2];
shared [2] int *pa3;
```

Listing 2.23: Configuring affinity in Unified Parallel 'C'.

Pointers in UPC may be shared or private in two senses. A UPC pointer may reside in a location which is shared or private; and may also target memory which is shared or private. There are therefore 4 valid UPC pointer configurations. Offload C++, in contrast, provides only 3: there is no *inner* pointer accessible in an *outer* context. UPC documentation does however strongly advise against the use of the analogue. Also distinct from Offload C++, UPC pointers may be cast between localities. This is, however, a consequence of a concept absent from Offload C++: shared data. A shared UPC array is composed entirely of sections, each with affinity to one distinct thread. It follows that the only legal cast between localities in UPC is one from shared to private, and is only well defined when the target has affinity with the local thread.

Pointer arithmetic in UPC is also distinctively allied with the "blocked" layout qualification shown in Listing 2.23. Incrementing the value of the pointer declared on line 2, suitably initialised to the first element of `a3`, will traverse its entirety, as if indexed by an integer ranging from `0` to `THREADS*2-1`. Pointers may also have a different blocking layout to their targets, allowing more complex traversals.

2.5.3 MPI Virtual Process Topologies

MPI (Message Passing Interface Forum, 2009) includes the notion of a process topology. Graph and grid (cartesian) topologies are given particular emphasis. While an MPI topology could be considered to address similar concerns of the PGAS languages, in practise it is more often used to simplify the expression of communication between nodes. For example, an MPI topology may specify that a node *up*, *left*, or *southeast*, of the current node has some meaning. The expectation of a performance improvement through the use of MPI topologies is however non-standardised, and is instead rather ad hoc: the MPI standard does not discuss physical mapping, and allows vendors to ignore the mapping of processes to topologies. To gain in performance terms from the use of MPI topologies typically requires that both the machine, and the MPI implementation, are purchased from the same, motivated, vendor.

Chapter 3

Methodology

Chapter 1 introduced the idea at the heart of the research, which is to evaluate the design and performance implications of pursuing a collection-oriented language approach to heterogeneous multicore parallelism. This chapter will present the methodology adopted to advance these aims.

Architecturally, the research aims to target heterogeneous multicore, and for this the STI Cell Broadband Engine (CBE) was selected. At the 2011 Tokyo Games Show Sony stated that it has sold over 50 million PlayStation 3 consoles, each of which includes a Cell processor. The CBE has also demonstrated its applicability to HPC as part of IBM's Roadrunner (Barker et al., 2008), which in 2008 became the first supercomputer to achieve a sustained Linpack performance of 1 petaflop; while the June 2011 Top500 list still ranks Roadrunner within the top 10 (Meuer et al.).

The programming language chosen is a well-defined subset of modern Fortran, and is therefore relevant to the high performance and scientific computing community. Like Fortran, the 'F' language includes first-class support for arrays, and array expressions, with which the intention is to provide an implicit parallel interface, while also demonstrating scalable and high performance. In this chapter, the 'F' language will be introduced, along with associated programming idioms required to facilitate good performance.

Codeplay's Offload toolkit includes support for an extended form of the C++ language featuring integrated asynchronous thread control; advanced pointer types, assigned a static locality; and an additional function qualifier compatible with the C++ overloading mechanism. The Offload C++ compiler targets heterogeneous architectures including specifically the PlayStation 3.

The ‘F’ language is translated into Offload C++ using a custom compiler developed for the task, E♯. The Haskell programming language was selected as the development language for E♯ both for the relative safety of its strong, static type system, and for the native high-level language facilities such as polymorphism and pattern matching, which have made it a popular language among compiler authors. As far as possible, the C++ which is output by E♯ should rely on runtime libraries. One goal is to make as much use as possible of the Fortran runtime libraries provided by each compiler vendor. This is particularly significant when we approach the interface to the non-standard ABIs used by the numerous dope vector incarnations. The use of integer template arguments to represent locality in this area will demonstrate a highly distinctive, and ultimately intuitive interface to pointer locality.

Evaluation is framed by a small suite of realistic applications taken from the realm of computational science, and includes mathematical, physical, and financial simulations. As the method of constructing and performance-tuning these programs within the array style is novel, each is presented in detail in the form of a case study.

3.1 The Cell Broadband Engine

Development of the processor now known as the Cell Broadband Engine (CBE)[1] began with approval from the CEOs of Sony Computer Entertainment Incorporated (SCEI), Toshiba, and IBM (Kahle et al., 2005). High-level architecture discussions in Japan followed in the summer of 2000 before a joint investment of $400,000,000 led to the construction in 2001 of the STI (SCEI-Toshiba-IBM) Design Center in Austin, Texas. Program objectives for the new chip focused on attaining the responsive, performant processing of multimedia. The Cell project also sought applicability to platforms beyond gaming or multimedia, and to this end was developed with an open, Linux, software development environment. Aiming for a release in 2005, an entirely new architecture was considered infeasible, and consequently the core of the CBE became a modification of an existing IBM Power Architecture series.

According to Flynn’s taxonomy, the architecture of the CBE may be classified as MIMD (Flynn, 1966). The first generation CBE (Kahle et al., 2005; Hofstee, 2005) includes a 64-bit RISC PowerPC processor element (PPE), augmented by 8 accelerators: SIMD (Flynn, 1966) synergistic processor elements (SPE). The PPE and SPE are each clocked at 3.2GHz. With both capable of dual-issue, 128-bit vector instructions, a theoretical peak of 230 (9 x 25.6) GFlops is obtained. As shown on Figure 3.1, along with a memory controller, and bus interface controller, the PPE and SPEs are interconnected through a coherent on-chip ele-

[1] Other abbreviations for the CBE include the Cell; the Cell B.E.; and the Cell Processor.

ment interconnect bus (EIB) with 96 bytes/cycle bandwidth. Rambus XDR DRAM memory delivers 12.8 Gb/s per 32-bit memory channel giving a total bandwidth of 25.6 Gb/s.

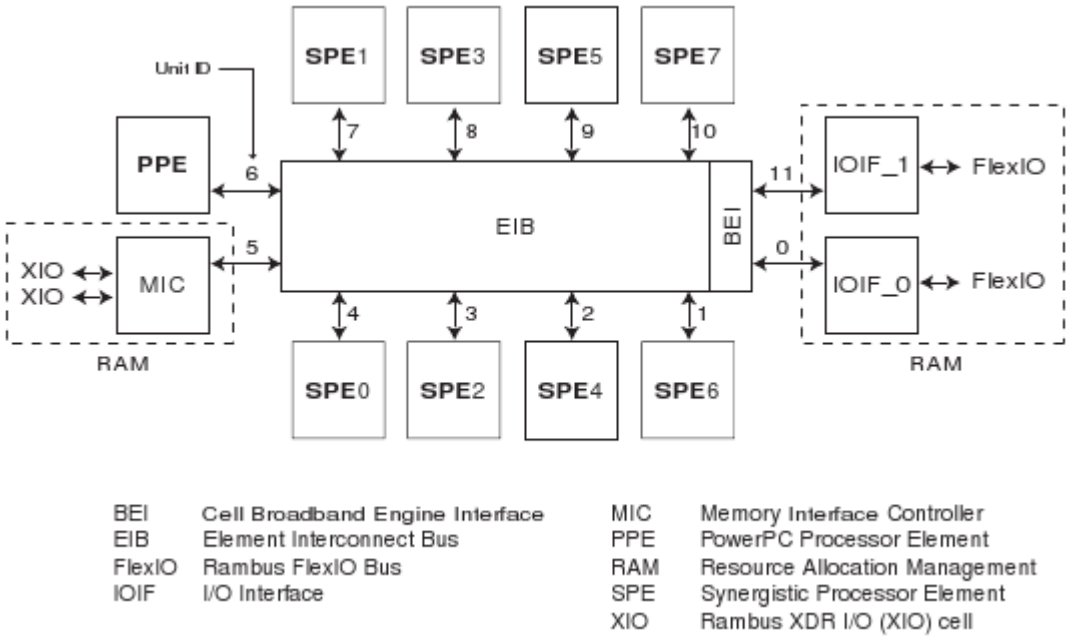

Figure 3.1: Cell Broadband Engine Overview (Image taken from IBM Systems and Technology Group (2007))

The PPE has 32KB first-level instruction and data caches, and a 512KB second-level cache, and implements IBM's "Amazon" PowerPC AS ISA. PowerPC support for virtualisation and large page sizes is inherited, enabling the PPE to support multiple operating systems. The PPE provides dual, in-order instruction issue and is fed by two simultaneous hardware threads. The PPE has 32 general purpose registers, and 32 floating-point registers; both are 64-bit. A vector multimedia extension unit (VMX) also provides the PPE with its own set of 32, 128-bit wide, SIMD registers using an Altivec SIMD ISA.

The 8 SPEs are composed of 256KB of local store SRAM; a coherent, asynchronous DMA engine; and a 128-bit wide SIMD symmetric processing unit (SPU) equipped with a large, 128 entry register file. Each SPE has a memory flow control unit which executes DMA commands, of which up to 16 may be outstanding. The DMA operations issued by an SPE include a *get*, and *put*, which involve both a local and a remote address. The remote, virtual address may either reference the local memory of another SPE; or main memory. The EIB facilitates SPE DMA operations to or from main memory which are fully cache coherent. Distinct from the PPE, the SPE has no cache, and its own ISA; though the SPE does operate across the same range of integer and floating-point data element widths (2 x 64-bit, 4 x 32-bit, 8 x 16-bit, 16 x 8-bit, and 128 x 1-bit). Each SPE can issue two instructions per cycle, with simple fixed-point operations taking two cycles, and single-precision floating-

point operations taking six. Two-way *double-precision* operations are limited to a maximum issue rate of one per seven cycles. The SPE has no hardware branch prediction, instead providing a similar mechanism via a software branch hint instruction.

The most prevalent use of the CBE is within the Sony PlayStation 3™ (PS3) games console, of which over 50 million have been purchased in 2011. To increase fabrication yields, the CBE in the PS3 is equipped with only seven SPEs. Linux developers will also discover that one of the remaining seven SPEs is unavailable due to a reserved status assigned to it by the hypervisor (IBM Systems and Technology Group, 2007, Chapter 11).

The Cell processor is one of a collection referred to as the Broadband Processor Architecture series, and have regularly appeared in systems other than the PS3. An early Cell system, the IBM BladeCenter QS21, used two of the 90nm first-generation CBEs; while IBM's later QS22 employs a pair of second-generation Cell processors with enhanced double-precision performance: the PowerXCell 8i processors. More recently, Toshiba's SpursEngine SE1000 coprocessor (Hayashi, 2008) has incorporated four SPEs clocked at only 1.5Ghz; as seen in their Qosmio laptops[2]. Leadtek have in 2010 launched an accelerator card, the WinFast HPVC1111 SpursEngine x4, using four SpursEngine SE1000 processors across PCI Express x4. The second-generation Cell was also incorporated by IBM on the commissioned U.S. Department of Energy's Los Alamos National Laboratory Roadrunner (Barker et al., 2008) supercomputer. Roadrunner is an Opteron cluster using the 65nm PowerXCell 8i as accelerators. The PowerXCell 8i benefits both from reduced power consumption, due to the reduced die size; and improved double precision performance, up from 21 GFlops to 108.5 GFlops. Recently, a move to 45 nm SOI has been used in a redesign of the PS3 known as the *PS3 Slim*. This 45nm CBE draws 38 percent less power than the 65nm version; and less than half that of the original 90nm design (Takahashi et al., 2008).

As was discussed at length in Chapter 2, software development for the CBE routinely centres on the 'C' and Fortran programming languages. A reasonable point of entry is the installation of the IBM Software Development Kit for Multicore Acceleration (IBM SDK). Custom versions of the GNU Compiler Collection (GCC) toolchain, developed at the Barcelona Supercomputing Centre as part of their *Linux on Cell* program, are also included as optional components within the IBM SDK.

[2]The Qosmio SpursEngine is referred to as the Quad Core HD processor.

3.2 Offload C++

The Offload distribution for PS3 (Cooper et al., 2010; Russell et al., 2010) is a suite of tools for C++ development on Microsoft Windows, targeting the Cell Broadband Engine (CBE) processor; as a component of the Sony PlayStation 3 (PS3). The suite includes the Offload C++ compiler; an Eclipse-based IDE; and an interactive source-level debugger. The Offload C++ language design, including the patented call-graph duplication technology, is a product of Codeplay Software Ltd.

The Offload C++ language is used as the target language by the E♯ source-to-source compiler. The output of the Offload compiler is also a medium-level language target: 'C'. This is itself compiled by a system compiler; which under PS3 Linux will be GCC. Further details on the aggregate system of compilers is described in Chapter 4.

3.2.1 Launching Threads

The first extension to the C++ language provided by Offload C++ enables simplified thread creation via the use of the *offload block*. Lexically this new construct is identified by the `offload`[3] keyword, followed by a compound statement, or block. An offload block prescribes that the sequence of statements within its scope should be processed in a new thread of execution running on an SPU [4]. As part of a statement, the offload block in Listing 3.1 occurs synchronously, and results in the value stored by `x` being incremented by one.

```
offload { x = x + 1; };
```

Listing 3.1: Incrementing a globally scoped variable.

This code demonstrates the *synchronous*, or *blocking* form of the offload block. The `x` variable is defined outwith the lexical scope of the offload block, and may be described as a *free* variable within. Semantically, this block has the same meaning as if the `offload` keyword was removed. Although the cost of launching a thread for such a trivial assignment statement is prohibitive, it is nevertheless possible that, even in the synchronous form, a floating-point intensive offload block could reduce the elapsed real time of the surrounding code.

Grammatically, the offload block is a first-class *expression*. In Listing 3.1 the offload block expression is terminated with a semicolon; so forming an *expression statement*, with no vari-

[3] Each keyword in Offload C++ may optionally be preceded by double underscores, to avoid name collisions.
[4] More colloquially, the computation enclosed within an offload block is said to be *offloaded*.

able name bound to the expression result. Hence, the program state is altered solely through the side effect of modifying the `x` variable. This is akin to a C++ expression statement such as `a - b++;`.

The *value* of an offload block expression is a handle to a thread scheduled to execute the compound statement. A variable of type `offloadThread_t`, provided by the associated Offload C++ runtime library, can be used to store this value. In this second form, the thread associated with an offload block can run asynchronously, and must be forced to complete its execution by a call to `offloadThreadJoin`, with an `offloadThread_t` expression as its sole parameter.

The *synchronous* offload block used in Listing 3.1 can therefore be replicated using the *asynchronous* offload block expression in Listing 3.2, and demonstrates that an offload block is also a first-class expression.

```
offloadThreadJoin( offload { x = x + 1; } );
```

Listing 3.2: An offload block expression.

This asynchronous form of the offload block can therefore be used to split a calculation between the main PPE host thread, and one or more SPE threads. The example in listing 3.3 applies this approach to multiple *offloaded* threads. The first `for` loop schedules the threads for launch, subsequently joined at the second. Each thread is tasked with a call to the `calc` function, which is automatically compiled for both architectures. Note that a number of threads greater than the number of SPEs may be scheduled, though only six will execute in parallel. The PPU also makes a nominal contribution at line 15, and thus seven threads may in total operate in parallel.

The presentation so far is somewhat reminiscent of the approach taken by POSIX Threads (The Austin Group, 2010), though using a block rather than a function call at the threading interface. Intel's Cilk (Blumofe et al., 1995) language for homogeneous architectures is also comparable, though unlike Cilk's `cilk_join` construct, the corresponding Offload C++ `offloadThreadJoin` may also be called *after* the function creating the threads has returned.

3.2.2 Pointer Locality

In this section we will examine the static attribution of a memory space qualifier to each pointer within the extended C++ type system of Offload C++. It will be convenient to qualify memory addresses located in PPE main memory as *outer*; and to those in SPE local memory

```
1  #include <liboffload>
2
3  int i;
4  double data[100];
5
6  void calc(double *d) { *d = *d / 2.0; }
7
8  void ol() {
9
10   offloadThread_t handles[99];
11
12   for (i = 0; i < 99; i++)
13     handles[i] = offload { calc(data + i); };
14
15   calc(data + 99);
16
17   for (i = 0; i < 99; i++)
18     offloadThreadJoin(handles[i]);
19 }
```

Listing 3.3: Performance parallelism using multiple offload blocks.

as *inner*. Hence a pointer will either have inner or outer *locality*. Computations executed by the PPE, or SPE, will also be described as occurring within, respectively, outer and inner *contexts*, and correspond similarly to code executed outside or inside the dynamic scope of an offload block[5].

We have already observed that free variables may be referenced within the scope of an offload block. In the heterogeneous multicore environment of the CBE, a free variable must be transferred by DMA between the main memory of the host, and the local store of the SPU, to ensure behaviour compliant with the program specification.

Pointer references are more complex and are treated differently by Offload C++. Naïvely dereferencing a free pointer variable would result in a runtime error, or at best garbage, as the memory location referenced will not exist in the address space of the executing SPU thread. Offload C++ implicitly generates the additional data movement instructions to ensure the result of such pointer dereferencing, within an offload block, is equivalent to an otherwise identical compound statement, with the `offload` keyword removed.

A pointer declaration occurring outwith the dynamic scope of an offload block creates a pointer with a conventional *outer* locality; a similar declaration made within the dynamic scope of an offload block has an implicit *inner* locality. Listing 3.4 makes use of an *explicit* `outer` pointer qualifier, within an offload block, to increment an integer in PPE main mem-

[5]The Embedded C standard (WG14, 2006, pages 37-39) also includes named address spaces; though without C++ features such as polymorphism and function overloading.

```
1  int i0 = 1;
2
3  void succ()
4  {
5    offload {
6      int i1;
7      int outer *po;
8      po = &i0;
9      i1 = *po;
10     i1++;
11     *po = i1;
12   };
13 }
```

Listing 3.4: Using an outer pointer in an inner context.

ory. The executable statements of the offload block begin on line 8 with the only legal form of pointer assignment: that occurring between pointers of equal locality. A cast or type conversion between differing pointer localities is also impossible. In this case both have outer locality. Line 9 then assigns a variable located in the inner memory space, `i1`, with the value targeted by an outer address, and so causes an implicit DMA transfer into the SPE local store. The `i1` variable is then incremented, before its value is transferred in the opposite direction, into PPE main memory, by the assignment of line 11. A significant consequence of such DMA transfers being implicitly managed by the compiler is not only the brevity, but the removal of a class of errors due to miscalculating the size in bytes of otherwise mandatory calls to low-level macros such as `mfc_get` or `mfc_put` (IBM Corporation, 2007a).

Outer and inner contexts are each identified with an integer value of 0 and 1 respectively. This value may be referred to as the offload *depth* of a context. The `outer` keyword for pointer qualification is in fact defined by macro substitution using its corresponding depth value: 0. The definition provided by the Offload C++ runtime library is equivalent to that of Figure 3.5, and employs both the `__declspec` keyword from the Microsoft-specific extended attribute syntax; and a new Offload C++ modifier, parameterised by depth: `__setoffloadlevel__`.

```
#define outer __declspec(__setoffloadlevel__(0))
```

Listing 3.5: Macro definition of the outer locality qualifier.

As a convenience for the user, a pointer declared in an inner context, yet initialised to an outer-located address, will be recognised as having *outer* locality. The code of Listing 3.6 therefore declares an outer pointer variable named `p1`.

The `outer` keyword may also be omitted within a cast. As demonstrated in Listing 3.7, the

```
int i0;
offload {
  int *p1 = &i0;
};
```

Listing 3.6: Implicit outer pointer qualification from initialisation.

cast succeeds despite eschewing the more verbose `(B outer *)`.

```
A *pA;
offload {
  B outer * p;
  p = (B *)pA;
};
```

Listing 3.7: Implicit locality in casts.

3.2.3 Template Declarations

Offload C++ provides a built-in integer constant, `OFFLOAD_DEPTH`, This constant intrinsic is set by the compiler to `0` when appearing in an outer context, and `1` when it appears in an inner. As a template argument, `OFFLOAD_DEPTH` may be used to invoke different behaviour based on the offload depth of the calling environment. The `tdepth` function of Listing 3.8 uses `OFFLOAD_DEPTH` as a default template argument, and so may be called in either context using `tdepth<>()`.

```
template <int DEPTH=OFFLOAD_DEPTH>
int tdepth() { return DEPTH; }
```

Listing 3.8: A depth inquiry function with default template argument.

When larger function definitions are involved, it may be possible to predicate the execution of a few statements upon the value of the `OFFLOAD_DEPTH` constant, and so promote code reuse. This approach is unsuitable when such branches contain functions or intrinsics specific to a single architecture.

The default assignment of the `outer` qualifier to pointers declared in an outer context also applies to the members of user-defined types such as structs or classes. Therefore, in listing 3.9 the pointer member of both `a0` and `a1` will have `outer` locality.

```
struct X1 {
  int *m_p0;
};

void foo() {
  X1 a0;
  offload {
    X1 a1;
    a1.m_p0 = a0.m_p0;
  };
}
```

Listing 3.9: Default outer locality of member pointers.

To obtain more flexibility regarding the *inner* and *outer* attributes of pointer members, C++ templates may be used to create user-defined types parametrised by the fully qualified pointer types involved. Listing 3.10 demonstrates the use of an outer pointer template argument, the default, on line 8; and an *inner*[6] pointer template argument on line 11.

```
1  template <typename T>
2  struct X2 {
3    T m_p;
4  };
5
6  void foo() {
7    int i0;
8    X2<int outer *> a0;
9    offload {
10     int i1;
11     X2<int inner *> a1;
12     a0.m_p = &i0;
13     a1.m_p = &i1;
14   };
15 }
```

Listing 3.10: Explicit locality in pointer template arguments.

3.2.4 Call Graph Duplication

Each function called from within an offload block will implicitly be duplicated for the SPU by the Offload C++ compiler. Functions with reference or pointer parameters will be compiled once for *each* configuration of argument qualifiers found at a call site. Consider the code excerpt in Listing 3.11.

[6]The syntax of the `inner` pointer qualifier is introduced in the following Section 3.2.7.

```
int bar(int &a, int &b) { return a * b; }

int main() {
  int i0;
  bar(i0,i0);
  offload {
    int i1;
    bar(i0,i1);
    bar(i1,i0);
    bar(i1,i1);
  };
}
```

Listing 3.11: Call graph duplication.

Here, the `bar` function is compiled into four distinct forms. The first corresponds to the sole call to `bar` made in an outer context, and is compiled for the PPE host architecture. The subsequent three calls to `bar`, made within the inner context of an offload block, induce the three further versions of `bar`, targeting the SPE architecture.

This approach to duplication stands in contrast to the simpler method employed by the IBM Octopiler (Eichenberger et al., 2006). The octopiler does not take pointer locality into consideration, and consequently a maximum of one duplicated function instance may arise. All pointer arguments to duplicated functions default suboptimally to the equivalent of outer locality, with attendant DMA transfers inserted as required.

If the four duplicated functions of Listing 3.11 were constructed by hand, their function declarations would be as shown in listing 3.12. This code demonstrates a further use of the `offload` keyword: as a *function qualifier*. This can be used both to identify functions to be compiled for use *only* in an inner context; and as a method of overloading an existing PPE function definition explicitly.

```
        int bar(int       &a, int       &b);
offload int bar(int outer &a, int       &b);
offload int bar(int       &a, int outer &b);
offload int bar(int       &a, int       &b);
```

Listing 3.12: Declarations of functions overloaded by offload attribute.

Idiomatic overloading of this kind can be useful for optimisation, as with the application of SIMD intrinsics. Such intrinsics may be unavailable on the PPE host architecture, and a runtime test predicated by the offload depth would be of little help, requiring still a PPE version of the relevant intrinsic. As a simple example, consider the function `f` in Listing 3.13 which is defined by two completely distinct functions, according to whether or not each

shall run in an *inner* or an *outer* context. This is an extension of the C++ overload resolution mechanism, according to the presence or absence of the `offload` function attribute.

```
        int f() { return 0; }
offload int f() { return 1; }
```

Listing 3.13: Overloading based on the offload qualifier of a function.

3.2.5 Translation Units

Call graph duplication as so far discussed, assumes that the function is defined in the same translation unit as the offload block. Functions defined in separate translation units must have the header of their definition succeeded by a *duplication obligation*. Often this will be as simple as adding the GNU attribute specifier `__attribute__((duplicate))` to a definition. Reference arguments would however default to have the `inner` qualification; which may not always be desired. If, for example, the `mult` function of listing 3.11 was defined in a separate translation unit, optional arguments to `duplicate`, to specify each individual function signature, would be required; as shown in Listing 3.14.

```
int mult(int &a, int &b)__
  attribute__((
    duplicate(int (int outer &, int &),
              int (int &, int outer &),
              int (int &, int       &)
             )
           ))
{ return a * b; }
```

Listing 3.14: Forcing function duplication with the GNU attribute specifier.

3.2.6 Offload Block Parameters

The offload block accepts two optional comma separated parameter lists. The first is delimited by round brackets and lists the names of local, stack-allocated, variables declared outside the scope of an offload block, yet used within it. These are referred to as *outer stack locals*. The performance of code using stack variables within an offload block can be improved by so listing their names; as they will be pre-emptively *pushed* on to the inner memory space. With the *asynchronous* form of the offload block, *all* outer stack locals must be listed as parameters, and can only be used in a read-only capacity. This is due to the possibility that the

life of the stack variables may be shorter than the offload block. In Listing 3.15, the offload block lists two stack arguments, and joins the thread launched by `launch`, in the call-graph parent function, `land`.

```
int g0;

void launch(offloadThread_t &h, int i0) {
  int j0 = 2;
  h = offload (i0,j0) { g0 = i0+j0; };
}

void land() {
  offloadThread_t h;
  foo(h,1);
  offloadThreadJoin(h);
}
```

Listing 3.15: Stack variables and asynchronous offload blocks.

The second optional list of parameters to an offload block specifies the *function domain*: a comma-separated list of cast expressions representing function names; delimited by square brackets. A function domain solves the otherwise intractable compilation problem of identifying which function or virtual method a function pointer is targeting, to ensure it is compiled and loaded onto the memory of the *inner* SPE machine architecture in advance of its application. The `virt_test` function of Listing 3.16 demonstrates the use of an offload block with a function domain.

```
struct VObj {
  virtual void vmeth()      {}
  virtual void vmeth(int *) {}
};

void virt_test()
{
  X a, *pa = &a;

  offload [VObj::vmeth] {
    int i1=1;
    pa->vmeth();
    pa->vmeth(&i1);
  };
}
```

Listing 3.16: Function domains as offload block arguments.

The offload block in Listing 3.15 uses a function domain which demands all overloads of the `VObj::vmeth` function are made available to the SPE. A restricted set of overloaded functions can instead be specified by providing only *their* signatures in the function domain.

For example, assuming the previous definition for `VObj`, the offload block in listing 3.17, is permitted only to call the `vmeth` method defined for an integer pointer argument.

```
void virt_test2()
{
  X a, *pa = &a;

  offload [(void(VObj::*)(int *)) &VObj::vmeth] {
    int i2=2;
    pa->vmeth(&i2);
  };
}
```

Listing 3.17: Restricting function domains to specific function signatures.

By default the functions listed in each domain are compiled with *inner* pointer, or reference, argument types. If a parameter should instead accept outer argument types, this must be specified explicitly within the function domain by flagging the relevant pointer types with the `outer` keyword. The first offload block of Listing 3.18, which again uses the `VObj` struct, demonstrates the application of the `vmeth` to a pointer with outer locality.

```
int g1=0;

void virt_test3()
{
  X a, *pa = &a;

  offload [(void(VObj::*)(int outer *)) &VObj::vmeth] {
    pa->vmeth(&g1);
  };
  offload [(void(VObj::*)(int *)) &VObj::vmeth this] {
    int i3=3;
    X b, *pb = &b;
    pb->vmeth(&i3);
  };
}
```

Listing 3.18: Using the outer keyword within a function domain.

Methods listed within a function domain are compiled by default with a `this` pointer of outer locality. To be able to call virtual methods on *inner* object pointers, the relevant signature in the domain list should be followed by the optional `this` keyword. The presence of the `this` keyword in a function domain signifies that the corresponding method should have an inner `this` pointer type. The last offload block in Listing 3.18 demonstrates a function domain specifying such an inner `this` version of the `vmeth` method, subsequently applied via the locally declared `pb` pointer.

3.2.7 Explicit Inner Pointers

It is often useful to explicitly set the locality of an Offload C++ pointer, and the built-in `outer` qualifier can be included in the syntax of a pointer declaration when required. A pointer can also be assigned to an *inner* address space, using the GNU attribute syntax, along with the Offload-specific, `__setoffloadlevel__` property. Listing 3.19 demonstrates the declaration of such a pointer.

```
void __declspec(__setoffloadlevel__(1)) *pv;
```

Listing 3.19: Declaring a pointer to address space 1 - an *inner* pointer.

It is convenient to define a 'C' preprocessor macro to reduce the verbosity of this syntax. Listing 3.20 defines such a macro. Also listed is a concise single-parameter macro to specify either locality with a 0 or 1 integer argument.

```
#define inner __   declspec(__setoffloadlevel__(1))
#define inout(D) __declspec(__setoffloadlevel__(D))
```

Listing 3.20: Macros to configure a pointer's locality.

A productive application of inner pointers will be discussed in Section 5.4. In contrast, valid usage of inner pointers in an *outer* context is severely curtailed; restricted to declaration, assignment and basic arithmetic. For example, the call to the `in` function on line 6 of Listing 3.21 fails to compile; the *inner*-qualified formal pointer argument will type check only with the perverse call to `getI` on line 7.

```
1 void in(void inner *p) {}
2 void inner *getI()     { return NULL; }
3
4 int main(int argc, char *argv[]) {
5   void inner *pv;
6   in(pv);
7   in(getI());
8 }
```

Listing 3.21: Type checking against formal `inner` pointer argument.

In a similar vein, a call to the `assgn` function in Listing 3.22 will compile when called from an outer context; regardless of the depth of either argument.

```
void assgn(void *p1, void *p2) { p1 = p2; }
```

Listing 3.22: Permissive unqualified pointers.

3.3 Fortran and F

Fortran is a compiled, statically-typed, imperative programming language popular within the domain of computational science and high performance computing. Fortran[7] originated in 1954 as an internal IBM project led by John Backus, and originally targeting the IBM 704 mainframe computer (Backus et al., 1957). Subsequent compiler and language design iterations targeted multiple IBM machines, leading to versions of Fortran developed by third parties; and ultimately to participation in an ANSI and ISO standardisation procedure which today governs the regular publication of Fortran standards. The most recent standard, informally referred to as *Fortran 2008*[8], was published in October 2010 as ISO/IEC 1539-1:2010. The following tongue-in-cheek quote from Tony Hoare in 1982 reflects on the longevity and evolution of Fortran:

> "I don't know what the language of the year 2000 will look like, but I know it will be called Fortran." Backus (1998, page 73)

The 'F' programming language is an informally specified subset of *Fortran 95* designed with the intention of providing a lightweight version of Fortran 90, free of the requirement to support 40 years of language artifacts. The primary motivation of the language design was to create an introductory-level Fortran-based language. 'F' is nevertheless an adequate general-purpose language. Furthermore, any Fortran compiler will compile a program conforming to the 'F' language standard. Specific compiler support for 'F' is, however, rare. The G95 compiler (Andy Vaught, 2009) is an exception, providing the command line switch, `-std=F`, to enforce adherence to the standard. The canonical reference for the 'F' language is Metcalf and Reid (1996).

3.3.1 Modules and Variables in 'F'

Listing 3.23 demonstrates a simple, modular, 'F' program to display a greeting, "Hello", to the standard output. The program is constructed from two *program units*: the first program

[7] The name "Fortran" is formed from the phrase: *Formula Translation*.
[8] *Fortran 2003* is the most recent Fortran standard prior to *Fortran 2008*.

unit, `m`, is a *module*; and the second, `p`, is the *main program*. The main program represents the entry and exit points of a program, and in this example, performs the single action of calling the `greet` subroutine, which itself calls the built-in Fortran `print` statement. The `greet` subroutine is defined within the `m` module, where it is also given *public* access at line 3. Both a module's data and its procedures may be given public *or* private accessibility. A program unit which has specified a module name in a *use statement*, such as `m` at line 15, may then access or call public members of the relevant module.

```
1  module m
2
3    public :: greet
4
5    contains
6
7    subroutine greet()
8      print *, "Hello"
9    end subroutine greet
10
11 end module m
12
13 program p
14
15   use m
16   call greet()
17
18 end program p
```

Listing 3.23: A simple modular 'F' program.

As in 'C', the 'F' language requires that variable declarations precede any executable statements. The program in Listing 3.24 declares `i`, a scalar integer variable, and `a`, a two-dimensional real-valued *array*. Array declarations use the `dimension` attribute, including a parenthesised integer list, `(3,2)` here, which specifies the extent of array elements in each dimension. This list of extents is referred to as the array's *shape*. The dimensionality of an array may also be referred to as its *rank*, and is equal to the length of its one-dimensional shape array, or *vector*. Array `a` therefore has a rank of 2, and a shape of `(/3,2/)`. The `reshape` function is one of many predefined, or *intrinsic*, functions supplied by every compliant Fortran compiler. In its simplest invocation, `reshape` will return a new array, with the elements of its first argument, and the shape of its second. The call to `reshape` in Listing 3.24 is used to initialise the six elements of array `a`. The arguments to `reshape` demonstrate the syntax for array literals, or *array constructors*, which must be of rank 1.

The executable portion of Listing 3.24 begins on line 5 with an assignment from an expression involving both the `size` and `shape` intrinsic functions. The `shape` function returns a vector describing the shape of its argument, while `size` returns the extent of the array first

```
1 program p2
2
3   integer               :: i
4   real, dimension(3,2) :: a = reshape((/1,2,3,4,5,6/),(/3,2/))
5   i      = size(shape(a),1)
6   a(1,1) = i
7   print *, a
8
9 end program p2
```

Listing 3.24: Declaring and manipulating scalars and arrays in 'F'.

argument, along the axis specified by the integer second argument. Functions such as `shape` and `size` provide essential support to the first-class presentation of 'F' arrays. The assignment statement on the following line 6 demonstrates the syntax for array element indexing. Note that element indices in 'F' *by default* start at 1, and therefore it is the first element of `a` which is updated. The output of the final `print` statement is `2.0 2.0 3.0 4.0 5.0 6.0`.

Functions may have optional parameters. The second argument of the intrinsic `size` function, for example, is optional; its omission corresponds to a request for an array argument's entire element count. While C++ also has optional *trailing* arguments, 'F' offers greater flexibility by allowing arguments to be specified by name. As a quick example, the earlier call to `size` in listing 3.24 could also be constructed with a reversed argument order as `size(dim=1,array=shape(a))`.

Listing 3.24 declares variables with `integer` and `real` *base types*. Six base types are provided in 'F', ranging over the alternatives shown in the Backus-Naur form (BNF) production rule, *type-spec*, of Figure 3.2. The rule definition is taken from F Syntax Rules (1996), though expressed here using a variant of BNF employed by recent Fortran standards, as described in INCITS/J3 (2010, pages 21-22); in particular, square brackets delimit optional terms.

The *kind-selector* of a type declaration references a named constant integer object. This type parameter identifies an object's *kind*, and often corresponds to the width in bytes of a scalar object with that type[9]. The *length* of a `character` is a second type parameter, however of the two, only the kind is always known statically. The kind can be used to specialise a generic interface, and so resolve operator and procedure overloading at compile time.

A named constant is declared using the `parameter` attribute, and may not be modified after initial definition. Listing 3.25 declares two such constants, and uses them to specify the

[9]The exact correspondence between an object's kind, and its in-memory representation, is permitted to vary between compiler vendors. For example, an object with a kind value of 1 could occupy more than 1 byte.

```
type-spec     is INTEGER [ kind-selector ]
              or REAL [ kind-selector ]
              or CHARACTER char-selector
              or COMPLEX [ kind-selector ]
              or LOGICAL [ kind-selector ]
              or TYPE ( type-name )

kind-selector is ( KIND = scalar-int-constant-name )

char-selector is ( LEN = char-len-param-value &
               [ , KIND = scalar-int-constant-name ] )
```

Figure 3.2: Base types available in 'F'.

kind parameter of the single and double precision variables: `r4` and `r8` respectively. The initialisation of `r8` also illustrates that numeric literals too, might require an explicit kind if the vendor default is unsuitable.

```
program p3
  integer, parameter :: k4 = 4, k8 = 8
  real(kind=k4) :: r4 = 0.0
  real(kind=k8) :: r8 = 0.0_k8
end program p3
```

Listing 3.25: Single and double precision floating point variables.

3.3.2 Array Sections

An *array section* is an 'F' language feature which allows a rectangular subsection of an existing array to be addressed *by reference*. Though a reference to existing data, an array section produces a new array, and may consequently be passed as an array argument to any suitable procedure. The syntax of an array section extends that of array element indexing, with the use of a colon to separate the lower and upper integer bounds of a *range* of elements. An optional third value, the *stride*, may also be supplied, following a second colon. A stride of n can specify that only one from every n elements in a range is selected. For example, `q(1:6:2)` will produce a rank 1 array of size 3, addressing elements 1, 3 and 5 of an array `q`.

In Listing 3.26, the target of the assignment on line 7 is an array section, having a shape of `(/3,3/)`. This assignment is legal as `d` too has a shape of `(/3,3/)`, and so an element by element copy occurs. An array section does not require a range for each dimension involved: on line 8, `d(:,3)` specifies the entire range of `d`'s *first* dimension; with only the third element of its *second*. In terms of rows, for rank 1, and columns, for rank 2, this section provides the

```
1  program p4
2
3    real, dimension(4,5) :: c
4    real, dimension(3,3) :: d
5    real, dimension(3)   :: e
6    read *, d
7    c(2:4,1:5:2) = d
8    e            = d(:,3)
9
10 end program p4
```

Listing 3.26: Examples of array sectioning in 'F'.

entire third column of `d`. The assignment is therefore between two rank 1 arrays of shape `(/3/)`.

As shown on line 8, a section may omit an upper or lower bound specification, or both. The compiler then presumes to use the default bounds of the sectioned array, as returned by the `lbound` and `ubound` intrinsic functions. The result of an array section will always have a 1 lower bound for each dimension. Note that a section of a section is not permitted.

The presence of a scalar index in a section, such as the `3` in `d(:,3)`, has the effect of reducing the dimensionality of the outcome by one; and hence this section has a rank of 1. Arrays in 'F' have a statically defined rank, and the use of a lexical token, the colon, ensures this remains true for array sections. For example, sections such as `d(:,:)`, and even `d(:,3:3)`, are both rank 2, having shapes of `(/3,3/)`, and `(/3,1/)` respectively. The intrinsic `reshape` function is also unable to avoid static rank evaluation due to the mandate that its shape argument be statically defined. This observation allows a C++ compiler to perform the same type-checking of array ranks as an 'F' compiler, assisted by a new templated array class representation, described in Section 5.2.

3.3.3 Scalar and Array Pointers

Pointers in 'F' may target only those entities, or subobjects of entities, which have been declared with the `target` attribute. A pointer can though also target another pointer. The target of a pointer is set using a pointer assignment, as shown on line 8 of Listing 3.27. This design eliminates the need for a pointer dereferencing operator, and any pointer operand in an expression will automatically dereference to the underlying target.

A pointer must have the same type as its target. Consequently, to target an array, a pointer must also be declared with the `dimension` attribute. Only the rank is specified in an array

```
1  program p5
2
3    complex, pointer :: pc
4    complex, target  :: c
5    logical, pointer, dimension(:)   :: pi
6    logical, pointer, dimension(:,:) :: pa
7    logical, target,  dimension(2,4) :: a
8    pc => c
9    pa => a
10   pi => a(2,1:)
11   allocate(pa(3,9))
12   deallocate(pa)
13
14 end program p5
```

Listing 3.27: Using pointers in 'F'.

pointer declaration, with the shape *assumed*; as shown on lines 5 and 6. Line 10 demonstrates pointer targeting of a valid subobject of an array, an array section. Here the section and pointer are both of rank 1. This pointer methodology therefore extends the mandate on the static definition of array ranks, to also include array *references*.

Memory may also be allocated dynamically using the `allocate` statement, and freed with the `deallocate` statement. Listing 3.27 demonstrates the use of a pointer to receive and then release memory obtained in this way. An array declared with the `allocatable` attribute, and *assumed* shape, may be used in a similar fashion. Initially allocated no memory, an array so declared may also be dynamically associated with a new target using the `allocate` and `deallaocate` statements.

3.3.4 The Do Construct

'F' provides a single iteration, or "loop", construct: the *do construct*, and is provided in two forms: *bounded* and *unbounded*. The bounded version has the general form shown in Figure 3.3; and taken from Metcalf and Reid (1996, page 54).

```
do variable = expr1,expr2 [,expr3]
  block
end do
```

Figure 3.3: General form of the bounded do construct.

The internal section marked as *block*, the "loop body", denotes a sequence of statements; which may include another, *nested*, do construct. Upon encountering a do construct, the

program will initially assign *variable* to *expr1*. The block may or may not then be executed, but in either case, *variable* is incremented by the value of *expr3*; which has a default value of `1`. Execution may then begin an iterative process by transferring control back to the start of the block. The number of such iterations is determined in advance according to equation 3.1. Consequently, the alteration of any variable used as a term in the three parameter expressions, within the loop body, will have no effect on the loop trip count, nor the values assigned to *variable*.

$$niters = max((expr2 - expr1 + expr3)/expr3, 0) \tag{3.1}$$

Listing 3.28 shows both a bounded and an unbounded do construct populating a two-dimensional array with ascending integer values. The bounded do construct, starting on line 6, is doubly nested, and exhibits a cache-friendly traversal of `a`'s elements. In 'F', the arrays are stored in *column-major* format; and the "fastest moving" index is the leftmost; in this case: `i`.

The second do construct, on line 12, is unbounded. An unbounded do loop will iterate endlessly, or until an `exit` statement occurs. In this example, the loop is traversed 6 times, before the predicate of the *if construct* on line 14 is satisfied, and the `exit` statement on line 15 causes execution flow to transfer outside of the loop; and the program ends. This mechanism is essentially distinct from the bounded form, in that the unbounded variety is guaranteed to execute the loop body at least once.

```
1  program p6
2
3    integer, dimension(2,3) :: a
4    integer :: i, j, k = 1
5
6    do j = 1,3
7      do i = 1,2
8        a(i,j) = (j-1)*2+i
9      end do
10   end do
11
12   do
13     a(mod(k-1,2)+1,(k-1)/2+1) = k
14     if (k==6) then
15       exit
16     end if
17     k = k + 1
18   end do
19
20 end program p6
```

Listing 3.28: Bounded and unbounded do loops.

3.3.5 Subroutines and Functions

Procedures in 'F', also known as subprograms, come in two varieties: *subroutines* and *functions*. As with the `greet` subroutine of Listing 3.23, these must be defined within a module, and given an accessibility attribute using a `public` or `private` statement. Formal procedure arguments in 'F', also known as *dummy arguments*, must be declared with one of three `intent` attributes. An `intent(in)` attribute signifies a parameter which must only be read; an `intent(out)` parameter must be written to, and originate from an actual argument which is a variable; an `intent(inout)` argument must also arise from a variable, and should be both read and written to. Listing 3.29 demonstrates the use of all three intent attributes in a subroutine to calculate a running sum of squares. A subroutine must be invoked using a *call statement*, as shown in Listing 3.23.

```
subroutine sos(x,x2,total)
  real, intent(in)    :: x
  real, intent(out)   :: x2
  real, intent(inout) :: total
  x2    = x*x
  total = total + x2
end subroutine sos
```

Listing 3.29: Dummy arguments and the intent attribute.

Unlike subroutines, functions emphasise *purity*: an 'F' function may not alter an argument; and hence all parameters must have the `intent(in)` attribute. Furthermore, a function is permitted neither to alter a module's data objects; nor perform IO operations, with the exception of the `print` and `read` functions only.

Listing 3.30 demonstrates a simple function definition. The function `amean` returns the arithmetic mean of a two-dimensional array of reals, and uses two intrinsic functions: the additive reduction, `sum`; and the real type conversion, `real`. Functions are also distinguished from subroutines in that their application does not require a call statement, and can form part of an expression; for example: `amean(a) * ndays`. Consequently, a function must return a value, and declare a *result* variable with this role. Note that a result variable is not a dummy argument, and hence is not declared with an `intent` attribute.

Recursion of procedures is permitted, however not by default. Any function which calls itself directly or indirectly must have its declaration prefixed with the `recursive` keyword.

```
function amean(a) result(r)
  real, intent(in), dimension(:,:) :: a
  real                             :: r
  r = sum(a) / real(size(a))
end function amean
```

Listing 3.30: Function definition in 'F'.

3.3.6 Derived Data Types

User-defined aggregate data types, known as *derived data types*, can be defined using a *type declaration statement*, as shown in Listing 3.31. In terms of type theory, this is a labelled, n-ary product type constructor (Pierce, 2002, page 127), with the result known as a *record* type. While in this example `vec4` is formed from four real types, any set of fields is permitted, including derived types which have been previously defined. Pointer fields are also permitted, and these alone may reference types which are not yet defined; including the type being defined. As with procedures, each new type must be declared in a module, though the definition of a derived type must appear *prior* to that module's *contains statement*[10]. A type declaration must also receive an accessibility attribute, and this is specified as a clause of the declaration. The `private` accessibility of the `vec4` type restricts its use to the module of its declaration.

```
type, private :: vec4
  real :: x,y,z,w
end type vec4
```

Listing 3.31: A 4-tuple vector derived data type.

Listing 3.32 demonstrates usage of a derived type object. On line 3 the subroutine `zero` sets all four fields of its derived type argument to zero using the `vec4` type's *structure constructor*, and its overloaded assignment operator; both of which are defined automatically. To address specific fields of a structure, the *component selector*, (`%`), may used; for example: `v%x`. As with arrays, a derived type object may be used holistically within IO statements, as shown in the `print` statement of line 4.

A number of components of the 'F' programming language have been omitted from discussion here, partly due to space considerations; but primarily owing to their lack of relevance to the core research. Subjects left untouched include: kinds; character types; public modules; module data objects; the parameter attribute; operators; interfaces; overloading; procedure arguments; implied-do loops; vector subscripts; formatted I/O; the where and case

[10] A *contains statement* is shown on line 5 of Listing 3.23.

```
1 subroutine zero(v)
2   type(vec4), intent(out) :: v
3   v = vec4(0.0, 0.0, 0.0, 0.0)
4   print *, "v is:", v
5 end subroutine zero
```

Listing 3.32: Using a derived data type.

constructs; and additional statements and intrinsic procedures. All such language aspects are adequately explained in Metcalf and Reid (1996).

Significant language topics remain in need of address; specifically: *array expressions* and *elemental functions*. Both play a central role in the parallel model to be developed, and will be examined at length in Section 3.4, in the context of their application to implicit parallelism.

3.3.7 Differences between 'F' and Fortran

Unlike Fortran, backwards compatibility is not a concern of 'F', and a relative minimalism arises firstly from the removal of historic, or obsolescent, language features. The main demonstration of this is the absence of support for *fixed source form* (INCITS/J3, 2010, pages 45-47). Fixed source form attributes special significance to the characters placed in columns 1-5, 6, and 73-80 of the source code, and originates from the peculiarities of 80-column punched cards used to prepare programs for early computers. 'F' instead uses *free source form* exclusively, where no restriction on the placement of statements exist within a line. Simplicity is also improved by removing alternative methods to achieve the same result, for example, 'F' has no while clause following an unbounded `do` construct, relying instead on the `exit` statement.

Fortran's `implicit` statement is also absent, and an 'F' program behaves as if an `implicit none` statement were the default in all scoping units. This requires that all variables must be declared, and is an alternative to traditional Fortran rules for implicit declaration based on the initial character of a variable.

Of relevance for parallelism, functions in 'F' are free of side-effects (Metcalf and Reid, 1996, page 197). Fortran's `pure` keyword is absent from F, while each function definition is implicitly given its effect.

3.3.8 Differences between Fortran and 'C'

'C' is also a compiled, statically-typed, imperative programming language, and often compared with Fortran. 'C', however, has become identified as a systems programming language, while Fortran is more often associated with computational science. Minor aspects such as the long-standing support for complex numbers in Fortran may contribute here.

At a language level, Fortran has differentiated itself from 'C' by its lack of a low-level operator to provide the machine address of a variable[11]. Pointer arithmetic is therefore largely absent. While Fortran 90 introduced pointers, pointer targets must be declared explicitly. Consequently, alias analysis performed by a Fortran compiler is relatively simple compared to a 'C' compiler.

In Fortran, arguments are passed by reference, while in 'C', they are passed by value. Unlike 'C', Fortran has intrinsic support for first-class array types, which may be generically interrogated for their extents. Procedure arguments in 'F' do not, therefore, suffer from the pointer decay observed in 'C'. While once referred to as high-level programming languages, Fortran and 'C' are perhaps now best described as medium-level languages.

3.4 Implicit Parallelism

It is the purity of an expression which governs its possible parallelisation. The compiler to be presented will parallelise *array expressions*. In the following section we will present the opportunities available for the automatic parallelisation of a useful class of expressions with rank greater than one.

3.4.1 Scalar Expressions

Expressions in 'F' are formed from operators, operands, and parentheses. Operands can include variables; constants; constant subobjects; function references; array and structure constructors; and, significantly, expressions themselves. Binary and unary operators use infix and prefix notation respectively. In the absence of parentheses, an expression involving multiple operators will be evaluated according to the precedence level of each participating operator, as listed for the 'F' language in Metcalf and Reid (1996, page 40). If an expression

[11]The address of a Fortran variable *can* be obtained, though non-portably, using compiler extensions. For example, the `loc` intrinsic function of GNU Fortran.

involves only operators of the same precedence level, such operators will be evaluated from left to right; and in so doing, force the evaluation of their operands. The exponentiation operator (`**`) is an exception, being evaluated instead from right to left.

An operand enclosed by parentheses will be evaluated entirely before participating in the evaluation of its operator. Hence, the use of parentheses may raise the precedence level of enclosed operators.

While there need be no mathematical distinction between the application of a function, and an operator of equal arity, in 'F', a function application is only an operand, and must follow the rules of operator evaluation. Consequently, the application of a function is similar to a parenthesised expression; and effectively maximises the precedence of the function involved. For example, the `*` operator has a higher precedence that the `+` operator, yet with a minimal user-defined addition function, `add`, expression `2 * add(3,4)` will evaluate to 14.

An expression involving only pure terms may be evaluated in parallel. Functions in 'F' *must* be pure, and therefore evaluation of an expression such as `f(x) + g(x)` can proceed by calculating the result of the operands `f(x)` and `g(x)` independently, and in parallel.

Parallel evaluation of subexpressions may also be possible, though this will depend on the *associativity* of the operators concerned. For example, the use of real-valued addition in the expression, `f1(a) + f2(b) + f3(c) + f4(d)`, allows for the parallel evaluation of subexpressions `f1(a) + f2(b)` and `f3(c) + f4(d)`. In contrast, `f1(a) - f2(b) - f3(c) - f4(d)`, must evaluate the three real-valued subtraction operations in serial, as subtraction is non-associative [12].

Pure functions also facilitate *short-circuit evaluation* (Aho et al., 1986, pages 490-491), wherein operands which have no effect on the enclosing expression may remain unevaluated; and potentially improve performance. Consider the boolean expression in the `if` construct of Listing 3.33. Due to the familiar definition of the boolean conjunction operator, `.and.`, the function application, `fin(x)`, may remain unevaluated whenever `i` is non-zero.

```
if (i==0 .and. fin(x)) then
  exit
end if
```

Listing 3.33: Short-circuit evaluation.

Short-circuit evaluation is available in both 'F' (Metcalf and Reid, 1996, pages 37 and 73) and Fortran (INCITS/J3, 2010, page 147).

[12] Note that floating-point arithmetic is itself non-associative.

When considering parallelisation specifically, it is helpful to know that the functions in ‘F’ are side-effect free. Nevertheless, to aid debugging, the use of the `print` and `read *` statements are also permitted. Formatted `read` and `write` statements too can be used, though only with memory buffers known as *internal files*. Although not explicitly specified, it also seems likely that the internal files should be declared as variables local to the function concerned[13]. The set of side-effecting operations permitted is therefore highly restricted, and easily identified by a compiler.

Unlike ‘F’, the Fortran 2008 specification does not require a function to be pure, though the order in which function applications are performed, within a statement or expression, must not change the final outcome:

> “If more than one function reference appears in a statement, they may be executed in any order (subject to a function result being evaluated after the evaluation of its arguments) and their values shall not depend on the order of execution. This lack of dependence on order of evaluation permits parallel execution of the function references.” (INCITS/J3, 2010, page 479)

Consequently, the lack of purity in Fortran functions would not prevent a parallel execution model similar to that of E♯; though such an implementation must of course ensure that side effects are properly executed in advance of their visibility.

3.4.2 Elemental Functions and Array Expressions

A scalar procedure or operator which is *pure*, may also be *elemental*. An elemental operation can be applied to both scalar and array operands. With array operands, the result too will be an array of the same shape. Each element of the output array is equal to the result of applying the elemental operation to corresponding elements of the input arrays. Many of the ‘F’ intrinsic functions are elemental; for example, the real-valued expression in Listing 3.34’s print statement uses the elemental functions `sin` and `cos`; and the elemental operators `*` and `+`.

```
print *, cos(a) * cos(a) + sin(a) * sin(a)
```

Listing 3.34: Elemental intrinsic functions and operators in use.

[13]The g95 compiler provides an error when a host associated internal file is accessed from within a function.

It is straightforward to define a new elemental procedure. Figure 3.35 shows the definition of an elemental function: `first`. Given three arrays as input, the result will be equal to the first array. Subroutines may also be elemental, though obviously they cannot be part of an expression. Subroutines do however allow for dummy arguments to be modified.

```
elemental function first(a,b,c) result(r)
  real, intent(in) :: a, b, c
  r = a
end function first
```

Listing 3.35: Defining an elemental function.

The main restriction on the use of elemental procedures is the requirement that the shapes of the operands must be the same; or *conform*. An exception is made for scalar operands among otherwise conforming array operands, in which case the scalar value is *lifted*, and treated as if it were a conforming array, populated entirely by the replication of its original value.

An expression with a rank greater than zero may be referred to as an *array expression*, and can of course include non-elemental functions and operators, returning either scalar or array values. As with procedures, user-defined non-elemental operators can include operands of varying shapes, and ranks. As is the case for a data object, the rank of an expression is known statically.

3.4.3 Array Assignment

The semantics of the *assignment statement*, targeting an array variable, are highly distinctive, and facilitate parallelism through a specification which is independent of the evaluation order of individual elements. The expression appearing on the right-hand side of an assignment statement is evaluated in its *entirety* before any component of the variable on the left-hand side is updated[14]. Consider the cyclic vector right-shift algorithm provided in the `cshiftr` function of Figure 3.36. The majority of the calculation is performed by the assignment statement on line 6. Assuming array `a` has n elements, the statement assigns the *first* $n-1$ elements of `a`, to its *last* $n-1$ elements. For example, an array input as `(/1,2,3,4/)` would result in the array `(/4,1,2,3/)`.

The significance of these semantics occurs at the implementation level. Evaluation of an array expression can be efficiently undertaken within a loop, or loops, nested to a depth equal to the rank of the expression. The straightforward loop described by Algorithm 1,

[14]Array assignment in APL proceeds similarly. PL/I, however, updates the assignee iteratively, and in-place; in sequential, lexicographic order (Barron, 1977, page 92).

```
subroutine cshiftr(a)
  integer, dimension(:), intent(inout) :: a
  integer :: n, tmp
  n      = size(a,1)
  tmp    = a(n)
  a(2:) = a(1:n-1)
  a(1)   = tmp
end subroutine cshiftr
```

Listing 3.36: Dependency in array assignment.

however, is an unsuitable implementation; an input of `(/1,2,3,4/)` will result in the array `(/4,1,1,1/)`. This situation can be corrected by the use of a scalar temporary upon each iteration. However, a simpler and more expedient implementation will traverse the loop *backwards*, as in Algorithm 2. General solutions to the problem of efficiently implementing Fortran array assignments are presented in Kennedy and Allen (2002, Chapter 13). The use of n-point stencil convolutions in computational science and image processing present a common use case for more intricate array assignments.

Algorithm 1 Incorrect cyclic naïve right shift.

Require: An integer n and array A of length n.

1: $tmp \leftarrow A[n]$
2: **for** $i = 2$ to n **do**
3: $\quad A[i] \leftarrow A[i-1]$
4: **end for**
5: $A[1] \leftarrow tmp$

Algorithm 2 Correct cyclic right shift using a reversed loop.

Require: An integer n and array A of length n.
Ensure: The elements of A are shifted right by one.

1: $tmp \leftarrow A[n]$
2: **for** $i = n$ to 2 **do**
3: $\quad A[i] \leftarrow A[i-1]$
4: **end for**
5: $A[1] \leftarrow tmp$

3.4.4 Parallel Execution

Sections 3.4.1 and 3.4.2 have identified two aspects of array expressions which make them suitable for execution in parallel:

- The terms of all array expressions are *pure*, and hence cause no side-effects which could impose a sequence on any evaluation order; and

- Many intrinsic operators and functions are *elemental*, therefore providing no restriction on the order of evaluation of individual array elements comprising the aggregate result.

Consequently, it is proposed that array expressions involving elemental operators or procedures, whether intrinsic or user-defined, be automatically parallelised. In addition, a `call` statement applying an elemental subroutine shall also be executed in parallel. Expressions involving no elemental procedures or operators will retain a serial execution.

Many array expressions are composed of a combination of both elemental and non-elemental procedures. This need not prevent parallel execution entirely. Mixed expressions of this sort can be transformed into multiple statements, so permitting the non-elemental procedures to be evaluated in advance of the parallel execution of the remainder. This requires a *hoisting* operation to be executed by the compiler, wherein temporary array declarations are inserted, along with assignments sourced from the obstructing non-elemental procedure application. Subsequently, a reference to each such array then replaces the procedure call in the original expression. This entire compiler transformation must also perform recursively, as the arguments of a hoisted, non-elemental function call may themselves be parallelisable array expressions.

There is more expressivity in the construction of algorithms using arrays and elemental operations than may at first be apparent. Consider again a notable restriction: elemental procedures can operate only upon scalar types; and therefore subdimensions of an array may not be so handled. For example, the columns of a two-dimensional array would be an unsuitable argument to an elemental procedure. Derived data types, however, provide a general, and straightforward means to overcome this problem. In Listing 3.37 the columns of a two-dimensional array of reals, `a`, are sorted by a call to the elemental subroutine, `esort`. This is made possible by the definition, at line 6, of a proxy, derived type, `vec`, containing a single field: the one-dimensional array pointer, `pr`. By first initialising a one-dimensional array of `vec` types, using the pointer assignment statement on line 29, each `esort` subroutine argument provides access, by reference, to the original elements of `a`. Elemental *subroutines* are also suitable targets for automatic parallelism, and so the `call` statement of line 32, the kernel of the program, will sort each column in parallel with the others[15].

That the `esort` subroutine applies a sort to each of the *columns* of `a`, as opposed to the *rows*, is due to the first-position placement of the colon in the pointer assignment target; on line 29. Any single dimension could similarly be selected from an array of greater or equal rank.

[15]The subroutine `sort1d` is unspecified here; and is assumed to be defined in module `sorting`.

```
1  module m
2
3    use sorting
4    public :: esort
5
6    type, public :: vec
7      real, pointer, dimension(:) :: pv
8    end type vec
9
10   contains
11
12   elemental subroutine esort(a)
13     type(vec), intent(inout) :: a
14     call sort1d(a%pv)
15   end subroutine esort
16
17 end module m
18
19 program p7
20
21   use m
22   integer                                  :: i
23   real,      target, dimension(1000,2000) :: a
24   type(vec),         dimension(2000)      :: acols
25
26   read *, a
27
28   do i = 1,2000
29     acols(i)%pv => a(:,i)
30   end do
31
32   call esort(acols)
33
34 end program p7
```

Listing 3.37: Elemental operation on a derived type.

While ‘F’ provides a small selection of reduction operations, these are neither elemental; nor generic, in that they are *first-order*, and so lack a procedure argument. For example, the intrinsic `sum` and `product` functions provide specifically additive and multiplicative reduction respectively. Nevertheless, a flexible definition schema for parallel reduction can be obtained through recognising that an elemental operation may return a scalar value of a type distinct from each of its arguments.

Listing 3.38 provides an elemental function definition which might also be applied to the real array `a` of previous listing 3.37. In this case, the operation applied to each single dimension of the original array is the intrinsic `sum` reduction function.

As this research does not extend the ‘F’ language, there is the possibility that legacy codes, which use small array expressions, may in fact run more slowly as a consequence of a parallel

```
elemental function esum(a) result(r)
  type(vec), intent(in) :: a
  real :: r
  r = sum(a%pv)
end function esum
```

Listing 3.38: Elemental reduction operation on a derived type.

execution. This is due to the overhead of launching threads; and transferring data between the host memory and the local memory of the SPEs. For such situations, the recommended solution is to construct an equivalent loop, using the `do` construct, which retains its traditional serial execution model. For example, the parallelised `call` statement of Listing 3.37 could be replaced with listing 3.39 and so regain a serial execution.

```
do i = 1,2000
  call esort(acols)
end do
```

Listing 3.39: Sorting in serial using the traditional `do` construct.

A final, relevant, though auxiliary, advantage of the system described concerns code longevity. Any significant investment by a user in a new method of expressing parallelism, is typically associated with a high risk of future obsolescence. E♯ minimises this risk by the following two design principles. First of all, the 'F' language is not extended. Consequently, code may be compiled by any Fortran compiler for the foreseeable future. Hence the approach is highly *portable*. Secondly, the constraints placed on code structure, to facilitate parallel execution using E♯, are also highly conducive to performant *serial* execution; using conventional Fortran compilers. The approach should not, therefore, reduce the performance of existing serial codes modified for compatibility with E♯.

Chapter 4

Compiler Implementation

This chapter will look at the E♯ compiler, which, as part of a larger compilation toolchain, translates sequential 'F' programs which utilise array expressions, into executable parallel programs targeting the heterogeneous architecture of the PS3. Together with its associated runtime library, the E♯ compiler is a substantive component of the presented research. The compiler is written in Haskell, and comprises 12 files, with 4814 lines of code; 1605 lines of comment; and 1268 blank lines. While Haskell is commonly applied to compilation tasks, the 'F' input language is, distinctively, an industrial language of notable scale. The parsing module of E♯, also demonstrates that monadic parsing combinators, such as the Haskell Parsec (Leijen and Meijer, 2001) library, are applicable to complex grammars such as 'F'[1].

Alternative compilation technology to the bespoke Haskell compiler, E♯, were also considered: Though LLVM (Lattner, 2002) had shown promise, it has been unstable for much of the last few years; frequently undergoing radical changes in its design and API; and no GCC-like (GNU Project, 2012) policy of retrospectively applying bug fixes to update existing releases. The ROSE compiler (Liao et al., 2010) appeared half-way through the full project schedule, and was nevertheless fully evaluated. ROSE was found to be immature, and with poor support for the Windows OS. The advertised, simple code transformation API was also observed to be no better than that offered offered by Parsec and SYB (Lämmel and Jones, 2003, 2004, 2005).

We begin this chapter with an overview of the E♯ toolchain; placing E♯ relative to the existing tools and libraries which process its output. E♯'s internal operation is then introduced with a description of the compiler's three intermediate representations (IRs). There follows a description of the monadic parsing combinators used to populate the first IR: the parse tree. Object binding follows, wherein the contents of the symbol table are used to allocate

[1] Language.C(Huber, 2011) is another good Parsec example; for C99.

each untyped parse tree expression with a type. The translation from ‘F’ construct to an equivalent C++ representation is then considered, before focusing on the particular challenge of transforming an array expression to a form suitable for parallel execution on the target architecture. Finally, the method used by E♯ to support an optimised representation of constant-sized ‘F’ arrays and strings is presented.

4.1 Overview of the E♯ toolchain

The diagram in Figure 4.1 presents an overview of the executable components of the toolchain associated with the E♯ compiler at an operational level. E♯, Offload, and the GNU toolchain all run under Cygwin on Windows, and are together capable of producing a 32-bit ELF executable suitable for execution on the PS3 under Linux[2]. As shown, the SPU object files are embedded within a PPU object file using the GNU ppu-embedspu tool as described previously in figure 2.1 of chapter 2. The GNU toolchain and cross compilers used are 32-bit and are optionally installed alongside the Offload compiler. For market and stability reasons, these tools remain fixed at the version levels supplied with version 3.0 of the IBM Cell SDK[3]. Nevertheless, the compilation stages performed by the GNU tools are fully configurable; and all components are potentially interchangeable.

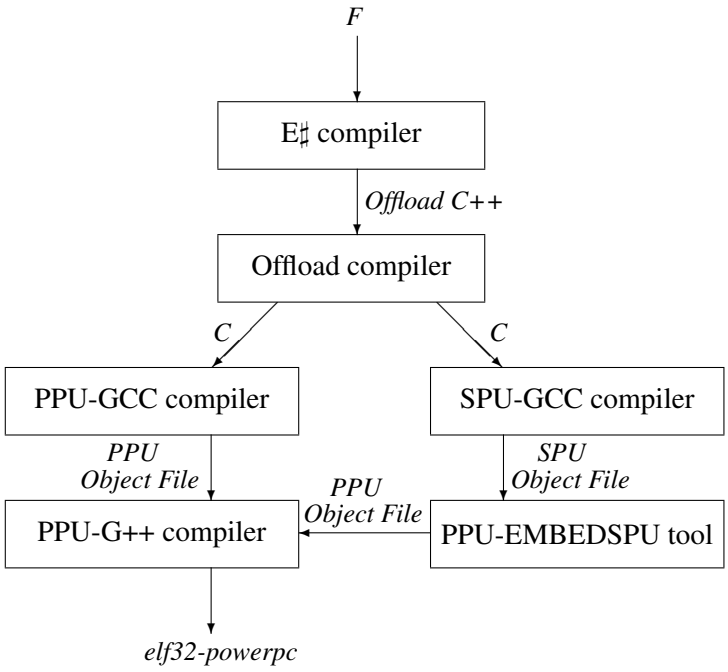

Figure 4.1: The E♯ compiler and the system at-large.

[2]The PS3 test system runs Fedora Core 7 “Moonshine”.

[3]Version 3.0 of the Cell SDK includes GCC version 4.1.1.

The E♯ compiler accepts individual source programs written in the ‘F’ programming language, before translating them into the extended C++ dialect, Offload C++. The purpose of this transformation is to extract and partition suitable ‘F’ array expressions into multiple offload blocks, later compiled by the Offload compiler for parallel evaluation. Of course E♯ must also convert the base language from ‘F’ to C++. This is not in itself too onerous, as both languages provide a comparable level of abstraction. Nevertheless, a consistent treatment of the novel pointer locality in Offload C++, will render many ‘F’ procedures as template functions, parametrised by the offload-depth.

The language output by the E♯ compiler is in fact configurable using the ‘C’ preprocessor via macros embedded within the generated code. If the `__sievecplusplus` macro is defined, as it is by default when compiling with the Offload tools, the extended Offload C++ portions of the generated code are enabled; facilitating the expected parallel execution. With the `__sievecplusplus` macro *undefined*, the generated serial C++ code may be compiled by any C++ compiler. As will be described, the existence of a serial reference is useful for debugging, testing, and also performance profiling.

The Offload compiler is provided with the extended C++ output from the E♯ compiler as input. The Offload compiler requires access to the E♯ runtime header libraries; and may need a Fortran runtime library compatible with the PPU, the SPU, or both. As part of the Offload compilation process, two nested directories containing mainly ‘C’ source and header files are created: the outer targeting the PPU; the inner targeting the SPU. A GNU makefile is also created in each directory, and a successful build typically applies the GNU make tool to the PPU makefile, which then applies make recursively to the SPU makefile.

Sony’s Multicore Application Runtime System (MARS) ((**alias?**)) libraries are used to implement the threading support required by the Offload runtime system.

The Offload compiler can also be invoked with the *-nomake* switch, in which case the generated source and makefiles are not deleted; and the GNU make tool is not applied. This option is essential when different tools, or versions other than the defaults, are required. For example, in the configuration shown in Figure 4.1, the GCC compilers conventionally operate as cross-compilers. A recent optimisation in GCC has reduced the amount of SPU local store required to accommodate the GFortran runtime library, and was included in the version of GCC released in March of 2011; GCC 4.6. The 4.6 versions of PPU-GCC, PPU-G++, SPU-GCC, and associated tools were consequently built and installed natively on the PS3. Accounting for this alternative, the bottom four GNU tools of Figure 4.1 may therefore execute either as cross-compilers under Windows Cygwin, or natively under Linux.

The approach to E♯’s development has attempted to both provide a level of support for ar-

ray language features appropriate to the research at hand; while also hosting a substantial proportion of the familiar ‘F’ language feature set. For example, E♯ includes support for kinds; pointers; derived types; array literals; character types; IO operations; timing, random number and matrix multiplication procedures. From a language perspective, the main *omissions* include overloading and generic interfaces; statement labels; edit descriptors; procedure arguments; named and optional arguments; and the where construct. ‘F’ also provides numerous intrinsic functions. The majority of these are provided by the associated Fortran runtime library, though a proportion of these lack the necessary interoperability layer, or *glue*, to facilitate their use. Of those intrinsic functions which are typically absent from Fortran runtime libraries, E♯ provides direct support, typically with an inlined implementation, avoiding a function call; for example, the `kind` inquiry function. E♯ is currently restricted to programs contained within a single file.

The E♯ compiler is written in the statically-typed, pure, non-strict, functional programming language, Haskell. Haskell provides a very high level of abstraction, and can eliminate many kinds of runtime errors entirely. It also enables directly the kind of transformations that a compiler writer would more commonly associate with a separate tools package; such as ANTLR (Parr, 2007). Haskell is though an unfamiliar language to many, and for those from an imperative, block-structured, programming background, a number of its characteristics can appear surprising. Before examining the internal details of the E♯ compiler, the brief description of the essential details of Haskell provided in Appendix C may be a useful reference.

4.2 Intermediate Representations

The relative parity between ‘F’ and Offload C++ permits a translation strategy for E♯ wherein the majority of language statements and constructs in the former, may be translated directly into a corresponding statement or construct in the latter.

Figure 4.2 illustrates, from left to right, the three internal representations of an ‘F’ program at successive stages within the E♯ compiler. Lexical and syntactic analysis are first performed on the text of an ‘F’ program file, allowing the creation of a parse tree, or concrete syntax tree, represented as the diagram’s leftmost box. The successful creation of a parse tree confirms that the production rules of the ‘F’ grammar can generate the input.

Names within the parse tree remain untyped, and are essentially raw strings. ‘F’ requires that program units, procedures, procedure interfaces, and derived type definitions are named *twice*; at the start and end. Both names should be the same, however such checks are deferred

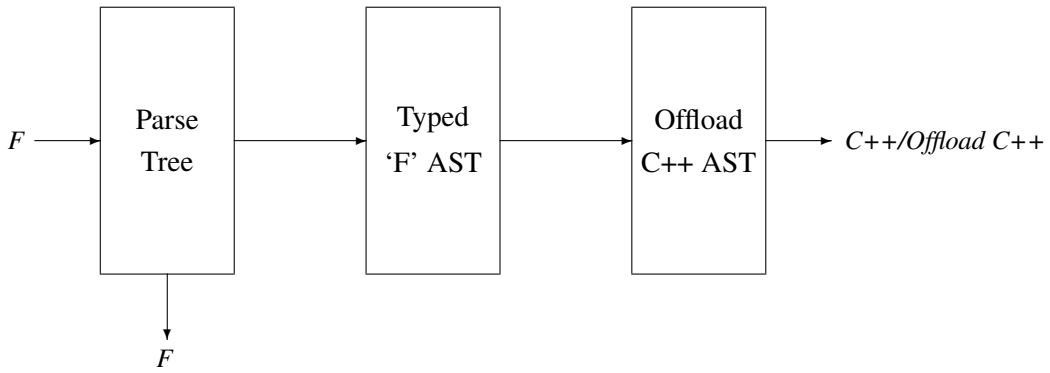

Figure 4.2: The intermediate representations of the E♯ compiler.

until after creation of the parse tree to emphasise modularity. Unlike C++, 'F' is case-insensitive, and any variation in case is preserved by the parse tree. Comments are however removed.

The parse tree has an associated code generation module which can generate 'F' code which is identical to the input, modulo formatting and comments. This component can be used to aid testing and debugging of the parser.

Type checking and object binding allow the parse tree to be converted to an abstract syntax tree (AST) representation of the 'F' programming language, free of unnecessary syntactical details such as repeated names and parentheses. The typed 'F' AST is a suitable form upon which to apply source language transformations such as the hoisting and precalculation of nested array expressions.

Like the typed 'F' AST, the Offload C++ AST is a representation of source code. Significant transformations between the two forms include the lowering of array expressions to nested loops, and insertion of offload blocks to allow a parallel execution. The code generator can then serialise this final intermediate representation directly into the combined, macro-configurable C++/Offload C++.

Each of the three intermediate forms described are represented by a corresponding Haskell algebraic data type (ADT). Of these, each is defined using a collection of related ADTs specific to an intermediate representation. The following sections will examine each in turn.

4.2.1 The Parse Tree

The ADT of the parse tree follows a similar structure to the grammar of the 'F' programming language (F Syntax Rules (1996)), itself defined using a custom, extended form of

BNF (EBNF). Figure 4.3 shows the first EBNF *production* of the context-free 'F' grammar: *program*. Brackets here surround an optional production, and an ellipsis indicates that the preceding symbol can occur zero or more times. The rule therefore specifies that an 'F' program is constructed entirely from multiple successive program units, of which there is at least one.

```
program    is program-unit
              [ program-unit ] ...
```

Figure 4.3: The *program* production rule from the 'F' grammar.

The Haskell ADT corresponding to the *program* production rule is shown in Listing 4.1[4]. Structurally, the pair are highly similar. In particular, the `Program` type is formed by the cartesian product of a `ProgUnit` and a list of `ProgUnit`s. While a *list* of `ProgUnit`s alone may have zero elements, such a definition for the `Program` type *demands* at least one value of type `ProgUnit`. Excepting *bottom*, $\perp$, and assuming each `ProgUnit` is well formed, we can be certain that a value of type `Program` represents a valid 'F' program.

```
data Program = Program ProgUnit [ProgUnit]
 deriving (Show, Eq, Typeable, Data)
```

Listing 4.1: The Haskell ADT corresponding to the *program* production rule.

Mapping subsequent grammar rules to Haskell ADTs is somewhat less symmetric. Consider the grammar rules of figure 4.4[5]. The ADT corresponding to the *program-unit* production, `ProgUnit`, is derived from the structure of neighbouring productions, as well as its own. A *program-unit* may be a *main-program*, or a *module*; while a *module* itself may be private or public. The `ProgUnit` is consequently a 3-way sum type, with tags `MainProg`, `PrivateMod` and `PublicMod`, as shown in Listing 4.2.

The main program is framed by *program* and *end-program* statements. Within these statements, the *program-name* rule, which matches to a restricted string of characters, is stored twice, by a tuple of `Name`s; each an alias for the common Haskell `String` type. The remainder of the structure, the case-insensitive `END` and `PROGRAM` terminals, are implicitly encoded as part of every `MainProg` data constructor. Similarly, `PrivateMod` and `PublicMod` implicitly encode the `END` and `MODULE` terminals.

[4]The deriving clause of each E♯ AST type lists 4 type classes. Of these, `Typeable` and `Data` are described in Section 4.6.1. The common `Show` and `Eq` type classes are taken from the Haskell prelude; `Show` is also described in Appendix C.

[5]The published 'F' grammar (F Syntax Rules (1996)) contains a typographical error in the `main-specification` production rule.

```
program-unit        is main-program
                    or module

main-program        is program-stmt
                         [ use-stmt ] ...
                         [ main-specification ] ...
                         [ execution-part ]
                       end-program-stmt

program-stmt        is PROGRAM program-name

end-program-stmt    is END PROGRAM program-name

main-specification is type-declaration-stmt
                    or intrinsic-stmt

module              is public-module
                    or private-module
```

Figure 4.4: Additional grammar rules for 'F'.

```
data ProgUnit
 = MainProg       SmT (Name,Name) [SpecStmt] [ExecStmt]
 | PrivateMod     SmT (Name,Name) [SpecStmt] [SubProg]
 | PublicMod      SmT (Name,Name) [SpecStmt]
 deriving (Show, Eq, Typeable, Data)
```

Listing 4.2: The Haskell ADT representing a parsed program unit.

The remainder of the *main-program* production rule is indented only for readability. The *main-specification* rule can produce either a *type declaration statement*, or an *intrinsic statement*. Along with the *use statement*, and others, each is classified, in Metcalf and Reid (1996), as a *specification statement*; and a Haskell type is defined to represent them: `SpecStmt`. The optional *execution part* of a main program produces one or more *executable statements*, and a type is also defined for these: `ExecStmt`. We therefore arrive at the definition of `ProgUnit` given in Listing 4.2. The grammar corresponding to the `PrivateMod` and `PublicMod` data constructors is omitted, and are both similar to that of `MainProg`. The `SubProg` type corresponds to the *subprogram* production.

The only type in `ProgUnit` which does not correspond to a grammar production is `SmT`. As shown in Listing 4.3, an `SmT` type is an alias for an association list; and an `SmT` value is a symbol table. As with `ProgUnit`, a symbol table is attached to those nodes of the parse tree which may have descendants parsed only as names, and in need of type assignment; expressions, for example. The `SmData` ADT is defined by a sum type alternating between

a specification statement, and a subprogram, or procedure[6]. Initially empty, each symbol table is populated prior to object binding to provide the environmental information necessary to allocate each expression with a type. This occurs prior to the creation of the second intermediate form, the typed 'F' AST.

```
type Sm               = Name
data SmData           = DsSmData SpecStmt | SpSmData SubProg
 deriving (Show, Eq, Typeable, Data)
type SmTEntry         = (Sm,SmData)
type SmT              = [SmTEntry]
```

Listing 4.3: The data structure of a symbol table `SmT`.

Like the formal grammar of 'F', the parse tree represents only the basic structure of a program, and makes no association between an expression, and its rank and type. The most elemental expression forms of any language are often known as *primary expressions*, and include function application, or variable references. In the Haskell parse tree, these are distinguished only at a syntactic level with respect to presence or absence of terminal symbols such as parentheses. For example, no distinction can be made at the parsing stage between the indexing of an array, and a function application; consider: `x(y)`.

The representation of a parsed primary expression in E♯, as a Haskell ADT, is shown in Listing 4.4. The three value constructors are used to differentiate, to the degree possible at a syntax level, one primary expression from another. As part of an expression, a name alone may be a scalar or array value, perhaps with attributes such as `pointer` or `intent(in)`. These correspond to a `Prim` value tagged with the `UnknownName` constructor. When instead followed by comma-delimited parenthesised expressions, the denoted primary expression may resolve to the application of a user or intrinsic function; an array element; or a *structure constructor*. Such a parse is then associated with the `UnknownApp` value constructor. Members of a derived type are accessed using the `%` symbol, and may occur both unadorned; and once again as subobjects, including array elements. Parsing such *structure components* will produce a value constructed by `UnknownComp`.

As is commonly observed, the parse tree represents verbose syntactical details, such as parentheses and terminal production symbols, with type and structural information. The parse tree also removes unnecessary hierarchical structure present in the formal grammar, which exists only to aid readability.

[6]The `SmData` type is isomorphic to the following application of the Haskell prelude's `Either` type constructor: `Either SpecStmt SubProg`. The `SmData` ADT shown in Listing 4.3 was chosen in preference due to the relative mnemonic enhancement of Haskell patterns based on `DsSmData` and `SpSmData`, rather than `Left` and `Right`.

```
data Prim
 = UnknownName SmT Name
 | UnknownApp  SmT Name [(Maybe Name, Expr)]
 | UnknownComp SmT [(Prim,[Subscript])] (Maybe Subscript)
 deriving (Show, Eq, Typeable, Data)
```

Listing 4.4: The Haskell ADT representing a parsed primary expression.

4.2.2 The Typed 'F' Abstract Syntax Tree

Associating types with expressions, including primary expressions, is essential to E♯. First of all, an array expression, the source of parallelism for E♯, is defined as having a rank greater than zero. In 'F', rank is part of an object's type. In addition, the temporary data structures created at the transformed site of parallelism, require the appropriate types to specify the variable declarations. Looking now at the ADTs representing the typed AST we will see that it shares much of the structure of the parse tree, and many of the constructor names are identical[7]. We will therefore focus on the pertinent differences between the two.

The Haskell ADT representation of the 'F' program unit is shown in listing 4.5. Comparing this type to that of Listing 4.2 firstly reveals that the symbol table field, `SmT`, is absent. By this stage in the pipeline, each object name from the parse tree has full type information, local to each node. The symbol tables are therefore no longer required. Another difference from the parse tree is that no distinction is made between a specification statement and an executable statement: a single statement type, `Stmt`, now represents both forms. The significance of this recategorisation, is that code transformations, which may require new variable declarations, can be applied locally at the site of each targeted statement or construct. A new data constructor definition, `MultiStmt [Stmt]`, is added to those already listed in the extended `Stmt` type. Unlike 'F', C++ places no restriction on the lexical ordering of the two statement types, and the `MultiStmt` constructor can assist in replacing a single statement with a statement sequence including declarations. Finally, for each of `ProgUnit`'s constructors, a single name component, `Name`, is sufficient, as a check ensuring the original pair were identical has been completed.

Each type in 'F' is associated with an integral kind value which often corresponds to the size in bytes of a corresponding value. The kind is a named constant, and while the parse tree identified the kind with an `UnknownName` constructor, constant folding allows the typed AST to represent a type's kind, and other constant expressions, by an integer. A derived type also

[7]Though many names such as `ProgUnit` and `SubProg` are shared between syntax trees, there is no namespace collision as each ADT exists within a separate module. For example, a fully qualified reference to `ProgUnit` should be preceded by the module name, as in `ParseTree.ProgUnit`, `Fast.ProgUnit`, or `Cast.ProgUnit`.

```
data ProgUnit
 = MainProg       Name [Stmt]
 | PrivateMod     Name [Stmt] [SubProg]
 | PublicMod      Name [Stmt]
 deriving (Show, Eq, Typeable, Data)
```

Listing 4.5: The Haskell ADT representing a typed program unit.

receives a boolean component, to indicate if the final C++ structure representation should be a C++ template, with an integer offload-depth parameter. A boolean with similar effect is also found within the `Prim` ADT's function reference constructor, `FuncRef`; the `Stmt` ADT's call statement constructor, `CallStmt`; and both of the `SubProg` ADT's procedure constructors, `FuncSubProg` and `SubrSubProg`[8].

Distinct from the parse tree's placeholder representation for primary expressions, seen already in Figure 4.4, the `Prim` ADT in the typed AST is fully specialised, with constructors for `VarName`, `ArrVarName`, `FuncRef`, `IntrRef`, `StructCtor`, `StructComp`, `ArrElem`, `ArrElemC`, `ArrSection`, `Substring`, and `CArray`. Like the `MultiStmt` constructor, two of these do not represent 'F' primary expressions. The constructors `CArray` and `CArrElemC` correspond to pointer-based 'C' array and array element values. The `Stmt` ADT has a related constructor, `CArrayDecl`, which declares such arrays. These constructors together provide a flexible mechanism to declare, *initialise*, and use automatically-allocated 'C'-style arrays.

It is found useful for static memory allocation to identify arrays which are declared with a shape defined by constant values. Again the constant folder is employed, and an additional component is added to the `TypeDeclStmt` constructor of `Stmt`: an integer list corresponding to the array's shape, and signifying that static allocation may be performed. Should the list be empty, dynamic memory allocation is again the default. Character strings are treated similarly, though respectful of their status as scalars, it is now the `Type` ADT which receives a new constructor: `CharNType`; distinct from `CharType`.

4.2.3 The Offload C++ Abstract Syntax Tree

The third and final syntax tree represents a version of the original 'F' program suitable for direct serialisation into the Offload dialect of C++. Preparing the Offload C++ AST involves the *lowering* of 'F' language elements absent from C++, such as array expressions and the *case construct*, to a valid native C++ representation. C++ also provides the opportunity for

[8]The boolean components of the parse tree's `SubProg` ADT also perform this role.

compile-time metaprogramming using templates, and these must also be supported by, and utilised throughout, the AST.

The following code Listing 4.6 begins with a definition of the representation for a template argument, `TemplArg`. C++ template parameters may either be types, or constant integral expressions[9], and hence the binary type constructor, `Either`, from the Haskell prelude can be used.

```
1  type TemplArg = Either Type Expr
2
3  data Type
4   = CharType | ShortType | IntType | Size_tType | LLongType | FltType
5   | DblType | FltCmplxType | DblCmplxType | BoolType | EnumType Name | Void
6   | CharStringType  (Maybe Expr) (Maybe [TemplArg])
7   | CharNStringType              (Maybe [TemplArg])
8   | ObjTypeArrayT                (Maybe [TemplArg])
9   | ObjTypeArrayTN               (Maybe [TemplArg])
10  | ObjType         Name         (Maybe [TemplArg])
11  deriving (Show, Eq, Typeable, Data)
```

Listing 4.6: The Haskell ADT representing an Offload C++ type.

The representation of Offload C++ types relies on template arguments. The `Type` definition on line 3 starts with a number of representations for the simple *plain old data* types. Starting on line 6, five data constructors are then defined, each of which employs the `TemplArg` type. The `ObjType` constructor on line 10 represents the syntax of a generic object type, while the preceding `ObjTypeArray` and `CharStringType` constructors correspond to custom C++ objects which stand in for 'F' array and character types. `ObjTypeArrayTN` and `CharNStringType` are similar, though with objects optimised for a size specified as a compile-time constant; in fact a template argument. Each of the five data constructors make use of the `Maybe` type constructor. This is required *in addition* to a list of template arguments, with `Nothing` signifying a non-template object. Template types with defaults arguments for all parameters can then be represented by `Just []`.

The ADT representation of a C++ *program*, is lexically identical among all three syntax trees; and was seen earlier in Listing 4.1, The program unit components of a program are distinct, and defined at line 6 of Listing 4.7. Again the constructor definitions, `MainProg` and `Namespace`, are constituted by lists of statements and procedures; or in this case, functions. The corresponding `Function` definition appears at line 11. Table 4.1 relates each component of the `Function` definition, with the corresponding C++ function part. As indicated there by the description of the `[Init]` component, the `Function` type is also used to represent C++ methods, including constructors.

[9] E♯ does not have or require support for pointer and reference template parameters.

```
1  data TemplParam
2   = Typename            Name (Maybe Type)
3   | NonType        Type Name (Maybe Expr)
4   deriving (Show, Eq, Typeable, Data)
5
6  data ProgUnit
7   = MainProg       [Stmt]
8   | Namespace      Name [Stmt] [Function]
9   deriving (Show, Eq, Typeable, Data)
10
11 data Function
12  = Function       [TemplParam] Name ([AttrSpec],Type) [Init] [Stmt] [Stmt]
13  deriving (Show, Eq, Typeable, Data)
```

Listing 4.7: Haskell ADTs representing basic Offload C++.

Haskell `Function` component	Corresponding C++ construct
`[TemplParam]`	Template Parameters
`Name`	Function Name
`([AttrSpec],Type)`	Functions Qualifiers and Return Type
`[Init]`	Object Constructor Initialisation List
`[Stmt]`	Parameter List
`[Stmt]`	Function Body Definition

Table 4.1: Haskell `Function` constructor components and their C++ equivalents.

Template parameters are an optional component of a C++ function definition, and appear as the first parameter of the `Function` constructor definition. The corresponding `TemplateDecl` ADT definition appears at line 1 of Listing 4.7, Unlike template arguments, template parameters display an asymmetry, between their *non-type* template parameters, which must specify the type across which they range; and *type* parameters, which implicitly range across all C++ types, and so need not.

The asynchronous offload block is an expression, and is consequently represented by a constructor definition of the `Expr` ADT: `Offload [Prim] [Expr] ControlStmt`. Table 4.2 elucidates the correspondence between constructor components and Offload C++ constructs.

Haskell `Offload` component	Corresponding Offload C++ construct
`[Prim]`	Offload Block Arguments
`Expr`	Function Domain
`ControlStmt`	Compound Statement

Table 4.2: Haskell `Offload` constructor components and their Offload C++ equivalents.

4.3 Parsing with Monadic Combinators

Libraries of parsing combinators offer functional languages like Haskell an alternative to discrete parsing tools such as C/C++'s Flex (The Flex Project, 2007) and Bison (Free Software Foundation, 2007); Haskell's Alex (Dornan and Marlow, 2011) and Happy (Marlow and Gill, 2009); or the Java-oriented ANTLR (Parr, 2007). Parsing combinator libraries allow the production rules specifying a grammar to be encoded directly as a set of recursively defined combinators, thereby providing familiar access to the full range of features provided by the host language. In addition, similar combinators may be defined to implement *lexing*, integrating fully with the parsing combinators, and hence simplifying parser development by eliminating the requirement for a separate tokenising lexical phase. Furthermore, composition of parsing combinators is straightforward, allowing a parser created to handle, for example, C/C++ or Fortran, to be extended by an OpenMP parser, defined separately; or a Java Server Pages (JSP) parser composed from discrete Java and HTML parsers. Parsec is a *monadic* parsing combinator library for Haskell, used by the E♯ compiler.

The observation that a parser can be modelled as a monad was first described in Wadler (1990), and later in tutorial form by Hutton and Meijer (1998). A common observation is that a parser can be given the type, `Parser a`, shown on line 1 of Listing 4.8. A value of this type is a function accepting a character string as input, and returning a singleton list on success, or an empty list on failure[10]. On success, the singleton element is a tuple formed from an appropriate representation of the parsed structure, of type `a`, along with the remainder of the input string still to be parsed. A parser may return more than one tuple, in which case the parser is ambiguous; the 'F' parser, however, is unambiguous. Also, while the `Parser` type shown is restricted to `String` input types, this need not be the case.

The monad instance definition on line 11 of listing 4.8 provides the requisite *unit* (`return`) and *bind* (`>>=`) definitions expected by the Haskell `Monad` type class. In this formulation, `fmap` and `join` together facilitate the universal definition of `(>>=)`, valid for every monad. The parser's *unit* function, of type `a -> Parser a`, returns a simple parser which produces a singleton tuple of `x` and its unmodified input string, `i`. The *bind* combinator meanwhile, with signature `Parser a -> (a -> Parser b) -> Parser b`, can enable the *sequencing* of parsers.

Listing 4.9 defines the `program'` function which demonstrates how parsing combinator applications may be sequenced, using `(>>=)`, and applied to the task of parsing text corresponding to the top-level grammar production of an 'F' program. The syntax may initially

[10]Note the use of the term *parser*: each of the possibly numerous parsing combinators participating as *components* in a parser, are also referred to individually as parsers.

```
1  newtype Parser a = Parser (String -> [(a,String)])
2
3  parse (Parser p) = p
4
5  instance Functor Parser where
6    fmap f p = Parser $ \i -> [(f p',i') | (p',i') <- parse p i]
7
8  join :: Parser (Parser a) -> Parser a
9  join pp = Parser $ \i -> concat [parse pp' i' | (pp',i') <- parse pp i]
10
11 instance Monad Parser where
12   return x = Parser $ \i -> [(x,i)]
13   p >>= f  = join $ fmap f p
```

Listing 4.8: A simple parser monad.

```
1  program' = whiteSpace              >>= \_  ->
2             newlines                >>= \_  ->
3             program_unit            >>= \p  ->
4             many program_unit       >>= \ps ->
5             eof                     >>= \_  ->
6             (return $ Program p ps)
```

Listing 4.9: The 'F' `program` parser with explicit monadic bind.

appear unwieldy, however, compared to a non-monadic sequencing combinator, with signature `Parser a -> Parser b -> Parser (a,b)`, there is a pleasant lack of nested parse result tuples (Hutton and Meijer, 1996, pages 5-6). The use of the non-binding _ pattern in the nested lambda expressions signifies that each preceding parser is exercised only to enforce structural aspects of the grammar; the result of the parse is unused.

Haskell's `do` notation can further improve the readability of such functions. Listing 4.10 shows the `program` parser from E♯, equivalent to `program'`, though using `do` notation in place of explicit monadic bind and lambda expressions. As the entry point to the parsing combinators for the 'F' language, the `program` combinator has peripheral details not found elsewhere; for example the `whiteSpace`, `newlines`, and `eof` combinators appear directly only in this function. Excepting such aspects, the central structure of lines 3 and 4 is highly similar to the corresponding BNF *`program`* grammar production; seen earlier in Figure 4.3.

The `program` parser employs a number of useful combinators. A *lexeme* parser will consume successive *white space* following a successful parse, and many of the combinators defined for the E♯ parser are so designed. The 'F' language, however, does not include new lines in its definition of white space. 'F' has no line termination symbol, such as a semicolon, and expects explicit carriage returns whenever implicated by the grammar. The `whiteSpace`

```
1 program = do whiteSpace
2              newlines
3              p  <- program_unit
4              ps <- many program_unit
5              eof
6              return $ Program p ps
```

Listing 4.10: The 'F' `program` parser using `do` notation.

parser used in Listing 4.10, then, begins by consuming any white space which exists prior to either a carriage return, or program unit. In fact, multiple blank lines may occur before the first program unit, each parsed by a lexeme parser matching a new line terminal. A version of Parsec's `lexeme` combinator[11], modified for non-consumption of new lines, can be applied to an existing parser, transforming it into a lexeme parser. The `newlines` parser shown in Listing 4.11 uses the `lexeme` combinator; and also Parsec's `many` combinator. Applying `many` to one parser, produces a second parser which matches zero or more of the first parser's input, returning the appropriate result list. The sole use of the `newlines` parser is restricted to its supporting role here at the start the `program` function. Parsec's `many1` combinator is similar to `many`, however it will apply the original parser *one* or more times. The `newlines1` combinator, again a lexeme parser, also matches one or more new lines, though unlike `newlines`, it is used frequently throughout the 'F' parser.

The `p` and `ps` bindings, from lines 3 and 4 of Listing 4.10, have type `ProgUnit` and `[ProgUnit]` respectively, where `ProgUnit` is a type from the parse tree, seen in Listing 4.1. Following `eof`, Parsec's *end of file* parser, the `return` function *lifts* the `Program` value formed from `p` and `ps` into the appropriate parser monad type.

```
newlines  = many  $ lexeme newline
newlines1 = many1 $ lexeme newline
```

Listing 4.11: Composition of parsing combinators.

The type of the parser used by Parsec is more generic than the parser type presented thus far. Constructed as a *monad transformer* (O'Sullivan et al., 2008, chapter 18), `ParsecT` parametrises not only the input stream type, but also the type of the *user state*. User state emulates the effect of a global variable, delimited by the dynamic extent of the enclosing monad. Manipulated through functions such as `putState` and `modifyState`, user state is implemented via a mechanism analogous to that of the *state monad* (Wadler, 1990). One obvious application of this facility is to construct a symbol table. For E♯, however, it was

[11] Many of Parsec lexical combinators were modified for E♯ due to their reliance on a definition of white space which includes carriage returns.

decided that the readability of the parser suffered by the inclusion of such details, which were duly postponed.

4.3.1 Predictive Parsing

Parsec facilitates the creation of top-down parsers implemented by the method of recursive descent. Parsers built using Parsec will also by default perform no *backtracking*, and are known as *predictive parsers*. Top-down, predictive parsers are often referred to as LL(1). Parsec is also referred to as an LL(k) parser, indicating that up to k tokens from the incoming parse stream can be examined to resolve ambiguity. The precise value of k, however, may be unknown, and unbounded, leading Terence Parr to categorise (Parr, 2007) such parsers as LL($*$). The facility to perform such unbounded *lookahead* in Parsec, however, must be performed *explicitly* by the user; with combinators such as `try`. By default, Parsec will look ahead only by one lexical token, say an 8-bit character; if this forms a partial match, it is then *removed* from the input stream.

The deterministic choice combinator, `(<|>)`, is used frequently by the 'F' parser module of the E♯ compiler. Given two parser arguments, `a` and `b`, the composite parser `a <|> b` will first attempt to match the input stream using parser `a`. If this succeeds, the result of `a` is returned. On the other hand, if `a` fails to match its input, and has consumed no input, the second parser, `b`, is tried. If `a` *has* consumed input, the composite parser fails entirely. The `(<|>)` combinator represents a similar concept to the BNF *alternative* operator; denoted either `|`, or as earlier: `or`.

```
module'' =     public_module
           <|> private_module
```

Listing 4.12: First attempt at an 'F' module parser.

Listing 4.12 uses the choice operator to produce a faithful rendering[12] of the equivalent BNF production, *`module`*; shown earlier, on line 17 of Figure 4.4. The 'F' grammar, however, in the worst case, requires the examination of an unbounded sequence of tokens from the input stream to discriminate between alternate grammar rules. The `public-module` and `private-modules` grammar productions *both* start by matching a `module` statement, followed by multiple, optional, `use` statements. The `public-module` production rule *must* then match with a `public` statement; any other match signifies a `private` module has been found. Matching a `private` module will therefore require that all tokens removed from the input

[12]The hyphens used in the 'F' grammar rule names, are replaced by dashes in the names of the E♯ parser combinators. Haskell reserves the use of the hyphen as a subtraction operator.

stream by preceding, successful matches within the `public-module` parser, be reinserted, prior to a fresh match attempt by the `private_module` parser. To recover an earlier state of the input stream, Parsec's `try` combinator can be employed.

```
module' =     (try public_module)
          <|> private_module
```

Listing 4.13: The parser to match an 'F' module.

The `try` combinator will apply its argument, a parser, `p`, to the input stream. On success, the result of `p` is returned. On failure, the input stream is restored to its state prior to the match attempt. Backtracking is a computationally expensive operation, and the use of an explicit combinator holds the potential to exploit a user's specific knowledge of a grammar. On the other hand, it compromises the readability of the parser. The parser used by E♯ to recognise an 'F' module, shown in Listing 4.13, makes use of the `try` combinator to backtrack from the partial matches of the `public_module` parser.

On other occasions, `try` may be less applicable. A useful combinator in this situation is `notFollowedBy'`[13]. `notFollowedBy' p` will try to match the input with parser `p`, and *fail* if it does so, while consuming no tokens from the input stream. Listing 4.14 provides the definition, using the Haskell prelude function, `fail`, instantiated by the `ParsecT` monad to the internal Parsec function, `parserFail`, which creates a parser which always fails.

```
notFollowedBy' p = try $ (try p >> fail "notFollowedBy'") <|> return ()
```

Listing 4.14: `notFollowedBy'` fails on the success of its parser argument.

Another situation where a straightforward mapping of the 'F' grammar into Parsec combinators, is not possible, is the list order of choices between grammar symbols. LL grammars are non-ambiguous (Leijen and Meijer, 2001), and consequently an aggregate parser `(a <|> b)` would not attempt a match with parser `b`, should parser `a` succeed. The order of `(<|>)` parser operands are therefore sorted to ensure those production rules in the 'F' grammar requiring larger lookahead, occur earlier. For example, the parser corresponding to the 'F' *`pointer-object`* production shown in Figure 4.5, must reverse the relative order of the two non-terminals, as shown in Listing 4.15. Without this alteration, `structure_component` could never match. Parse errors would then soon arise from the unmatched lexical tokens of each structure component, mismatched as a variable name, remaining in the token stream.

[13] `notFollowedBy'` differs from the Parsec `notFollowedBy` by handling parsers with a result type which is not an instance of the `Show` type class, allowing, for example: `notFollowedBy'(reservedOp "=")`.

```
pointer-object is variable-name
               or structure-component
```

Figure 4.5: The *pointer-object* production rule from the 'F' grammar.

```
pointer_object =     try structure_component
                 <|> do n <- variable_name
                        return $ UnknownName [] n
```

Listing 4.15: The parser to match an 'F' pointer object.

Other 'F' grammar symbols cannot be distinguished without contextual information. For example, "var" could be either a named constant or a variable. The BNF *primary* production rule lists both possibilities as alternative non-terminal symbols. In the corresponding Haskell parser, both combinator references appear too. Though only the first has a chance of matching, both return the same `UnknownName` type on success. For each grammar rule, the E♯ parser has a corresponding parser combinator definition. Assuming access to the 'F' BNF grammar and documentation, the intention is to enhance maintainability, and readability. The presence of such ambiguity in the formal grammar is likely also to enhance readability, and relies on the presence of an informal, supplementary commentary in the periphery of the grammar text. A separate object binding pass ensures E♯ attributes the appropriate qualities to each name.

An LL parser requires that no *left-recursion* is present in the grammar. Most of the 'F' grammar is provided in extended BNF (EBNF), which adds regular expression support to BNF, including operators for repeated symbol patterns. Use of these operators allows the substance of the 'F' grammar to avoid explicit recursion, whether left or right-based. Nevertheless, expressions, and particularly arithmetic expressions, are defined therein using left-recursion, which will lead to non-termination in LL parsers. Solutions to this problem include left-factoring the grammar, which affects readability; or using `chain` combinators; see Hutton and Meijer (1996). The `chain` combinators must partition a parse into operators of equal precedence; and associativity, whether left or right. The entirety of such functionality is provided by the Parsec utility combinator, `buildExpressionParser`, and used within E♯ to define the expression parser shown in Listing 4.16. The first argument to `buildExpressionParser` is a table, much like that of Metcalf and Reid (1996, page 40), with operators listed in decreasing order of precedence. The second argument, a parser, is used to match the terms of the expression.

The `opL`, `opR`, and `opP` combinators listed in the `where` clause of Listing 4.16 are used soley to improve legibility. `add_op'` exists to assist with the overloaded 'F' grammar rule, `add-op`,

```
expr = buildExpressionParser table primary

table = [ [opP defined_unary_op ],
          [opR power_op         ],
          [opL mult_op'         ],
          [opP add_op'          ],
          [opL add_op           ],
          [opL concat_op        ],
          [opL rel_op           ],
          [opP not_op           ],
          [opL and_op           ],
          [opL or_op            ],
          [opL equiv_op         ],
          [opL defined_binary_op] ]
 where
   opL p    = Infix  (p >>= \o -> return $ BExpr [] o) AssocLeft
   opR p    = Infix  (p >>= \o -> return $ BExpr [] o) AssocRight
   opP p    = Prefix (p >>= \o -> return $ UExpr [] o)
   add_op'  = add_op >>= \o -> case o of Add -> return Plus
                                        Sub -> return Minus
   mult_op' = try $ mult_op                 >>= \mo ->
                    notFollowedBy' (char ')') >>= \_  ->
                    return mo
```

Listing 4.16: The 'F' expression parser created with `buildExpressionParser`.

which will match the strings "+" and "-", either as a unary prefix, or a binary infix operator. As the parse tree distinguishes between such types, `Add` and `Sub` results are mapped to `Plus` and `Minus` respectively. The solution presented supports the 'F' grammar, and makes similar dual use of the existing `add_op` parser.

The parser for multiplication and division operators, `mult_op`, is unsuitable for use within an expression context. The division operator followed by a right parenthesis is lexically the same as the closing delimiter of an 'F' *array constructor*, such as `(/1,2,3/)`. Many parsers for 'F' operators must consider successive lexical tokens, and make use of `notFollowedBy'`. For example, `<`, represents the *less than* operator only when not followed by `=`. The division operator, however, *can* be followed by a right parenthesis when used as part of a *generic specifier*; for example, `operator (/)`. Therefore it is only here, within the context of an expression, that this is prohibited by the operation of `mult_op'`. Again, code and grammar coverage are emphasised by reusing `mult_op`.

4.3.2 Combinators for Parsing 'F'

A few additional combinators were created to provide custom assistance with the task of parsing 'F'[14]. First of all, 'F' is case insensitive. A basic combinator, `nocase`, is thus defined, akin to the Parsec `string` combinator, accepting a Haskell `String` as its argument, though matching without regard for the letter case.

The 'F' grammar often treats white space separating pairs of non-terminal symbols as optional. A combinator, `gapped`, given two string arguments, will match the same strings as input, either with or without intervening white space. For example, an 'F' *end program statement*, appearing either as `end program p`, or `endprogram p`, can each be parsed using `gapped "END" "PROGRAM"`. A version for situations where white space *must* be present is also defined, `gapped1`; for example, `gapped1 "MODULE" "PROCEDURE"`.

4.4 Object Binding

Translating the parse tree into a typed 'F' AST first requires that the former's symbol table components, the `SmT` values[15], become properly initialised. Symbol tables are present at most nodes in the parse tree, such as the program unit of Listing 4.2. Subprograms are children of the program unit and, as with most node values, have their own symbol tables. These will often be larger than the symbol tables of their parents due to formal parameters, and additional use statements. The placeholder primary expression values, with ADT shown previously in Listing 4.4, will be transformed by the complete type information obtained from the symbol table.

As the parse tree prepares to initialise the typed 'F' AST, it is traversed a number of times to correctly prepare the symbol tables. Three significant passes, implemented by bottom-up traversal, are performed in the following order:

1. Basic initialisation of module and subroutine symbol tables only. The order of entries is significant. For example, the name of a local function variable may be the same as a module variable;

2. For each module or subroutine symbol table, add the non-private entries, brought into scope by use statements; and lastly

[14] `notFollowedBy'`, `newlines`, and `newlines1` have already been discussed.

[15] The Haskell `SmT` type, shown earlier in listing 4.3, is a type synonym for an association list: with the key, a name; the value, a type.

3. Initialise the symbol table component of all remaining descendant nodes to that of their closest module or subroutine ancestor node.

Derived types containing array or character members are output as 'C' template structures by the back end. Subroutines with such types as parameters are flagged by a further pass performed at this stage. Another pass ensures declarations with constant shape also provide an appropriate indication, including extents for each dimension[16].

Having prepared the symbol tables throughout the parse tree, names are then assigned types through the process of *object binding*, with the assistance of the `lookup` function from the Haskell prelude. These changes reflect the requirements of the 'F' AST, which is produced from the final traversal of the parse tree.

As this final traversal changes the Haskell AST type, it is convenient to define a type class, `OB`, involving *two* parameters; thereby requiring the Haskell extension, `MultiParamTypeClasses`. The type parameters of `OB` correspond to the domain and range types of its sole function, `ob`. The definition of `OB` is shown in Listing 4.17.

```
{-# LANGUAGE MultiParamTypeClasses #-}

class OB a b where
  ob :: a -> b
```

Listing 4.17: The $E\sharp$ object binding type class.

An instance of the `OB` type class is defined for each ADT of the parse tree. For each instance, the names of the domain and range types are the same; excepting their base module; in this case either `PT` or `Fast`. As was the case in the recursive bottom-up traversals involved in the preparation of the parse tree symbol tables, most `ob` instances are boilerplate, and merely facilitate the update of the expression, and primary expression values; that is, the *leaves* of the AST. Listing 4.18 demonstrates an `ob` definition for such a constructor: `PT.UnknownName`; `PT.UnknownApp` and `PT.UnknownComp` are here omitted. In this instance, `ob` is defined for primary expressions, with a domain of type `PT.Prim`; and a range of type `Fast.Prim`.

The primary expression instance for the `OB` type class provides a source for rank information. In Listing 4.18, the rank of the named variable can be fully evaluated, allowing this information to be propagated upwards to other 'F' AST types, such as `Expr`. The call to `lookup` on line 3 may produce a type declaration statement, including a list of attribute specifications. The omission of a dimension specification from such a list signifies a scalar variable, represented by `Fast.VarName`; as shown on line 8. When a dimension attribute *is* present, the

[16]The Haskell types involved here were described in Section 4.2.2.

```
1 instance OB PT.Prim Fast.Prim where
2   ob (PT.UnknownName st n) =
3     case lookup n st of
4            Just (PT.DsSmData (PT.TypeDeclStmt _ t as _)) ->
5              let t'  = ob t
6                  as' = map ob as
7                  ss' = [ss | Fast.DimensionSpec ss <- as']
8              in case ss' of [ ] -> Fast.VarName    n 0          t' as'
9                              [s] -> Fast.ArrVarName n (length s) t' as'
```

Listing 4.18: An `OB` type class instance for a primary expression constructor.

non-zero integer rank of the array variable is obtained, on line 9, from the length of its shape specification list; and the `Fast.ArrVarName` constructor is invoked.

```
1  instance OB PT.Expr Fast.Expr where
2    ob (PT.ExprPrim st p)       =
3      let p'           =  ob p
4      in Fast.ExprPrim (getRankP p') (getTypeP p') p'
5    ob (PT.UExpr    st o e)     =
6      let (e',o')      = (ob e,  ob o)
7      in Fast.UExpr (getRank  e') (getType  e') o' e'
8    ob (PT.BExpr    st o e1 e2) =
9      let (e1',e2',o') = (ob e1, ob e2, ob o)
10     in Fast.BExpr (max (getRank e1') (getRank e2'))
11                   (chooseType o' e1' e2') o' e1' e2'
```

Listing 4.19: Part of an `OB` type class instance for an expression.

Listing 4.19 shows the `OB` instance definition for three `PT.Expr` constructors. The bottom-up propagation of rank information can be observed in the use of the named field accessor functions `getRank`, and `getRankP`; provided automatically for Haskell types declared using record syntax. The type of values given to these functions originate within the requisite 'F' AST module, and are produced by each application of the `ob` function to the components of the input constructors. Line 10 uses the Haskell prelude's `max` function to calculate the rank of an expression formed from a binary intrinsic operation, with operands of potentially differing ranks. Likewise, the `getType` and `getTypeP` record functions help to assign `Fast.Type` components in the expression constructors. The `chooseType` function of line 11 follows the 'F' and Fortran conventions regarding *mixed-mode expressions*, as described by the rule tables of Metcalf and Reid (1996, pages 32-33), and also INCITS/J3 (2010, page 140).

4.5 Translating into C++

This section begins with a description of the overall strategy of representing 'F' language constructs within a C++ AST, along with the code generation mechanism necessary for file output. Following this, a selection of 'F' constructs are examined in turn, concluding with a more thorough examination of the implementation of array expressions. Additional program transformations are required to ensure that nested array expressions and functions are properly dispatched, however these will be discussed in detail in the following Section 4.6.

'F' and C++ provide comparable levels of abstraction, and it has been feasible to avoid the comprehensive phase of lowering found in compilers producing assembly or object file output. Conversion from an 'F' to a C++ AST is controlled by a bottom-up traversal of the program tree, applying a transformation function to each node. A type class, `F2C`, facilitates an ad hoc polymorphic interface for this transform function, `f2c`. The `F2C` type class is in fact isomorphic to the object binding type class, `OB`, shown in Listing 4.17, and similarly, an instance of the `F2C` class is defined for each ADT involved. In some cases, this will simply involve replacing each `Fast` constructor with its counterpart from the `Cast` module; and applying `f2c` to each child. In other cases, an equivalent C++ construct will not exist, and a more elaborate representation, perhaps involving a compound statement may be required.

Once the program is represented as a C++ AST, the final stage is code generation. For this, the Haskell *pretty* package (Jones and Hughes, 2011) is used. This package is a revised implementation of the algebraically-derived combinator library design described in Hughes (1995). The API of the pretty package normalises the representation of each program fragment to a document type, `Doc`, and defers the textual concatenation of document data to a final render function such as `renderStyle :: Style -> Doc -> String`.

```
1 {-# LANGUAGE OverloadedStrings #-}
2 import GHC.Exts( IsString(..) )
3
4 instance IsString Doc where
5  fromString = text
6
7 ppC ContStmt = sep ["continue", semi]
```

Listing 4.20: Pretty printing a C++ continue statement.

As an example, the C++ continue statement, represented by the constructor, `Cast.ContStmt`, can be processed using the `ppC` function on line 7 of Listing 4.20; which has type: `Cast.ControlStmt -> Doc`. The `sep` combinator will separate the abstract documents listed by its argument, either vertically using new lines, or horizontally using spaces, according to the *style* argument

provided to the rendering function. The GHC *overloaded string literals* extension allows the overloaded `fromString` function to be applied automatically to string literals found in locations where the Haskell compiler expects a type which is an instance of the `IsString` class. E♯ provides the simple `IsString` instance for the `Doc` type shown on line 4, using the combinator `text :: String -> Doc`. Hence, in this context, the string literal, `"continue"`, becomes syntactic sugar for `text "continue"`. The `semi` combinator creates a value of `Doc` type, representing a semicolon.

```
class PPC a where
  ppC :: a -> Doc
```

Listing 4.21: The E♯ pretty printing type class.

A bottom-up traversal of the abstract C++ program representation is sought to construct a value of type `Doc`. Here E♯ again uses a type class, `PPC`, to provide an intuitive ad hoc polymorphic interface. As a `Doc` is always produced, a traditional single parameter type class is sufficient, as shown in listing 4.21.

4.5.1 Basic Types

Integer and floating-point types capable of representing those of 'F' are available natively in C++. The most prominent typing feature present in 'F', though *absent* in C++, is the system of kinds, typically used to select the size in bytes of each type. An initial implementation of 'F' kinds was entirely internal to the E♯ compiler. Hence, an 'F' `real` type, with a kind of 8, became a C++ `double`; while a kind of 4 would induce a `float` instead. A C++ `typedef` could then be used to provide a syntactic alias more familiar to a Fortran user; `real8` and `integer8`, for example.

An alternative approach was, however, ultimately adopted in which the kind, a compile-time constant in 'F', becomes a template argument for a set of specialised template classes. The primary advantage of this method is that a type's kind may be inspected, and even calculated, at compile-time using the template metaprogramming idiom known as *type traits* (Lippman, 1997, pages 451-457). By this route, specialised overloaded versions of functions may be automatically selected by the C++ compiler. This aspect is fully explained in the `matmul` example of Section 5.3.

A C++ type representing an 'F' type is then obtained with the pairing of a base template class, whether `FintegerT`, `FrealT`, `FlogicalT`, or `FcomplexT`; and a constant integer: the kind. For example, a real type with a kind of 4 is obtained from a specialisation of the

`FrealT` class, wherein the member typedef, `type`, is assigned to a `float`. C++ syntax such as `FrealT<4>::type` may then be used; or the more concise `Freal4` via a global `typedef`.

In C++, `char` is the only fundamental type with a prescribed width (Becker, 2011, page 110): 1 byte. A conventional implementation of kinds for the 'F' integer family therefore requires library support. The set of *signed* integer types included with the C99 standard width integer library, and accessed through `stdint.h`, or `cstdint`, provide the necessary definitions.

Unlike integer and floating-point types, a family of C++ boolean types does not exist. Furthermore, the C++ `bool` type has a size of *one byte* under GCC whereas the GFortran runtime library provides many library functions only for `logical` type widths of four. The standard width integer library is used again, this time the set of *unsigned* integer types are employed. For example, a logical type, with a kind of 8, would be represented by a `uint64_t` type; obtained through `FlogicalT<8>::type`, or `Flogical8`. Automatic type promotion ensures C++ boolean literals may still be assigned to variables of such types.

No standard library exists to provide fixed size floating-point types. This is, however, far less of an issue than with integers. The relevant C++ types, `float` and `double`, are routinely 32 and 64 bit floating-point types adhering to the IEC 559[17] standard conventions[18]. While higher precision floating-point types are feasible, 'F' requires only that one kind with higher precision than the default is provided (Metcalf and Reid, 1996, page 13). Requirements for the real types of kind 4 and 8 are therefore met by the single and double-precision C++ floating-point types.

The 'F' `complex` type is also represented using the kind template class pattern. The underlying C++ `typedef` class data member is then set to the templated `complex` type provided through the `complex` header of the C++ standard library.

Arrays in 'F' correspond to far more than a memory address, and consequently are inadequately represented by a simple C++ pointer type. The underlying data structure of an 'F' array is known as a *dope vector*, and no standard yet governs their structure. The low-level 'C' API, Chasm (Rasmussen et al., 2006), provides the necessary boilerplate code to interface with the dope vectors of most Fortran runtime libraries. Higher-level template class abstractions, `ArrayT` and `ArrayTN`, have been developed for E♯, and are described in Section 5.2. It is, however. useful to mention here a few aspects upon which E♯ depends. The `ArrayT` and `ArrayTN` classes have four and five template parameters respectively. The final integer parameter of each corresponds to the memory space occupied by the array data: the

[17]The international standard for floating-point values, IEC 559, is also an American standard: IEEE 745.

[18]The `numeric_limits::is_iec559` template, a type trait from the C++ standard header, `limits`, allows a compile-time test for IEC 559 support at specific types.

offload depth, either inner or outer. Of all the template parameters, this will uniquely depend on the execution context. All common non-elemental 'F' array operations, such as element access; sectioning; assignment; and serialisation are supported by appropriate methods of the `ArrayT` class. Elemental operations on arrays require additional support for parallelism.

The 'F' `character` type represents strings. Two custom class templates, `FChar`, and the automatically-allocating variant `FCharN`, have been developed, with appropriate class methods defined for most common 'F' string operations, such as assignment; serialisation; and relational operations. Like the array classes, the character classes also contain pointers to memory allocated from the heap. Consequently, the offload depth must also be provided as a template argument to a C++ character object.

4.5.2 Pointers

Pointers to scalar values in 'F' are distinct from those in C++ in two ways. Firstly, a pointer target in 'F' must be declared with the `target` attribute. Secondly, no indirection operator is provided, nor required, to address the target of an 'F' pointer; a distinct *pointer assignment* operator, `=>`, can re-target a pointer. Listing 4.22 demonstrates the use of a pointer to assign the `real` variable `r` to 9.

```
1 real, pointer :: pr
2 real, target  :: r
3 pr => r
4 pr = 9
```

Listing 4.22: Scalar pointers in 'F'.

```
Freal4 * pr;
Freal4 r;
pr = & r ;
(*pr) = 9;
```

Listing 4.23: Scalar pointers in C++.

The translated C++ version in Listing 4.23 corresponds closely to Listing 4.22. Again a scalar pointer is declared, but now the pointer assignment operator of line 3 becomes a conventional C++ assignment, with the target preceded by the *address-of* operator. Each scalar object accessed through an 'F' pointer must, in the C++ translation, be provided as an argument to the *indirection* operator; as seen on line 4.

If the target of an 'F' pointer is of derived type, the translation of the 'F' *component selector*, (`%`), must choose the appropriate C++ class member access operator: arrow (`->`), when the parent object is a pointer; otherwise dot (`.`) should be used.

An 'F' pointer to an array contains information in addition to that stored by its target. For example, an array pointer targeting an array section will have a lower bound of 1 for each dimension, irrespective of the bounds of the target. Therefore, while a traditional C++ pointer

to an object of type `ArrayT` *could* adequately represent an 'F' pointer targeting an *array*, from `p => a` for example, a solution for the general case will require that a new `ArrayT` object is created, with its bounds and data pointer assigned appropriately.

Implementing array pointers in E♯ begins by creating an `ArrayT` stub object. The parameter-free constructor of `ArrayT` sets no dimension bounds, and performs no heap allocation. The `fromArray` method can then be utilised to set the object's attributes to that of the sole array argument. Similarly, for array sections, the `fromArraySection` method was developed, and accepts integral stride, plus lower and upper bound arguments for each dimension, in addition to the array argument.

```
1 real, pointer, dimension(:)    :: pra
2 real, target,  dimension(0:99) :: ra
3 pra => ra
4 pra => ra(:)
5 pra(100) = 9
```

Listing 4.24: Array pointers in 'F'.

```
1 ArrayT<C,Freal4,1,Od> pra;
2 ArrayTN<C,Freal4,1,100,Od> ra (0,99);
3 pra.fromArray (ra) ;
4 pra.fromArraySection (ra,0,99,1) ;
5 pra(100) = 9;
```

Listing 4.25: Array pointers in C++.

Listing 4.25 demonstrates the E♯ translation[19] of Listing 4.24. Note the access to element 100 of the array pointer `pra` on the 5th line of each; made valid by the pointer assignments of an array section on the 4th lines. Line 5 of both listings also assigns the last element of `ra`, which is `ra(99)`, to 9. In contrast, the redundant pointer assignments on the 3rd line of each listing, assigns `pra` to the same extents as its target: 0-99.

4.5.3 Derived Types

A derived data type in 'F', presented in Section 3.3.6, is essentially a tuple of types. Such user-defined types in 'F' provide similar functionality to the record, or structured types of 'C'; and this affinity is exploited by E♯. It is not, however, sufficient to replace an 'F' type

[19]The first template argument of the array objects in listing 4.25 is a global enumerated constant corresponding to a compiler. `F90_NAG_C`, for example, prepares array descriptors compatible with the NAG Fortran compiler runtime.

declaration statement with a C++ structure declaration: additional functionality is provided by the 'F' compiler for every derived type.

The solution adopted by E♯ is to provide the missing functionality by automatically generating individual C++ operator overloads for the derived types translated to C++. This design choice adheres to a general principle: emphasising the migration of local, compiler-generated code, towards the runtime library. This approach not only simplifies the E♯ compiler, but also improves the readability of the generated C++.

Before we turn our attention to the generated methods, a second, pervasive concept relevant to the representation of derived types must be revised. As mentioned in Section 4.5.1, an integer template parameter corresponding to the memory space occupied by a container's elements is used by E♯ template classes such as `ArrayT`, `FChar`, and their statically-allocated variants. In fact, any C++ class containing such types, will often *itself* require a similar template parameter.

```
template <Finteger4 Od>
struct StrArr {
  FChar<char,Od>                    str;
  ArrayT<F90_GNU_C,Finteger4,1,Od> arr;
};
void q() {
  StrArr<OFFLOAD_DEPTH> sa;
}
```

Listing 4.26: Pervasive depth templates in classes.

For example, the function `q1` of Listing 4.26 may be called from within either an inner, or an outer context. Hence, `OFFLOAD_DEPTH` may be instantiated to 0 or 1. Consequently, to ensure the array and string members of the `sa` object can be properly initialised, the `StrArr` class must *also* be given an integer template parameter, `Od`. It follows that any class generated by E♯ to represent an 'F' derived type, must be given an integer template parameter, *whenever the original type contains an array or character component*. Such types may be described as in need of a depth template; or NDT types. Furthermore, any derived type which contains an NDT type, is also an NDT type; and must too receive the requisite integer template parameter[20].

Unlike C++ classes, an 'F' derived type is automatically serialisable and deserialisable. The common C++ *insertion* operator, `<<`, overloaded for the C++ standard library output stream class, `ostream`; and the *extraction* operator, `>>`, overloaded for the equivalent input class,

[20]The integer template parameter must be consistently named. The name `Od` is chosen as it relates to the Offload depth.

```
type, public :: vec2
  integer :: x,y
end type vec2
```

Listing 4.27: A 2-tuple as an ‘F’ derived type.

`istream` are already defined for basic types, and may be further overloaded on their second parameter[21]. E♯ generates an overload for both of these stream operators, as *friends* of the class created to represent each derived type. Lexicographical ordering is then used to sequence the input and output of derived type components. For example, the ‘F’ derived type 2-tuple shown in listing 4.27, would be translated into the C++ `vec2` class of Listing 4.28; with the streaming IO operators at lines 4 and 8.

```
1  struct vec2 {
2    inline vec2 () {};
3    inline vec2 (const Finteger4 &x,const Finteger4 &y) : x(x),y(y) {};
4    inline friend ostream & operator << (ostream &o,const vec2 &__this) {
5      o << __this.x << ' ' << __this.y;
6      return o ;
7    };
8    inline friend istream & operator >> (istream &i,vec2 &__this) {
9      i >> __this.x >> __this.y;
10     return i ;
11   };
12   Finteger4 x,y;
13 };
```

Listing 4.28: An ‘F’ 2-tuple derived type translated into C++ by E♯.

A *structure constructor* is also implicitly created for each ‘F’ derived type. A structure constructor creates an object of derived type, an *rvalue*, analogous to a derived type literal. It is applied as a function, named after the derived type, with each parameter typed as each component. For example, `vec2(3,4)`, applies the structure constructor implicitly created by the `vec2` type of Listing 4.27. This functionality is added to the class by generating a C++ constructor with the same signature and functionality. An efficient implementation using an initialisation list for all arguments is adopted to minimise the creation of temporary values. Line 3 of Listing 4.27 presents a constructor of this kind for the `vec2` class.

The presence of such an *explicit* C++ constructor, however, halts the *implicit* creation of the default, *nullary* constructor of the class. An unfortunate, related point is that the element type of a C++ array *must* have a nullary constructor. Consequently a nullary constructor constructor must also be automatically generated by E♯. The body of this constructor should

[21]The E♯ C++ array and character classes also have both streaming IO operators defined.

also initialise the bounds of any array members; and may be required to allocate memory for array and character members. The nullary constructor for the `vec2` type is shown on line 2 of Listing 4.28.

```
type, public :: vec2p
  integer :: x,y
  integer, pointer, dimension(:) :: pa
end type vec2p
```

Listing 4.29: An 'F' derived type with an array pointer component.

An 'F' compiler provides an assignment operator, =, valid for two values of the same derived type. Similarly, in C++, an assignment operator is implicitly created for each class; the *copy assignment* operator. For E♯, we must also facilitate assignment between values of NDT types, instantiated with *differing* integer template arguments. In such scenarios, an explicit templated assignment operator must be generated. The derived type `vec2p` of Listing 4.29, for example, combines an array pointer with two integer components. The generated C++ template operator for assignment between `vec2p` values is shown on lines 1-6 of listing 4.30. Note that for 'F' components with the *pointer* attribute, the appropriate component-wise operation is *pointer assignment*, rather than common assignment. As discussed in Section 4.5.2, this is implemented in C++ by E♯ using the `fromArray` method; demonstrated on line 4.

```
1  template <Finteger4 Od2> inline vec2p &operator=(const vec2p<Od2> & rhs) {
2    x = rhs.x;
3    y = rhs.y;
4    pa.fromArray (rhs.pa) ;
5    return (* this) ;
6  };
7  inline vec2p &operator=(const vec2p & rhs) {
8    x = rhs.x;
9    y = rhs.y;
10   pa.fromArray (rhs.pa) ;
11   return (* this) ;
12 };
```

Listing 4.30: The generated assignment operators of NDT type `vec2p`.

Despite the presence of an explicit template assignment operator in an NDT type, the default, implicit copy assignment operator is still provided by the C++ compiler. Furthermore, for two values instantiated with the same template arguments, the overload resolution mechanism of C++ will select the implicit, non-template copy assignment operator. This default copy assignment operator, however, merely copies each data member using its own copy assignment operator. This is adequate for scalar pointers, corresponding as it does to the

requisite ‘F’ pointer assignment operation. For array pointers, the `fromArray` method is required. A second overload, of the copy assignment operator, is therefore also generated in these situations; as shown on lines 7-12 of Listing 4.30 for the `vec2p` type.

4.5.4 Functions and Subroutines

Functions and subroutines in ‘F’ declare their parameters as part of the *procedure-specification*, prior to the *execution-part*. While the order of these declarations is arbitrary, the list of dummy argument names provided to the preceding subroutine or function statement is not, and must be honoured by each function application or subroutine call. Both ‘F’ procedure forms may be represented as C++ functions, and E♯’s translation begins by moving every declaration, which binds a name from the list of dummy argument names, into the list of formal C++ function parameters. While a declaration, both in ‘F’ and C++, may include multiple objects, a C++ function parameter declares only one. Hence multiple parameter declarations in a single statement must also first be separated.

Functions in ‘F’ return values named in the *result clause* of a function statement. By locating this name among the function’s declaration statements, its type may be obtained, and so the return type of the generated C++ function be specified. The result name is used once again, as the argument to a C++ *return statement*, appended to the list of statements comprising the generated function. An ‘F’ subroutine returns nothing, and hence the C++ return type of a translated ‘F’ subroutine is `void`.

By default, C++ uses a call-by-value evaluation strategy for function arguments; however, the call-by-reference strategy of ‘F’ may be obtained by using C++ reference types as the formal parameters for each translated procedure. In ‘F’, each procedure parameter declaration must specify an *intent*; see Section 3.3.5. Parameters specified with an `intent(in)` attribute are not modified, and hence may be translated to equivalent C++ reference types qualified by the `const` qualifier.

An ‘F’ procedure with one or more NDT parameter types[22] must be translated into a template function, with a single integer template parameter; representing the offload depth[23]. This allows for proper instantiation of the template arguments. As with NDT *types*, procedures having at least one NDT type parameter may also be classified as NDT. Furthermore, any procedure which calls an NDT procedure may too be classified as NDT. In terms of imple-

[22] NDT types were introduced in Section 4.5.3.

[23] The use of the special Offload C++ integer constant, `OFFLOAD_DEPTH`, as a default template argument for NDT types, cannot avoid the requirement for template functions. In a declarative setting, such as the formal parameter list of a function, `OFFLOAD_DEPTH` is always instantiated to 0.

mentation, E♯ in fact generates *all* functions as template functions of one integer parameter. The implications of this modest redundancy leads to only a slight increase in compilation times, and no effect on runtimes. Local NDT declarations are also instantiated using the same integer parameter, `Od`.

This representation of NDT procedures as C++ template functions of a single integer parameter assumes that in each case, all function arguments are correctly instantiated by this one integer; the offload depth. The correctness of this can be recognised intuitively by considering that only one address space exists in the 'F' language. Only at the boundary that is an offload block are the two memory spaces manipulated together; for example, DMA transfers invoked by an assignment between arrays of differing memory locality.

Function *application* in 'F' corresponds to the same in C++. The call statement, responsible for calling an 'F' subroutine, is represented as a C++ *expression statement* of one function application. When a C++ NDT function application requires an integer template argument, the call will occur within another NDT function. An `Od` template argument is therefore available, and utilised.

4.5.5 Program Units

Program units in 'F' include modules, and the main program. Modules correspond closely to C++ namespaces, and E♯ adopts this model in its C++ translation. To ensure the names of all translated module procedures are accessible throughout the namespace, all function declarations precede any definition. An 'F' module's public members are made available to a program unit or procedure by including the module name in a *use statement*; the C++ representation adopted by E♯ similarly applies the *using* directive, along with the relevant namespace identifier. C++ namespaces do lack the public and private access specifications of 'F' modules. Such access restrictions will have nevertheless already been enforced at the earlier object binding stage.

Unlike other program units, the singular 'F' main program contain no subprograms and hence is a good match for the C++ *main* entry function. The C++ main function could potentially also be an NDT function; however, main is not permitted to be a template. Hence the main function is used only to call a proxy function, `mini_main`, with the same arguments, but with the potential to be a template; and an NDT function.

4.6 Transforming Array Expressions

The transformation of an array expression into a series of nested C++ `for` loops is applicable only to array expressions constructed from the following: `elemental` function calls; array variables; array constants; array sections; or scalar expressions, promoted to arrays by context. Two problem cases remain. Firstly, non-elemental functions which return arrays may be present. A direct translation of these would result in a call to the function on *each* loop iteration. The C++ compiler would not know that the original function is pure, and would inefficiently, and unnecessarily, evaluate the function call on each cycle. Secondly, an array expression may contain array *subexpressions*, as function arguments; thus providing a recursive instance of the problem at large. Listing 4.31 provides a simple example containing both situations.

```
a = b + f(c + d)
```

Listing 4.31: An array assignment with a function call and subexpression.

Assuming the terms in Listing 4.31 produce conforming arrays, and `f` is a non-elemental function, E♯ must first evaluate the array subexpression `c + d`; then the call to function `f`; in advance of a direct translation into parallelisable C++ loops. Listing 4.32 provides such an alternative specification for the update of array variable `a`; using four assignment statements, and three temporary arrays.

```
1 t1 = c + d
2 t2 = f(t1)
3 t3 = b + t2
4 a  = t3
```

Listing 4.32: Array assignments with expressions representable by C++ loops.

The code in Listing 4.32 is more verbose than the original, however its structure has notable characteristics beyond readiness for translation to C++: as the assignment on line 2 is from an r-value, secondary allocation of memory for temporary array `t2` is not necessary. Furthermore, the translation now provides *two* array expressions, on lines 1 and 3; both of which may be parallelised.

Though redundant in Listing 4.32, the use of the temporary array `t3` is required in general for situations where dependency issues arise through overlapping array sections; as described in Section 3.4.3. In many situations E♯ can avoid this step.

The following sections first outline the Haskell technology which provides the building blocks for generic transformations and queries. There follows a presentation of the algorithm and combinators developed within E♯ to apply this to the task of transforming the 'F' AST into a form suitable for direct translation into C++ loops.

4.6.1 Scrap Your Boilerplate

Scrap Your Boilerplate (SYB) refers to the Haskell generic programming techniques described in three influential papers (Lämmel and Jones, 2003, 2004, 2005) by Ralf Lämmel and Simon Peyton Jones. The methods presented in the first of these papers facilitate the fundamental operations necessary to implement array expressions in E♯.

SYB requires two common extensions to the Haskell type system. The first of these is a type coercion, or *type cast* operator, which allows a runtime test of an expression's type; while producing no change in representation. Types specified as instances of the `Typeable` type class may participate in this type cast. Recent versions of GHC provide `Typeable` instances for all types included with the Haskell standard prelude. Furthermore, with another GHC extension, `DeriveDataTypeable`, the `deriving` clause can produce `Typeable` instances for user-defined types automatically[24]. An attempt to cast a value of type `String` to a value of type `Char`, or any non-`String` type, will "fail": `(cast "Ok":: Maybe Char)`, for example, results in `Nothing`. Casting a value of type `String` to `String` will "succeed", `(cast "Ok":: Maybe String)` evaluates to `(Just "Ok")`. By pattern matching on the `Maybe` constructor returned by an application of `cast` to an arbitrary unary transformation function, that function can be automatically selected for application *only* when its potential argument has the correct type; and use the identity function `id` otherwise. This is the behaviour of the SYB combinator, `mkT`, which, given a unary function argument, constructs a generic transformation function. For example, `(mkT not "Ok")` evaluates to `"Ok"`; while `(mkT not False)` equals `True`.

The second extension to Haskell required by SYB is for rank two types (Voigtländer, 2009, pages 8-11), enabled within GHC by either the `Rank2Types` or `RankNTypes` extension. The type variables in a Haskell type such as `((a -> Int) -> Int)` are implicitly quantified at the outer level as `(forall a. (a -> Int) -> Int)`. A rank two type can permit explicit placement of the quantifier within an inner scope, such as as `((forall a. a -> Int) -> Int)`. Such a signature declares that the function *argument* is polymorphic; rather than the function itself. For example, the function definition in Listing 4.33 will fail to type-check, as function argument `g` requires its polymorphic nature to be specified explicitly.

[24] In E♯, all ADTs used in the definition of the 'F' AST are made `Typeable` instances by this method.

```
f g = g "Ok" + g True
```

Listing 4.33: A Haskell function definition which requires rank two types.

Listing 4.34 adds support both for rank two types; and an explicit rank two type signature for `f`. The `main` function would thus produce a value of 4.

```
{-# LANGUAGE Rank2Types #-}

f :: (forall a. a -> Int) -> Int
f g = g "Ok" + g True

main = print $ f (const 2)
```

Listing 4.34: A Haskell function definition utilising rank two types.

Rank two polymorphism is used by SYB to perform traversals of arbitrary data structures. To achieve this, the first of two steps requires a non-recursive map-like operator, `gmapT`. `gmapT` applies its first argument, a function, to each component of the constructor definitions of its second; and hence the requirement for rank two types. `gmapT` is specified by a second type class: `Data`[25]. Again, instance definitions are provided for common types, and `deriving(Data)` will automatically create instances of `Data` for most user-defined types. The *second step* inserts the necessary ingredient of recursion: the SYB combinator, `everywhere`, is defined using `gmapT`, and applies a generic transformation, its first argument, to every node of its second; in a bottom-up fashion[26]. For example, `(everywhere (mkT not) x)`, will apply `(not :: Bool -> Bool)` to every node of `x` with type `Bool`; all other nodes will have `id` applied. The definition of `everywhere` from Lämmel and Jones (2003) is shown in Listing 4.35.

```
everywhere :: Data a => (forall b. Data b => b -> b) -> a -> a
everywhere f x = f (gmapT (everywhere f) x)
```

Listing 4.35: The SYB `everywhere` bottom-up transformation traversal.

SYB also supports generic *monadic transformations*. Monadic functions, of type `(Monad m => a -> m b)`, may also be made generic using another set of combinators: `mkM`, to construct a monadic transformation; `gmapM`, the monadic analogue of the map-like `gmapT`; and a monadic traversal combinator, `everywhereM`, together facilitate such activity.

[25] In the original SYB paper, the `Data` type class was named `Term`.

[26] A top-down traversal combinator, `everywhere'` is also provided.

Generic queries too are possible. `mkQ` constructs a generic query; the `gmapQ` function of the `Data` type class returns a list of results; while another recursive traversal combinator, `everything`, performs a left-associative fold.

4.6.2 Transforming the 'F' AST using SYB

Prior to a translation into C++ loops, statements encoded within E♯'s internal 'F' AST must first be transformed, to ensure non-elemental function calls, and array subexpressions, are calculated in advance; as described in Section 4.6. The precise method employed to execute the necessary *hoisting* operations is specified in Algorithm 3. This algorithm is written using an imperative style, in that variables are described as being updated. Of course E♯ is written in the pure, functional, programming language Haskell, and so each "update" is in actuality a fresh construction formed from both new and existing components. As will be demonstrated, the Haskell language together with the SYB primitives can permit an implementation which remains at the high level of abstraction used in Algorithm 3.

Of the four `for each` iterations undertaken within Algorithm 3, only the iteration across `ss2` from line 11 is non-generic, using a variation on a conventional `map`; a `zipWith` combined with a list of unique names.

Fresh names for the array temporaries introduced on lines 4 and 14 are generated based on a prefix common to all like substitutions: `__tmpF` or `__tmpA`; for function calls and array subexpressions respectively[27]. The remainder of each name is formed by catenating with the "stringified" integer corresponding to a term's position in the order of traversal. C++ allows variable declaration statements to be freely interspersed with executable statements. Consequently it is possible to declare temporary arrays local to the statement undergoing transformation. Additionally, C++ allows the names of declared variables to be used multiple times whenever the reused name occurs uniquely within a block scope. Hence, names generated in the implementation of Algorithm 3 need only be unique relative to the substituted term, within each transformed statement.

The requirement to provide a unique name for each generated array declaration node indicates that a traversal must also update *state*. The monadic transformations provided by SYB can facilitate such operations by utilisation of the *state monad* (O'Sullivan et al., 2008, 346-253). For example, the `incInts` function in Listing 4.36 uses the combinators of the state monad to both increment each `Int` value encountered, by the value stored in the current state;

[27]The use of double underscores in C++ names is reserved for the implementation (Becker, 2011, Section 17.6.4.3.2).

Algorithm 3 The hoisting transformation of the 'F' AST.

Require: An 'F' Abstract Syntax Tree, AST.
Require: Two statement iterators, $s1$ and $s2$.
Require: Two empty lists, $ss1$ and $ss2$, of 'F' AST statement nodes.
Require: A function call iterator, f, and a subexpression iterator, e.
Ensure: No non-elemental function calls and array subexpressions exist in AST.

1: **for each** $s1$ in AST **do**
2: **for each** f in $s1$ **do**
3: **if** $isElemental(f) = False$ **then**
4: Create $a1$, an AST declaration statement node for a temporary array.
5: Initialise $a1$ to f.
6: Append $a1$ to the statement list $ss1$.
7: Replace the reference to f in $s1$ with a reference to $a1$.
8: **end if**
9: **end for**{Bottom-up traversal}
10: Append the possibly modified $s1$ to $ss1$.
11: **for each** $s2$ in $ss1$ **do**
12: **for each** e in $s2$ **do**
13: **if** $rank(e) > 0$ **then**
14: Create $a2$, an AST declaration statement node for a temporary array.
15: Initialise $a2$ to e.
16: Append $a2$ to the statement list $ss2$.
17: Replace the reference to e in $s2$ with a reference to $a2$.
18: **end if**
19: **end for**{Bottom-up traversal}
20: Append the possibly modified $s2$ to $ss2$.
21: **end for**
22: Create c, an AST compound statement node.
23: Initialise c with the statement list $ss2$.
24: Replace the reference to $s1$ in AST with c.
25: **end for**

and increment the current state by one. Thus, assuming the following numeric literals are each of type `Int`, `(incSt 1 ([5],6))` evaluates to `(([6],8),3)`; a tuple formed from the modified input, along with the final state.

The traversal of all statements within the 'F' AST, specified on line 1 of Algorithm 3, follows an almost identical pattern to that of the `incInts` function in Listing 4.36. In E♯'s traversal, however, the type signature of the monadic transformation, `grabStmt`, is `(Stmt -> State Int Stmt)`. Hence the state remains of type `Int`, while the targeted type is now `Stmt`.

The remainder of Algorithm 3 can be considered in two phases: firstly, the hoisting of function calls; covered in lines 2-9, and secondly, the hoisting of array subexpressions in line 12-19. Each will now be examined in turn.

```
import Control.Monad.State(get,put,runState,State)
import Data.Generics.Schemes(everywhereM)
import Data.Generics.Aliases(mkM)

grabInt :: Int -> State Int Int
grabInt i = do g <- get
               put (g+1)
               return (i+g)

incInts st x = runState (everywhereM (mkM grabInt) x) st
```

Listing 4.36: Using `mkM` and `everywhereM` for monadic transformation.

Hoisting Function Calls

The traversal of function calls bounded by the `for each` on line 2 of Algorithm 3 must be a bottom-up traversal to ensure that deeply nested function calls are hoisted first; and a single pass may suffice. A monadic traversal is again required here, to ensure the substitution of each function call, within a statement, receives a unique name. Some further subtlety is also now required: traversal should halt whenever a nested statement is encountered; for example, within a compound statement such as a `do` statement. Each such nested statement will be caught by the traversal mechanism already, and should not be processed more than once. A monadic version of the `everywhereBut` combinator from Lämmel and Jones (2003) was created: `everywhereButM`, shown in Listing 4.37, provides a suitable traversal.

```
everywhereButM :: Monad m => GenericQ Bool -> GenericM m -> GenericM m
everywhereButM q f x
 | q x       = return x
 | otherwise = do x' <- gmapM (everywhereButM q f) x
                  f x'
```

Listing 4.37: A bottom-up monadic traversal combinator with a stop condition.

The `runState` invocation to execute the function call traversal is as shown in Listing 4.38. The first argument to `everywhereButM` is a generic query, which operates as a generic predicate in this instance: `(const True)` will, given matching types, always return `True` to indicate that traversal should cease. Note also that `gmapM` is used initially, to ensure that the input statement, `s`, always of type `Stmt`, is not itself a trigger to stop traversal. The type of the monadic transformation, `hoistPrim`, is given in Listing 4.39. `hoistPrim` will initialise a temporary array declaration node with its argument, a function call node, and substitute a name reference to it at the original call site; assuming the function is not `elemental`.

```
runState (gmapM (everywhereButM (False `mkQ` (const True::Stmt -> Bool))
                                (mkM hoistPrim)) s) st
```

Listing 4.38: Specifying the function call hoisting traversal.

```
hoistPrim :: Prim -> State (String,Int,[Stmt]) Prim
```

Listing 4.39: The type of the monadic tranformation, `hoistPrim`.

Hoisting Array Subexpressions

The hoisting of array subexpressions specified between lines 12-19 of Algorithm 3 again requires a bottom-up monadic traversal. However, a more elaborate traversal strategy is now required, based on E♯'s `everywhereButM` combinator.

```
a = f(b + c + d)
```

Listing 4.40: An array assignment with multiple subexpressions.

The need for a modified traversal is due to the recursive definition of expressions in terms of subexpressions; in E♯ both have the same type: `Expr`; or fully, `Fast.Expr`. A bottom-up SYB traversal targeting expressions may therefore be triggered multiple times unnecessarily. For example, in Listing 4.40, `b`, `c`, and `d` are conforming arrays, and `f` is a non-elemental function. A traversal of the array expressions within the assignment statement will encounter *three* subexpressions; rather than `(b + c + d)` alone.

Listing 4.41 demonstrates a translation of Listing 4.40 where all subexpressions are hoisted. The most significant failing of this conversion is that the opportunities for parallelism have been reduced. Parallelising the statements on lines 2 and 3 will entail both the unnecessary DMA transfer of temporary array `t2`, both from and to SPU scratch memory; and the launch of two thread teams, rather than one. A translation optimised for parallelism is shown in Listing 4.42.

The code in Listing 4.43 is similar to Listing 4.36, except that the transformation, `grabStr`, now applies to the common, recursively defined data-type: `String`. The transformation will both append the "stringified" integer comprising the state of the monad, to the string argument; and increment the state. Consequently, `(renameStrs 1 (["ab"],"c"))` will evaluate to `((["ab123"], "c45"),6)`; matching against the individual characters of each string, and also each empty list.

```
t1 = d
t2 = c + t1
t3 = b + t2
a = f(t3)
```

Listing 4.41: A translation of Listing 4.40 for C++ loops, but restricted parallelism.

```
t1 = b + c + d
a = f(t3)
```

Listing 4.42: An ideal translation of Listing 4.40 for parallel C++ loops.

The `everywhereBarM` traversal combinator, shown in Listing 4.44 was created to circumvent the issue with transformations applied to recursively defined data-types. A predicate, `q`, is introduced to extend the definition of SYB's `everywhereM`, applying the transformation, `f`, *only* when the type of a child node, `x`, is different from that of its parent. This is distinct from SYB's `everywhereBut`, and E♯'s `everywhereButM`, in two ways: the predicate cannot halt traversal; and the predicate is updated throughout the traversal, using the partial application of the local function, `typeEq`. The `typeOf` function is the sole member of the `Typeable` type class.

In Listing 4.45, the `everywhereBarM` combinator is used to define a variation of Listing 4.43's `renameStrs` function. The `renameBarStrs` combinator uses the previous `grabStr` definition, and also the new `everywhereBarM` traversal combinator[28]. Providing the new function, with the earlier arguments, `(renameBarStrs 1 (["ab"] ,"c"))`, evaluates to `(("ab1" ,"c2"),3)`[29]. As anticipated, only the largest `String` value is transformed.

The `everywhereBarM` combinator is still lacking. As was the case when hoisting function calls using `everywhereButM`, the traversal over expressions should again *halt* upon encountering a nested expression. Therefore a fusion of E♯'s `everywhereButM` and `everywhereBarM` combinators is performed, and provided in Listing 4.46 as `everywhereButBarM`. The first of the two predicate arguments, `q1`, corresponds to the "but", halting component; while the second, `q2`, corresponds to the "bar", dynamic parent-child type inequality component.

The `runState` invocation to execute the traversal of expressions is as shown in Listing 4.47. As with the traversal of function calls, the first argument is a generic predicate;

[28] An *initial* predicate function must be provided as `everywhereBarM`'s second argument, and corresponds to a specification of whether the generic transformation should be applied to the outermost value; the original input value. In this example, the transformation applies to values of type `String`, while the input is of tuple type; therefore the choice between `(const True)` or `(const False)` is irrelevant here; the identity function, `id`, would be applied anyway.

[29] Once again, the second tuple member, `3`, provides the final state of the monadic transformation.

```
grabStr :: String -> State Int String
grabStr s = do g <- get
               put (g+1)
               return $ (s ++ show g)

renameStrs st x = runState (everywhereM (mkM grabStr) x) st
```

Listing 4.43: Monadic transformation on all strings and substrings.

```
everywhereBarM :: Monad m => GenericQ Bool -> GenericM m -> GenericM m
everywhereBarM q f x
 | q x       =          gmapM (everywhereBarM (typeEq x) f) x
 | otherwise = do x' <- gmapM (everywhereBarM (typeEq x) f) x
                  f x'
 where typeEq p c = typeOf p == typeOf c
```

Listing 4.44: Monadic transformation predicated upon parent and child type inequality.

`(const True)` will, for nested expressions, return `True` to indicate that traversal should cease. The *value* of the argument provided as the *initial* second predicate, `const True`, is in fact irrelevant in this situation, as the transformation, `hoistArrExpr`, is not applicable to the argument, which is of type `Stmt`; and hence only the identity function, `id`, would be applied. Upon each recursive application of `everythingButBarM`, the second predicate is updated to judge the equality of the parent and child types. Note again that `gmapM` is used initially, to ensure that the input statement, `s`, always of type `Stmt`, is not itself a trigger to stop traversal.

The type of the monadic transformation, `hoistArrExpr`, is given in Listing 4.48. `hoistArrExpr` will initialise a temporary array declaration node with its argument, an expression node, and substitute a name reference to it at the location of the original expression; assuming the expression is not scalar.

A Transformation Example

Having specified the algorithm and traversal combinators we can return to the example which introduced the section. Figure 4.6 presents a schematic representation of the function and array expression hoisting transformations, applied to the sample assignment statement from Listing 4.31; now shown in the left-most panel of Figure 4.6. From the left, we first see the creation of a second statement resulting from a single function call hoist. The subsequent first and second array expression hoists originate from these two statements. The rightmost panel is suitable for direct translation into C++ loops.

```
renameStrsBar st x = runState (everywhereBarM (const True)
                                              (mkM grabStr) x) st
```

Listing 4.45: Monadic transformation only on the largest strings.

```
everywhereButBarM :: Monad m => GenericQ Bool -> GenericQ Bool
                             -> GenericM m    -> GenericM m
everywhereButBarM q1 q2 f x
 | q1 x       =          return x
 | q2 x       =          gmapM (everywhereButBarM q1 (typeEq x) f) x
 | otherwise  = do x' <- gmapM (everywhereButBarM q1 (typeEq x) f) x
                   f x'
 where typeEq p c = typeOf p == typeOf c
```

Listing 4.46: Monadic transformation with two predicates.

4.7 Optimisation for Constant Sizes

Memory allocation for 'F' arrays and character strings may be performed *automatically* if the sizes can be determined at the time of compilation. E♯ is then configured to take advantage of this using automatically-sized versions of the array and character classes: `ArrayTN` and `FCharN`[30]. Both these classes provide an additional integer template parameter, `N`, which provides the size expression to the 'C' array data member of each.

A function, `eval`, was added to the symbol table module to evaluate whether a given parse tree expression may be calculated at compile-time; the type signature of `eval` is `(Expr -> Maybe Integer)`. The `eval` function is implemented as a recursive bottom-up traversal of the expression tree, with appropriate inspection to the lookup table for named variables. Support was also added for a subset of 'F' intrinsic functions, including the common array inquiry functions, `size`, `lbound`, and `ubound`. The application of such functions to array arguments with constant-extent shapes, can thus also be evaluated at compile-time. Another function, `getStaticShape`, is defined mutually recursive with `eval`. The type signature of `getStaticShape` is `([ParseTree.AttrSpec] -> [(Integer, Integer)])`; where the first argument originates from the third component in the `TypeDeclStmt` value constructor, and may contain a *dimension* attribute specifying extents at compile-time. The integer 2-tuples of the second argument define upper and lower bounds.

The `getStaticShape` function adds optional compile-time shape information to each `PT.Prim` value. This information is propagated using a custom value constructor added to the `Fast`.

[30]The canonical, dynamically-allocated classes, `ArrayT` and `FChar`, are described in Sections 5.2 and 5.1.1.

```
runState (gmapM (everywhereButBarM
                  (False `mkQ` (const True::Stmt -> Bool))
                  (const True)
                  (mkM hoistArrExpr)) s) st
```

Listing 4.47: Specifying the array expression hoisting traversal.

```
hoistArrExpr :: Expr -> State (String,Int,[Stmt]) Expr
```

Listing 4.48: The type of the monadic transformation, `hoistArrExpr`.

`Subscript` ADT: `(ConstSS (Integer, Integer))`. The definition of the `OB` type class's `ob` function seen in the earlier Listing 4.18 is modified to use `getStaticShape`.

The `getStaticShape` function is ultimately also applied within the application of `ob`, to the `PT.TypeDeclStmt` constructor pattern. A positive result allows for a multiplicative fold on the result; inducing an `ArrayTN` type and its `N` template argument. For example, an array with attribute `dimension(2,3,4)` would produce a value of 24 for `N`.

Evaluation of the compile-time nature of a character string's length is performed comparably. Again, a modification to `ob` is required, this time when matching against the `PT.CharType` value constructor. A new AST node is also created: the `Fast.CharNType` constructor of the `Fast.Type` and signifies the discovery of a constant length.

The `CPP` Haskell language extension allows the 'C' preprocessor to be applied to Haskell source code. E♯ enables the above optimisations as two of a number of compilation options within `defines.h`; using `#if` and `#define` directives to control the active optimisations.

Input statement

```
a = b + f(c + d)
```

Function hoist

```
t1 = f(c + d)
a  = b + t1
```

1st expression hoist

```
t2 = c + d
t1 = f(t2)
a  = b + t1
```

2nd expression hoist

```
t2 = c + d
t1 = f(t1)
t3 = b + t2
a  = t3
```

Figure 4.6: Schematic pseudocode representation of a hoisting transformation sequence.

Chapter 5

Runtime Support and Static Pointer Locality

In this chapter, we focus our attention upon the back-end of the E♯ compiler. The target, or output, language is the Offload C++ language, an extension of C++ introduced in Section 3.2. The call-graph duplication of Offload C++, along with the static, often implicit, attribution of integer-encoded address locality to raw pointers is the patented technology of Codeplay Software Ltd. Beyond the individual problems and solutions described below, this chapter contributes a significant case-study regarding the highly distinctive Offload C++ language, and its compiler. Particular attention has been paid to the use of integer template arguments to encode address spaces. This is otherwise unreported in the literature relating of Offload C++, excepting a brief introduction in the documentation supplied with the Offload SDK (Dolinsky, 2010, pages 8).

The ABI of the Fortran array format is non-standard, and we first present two new array class templates, `ArrayT` and `ArrayTN`, designed both to tightly integrate with the runtime libraries of the majority of Fortran compilers; and also provide compatibility with the enhanced pointer types of the Offload C++ language. Such pointer types are subsequently demonstrated to present interface challenges due to the combination of call-graph duplication and eager template instantiation. A library-based solution involving the use of both a novel pointer container template class, `oi_ptr`, and a pair of C++ template memory allocator methods, `New` and `NewA` are introduced. Template metaprogramming is also utilised to encode the Fortran standard rules of type promotion, and so facilitate the static resolution of overloaded functions in the selection of runtime library calls.

5.1 Scalar ‘F’ Types in E♯

There are two primary requirements when translating the intrinsic scalar ‘F’ types into their equivalents in C/C++. Firstly, the C/C++ types should have a native memory and performance profile; appropriate for an HPC context. Secondly, the types should be compatible with the selected Fortran compiler. Section 4.5.1 has already described the approach taken by the E♯ compiler. The code in Listing 5.1, however, taken from the E♯ runtime library, provides a convenient reference for our current discussion Here, only the floating-point types are shown. The three `typedef` declarations provide the strings used by the compiler to represent the two `real` kinds supported by E♯.

```
template <int K> struct FrealT;
template <> struct FrealT<4>    {
  typedef float type;
  static const int kind = 4;
};
template <> struct FrealT<8>    {
  typedef double type;
  static const int kind = 8;
};
typedef FrealT<4>::type Freal4;
typedef FrealT<8>::type Freal8;
typedef Freal4          Freal;
```

Listing 5.1: The ‘F’ real types in C++.

5.1.1 Representing ‘F’ Character Strings

A C++ class, `FChar`, is defined to represent an ‘F’ character string. Via the relevant member methods, `FChar` provides common operations applied to `character`-type variables, such as assignment, serialisation, and relational operations. The `FChar` class supports the general requirement for dynamic memory allocation. A statically allocated variant, `FCharN`, is also provided to facilitate potential compiler optimisations; described in Section 4.7.

Forward declarations of the two string classes are shown in Listing 5.2. Each template parameter is provided with a default argument[1], allowing a typical string of characters to be declared using the concise `FChar<>` type.

For procedures having `character` arguments, E♯ only generates function parameters for the more common, dynamically allocated variant, `FChar`. Implicit conversion operators are

[1]The OFFLOAD_DEPTH intrinsic was introduced in Section 3.2.3.

```
template <typename T=char,           int Od=OFFLOAD_DEPTH>
  class FChar;
template <typename T=char, int N=1, int Od=OFFLOAD_DEPTH>
  class FCharN;
```

Listing 5.2: Forward declarations for the C++ character string classes.

provided by the `FCharN` class to implement the necessary translation of `FCharN` arguments; described fully in Section 4.7. A further implicit conversion operator allows access to the 'C' pointer to the underlying data.

5.2 The E♯ Array Runtime Library

Since the 1990 revision of the language standard, Fortran has provided first class support for arrays. Unfortunately the precise arrangement of a Fortran array in memory, its embedded ABI, is not standardised. Consequently, each Fortran compiler vendor has developed a unique representation.

Each compliant Fortran implementation is accompanied by a suite of intrinsic functions applicable to scientific computing. For the E♯ compiler, utilisation of these libraries offers a number of benefits:

- E♯'s runtime performance matches the fastest implementations;
- Results and performance from different vendor implementations may be compared;
- Edge cases associated with legacy specifications are handled;
- E♯'s array class is strengthened through exposure to more tests;
- Library implementation time is minimised.

Scalar types in Fortran and 'C' are essentially interchangeable; assuming types are paired by equal bit width. Further support is provided by the intrinsic ISO_C_BINDING module. Consequently, Fortran functions using scalar values are accessible to a C/C++ programmer equipped with the function prototypes. Consequently it is the Fortran intrinsic functions which pass array arguments that present an obstacle.

The Chasm project (Rasmussen et al., 2006) from the Advanced Computing Laboratory of the Los Alamos National Laboratory was developed to help overcome this infelicity. Chasm

provides a low-level ‘C’ library, offering cross-vendor access to the distinct Fortan array representations deployed within the runtime libraries of the thirteen supported Fortran compilers.

5.2.1 Array Class Templates

To facilitate the full set of ‘F’ intrinsic functions, E♯ provides a new C++ template class to interface with the Chasm API. The `ArrayT` and `ArrayTN` classes are provided in a portable, and extensively inlined, header-only implementation. Functions and member methods of these array classes provide support for high-level ‘F’ operations including array sections, assignment, and serialisation; while element indexing provides both the syntax and index order of Fortran.

Type safety is emphasised through template parameters which include not only the element type, but also the rank. This corresponds exactly with arrays in ‘F’, which also fix an array’s rank at compile-time. This allows for C++ overload resolution to select a version of a library function specialised for the type and rank of the inputs. By including the rank parameter, and using template metaprogramming it is also possible to calculate the type parameters of the output, as demonstrated in Section 5.3. Library bindings for a selection of the GNU Fortran library routines are provided.

Support for implicit, strongly-typed, transfer of data between the memory spaces of heterogeneous architectures is integral to the design of the `ArrayT` and `ArrayTN` array classes. The most visible aspect of this is the addition of a further, integer, template parameter governing locality for the array classes.

5.2.2 Array Declarations

The code in Listing 5.3 demonstrates the declaration, initialisation, and destruction of the array descriptor, or handle, `desc`, using the low-level ‘C’ Chasm API. The GNU compiler is selected on line 10; the array rank is chosen on line 19; and the element type[2], `double`, is selected by the `F90_Double` enumerated value, also given on line 19. All such options are chosen at runtime; the `desc` handle, a `void` pointer, provides no static information such as the array rank, or element type. More advanced compile-time information, such as the runtime library and associated dope vector format; or the address space of the underlying array data, are thus also absent.

[2]The array data is defined and initialised on line 8.

```
1  F90_CompilerCharacteristics cc;
2  void *desc;
3  int i;
4  const int rank = 2;
5  const long                                          lowerBound[] = {1,1};
6  const unsigned long element_size = sizeof(double), extent[]     = {5,2};
7  long                strideMult[rank];
8  double data[] = {1,2,3,4,5,6,7,8,9,0};
9
10 F90_SetCompilerCharacteristics(&cc, "GNU");
11
12 desc = malloc(cc.getArrayDescSize(rank));
13 assert(desc);
14
15 strideMult[0] = element_size;
16 for (i = 1; i < rank; i++)
17   strideMult[i] = extent[i-1] * strideMult[i-1];
18
19 cc.setArrayDesc(desc, data, rank, F90_ArrayPointer, F90_Double,
20                 element_size, lowerBound, extent, strideMult);
21
22 free(desc);
```

Listing 5.3: Creating an array using the Chasm 'C' API.

In contrast, Listing 5.4 provides equivalent functionality to that of Listing 5.3, though using E♯'s `ArrayT` class. While clearly concise, a more valuable distinction is witnessed by the selection of compiler, element type, and rank; all of which are static compile-time constants: the template arguments `F90_GNU_C`, `double` and `2`. These static values are thus capable of assisting the C++ compiler to maintain type safety.

```
double data[] = {1,2,3,4,5,6,7,8,9,0};
{
  ArrayT<F90_GNU_C,double,2> desc(data,1,5,1,2);
}
```

Listing 5.4: Creating an array using a dynamically sized E♯ array object.

The `desc` value of both listings in fact represents a Chasm *dope vector*: in this case `dope_vec_GNU`. `dope_vec_GNU` is one of thirteen dope vectors provided by the Chasm API. Each is represented by a 'C'-style `struct`, and corresponds to the internal array representation used by each supported compiler. To facilitate the dual address space of Offload C++, discussed in Section 3.2.2, each is modified to allow the locality of its pointer members to be controlled via a template argument. Listing 5.5 provides a concrete example for the GNU Fortran dope vector, `dope_vec_GNU`, with the `inout` macro[3] on line 4 configuring the address localities.

[3]The `inout` macro was defined in Listing 3.20.

```
template<int Od=OFFLOAD_DEPTH>
struct dope_vec_GNU {

  inout(Od)
  void *base_addr, *base; /* base address of the array, and base offset */

  size_t  dtype;         /* elem_size, type (3 bits) and rank (3 bits) */

  struct {
    size_t stride_mult;  /* distance between successive elements */
    size_t lower_bound;  /* first array index for a given dimension */
    size_t upper_bound;  /* last array index for a given dimension */
  } dim[7];
};
```

Listing 5.5: The GNU Fortran dope vector extended for pointer locality.

The primary template of the `ArrayT` class is shown in Listing 5.6. This is then partially specialised upon the `F90_CompilerID` non-type template parameter for each of the thirteen compiler identifiers; the version specialised for `F90_GNU_C`, for example, inherits from `dope_vec_GNU`. Almost all of the data members of `ArrayT` are so inherited. When a Fortran runtime function receives an `ArrayT` object, it is the dope vector base class which provides the essential, portable, data structure. The three remaining type parameters, `T`, `R`, and `Od` correspond to the array element type, rank, and address space of the underlying data.

```
template<F90_CompilerID C, typename T, int R, int Od=OFFLOAD_DEPTH>
  class ArrayT;
```

Listing 5.6: The dynamically sized `ArrayT` primary class template.

A statically allocated array class is also provided, as a partner to `ArrayT`. The `ArrayTN` class includes an additional integer template parameter corresponding to the array size[4]. The runtime performance penalty of dynamic memory allocation and bounds checking can thus be elided. The forward declaration for `ArrayTN` is shown in Listing 5.7.

```
template<F90_CompilerID C, typename T, int R, int N,
         int Od=OFFLOAD_DEPTH>
  class ArrayTN;
```

Listing 5.7: The statically sized `ArrayTN` primary class template.

As an optimisation, the `ArrayTN` class is used by E♯ to represent any ‘F’ array with a size which can be determined at compile-time. As with dynamic character parameters, functions

[4]The C++11 library class template `std::array` (Becker, 2011, pages 746-749) makes similar use of its second `size_t` template parameter, `N`.

with array parameters are generated by E♯ only for *dynamically* allocated arrays. An `ArrayT` constructor is thus provided which accepts an `ArrayTN`-typed object, to implicitly facilitate the minimal conversion step.

5.2.3 Implementing Essential 'F' Array Operations

E♯ defines a number of C++ member methods to encapsulate intrinsic 'F' language operations applicable to arrays. Each is provided for both dynamically and statically sized array classes, occasionally with minor differences; for example, bounds checking and memory allocation are unrequired by the latter. For brevity, the discussion focuses only on the `ArrayT` class[5].

Array Indexing

The lower bound index for an 'F' array is, by default, 1. This is configurable, and each array may specify its own bounds upon declaration. So it is with the overloaded `ArrayT` class constructor, which uses the rank integer template argument, `R`, and default parameter values, to set the extents for each dimension[6], as shown by the declaration in Listing 5.8.

```
ArrayT( long l1,     long u1,
        long l2 = 1, long u2 = 1, long l3 = 1, long u3 = 1,
        long l4 = 1, long u4 = 1, long l5 = 1, long u5 = 1,
        long l6 = 1, long u6 = 1, long l7 = 1, long u7 = 1 );
```

Listing 5.8: An `ArrayT` constructor declaration to configure array extents.

The array indexing operation must then also participate, by respecting these configured extents. Elements from an 'F' array will often be arranged in memory in the same order as an equivalent C++ array. The order of *indices*, however, is reversed; with the fastest moving index now in the leftmost position. The `ArrayT` class defines `operator::()` to respect all such conventions.

[5] Likewise, `const` versions of most `ArrayT` and `ArrayTN` methods are also implemented, though omitted from further discussion.

[6] An 'F' array can have up to seven dimensions.

Array Sectioning

A contiguous, or strided, section of an 'F' array may be selected using an extended form of the familiar scalar element indexing, using colon delimited ranges and strides; see Section 3.3.2 for details. The `section` method of `ArrayT` returns a newly allocated `ArrayT` pointer, referencing regularly partitioned elements of the input array. The `section` method declaration accepts up to 21 arguments, and makes use of default arguments, comparable to the `ArrayT` constructor of Listing 5.8. An 'F' section may also result in an array reference with rank less than the original. Only a scalar valued index will reduce the result dimensionality; a range such as `(1:1:1)` merely specifies a range extent of 1. Rank reduction in 'F' is denoted statically by the absence of colons in a set of index tuples; such as the section, `A(1:3,4)`. Without this syntax in C++, the `section` method is defined as a method template with a single integer template argument, `R2`, corresponding to the rank of the result; `section<1>(1,3,1,4,4,1)`, for example, describes a 1 dimensional array section[7].

Array Element Traversal

Traversal of an array's elements is required both by the intrinsic 'F' array operations of serialisation and deserialisation; and array assignment. Intrinsic *reduction* functions, such as `sum`, `maxval`, and `minval` also require traversal. Complications in defining a straightforward, generic C++ traversal of an `ArrayT` object's elements arise principally from the possibility of noncontiguous data. Nevertheless, it is feasible to construct all of the 'F' operations requiring traversal, by the application of a familiar recursive pattern, involving a base case and an inductive clause. Listing 5.9 shows the preamble to this recursive traversal: the definition of the `ArrayT` copy assignment operator; and a helper method, `dataCopy`.

```
inline ArrayT &operator=( const ArrayT &rhs )          {
  this->dataCopy(rhs);
  return *this;
}

template<typename T2, int Od2>
inline void dataCopy( const ArrayT<C,T2,R,Od2> &arr ) {
  char *pd = reinterpret_cast<char *>(this->baseAddress());
  char *p  = reinterpret_cast<char *>(arr.baseAddress());
  LoopC<R>::copy(&pd,&p,*this,arr);
}
```

Listing 5.9: The `ArrayT` copy assignment operator and helper method.

[7]For each dimension, three index arguments are mandated by the `section` method.

Listing 5.10 defines a specialisation of a class, `LoopC`[8], which represents the base case for the recursive traversal. `LoopC`'s integer template variable is specialised to 1, and corresponds to the rank of the source and destination arrays provided as input. The static `copy` method performs the entire operation, with line 10 performing the assignment of a single array element. The Offload C++ compiler will here identify the locality of each array's data, and perform DMA transfers between memory spaces if required; otherwise a conventional copy is performed (Dolinsky, 2010). The data of either vector may be noncontiguous, and are accounted for separately by the pointer incrementation on lines 11 and 12.

```
1  template <> struct LoopC <1> {
2    template <F90_CompilerID C,typename T,typename T2,int R,int Od,int Od2>
3    static inline void copy(char **ppD, char **ppS,
4                            const ArrayT<C,T, R,Od>  &dst,
5                            const ArrayT<C,T2,R,Od2> &src){
6      long eS  = src.extent(1);
7      long smS = src.stride(1);
8      long smD = dst.stride(1);
9      for (long k = 0; k < eS; k++) {
10       *reinterpret_cast<T *>(*ppD) = *reinterpret_cast<T2 *>(*ppS);
11       *ppS += smS;
12       *ppD += smD;
13     }
14   }
15 };
```

Listing 5.10: The base case for array assignment traversal.

Where the source and destination array ranks are greater than 1, the recursive definition of the `copy` method in Listing 5.11 is invoked. When $n > 1$, a rank n array can be considered as a nested collection of rank $(n - 1)$ arrays. Array assignment is thus formed from the aggregate of all assignments performed upon corresponding subarrays of each input.

The inductive case of Listing 5.11 differs from the base case of Listing 5.10 in two notable ways. Firstly, there is the recursive call, on line 12, to the `copy` method of the `LoopC` class. This appears in place of element assignment. Secondly, two temporary pointer variables, `prevS` and `prevD`, are declared and initialised on lines 10 and 11. The pair are subsequently employed on lines 13 and 14, to restore the current array traversal data pointers, `ppS` and `ppD`, to their states prior to each recursive call.

As Fortran arrays have a maximum rank of 7, the depth of recursion is likewise constrained. The `ArrayT` class, and traversal classes such as `LoopC`, are defined entirely within the same header file, and it is anticipated that a C++ compiler will implement each full traversal by

[8]Either a primary template for `LoopC`, such as `template <int> struct LoopC;`; or Listing 5.11 should lexically precede Listing 5.10.

```
template <int i> struct LoopC {
  template <F90_CompilerID C,typename T,typename T2,int R,int Od,int Od2>
  static inline void copy(char **ppD, char **ppS,
                          const ArrayT<C,T, R,Od>  &dst,
                          const ArrayT<C,T2,R,Od2> &src){
    long eS  = src.extent(i);
    long smS = src.stride(i);
    long smD = dst.stride(i);
    for (long k = 0; k < eS; k++) {
      char *prevS = *ppS;
      char *prevD = *ppD;
      LoopC<i-1>::copy(ppD,ppS,dst,src);
      *ppS = prevS + smS;
      *ppD = prevD + smD;
    }
  }
};
```

Listing 5.11: The recursive step for array assignment traversal.

textual substitution at each call site; that is, by *inlining*.

Serialisation and deserialisation operations, on `ArrayT` values, are accessed through overloads of the C++ insertion and extraction operators, `(<<)` and `(>>)`, with assistance from the `friend` keyword. Intrinsic reduction procedures, such as `sum` and `maxval`, are each provided as a function template. All such operations are implemented using a similar traversal strategy to that of the assignment operation demonstrated above.

Lastly, a second, template version of the assignment operator must also be defined, to accommodate source and target arrays of differing *locality*; and this is shown in Listing 5.12[9]. The `dataCopy` method of Listing 5.9 is deployed once again. Internally, the application of the overloaded assignment operator to array *elements*, performs a DMA transfer between distinct memory spaces.

```
template<typename T2, int Od2>
inline ArrayT<C,T,R,Od> &operator=(const ArrayT<C,T2,R,Od2> &rhs)
{
  this->dataCopy(rhs);
  return *this;
}
```

Listing 5.12: An `ArrayT` copy assignment operator handling differing address spaces.

[9]This template version of the assignment operator is insufficient alone; a default, non-template, copy assignment operator would still be generated by the C++ compiler; and selected by C++ overload resolution for array assignment between homogeneous memory locations. The same issue was observed in Listing 4.30 of Section 4.5.3.

5.3 Interfacing with the ‘F’ Runtime Libraries

Having defined the high level, and portable, Fortran array template classes, `ArrayT` and `ArrayTN`, it becomes possible to utilise template metaprogramming techniques to perform operations equivalent to table lookups, type checking, and overload resolution; otherwise implemented within the ‘F’ compiler itself. A secondary effect is that an intuitive, user-level, C++ interface exploiting function overloading can be obtained, radically simplifying the foreign function interface for ‘F’ runtime library calls.

At an implementation level, multiple versions of the ‘F’ intrinsic functions may be provided for the different possible argument type combinations. For example, version 4.1.1 of the GNU Fortran runtime library has 10 different versions of the matrix multiplication function `matmul` on PPU, and 12 on SPU; while version 4.6.1 on x86 has 14. Version 0.92 of the G95 compiler runtime library on x86 includes 348 different versions of `matmul`. Using C++ overloading it is possible to implicitly select the correct version at compile-time, as would be the case for a native Fortran compiler. The following discussion is motivated by matrix multiplication as a running example; and demonstrates concretely the degree to which overload resolution and type checking may be delegated from E♯, to the C++ host compilers, with typeful wrappers provided by a C++ library.

5.3.1 Calculating the Result Type and Kind

Metcalf and Reid (1996, Page 32) provides a table which reports the type of each result whenever one of the ‘F’ arithmetic operators, `(+)`, `(-)`, `(*)`, or `(/)` are applied; given the types of the operands. A method is also described there to obtain the *kind* of the result, given too the kinds of the operands[10]. A function will now be defined that too is capable of reporting this information at compile-time to the C++ compiler.

A *subset* of C++ types are used to implement the built-in types of ‘F’; as described in Section 5.1. The first component we require is a type function which will provide the ‘F’ type corresponding to a given C++ type from this subset. Abstractly, this can be represented as a set of partial functions; with the implementation defined as a series of template object specialisations. Listing 5.13 includes just one of these, the specialisation of `FType` starting on line 4 is selected for an input type of `char`. For example, `FType<char>::type` evaluates to `FintegerT<1>`.

Given the C++ types of the operands, the remaining partial specialisations of `FType`, here

[10]The `matmul` intrinsic function also observes these rules.

```
1 template <typename T>
2   struct FType;
3
4 template <>
5 struct FType<char> {
6   typedef FintegerT<1> type;
7 };
```

Listing 5.13: A type level function to convert a `char` to an `FintegerT<1>`.

omitted, enable the calculation of the result kind, using the template object in Listing 5.14 as a compile-time function.

```
1 template <typename A, typename B>
2 struct FResType {
3   typedef typename FType<A>::type FTA;
4   typedef typename FType<B>::type FTB;
5   static const int Ak = FTA::kind, Bk = FTB::kind;
6   static const int Rk = Ak > Bk ? Ak : Bk;
7 };
```

Listing 5.14: A template object to calculate the result kind, `Rk`, of an arithmetic operation.

The `A` and `B` template parameters of the `FResType` class from Listing 5.14 range over the C++ types which have an 'F' representation. They are consequently valid arguments to the `FType` template object used on lines 3-4. Given `float` and `double` types as input, the kind of the result may be obtained using the expression `FResType<float,double>::Rk`.

To obtain the result *type* information, a helper class is used: `FResTypeHelper`. This class is specialised by all valid combinations of `FType` representations and their kinds; though for brevity Listing 5.15 shows only the pairing of `FintegerT` and `FrealT`. Using `FResType` we can obtain both the *type* of the operands; and the *kind* of the result. We are thus able to provide all the type and constant integer template arguments to the specialised `FResTypeHelper` template class, and inspect its `type` member.

```
template <typename A, int Ak, typename B, int Bk, int Rk>
  struct FResTypeHelper;

template <int Ak, int Bk, int Rk>
struct FResTypeHelper<FintegerT<Ak>,Ak,FrealT<Bk>,Bk,Rk> {
  typedef FrealT<Rk> type;
};
```

Listing 5.15: A helper class providing the full result type.

As a demonstration of `FResTypeHelper`, the expression in Listing 5.16 calculates the type of the result of an arithmetic operation involving two real types, `float` and `double`. The type obtained is `FrealT<8>`, its static members providing type and kind information as shown in Listing 5.1.

```
FResTypeHelper<
  FType<float>, FType<float >::type::kind,
  FType<double>,FType<double>::type::kind,
  FResType<float,double>::Rk
>::type
```

Listing 5.16: A verbose type expression providing the result type of an arithmetic expression.

The existing members of the `FResType` class provide all five of the arguments to `FResTypeHelper`, and so `FResType` is extended by a sixth static member, `type`, a `typedef`. Listing 5.17 shows the *extended* `FResType` definition. The type obtained from the expression in Listing 5.16 is then available through the more concise `FResType<float, double>::type`. The C++ type and kind are also then available through the constant `type` and `kind` members.

```
template <typename A, typename B>
struct FResType {
  typedef typename FType<A>::type FTA;
  typedef typename FType<B>::type FTB;
  static const int Ak = FTA::kind, Bk = FTB::kind;
  static const int Rk = Ak > Bk ? Ak : Bk;
  typedef typename FResTypeHelper<FTA, Ak, FTB, Bk, Rk>::type::type type;
};
```

Listing 5.17: A template object to calculate the result type of an arithmetic operation.

5.3.2 Application to Matrix Multiplication

We now consider how the template machinery of the previous section can be applied, within the E♯ runtime library, to the challenge of static type checking and overload resolution in the 'F' intrinsic matrix multiplication function: `matmul`. The GNU Fortran runtime library is selected here due to its compatibility with the PS3 architecture. Only the latter steps below are specific to the version of Fortran runtime library used.

Both arrays provided to a `matmul` function template can have different respective element types and ranks; the locality, and runtime library identifier, however, will be the same. Hence the `matmul` template function requires four template parameters. A function declaration for

each version of `matmul` provided by the runtime library and exposed to the linker, must also be accessible[11].

The rank of the matrix multiplication result may be obtained from a simple compile-time function of the ranks of the two input arrays. Only ranks of 1 or 2 are supported, and while both arrays may have different ranks; when equal, both must be of rank 2. This can be encoded in a straightforward fashion by specialising a template class `MMRank`, as shown in Listing 5.18.

```
template <int,int> struct MMRank;
template <> struct MMRank<1,2> { enum { value = 1 }; };
template <> struct MMRank<2,1> { enum { value = 1 }; };
template <> struct MMRank<2,2> { enum { value = 2 }; };
```

Listing 5.18: Calculating the rank of a matrix multiplication.

A final subcomponent requires a mechanism to call the correct version of the `matmul` function in the runtime library. The result type has a one-to-one correspondence with the GNU Fortran library function, and we choose to specialise a new template struct `MM`, having a single method, on the `FType` representation. For example, if the `matmul` result type is computed to be a C++ `float`, which corresponds to a Fortran `real` with a kind of 4, the library routine to call is `_gfortran_matmul_r4`. This is obtained using the specialisation of the `MM` class shown in Listing 5.19. The thirteen other GNU `matmul` specialisations are also included with E♯.

```
template <typename>
  struct MM;

template <>
struct MM<FrealT<4> > {
  static inline void *_(void *a, const void *b, const void *c,
                        int d, int e, void *f) {
    return _gfortran_matmul_r4(a,b,c,d,e,f);
  }
};
```

Listing 5.19: Specialisation based on the result type of `matmul`.

A test call to the GNU `matmul` function may then be invoked by the expression shown in Listing 5.20.

The final definition of E♯'s `matmul` is shown in Listing 5.21. Note the contrast in the typeful

[11] This is created by hand. In general there is little specific support for calling routines in a Fortran runtime library from C/C++.

```
MM<FrealT<4>>::_(NULL,NULL,NULL,NULL,NULL,NULL);
```

Listing 5.20: A test invocation of `MM<FrealT<4>>::_`.

definition of `matmul` against the typeless implementation layer provided by the `MM` class. The use of `typedef`s and enumeration types within the `matmul` function are solely to aid readability, and do not add to the memory footprint or runtime performance of the function. The parameters given `NULL` values on line 13 provide the option for the GNU Fortran `matmul` implementation to use BLAS libraries, if available; and gives an indication of the low-level, and internal nature of this interface.

```
1  template <typename T1, int R1, typename T2, int R2>
2  inline
3  ArrayT<F90_GNU_C,typename FResType<T1,T2>::type,MMRank<R1,R2>::value> &
4  matmul( const ArrayT<F90_GNU_C,T1,R1> &a,
5          const ArrayT<F90_GNU_C,T2,R2> &b )
6  {
7    enum { result_rank = MMRank<R1,R2>::value };
8    typedef typename FResType<T1,T2>::type element_type;
9    typedef ArrayT<F90_GNU_C,element_type,result_rank> result_type;
10
11   result_type &out = *new result_type();
12   MM<typename FType<element_type>::type>::_(&out,&a,&b,
13                                             NULL,NULL,NULL);
14   out.m_bMemoryAllocated = true;
15   return out;
16 }
```

Listing 5.21: The final E♯ `matmul` function template.

5.4 Manual Replication in Offload C++

Pointer members of C++ classes, which may be inner or outer depending upon the instantiation of a template parameter, can lead to unexpected type-mismatches, and hence produce a compilation error. This is due to the template instantiation mechanism of the Offload C++ compiler, which may "eagerly" require a definition to exist which is compatible with both inner *and* outer pointer types. The current workaround for this situation requires the user to provide additional overloaded definitions of all functions involved, Such overloaded definitions routinely contain almost exact replications of the original function bodies, and consequently presents a maintenance problem as a project develops. This problem was encountered in the array and string library classes used in code generated from the E♯ compiler.

5.4.1 Problem Description

Often a class will manage its own access to memory. With Offload C++ we will require that the pointer referencing this memory is declared in a way which allows its locality, whether inner or outer, to be defined by an integer template argument provided upon instantiation. Consider the simple character string class shown in Listing 5.22.

```
template <int Od>
struct Str
{
  void alloc() { m_p = new char[8]; }
  char &operator[](int i) { return m_p[i]; }
  char inout(Od) *m_p;
};
```

Listing 5.22: A template class which manages memory resources.

In an *outer* context the declaration of an `Str` object, `Str<0>`, and invocation of the `alloc` method, demonstrated in Listing 5.23, compiles without error using the Offload compiler.

```
Str<0> s0;
s0.alloc();
```

Listing 5.23: Legal object declaration and method invocation in an outer context.

An `Str<0>` object may also be declared within an inner context. However, the same code snippet in an inner context will give a compilation error, as the `new` operator used in the `alloc` method defaults to provide local memory. The compiler complaint is due to the illegal attempt to assign an inner pointer to an outer. A first attempt to circumvent this issue could alter the `alloc` method body to instead use the overloaded, outer, version of `new`, as shown in Listing 5.24.

```
void alloc() { m_p = new outer char[4]; }
```

Listing 5.24: Allocation using the outer new operator.

We will routinely also expect to manipulate inner pointers within an inner context, through the `Str<1>` type. As may be anticipated, the modified `alloc` method applied to an `Str<1>` object, within an inner context, as shown in Listing 5.25, also results in a compilation error; this time due to an attempted assignment of an outer pointer to an inner.

Specialising the template parameter of the `Str` class can provide the partial solution of Listing 5.26. To avoid the manual duplication of the overloaded array subscript operator `[]`

```
Str<1> s1;
s1.alloc();
```

Listing 5.25: Illegal object declaration and method invocation in an inner context.

definition, the definition of `Str` may be refactored using a base class, as shown in listing 5.27. We also observe that the definition of `Str` we seek should be applicable in the same three valid settings as that of its pointer member; shown in Table 5.1.

```
template <int> struct Str;
template <>
struct Str<0>
{
  void alloc() { m_p = new outer char[8]; }
  char &operator[](int i) { return m_p[i]; }
  char inout(0) *m_p;
};

template <>
struct Str<1>
{
  void alloc() { m_p = new char[8]; }
  char &operator[](int i) { return m_p[i]; }
  char inout(1) *m_p;
};
```

Listing 5.26: Specialising the string class declaration by offload depth.

	Inner Pointer	Outer Pointer
Outer Context	✘	✔
Inner Context	✔	✔

Table 5.1: Valid instantiations of pointers bearing locality

Nevertheless, the above solution will require care in some use cases. For example, consider the template function, `init`, of Listing 5.28, which accepts a `Str` object reference, and calls its `alloc` method.

We require this function to be callable from both inner and outer contexts. In particular, a call to `init` from within an inner context, with an argument of type `Str<1>`, should be valid. Doing so, however, will result in a compilation error. The reason for this is that the compiler has *now* identified that it is also *possible* for `init` to be called from within an outer context, with an `Str<1>` argument. Such a call would require the illegal assignment of an outer pointer to an inner. This is due to the default action of the `new` operator in an outer context,

```
template <int Od>
struct BaseStr
{
  char &operator[](int i) { return m_p[i]; }
  char inout(Od) *m_p;
};

template <int> struct Str;
template <>
struct Str<0> : public BaseStr<0> {
  void alloc() { m_p = new outer char[8]; }
};

template <>
struct Str<1> : public BaseStr<1> {
  void alloc() { m_p = new char[8]; }
};
```

Listing 5.27: Specialising the string class incorporating a base class.

```
template <int Od>
void init(Str<Od> &s) {
  s.alloc(); s[0] = '\0';
}
```

Listing 5.28: A candidate for offloading with a template object.

as called from within the `alloc` method of `Str<1>`, which is to return a conventional outer pointer.

The `init` function can though be made to compile by overloading both conventionally on the argument type, and also through the `offload` function qualifier, as presented in Listing 5.29.

```
void init(Str<0> &s) {
  s.alloc(); s[0] = '\0';
}

offload void init(Str<1> &s) {
  s.alloc(); s[0] = '\0';
}
```

Listing 5.29: Specifying the omission of the `init` host overload for `Str<1>`.

This approach specifies `init` by omission of the unwanted definition on `Str<1>` arguments in an outer context. The init definition for `Str<0>` meanwhile is made available in both inner and outer contexts using familiar call-graph duplication. With this approach, all valid calls

to `init` will succeed. Unfortunately this has required us to duplicate the `init` definition by hand; and furthermore, such need of error-prone replication is infectious. Consider, in Listing 5.30, another function, `foo`, which calls the `init` function of Listing 5.29.

```
template <int Od>
void foo(Str<Od> &s) {
  init(s);
}
```

Listing 5.30: Calling an offload-specialised template function.

Of the three valid call configurations, a call with an `Str<1>` argument in an inner context again causes a compilation error. On this occasion, the error identifies that `foo` *could* be called with an `Str<1>` argument from within an outer context, and that a corresponding definition of `init` is missing. This absent overload of `init` is exactly the one we have previously been compelled to omit. Therefore `foo` must also be overloaded in a similar fashion, as shown in Listing 5.31.

```
void foo(Str<0> &s) {
  init(s);
}

offload void foo(Str<1> &s) {
  init(s);
}
```

Listing 5.31: Specifying the omission of the `foo` host overload for `Str<1>`.

The requirement for manual replication of function bodies will persist for each template function which both contains a call to `Str<1>::alloc` within its call graph; and which also carries the `Str` class template argument.

More generally, the problem trigger is a pointer operation involving locality that is governed by template instantiation. Declaring an object in an *inner* context, will produce a response from the Offload compiler as if the object had also been declared in an outer context. If declaring that object in an outer context *would* produce an error, an error will appear. In contrast, an object declared in an *outer* context will not will produce a response as if the object had also been declared in an inner context. This asymmetric behaviour is at the root of the manual replication problem.

5.4.2 New with Locality

The source of the manual replication problem with respect to the issues identified with the `Str` class, centre on the `new` operator, and specifically the extended version of `new` provided by the Offload C++ compiler. There follows an explanation of an alternative definition for the `new` operator which will overcome the problem.

The specific issue with respect to the extended Offload `new` operator is that no method is available to allocate memory to a pointer which may be instantiated *either* with inner or outer locality. This may be observed not only in `inout` template-based pointer declarations; such as the `m_p` pointer member at the centre of the prior `Str` examples. The call-graph duplication feature of Offload C++ means that functions applied to pointer arguments in all three valid states of Table 5.1, will require manual replication. Consider the test example of Listing 5.32.

```
void f(int *p) { p = new int; };
```

Listing 5.32: A function making an allocation.

Here, the call to `new` will fail when `f` is given an outer pointer argument in an inner context. A similar function instead defined using the *outer* `new` operator, will fail when an *inner* pointer argument is provided in an outer context. We must again overload the function definition, as shown in Listing 5.33.

```
        void f(int *p) { p = new outer int; };
offload void f(int *p) { p = new       int; };
```

Listing 5.33: Overloading a function making an allocation.

The `new` and `delete` keyword operators may be overloaded; both globally and as class members. In the present situation, however, such an operation is ineffectual: the function signature for `new` provided to the user for overloading, declares only a C-style unary function, accepting an `unsigned int` parameter, and returning a `void` pointer. Overloading based solely on the return type is not permitted in C++.

The lesser known *placement new* provides a signature including a pointer argument, and so *can* be overloaded for different pointer localities. Any solution involving the routine use of placement `new` would though be cumbersome, as the user must pre-allocate the necessary memory. Nevertheless, consider the function template, `g`, shown in Listing 5.34.

```
template <typename T, int Od>
void g() {
  void inout(Od) *place = malloc(10);
  T    inout(Od) *p;
  p = new (place) T;
  free(place);
}
```

Listing 5.34: Limitations of placement new.

The `g` function template outlines a strategy wherein `p` is assigned a pointer with a depth equal to `Od`. Calling the `g` function of Listing 5.34, with the three valid locality configurations, will not compile, for two reasons. First of all, the `g` function would again require manual replication due to the eager instantiation of a placement `new` call from an outer context, with an *inner pointer* as the place argument. Such an illegal call actually produces an outer pointer, incompatible with the inner pointer it is here assigned to. Secondly, the `malloc` function will only allocate memory at a locality equal to that of its calling context; this should not be a surprise, as there is no opportunity in C++ to overload functions based on the return type.

It may also be thought that E♯'s `inout` qualifier could be of assistance. In a declarative setting, the `outer` and `inout(0)` qualifiers are equivalent, and may be utilised interchangeably; both are defined as `__declspec(__setoffloadlevel__(0))`. The outer `new` operator, however, requires precisely the literal string "`outer`". Any use of the `inout` qualifier in this context is quietly ignored by the Offload compiler.

The framework for a locality-aware version of the `new` operator is shown in Listing 5.35. A class template, `New`, is defined with two template parameters: the type to be given to the relevant `new` operator; and an integer representing the depth at which the memory should be obtained. The `New` class has one static method[12], the concisely named _. Using partial specialisation, the member methods of `New`, specialised with a zero (`0`) for their second template argument, make explicit use of the `outer` overloaded version of `new`. The specialisation of `New` with a one (`1`) for the second template argument, uses the common version of `new`, which returns memory local to the calling context. This, second version of the _ function is also overloaded using the `offload` qualifier. There is therefore no possibility of an attempt to allocate inner memory from an outer context at runtime.

The second version of each _ method, overloaded with a single parameter, corresponds to the traditional overload of `new` which initialises the freshly created value.

[12] C++ operators cannot be static members of a class.

```
template <typename,int Od=OFFLOAD_DEPTH>
  struct New;

template <typename T>
struct New<T,0> {
          static inline T *_()     { return new outer T; }
          static inline T *_(T x) { return new outer T(x); }
};

template <typename T>
struct New<T,1> {
          static inline T *_()     { assert(0); return 0; }
          static inline T *_(T)    { assert(0); return 0; }
  offload static inline T *_()     { return new       T; }
  offload static inline T *_(T x) { return new       T(x); }
};
```

Listing 5.35: A locality aware version of the `new` operator.

The traditional C++ *array* version of `new` is syntactically distinguished by its use of the `[]` token string. The `NewA` class template in Listing 5.36 is similarly distinguished, statically, from `New` by its name. This is required to ensure there is no ambiguity surrounding a numeric argument, which could signify either an initial value, or a quantity of array elements.

```
template <typename,int Od=OFFLOAD_DEPTH> struct NewA;

template <typename T>
struct NewA<T,0> {
          static inline T *_(int nelems) { return new outer T[nelems]; }
};

template <typename T>
struct NewA<T,1> {
          static inline T *_(int)         { assert(0); return 0; }
  offload static inline T *_(int nelems) { return new       T[nelems]; }
};
```

Listing 5.36: A locality aware version of the `new[]` operator.

The definitions of `New<T,1>::_(T)`, `New<T,1>::_` and `NewA<T,1>::_` exist precisely to placate the eager Offload C++ type-checker. The `assert(0)` within each one is primarily for documentation; the methods should never actually be called. The use of static *class* method members for `New` and `NewA` is required as partial function template specialisation is permitted neither in C++, nor the final draft of C++11(Becker, 2011). The implementation follows the conventions of the *nothrow version* of the C++ `new` operators; which throw no exception on failure. An illegal request for inner memory made from within an outer context also adopts this protocol. This is appropriate as the Offload C++ compiler does not support exceptions

within inner contexts.

The `static` member functions of `New` and `NewA` may now be used as drop-in replacements for the C++ `new` operator. Due to the use of the special Offload C++ macro, `OFFLOAD_DEPTH`, as a default template argument, the second template argument can often be omitted. In an inner context, where outer memory is required, a zero (`0`) locality value must still be provided explicitly. The *syntax* is distinct from the traditional `new` operator, however, application of the `New` class's _ method requires only the addition of a depth argument. Listing 5.37 demonstrates straightforward usage in the three valid configurations of Table 5.1.

```
int main(int argc, char *argv[])
{
  double *po1 = New<double>::_();

  offload {
    double *pi  = New<double>::_();
    double *po2 = New<double,0>::_();
    delete pi, po2;
  };

  delete po1;
  return 0;
}
```

Listing 5.37: Demonstrating the new allocators in the three valid contexts.

In the following section we will demonstrate that this new "new" is only one component in a general solution to the problem of manual replication. Consequently, the definition of the `New` class will shortly receive a minor addition.

5.4.3 Inner Pointers and Outer Duplication

The code in Listing 5.38 makes use of the earlier `Str` class defined in Listing 5.27. Surprisingly, though line 4 appears to illegally assign a value to an inner memory address from an outer context, it both compiles and runs.

The overloaded `[]` operator of `Str` returns a C++ reference. The reason the above code first of all compiles, is due to the Offload C++ compiler replacing the inner locality attribute for outer on pointers or reference types returned by functions called in an outer context. Then, having bypassed the type-checker, the code is free to run. The actual value of the address returned by `operator[]` happens to point to an address on the stack: an outer location.

The program in Listing 5.39 also compiles, and expresses this detail more directly. In the

```
1 int main(int argc, char *argv[])
2 {
3   Str<1> s1;
4   s1[0] = '\0';
5   return 0;
6 }
```

Listing 5.38: Apparently illegal assignment to an inner locality.

outer context of `main`, the character pointer `pc` is an outer pointer, and the `inner` function qualifier of the `getInnerPtr` function return type is therefore essentially ignored.

```
char inner *getInnerPtr() { return 0; }

int main(int argc, char *argv[])
{
  char *pc;
  pc = getInnerPtr();
  return 0;
}
```

Listing 5.39: Inner pointers ignored in outer contexts.

Other locality-sensitive pointer operations will return more familiar compilation errors. In Listing 5.40, we observe a reduced version of the earlier `init` function from Listing 5.28, where instead of the overloaded `operator[]`, we use raw pointer dereferencing to assign with the null terminator.

```
template <int Od>
void init(Str<Od> &s) {
  *s.m_p = '\0';
}
```

Listing 5.40: Eager template instantiation when in an inner context.

With this code, we again face a familiar issue: a legal call to the `init` function in an inner context, with an argument of type `Str<1>`, results in a static error relating to the compiler's concern that were it instead called in an *outer* context, an illegal assignment to an inner address would occur.

While C++ encapsulation can prohibit direct access to data members such as `m_p`, this is an ad-hoc solution. Furthermore, encapsulation cannot remove every problem, such as the example in Listing 5.41 involving a `friend` method for equality.

Again, a legal inner-context call to the equality operator in Listing 5.41, with an `Str<1>`

```
inline bool operator ==(const Str &lhs, const char c) {
  return lhs.m_p[0] == c && lhs.m_p[1] == '\0';
}
```

Listing 5.41: A `friend` equality operator producing an inner error.

argument, produces a compilation error, and compels us towards the familiar overload and replication seen in Listing 5.42.

```
       inline bool operator ==(const Str<0> &lhs, const char c) {
  return lhs.m_p[0] == c && lhs.m_p[1] == '\0';
}
offload inline bool operator ==(const Str<1> &lhs, const char c) {
  return lhs.m_p[0] == c && lhs.m_p[1] == '\0';
}
```

Listing 5.42: Overloading an equality operator for inner contexts.

5.4.4 A Pointer Locality Class

A legal pointer operation occurring solely in an inner context may cause an obstructive compilation error relating to its *potential* illegality in an outer context. A specific example involving the dereferencing of the pointer data member of an `Str` object was presented in Listing 5.41; Listings 5.25 and 5.28 are similar. The proposed solution is to abstract pointers and their localities by a class which can routinely wrap Offload C++ outer and inner pointer variables. This class, `oi_ptr`, is reminiscent of a smart pointer, and overloads all potentially illegal operations with minimal, perfunctory "stub" methods; much like those of `New<T,1>` and `NewA<T,1>`. Again, these should never actually be called. The `oi_ptr` class definition begins in Listing 5.43.

The `oi_ptr` class is, like `New` and `NewA`, declared as a template class with two parameters. The first, a type parameter, corresponds to the base type of the internal pointer; while the second is an integer parameter, and corresponds to the depth, or locality, of the memory space that said pointer targets. The static `element_type` and `depth` members provide a form of reflection, or *trait*, and are occasionally useful for meta-programming.

The `oi_ptr` class, like the new locality allocators, is defined by two partial specialisations of the second template argument. The first, shown as `oi_ptr<0>` in Listing 5.43, corresponds to an abstraction of a pointer with outer locality. This may be seen from the `inout(0)`

```
template <typename T,int Od=OFFLOAD_DEPTH> struct oi_ptr;
template <typename T>
struct oi_ptr<T,0> {

  typedef T element_type;
  static const int depth = 0;

  inline oi_ptr &operator= (T *p)         { m_p = p; return *this;  }
  inline      T &operator[](const int i) {          return m_p[i]; }
  inline      T &operator* ()            {          return *m_p;   }

private:
  T inout(0) *m_p;
};
```

Listing 5.43: A pointer class abstraction specialised for outer memory.

qualifier[13] in the only non-static data member: the pointer `m_p`. In contrast to those that follow, the definitions of the three operators of `oi_ptr<T,0>` are entirely straightforward.

The abstraction of an *inner* pointer is required to contain operator definitions, both for the valid actions it may perform within an *inner* context; plus minimal "stub" operator overloads *representing* those operations which are invalid in an *outer* context, when acting upon raw inner pointers.

```
template <typename T>
struct oi_ptr<T,1> {

  typedef T element_type;
  static const int depth = 1;

  inline oi_ptr &operator= (T * )         { assert(0); return *this;  }
  inline      T &operator[](const int i) { assert(0); return m_p[i]; }
  inline      T &operator* ()            { assert(0); return *m_p;   }

  offload
  inline oi_ptr &operator= (T *p)         { m_p = p;   return *this;  }
  offload
  inline      T &operator[](const int i) {            return m_p[i]; }
  offload
  inline      T &operator* ()            {            return *m_p;   }

private:
  T inout(1) *m_p;
};
```

Listing 5.44: A pointer class abstraction specialised for inner memory.

[13]The `inout(0)` qualifier here is actually redundant, though it does have a pleasing symmetry with its partner in the definition of `oi_ptr<T,1>` in Listing 5.44 .

The specialisation of `oi_ptr<T,1>` overloads the same operator set as `oi_ptr<0>`, *twice*: the first set of three are selected for execution when in an outer context; the second set, each prefixed by the `offload` keyword, are selected for execution when within an inner context. Although the definitions of both sets are similar, this is due to the simplicity of these operators. Each of the first set requires only to both match the signature of its `offload`-overloaded partner; and satisfy the compiler's type-checker. Note that although the `*` and `[]` operators of `oi_ptr<T,1>` are specified to, and do, return an outer reference in an outer context, the `*m_p` and `m_p[i]` arguments, provided to their respective return statements, *denote* values stored within the incompatible *inner* locality. The code nevertheless compiles, and as we will see, serves its purpose.

Again, the `assert` macros in the non-`offload` operator definitions of `oi_ptr<T,1>` are used primarily as a form of documentation; emphasising that the purpose of these operators is not found in their execution. For brevity, only the non-`const` versions of the `oi_ptr` operators necessary to complete the running `Str` examples are included in Listings 5.43 and 5.44.

```
int main(int argc, char *argv[])
{
  char inner *pi;
  *pi = '\0';
  return 0;
}
```

Listing 5.45: Illegal inner address assignment from outer context.

To illustrate the operation of `oi_ptr`, we once again consider the issues regarding pointer dereferencing and assignment in invalid contexts observed in the `Str` class; Listing 5.45 presents a brazen example. The issue in this code centres on the attempt to assign an outer value to an inner address, in an outer context; to which the compiler rightly objects. The code in Listing 5.46 instead uses an `oi_ptr` to express a very similar schematic. In this case, however, no compilation error occurs. Should the code be executed, the assignment would do nothing other than fail a runtime assertion. This is analogous to the use of `operator::[]` with the `Str<1>` class in Listing 5.38.

```
int main(int argc, char *argv[])
{
  oi_ptr<char,1> pi;
  *pi = '\0';
  return 0;
}
```

Listing 5.46: Illegal inner address assignment from outer context without compilation error.

With the pointer locality class, `oi_ptr`, and the novel locality-aware versions of the C++ `new` operators present in the `New` and `NewA` classes, we may finally define a concise representation of the `Str` class, shown in Listing 5.47, and free of replication.

```
template <int Od>
struct Str
{
  void alloc() { m_p = NewA<char,Od>::_(8); }
  char &operator[](int i) { return m_p[i]; }
  oi_ptr<char,Od> m_p;
};
```

Listing 5.47: Final version of the string class using the located pointer class.

The `Str` class definition in Listing 5.47 replaces the earlier use of a raw `char` pointer member, `m_p`, with a template object of type `oi_ptr`. The `m_p` member is parametrised by both the target type of the original base pointer; and a constant integer to represent locality; facilitated by the enclosing class template's `Od` argument. Note the *essential* use of the 3 non-offloaded methods from the specialised `oi_ptr<1>` definition from Listing 5.44. These definitions are never actually executed, but provides an appeasement to the Offload compiler. The `Od` parameter is also now used to specify the locality of the memory allocated by the `NewA` class's `static _` method.

In the earlier `init` template example, the Offload compiler's template instantiation mechanism required that a definition of the `Str<1>` object's `alloc` method also be available in an outer context, and requires a call to the overloaded non-`offload _` method definition of `NewA`. This is now exercised by the test example shown in Listing 5.48[14].

```
int main(int argc, char *argv[])
{
  oi_ptr<char,1> pi;
  pi = NewA<char,1>::_(7);
  delete pi;
  return 0;
}
```

Listing 5.48: Ostensible assignment and allocation of inner memory in an outer context.

The call to `NewA<char,1>::_` returns a `0`, or `NULL`, outer pointer. According to its definition, the overloaded assignment operator then makes no modification to the `pi` object, simply returning a reference to it. Like the non-`offload` operator overloads in the partially specialised `oi_ptr<T,1>`, though executable, the comparable definition of `NewA<T,1>::_` performs no

[14]The raw `m_p` pointer member of `pi` is provided to the `delete` operator by application of an implicit conversion operator. The full suite of `oi_ptr` operator definitions are omitted for brevity.

conventionally useful actions, and is designed with the sole intention of placating the type-checker. These definitions are though essential in allowing conventional definitions of function templates, such as `init`; and class templates, such as `Str`, parametrised by the memory locality of their internal state. With the new allocators and pointer containers this may be executed conventionally; without error nor code repetition.

5.4.5 Closing Thoughts on Manual Replication

Disciplined use of the `oi_ptr` class can eliminate a class of compilation errors described above. For maximum effectiveness, *all* pointers should be constructed within, or wrapped by, the `oi_ptr` class. The `oi_ptr` class has not however been designed to thwart the determined programmer who would seek to extract a raw pointer from an `oi_ptr` object; this may easily be performed using, say, `&*p`, or other techniques involving the otherwise convenient implicit conversion operator.

It may be possible to define a fully encapsulating `oi_ptr`, however it is instead recommended that the `oi_ptr` class be taken as a temporary solution, used only for as long as it serves its current purpose, which is to ameliorate the eager template instantiation issue which otherwise leads to manual code replication. A more *permanent* solution could appear as a modified version of the Offload compiler, which, provided with an *inner* context object declaration, or function call, which *would* be illegal if declared or used in an *outer* context, emits *no error*. To be clear, should such an object be illegally declared in an outer context, the contentious operations will reside within its methods, and relate to illegal operations involving inner pointers. Beyond avoiding the need for manual replication, this would be a symmetric design, reflecting the existing Offload compiler action, wherein an *outer* context object declaration, which *would* be illegal if declared in an *inner* context, emits *no error*.

The minimal example of Listing 5.49 can illustrate the symmetry which is, in this instance, absent in the design of the Offload C++ language; an *extension* of the C++. A consequence of this asymmetric design is the need for code replication outlined in this section.

A call to `error<1>()` in an *outer* context will produce a valid error; a call to `error<0>()` in an *inner* context will also produce a valid error. Moving on from deliberate errors, a call to `error<0>()` in an *outer* context will produce no errors, as expected. However, a call to `error<1>()` in an *inner* context, does produce an error. This is unexpected, asymmetric, and due to the implementation mechanism of the Offload compiler's call-graph duplication, which makes an unwelcome "internal" instantiation of `error<1>()` in an outer context.

On the other hand, the `oi_ptr` interface may still remain useful as a *model* of the Offload C++

```
template <int Od>
void error() {
  int inout(OFFLOAD_DEPTH) *p1;
  int inout(Od)            *p2;
  p1 = p2;
}
```

Listing 5.49: A function template which can produce locality-related errors.

language within standard C++, allowing for more rapid experimentation with the precise semantics of call graph duplication 3.2.4. Secondly, it should also be possible for a newly designed library, perhaps using the OpenCL API (Khronos OpenCL Working Group, 2011), to obtain *some* of the benefits of Offload C++, such as implicit DMA transfers[15], using the assignment operator, and hence remaining standard C++.

Furthermore, while the requirement for the `oi_ptr` pointer locality class *may* diminish, memory allocation functions or operators such as `New` and `NewA`, using integer template parameters to specify address space locality seems like a perfect fit with recommended practise (Dolinsky, 2010, pages 8) regarding Offload C++ pointers with parametrically defined locality. Beyond Offload C++, the use of static address spaces within programming languages is likely to become more prevalent, as evidenced by the four memory spaces of OpenCL C Khronos OpenCL Working Group (2011). The `New` and `NewA` class templates fulfill the requirements of a locality-based allocator in a form sufficiently general to serve such future developments; simply, and using standard C++. Further enhancement to these two memory allocation classes may also be permitted using C++11: for example, the _ method of `New` and `NewA` classes cannot currently *initialise* values having constructors with multiple arguments; variadic templates in C++11 (Becker, 2011, pages 337-338) can re-enable this feature.

The full definition of the `oi_ptr` class is shown in Appendix A, and contains additional members including implicit conversion, arrow, `const` overloaded operators; and constructors. The `New` and `NewA` classes were shown earlier in Listings 5.35 and 5.36.

5.5 Test Driven Development

Development of the E♯ compiler was invaluably assisted by the creation and application of both a simple, functional regression testing framework; and a suite of ‘F’ test programs. Given a file base name, `FILE`, the bash script, `ctest.sh`, will compile both `FILE.f95` and

[15]More generally: implicit inter-address space data access.

`FILE.cpp` using the native Fortran and C++ compilers respectively. Both of the resulting executable programs are then run, and their outputs compared using the *numeric diff* program, Numdiff (Ivano Primi, 2010). Even among Fortran compilers, output formatting differs, most noticeably in the amount of whitespace used; and also in the display of floating-point values. The Numdiff program may be configured for such aspects, and so treats multiple whitespace characters as one; and a minimum absolute error of 1.0×10^{-3} is specified. Should a difference exist, it is displayed to standard output.

Test file names used with E♯ each have a prefix of `test`, and a numeric suffix; currently occupying the range $1 - 43$; for example, `test43.f95`. The `ctest.sh` script, described above, is then applied within a second bash script, `multitest.sh`. The `multitest.sh` script accepts parameters including a file base name; and upper and lower bounds for the numeric name suffix. The E♯ compiler is then invoked on each file formed by the concatenation of file base name; each numeric suffix; and the `.f95` extension; thereby generating C++ files of the same name; modulo extension. Following this, `ctest.sh`, is applied to each pair, and the returned bash error code from it tallied. `multitest.sh` also accepts an integer to specify how many of the tests *should* pass, allowing a clear message of success or failure.

Whenever a new language feature is added to E♯, or a bug fixed, a small 'F' program is created to test firstly for compilation success; and then result accuracy. Upon completion of this stage, the `multitest.sh` script is invoked, via `make test`, to ensure any recent changes have not caused a regression. If no such problems are discovered, a new test is added to the suite; either as a file or a function.

Native, parallel execution of benchmark programs upon the PS3 would also allow *performance regression* to be monitored within the test framework. So too, the existing functional regression testing would be significantly strengthened by such an addition.

Chapter 6

Experimental Results

Empirical performance measurements from the E♯ compiler can provide valuable insight to the parallel execution model; the feasibility of the chosen array abstraction; and highlight opportunities for further optimisation. Four medium-size[1] benchmark programs are used to assist in the evaluation of the E♯ compiler. The first performs an estimation of the members of the mathematical model known as the Mandelbrot set, and allows us to look at DMA transfer bottlenecks and load balancing, while exploring differing approaches to parallel decomposition. The following two benchmark applications, BlackScholes and Swaptions, are financial simulations from Princeton University's PARSEC benchmark suite (Bienia et al., 2008), converted by hand to 'F' from original 'C' and C++ respectively. These provide examples of emerging workloads. Finally, an implementation of the $O(n^2)$, *all-pairs* physical dynamics simulation known as the n-body problem, is examined. This last program was originally developed as part of a series of workshops organised by the Scottish Informatics and Computer Science Alliance (SICSA), known as the SICSA Multicore challenge.

Note that, while all of the benchmark programs are controlled by an array expression, analogous to a parallel for loop, E♯'s execution model is task parallel. This allows the MIMD architecture of the CBE to contribute to the load balancing. The only truly data parallel algorithm is the n-Body benchmark; where each thread encounters the same control path.

All of the performance measurements presented in the following sections are averaged across six runs on a PlayStation 3 running Fedora Core 7 - Moonshine. Single-precision was used throughout, due to the significantly reduced hardware support for double-precision calculations on the SPU. In addition to the 4.1.1 versions of the GNU C, C++, and Fortran compilers provided with the IBM Cell SDK v3.0, version 4.6.0 of GCC was also installed. E♯ uses version 2.0.2 of the Offload C++ compiler, with the runtime patched for E♯ to utilise version

[1] The size of the benchmarks, in terms of lines of code, is discussed in Section 6.1.6.

4.6.0 of GCC for the SPU; and version 4.1.1 of GCC and G++ for the PPU. Unless explicitly mentioned, all compilers use the `-O3` switch throughout. Whenever quoted, the run time duration of ad hoc binaries is obtained using the unix `time` command. E♯ uses 128 software threads throughout, as further explained in the microbenchmark described in Section 6.1.5 below. Speedups are relative to the fastest serial version available, and similar to the *absolute speedup* metric. However, rather than run each serial reference version on a single SPU, a single PPU is used. This is first of all honest; reflecting the common serial execution paradigm of the PS3 platform. Secondarily, this is a practical approach, as most problem sizes will not fit within the 256 KB of SPU local store; and also require the assistance of the PPU to perform system calls.

6.1 Preliminary Setup and Experiments

This section first reports on the selected method for timing measurements. E♯'s SPU memory footprint macro is then introduced, followed by its role in obtaining a *significant* reduction in the SPU memory consumed by both the GNU Fortran runtime library; and the C++ `new` operator. The time for Offload C++ to launch and join a thread is then established experimentally; followed by a measure of the benchmark code sizes, via line counts; and the compilation times of each.

6.1.1 Benchmark Timing Routines

A single 'F' timing subroutine is provided by E♯: `date_and_time`. This subroutine is implemented by an inline C++ call to the corresponding GNU Fortran runtime library[2] function: `date_and_time`. The `date_and_time` subroutine is convenient in that results are readily presentable. Internally, `date_and_time` calls the common 'C' function, `gettimeofday`; provided through `unistd.h`, if available. This allows the serial 'C' programs to be timed using the same function as the 'F' programs.

6.1.2 SPU Memory Footprint Macro

A minimally invasive 'C' preprocessor macro was created to display the amount of SPU local store memory used at any point in a program's execution. The macro, `SPU_LS_PROFILE`, uses the SPE's `printf` facility to output the relevant figures to the screen. `SPU_LS_PROFILE`

[2]The GNU Fortran runtime library is written in 'C'.

accepts two arguments corresponding to an integer thread identifier, and a character string; to identify individual threads, and discriminate between named serial stages respectively. The operation of the macro is simply to repeatedly allocate chunks of 1024 bytes using `malloc` until no more are available. The amount of free local store memory available is then displayed, and the chunks are freed. The code is provided in Appendix B.

6.1.3 GNU Fortran Runtime SPU Memory Footprint

The memory footprint associated with the GNU Fortran runtime library targeting the SPU can be extremely high. A minimal 'C' test program was created which comprises *only* a single call to either the single-precision, floating-point, Fortran intrinsic function, `random_number`; or the generic version of `transpose`.

For our first test, we are using version 4.1.1 of GCC and the accompanying GNU Fortran runtime library, `libgfortran.a`; as provided with version 3.0 of the IBM Cell SDK. Calling neither intrinsic function, the program comprises only the measurement code introduced by the `SPU_LS_PROFILE` macro. In this form, a mere 13 KB of SPU local store memory is used. With a single call to either `random_number` or `transpose`, however, 222 KB and 223 KB is used respectively. The same code using *both* functions also uses 223 KB.

Hence, the cost of calling a single Fortran intrinsic function is approximately 209 KB: over 80% of SPU local store memory. The local store is a *vital* resource for high performance Cell.B.E. codes, and this is clearly a critical obstacle for developers of Fortran code targeting the SPE.

Version 4.5 of GCC integrates a patch from Ulrich Weigand of IBM, ensuring that the GNU Fortran runtime library is built using the `-ffunction-sections` and `-fdata-sections` switches. This allows the linker to include only the functions which are actually called; rather than the monolithic approach of earlier versions. In use, the linker must also be instructed, by the end user, to garbage collect redundant sections using the following switch: `-Wl,-gc-sections`.

The bespoke version of GCC 4.1.1. included with the Cell SDK supports an extended version of C/C++ (IBM Corporation, 2007a). While version 4.5 of GCC provides the crucial Fortran library configuration explained above, it then fails to build a number of common libraries provided by the IBM Cell SDK. The lack of support for vector data types, and the `vector` keyword, is largely responsible. The latest GCC release, version 4.6, fully supports these vector types for a set of machine targets including the SPU. A version of GCC 4.6.0, targeting

the SPU, was built for integration in E♯. Therein, the Offload C++ compiler was configured to use GCC 4.6.0 for the final back-end code generation stage.

By this route, the cost of using a Fortran intrinsic routine on SPU has dropped considerably; to the point where they are once again useable. Table 6.1 presents the contrast between the two GNU Fortran runtime versions. The cost of including one call to the GNU Fortran runtime library is now approximately 40 KB. A call to another function requires only an additional 1-3 KB.

Configuration	GFortran 4.1.1 Footprint	GFortran 4.6.0 Footprint
No intrinsic calls	13 KB	22 KB
random_number only	222 KB	60 KB
transpose only	223 KB	62 KB
random_number and transpose	223 KB	63 KB

Table 6.1: SPU Local Store Memory Footprint when calling Fortran Intrinsics

6.1.4 New Operator SPU Memory Footprint

The `new` operator can also have a high cost in terms of the SPU local store memory footprint. An Offload C++ program with a single offload block, containing only the `SPU_LS_PROFILE` macro, will report that 43 KB of local store is used. Adding a call to `malloc`, to allocate `sizeof(int)` bytes, does not change this figure. However, calling the `new` operator, to provide an `int`, raises the local store footprint to a substantial 129 KB.

The `new` operator does more than allocate memory, for example it also calls the constructor of an object. Nevertheless, it seems unlikely that an additional 86 KB is required for such operations. Indeed, this contrast in memory footprints is not observed by a similar `new` and `malloc` test using the SPU version of G++ alone. Within E♯, the `new` operator is invoked by the constructor methods of the `ArrayT` and `FChar` classes. The `new` operator is thus integral to the object-oriented design of the runtime library; using `malloc` instead would require substantial code refactoring.

```
inline void *operator new     (size_t size) { return malloc(size); }
inline void  operator delete  (void *p)     { free(p); }
inline void *operator new[]   (size_t size) { return malloc(size); }
inline void  operator delete[](void *p)     { free(p); }
```

Listing 6.1: Overloaded C++ Allocators Reduce SPU Memory Footprint.

C++ operator overloading allows for custom definitions of `new` and `delete`. By overloading both, and also the two array versions, the memory footprint is observed to return to the zero level seen when using only `malloc`: 43 KB. Consequently, E♯ adds the definitions seen in Listing 6.1, at global scope, to every C++ file it generates.

6.1.5 Launch and Join Microbenchmark

To allay the memory resource limitations of the SPU, E♯ by default launches 128 threads to calculate the result of each array expression. Threads are administered from a task queue in a *round-robin* fashion, with only 6 parallel threads active at any one moment. Underpinning this approach is the assumption that the cost in time to launch and join the many threads is not prohibitive. Additionally, some array expressions may involve less than 128 elements, resulting in threads with zero workload. Hence, a *microbenchmark* was created, wherein the averaged time to launch and join n threads is measured. A minimal calculation involving incrementing an integer value by one is used to ensure the Offload C++ compiler does not optimise the thread launch away.

The timing results for the launch and join benchmark are provided in Figure 6.1. The results demonstrate a gradual transition from around 0.48 towards 0.5 milliseconds to complete the operation as the number of threads increases. These times are very low relative to the runtime of the benchmarks, and the almost linear response to rising thread counts is ideal. The result for one thread is the most notable outlier, and is presumably attributable to a one-off initialisation cost.

6.1.6 Benchmark Program Code Sizes

The 4 benchmark programs are provided as 'F' programs which are firstly converted to Offload C++ by the E♯ compiler. Table 6.2 provides a concise overview of the scale of these programs using rudimentary line statistics obtained using the Count Lines of Code (CLOC) program (Northrop Grumman Corporation, 2011). The 'F' Mandelbrot program is the smallest, at only 77 lines of code, while the Swaptions benchmark, formed by concatenating a number of separate source files, is the largest. No blank lines are present in the *generated* code, and precisely 5 lines of comment are present at the top of each file; describing the internal optimisation flags enabled by the current E♯ compiler build.

By way of comparison, the serial version of the most recent NAS Parallel Benchmark Suite, version 3.3.1, includes 10 benchmarks; ranging in size from the smallest, Embarrassingly

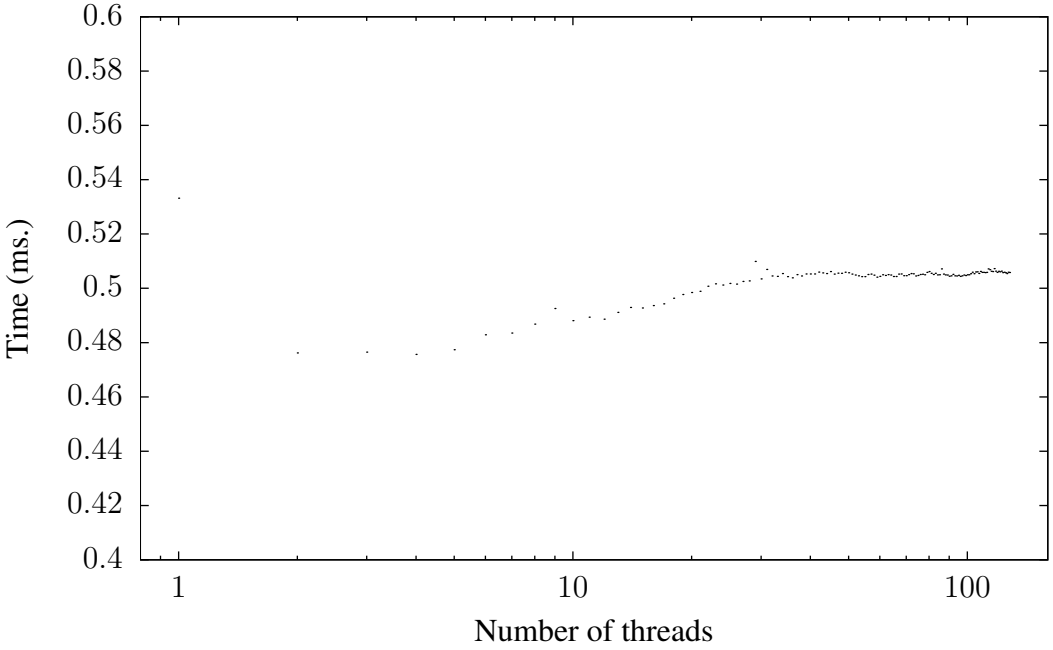

Figure 6.1: Average Duration of Minimal Kernel Launch and Join

	Original 'F' Line Counts			Generated C++ Line Counts		
Benchmark Name	Code	Comments	Blanks	Code	Comments	Blanks
Mandelbrot	77	21	27	173	5	0
BlackScholes	163	38	53	299	5	0
Swaptions	564	57	91	1022	5	0
n-Body	322	57	91	786	5	0

Table 6.2: Code Size Measurements for the 4 Canonical Benchmarks

Parallel (EP), at 150 lines, to the largest, Unstructured Adaptive (UA), at 5000 lines.

6.1.7 Compilation Times

The compilation times for the four canonical benchmarks across the three main stages of the E♯ pipeline are shown in Table 6.3. Times are roughly proportional to the number of lines of code, with the GNU Make tool requiring by far the longest time to complete each operation. Times to transfer code between PC and PS3 are not included.

Benchmark Name	E♯	Offload C++	GNU Make
Mandelbrot	0.25	1.7	11
BlackScholes	0.47	2.1	9.8
Swaptions	2.14	4.5	63.2
n-Body	1.89	3.0	19.7

Table 6.3: E♯ Compilation Stage Times in Seconds

6.2 Mandelbrot

Estimation of the Mandelbrot set requires iteration of the complex function $z_{n+1} = z_n^2 + c$. A pair of extents are specified from the command line and correspond to the dimensions of the 8-bit, grey-scale output image. An array of the same extents is first initialised with single-precision complex values representing instances of c in the equation. Each of these values is obtained through linear interpolation across the range between $(-1.5, -1.0i)$ and $(0.5, 1.0i)$. The requisite 'F' `elemental` function is then applied to each complex-valued element, and continues to iterate; either until $|z_{n+1}| \geq 2$, and so the original c value is *escaping*, and therefore not a member of the Mandelbrot set; alternatively, after a sufficiently large number of iterations, in our case 256, the value *is* considered a member. Note that the MIMD (Flynn, 1966) architecture of the CBE allows a Mandelbrot set candidate, rejected by an early *escape*, to contribute to dynamic load balancing; in contrast to the inefficient warp divergence of a SIMD data-parallel GPU. Each thread may therefore perform fewer than 256 iterations.

6.2.1 Serial Measurements

The measurements of Table 6.4 relate to the reference Mandelbrot set benchmark program, operating on five different problem sizes, using three different compilers. Listed along with each kernel duration, is the percentage of real time taken by the kernel, relative to that of the entire executable; as reported by the unix *time* command. The executable binary file size is also listed. E♯ uses G++ 4.6.0 to compile the generated C++.

Across the three problem sizes examined, the five years of development effort between versions 4.1.1 and 4.6.0 of GNU Fortran, elicit an average reduction in the Mandelbrot runtimes of 7%. Meanwhile, executable sizes have dropped by 43%. E♯ is on average 34% slower than GFortran 4.6.0, and produces executable files almost three times as large. The relatively high, and stable, proportion of the program's execution time spent within the E♯ kernel, 98%, can be taken as a positive indicator through consideration of Amdahl's Law in anticipation

of the parallel versions.

Problem Size	Compiler	Exe Size	Kernel Duration	% Real Time
512x512	GFortran 4.1.1	21 KB	8.2 secs.	97%
	GFortran 4.6.0	12 KB	7.6 secs.	96%
	E♯	33 KB	10.2 secs.	98%
768x768	GFortran 4.1.1	21 KB	18.7 secs.	96%
	GFortran 4.6.0	12 KB	17.2 secs.	97%
	E♯	33 KB	23.0 secs.	98%
1024x1024	GFortran 4.1.1	21 KB	32.9 secs.	96%
	GFortran 4.6.0	12 KB	30.5 secs.	97%
	E♯	33 KB	40.8 secs.	98%
1536x1536	GFortran 4.1.1	21 KB	74.1 secs.	93%
	GFortran 4.6.0	12 KB	68.7 secs.	92%
	E♯	33 KB	91.8 secs.	98%
2048x2048	GFortran 4.1.1	21 KB	131.7 secs.	94%
	GFortran 4.6.0	12 KB	122.2 secs.	88%
	E♯	33 KB	163.7 secs.	98%

Table 6.4: Mandelbrot Serial Results with -O3 Optimisation

6.2.2 Parallel Measurements

Measurements of the 'F' Mandelbrot program parallelised using E♯ were only viable for output images with side lengths of 512, 768, and 1024. This is due to E♯'s reliance on static partitioning, and the rising memory footprint of both input and output data; at the largest image width of 1024, 157 KB of the 256 KB provided by the SPU local store is used. This is unfortunate as the general tendency, also observed here in Table 6.5, is for speedup values to increase with data size.

Absolute speedup values of around 22 are obtained for each of the three problem sizes, relative to the PPU-only GFortran 4.6.0 serial version of Table 6.4. While only 6 SPU accelerators are utilised, the misleading term *super-linear speedup* is avoided. Such high speedup values can be explained by the cache-like effect of partitioning the data into portions which fit entirely within each SPU local store. Also, though the PPU and SPU have equal clock rates, the SPU has an entirely different SIMD architecture, one well suited to the Mandelbrot program.

Problem Size	Final SPU footprint	Kernel Duration	% Real Time	Abs. Speedup
512x512	86 KB	0.35 secs.	55%	21.9
768x768	115 KB	0.76 secs.	55%	22.6
1024x1024	157 KB	1.34 secs.	56%	22.8

Table 6.5: E♯ Mandelbrot Results

Complex Number Optimisation

The C++ standard library template class, `std::complex`, is used by E♯ as the default representation for a complex number. To probe the efficiency of its implementation, a minimal and new complex number class, `cx`, was developed. The results shown in Table 6.6 pertain to an experiment identical to the previous one, with the exception that the new complex number class is used. While this is a general optimisation, it is presented here as an application-specific optimisation, as complex numbers are not used by any of the other benchmarks.

Both complex number representations occupy the same amount of storage: 8 bytes for single precision; and 16 for double. The modest 1 KB reduction, relative to the use of `std::complex`, in SPU memory consumption shown in the second column of Table 6.6 is therefore likely due to the size and quantity of related C++ standard library definitions introduced by the linker. Timing results, and hence speedup values, are improved by a significant factor of approximately 3. This factor is accounted for by the implementation of the potentially expensive built-in 'F' function, `abs`, which calculates the absolute value of a number; in this case a complex number. For the new `cx` class, the `abs` function is defined in the same header file; qualified by the `inline` attribute; and defined by a single application of the real-valued C++ library built-in function for square root. In practise, the assembly language output of `abs`, applied to a value of type `cx`, manifests as an invocation of the SPU ISA instruction, `frsqest`(IBM Corporation, 2007c, page 21), the floating reciprocal square root estimate; when applied to a value of type `std::complex`, meanwhile, an apparently expensive call to the `cabsf` function from the C++ standard math library remains. The elevated performance consequently also reduces the percentage of total execution time taken by the kernel, to 29 %.

Problem Size	Final SPU footprint	Kernel Duration	% Real Time	Abs. Speedup
512x512	85 KB	0.12 secs.	29%	64.6
768x768	114 KB	0.25 secs.	29%	69.9
1024x1024	156 KB	0.43 secs.	29%	71.0

Table 6.6: E♯ Mandelbrot Results with Custom Complex Type

6.3 BlackScholes

The Black-Scholes, or Black-Scholes-Merton model, provides an abstract mathematical representation of a financial system, allowing the pricing of a portfolio of stock options using a partial differential equation known as the *Black-Scholes equation*. The BlackScholes benchmark program is the first of two financial simulations included within Princeton University's PARSEC benchmark suite (Bienia et al., 2008). The program was originally developed by Intel, with reference to the financial theory covered in Hull (2011, chapter 14). The code has been translated by hand from C/C++ into 'F' as part of the present research.

The implementation of the BlackScholes program included with the Parsec benchmark suite is written in C/C++, and provides the option to use either OpenMP, TBB, or POSIX Threads to facilitate parallelism using a data-parallel decomposition built around a *struct of arrays*. Performance scalability is obtained using a chunked, fine-grained partitioning of the serialised input and output data, calculating multiple options in parallel. Aside from the requisite file access and threading boilerplate, there are two user functions within the call graph of the parallel region: `BlkSchlsEqEuroNoDiv`, to calculate the option value; and a *callee* function, `CNDF`, which directly evaluates the cumulative normal distribution function.

```
for (j=0; j<NUM_RUNS; j++) {
  tbb::parallel_for(tbb::blocked_range<int>(0, numOptions), doall);
}
```

Listing 6.2: Multiple launches of the C++ TBB BlackScholes kernel

The benchmark runs the short `BlkSchlsEqEuroNoDiv` kernel 100 times, with each run launched by an application of the TBB `parallel_for` template function, shown in Listing 6.2. The first argument to `parallel_for` represents the indices of the domain; the second: an object of a user-defined class, `mainWork`, representing the calculation on one index. This invokes multiple parallel calls to `mainWork`'s overloaded function operator, `operator::()`, as shown in Listing 6.3. In 'F', the array assignment to launch the kernel is as shown in Listing 6.4. The 'F' version requires that the `BlkSchlsEqEuroNoDiv` function is qualified as `elemental`; and the `CNDF` function as `pure`. Whereas in C++ the six arguments to `BlkSchlsEqEuroNoDiv` are global pointers, in 'F' they are declared as `allocatable` arrays, local to the main program.

Input is provided by a set of files, with one line corresponding to one option. Each line has 9 fields comprising 8 real number values, and a character: either "P" for a *put*, or "C" for a *call* option. The output file contains one real number value, the calculated option price, on each line. Like the input, the first line of the output contains a single field: the number of options

```
void mainWork::operator()(const tbb::blocked_range<int> &range) const {
  int begin = range.begin();
  int end   = range.end();

  for (int i=begin; i!=end; i++) {
    prices[i] = BlkSchlsEqEuroNoDiv(sptprice[i],   strike[i], rate[i],
                                    volatility[i], otime[i],  otype[i]);
  }
}
```

Listing 6.3: Within the C++ TBB BlackScholes kernel

```
do i = 1, nruns
  prices=BlkSchlsEqEuroNoDiv(sptprices,strikes,rates,&
                             volatilities,times,otypes)
end do
```

Listing 6.4: Multiple launches of the 'F' BlackScholes kernel by array assignment.

in the file. Differences between the floating point results are comprehensively minimised within the code; however, the textual representation of these values is more notable. The Numdiff (Ivano Primi, 2010) program is hence used again to validate results based on a minimum absolute error of 1.0×10^{-4}.

6.3.1 Serial Measurements

Seven instances of the BlackScholes simulation were tested in serial. First of all, the original serial 'C' version of the program from the PARSEC suite is available. As with the Mandelbrot benchmark program, the two GNU Fortran compilers, and E♯ also provide the results from the hand-translated 'F' code. In addition, a second 'F' translation adopts an *array of structs* (AoS) design in contrast to the incumbent *struct of arrays* (SoA) approach. In this configuration, the kernel's `elemental` function accepts only one argument, of derived type `OptionType`.

Figure 6.2 shows that the original PARSEC 'C' version, compiled with version 4.6.0 of the GNU C compiler is the fastest, and hence will provide the reference for the speedup metric. Both versions of GNU Fortran show no significant timing difference between either the AoS, or SoA, approach. E♯, however, varies notably between a set of competitive serial times using the SoA approach, and the slowest recorded times using AoS; almost 80% slower than the PARSEC code at the largest problem size.

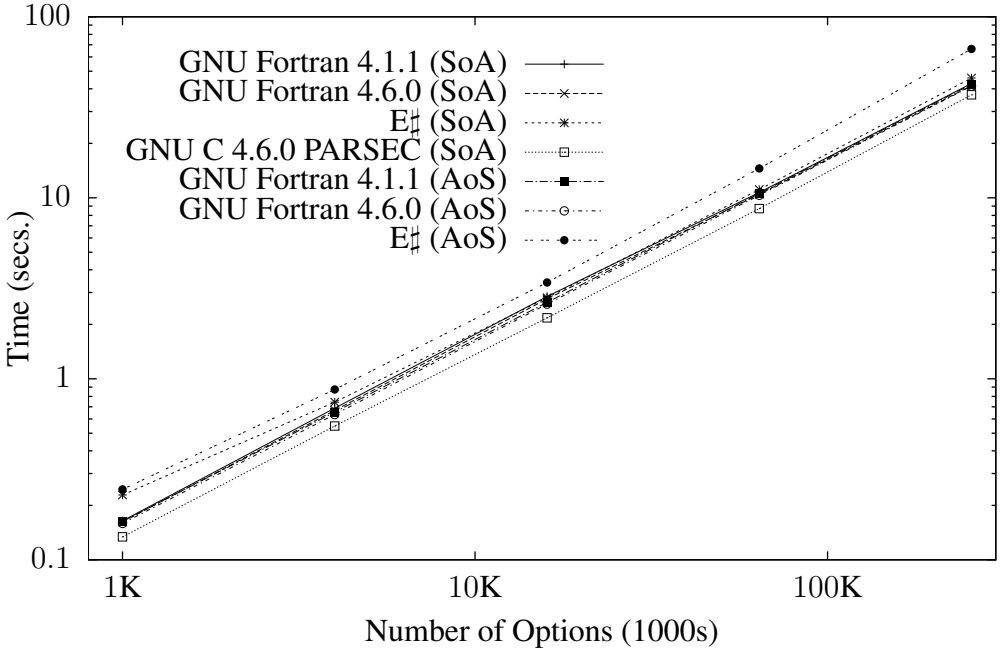

Figure 6.2: Log-Log Serial Timings for BlackScholes using -O3

The executable file sizes are shown in Table 6.7, with E♯ and GCC 4.6.0 again at the extremities. A slightly smaller executable is produced throughout by the AoS approach. This is accounted for by the AoS code's use of existing `OptionData`-based arrays which are used solely for parsing in the alternative SoA version.

Compiler	Struct of Arrays	Array of Structs
GNU Fortran 4.1.1	23 KB	22 KB
GNU Fortran 4.6.0	13 KB	12 KB
E♯	32 KB	34 KB
GNU 'C' 4.6.0	13 KB	-

Table 6.7: Executable File Sizes for Serial BlackScholes

The BlackScholes program can also be characterised by the percentage of total runtime taken by the kernel. Significant time is expended on both file input and output here, and this is reflected in the percentage figures shown in Table 6.8. The PARSEC code which was the fastest, also spends the largest proportion of its time within the kernel. E♯'s output, meanwhile, demonstrates a similar work ratio to the Fortran compilers which, after all, receive precisely the same inputs.

Data Structure	Compiler	1K	4K	16K	64K	256K
Struct of Arrays	GNU Fortran 4.1.1	76%	79%	80%	79%	79%
	GNU Fortran 4.6.0	81%	82%	83%	83%	83%
	E♯	79%	79%	79%	79%	80%
	GNU 'C' 4.6.0	88%	90%	91%	92%	91%
Array of Structs	GNU Fortran 4.1.1	72%	78%	78%	79%	78%
	GNU Fortran 4.6.0	80%	82%	82%	83%	82%
	E♯	81%	82%	81%	76%	83%

Table 6.8: Percentage of Runtime in Kernel for Serial BlackScholes

6.3.2 Parallel Measurements

The most notable feature of the E♯ results for parallel BlackScholes shown in Table 6.9 is the similarity in runtimes across all problem sizes. This is doubly significant as the absolute speedup values are relatively low. The explanation for this effect is that each SPU is given insufficient work to justify the transfer of data from host memory to SPU local store, and back. This postulation was confirmed by artificially increasing the workload within the main kernel function: a computationally expensive loop was added, resulting in far greater variation between the runtimes of different problem sizes. Indeed, beyond around 20 lines of straightforward floating-point arithmetic the main kernel function `BlkSchlsEqEuroNoDiv` makes only: two calls to the shorter, user-defined function `CNDF`; and one call each to the standard C++ math library functions `sqrt`, `log` and `exp`. Neither `CNDF` nor `BlkSchlsEqEuroNoDiv` contain a loop.

Problem Size	Final SPU footprint	Kernel Duration	% Real Time	Abs. Speedup
1K	58 KB	6.41 secs.	98%	0.02
4K	59 KB	6.37 secs.	97%	0.09
16K	61 KB	6.38 secs.	89%	0.34
64K	72 KB	6.51 secs.	69%	1.34
256K	113 KB	6.38 secs.	36%	5.81

Table 6.9: E♯ BlackScholes SoA Results

Times for the AoS variant of the BlackScholes program, shown in Figure 6.10 are remarkably similar; the difference in E♯ times observed in the serial test results of earlier Figure 6.2 have disappeared. One distinction remaining is the escalating consumption of SPU local store here. This is due to the existence of unused components in the derived type `OptionType`, used as the main `elemental` function's sole argument type.

The trend in both of these experiments is encouraging. The larger problem sizes are of course more interesting, and in both cases the absolute speedup rises above 1.0 when pricing with

64K options, and reaching almost 6.0 for 256K options.

Problem Size	Final SPU footprint	Kernel Duration	% Real Time	Abs. Speedup
4K	57 KB	6.34 secs.	97%	0.09
16K	62 KB	6.42 secs.	89%	0.34
64K	83 KB	6.47 secs.	68%	1.35
256K	166 KB	6.40 secs.	35%	5.80

Table 6.10: E♯ BlackScholes AoS Results

6.4 Swaptions

The Swaptions program prices a portfolio of interest-rate swap options using the Heath-Jarrow-Morton framework (Heath et al., 1990) using Monte-Carlo simulation. The program is the second of two financial simulations included with the PARSEC benchmark suite, and again originates from Intel. The original PARSEC code consists of around fifteen C++ source files, subsequently converted by hand to a single 'F' file as part of the research presented here. In either form, the swaptions benchmark program is the largest of the benchmarks tested with E♯.

A significant challenge in converting Swaptions to 'F' concerned the accuracy of results. Use of classic one-based array indexing in 'F', alongside untouchable zero-based additive offset iterators, gave early rise to numerous off-by-one errors. Plentiful non-default precision floating-point numeric literals were also handled with care. As with the BlackScholes benchmark program, results were inspected closely, and observed as essentially identical up to the least significant digit of the floating-point significands in the output. Textual differences in the output nevertheless remained, though again the Numdiff program provided some automated assistance within test scripts.

Parallel decomposition on both TBB and POSIX Threads implementations was, like BlackScholes, static and course-grained, though distinguished by a significantly larger working set. A SoA configuration was again present in the C++ code, and the kernel was dominated by a single 16-parameter function, `HJM_Swaption_Blocking`, applied in parallel to chunks from a one-dimensional iteration space.

As with the BlackScholes benchmark program, an alternative implementation was developed, wherein the main data structure is an array of a derived type, with a scalar field corresponding to the element type of each array in the original. This second version was created to explore the effects of data layout on performance; and is identified as the AoS configuration.

The C++ `HJM_Swaption_Blocking` function was ultimately a suitable target for `elemental` status in the 'F' translation, however the element type of two of its arguments are pointers to 1D and 2D arrays. An 'F' `elemental` function may only be defined for *scalar* arguments, so necessitating the definition of two new, derived, scalar types; to wrap each array. For example, with the 1D *pdYield* array, this amounts to the type shown in Listing 6.5.

```
1 type, public :: yieldT
2   real(kind=ki), dimension(m_iN) :: y
3 end type yieldT
```

Listing 6.5: A scalar 'F' datatype wrapping an array.

Input is provided at the command prompt by two integer arguments specifying the number of swaptions, and the number of simulations to run for each swaption. Each element of the arrays provided as arguments to the 'F' `elemental` function `HJM_Swaption_Blocking` is initialised from a single set of identical non-random constant values. Each element corresponds to an attribute of a swaption. Output is a verbose list of swaption prices to standard output. The computation of each swaption price is performed independently, and it follows that all output values are identical.

6.4.1 Serial Measurements

Serial timings for the Swaptions benchmark are shown in Figure 6.3. Again G++ 4.6.0, and three 'F' compilers were used; GFortran 4.6, GFortran 4.1.1, and E♯. Seven configurations were tested: unmodified PARSEC C++ code, which is in SoA form; hand converted 'F' code, also in SoA form and compiled by each of the three 'F' compilers; and 'F' code in *AoS* form, again compiled by the three 'F' compilers[3]. For clarity Figure 6.3 shows only the times for the SoA form of the 'F' code compiled by E♯. Times for the AoS version were almost the same: the arithmetic mean of the RMS difference between the times of the two methods, across all problem sizes, was only 0.28% $\pm$ 0.5. The regular spacing of the results between each swaption count, and the constant gradient of each curve indicate an $O(nm)$ algorithmic complexity, where n and m represent the number of swaptions, and number of simulations per swaption respectively.

As with the BlackScholes benchmark, the original PARSEC version, in this case a C++ code, produces the fastest kernel execution times among all seven configurations. This is on average only 9.59% $\pm$ 0.59% faster than E♯. The percentage of total runtime spent within

[3]Only -O2 optimisation was possible for the AoS GFortran 4.6 configuration, due to an unexplained runtime error when using -O3. Executable file sizes are still quoted for -O3.

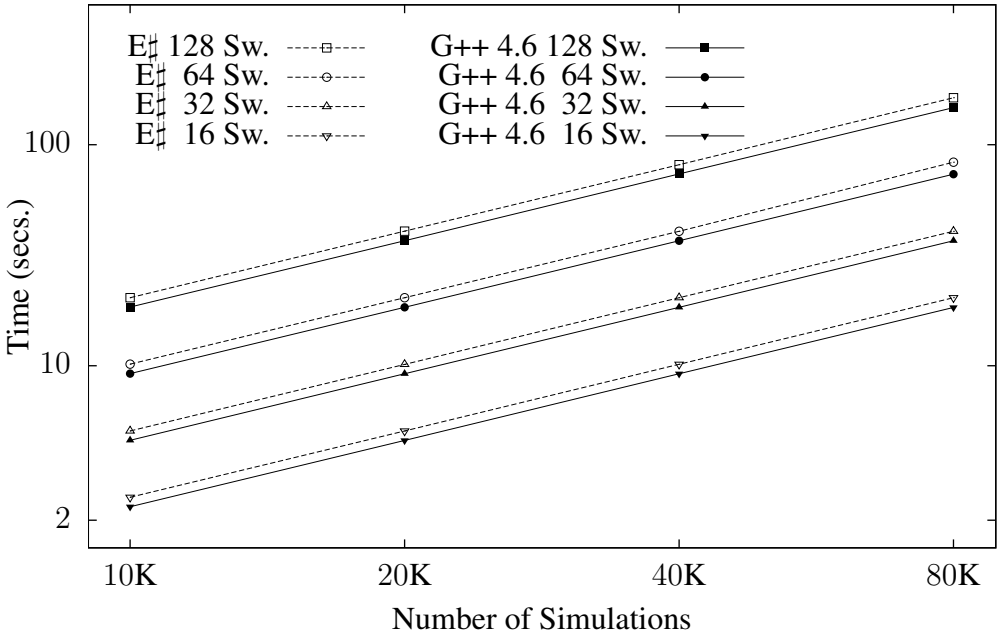

Figure 6.3: Log-Log Serial Timings for Swaptions (SoA) using -O3

the kernel is well above 99% for all versions, which reflects the minimal IO, initialisation, and deinitialisation in this benchmark. Relative to E♯, the PARSEC code kernel is active for a 0.18% $\pm$ 0.32 greater percentage of the total program execution time, averaged similarly across all problem sizes. Executable file sizes are shown in Table 6.11.

Compiler	Struct of Arrays	Array of Structs
GNU Fortran 4.1.1	34 KB	35 KB
GNU Fortran 4.6.0	35 KB	35 KB
E♯	43 KB	49 KB
GNU C++ 4.6.0	28 KB	-

Table 6.11: Executable File Sizes for Serial Swaptions

6.4.2 Parallel Measurements

When running the Swaption benchmark program, compiled by E♯, in *parallel*, the percentage of total program execution time located within the parallel kernel remains high: 99.36% $\pm$ 0.69 for the default SoA version; and 99.29% $\pm$ 0.80 for the AoS version.

Absolute speedup results are shown in Figure 6.4, and are quoted relative to the fastest available serial version: the C++ PARSEC implementation, compiled using G++ version 4.6.0. Two interesting features may be observed here. Firstly, though well above one, the absolute

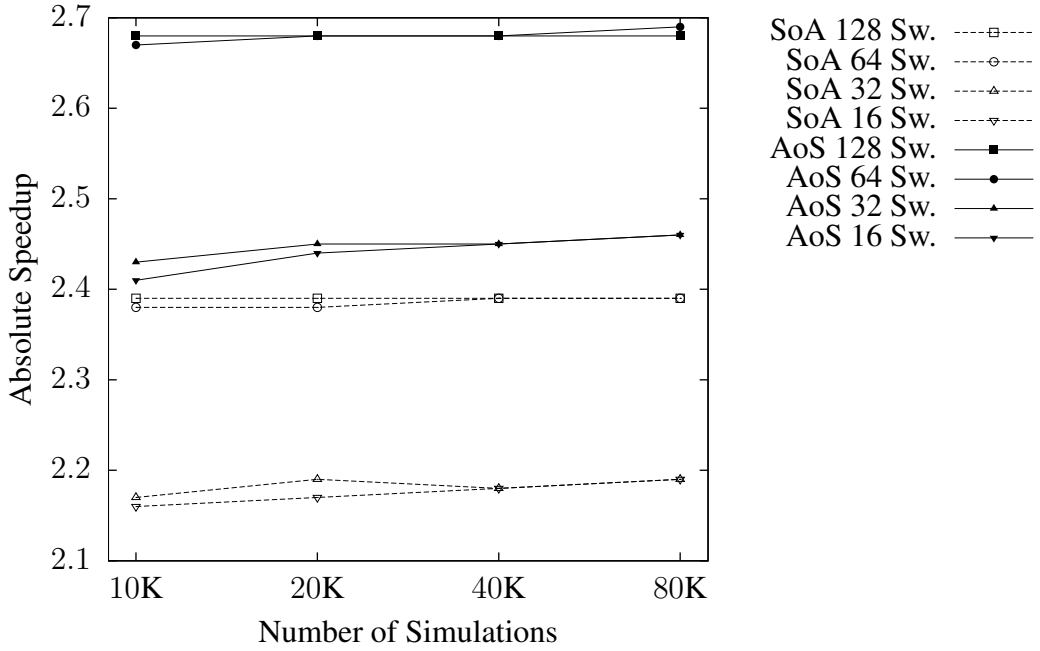

Figure 6.4: Speedup against Swaption and Simulation Quantities using 128 Threads

speedup values are the lowest of all the benchmarks. Secondly, the response to variations in the problem input parameters is minimal.

The amount of SPU memory used by the kernel is invariant across all 16 combinations of swaption and simulation quantities: 100KB and 91 KB for the SoA and AoS versions respectively. Ideally, these values would increase: a speedup greater than one indicates that the parallelisation is efficient; while larger working sets benefit from the minimised startup costs of bulk DMA transfer. The ratio of time spent by the SPU on computation, compared to that required to transfer an element of said computation is low, and hence the Swaptions program has a low arithmetic intensity.

The increase in speedup values experienced by both versions of E♯ swaptions may relate to the correspondence between the number of array elements active in a kernel, and the number of swaptions. With less than 128 swaptions, threads will be launched which perform no work. If t is the number of threads; sw is the number of swaptions, and $sw < t$, then there will be $t - sw$ such threads.

The range of problem parameters tested are comparable with the configurations provided by the PARSEC benchmark suite. The largest configuration, *native* was also tested: 128 swaptions, and 1 million simulations. Peak SPU local store usage for the AoS and SoA versions was again 100 KB and 91 KB respectively. Speedup values were 2.39 and 2.69 respectively; a trivial increase in the latter over the other problem sizes.

A similar test was performed with the largest number of swaptions supported by the SPU local store: 8192; and 1000 simulations. Speedup values remained unchanged again here: 2.39 and 2.68 for the two cases.

6.5 The n-Body Problem

Development of the bespoke 'F' language n-Body simulation began with Simon Geard's serial Fortran Jovian planet simulation code, from the top of the corresponding table on the Computer Language Benchmarks Game (CLBG) website (Fulgham, 2011). This code was then overhauled. First of all the code was altered to accept a variable quantity of initial values: either read in from a file, or obtained by a repeatable stochastic process. Secondly, and more substantially, it was apparent from earlier work in Donaldson et al. (2008) that an $O(n^2)$ *all-pairs* n-body simulation on the Cell B.E. can exhibit good scaling, though with little impact on the wall clock time; and producing a maximum speedup of only 1. To address this, a tiled decomposition of the problem was developed, based on an Nvidia algorithm (Lars Nyland and Prins, 2007) for the heterogeneous CUDA GPU architecture. Though the complexity of the modified *all-pairs* algorithm remains $O(n^2)$, the use of computational tiles maximises the ratio of computation to data transfer.

The kernel of our n-body algorithm performs the $O(n^2)$ force calculation in parallel while the remaining leapfrog-Verlet integration (Verlet, 1967), which updates the positions and velocities, executes in serial on the PPU host processor. This choice seems reasonable as having only linear complexity, the percentage of runtime expended on the remaining integration stage becomes insignificant with larger body counts. A square-shaped tile of the pairwise body interactions, maximises the number of calculations that can be performed per body. That is to say, a DMA transfer of $2p$ body positions and masses, will provide p^2 components of force for the integrator.

Derived types are once more required for the input and output array elements, which again must be scalar. Hence, for input and output respectively, the two types, `pchunk2d` and `accel_chunk`, were created, and are as shown in Listing 6.6. The `pchunk2d` is used to point to pairs of contiguous sections from an array of real-valued 4-tuples, `vec4`, representing the velocity and mass of each body. Partitioning the `vec4` velocity and mass data of the bodies into ns sections produces ns^2 *chunks*; that is, values of type `pchunk2d`.

The use of pointers within the `pchunk2d` is a concession to the memory limitations of the host. In addition to the original data, the `pchunk2d` array requires only $8 \times ns^2$ bytes; where 8 is the size of a `pchunk2d` value. The alternative approach of allocating fresh memory for

```
integer, public, parameter :: XCHUNKS    = 16
integer, public, parameter :: YCHUNKS    = XCHUNKS
integer, public, parameter :: NUM_CHUNKS = XCHUNKS * YCHUNKS
integer, public, parameter :: CHUNK_SIZE = NBODIES/XCHUNKS

type, public :: pchunk2d
  type(vec4), pointer, dimension(:) :: ivec4, jvec4
end type pchunk2d

type, public :: accel_chunk
  type(vec3), dimension(CHUNK_SIZE) :: avec3
end type accel_chunk
```

Listing 6.6: The n-body kernel input (`pchunk2d`) and output (`accel_chunk`) wrapper types

all ns^2 section combinations requires $n \times ns^2$ bytes.

The E♯ compiler parallelises only the outermost of the generated loops. To fully exploit the two-dimensional decomposition already outlined, a flattened, *one-dimensional* array of `pchunk2d` elements is used to feed the obligatory, parallelising 'F' `elemental` function.

For benchmarking purposes, input is provided statically through integer constants defining the number of planets; the number of timesteps; and the size of a timestep. Position, velocity and mass are defined pseudo-randomly for each body using a constant random seed. Verification of the accuracy of results uses an energy checksum which is calculated before and after the simulation; and once again monitored using the Numdiff program. Of the four benchmarks, only the n-Body problem is a data parallel algorithm: each thread is guaranteed the same control path, and an equal portion of the workload.

6.5.1 Serial Measurements

Four versions of the n-body benchmark are measured. The 'F' language version is compiled by E♯, and by both version 4.1.1, and version 4.6 of the GNU Fortran compiler. Additionally, a 'C' version of the program, also derived from code on the CLBG website, was included and compiled using GCC 4.6. Serial timings were measured over 20 iterations, and averaged to provide times for 1 iteration. For all test configurations written in 'F', a 16×16 decomposition of the velocity and mass data; i.e. the `XCHUNKS` was set to 16.

Figure 6.5 demonstrates a log-log relationship between the problem size and the kernel runtime. While all four versions of the program performed comparably, the three versions coded in 'F' produce particularly similar timings. It is though the 'C' version which is the fastest, for all problem sizes tested; 40% faster, on average, than the version compiled by E♯; falling

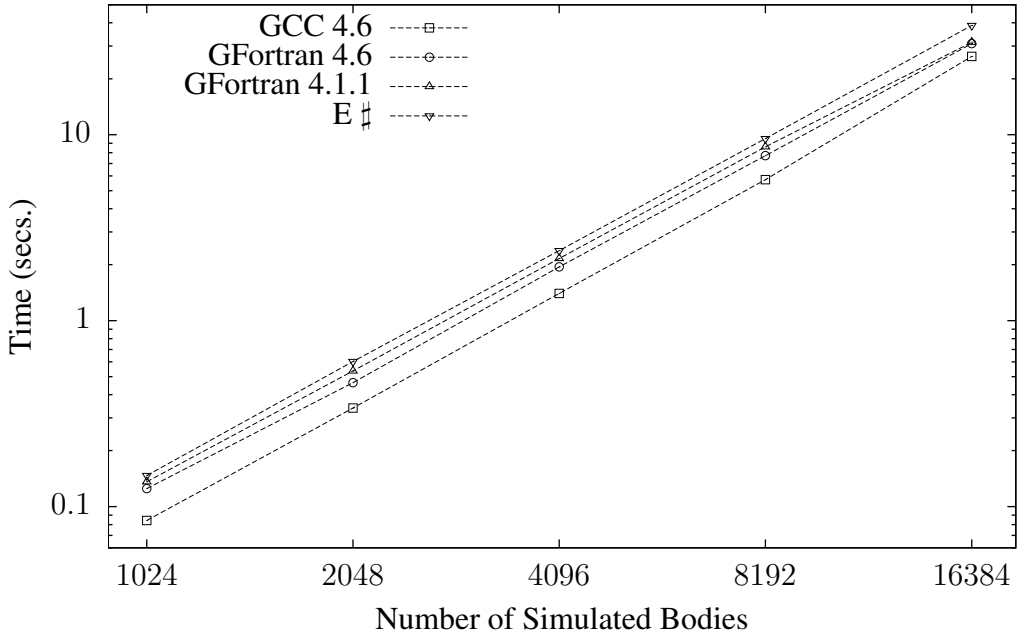

Figure 6.5: Log-Log Serial n-body Timings for a Single Timestep

to 32% with 16K bodies.

Table 6.12 provides the file sizes of the four program versions. The figures are provided for all problem configurations, as each requires recompilation due to the use of static specification. Most of the sizes for each compiler are nevertheless very similar, as the data for the bodies is not accounted for by such measurements. This is the only benchmark program where E♯ does not produce the largest files; GFortran 4.1.1 has that distinction. This may be due to the static specification of the problem configuration. Also notable is the inexplicably large file size when simulating 2048 bodies with GNU Fortran 4.6.

Compiler/Body count	1024	2048	4096	8192	16384
GCC 4.6.0	8453	8453	8453	8453	8453
GNU Fortran 4.6	15134	70203	15220	15220	15188
GNU Fortran 4.1.1	27873	27854	27854	27854	27854
E♯	23022	23099	23156	23156	23156

Table 6.12: Executable File Sizes in Bytes for Serial n-Body

The percentage of time spent within the n-body kernels, which includes the serial update of positions and velocities, is shown in Table 6.13. That a high proportion of wall clock time is spent within the kernel indicates that the n-body program is a viable target for acceleration.

Compiler/Body count	1024	2048	4096	8192	16384
GCC 4.6.0	90%	90%	90%	90%	91%
GNU Fortran 4.6	95%	90%	95%	95%	95%
GNU Fortran 4.1.1	95%	96%	96%	96%	87%
E♯	95%	95%	95%	95%	95%

Table 6.13: Percentage of Runtime in Kernel for Serial n-Body

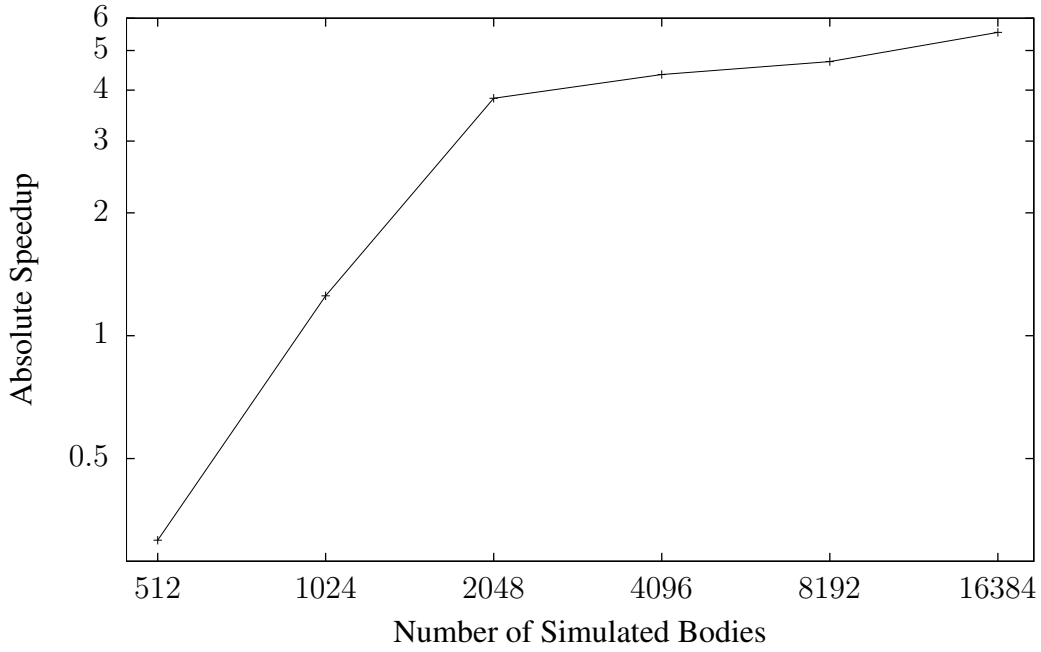

Figure 6.6: Log-Log n-body Absolute Speedup using 128 Threads

6.5.2 Parallel Measurements

The n-body program was compiled by E♯ for *six* different problem sizes. Again, a 16×16 decomposition of the velocity and mass data was employed. Figure 6.6 presents the results as the relationship between the problem size and the absolute speedup, relative to the fastest serial version running on PPU only; i.e. the 'C' version. The 512-body problem size was added to the original set of five, which were examined for the serial execution timings, to help explain the jump in speedup shown by the curve between 1024 and 2048 bodies. The curve demonstrates the high sensitivity to the ratio of computation to data transfer size seen with the smaller problem configurations of 512 and 1024. Problem sizes of 2048 and above revert to a more moderate, though still increasing trend. A problem size of 32768 was not possible due to the limited memory resource of the SPU in tandem with E♯'s static partitioning. The final speedup value obtained from the largest problem size of 16384 is a reasonable 5.5.

The percentage of time spent within the parallel kernel, shown in Table 6.14 reflects the same

Problem Size	512	1024	2048	4096	8192	16384
Percent	94.30%	87.00%	71.33%	70.46%	69.74%	69.29%

Table 6.14: Percentage of Runtime in Kernel for Parallel n-Body

sensitivity to the kernel's arithmetic intensity observed by the speedup metric.

Problem Size	512	1024	2048	4096	8192	16384
SPU Memory	99 KB	101 KB	107 KB	117 KB	139 KB	182 KB

Table 6.15: Final SPU Memory Footprint within Parallel n-Body kernel

The quantity of SPU local store memory utilised at the point of the kernel's completion are shown in Table 6.15. The change in memory consumption is apparently proportional to the problem size, and indicates that with 32768 bodies, the 256 KB of SPU local store would be insufficient to run the program. This is confirmed by observation. The executable file size is 176 KB for all configurations of the n-body program compiled by E♯.

6.6 Conclusion

We conclude the investigation into the performance characteristics of the E♯ compiler positively. Absolute speedup for the largest problem sizes is well above one for each of four medium-size, bespoke, HPC benchmarks, measured under a number of configurations. A small E♯ local store footprint transpired to be critical for strong performance, and this was attained most notably by an overload of the default `new` operator; and use of an updated GCC back-end; as described in Section 6.1.4 and Section 6.1.3 respectively.

For each performance data point reported, the SPU memory footprint is included, and governs the upper limit on problem size for all four benchmarks. The Mandelbrot, BlackScholes, and n-Body programs all demonstrate a direct proportionality between problem size and absolute speedup; though while Mandelbrot and n-Body appear to approach a peak speedup, at the largest problem sizes, Table 6.9 and Table 6.10 indicate that the BlackScholes benchmark has more to offer, and is especially constrained by the 256Kb limit on SPU local store. The curious speedup curve for the Swaption benchmark in Figure 6.4 appears to have reached its peak even at the smallest data size. This indicates that the Swaptions algorithm is providing few calculations per DMA datum transferred; put another way, it exhibits low *arithmetic intensity*.

All of the benchmarks have a somewhat regular structure, insomuch as they can be represented by an array expression; equivalent to a parallel for loop. Nevertheless, only the n-Body benchmark is actually a data parallel algorithm. Each thread in the remaining benchmarks has the opportunity to return early from a calculation, allowing it to assist in a dynamic balancing of an irregular workload. This is particularly noticeable in the Mandelbrot benchmark, which will encounter numerous contiguous areas, within which *all* set candidates are rejected; corresponding to a region of background colour in the output image.

In a similar vein, it would be rewarding to quantify the irregularity of each benchmark, by measuring the deviation in workload across the participating thread team. A parallel reduction benchmark would make a useful addition, providing a useful insight to the overheads of data transfer in the E♯ implementation.

The Mandelbrot benchmark program exhibits speedup values well in excess of six; corresponding to an *efficiency* greater than one. Six, however, is merely the number of available SPUs. As the architecture of the SPU is entirely different from the PPU, upon which the serial reference version runs, the term *super-linear* speedup is inappropriate; as intimated in Section 6.2.2. A serial SPU reference benchmark could *also* produce an absolute speedup greater than 6, due to a cache-like effect when a problem's entire working set fits within the SPU local store. Only with an idealised SPU, having infinite local store, should a strict limit of six on the absolute speedup be expected.

In light of such intricacy, it may also be helpful to evaluate the number of floating point operations per second (flops) used by each benchmark. This would provide an *absolute* indication of the performance of each kernel, relative to the theoretical maximum for the CBE architecture.

Chapter 7

Conclusion

The competitive market for high performance semiconductor components continues to foster rising transistor counts in accordance with Gordon E. Moore's famous observation from 1965 (Moore, 1965). A decrease in die size, and so wire diameter, allows a corresponding reduction in power consumption; so facilitating a proportional increase in clock speeds. Successive processor generations ran existing, unmodified, programs faster; and with apparent inevitability.

Unfortunately, fundamental physical factors no longer permit such a straightforward relationship. Resistive-capacitive delays in signal transmission lengthen as clock speeds rise; while transistor leakage increases as microfabrication scales fall. Meanwhile the relatively high latency to access off-chip memory, known as the memory wall, becomes increasingly pronounced.

The packing of multiple processor cores on a single chip has become a mainstream solution to some of these issues; providing lower clock speeds per core, while aggregate floating-point capability increases. Novel transistor designs too, such as Intel's Tri-Gate transistor, provide a stay of execution for the slowing decrease in CMOS processing scales. *Heterogeneous multicore* chips, such as those found in the Cell B.E.; graphics processing units; or the Intel SCC; also address the problem of the memory wall, by the provision of addressable caches; increased bandwidth to main memory; and fast context switching. An associated cost is often reduced functionality of the individual accelerator cores.

Inevitably the burden of increased hardware complexity is transferred to software developers. However, while the operating system can assist with the concurrent scheduling of small, discrete programs; large-scale commercial, scientific, and enterprise software requires an intuitive parallel interface within the application layer. An ideal solution should combine traits

including performance, programmability, portability, and reliability. Perhaps inevitably, no such amalgamation was manifested by review, and it looks increasingly unlikely that any single solution can address the needs of all domains. Thus, the software landscape looks likely to continue its reflection of an equally disparate hardware one.

7.1 Thesis Review

The central thesis presented by this dissertation asserts that the challenges of programming heterogeneous multicore architectures can be radically simplified, while simultaneously providing high-performance, by the use of an implicitly parallel, array-based programming language.

As introduced in Chapter 1, the syntax and semantics of a well-defined subset of Fortran, have been selected, and combined with an implicitly parallel execution model, targeted at the contemporary, heterogeneous multicore architecture of the CBE processor. The E♯ compiler was developed for the sole purpose of transforming serial programs written in 'F', into parallel Offload C++ (Cooper et al., 2010; Russell et al., 2010) programs, producing identical outputs. A suite of tests and benchmarks confirmed the correctness of results, and facilitated the measurement of timing, scaling and performance figures.

Chapter 2 provided an overview of contemporary industrial and academic research on programmability and performance of libraries and languages targeting heterogeneous multicore architectures. It became evident that the field is highly active due to the ongoing multicore revolution. Commensurate with this, no single paradigm, and certainly no one solution prevails.

Chapter 3 presented the methodology adopted by the research. Discussed therein are the technologies logistically at my disposal, including the heterogeneous multicore architecture of the CBE, and the PS3; the dual address space Offload C++ language and compiler; and also the 'F' programming language specification. The opportunities available for automatic parallelisation in 'F' are also examined, and it is established that suitable targets are array expressions involving elemental operators or procedures, whether intrinsic or user-defined.

In Chapter 4 the design of the E♯ compiler was presented in detail. The E♯ compiler is shown as part of a toolchain including the Offload C++ compiler; and both GCC compilers, and cross-compilers. A monadic approach to parsing populates the *parse tree*, which, taken together with the typed 'F' AST, and Offload C++ AST, comprise the three intermediate program representations of the E♯ compiler. The C++ language is found to provide a

comparable level of abstraction to ‘F’; with some exceptions. Derived types, for example, require the careful generation of a number of member methods to support features such as serialisation, structure constructors, and assignment. The locality of underlying data must also be statically expressed using integer template parameters, in keeping with the statically-typed, dual address space Offload C++ extension. Array expressions, of course, also require bespoke handling in C++, and their representation as nested loops is also developed. The application of a high-level, generic transformation technique to the parallelisation of array expressions is then described, building on the strengths of the implementation language; Haskell. Finally, the completed optimisations for fixed-size, stack-allocated arrays and character strings are presented; though regrettably there has not been time to test their impact on performance.

In compiler development, one should aim for as much functionality as possible to reside in the accompanying runtime library. Chapter 5 firstly presents a new C++ class template, `ArrayT`, directly compatible with the GNU Fortran runtime library; yet configurable for *all* Fortran compilers. This is essential for E♯, as no common ABI exists for the multifarious dope vectors employed by Fortran compiler vendors to represent arrays. A class template to represent ‘F’ character strings, `FChar` is also developed. Through their methods, both classes provide all of the functionality expected of their ‘F’ analogues. A unique aspect of these class templates is the integration of an integer template argument to set the address space of each data pointer. The address space, or depth, template argument is used by template metaprograms to compare the relative depth of method arguments, and act accordingly. Likewise, template arguments of the `this` pointer, may be mixed with new template arguments to construct fresh parameter and result types; as seen in the `ArrayT::section` method of Section 5.2.3. Template metaprograms are also demonstrated which specify type-level operations for the C++ compiler to perform, which would otherwise exist within the ‘F’ compiler. The chapter concludes by introducing and solving a problem of code replication within Offload C++ by the provision of two new statically-located, foundational class templates: a located smart pointer; and a located version of the `new` operator. Together they can eliminate the class of problems outlined.

Chapter 6 provides experimental results which contribute towards a strong conclusion regarding the performance benefits of using the ‘F’ language and E♯ compiler. Four moderately-sized benchmark programs are introduced. Significantly, and fundamental to E♯’s approach, each of the benchmark programs is written in the ‘F’ dialect, a proper subset of the dominant HPC language, Fortran. Hence, no parallel language constructs or extensions are required; each program may be debugged and executed in serial. Serial runtimes, facilitated via the E♯ toolchain, are shown to be between 9% and 32% slower than the fastest available alternative for the largest problem instances. Although slower, these are encouraging results, given that

no serial optimisation has yet been pursued. Parallel results on the largest data sets provide absolute speedups consistently well above one: 23-71 for Mandelbrot; 6 for Blackscholes; 2.1-2.7 for Swaptions; and 5.5 for the n-Body simulation. Performance results are sensitive to the SPU memory footprint, and such measurements are also reported. Additional statistics concerning the size of codes; the size of binary files; compilation times; and kernel durations with respect to program runtimes are also supplied.

7.2 Limitations

Certain limitations and early decisions in the research should be acknowledged. Therefore, rather than categorise the following points as future work, they are presented as related, though with substantial divergence from the existing work.

The lack of a GPU implementation is apparent in the current investigation. The field of research concerned with using commodity graphics chips, to perform general, and high performance computing, has blossomed incredibly in the last five years. This is not a surprise; HPC research compilers with CUDA back-ends have existed for some time (Keir, 2007). Nevertheless, although current graphics chip architectures remain steadfastly SIMD, the scion of Intel's Larrabee (Seiler et al., 2008) is a reminder that alternative ideas may emerge. Like Larrabee, the CBE is also a MIMD design, allowing a data-parallel interface to drive a more flexible task-based implementation; exemplified by E♯'s realisation of the Mandelbrot benchmark.

That *all* array expressions are parallelised by E♯ can be seen as a limitation. On the other hand, the compiler-related issues with performance portability experienced by HPF (Kennedy et al., 2007) may speak to the contrary. In any case, there remains enormous potential for research into accurate cost-models for contemporary multicore architectures. It should also be mentioned that E♯ does include support for a parallelism-prohibiting, `nopar` statement and construct modifier. The design of E♯ was considered to be weakened by its presence; from the point of view that the language was no longer standard 'F'; thereby reducing compiler compatibility. An OpenMP-style compiler directive embedded within a comment string would be a better approach.

The language selected as an interface to the present research is based on the 'F' subset of Fortran. Along with the highly competitive performance results of Chapter 6, this ensures the research remains relevant to the field of HPC. Nevertheless, such emphasis on the support of a legacy, industrial language, albeit the leaner 'F', applies a corresponding restriction on the opportunities for language design. As an example, it is a notable deficiency that only

scalar functions may be promoted for application to array expressions. A similar facility to apply functions operating on arrays of rank lower than the target expression, would remove the present requirement to wrap arrays in bespoke user-defined types.

Nested data-parallelism allows a greater range of problems to be tackled than is possible using only traditional, flat arrays. Research in this area has also been applied to array languages, such as Guy Blelloch's classic work on NESL (Blelloch, 1996), or the more recent Data Parallel Haskell project (Peyton-Jones et al., 2008). A nested data-parallel extension to, or new language based on, 'F' or Fortran could be a rewarding research direction.

7.3 Future Work

The following opportunities for further research have arisen naturally from the current work, and include only the most promising or immediate from a broad range of possibilities.

7.3.1 Homogeneous Parallel Architecture

A back-end for *homogeneous multicore* would introduce support for a greater range of parallel hardware. Adding this feature to E♯ could be achieved most straightforwardly using OpenMP. The Offload C++ compiler's intrinsic overload of pointer dereferencing for DMA transfers, ensures the generated code, for each loop body, is already in a form suitable for a traditional single address space execution context. Thus, the currently explicit, static, partitioning of the outermost array dimension would become redundant, and may be replaced by that arising implicitly from an OpenMP *parallel for* directive.

7.3.2 Alternative Partitioning

E♯ currently partitions the outermost dimension of an array expression into contiguous segments, or chunks; with each allocated as a task for processing by an individual SPE thread. The number of these segments is equal to the number of threads; and set by the environment variable, `ESHARP_NUM_THREADS`, having a maximum value of 128. Operationally, each of the 6 available SPEs hosts one thread at a time, with threads administered from the PPE using a FIFO queue. Consequently, the SPU memory footprint of a chunk is inversely proportional to the number of threads. By increasing the number of threads, a user modifies the SPU

memory footprint of each segment; potentially accommodating expressions with a larger footprint per element.

This interface is lacking, primarily because the minimum segment length is controlled solely by the choice of thread count. For example, with 128 threads, a 128 million element array expression would require one million input and output elements to be resident on each SPE. A secondary deficiency is the requirement to introduce the low-level implementation details of threads, and SPU local memory, to a user. Finally, the number of threads selected affects *all* parallelised array expressions in a program; so providing a suboptimal configuration for expressions with a small memory footprint.

Inner Dimensional Partitioning

Partitioning across only the *outermost* dimension of an array expression, places an unnecessary restriction on the lower limit of the resulting SPU memory footprint. For example, a $128 \times 64 \times 64$, three-dimensional array expression will require that each SPU processes a *minimum* of 64×64 array elements. It is hence recommended that should the user request a number of threads greater than the outermost array expression extent, successive inner dimensions should also be partitioned for parallel execution.

Unlimited Threads

The finite limit of 128 Offload threads relates to a constant value defined within the MARS runtime library (Sony, 2008): `MARS_WORKLOAD_MAX`. While increasing this value, and rebuilding MARS, could facilitate benchmarks otherwise excluded due to their specific memory footprint, a more scalable solution is at hand. Upon user specification of a large number of threads, launch and join them in batches of 128. This approach is powerful enough to decompose any array expression, into single element chunks if required; though, alas, the user still tinkers with implementation details.

Dynamic Partitioning

An alternative, more intricate proposal is a *dynamic* work queue, wherein a thread is permanently resident on each of the six participating SPEs throughout the evaluation of an array expression. Each thread is initially allocated the largest chunk of the array expression contained by 256 KB of SPU memory. After evaluating a segment, the result is returned to main

memory, allowing a similarly-sized segment from those remaining to be assigned; until no more remain.

A dynamic work queue would permit the execution of a range of array expressions which are both comprised of numerous elements; and excluded due to the large memory footprint of a segment. The performance of array expressions which already run, should either remain the same; or, in situations where more threads than required were configured, improve.

Implementation will require some care. Ideally the SPU memory footprint could be calculated in advance of a thread's execution. This is, however, rendered intractable by the presence of recursion: while an 'F' *elemental* function cannot itself be recursive, a pure, recursive function may be called from one; an unbounded `do` construct presents similar complexity. An empirical approach could provide initial task threads with minimal, single-element segments, before attempting larger segments with each successive task.

Such a system would remove the need for the user to specify the number of threads. Instead, the number of participating coprocessors could be chosen; though *all available* would be a reliable default.

7.3.3 Host Participation

The host processor is largely idle during the evaluation of an array expression. There is therefore potential to treat the PPU as another accelerator, and provide it also with segments of a calculation. The design of the MARS runtime library (Sony, 2008) is particularly supportive in that the MARS kernel on each SPE often executes with complete autonomy from the PPU. Communication with the PPU is required only for the thread-join synchronisation following conclusion of the array expression workload. Hence, the two-way hyperthreaded architecture of the PPU can allow *two* host threads to participate in the processing of each workload.

The implementation can build upon E♯'s current partitioning and scheduling system. The major percentage of the threads requested by the user will be created on the SPUs, exactly as before. The work of those remaining will be executed by the PPU; after the asynchronous launch of the SPU threads; and prior to the join. The additional code generated for the PPU need only omit the `offload` keyword, and may otherwise remain structurally identical to the SPU code. Some optimisation to remove redundant copying may though be beneficial.

7.3.4 Reduction Operations

Reduction operations on 'F' arrays are restricted to a small set of logical and arithmetic operators such as `sum`, `product`, and `any`. Rather than parallelise such an ad hoc collection, a general, higher-order, reduction operation is proposed; `reduce`. 'F', and Fortran 95, both support procedure arguments, however the generic nature of the `reduce` type signature would require the support for polymorphism introduced in Fortran 2003/2008 (INCITS/J3, 2010). Additionally, the scalar binary reduction operator provided via the procedure pointer parameter, must be both associative and commutative. Confirmation of this would remain the responsibility of the end user, as no relevant proof mechanism exists within 'F'.

Support for generic parallel reduction would substantially enhance the E♯ programming model, and facilitate a far larger range of problems; including those adhering to the *map-reduce* model (Dean and Ghemawat, 2004). The OpenMP ARB too is currently also investigating user-defined reductions (Kambadur et al., 2008; Duran et al., 2010), so ensuring a timely and robust discussion.

7.3.5 Functional Programming Constructs

E♯'s focus on *expressions*, and their execution in parallel, *can* seem mildly at odds with the procedural, statement-oriented nature of 'F', or Fortran. This could be addressed by extending E♯'s palette outside the 'F' language.

Let Expressions

The introduction of the *associate construct* (INCITS/J3, 2010, 170-171) to Fortran 2003 provides a *let expression* to the language. The construct allows implicitly-typed names to be associated with subexpressions for the lexical scope of its *block*. For E♯, the benefit of supporting the construct is one of readability: in attempting to reduce the performance overheads peripheral to successive parallel executions, an E♯ user may choose to create lengthy, single-line array expressions. Such usage patterns can be supported in a more readable format by implementing the associate construct.

Recursion

Often an entire array expression must be executed repeatedly, as in the case of a relaxation algorithm. A straightforward approach would enclose, say, an array assignment, in a `do` construct. Currently, E♯ will move the working data set between main memory and SPU local stores at each such iteration. While the array assignment is "pure", the `do` construct may not be, and further analysis would be required for non-trivial cases.

'F' procedures may call themselves only when declared with a preceding `recursive` clause. Nevertheless, in 'F', the common process of iteration is rarely expressed using recursion; looping constructs such as `do` loops are used ubiquitously instead. Beyond tradition, the significant reason that iteration through recursion is avoided in 'F' and Fortran is due to the correlated consumption of stack memory. However, if the result of a function call is immediately returned by the caller, an optimisation known as *tail call elimination* may be employed, wherein the call adds no stack frame to the call stack; the recursive call graph is essentially implemented as a loop. Some, especially functional, languages *guarantee* that this optimisation will be performed. In 'F' there is no such assurance, and a performance-oriented user is, currently, highly unlikely to program in such a manner.

It is proposed that E♯ support a new clause inspired by the `recur` "special form", from the Clojure language (Halloway, 2009, page 150). In Clojure, `recur` accepts the same parameters as the enclosing function, and has the effect of calling that function with the given arguments[1]. Distinctively, an error will be issued if the call to `recur` is not in a valid tail call position, thus providing assurance to the programmer. An equivalent construction in 'F' should accommodate the difference between calling an' F' function, and a subroutine. A suitable addition is a `TAILCALL` clause, handled as a new terminal in the grammar production: *prefix* of 'F'; or *prefix-spec* of Fortran 2008 (INCITS/J3, 2010, page 305).

7.3.6 Native Execution of Test Cases

As the complexity of compiler transformations has increased, the E♯ compiler test framework, described in Section 5.5, has provided invaluable assurance against *functional regressions*. The system is, however, restricted to the examination of *serial* builds of the test programs; and under Windows. As most recent development relates to parallelisation transformations, extending the system to also compile and execute parallel test programs, natively on the PS3, would be highly beneficial.

[1] The *become statement* (Winterbottom, 1993, page 22) of Plan 9's Alef language, was similarly constructed for explicit tail call optimisation.

Operationally, the current system applies shell scripts to compile a set of ‘F’ test programs from ‘F’ into executable programs, either directly using GFortran; or, via C++, with E♯ and G++. The standard output from both programs are then compared in a pairwise fashion using the numeric *diff* tool, Numdiff (Ivano Primi, 2010). To extend this system, either the Windows and PS3 machines must communicate; or the compilers should execute on the PS3.

Converting the E♯ toolchain to PS3 Linux is, alas, impractical. Emulation of the Offload compiler using *Wine* (Julliard, 2012) is impossible as it targets only x86 architectures. Meanwhile, to retain compatibility with the dormant Cell SDK, the version of Linux on the project’s PS3 is the ageing Fedora Core 7. Unmet package dependencies thereby prevent recent GHC distributions from building on PS3; as required for a native E♯ on PS3.

A pragmatic alternative would use scripts to facilitate communication and file transfer between the Windows host and the PS3. Under Windows, the E♯ and Offload compilers can together translate each ‘F’ input to the familiar low-level, intermediate ‘C’ form. A secure copy (SCP) of these files, along with each ‘F’ program, should then be made to the PS3. A *remote procedure call* can then invoke the following steps on the PS3: conclude the build using a native ‘C’ toolchain; execute and compare the outputs of both programs; and return the result to the Windows machine.

The ability to execute test programs on the PS3, natively in parallel, introduces a further opportunity, to monitor *performance regression*. The unix *time* command may again be used to report the execution time for the entire program, while each kernel should also output its duration; as each benchmark does currently. Further scripting would now be required, as a simple *diff* will be inadequate for monitoring such subtlety: while *Numdiff* (Ivano Primi, 2010) can report on relative differences in numeric values, it may require to differentiate performance timing values, from data values. Each output value which relates to performance, could be tagged with a distinct string. One approach would then partition the output using the unix stream editor, *sed*; then supplying two separate invocations of *Numdiff* with input.

7.4 A Final Thought

The potential of array, or collection-based, languages and libraries, to provide a simple and scalable interface targeting scalar, and both homogeneous and heterogeneous multicore architectures, remains as potent at the conclusion of this research, as at its inception. The foundations of future parallel software technology are perceived to lie within programming language research, and compiler engineering; which must itself also strive for increased accessibility.

Appendix A

Locality Allocator Classes

```
template <typename T,int Od=__OFFLOAD_DEPTH__>
  struct oi_ptr;

template <typename T>
struct oi_ptr<T,0> {

  typedef T element_type;
  static const int depth = 0;

  inline explicit oi_ptr(T *p = 0) : m_p(p) { };

  inline operator T *() const { return m_p; }

  inline oi_ptr &operator=(const oi_ptr &oip) {
    m_p = oip.m_p; return *this;
  }

  inline oi_ptr &operator=(T *p) { m_p = p; return *this; }

  inline       T &operator[](const int i)       { return m_p[i]; }
  inline const T &operator[](const int i) const { return m_p[i]; }
  inline       T &operator* ()                  { return *m_p;   }
  inline const T &operator* ()            const { return *m_p;   }
  inline       T *operator->()                  { return m_p;    }
  inline const T *operator->()            const { return m_p;    }

private:
  T inout(0) *m_p;
};
```

```
template <typename T>
struct oi_ptr<T,1> {

  typedef T element_type;
  static const int depth = 1;

#ifdef __sievecplusplus
                        inline          oi_ptr()          : m_p(0) {}
  template <typename U> inline explicit oi_ptr(U)                 {}
#endif

  offload               inline explicit oi_ptr(T *p = 0) : m_p(p) {}

  inline operator T inout(1) *() const { return m_p; }

#ifdef __sievecplusplus
          inline oi_ptr &operator=(const oi_ptr &   )   { return *this;  }
          inline oi_ptr &operator=(T * )                { return *this;  }
          inline       T &operator[](const int i)       { return m_p[i]; }
          inline const T &operator[](const int i) const { return m_p[i]; }
          inline       T &operator* ()                  { return *m_p;   }
          inline const T &operator* ()            const { return *m_p;   }
          inline       T *operator->()                  { return m_p;    }
          inline const T *operator->()            const { return m_p;    }
#endif

  offload inline oi_ptr &operator=(const oi_ptr &oip)   { m_p = oip.m_p;
                                                          return *this;  }
  offload inline oi_ptr &operator=(T *p)                { m_p = p;
                                                          return *this;  }
  offload inline       T &operator[](const int i)       { return m_p[i]; }
  offload inline const T &operator[](const int i) const { return m_p[i]; }
  offload inline       T &operator* ()                  { return *m_p;   }
  offload inline const T &operator* ()            const { return *m_p;   }
  offload inline       T *operator->()                  { return m_p;    }
  offload inline const T *operator->()            const { return m_p;    }

private:
  T inout(1) *m_p;
};
```

Appendix B

SPU Local Store Footprint Macro

```
#define SPU_LS_PROFILE(tid,str) \
{ \
  int i, *ps[256], ok = 1, cnt = 0; \
  for (i = 0; i < 256 && ok; i++) { \
    ps[i] = 0; \
    ps[i] = (int *)malloc(1024); \
    if (ps[i] != 0) \
      cnt++; \
    else \
      ok = 0; \
  } \
  printf("[Thread %3d] (%s)\t%d KiB of 256 KiB SPU Local Store used.\n", \
         tid, str, 256-cnt);  \
  for (i = 0; i < cnt; i++) { \
    if (ps[i] != 0) \
      free(ps[i]); \
  } \
}
```

Appendix C

A Brief Introduction to Haskell

An executable Haskell program consists of one or more pure function definitions, with `main` as the default entry point. A function definition consists of its name; parameters; the equality symbol; and an expression denoting the result. The following function will return the first of its two arguments:

```
const x y = x
```

As above, type signatures are routinely omitted from function definitions. The Glasgow Haskell Compiler (GHC) interpreter will report the type signature of `const` using the `:type` command. The response to `:type const` is `const :: a -> b -> a`, indicating that with two arguments of arbitrary type, the result will have the type of the first. The definition of `const` is therefore parametrically polymorphic. If we would prefer `const` to restrict the first argument to values of `Integer` type, an explicit type signature can be supplied:

```
const :: Integer -> b -> Integer
const x y = x
```

Function application is performed by juxtaposition. For example `const 1 2` will evaluate to 1. Parentheses may still be required to alter the evaluation order, with `const 1 2 + 3` evaluating to 4, while `const 1 (2 + 3)` evaluates to 1.

Haskell is a *lazy*, or *non-strict* language. The compiler of a *strict* language such as C++ *may*, for example, avoid evaluating the `(2 + 3)` subexpression of `const 1 (2 + 3)`, as an *optimisation*. In Haskell this is formalised, and *every* expression which is not required, *will not* be evaluated.

The function composition operator `(.)` can be used to represent, and apply functions in sequence. The expression `(g . f) x` means apply `f` to `x`, and then apply `g` to the result.

Functions in Haskell may be partially applied, or *curried*. For example, the following nullary function definition involves the composition of two curried `const` applications, and evaluates to `True`:

```
gf = (const True . const False) 3.142
```

The following definition of `toString` is restricted to argument types which are instances of the `Show` type class. The `Show` type class supports one function, `show`, which will produce a string, given a valid argument. The type signature of `show` is the same as `toString`; which is `(Show a) => a -> String`.

```
toString x = show x
```

User defined *algebraic data types* (ADTs) may be specified using the `data` keyword. In the code below, `Foo` is a *type constructor*, while `A` and `B` are *data constructors*. The types listed after the data constructors are known as *components*. Data constructors can be used to define *patterns*, introduced below, however they are first of all functions; albeit ones which start with a capital letter. For example, `A` has type `String -> Bool -> Foo`.

```
data Foo = A String Bool
         | B Double
```

Pattern matching can be used to prepare function definitions based on cases for ADT arguments. The following function, `fooString`, uses patterns such as `(A s b)` and `(B i)` on its left hand side. `fooString` also introduces string literals, such as `"A"`, and the string concatenation operator, `(++)`[1]. All basic Haskell types, such as `String`, `Bool`, and `Double`, are instances of the `Show` type class, and hence the function `show` may be applied to the component values, *without* requiring that `fooString` itself be an instance of `Show`; its type is simply `Foo -> String`.

```
fooString (A s b) = "A" ++ " " ++ show s ++ " " ++ show b
fooString (B i)   = "B" ++ " " ++ show i
```

The use of patterns may be avoided by declaring an ADT with *record syntax*. By this method, the constructor components are each preceded by a name and double colon, and delimited by

[1] Many such common functions, operators and data types are defined within a standard library known as the Haskell Prelude.

commas. Each name becomes an accessor function for the corresponding component. For example, a value `A s b` of type `Foo` defined below, could access its `String` component using `a1 (A s b)`.

```
data Foo = A { a1::String, a2::Bool }
         | B { b1::Double }
```

An alternative method to introduce a new type arises from the `newtype` keyword. The `newtype` keyword can replace the use of the `data` keyword in situations where the type has just *one* constructor and *one* component. After type checking, the Haskell compiler may implement values of such a type efficiently, using only the type of the `newtype`'s one constructor component.

A type is made an instance of a type class by providing a definition for each of the function signatures specified by that type class. Type classes facilitate a form of ad hoc polymorphism. An instance of the `Show` type class for `Foo` types is shown below:

```
instance Show Foo where
  show x = fooShow x
```

An appropriate definition for the `Show` type class is also shown:

```
class Show a where
  show :: a -> String
```

The definitions for a number of simple type classes, including `Show`, are in fact so straightforward that an automatic definition can be obtained. The following definition of the `Foo` ADT uses the `deriving` clause to instigate a `Show` instance for `Foo`, and applications of `show` will produce output identical to that of the hand-constructed code above.

```
data Foo = A String Bool
         | B Double
  deriving (Show)
```

An alias to a type may be created using a `type` declaration. Note, however, that the alias is not distinct: a value having an alias type can always be exchanged for a value of the original type. The example below defines `I` as an alias for `Integer`; while the definition of `const` will accept, and return, values of `Integer` type.

```
type I = Integer

const :: I -> a -> I
const x _ = x
```

Note above the use of the underscore pattern _ may be used whenever a name binding is not required.

A lambda expression can be used to define an *anonymous* function. For example, `\x y -> x` defines a lambda function which behaves as the first version of `const`. Lambda function application also uses juxtaposition; `(\x y -> x) 1 2` evaluates to 1. The type of a lambda expression can also be explicitly specified. Assuming `I` is again an `Integer` alias, the following is an alternative definition of the explicitly typed `const` function above, using a lambda expression[2].

```
const = (\x _ -> x) :: I -> b -> I
```

A list is a common, recursively defined ADT. A list is created either empty, or by prepending an element to an existing list. The list type in Haskell has special syntax. The infix constructor `(:)` correlates to the `Cons` constructor, while `[]` correlates to `Nil`. A value such as `Cons 1 (Cons 2 (Cons 3 Nil))` can then be denoted as `1:2:3:[]` and also `[1,2,3]`.

A common requirement is to apply a function to each element of a list. A higher order *map* function can be used to implement this operation. For a Haskell list this function is called `map`, and can be defined recursively as shown below. Note here the Haskell list syntax `[]` may also be used as a pattern, on the left hand side of `map`'s defining equations.

```
map _ []     = []
map f (x:xs) = f x : map f xs
```

A Haskell list is isomorphic to the following ADT, which introduces both a type constructor with one parameter[3], `List`, and the use of recursion in type definitions.

```
data List a = Nil | Cons a (List a)
```

A *map* function may be similarly defined for this `List` type:

[2]Haskell uses the \ symbol due to its similarity to the Greek letter lambda: λ.

[3]Analogous to a function, the `List` type constructor is distinguished from the `Foo` type constructor by its *kind*. While `Foo` is of kind `*`, `List` is of kind `* -> *`.

```
lmap _ Nil         = Nil
lmap f (Cons x xs) = Cons (f x) (lmap f xs)
```

A map-like function for a newly defined ADT is a common requirement. The idea may be precisely expressed in terms of a *Functor*, a concept borrowed from a branch of abstract mathematics known as category theory. In Haskell, `Functor` is a type class with one function, `fmap`:

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

The instance of `fmap` for a Haskell list is defined simply as `map`. Therefore `fmap f (x:xs)` will produce the same result as `map f (x:xs)`. A `Functor` instance of the `List` ADT can similarly be defined using the earlier definition of `lmap`. On this occasion the `Functor` type class expects a type constructor of kind `* -> *`; a partial application of `List`[4]. Note that a type synonym, created using a `type` declaration, cannot be partially applied so.

```
instance Functor List where
  fmap = lmap
```

The definition of `fmap` for `List` is relatively straightforward. For more complex ADTs it is useful to refer to the following rules, that all instances of the `Functor` type class should satisfy.

$$fmap\ id = id$$
$$fmap\ (g\ .\ f) = fmap\ g\ .\ fmap\ f$$

Monads also originate in category theory, and may be used to model the combination of type and effects presented, for example, by IO operations. Every monad is also a functor. The following `Monad` type class definition enforces this condition by constraining the `m` types to those which are instances of the `Functor` type class.

```
class (Functor m) => Monad m where
  return :: a -> m a
  (>>=)  :: m a -> (a -> m b) -> m b
```

The `Monad` instance for the `List` type will require a definition for both the `return` and `(>>=)` type class functions. The `m` type parameter, of kind `* -> *`, is instantiated to the `List` type

[4] A *full* application of the `List` type constructor, such as `List Float`, has a kind of `*`.

constructor. The `return` function definition is easily prepared: the definition should create a list from a single element. This may be achieved by adding the element to the start of an empty list:

```
return x = Cons x Nil
```

The `(>>=)` operator, also know as *bind*, is highly applicable in Haskell; however, another monadic primitive, *join*, often has a more intuitive definition. Furthermore, `(>>=)` may be defined in terms of `fmap` and `join`. The Haskell type signature of `join` is shown below.

```
join :: (Monad m) => m (m a) -> m a
```

The type signature of `join` becomes `List (List a) -> List a` when applied to the `List` ADT. The function should convert a list of lists into a list. A simple concatenation operation involving accumulated appends is the intuitive, and ultimately correct solution. The definitions for append, `appl`, and concatenate, `concatl`, are listed below.

```
appl :: List t -> List t -> List t
appl Nil         ys = ys
appl (Cons x xs) ys = Cons x (appl xs ys)

concatl :: List (List a) -> List a
concatl Nil         = Nil
concatl (Cons x xs) = appl x (concatl xs)
```

The `List` definition for the monadic *join* function follows:

```
join = concatl
```

The `Monad` type class for `List` may then be defined as follows. The definition of `(>>=)` given, in terms of `fmap` and `join`, is true for all monads.

```
instance Monad List where
  return x = Cons x Nil
  xs >>= f = join (fmap f xs)
```

For an instance of the `Monad` type class to meet the formal requirements of a monad, the following set of rules must be adhered to. Note that the Haskell compiler will not perform the verification of such rule sets.

$$join\ .\ fmap\ join = join\ .\ join$$
$$join\ .\ fmap\ return = join\ .\ return = id$$

Unlike the `fmap` operation of a functor, a monad's `(>>=)` operator can also alter the *shape* of a monad. Both of the following test functions produce a new list with elements equal to 2 removed; i.e. `[1,3]`, and `Cons 1 (Cons 3 Nil)`.

```
test1 = [1,2,3] >>= \x -> if x==2 then []  else [x]
test2 = let  xs = Cons 1 (Cons 2 (Cons 3 Nil))
        in   xs >>= \x -> if x==2 then Nil else Cons x Nil
```

Some monads are capable of expressing the sequencing of calculations or effects. The IO monad in particular allows IO to coexist alongside the purity and laziness of the Haskell language. In fact monads are sufficiently well supported by Haskell as to have their own syntax: *do notation*. For example, a personalised greeting program may be prepared using the bind combinator:

```
main = getLine >>= \n -> putStrLn ("Hello " ++ n)
```

Alternatively, the following, equivalent program uses do notation to provide a representation more akin to the sequenced statements of an imperative language.

```
main = do n <- getLine
          putStrLn ("Hello " ++ n)
```

Bibliography

Sarita V. Adve, Stephen Heumann, Rakesh Komuravelli, Jeffrey Overbey, Patrick Simmons, Hyojin Sung, and Mohsen Vakilian. A Type and Effect System for Deterministic Parallel Java. In *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2009.

Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986. ISBN 0-201-10194-7.

Gail A. Alverson, William G. Griswold, Calvin Lin, David Notkin, and Lawrence Snyder. Abstractions for Portable, Scalable Parallel Programming. *IEEE Transactions On Parallel and Distributed Systems*, 9:72–86, 1998.

C. Amza, A.L. Cox, S. Dwarkadas, P. Keleher, Honghui Lu, R. Rajamony, Weimin Yu, and W. Zwaenepoel. TreadMarks: Shared Memory Computing on Networks of Workstations. *Computer*, 29(2):18–28, Feb 1996.

Andy Vaught. The G95 Project. *http://www.g95.org*, 2009.

Krste Asanovic, Ras Bodik, Bryan C. Catanzaro, Joseph J. Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William L. Plishker, John Shalf, Samuel W. Williams, and Katherine A. Yelick. The Landscape of Parallel Computing Research: A View From Berkeley. Technical Report UCB/EECS-2006-183, University of California at Berkeley, December 2006.

Eduard Ayguadé, Marc Gonzàlez, Jesús Labarta, Xavier Martorell, Nacho Navarro, and José Oliver. NanosCompiler: A Research Platform for OpenMP Extensions. In *First European Workshop on OpenMP*, pages 27–31, 1999.

J. W. Backus, R. J. Beeber, S. Best, R. Goldberg, L. M. Haibt, H. L. Herrick, R. A. Nelson, D. Sayre, P. B. Sheridan, H. J. Stern, I. Ziller, R. A. Hughes, and R. Nutt. THE FORTRAN AUTOMATIC CODING SYSTEM. In *Proceedings of the Western Joint Computer Conference*, pages 188–198, February 1957.

John Backus. The History of FORTRAN I, II and III. *Annals of the History of Computing, IEEE*, 20(4):68–78, October-November 1998.

J. Balart, A. Duran, M. Gonzàlez, X. Martorell, E. Ayguadé, and J. Labarta. Nanos Mercurium: a Research Compiler for OpenMP. In *Proceedings of the 6th European Workshop on OpenMP*, EWOMP 2004, pages 103–109, 2004.

Kevin J Barker, Kei Davis, Adolfy Hoisie, Darren J Kerbyson, Mike Lang, Scott Pakin, and Jose C Sancho. Entering the Petaflop Era: The Architecture and Performance of Roadrunner. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, SC '08, pages 1–11. IEEE, November 2008.

D. W. Barron. *An Introduction to the Study of Programming Languages*. Cambridge University Press, 1977. ISBN 0521213177.

Ayon Basumallik, Seung jai Min, and Rudolf Eigenmann. Towards OpenMP Execution on Software Distributed Shared Memory Systems. In *Proceedings of the 2nd International Workshop on OpenMP: Experiences and Implementations (WOMPEI'02)*, volume 2327, pages 457–468. Springer Verlag, 2002.

Pete Becker. *Working Draft, Standard for Programming Language C++*, February 2011.

Pieter Bellens, Josep M. Perez, Rosa M. Badia, and Jesus Labarta. CellSs: a Programming Model for the Cell BE Architecture. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, SC '06, pages 86–86. ACM, 2006.

Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proceedings of the Seventeenth International Conference on Parallel Architectures and Compilation Techniques (PACT'08)*, pages 72–81. ACM, 2008.

Guy E. Blelloch. Programming parallel algorithms. *Communications of the ACM*, 39:85–97, March 1996.

W. Blume, R. Doallo, R. Eigenmann, J. Grout, J. Hoeflinger, and T. Lawrence. Parallel Programming with Polaris. *IEEE Computer*, 29(12):78–82, Dec 1996.

Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An Efficient Multithreaded Runtime System. *SIGPLAN Notices*, 30(8):207–216, 1995.

Robert L. Bocchino, Jr., Vikram S. Adve, Sarita V. Adve, and Marc Snir. Parallel Programming Must Be Deterministic by Default. In *First USENIX Workshop on Hot Topics in Parallelism*, HotPar'09, 2009.

François Bodin, Peter Beckman, Dennis Gannon, Srinivas Narayana, and Shelby X. Yang. Distributed pC++: Basic Ideas for an Object Parallel Language. *Scientific Programming*, 2(3):7–22, 1993.

Hans-J Boehm. Threads cannot be implemented as a library. *ACM Sigplan Notices*, 40(6): 261, 2005.

Eugene Brooks and Karen Warren. Development and Evaluation of an Efficient Parallel Programming Methodology, Spanning Uniprocessor, Symmetric Shared-memory Multiprocessor, and Distributed-memory massively Parallel Architectures. Poster session at the 1993 ACM/IEEE Conference on Supercomputing, December 1995.

Timothy A. Budd. An APL Compiler for a Vector Processor. *ACM Transactions on Programming Languages and Systems*, 6:297–313, July 1984.

J. M. Bull, M. D. Westhead, M. E. Kambites, and J. Obdržálek. Towards OpenMP for Java. In *Proceedings of the Second European Workshop on OpenMP*, EWOMP '00, pages 98–105, 2000.

William W. Carlson and Jesse M. Draper. Distributed data access in AC. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '95, pages 39–47, New York, NY, USA, July 1995. ACM. ISBN 0-89791-700-6.

K. Mani Chandy and Carl Kesselman. CC++: A Declarative Concurrent Object Oriented Programming Notation. Technical Report CS-TR-92-01, California Institute of Technology, 1993.

Paul Cockshott and Kenneth Renfrew. *SIMD Programming for Linux and Windows*. Springer Verlag, 2004. ISBN 1-84996-920-5.

Pete Cooper, Uwe Dolinsky, Alastair F. Donaldson, Andrew Richards, Colin Riley, and George Russell. Offload - Automating Code Migration to Heterogeneous Multicore Systems. In *Proceedings of the 5th International Conference on High Performance and Embedded Architectures and Compilers (HiPEAC'10)*, volume 5952 of *Lecture Notes in Computer Science*, pages 337–352. Springer, 2010.

David E. Culler, Andrea Dusseau, Seth Copen Goldstein, Arvind Krishnamurthy, Steven Lumetta, Thorsten von Eicken, and Katherine Yelick. Parallel Programming in Split-C. In *Proceedings of the 1993 ACM/IEEE Conference on Supercomputing*, volume 0, pages 262–273. ACM, November 1993.

Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*. USENIX Association, 2004.

Uwe Dolinsky. *Offload C++ language extension specification 1.0.1*, October 2010.

Alastair F. Donaldson, Paul Keir, and Anton Lokhmotov. Compile-time and Run-time Issues in an Auto-parallelisation System for the Cell BE Processor. In *Proceedings of the 2nd Euro-Par Workshop on Highly Parallel Processing on a Chip (HPPC'08)*, volume 5415 of *Lecture Notes in Computer Science*, pages 163–173. Springer, 2008.

Chris Dornan and Simon Marlow. Alex: A lexical analyser generator for Haskell. *http://www.haskell.org/alex/*, 2011.

Georg Dotzler, Ronald Veldema, and Michael Klemm. JCudaMP: OpenMP/Java on CUDA. In *Proceedings of the 3rd International Workshop on Multicore Software Engineering*, IWMSE '10, pages 10–17. ACM, 2010.

Allen B. Downey. *Physical Modeling in MATLAB*. Green Tea Press, 2008. ISBN 0615185509.

Alejandro Duran, Roger Ferrer, Michael Klemm, Bronis de Supinski, and Eduard Ayguadé. A Proposal for User-Defined Reductions in OpenMP. In *Beyond Loop Level Parallelism in OpenMP: Accelerators, Tasking and More*, volume 6132 of *Lecture Notes in Computer Science*, pages 43–55. Springer Berlin / Heidelberg, 2010.

A. E. Eichenberger, J. K. O'Brien, K. M. O'Brien, P. Wu, T. Chen, P. H. Oden, D. A. Prener, J. C. Shepherd, B. So, Z. Sura, A. Wang, T. Zhang, P. Zhao, M. K. Gschwind, R. Archambault, Y. Gao, and R. Koo. Using advanced compiler technology to exploit the performance of the Cell Broadband Engine™ architecture. *IBM Systems Journal*, 45(1):59–84, January 2006.

Rudolf Eigenmann, Jay Hoeflinger, Robert H. Kuhn, David A. Padua, Ayon Basumallik, Seung-Jai Min, and Jiajing Zhu. Is OpenMP for Grids? In *Proceedings of the 16th International Parallel and Distributed Processing Symposium*, IPDPS '02, pages 171–178. IEEE Computer Society, 2002.

F Syntax Rules (1996). *http://www.fortran.com/F/F_bnf.html*.

Adin D. Falkoff and Kenneth E. Iverson. The Evolution Of APL. *SIGAPL APL Quote Quad*, 9:30–44, September 1978.

Kayvon Fatahalian, Timothy J. Knight, Mike Houston, Mattan Erez, Daniel Reiter Horn, Larkhoon Leem, Ji Young Park, Manman Ren, Alex Aiken, William J. Dally, and Pat Hanrahan. Sequoia: Programming the Memory Hierarchy. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, page 83. ACM, 2006.

Michael J. Flynn. Very High-Speed Computing Systems. *Proceedings of the IEEE*, 54(12): 1901–1909, 1966.

Geoffrey Fox, Seema Hiranandani, Ken Kennedy, Charles Koelbel, Ulrich Kremer, Chau wen Tseng, and Min you Wu. Fortran D Language Specification. Technical Report TR90-141, Department of Computing Science, Rice University, December 1990.

Free Software Foundation. Bison - GNU parser generator. *http://www.gnu.org/s/bison/*, 2007.

Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The Implementation of the Cilk-5 Multithreaded Language. *SIGPLAN Notices*, 33(5):212–223, 1998.

Brent Fulgham. The Computer Language Benchmarks Game. *http://shootout.alioth.debian.org/*, 2011.

Youssef Gdura and Paul Cockshott. A Virtual SIMD Machine Approach for Abstracting Heterogeneous Multicore Processors. In *Proceedings of the 2nd Annual International Conference on Infocomm Technologies in Competitive Strategies*, ICT 2011, pages 12–17. Global Science and Technology Forum, 2011.

GNU Project. GCC, the GNU Compiler Collection. *http://gcc.gnu.org*, 2012.

Clemens Grelck and Sven-Bodo Scholz. SAC - A Functional Array Language for Efficient Multi-threaded Execution. *International Journal of Parallel Programming*, 34:383–427, August 2006.

Jing Guo, Jeyarajan Thiyagalingam, and Sven-Bodo Scholz. Breaking the GPU programming barrier with the auto-parallelising SAC compiler. In *Proceedings of the sixth Workshop on Declarative Aspects of Multicore Programming (DAMP'11)*, pages 15–24. ACM Press, 2011.

Stuart Halloway. *Programming Clojure*. Pragmatic Bookshelf, 2009. ISBN 1934356336.

Hiroo Hayashi. SpursEngine - Architecture Overview and Implementation. In *FAIS Multi-core Processor Workshop 2008*, October 2008.

David Heath, Robert Jarrow, and Andrew Morton. Bond Pricing and the Term Structure of Interest Rates: A Discrete Time Approximation. *The Journal of Financial and Quantitative Analysis*, 25(4):419–440, 1990.

High Performance Fortran Forum. High Performance Fortran Language Specification, version 1.0. Technical Report CRPC-TR92225, Center for Research on Parallel Computation, Rice University, January 1993.

J. P. Hoeflinger. Extending OpenMP to Clusters. White Paper, Intel Corporation, 2006.

Jay P. Hoeflinger and Bronis R. de Supinski. The OpenMP Memory Model. In *Proceedings of the first International Workshop on OpenMP*, volume 4315 of *Lecture Notes in Computer Science*, pages 167–177. Springer, 2005.

H. Peter Hofstee. Power Efficient Processor Architecture and The Cell Processor. In *HPCA*, pages 258–262. IEEE Computer Society, 2005.

Lei Huang, Barbara Chapman, and Ricky Kendall. OpenMP for Clusters. In *Proceedings of the Fifth European Workshop on OpenMP*, EWOMP 2003, pages 22–26, 2003.

Benedict Huber. Language.C - A C99 library for Haskell. *http://trac.sivity.net/language_c/*, 2011.

John Hughes. The Design of a Pretty-printing Library. In Johan Jeuring and Erik Meijer, editors, *Advanced Functional Programming*, volume 925 of *Lecture Notes in Computer Science*, pages 53–96. Springer Verlag, 1995.

John C. Hull. *Options, Futures, and Other Derivatives*. Prentice Hall, 8th edition, 2011. ISBN 0132777428.

Graham Hutton and Erik Meijer. Monadic Parser Combinators. Technical Report NOTTCS-TR-96-4, Department of Computer Science, University of Nottingham, 1996.

Graham Hutton and Erik Meijer. Monadic Parsing in Haskell. *Journal of Functional Programming*, 8:437–444, July 1998.

IBM Corporation. C/C++ Language Extensions for Cell Broadband Engine Architecture, Version 2.5, 2007a.

IBM Corporation. Cell Broadband Engine Linux Reference Implementation Application Binary Interface Specification, Version 1.2, 2007b.

IBM Corporation. SPU Assembly Language Specification, Version 1.6, 2007c.

IBM Research. OpenCL Development Kit for Linux on Power. *http://www.alphaworks.ibm.com/tech/opencl*, 2009.

IBM Systems and Technology Group. *Cell Broadband Engine Programming Handbook - Version 1.1*, April 2007.

Jean D. Ichbiah, Bernd Krieg-Brueckner, Brian A. Wichmann, John G. P. Barnes, Olivier Roubine, and Jean-Claude Heliard. Rationale for the Design of the ADA Programming Language. *SIGPLAN Notices*, 14:1–261, June 1979.

INCITS/J3. *Final Draft International Standard for Fortran 2008*. ISO/IEC JTC1/SC22/WG5, April 2010.

Intel Corporation. *Intel Threading Building Blocks - Reference Manual*, 2010.

Ivano Primi. Numdiff. *http://www.nongnu.org/numdiff/*, 2010.

Kenneth Iverson. *A Programming Language*. John Wiley & Sons., 1962.

Simon Peyton Jones and John Hughes. The pretty package. *http://hackage.haskell.org/package/pretty*, 2011.

Alexandre Julliard. WineHQ. *http://www.winehq.org/*, 2012.

J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the Cell multiprocessor. *IBM Journal of Research and Development*, 49(4.5): 589–604, July 2005.

Prabhanjan Kambadur, Douglas Gregor, and Andrew Lumsdaine. OpenMP Extensions for Generic Libraries. In *OpenMP in a New Era of Parallelism*, volume 5004 of *Lecture Notes in Computer Science*, pages 123–133. Springer Berlin / Heidelberg, 2008.

S. Karlsson and M. Brorsson. A Free OpenMP Compiler and Run-Time Library Infrastructure for Research on Shared Memory Parallel Computing. In *Proceedings of The 16th IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS 2004)*. ACTA Press, November 2004.

Sven Karlsson. An Introduction to Balder - An OpenMP Run-time Library for Clusters of SMPs. In *OpenMP Shared Memory Parallel Programming*, volume 4315 of *Lecture Notes in Computer Science*, pages 78–91. Springer, 2008.

Sven Karlsson, Sung-Woo Lee, and Mats Brorsson. A Fully Compliant OpenMP Implementation on Software Distributed Shared Memory. In *High Performance Computing - HiPC 2002*, volume 2552 of *Lecture Notes in Computer Science*, pages 195–206. Springer, 2002.

Paul Keir. Towards an Implementation of OpenMP on the NVIDIA G80 Series Architecture. Master's thesis, Edinburgh Parallel Computing Centre, The University of Edinburgh, 2007.

Paul Keir, Paul W. Cockshott, and Andrew Richards. Mainstream Parallel Array Programming on Cell. In *Proceedings of the 5th Euro-Par Workshop on Highly Parallel Processing on a Chip (HPPC'11)*, volume 7155 of *Lecture Notes in Computer Science*, pages 260–269. Springer, 2011.

Ken Kennedy and John R. Allen. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002. ISBN 1-55860-286-0.

Ken Kennedy, Charles Koelbel, and Hans Zima. The Rise and Fall of High Performance Fortran: An Historical Object Lesson. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, HOPL III, pages 7–1–7–22. ACM, 2007.

Khronos OpenCL Working Group. *The OpenCL Specification, version 1.2*, November 2011.

Michael Klemm, Matthias Bezold, Ronald Veldema, and Michael Philippsen. JaMP: an implementation of OpenMP for a Java DSM. *Concurrency and Computation: Practice and Experience*, 19(18):2333–2352, 2007.

Kazuhiro Kusano, Shigehisa Satoh, and Mitsuhisa Sato. Performance Evaluation of the Omni OpenMP Compiler. In *Proceedings of the first International Workshop on OpenMP: Experiences and Implementations (WOMPEI)*, volume 1940 of *Lecture Notes in Computer Science*, pages 403–414. Springer, 2000.

Ralf Lämmel and Simon Peyton Jones. Scrap Your Boilerplate: A Practical Design Pattern for Generic Programming. In *Proceedings of the ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI 2003)*, pages 26–37. ACM Press, 2003.

Ralf Lämmel and Simon Peyton Jones. Scrap More Boilerplate: Reflection, Zips, and Generalised Casts. In *Proceedings of the Ninth ACM SIGPLAN International Conference on Functional Programming*, ICFP'04, pages 244–255. ACM, September 2004.

Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate with class: extensible generic functions. In *Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming*, ICFP'05, pages 204–215. ACM, September 2005.

Mark Harris Lars Nyland and Jan Prins. Fast N-Body Simulation with CUDA. In Hubert Nguyen, editor, *GPU Gems 3*, pages 677–694. Addison-Wesley Professional, 2007.

Chris Lattner. LLVM: An Infrastructure for Multi-Stage Optimization. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, December 2002. *See* `http://llvm.cs.uiuc.edu`.

Seyong Lee, Seung-Jai Min, and Rudolf Eigenmann. OpenMP to GPGPU: A Compiler Framework for Automatic Translation and Optimization. In *Proceedings of the 14th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, PPoPP '09, pages 101–110. ACM, 2009.

Daan Leijen and Erik Meijer. Parsec: Direct Style Monadic Parser Combinators For The Real World. *Department of Information and Computing Sciences Utrecht University Tech Rep UUCS200135*, (UU-CS-2001-35), 2001.

Chunhua Liao, Daniel J. Quinlan, Thomas Panas, and Bronis R. de Supinski. A ROSE-Based OpenMP 3.0 Research Compiler Supporting Multiple Runtime Libraries. In *Proceedings of the 6th International Workshop on OpenMP*, pages 15–28, 2010.

Sam Lindley. Implementing deterministic declarative concurrency using sieves. In *Proceedings of the ACM SIGPLAN Workshop on Declarative Aspects of Multicore Programming (DAMP)*, New York, NY, USA, 2007. ACM Press.

Stanley B. Lippman. *C++ Gems: Programming Pearls from The C++ Report*. Cambridge University Press, 1st edition, 1997. ISBN 0135705819.

Anton Lokhmotov, Alan Mycroft, and Andrew Richards. Delayed Side-Effects Ease Multicore Programming. In *Proceedings of the 13th European Conference on Parallel and Distributed Computing (Euro-Par)*, volume 4641 of *Lecture Notes in Computer Science*, pages 641–650. Springer, 2007.

Anton Lokhmotov, Alastair F. Donaldson, Alan Mycroft, and Colin Riley. Strict and Relaxed Sieving for Multi-Core Programming. In *Proceedings of the HiPEAC Workshop on Programmability Issues for Multi-Core Computers*, 2008.

Honghui Lu, Y. Charlie Hu, and Willy Zwaenepoel. OpenMP on Networks of Workstations. In *Proceedings of the 1998 ACM/IEEE conference on Supercomputing*, Supercomputing '98, pages 1–15. IEEE Computer Society, 1998.

Simon Marlow and Andy Gill. Happy, the Haskell Parser Generator. *http://www.haskell.org/happy/*, 2009.

John Mellor-Crummey, Laksono Adhianto, William N. Scherer, III, and Guohua Jin. A New Vision for Coarray Fortran. In *Proceedings of the Third Conference on Partitioned Global Address Space Programing Models*, PGAS '09, pages 5:1–5:9. ACM, 2009.

John Merlin. Techniques for the automatic parallelisation of 'Distributed Fortran 90'. Technical report, Department of Electronics and Computing Science, University of Southampton, November 1991.

John Merlin, Bryan Carpenter, and Tony Hey. shpf: a Subset High Performance Fortran compilation system. *Fortran Journal*, 8(2):2–6, 1996.

John H. Merlin. ADAPTing Fortran 90 Array Programs for Distributed Memory Architectures. In *Proceedings of the First International Conference of the Austrian Center for Parallel Computation*, pages 184–200. Springer-Verlag, 1992.

Message Passing Interface Forum. MPI: A Message-Passing Interface Standard. Version 2.2. Hardcover, September 2009.

Michael Metcalf and John Reid. *The F programming language*. Oxford University Press, Inc., New York, NY, USA, 1996. ISBN 0-19-850026-2.

Hans Meuer, Erich Strohmaier, Jack Dongarra, and Horst Simon. TOP500 Supercomputer Sites. *http://www.top500.org*.

Daniel Millot, Alain Muller, Christian Parrot, and Frédérique Silber-Chaussumier. STEP: A Distributed OpenMP for Coarse-Grain Parallelism Tool. In *Proceedings of the 4th International Conference on OpenMP*, volume 5004 of *Lecture Notes in Computer Science*, chapter 8, pages 83–99. Springer, 2008.

Gordon E. Moore. Cramming More Components onto Integrated Circuits. *Electronics*, 38 (8):114–117, April 1965.

Lenore Restifo Mullin. *A Mathematics of Arrays*. PhD thesis, Syracuse University, December 1988.

Jarek Nieplocha, Manojkumar Krishnan, Bruce Palmer, Vinod Tipparaju, and Jialin Ju. The global arrays user's manual, 2006.

Northrop Grumman Corporation. Count Lines of Code. *http://cloc.sourceforge.net/*, 2011.

Robert W. Numrich. F^{--} : A Parallel Extension to Cray Fortran. *Scientific Programming*, 6 (3):275–284, 1997.

Robert W. Numrich and Jon L. Steidel. F^{--}: A Simple Parallel Extension to Fortran 90. *SIAM News*, 30(7):1–8, September 1997.

NVIDIA Corporation. *CUDA C Programming Guide, Version 4.0*, May 2011.

Kevin O'Brien, Kathryn M. O'Brien, Zehra Sura, Tong Chen, and Tao Zhang. Supporting OpenMP on Cell. *International Journal of Parallel Programming*, 36(3):289–311, 2008.

Satoshi Ohshima, Shoichi Hirasawa, and Hiroki Honda. OMPCUDA : OpenMP Execution Framework for CUDA Based on Omni OpenMP Compiler. In *Proceedings of the 6th International Workshop on OpenMP*, volume 6132 of *Lecture Notes in Computer Science*, pages 161–173. Springer, 2010.

OpenMP Architecture Review Board. *OpenMP Application Program Interface, Version 3.1*, July 2011.

Bryan O'Sullivan, John Goerzen, and Don Stewart. *Real World Haskell*. O'Reilly Media, Inc., 1st edition, 2008. ISBN 0596514980.

Terence Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages*. The Pragmatic Bookshelf, 2007. ISBN 0978739256.

Howard A. Peelle. *Mathematical Computing in J: Volume 1 - Introduction*. Research Studies Press, 2004. ISBN 0863802818.

J. M. Perez, P. Bellens, R. M. Badia, and J. Labarta. CellSs: Making it easier to program the Cell Broadband Engine processor. *IBM Journal of Research and Development*, 51(5): 593–604, September 2007.

Simon Peyton-Jones, Roman Leshchinskiy, Gabriele Keller, and Manuel M. T. Chakravarty. Harnessing the Multicores: Nested Data Parallelism in Haskell. In *Foundations of Software Technology and Theoretical Computer Science*, 2008.

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

C. E. Rasmussen, M. J. Sottile, S. S. Shende, and A. D. Malony. Bridging the language gap in scientific computing: the Chasm approach. *Concurrency and Computation: Practice and Experience*, 18:151–162, February 2006.

John Reid. Coarrays in the next Fortran Standard. *SIGPLAN Fortran Forum*, 29:10–27, July 2010.

John C. Reynolds. *The Craft of Programming*. Prentice-Hall International, 1981. ISBN 0-13-188862-5.

George Russell, Paul Keir, Alastair F. Donaldson, Uwe Dolinsky, Andrew Richards, and Colin Riley. Programming Heterogeneous Multicore Systems using Threading Building Blocks. In *Proceedings of the 4th Euro-Par Workshop on Highly Parallel Processing on a Chip (HPPC'10)*, volume 6586 of *Lecture Notes in Computer Science*, pages 117–125. Springer, 2010.

Mitsuhisa Sato, Mitsuhisa Sato Shigehisa, Kazuhiro Kusano, and Yoshio Tanaka. Design of OpenMP Compiler for an SMP Cluster. In *The second European Workshop on OpenMP*, pages 32–39, 1999.

Jügen Sauermann. A parallel apl machine. *SIGAPL APL Quote Quad*, 20:342–347, May 1990.

Sven-Bodo Scholz. Single Assignment C: Efficient Support for High-level Array Operations in a Functional Setting. *Journal of Functional Programming*, 13(6):1005–1059, 2003.

Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, and Pat Hanrahan. Larrabee: A Many-Core x86 Architecture for Visual Computing. *ACM Transactions on Graphics*, 27:18:1–18:15, August 2008.

Lawrence Snyder. *A Programmer's Guide to ZPL*. The MIT Press, 1999. ISBN 0-262-69217-1.

Sony Computer Entertainment. SCE DevNet. *http://www.scedev.net/*.

Sony Corporation of America. MARS - Multicore Application Runtime System. *ftp://ftp.infradead.org/pub/Sony-PS3/mars/*, 2008.

P. Stpiczynski. Ada as a language for programming SMP clusters. *Annales UMCS, Informatica*, 3(1):73–79, 2003.

Kx Systems. Kx Systems - Fast database for real-time and historical data. *http://www.kx.com/*, 2012.

O. Takahashi, C. Adams, D. Ault, E. Behnen, O. Chiang, S.R. Cottier, P. Coulman, J. Culp, G. Gervais, M.S. Gray, Y. Itaka, C.J. Johnson, F. Kono, L. Maurice, K.W. McCullen, L. Nguyen, Y. Nishino, H. Noro, J. Pille, M. Riley, M. Shen, C. Takano, S. Tokito, T. Wagner, and H. Yoshihara. Migration of Cell Broadband Engine from 65nm SOI to 45nm SOI. In *2008 International Solid-State Circuits Conference - Digest of Technical Papers*, pages 86–87;597. IEEE, February 2008.

Andrew S. Tanenbaum. A Tutorial on Algol 68. *ACM Computing Surveys*, 8:155–190, June 1976. ISSN 0360-0300.

Christian Terboven, Dieter An Mey, Dirk Schmidl, and Marcus Wagner. First Experiences with Intel Cluster OpenMP. In *Proceedings of the 4th International Conference on OpenMP*, IWOMP'08, pages 48–59. Springer-Verlag, 2008.

The Austin Group. *Single UNIX Specification, Version 4*, 2010.

The Flex Project. flex: The Fast Lexical Analyzer. *http://www.gnu.org/software/flex/*, 2007.

The Parallel Computing Forum. PCF parallel Fortran extensions. *SIGPLAN Fortran Forum*, 10(3):1–57, 1991.

The Portland Group. *PGI®Visual Fortran User's Guide*, 2011.

Thinking Machines Corporation. CM Fortran Reference Manual, Version 1.0, February 1991.

Ronald Veldema, Thorsten Blass, and Michael Philippsen. Enabling multiple Accelerator Acceleration for Java/OpenMP. In *Proceedings of the 3rd USENIX Workshop on Hot Topics in Parallelism*, HotPar '11, pages 6–6. USENIX Association, 2011.

Loup Verlet. Computer "Experiments" on Classical Fluids. I. Thermodynamical Properties of Lennard-Jones Molecules. *Physical Review*, 159:98–103, July 1967.

Janus Voigtländer. Types for Programming and Reasoning, 2009.

Philip Wadler. Comprehending Monads. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, LFP '90, pages 61–78. ACM, 1990.

Haitao Wei and Junqing Yu. Loading OpenMP to Cell: An Effective Compiler Framework for Heterogeneous Multi-core Chip. In *Proceedings of the 3rd International Workshop on OpenMP*, volume 4935 of *Lecture Notes in Computer Science*, pages 129–133. Springer, 2008.

WG14. *Programming languages - C - Extensions to support embedded processors*. ISO/IEC JTC1/SC22, April 2006.

Phil Winterbottom. *ALEF Language Reference Manual*, 1993.

Niklaus Wirth. The Programming Language Pascal. *Acta Informatica*, 1:35–63, 1971.

XcalableMP Specification Working Group. *XcalableMP Language Specification Version 1.0*, November 2011.

Hans Zima, Peter Brezany, Barbara Chapman, Piyush Mehrotra, and Andreas Schwald. Vienna Fortran - A Language Specification, Version 1.1. Technical Report ACPC/TR 92-4, International Austrian Center for Parallel Computation, March 1992.