Compiling Parallel Functional Code with Data Parallel Idealised Algol

ROBERT ATKEY, University of Strathclyde
MICHEL STEUWER, University of Edinburgh
SAM LINDLEY, University of Edinburgh
CHRISTOPHE DUBACH, University of Edinburgh

Graphics Processing Units (GPUs) and other parallel devices are widely available and have the potential for accelerating a wide class of algorithms. However, expert programming skills are required to achieve maximum performance. These devices expose low-level hardware details through imperative programming interfaces which inevitably results in non-performance-portable programs highly tuned for a specific device.

Functional programming models have recently seen a renaissance in the systems community as they offer a solution to tackle the performance portability challenge. Recent work has shown that a functional representation can be effectively used to automatically derive and encode low-level hardware-specific parallelism strategies resulting in high performance across a range of target architectures. However, the translation of the functional representation to the actual imperative program expected by the hardware interface is typically ad hoc with no correctness guarantees.

In this paper, we present a formalised strategy-preserving translation from high level functional code to low level data race free parallel imperative code. This translation is formulated and proved correct within a language we call Data Parallel Idealised Algol (DPIA), a dialect of Reynolds' Idealised Algol. Performance results on GPUs and a multicore CPU show that the formalised translation process leads to performance on a par with ad hoc approaches.

1 INTRODUCTION

Modern parallel hardware such as Graphics Processing Units (GPUs) are difficult to program and optimise for. They require the use of imperative programming interfaces such as OpenMP, OpenCL, or CUDA, which expose low-level hardware details. Although these interfaces enable experts to squeeze the last bit of performance from the device, they keep most non-expert programmers at bay. Furthermore, software must be rewritten and tuned specifically for each new generation of device, leading to performance portability problems.

Recent years have seen an accelerating trend towards high level *functional* programming models for expressing parallel computation. The absence of side-effects, the use of higher order functions, and the compositionality of functional languages makes them particularly attractive for expressing parallel operations using primitives such as **map** and **reduce**. Nesl (Blelloch 1993), CopperHead (Catanzaro et al. 2011), LiquidMetal (Dubach et al. 2012), Accelerate (McDonell et al. 2013), and Delite (Sujeeth et al. 2014) are examples of such functional approaches.

While these approaches offer a high level programming abstraction, they struggle to deliver high performance across the wide range of currently available parallel hardware or even across different GPUs. Ultimately, functional abstractions must be translated into imperative code, as dictated by contemporary parallel hardware interfaces. However, there is a distinct mismatch between high-level functional abstractions and low-level hardware primitives. Current approaches rely on ad hoc techniques that produce imperative parallel code typically using mostly templated code written specifically for each device or other ad hoc code generation strategies. While these approaches work well for a fixed device, they often fail to deliver high performance when targeting different types of device due to the limited ability to explore the optimisation space.

Steuwer et al. (2015) showed that it is possible to use functional abstractions to represent low-level device-specific optimisations. Their approach uses a rewrite system that encodes semantics preserving transformations

at the functional level. The rewrite rules are classified into hardware agnostic, high level rules and lowering rules that transform high level primitives into hardware specific functional primitives. An automated search discovers a sequence of rewrites that yields high performance code. However, the last step of their process, *the conversion of functional into imperative code*, is still performed in an ad hoc manner with no correctness guarantees.

In this paper, we formalise the semantics preserving translation from functional to deterministic parallel imperative code using *Data Parallel Idealised Algol* (DPIA), a novel variant of Reynolds' *Idealised Algol* (Reynolds 1997). Idealised Algol orthogonally combines typed functional programming with imperative programming, a property that DPIA inherits. This allows us to start in the purely functional subset with a high level specification of the computation we wish to perform, and then to systematically rewrite it to "purely imperative" code (Section 4.1) that has a straightforward translation to the C-like OpenCL code required by GPUs (Section 4.3). DPIA incorporates a substructural type system, incorporating ideas from Reynolds' *Syntactic Control of Inteference* (O'Hearn et al. 1999; Reynolds 1978), which ensures that the generated programs are data race free. We show that our translation from functional to imperative is correct (Section 5). Our experimental results in Section 7 show that our approach delivers high quality code with performance on a par with hand-tuned OpenCL code on several GPUs, along with a correctness guarantee.

Data Parallel Idealised Algol provides a greater benefit than just a correctness proof. Reynolds' Syntactic Control of Interference discipline provides a framework in which to explore the boundaries between functional and imperative parallel programming. We demonstrate the flexibility of this approach in Section 6 where we augment the basic design of DPIA with OpenCL specific primitives that describe how we use the parallelism and memory hierarchies of OpenCL, conveying this usage information through the translation to imperative code, whilst retaining the correctness and data race freedom properties.

Contributions. This paper makes three major contributions:

(1) We describe a formal translation from high level functional code annotated with parallelism strategies to parallel imperative code. We describe this translation fully in Section 4 and prove it correct in Section 5.
(2) In order to formulate this translation, we introduce a variant of Reynolds' Idealised Algol, called *Data Parallel Idealised Algol* (DPIA). This is an extension of (O'Hearn et al. 1999)'s *Syntactic Control of Inteference Revisited* with indexed types, array and tuple data types, and primitives for data parallel programming. These extensions are essential for data parallel imperative programs. We describe DPIA in Section 3.
(3) We specialise our framework to OpenCL, demonstrating how to incorporate the parallelism and memory hierarchy requirements of OpenCL into our methodology (Section 6). Our experimental results in Section 7 demonstrate that our formalised translation yields efficient parallel OpenCL code with no overhead.

In the next section we introduce and motivate our approach and the use of Idealised Algol as a foundation. After the technical body of the paper in Sections 3-7, we discuss related work in Section 8 and our conclusions and future work in Section 9.

2 PARALLELISM STRATEGY PRESERVING COMPILATION

We describe a compilation method that takes high level functional array code to low level parallel imperative code. Our approach is characterised by (a) expression of parallelisation strategies in high level functional code, where their semantic equivalence to the original code can be easily checked, and new strategies readily derived; and (b) a predictable and principled translation to low level imperative code that preserves parallelisation strategies.

2.1 Expressing Parallelisation Strategies in Functional Code

Here is an expression that describes the dot product of two vectors xs and ys:

$$\mathbf{reduce}\ (+)\ 0\ (\mathbf{map}\ (\lambda x.\ \mathbf{fst}\ x * \mathbf{snd}\ x)\ (\mathbf{zip}\ xs\ ys)) \tag{1}$$

This expression can be read in two ways. Firstly, read mathematically, it is a declarative specification of the dot product. Secondly, it can be read as a strategy for computing dot products. Reading right-to-left, we have a pipeline arrangement. Let us make the following assumptions: *i)* **zip** is not materialised (it only affects how later parts of the pipeline read their input); *ii)* **map** is executed in parallel across the array; and *iii)* **reduce** is executed sequentially. Then we can read this expression as embodying a naive "parallel map, sequential reduce" strategy.

Such a naive strategy is not always best. If we try to execute one parallel job per element of the input arrays, then depending on the underlying architecture we will either fail (e.g., on GPUs with a fixed number of execution units), or generate so many threads that coordination of them will dominate the runtime (e.g., on CPUs). The overall strategy of "parallel, then sequential" is likely not the most efficient, either.

We can give a more refined strategy given information about the underlying architecture. For instance, GPUs support nesting of parallelism by organising threads into groups, or *work-items* into *work-groups*, using OpenCL terminology. If we know that the input is of size $n \times 128 \times 2048$, we can explicitly control how parallelism can be mapped to the GPU hierarchy. The following expression distributes the work among n groups of 128 local threads, each processing 2048 elements in one go, by directly reducing over the multiplied pairs of elements:

$$\begin{aligned}&\textbf{reduce}\ (+)\ 0\ (\textbf{join}\ (\textbf{mapWorkgroup}\ (\lambda zs_1.\ \textbf{mapLocal}\ (\lambda zs_2.\ \textbf{reduce}\ (\lambda x\ a.\ (\textbf{fst}\ x * \textbf{snd}\ x) + a)\ 0\ (\textbf{split}\ 2048\ zs_2))\ zs_1)\\ &\qquad\qquad (\textbf{split}\ (2048 * 128)\ (\textbf{zip}\ xs\ ys)))))\end{aligned} \tag{2}$$

Although this expression gives much more information about how to process the computation on the GPU, we have not left the functional paradigm, so we still have access to the straightforward mathematical reading of this expression. We can use equational reasoning to prove that this is semantically equivalent to (1). Equational reasoning can also be used to generate (2) from (1). Indeed Steuwer et al. (2015) have shown that stochastic search techniques are effective at automatically discovering parallelisation strategies that match hand-coded ones.

However, even with a specified parallelisation strategy we cannot execute this code directly. We need to translate the functional code to an imperative language like OpenCL or CUDA in a way that preserves our chosen strategy. This paper presents a formal approach to solving this translation problem.

2.2 Strategy Preserving Translation to Imperative Code

What is the simplest way of converting a functional program to an imperative one? Starting with our zip-map-reduce formulation of dot-product (1), we can turn it into an imperative program simply by assigning its result to an output variable *out*:

$$out := \textbf{reduce}\ (+)\ 0\ (\textbf{map}\ (\lambda x.\ \textbf{fst}\ x * \textbf{snd}\ x)\ (\textbf{zip}\ xs\ ys))$$

Unfortunately, this is not suitable for compilation targets like OpenCL or CUDA. While assignment statements are the bread-and-butter of such languages, their expression languages certainly do not include such modern amenities as higher order **map** and **reduce** functions. To translate these away, we introduce a novel *acceptor-passing* translation $\mathcal{A}(\!|E|\!)_\delta(out)$. The key idea is that for any expression E producing data of type δ, the translation $\mathcal{A}(\!|E|\!)_\delta(out)$ is an imperative program that has the same effect as the assignment $out := E$ and is free from higher-order combinators. This translation is mutually defined with a continuation passing translation $\mathcal{C}(\!|E|\!)_\delta(C)$ that takes a parameterised command C that will consume the output, instead of taking an output variable.

The definition of the translation is given in Section 4.1. We introduce it here by example. Applied to our dot-product code, our translation first replaces the **reduce** by a corresponding imperative combinator **reduceI**. We will see below that **reduceI** is straightforwardly implemented in terms of variable allocation and a for-loop.

$$\begin{aligned}&\mathcal{A}(\!|\textbf{reduce}\ (+)\ 0\ (\textbf{map}\ (\lambda x.\ \textbf{fst}\ x * \textbf{snd}\ x)\ (\textbf{zip}\ xs\ ys))|\!)_{\mathsf{num}}(out)\\ =\ &\mathcal{C}(\!|\textbf{map}\ (\lambda x.\ \textbf{fst}\ x * \textbf{snd}\ x)\ (\textbf{zip}\ xs\ ys)|\!)_{n.\mathsf{num}}(\lambda x.\\ &\quad \mathcal{C}(\!|0|\!)_{\mathsf{num}}(\lambda y.\ \textbf{reduceI}\ n\ (\lambda x\ y\ o.\ \mathcal{A}(\!|x + y|\!)_{\mathsf{num}}(o))\ y\ x\ (\lambda r.\ \mathcal{A}(\!|r|\!)(A))))\end{aligned}$$

The **map** is now translated, by the continuation passing translation, into allocation of a temporary array and an imperative **mapI** combinator. As with **reduceI**, the **mapI** combinator is straightforwardly implementable in

terms of a (parallel) for-loop. The operator **new** δ *ident* declares a new storage cell of type δ named *ident*, where storage cells are represented as pairs of an acceptor (i.e., "writer", "l-value") part *ident*.1 and an expression (i.e. "reader", "r-value") part *ident*.2. Our language, which we introduce in Section 3, is a variant of Reynolds' Idealised Algol (Reynolds 1978).

$$\begin{array}{rl} = & \textbf{new}\ (n.\mathsf{num})\ (\lambda tmp.\ C(\!|\textbf{zip}\ xs\ ys|\!)_{n.(\mathsf{num}\times\mathsf{num})}(\lambda x.\ \textbf{mapl}\ n\ (\lambda x\ o.\ \mathcal{A}(\!|\textbf{fst}\ x * \textbf{snd}\ x|\!)_{\mathsf{num}}(o))\ x\ tmp.1); \\ & \qquad C(\!|0|\!)_{\mathsf{num}}(\lambda y.\ \textbf{reducel}\ n\ (\lambda x\ y\ o.\ \mathcal{A}(\!|x+y|\!)_{\mathsf{num}}(o))\ y\ tmp.2\ (\lambda r.\ \mathcal{A}(\!|r|\!)(A)))) \end{array}$$

Readers familiar with other translations of data parallel functional programs into imperative loops may be surprised at the allocation of a temporary array here. Typically, the compilation process would be expected to automatically fuse the computation of the **map** into the translation of the **reduce**. However, this is precisely what we do *not* want from a predictable compilation process for parallelism. If fusion is desired, it is carried out before this translation is applied and directly encoded in the functional program, as seen earlier in example (2). The parallelism strategy described by the functional code here precisely states "parallel map, followed by sequential reduce". Predictability of the translation is essential for more complex parallelism strategies that exploit parallelism hierarchies and even different memory address spaces as we will see later in Section 6.2.

Continuing the translation process, we substitute *out*, the arithmetic expressions and the **zip**, leaving two uses of the "intermediate-level" combinators **mapl** and **reducel**:

$$\begin{array}{rl} = & \textbf{new}\ (n.\mathsf{num})\ (\lambda tmp.\ \textbf{mapl}\ n\ (\lambda x\ o.\ o := \textbf{fst}\ x * \textbf{snd}\ x)\ (\textbf{zip}\ xs\ ys)\ tmp.1; \\ & \qquad \textbf{reducel}\ n\ (\lambda x\ y\ o.\ o := x + y)\ 0\ tmp.2\ (\lambda r.\ out := r)) \end{array}$$

These combinators are now replaced by parallel and sequential for-loops, which we describe in a further translation stage in Section 4.2.

$$\begin{array}{rl} = & \textbf{new}\ (n.\mathsf{num})\ (\lambda tmp.\ \textbf{parfor}\ n\ tmp.1\ (\lambda i\ o.\ o := \textbf{fst}\ (\textbf{idx}\ (\textbf{zip}\ xs\ ys)\ i) * \textbf{snd}\ (\textbf{idx}\ (\textbf{zip}\ xs\ ys)\ i)); \\ & \qquad \textbf{new}\ \mathsf{num}\ (\lambda accum.\ accum.1 := 0; \\ & \qquad\qquad \textbf{for}\ n\ (\lambda i.\ accum.1 := accum.2 + \textbf{idx}\ tmp.2\ i); \\ & \qquad\qquad out := accum.2)) \end{array} \tag{3}$$

The sequential **for** loops of our intermediate language are standard; **for** n $(\lambda i.\ b)$ executes the body b n times with iteration counter i. Parallel **parfor** loops are slightly more complex due to the way they explicitly take a parameter (here named *tmp*.1) that describes where to place the results of each iteration in a data-race free way. We describe this fully in Section 3.3.

We are now left with an imperative program, albeit with a non-standard parallel-for construct and complex data access expressions involving **fst**, **zip**, **idx** and so on. Our final translation to pseudo-C (Section 4.3) resolves these data layout expressions into explicit indexing computations:

```
1 float tmp[N];
2 parfor (int i = 0; i < N; i += 1)
3   tmp[i] = xs[i] * ys[i];
4 float accum = 0.0;
5 for (int i = 0; i < N; i += 1)
6   accum = accum + tmp[i];
7 output = accum;
```

This resulting low level imperative code precisely implements the strategy "parallel map, followed by sequential reduce" described by our original functional expression (1).

Our original dot-product code does not produce particularly complex code, but our translation method scales to more detailed parallelism strategies. The alternative dot-product code in (2), which rearranges the **map** and **reduce** combinators in order to better exploit parallel hardware, yields the following code:

```
float tmp[N/2048];
parfor (int i = 0; i < N/(2048*128); i += 1) {
  parfor (int j = 0; j < 128; j += 1) {
    float accum = 0.0;
    for (int k = 0; k < 2048; k += 1) {
      accum = (xs[(2048*128 * i) + (128 * j) + k] * ys[(2048*128 * i) + (128 * j) + k])
            + accum; }
    tmp[((128 * i) + j)] = accum;
  }
}
float accum = 0.0;
for (int i = 0; i < N/2048; i += 1) {
  accum = accum + tmp[i];
}
output = accum;
```

As we shall see in Section 7, given a target-architecture optimised parallelisation strategy defined in functional code, our translation process produces OpenCL code with performance on a par with previous ad hoc code generators, and with hand written code.

Key to our translation methodology is a single intermediate language that can express pure functional expressions and deterministic race free parallel imperative programs, and which is amenable to formal reasoning. In the next section, we describe our language for this task, DPIA: *Data Parallel Idealised Algol.*

3 DATA PARALLEL IDEALISED ALGOL

Our intermediate language for code generation is an extension of Reynolds' *Idealised Algol* (Reynolds 1997). Idealised Algol is the orthogonal combination of typed λ-calculus and imperative programming. To support deterministic race free parallel programming, we use the "Syntactic Control of Interference Revisited" variant of Idealised Algol (O'Hearn et al. 1999), extended with size-indexed array and tuple data types. We call our language *Data Parallel Idealised Algol* (DPIA). We saw examples of DPIA in the previous section. We now highlight the major features of DPIA, and then give the formal presentation of its types (Section 3.1), type system (Section 3.2), and data parallel programming primitives (Section 3.3). We discuss the formal semantics of DPIA in Section 5.

Orthogonal Combination of Typed λ-Calculus and Imperative Programming. The central feature of the Idealised Algol family of languages is the orthogonal combination of an imperative language with a typed λ-calculus. Unlike traditional functional languages like Scheme, Haskell and ML, β-reduction is not the main engine of computation; for example, there is no pattern matching. The purpose of λ-abstraction is to add a facility for procedural abstraction to a base imperative language. In the absence of recursion in the λ-calculus component of the language, given a whole program it is possible to normalise away all λ-abstractions to yield a "pure" imperative program. The λ-calculus therefore becomes almost a meta-language for constructing imperative programs. We will exploit this feature during our translation process in Section 4.

Substructural Types for Interference Control. Reynolds (1978) notes that the introduction of a procedure facility into an imperative programming language destroys a nice property of imperative programs without procedures: that distinct identifiers always refer to distinct parts of the store. Such aliasing complicates reasoning, especially the reasoning required to show that running code in parallel is deterministic and data race free. To re-enable straightforward reasoning in the presence of procedures, Reynolds introduced a discipline, *Syntactic Control of Interference* (SCI), that ensures that distinct identifiers never interfere. This is more subtle than it may first appear, due to identifiers that are used *passively* (essentially read-only), which are allowed to alias. We build upon the *Syntactic Control of Interference Revisited* (SCIR) system introduced by O'Hearn et al. (1999), which presents

Reynolds ideas as a substructural typed λ-calculus (see Figure 3). Type based interference control ensures, by construction, that we have sufficient information to guarantee data race freedom in our generated code.

Primitives for Data Parallel Programming. Idealised Algol, with or without Syntactic Control of Interference, has primarily been studied theoretically as a core calculus combining imperative programming with λ-abstraction (with the notable exception of Ghica's use of SCI for hardware synthesis (Ghica 2007)). To use SCIR as an intermediate language for data parallel computation, we extend it with compound data types, tuples and arrays, and with indexed types to account for array size polymorphism and data type polymorphism. We also extend SCIR with primitives designed for data parallel programming. The central primitive is a data race free parallel for-loop primitive, which we describe in Section 3.3.

3.1 The Types of DPIA

The type system of DPIA, following Idealised Algol, separates *data types*, which classify data (integers, floats, arrays, *etc.*), from *phrase types*, which classify the parts of a program according to the interface they offer. Phrase types are a generalisation to first-class status of the syntactic categories in a standard imperative language that distinguish between expressions (r-values), l-values, and statements. Phrase types in DPIA comprise *expressions*, which produce data, possibly reading from the store; *acceptors*, which describe modifiable areas of the store (analogous to *l-values* in imperative languages (Strachey 2000)); *commands*, which modify the store; *functions*, which are parameterised phrases; and *records*, which offer a choice of multiple phrases. The separation into data and phrase types distinguishes Idealised Algol-style type systems from those for functional languages, which commonly use expression phrases for everything (permitting, for example, functional data).

To facilitate interference control, we identify a subset of phrase types which are *passive* (Section 3.1.2), i.e. essentially read-only, and so are safe to share across parallel threads. (We elaborate on what "essentially read-only" means in Section 3.1.2 and Section 3.2.)

3.1.1 Kinding rules. We extend SCIR with both data type and size polymorphism, so we need a kind system. Figure 1 presents the kinding rules for DPIA types. The kinds κ of DPIA include the major classifications into data types (data) and phrase types (phrase), along with the kind of type-level natural numbers (nat). Types may contain variables, so we use a kinding judgement $\Delta \vdash \tau : \kappa$, which states that type τ has kind κ in kinding context Δ. Figure 1b gives the variable rule that permits the use of type variables in well-kinded types. Figure 1d presents the rules for type-level natural numbers: either constants $\underline{n}$, addition $I + J$, or multiplication IJ (where I and J range over terms of kind nat).

The rules for data types are presented in Figure 1e. We use δ to range over data types. The base types are num for numbers; and a data type of array indexes $\mathbf{idx}(n)$, parameterised by the maximum array index. There are two compound types of data. For any data type δ and natural number term I, $I.\delta$ is the data type of homogeneous arrays of δs of size I. (We opt for a concise notation for array types as they are pervasive in data parallel programming.) Heterogeneous compound data types (records) are built using the rule for $\delta_1 \times \delta_2$.

The phrase types of DPIA are given in Figure 1f. We use θ to range over phrase types. For each data type δ, there are phrase types $\mathsf{exp}[\delta]$ for *expression* phrases that produce data of type δ, and $\mathsf{acc}[\delta]$ for *acceptor* phrases that consume data of type δ. The comm phrase type classifies *command* phrases that may modify the store. Phrases that can be used in two different ways, θ_1 or θ_2, are classified using the phrase product type $\theta_1 \times \theta_2$. This type is distinct from the *data* product type $\delta_1 \times \delta_2$: the data type represents a pair of data values; the phrase type represents an "interface" that offers two possible "methods". (For readers familiar with Linear Logic (Girard 1987), the phrase product is like "with" (&) and the data product like "tensor" ($\otimes$).) The final three phrase types are all variants of parameterised phrase types. The phrase types $\theta_1 \to \theta_2$ and $\theta_1 \to_{\mathrm{p}} \theta_2$ classify phrase functions. The p subscript denotes passive functions. The phrase type $(x{:}\kappa) \to \theta$ classifies a phrase that is parameterised either by a data type or a natural number.

$$\kappa ::= \text{data} \mid \text{phrase} \mid \text{nat}$$

(a) Kinds

$$\frac{x : \kappa \in \Delta}{\Delta \vdash x : \kappa}$$

(b) Kinding Structural Rules

$$\frac{\models \forall \sigma : dom(\Delta) \to \mathbb{N}.\sigma(I) = \sigma(J)}{\Delta \vdash I \equiv J : \text{nat}}$$

(c) Type Equality

$$\frac{}{\Delta \vdash \underline{n} : \text{nat}} \qquad \frac{\Delta \vdash I : \text{nat} \qquad \Delta \vdash J : \text{nat}}{\Delta \vdash I + J : \text{nat}} \qquad \frac{\Delta \vdash I : \text{nat} \qquad \Delta \vdash J : \text{nat}}{\Delta \vdash IJ : \text{nat}}$$

(d) Natural numbers

$$\frac{}{\Delta \vdash \text{num} : \text{data}} \qquad \frac{\Delta \vdash I : \text{nat}}{\Delta \vdash \mathbf{idx}(I) : \text{data}} \qquad \frac{\Delta \vdash I : \text{nat} \quad \Delta \vdash \delta : \text{data}}{\Delta \vdash I.\delta : \text{data}} \qquad \frac{\Delta \vdash \delta_1 : \text{data} \quad \Delta \vdash \delta_2 : \text{data}}{\Delta \vdash \delta_1 \times \delta_2 : \text{data}}$$

(e) Data Types

$$\frac{\Delta \vdash \delta : \text{data}}{\Delta \vdash \text{exp}[\delta] : \text{phrase}} \qquad \frac{\Delta \vdash \delta : \text{data}}{\Delta \vdash \text{acc}[\delta] : \text{phrase}} \qquad \frac{}{\Delta \vdash \text{comm} : \text{phrase}} \qquad \frac{\Delta \vdash \theta_1 : \text{phrase} \qquad \Delta \vdash \theta_2 : \text{phrase}}{\Delta \vdash \theta_1 \times \theta_2 : \text{phrase}}$$

$$\frac{\Delta \vdash \theta_1 : \text{phrase} \quad \Delta \vdash \theta_2 : \text{phrase}}{\Delta \vdash \theta_1 \to \theta_2 : \text{phrase}} \qquad \frac{\Delta \vdash \theta_1 : \text{phrase} \quad \Delta \vdash \theta_2 : \text{phrase}}{\Delta \vdash \theta_1 \to_{\text{p}} \theta_2 : \text{phrase}} \qquad \frac{\Delta, x : \kappa \vdash \theta : \text{phrase} \qquad \kappa \in \{\text{nat}, \text{data}\}}{\Delta \vdash (x{:}\kappa) \to \theta : \text{phrase}}$$

(f) Phrase Types

Fig. 1. Well-formed Types

The types of DPIA include arithmetic expressions, so we have a non trivial notion of equality between types, written $\Delta \vdash \tau_1 \equiv \tau_2 : \kappa$. The key type equality rule is given in Figure 1c: two arithmetic expressions are equal if they are equal as natural numbers for all interpretations (σ) of their free variables. This equality is lifted to all other types by structural congruence.

3.1.2 Passive Types. Figure 2 identifies the subset of phrase types that classify passive phrases. The opposite of passive is *active*. We use ϕ to range over passive phrase types. An expression phrase type $\text{exp}[\delta]$ is always passive — phrases of this type can, by definition, only read the store. A compound phrase type is always passive if its component phrase types are all passive. Furthermore, a passive function type $\theta_1 \to_{\text{p}} \theta_2$ is always passive, and a plain function type is passive whenever its return type is passive (irrespective of the argument type).

Passive types are essentially read-only. The one exception whereby a phrase of passive type may modify the store is a passive function with active argument and return types. Such a function can only modify the part of the store addressable through the active phrase it is supplied with as an argument.

3.2 Typing Rules for DPIA

The typing judgement of DPIA follows the SCIR system of O'Hearn et al. (1999) in distinguishing between passive and active uses of identifiers. Our judgement also has a kinding context for size and data type polymorphism. The judgement form has the following structure:

$$\Delta \mid \Pi; \Gamma \vdash P : \theta$$

$$\frac{\Delta \vdash \delta : \text{data}}{\Delta \vdash \exp[\delta] : \text{passive}} \qquad \frac{\Delta \vdash \phi_1 : \text{passive} \qquad \Delta \vdash \phi_2 : \text{passive}}{\Delta \vdash \phi_1 \times \phi_2 : \text{passive}} \qquad \frac{\Delta \vdash \theta : \text{phrase} \qquad \Delta \vdash \phi : \text{passive}}{\Delta \vdash \theta \to \phi : \text{passive}}$$

$$\frac{\Delta \vdash \theta_1 : \text{phrase} \qquad \Delta \vdash \theta_2 : \text{phrase}}{\Delta \vdash \theta_1 \to_{\text{p}} \theta_2 : \text{passive}} \qquad \frac{\Delta, x : \kappa \vdash \theta : \text{passive} \qquad \kappa \in \{\text{nat}, \text{data}\}}{\Delta \vdash (x{:}\kappa) \to \theta : \text{passive}}$$

Fig. 2. Passive Types

where Δ is the kinding context, Π is a context of passively used identifiers, Γ is a context of actively used identifiers, P is a program phrase, and θ is a phrase type. All the types in Π and Γ are phrase types well-kinded by Δ. The phrase type θ must also be well-kinded by Δ. The order of entries does not matter in any of the contexts. The contexts Δ and Π are subject to contraction and weakening; context Γ is not.

The split context formulation of SCIR recalls that of Barber's DILL system (Barber 1996), which also distinguishes between linear and unrestricted assumptions. The SCIR system differs in how movement between the zones is mediated in terms of passive and active types. Section 2.6 of (O'Hearn et al. 1999) discusses the relationship between SCIR and Linear Logic.

The core typing rules of DPIA are given in Figure 3. These rules define how variable phrases are formed, how parameterised and compound phrases are introduced and eliminated, and how passive and active types are managed. Any particular application of DPIA is specified by giving a collection of primitive phrases Primitives, each of which has a closed phrase type. We describe a collection for data parallel programming in Section 3.3.

Figure 3a presents the rule for forming variable phrases, implicit conversion between equal types, and the use of primitives. At point of use, all variables are considered to be used actively. If the final phrase type is passive, then an active use may be converted to a passive one by the Passify rule. Primitives may be used in any context. Figure 3b presents the rules for parameterised and compound phrases. These are all standard typed λ-calculus style rules, except the use of separate contexts for a function and its arguments in the App rule. This ensures that every function and its argument use non-interfering active resources, maintaining the invariant that distinct identifiers refer to non-interfering phrases. Note that we do not require separate contexts for the two parts of a compound phrase in the Pair rule. Compound phrases offer two ways of interacting with the *same* underlying resource (as in the with (&) rule from linear logic).

Figure 3c describes how passive and active uses of variables are managed. The Activate rule allows any variable that has been used passively to be treated as if it were used actively. The Passify rule allows active uses to be treated as passive, as long as the final phrase type is passive. The Promote rule turns functions into passive functions, as long as they do not contain any free variables used actively. The Derelict rule indicates that a passive function can always be seen as a normal function, if required.

DPIA*'s functional sub-language.* By inspection of the rules, we can see that if we restrict to phrase types constructed from $\exp[\delta]$, functions, polymorphic functions, and tuples, then the constraints on multiple uses of variables in DPIA cease to apply. Therefore, DPIA has a sub-language that has the same type system as a normal (non-substructural) typed λ-calculus with base types for numbers, arrays and tuples, and a limited form of polymorphism. When we introduce the functional primitives for DPIA in the next section, we will enrich this λ-calculus with arithmetic, array manipulators, and higher-order array combinators. It is this purely functional sub-language of DPIA that allows us to embed functional data parallel programs in a semantics preserving way.

$$\frac{}{\Delta \mid \Pi; \Gamma, x : \theta, \Gamma' \vdash x : \theta}\ \textsc{Var} \qquad \frac{\Delta \mid \Pi; \Gamma \vdash P : \theta_1 \quad \Delta \vdash \theta_1 \equiv \theta_2 : \mathsf{phrase}}{\Delta \mid \Pi; \Gamma \vdash P : \theta_2}\ \textsc{Conv} \qquad \frac{\mathsf{prim} : \theta \in \textsc{Primitives}}{\Delta \mid \Pi; \Gamma \vdash \mathsf{prim} : \theta}\ \textsc{Prim}$$

(a) Structural Rules

$$\frac{\Delta \mid \Pi; \Gamma, x : \theta_1 \vdash P : \theta_2}{\Delta \mid \Pi; \Gamma \vdash \lambda x.P : \theta_1 \to \theta_2}\ \textsc{Lam} \qquad \frac{\Delta \mid \Pi; \Gamma_1 \vdash P : \theta_1 \to \theta_2 \qquad \Delta \mid \Pi; \Gamma_2 \vdash Q : \theta_1}{\Delta \mid \Pi; \Gamma_1, \Gamma_2 \vdash P\,Q : \theta_2}\ \textsc{App}$$

$$\frac{\Delta, x : \kappa \mid \Pi; \Gamma \vdash P : \theta \qquad x \notin fv(\Pi, \Gamma)}{\Delta \mid \Pi; \Gamma \vdash \Lambda x.P : (x{:}\kappa) \to \theta}\ \textsc{TLam} \qquad \frac{\Delta \mid \Pi; \Gamma \vdash P : (x{:}\kappa) \to \theta \qquad \Delta \vdash e : \kappa}{\Delta \mid \Pi; \Gamma \vdash P\,e : \theta[e/x]}\ \textsc{TApp}$$

$$\frac{\Delta \mid \Pi; \Gamma \vdash P : \theta_1 \qquad \Delta \mid \Pi; \Gamma \vdash Q : \theta_2}{\Delta \mid \Pi; \Gamma \vdash \langle P, Q \rangle : \theta_1 \times \theta_2}\ \textsc{Pair} \qquad \frac{\Delta \mid \Pi; \Gamma \vdash P : \theta_1 \times \theta_2}{\Delta \mid \Pi; \Gamma \vdash P.i : \theta_i}\ \textsc{Proj}$$

(b) Introduction and Elimination Rules

$$\frac{\Delta \mid \Pi, x : \theta; \Gamma \vdash P : \theta'}{\Delta \mid \Pi; \Gamma, x : \theta \vdash P : \theta'}\ \textsc{Activate} \qquad \frac{\Delta \mid \Pi; \Gamma, x : \theta \vdash P : \phi}{\Delta \mid \Pi, x : \theta; \Gamma \vdash P : \phi}\ \textsc{Passify}$$

$$\frac{\Delta \mid \Pi; \cdot \vdash P : \theta_1 \to \theta_2}{\Delta \mid \Pi; \cdot \vdash P : \theta_1 \to_{\mathrm{p}} \theta_2}\ \textsc{Promote} \qquad \frac{\Delta \mid \Pi; \Gamma \vdash P : \theta_1 \to_{\mathrm{p}} \theta_2}{\Delta \mid \Pi; \Gamma \vdash P : \theta_1 \to \theta_2}\ \textsc{Derelict}$$

(c) Active and Passive Phrase Rules

Fig. 3. Typing Rules: Indexed Affine Linear λ-Calculus with Passivity (O'Hearn et al. 1999)

3.3 Data Parallel Programming Primitives

Section 3.2 has described a general framework for a language with interference control. We now instantiate this framework with typed primitive operations for data parallel programming, outlined in Figure 4. Our primitives fall into two principal categories: high-level functional primitives, and low-level imperative primitives. Programs that are the input to our translation process are composed of the high-level functional primitives. These programs contain uses of **map** and **reduce** that have no counterpart in low-level languages for data-parallel computation. Our translation process converts these into low-level combinators (Section 4.1). A final lowering translation removes all functional primitives except arithmetic (Section 4.3).

As primitives are treated specially by the Prim rule they can (with the aid of a little η-expansion) always be promoted to be passive. Thus, it is never necessary to annotate the arrows of a first-order primitive with a p subscript. The only such annotations that are necessary are those final arrows of function types occurring inside the type of a higher-order primitive that is required to be passive (in our case, only **parfor** and **mapI**).

Functional Primitives. Figure 4a lists the type signatures of the primitives used for constructing purely functional data parallel programs. These fit into three groups. The first group consists of numeric literals ($\underline{n}$) and first-order operations on scalars (**negate**, $(+)$, $(-)$, $(*)$, $(/)$). The second group contains the two key higher-order functional combinators for constructing array processing programs: **map** and **reduce**. These have (the Idealised Algol renditions of) the standard types for these primitives, extended with size information. The third group comprises functions for manipulating data layouts: **zip** joins two arrays of equal length into an array of pairs, **split** breaks a one dimensional array into a two dimensional array and **join** flattens a two dimensional array into a one

$\underline{n}$: $\mathsf{exp}[\mathsf{num}]$
negate : $\mathsf{exp}[\mathsf{num}] \to \mathsf{exp}[\mathsf{num}]$
$(+, *, /, -)$: $\mathsf{exp}[\mathsf{num}] \times \mathsf{exp}[\mathsf{num}] \to \mathsf{exp}[\mathsf{num}]$

map : $(n : \mathsf{nat}) \to (\delta_1\ \delta_2 : \mathsf{data}) \to (\mathsf{exp}[\delta_1] \to \mathsf{exp}[\delta_2]) \to \mathsf{exp}[n.\delta_1] \to \mathsf{exp}[n.\delta_2]$
reduce : $(n : \mathsf{nat}) \to (\delta_1\ \delta_2 : \mathsf{data}) \to (\mathsf{exp}[\delta_1] \to \mathsf{exp}[\delta_2] \to \mathsf{exp}[\delta_2]) \to \mathsf{exp}[\delta_2] \to \mathsf{exp}[n.\delta_1] \to \mathsf{exp}[\delta_2]$

zip : $(n : \mathsf{nat}) \to (\delta_1\ \delta_2 : \mathsf{data}) \to \mathsf{exp}[n.\delta_1] \to \mathsf{exp}[n.\delta_2] \to \mathsf{exp}[n.(\delta_1 \times \delta_2)]$
split : $(n\ m : \mathsf{nat}) \to (\delta : \mathsf{data}) \to \mathsf{exp}[nm.\delta] \to \mathsf{exp}[m.n.\delta]$
join : $(n\ m : \mathsf{nat}) \to (\delta : \mathsf{data}) \to \mathsf{exp}[n.m.\delta] \to \mathsf{exp}[nm.\delta]$
pair : $(\delta_1\ \delta_2 : \mathsf{data}) \to \mathsf{exp}[\delta_1] \to \mathsf{exp}[\delta_2] \to \mathsf{exp}[\delta_1 \times \delta_2]$
fst : $(\delta_1\ \delta_2 : \mathsf{data}) \to \mathsf{exp}[\delta_1 \times \delta_2] \to \mathsf{exp}[\delta_1]$
snd : $(\delta_1\ \delta_2 : \mathsf{data}) \to \mathsf{exp}[\delta_1 \times \delta_2] \to \mathsf{exp}[\delta_2]$

(a) Functional primitives

skip : comm
$(;)$: $\mathsf{comm} \times \mathsf{comm} \to \mathsf{comm}$
new : $(\delta : \mathsf{data}) \to (\mathsf{var}[\delta] \to \mathsf{comm}) \to \mathsf{comm}$ (where $\mathsf{var}[\delta] = \mathsf{acc}[\delta] \times \mathsf{exp}[\delta] : \mathsf{phrase}$)
$(:=)$: $\mathsf{acc}[\mathsf{num}] \times \mathsf{exp}[\mathsf{num}] \to \mathsf{comm}$
for : $(n : \mathsf{nat}) \to (\mathsf{exp}[\mathsf{idx}(n)] \to \mathsf{comm}) \to \mathsf{comm}$
parfor : $(n : \mathsf{nat}) \to (\delta : \mathsf{data}) \to \mathsf{acc}[n.\delta] \to (\mathsf{exp}[\mathsf{idx}(n)] \to \mathsf{acc}[\delta] \to_{\mathrm{p}} \mathsf{comm}) \to \mathsf{comm}$

splitAcc : $(n\ m : \mathsf{nat}) \to (\delta : \mathsf{data}) \to \mathsf{acc}[m.n.\delta] \to \mathsf{acc}[nm.\delta]$
joinAcc : $(n\ m : \mathsf{nat}) \to (\delta : \mathsf{data}) \to \mathsf{acc}[nm.\delta] \to \mathsf{acc}[n.m.\delta]$
$\mathbf{pairAcc_1}$: $(\delta_1\ \delta_2 : \mathsf{data}) \to \mathsf{acc}[\delta_1 \times \delta_2] \to \mathsf{acc}[\delta_1]$
$\mathbf{pairAcc_2}$: $(\delta_1\ \delta_2 : \mathsf{data}) \to \mathsf{acc}[\delta_1 \times \delta_2] \to \mathsf{acc}[\delta_2]$
$\mathbf{zipAcc_1}$: $(n : \mathsf{nat}) \to (\delta_1\delta_2 : \mathsf{data}) \to \mathsf{acc}[n.\delta_1 \times \delta_2] \to \mathsf{acc}[n.\delta_1]$
$\mathbf{zipAcc_2}$: $(n : \mathsf{nat}) \to (\delta_1\delta_2 : \mathsf{data}) \to \mathsf{acc}[n.\delta_1 \times \delta_2] \to \mathsf{acc}[n.\delta_2]$

idx : $(n : \mathsf{nat}) \to (\delta : \mathsf{data}) \to \mathsf{exp}[n.\delta] \to \mathsf{exp}[\mathsf{idx}(n)] \to \mathsf{exp}[\delta]$
idxAcc : $(n : \mathsf{nat}) \to (\delta : \mathsf{data}) \to \mathsf{acc}[n.\delta] \to \mathsf{exp}[\mathsf{idx}(n)] \to \mathsf{acc}[\delta]$

(b) Imperative primitives

mapI : $(n : \mathsf{nat}) \to (\delta_1\ \delta_2 : \mathsf{data}) \to (\mathsf{exp}[\delta_1] \to \mathsf{acc}[\delta_2] \to_{\mathrm{p}} \mathsf{comm}) \to \mathsf{exp}[n.\delta_1] \to \mathsf{acc}[n.\delta_2] \to \mathsf{comm}$
reduceI : $(n : \mathsf{nat}) \to (\delta_1\ \delta_2 : \mathsf{data}) \to (\mathsf{exp}[\delta_1] \to \mathsf{exp}[\delta_2] \to \mathsf{acc}[\delta_2] \to \mathsf{comm}) \to$
$\mathsf{exp}[\delta_2] \to \mathsf{exp}[n.\delta_1] \to (\mathsf{exp}[\delta_2] \to \mathsf{comm}) \to \mathsf{comm}$

(c) Intermediate imperative combinators

Fig. 4. Data Parallel Programming Primitives, Functional and Imperative

dimensional array, **pair** constructs a pair, and **fst** and **snd** deconstruct a pair. All of these primitives are data type indexed and those that operate on arrays are also size indexed.

Example: dot-product. The dot-product example from Section 2 is written using the functional primitives like so, for input vectors *xs* and *ys* of length *n*, the only difference being that all of the size and data type information

is described in detail (often we can infer these arguments and so in practice we often omit them):

$$\mathbf{reduce}\ n\ \mathsf{num}\ \mathsf{num}\ (\lambda x\ y.\ x + y)\ \underline{0}\ (\mathbf{map}\ n\ (\mathsf{num} \times \mathsf{num})\ \mathsf{num}\ (\lambda x.\ \mathbf{fst}\ \mathsf{num}\ \mathsf{num}\ x * \mathbf{snd}\ \mathsf{num}\ \mathsf{num}\ x)\ (\mathbf{zip}\ n\ \mathsf{num}\ \mathsf{num}\ xs\ ys))$$

Likewise, the specialised version of dot-product from Section 2 with nested **split**s and **join**s can be expressed with detailed size and type information throughout.

Imperative Primitives. Figure 4b gives the type signatures for the imperative primitives. These are split into two groups. The first group includes the standard Idealised Algol combinators that turn DPIA into an imperative programming language: (;) sequences commands, **skip** is the command that does nothing; **new** δ allocates a new mutable variable on the stack, where a variable is a pair of an acceptor and an expression; and (:=) assigns the value of an expression to an acceptor. For-loops are constructed by the combinators **for** and **parfor**. Sequential for-loops **for** $n\ b$ take a number of iterations n and a loop body b, a command parameterised by the iteration number. Parallel for-loops **parfor** $n\ \delta\ a\ b$ take an additional acceptor argument $a : \mathrm{acc}[n.\delta]$ that is used for the output of each iteration. The loop body for parallel for loops is required to be passive. This ensures that the side-effects of the loop body are restricted to their allotted place in the output array. This is illustrated by the non-typability of a phrase such as:

$$\mathbf{parfor}\ n\ \delta\ a\ (\lambda i\ o.\ b := \mathsf{idx}\ n\ \mathsf{num}\ E\ i)$$

where b is some identifier of type acc[num]. If this loop were executed in parallel, then it would contain a data race as each parallel iteration attempted to write to b. Thus, by ensuring its body is passive and explicitly passing in an acceptor o, **parfor** enables deterministic data race free parallelism in an imperative setting, a key feature of DPIA. We will see how the acceptor-transforming behaviour of our **parfor** primitive is translated into a normal, potentially racy, parallel for loop in Section 4.3.

Formally, newly allocated variables are zero initialised (and pointwise zero initialised for compound data), but in our implementation we typically optimise away the initialisation. In particular, it is never necessary to initialise dynamic memory allocations that are introduced by the translation of the functional primitives into imperative code as all dynamically allocated memory is always written to before being read.

The second group of imperative primitives include the acceptor variants of the **split**, **join** and **pair** functional primitives, and array indexing. The acceptor primitives transform acceptors of compound data into acceptors of their components. They will be used to funnel data into the correct positions in the imperative translations of functional programs. In the final translation to parallel C code, described in Section 4.3, all acceptor phrases will be translated into l-values with explicit index computations.

Intermediate Imperative Combinators. Figure 4c gives the type signatures for the intermediate imperative counterparts of **map** and **reduce**. These combinators will be used in our translation from higher-order functional programs to higher-order imperative programs in Section 4.1. In the second stage of the translation they will be substituted by implementations in terms of variable allocation and for-loops (Section 4.2).

4 FROM FUNCTIONAL TO IMPERATIVE

As we sketched in Section 2, the translation of higher-order functional array programs to parallel C-like code happens in three stages, which we describe in this section. First, the higher-order functional combinators **map** and **reduce** are translated into the higher-order imperative combinators **mapI** and **reduceI** using an acceptor-passing translation defined mutually recursively with a continuation-passing translation (Section 4.1). Secondly, the higher-order imperative combinators are translated into for-loops by substitution (Section 4.2). Finally, low-level parallel pseudo-C code is produced by translating expression and acceptor phrases into indexing operations (Section 4.3). We prove the correctness of the first two stages of our translation in Section 5.

$$
\begin{array}{lcl}
\mathcal{A}(\!|x|\!)_\delta(A) & = & A :=_\delta x \\
\mathcal{A}(\!|\underline{n}|\!)_{\mathsf{num}}(A) & = & A := \underline{n} \\
\mathcal{A}(\!|\mathbf{negate}\ E|\!)_{\mathsf{num}}(A) & = & C(\!|E|\!)_{\mathsf{num}}(\lambda x.\ A := \mathbf{negate}\ x) \\
\mathcal{A}(\!|E_1 + E_2|\!)_{\mathsf{num}}(A) & = & C(\!|E_1|\!)_{\mathsf{num}}(\lambda x.\ C(\!|E_2|\!)_{\mathsf{num}}(\lambda y.\ A := x + y)) \\
\mathcal{A}(\!|\mathbf{map}\ n\ \delta_1\ \delta_2\ F\ E|\!)_{n.\delta_2}(A) & = & C(\!|E|\!)_{n.\delta_1}(\lambda x.\ \mathbf{mapI}\ n\ \delta_1\ \delta_2\ (\lambda x\ o.\mathcal{A}(\!|F\ x|\!)_{\delta_2}(o))\ x\ A) \\
\mathcal{A}(\!|\mathbf{reduce}\ n\ \delta_1\ \delta_2\ F\ I\ E|\!)_{\delta_2}(A) & = & C(\!|E|\!)_{n.\delta_1}(\lambda x.\ C(\!|I|\!)_{\delta_2}(\lambda y. \\
 & & \qquad \mathbf{reduceI}\ n\ \delta_1\ \delta_2\ (\lambda x\ y\ o.\mathcal{A}(\!|F\ x\ y|\!)_{\delta_2}(o))\ y\ x\ (\lambda r.\mathcal{A}(\!|r|\!)(A)))) \\
\mathcal{A}(\!|\mathbf{zip}\ n\ \delta_1\ \delta_2\ E_1\ E_2|\!)_{n.\delta_1\times\delta_2}(A) & = & \mathcal{A}(\!|E_1|\!)_{n.\delta_1}(\mathbf{zipAcc}_1\ n\ \delta_1\ \delta_2\ A); \mathcal{A}(\!|E_2|\!)_{n.\delta_2}(\mathbf{zipAcc}_2\ n\ \delta_1\ \delta_2\ A) \\
\mathcal{A}(\!|\mathbf{split}\ n\ m\ \delta\ E|\!)_{n.m.\delta}(A) & = & \mathcal{A}(\!|E|\!)_{nm.\delta}(\mathbf{splitAcc}\ n\ m\ \delta\ A) \\
\mathcal{A}(\!|\mathbf{join}\ n\ m\ \delta\ E|\!)_{nm.\delta}(A) & = & \mathcal{A}(\!|E|\!)_{n.m.\delta}(\mathbf{joinAcc}\ n\ m\ \delta\ A) \\
\mathcal{A}(\!|\mathbf{pair}\ \delta_1\ \delta_2\ E_1\ E_2|\!)_{\delta_1\times\delta_2}(A) & = & \mathcal{A}(\!|E_1|\!)_{\delta_1}(\mathbf{pairAcc_1}\ \delta_1\ \delta_2\ A); \mathcal{A}(\!|E_2|\!)_{\delta_2}(\mathbf{pairAcc_2}\ \delta_1\ \delta_2\ A) \\
\mathcal{A}(\!|\mathbf{fst}\ \delta_1\ \delta_2\ E|\!)_{\delta_1}(A) & = & C(\!|E|\!)_{\delta_1\times\delta_2}(\lambda x.\ A :=_{\delta_1} \mathbf{fst}\ \delta_1\ \delta_2\ x) \\
\mathcal{A}(\!|\mathbf{snd}\ \delta_1\ \delta_2\ E|\!)_{\delta_2}(A) & = & C(\!|E|\!)_{\delta_1\times\delta_2}(\lambda x.\ A :=_{\delta_2} \mathbf{snd}\ \delta_1\ \delta_2\ x)
\end{array}
$$

(a) Acceptor-passing Translation

$$
\begin{array}{lcl}
C(\!|x|\!)_\delta(C) & = & C(x) \\
C(\!|\underline{n}|\!)_{\mathsf{num}}(C) & = & C(\underline{n}) \\
C(\!|\mathbf{negate}\ E|\!)_{\mathsf{num}}(C) & = & C(\!|E|\!)_{\mathsf{num}}(\lambda x.\ C(\mathbf{negate}\ x)) \\
C(\!|E_1 + E_2|\!)_{\mathsf{num}}(C) & = & C(\!|E_1|\!)_{\mathsf{num}}(\lambda x.\ C(\!|E_2|\!)_{\mathsf{num}}(\lambda y.\ C(x + y))\ ) \\
C(\!|\mathbf{map}\ n\ \delta_1\ \delta_2\ F\ E|\!)_{n.\delta_2}(C) & = & \mathbf{new}\ (n.\delta_2)\ (\lambda tmp.\ \mathcal{A}(\!|\mathbf{map}\ n\ \delta_1\ \delta_2\ F\ E|\!)_{n.\delta_2}(tmp.2);\ C(tmp.1)\ ) \\
C(\!|\mathbf{reduce}\ n\ \delta_1\ \delta_2\ F\ I\ E|\!)_{\delta_2}(C) & = & C(\!|E|\!)_{n.\delta_1}(\lambda x.\ C(\!|I|\!)_{\delta_2}(\lambda y.\ \mathbf{reduceI}\ n\ \delta_1\ \delta_2\ (\lambda x\ y\ o.\ \mathcal{A}(\!|F\ x\ y|\!)_{\delta_2}(o))\ y\ x)\ C) \\
C(\!|\mathbf{zip}\ n\ \delta_1\ \delta_2\ E_1\ E_2|\!)_{n.\delta_1\times\delta_2}(C) & = & C(\!|E_1|\!)_{n.\delta_1}(\lambda x.\ C(\!|E_2|\!)_{n.\delta_2}(\lambda y.\ C(\mathbf{zip}\ n\ \delta_1\ \delta_2\ x\ y))\ ) \\
C(\!|\mathbf{split}\ n\ m\ \delta\ E|\!)_{n.m.\delta}(C) & = & C(\!|E|\!)_{nm.\delta}(\lambda x.\ C(\mathbf{split}\ n\ m\ \delta\ x)\ ) \\
C(\!|\mathbf{join}\ n\ m\ \delta\ E|\!)_{nm.\delta}(C) & = & C(\!|E|\!)_{n.m.\delta}(\lambda x.\ C(\mathbf{join}\ n\ m\ \delta\ x)\ ) \\
C(\!|\mathbf{pair}\ \delta_1\ \delta_2\ E_1\ E_2|\!)_{\delta_1\times\delta_2}(C) & = & C(\!|E_1|\!)_{\delta_1}(\lambda x.\ C(\!|E_2|\!)_{\delta_2}(\lambda y.\ C(\mathbf{pair}\ \delta_1\ \delta_2\ x\ y))\ ) \\
C(\!|\mathbf{fst}\ \delta_1\ \delta_2\ E|\!)_{\delta_1}(C) & = & C(\!|E|\!)_{\delta_1\times\delta_2}(\lambda x.\ C(\mathbf{fst}\ \delta_1\ \delta_2\ x)\ ) \\
C(\!|\mathbf{snd}\ \delta_1\ \delta_2\ E|\!)_{\delta_2}(C) & = & C(\!|E|\!)_{\delta_1\times\delta_2}(\lambda x.\ C(\mathbf{snd}\ \delta_1\ \delta_2\ x)\ )
\end{array}
$$

(b) Continuation-passing Translation

Fig. 5. Acceptor and Continuation-passing Translations

4.1 Translation Stage I: Higher-order Functional to Higher-order Imperative

The goal of the first stage of the translation is to take a phrase $E : \exp[\delta]$, constructed from the functional primitives in Figure 4a, and an acceptor $out : \mathrm{acc}[\delta]$ and to produce a comm phrase that has the same semantics as the command

$$out :=_\delta E$$

where $(:=_\delta)$ is an assignment operator for non-base types defined by induction on δ below. The resulting program will be an imperative program that acts as if we could compute the functional expression in one go and assign it to the output acceptor. Since our compilation targets know nothing of higher-order functional combinators like **map** and **reduce** they will have to be translated away. We do not use any of the traditional methods for compiling higher-order functions, such as closure conversion (Steele 1978) or defunctionalisation (Reynolds 1998). Instead, we rely on the whole-program nature of our translation, our lack of recursion, and the special form of our functional primitives. Specifically, we are relying on a version of Gentzen's subformula principle (identified by Gentzen (1935) and named by Prawitz (1965)). Our approach is reminiscent of that of Najd et al. (2016) who use quotation and normalisation, making essential use of the subformula principle, to embed domain-specific

languages. An obvious difference with our work is that rather than stratifying a language into a host functional language and a quoted functional language, we seamlessly combine a functional and an imperative language.

We have already mentioned the use of assignment at compound data types. This is defined by:

$$\begin{array}{lcl} A :=_{\mathsf{num}} E & = & A := E \\ A :=_{n.\delta} E & = & \mathbf{mapI}\ n\ \delta\ \delta\ (\lambda x\ a.a :=_{\delta} x)\ E\ A \\ A :=_{\delta_1 \times \delta_2} E & = & \mathbf{pairAcc_1}\ A :=_{\delta_1} \mathbf{fst}\ E; \mathbf{pairAcc_2}\ A :=_{\delta_2} \mathbf{snd}\ E \end{array}$$

The translation of functional expressions to imperative code is accomplished by two type-directed mutually defined translations: the acceptor-passing translation $\mathcal{A}(\!|-|\!)_\delta$ (Figure 5a) and the continuation-passing translation $C(\!|-|\!)_\delta$ (Figure 5b). The acceptor-passing translation takes a data type δ, an expression of type $\mathsf{exp}[\delta]$ and an acceptor of type $\mathsf{acc}[\delta]$, and produces a comm phrase. Likewise, the continuation-passing translation takes a data type δ, an expression of type $\mathsf{exp}[\delta]$ and a continuation of type $\mathsf{exp}[\delta] \to \mathsf{comm}$, and produces a comm phrase.

It is straightforward to see by inspection, and using the fact that weakening is admissible in DPIA, that the two translations are type-preserving:

Theorem 4.1.

(1) *If* $\Delta \mid \Pi; \Gamma_1 \vdash E : \mathsf{exp}[\delta]$ *and* $\Delta \mid \Pi; \Gamma_2 \vdash A : \mathsf{acc}[\delta]$ *then* $\Delta \mid \Pi; \Gamma_1, \Gamma_2 \vdash \mathcal{A}(\!|E|\!)_\delta(A) : \mathsf{comm}$.

(2) *If* $\Delta \mid \Pi; \Gamma_1 \vdash E : \mathsf{exp}[\delta]$ *and* $\Delta \mid \Pi; \Gamma_2 \vdash C : \mathsf{exp}[\delta] \to \mathsf{comm}$ *then* $\Delta \mid \Pi; \Gamma_1, \Gamma_2 \vdash C(\!|E|\!)_\delta(C) : \mathsf{comm}$.

It is also important that these translations satisfy the following equivalences.

$$\mathcal{A}(\!|E|\!)_\delta(A) \simeq A :=_\delta E \qquad\qquad C(\!|E|\!)_\delta(C) \simeq C(E)$$

We define observational equivalence ($\simeq$) for DPIA and establish these particular equivalences in Section 5. Ultimately, our goal is to compute the result of $\mathcal{A}(\!|E|\!)_\delta(\mathit{out})$.

It might appear that we could dispense with the acceptor-passing translation and simply use $C(\!|E|\!)_\delta(\lambda x.\ \mathit{out} :=_\delta x)$. However, this would create unnecessary temporary storage, violating our desire for an efficient translation. There are clear similarities between our mutually-defined translations and tail-recursive one-pass CPS translations that do not produce unnecessary administrative redexes (Danvy et al. 2007).

The clauses for both translations split into four groups. The first group consists only of the clause for translating functional expression phrases that are just identifiers x. In the acceptor-passing case, we defer to the generalised assignment defined above; in the continuation-passing case, we simply apply the continuation to the variable. The second group handles the first-order operations on numeric data. In all cases, we defer to the continuation-passing translation for the sub-expressions, with appropriate continuations.

The third group is the most interesting: the translations of the higher order **map** and **reduce** primitives. For **map**: in the acceptor-passing case, we can immediately translate to **mapI** which already takes an acceptor to place its output into; in the continuation-passing case, we must create a temporary array as storage, invoke **mapI**, and then let the continuation read from the temporary array. This indirection is required because we do not know what random access strategy the continuation C will use to read the array it is given. For **reduce**, in both cases we translate to the **reduceI** combinator.

The fourth group of clauses handles the translations of the functional data layout combinators. In the continuation-passing translation, they are passed straight through. They will be handled by the final translation to low-level C-like code in Section 4.3. In the acceptor-passing translation, the combinators that construct data are translated into the corresponding acceptors. In the **fst** and **snd** cases, which project out data, we defer to the continuation-passing translation. In practice, the case of a projection in tail position rarely arises, since it corresponds to disposal of part of the overall computation.

4.2 Translation Stage II: Higher-order Imperative to For-loops

The next stage in the translation replaces the intermediate level imperative combinators **mapI** and **reduceI** with lower-level implementations in terms of (parallel) for-loops. This is accomplished by substitution and β-reduction

(DPIA includes full $\beta\eta$ reasoning principles). The combinator **mapI** is implemented as a parallel loop:

$$\textbf{mapI} = \Lambda n\ \delta_1\ \delta_2.\lambda F\ E\ A.\ \textbf{parfor}\ n\ \delta_2\ A\ (\lambda i\ a.\ F\ (\textbf{idx}\ n\ (n.\delta_1)\ i\ E)\ a)$$

The implementation of **reduceI** is more complex, involving the allocation of a temporary variable to store the accumulated value during the reduction. In this case, the for loop is sequential, since the semantics of reduction demands that we visit each element in turn:

$$\begin{aligned}\textbf{reduceI} = \Lambda n\ \delta_1\ \delta_2.\lambda\ F\ I\ E\ C.\ \textbf{new}\ \delta_2\ (\lambda\ acc.\ & acc.2 :=_{\delta_2} I;\\ & \textbf{for}\ n\ (\lambda\ i.\ F\ (\textbf{idx}\ n\ (n.\delta_1)\ i\ E)\ (acc.1)\ (acc.2))\ );\\ & C\ (acc.1)\ )\end{aligned}$$

These definitions define the intended semantics of the intermediate-level imperative combinators.

4.3 Translation Stage III: For-loops to Parallel Pseudo-C

After performing the translation steps in the previous two sections, we have generated a command phrase that does not use the higher-order functional combinators **map** and **reduce**, but still contains uses of the data layout combinators **zip**, **split** etc., and their acceptor counterparts. We now define a translation to a C-like language with parallel for loops that resolves these data layout expressions into explicit indexing expressions. The translation is defined in Figure 6. Parallel for loops can easily be achieved using OpenMP's `#pragma parallel for` construct, for example. In Section 6, we describe how to adapt this process (and the earlier stages) to work with the real parallel C-like language OpenCL.

The translation in Figure 6 is split into three parts: the translation of DPIA commands into C statements, the translation of acceptors into l-values, and the translation of expressions into r-values. (Recall the analogy made in Section 3.1 between the phrase types of DPIA and syntactic categories in an imperative language; we are now reaping the rewards of Reynolds' careful design of Idealised Algol.) We assume that the input to the translation process is in $\beta\eta$-normal form, so all **new**-block and loop bodies are fully expanded.

The translation of commands in Figure 6a straightforwardly translates each DPIA command into the corresponding C statement. The translation is parameterised by an environment η that maps from DPIA identifiers to C variable names. There is a small discrepancy that we overcome in that semicolons are statement terminators in C, but separators in DPIA, and that doing nothing useful is unremarkable in a C program, but is explicitly written **skip** in DPIA. The translation of assignment relies on the translations for acceptors and expression, which we define below. Variable allocation is translated using a `{ ... }` block to limit the scope. We omit initialisation of the new variable because we know by inspection of our previous translation steps that newly allocated variables will always be completely initialised before reading. Note how we explicitly substitute a pair of identifiers for the acceptor and expression parts of the variable in DPIA, but that these both refer to the same C variable in the extended environment. Translation of variable allocation makes use of generator $\textsc{CodeGen}_{\mathsf{data}}(\delta, \mathsf{v})$ that generates the appropriate C-style variable declaration for the data type δ. Since C does not have anonymous tuple types, this entails on-the-fly generation of the appropriate `struct` definitions.

DPIA **for** loops are translated into C `for` loops, DPIA **parfor** loops are translated into pseudo-parallel-for loops. In the body of the `parfor` loop, we substitute in an **idxAcc** phrase which will be resolved later by the translation of acceptors.

The variable names introduced in the translations of **new**, **for** and **parfor** are all assumed to be fresh.

The translation of acceptors (Figure 6b) is parameterised by an environment η, as for commands, as well as a path *ps*, consisting of a list of C-expressions of type `int` denoting array indexes, and `struct` fields, `.x1`, `.x2`, denoting projections from pairs. The path must always agree with the type of the acceptor being translated. During command translation, all acceptors being translated have type `num` so the access path starts empty. The acceptor translation clauses in Figure 6b all proceed by manipulating the access path appropriately until an

$$
\begin{aligned}
&\textsc{CodeGen}_{\text{comm}}(\textbf{skip}, \eta) && = \texttt{/* skip */}\\
&\textsc{CodeGen}_{\text{comm}}(\mathsf{P}_1; \mathsf{P}_2, \eta) && = \textsc{CodeGen}_{\text{comm}}(P_1, \eta)\ \textsc{CodeGen}_{\text{comm}}(P_2, \eta)\\
&\textsc{CodeGen}_{\text{comm}}(A := E, \eta) && = \textsc{CodeGen}_{\text{acc[num]}}(A, \eta, [])\ \texttt{=}\ \textsc{CodeGen}_{\text{exp[num]}}(E, \eta, [])\texttt{;}\\
&\textsc{CodeGen}_{\text{comm}}(\textbf{new}\ \delta\ (\lambda v.\ P), \eta) && = \{\ \textsc{CodeGen}_{\text{data}}(\delta)\ \texttt{v;}\\
&&& \quad \textsc{CodeGen}_{\text{comm}}(P[(v_a, v_e)/v], \eta[v_a \mapsto \texttt{v}, v_e \mapsto \texttt{v}])\ \}\\
&\textsc{CodeGen}_{\text{comm}}(\textbf{for}\ n\ (\lambda i.\ P), \eta) && = \texttt{for(int i = 0; i < } n\texttt{; i += 1) \{}\\
&&& \quad \textsc{CodeGen}_{\text{comm}}(P, \eta[i \mapsto \texttt{i}])\\
&&& \texttt{\}}\\
&\textsc{CodeGen}_{\text{comm}}(\textbf{parfor}\ n\ \delta\ A\ (\lambda i\ o.\ P)\ E, \eta) && = \texttt{parfor(int i = 0; i < } n\texttt{; i += 1) \{}\\
&&& \quad \textsc{CodeGen}_{\text{comm}}(P[\textbf{idxAcc}\ n\ \delta\ A\ i/o], \eta[i \mapsto \texttt{i}])\\
&&& \texttt{\}}
\end{aligned}
$$

(a) DPIA commands to C statements

$$
\begin{aligned}
&\textsc{CodeGen}_{\text{acc}[\delta]}(x, \eta, ps) && = \eta(x)(\mathit{reverse}\ ps)\\
&\textsc{CodeGen}_{\text{acc}[\delta]}(\textbf{idxAcc}\ n\ \delta\ A\ i, \eta, ps) && = \textsc{CodeGen}_{\text{acc}[\texttt{n}.\delta]}(A, \eta, \eta(i) :: ps)\\
&\textsc{CodeGen}_{\text{acc}[\texttt{nm}.\delta]}(\textbf{splitAcc}\ n\ m\ \delta\ A, \eta, \texttt{i} :: ps) && = \textsc{CodeGen}_{\text{acc}[\texttt{n.m}.\delta]}(A, \eta, \texttt{i/}m :: \texttt{i\%}m :: ps)\\
&\textsc{CodeGen}_{\text{acc}[\texttt{n.m}.\delta]}(\textbf{joinAcc}\ n\ m\ \delta\ A, \eta, \texttt{i} :: \texttt{j} :: ps) && = \textsc{CodeGen}_{\text{acc}[\texttt{nm}.\delta]}(A, \eta, \texttt{i*}m\texttt{+j} :: ps)\\
&\textsc{CodeGen}_{\text{acc}[\delta_1]}(\textbf{pairAcc}_{\textbf{1}}\ \delta_1\ \delta_2\ A, \eta, ps) && = \textsc{CodeGen}_{\text{acc}[\delta_1 \times \delta_2]}(A, \eta, \texttt{.x1} :: ps)\\
&\textsc{CodeGen}_{\text{acc}[\delta_2]}(\textbf{pairAcc}_{\textbf{2}}\ \delta_1\ \delta_2\ A, \eta, ps) && = \textsc{CodeGen}_{\text{acc}[\delta_1 \times \delta_2]}(A, \eta, \texttt{.x2} :: ps)\\
&\textsc{CodeGen}_{\text{acc}[n.\delta_1]}(\textbf{zipAcc}_{\textbf{1}}\ n\ \delta_1\ \delta_2\ A, \eta, \texttt{i} :: ps) && = \textsc{CodeGen}_{\text{acc}[n.(\delta_1 \times \delta_2)]}(A, \eta, \texttt{i} :: \texttt{.x1} :: ps)\\
&\textsc{CodeGen}_{\text{acc}[n.\delta_2]}(\textbf{zipAcc}_{\textbf{2}}\ n\ \delta_1\ \delta_2\ A, \eta, \texttt{i} :: ps) && = \textsc{CodeGen}_{\text{acc}[n.(\delta_1 \times \delta_2)]}(A, \eta, \texttt{i} :: \texttt{.x2} :: ps)
\end{aligned}
$$

(b) DPIA acceptors to C l-values

$$
\begin{aligned}
&\textsc{CodeGen}_{\text{exp}[\delta]}(x, \eta, ps) && = \eta(x)(\mathit{reverse}\ ps)\\
&\textsc{CodeGen}_{\text{exp[num]}}(\underline{n}, \eta, []) && = n\\
&\textsc{CodeGen}_{\text{exp[num]}}(\textbf{negate}\ E, \eta, []) && = \texttt{(-}\ \textsc{CodeGen}_{\text{exp[num]}}(E, \eta, [])\texttt{)}\\
&\textsc{CodeGen}_{\text{exp[num]}}(E_1 + E_2, \eta, []) && = \texttt{(}\textsc{CodeGen}_{\text{exp[num]}}(E_1, \eta, [])\ \texttt{+}\ \textsc{CodeGen}_{\text{exp[num]}}(E_2, \eta, [])\texttt{)}\\
&\textsc{CodeGen}_{\text{exp}[n.(\delta_1 \times \delta_2)]}(\textbf{zip}\ n\ \delta_1\ \delta_2\ E_1\ E_2, \eta, \texttt{i} :: \texttt{.x}j :: ps) && = \textsc{CodeGen}_{\text{exp}[n.\delta_j]}(E_j, \eta, \texttt{i} :: ps)\\
&\textsc{CodeGen}_{\text{exp}[m.n.\delta]}(\textbf{split}\ n\ m\ \delta\ E, \eta, \texttt{i} :: \texttt{j} :: ps) && = \textsc{CodeGen}_{\text{exp}[mn.\delta]}(E, \eta, \texttt{i*}n\texttt{+j} :: ps)\\
&\textsc{CodeGen}_{\text{exp}[mn.\delta]}(\textbf{join}\ n\ m\ \delta\ E, \eta, \texttt{i} :: ps) && = \textsc{CodeGen}_{\text{exp}[m.n.\delta]}(E, \eta, \texttt{i/}n :: \texttt{i\%}n :: ps)\\
&\textsc{CodeGen}_{\text{exp}[\delta_1 \times \delta_2]}(\textbf{pair}\ \delta_1\ \delta_2\ E_1\ E_2, \eta, \texttt{.x}j :: ps) && = \textsc{CodeGen}_{\text{exp}[\delta_j]}(E_j, \eta, ps)\\
&\textsc{CodeGen}_{\text{exp}[\delta_1]}(\textbf{fst}\ \delta_1\ \delta_2\ E, \eta, ps) && = \textsc{CodeGen}_{\text{exp}[\delta_1 \times \delta_2]}(E, \eta, \texttt{.x1} :: ps)\\
&\textsc{CodeGen}_{\text{exp}[\delta_2]}(\textbf{snd}\ \delta_1\ \delta_2\ E, \eta, ps) && = \textsc{CodeGen}_{\text{exp}[\delta_1 \times \delta_2]}(E, \eta, \texttt{.x2} :: ps)\\
&\textsc{CodeGen}_{\text{exp}[\delta]}(\textbf{idx}\ n\ \delta\ E\ I) && = \textsc{CodeGen}_{\text{exp}[n.\delta_1]}(E, \eta, \textsc{CodeGen}_{\text{exp}[\mathit{idx}(n)]}(I, \eta, []) :: ps)
\end{aligned}
$$

(c) DPIA expressions to C r-values

Fig. 6. Translation of Purely Imperative DPIA to Parallel Pseudo C

identifier is reached. At this point, the DPIA identifier is replaced with its corresponding C variable and the access path is appended.

Translation of expressions (Figure 6c) is parameterised similarly to the acceptor translation, and contains similar clauses for all the data layout combinators. Expressions also include literals and arithmetic expressions, which are translated to the corresponding notion in C.

Example. We demonstrate how the translation to C works by applying it to the **parfor** loop in the translation (3) of the simple dot product in Section 2. We use the environment $\eta = [\mathit{out} \mapsto \texttt{out}, \mathit{xs} \mapsto \texttt{xs}, \mathit{ys} \mapsto \texttt{ys}]$. The

command translation translates, in two steps, the **parfor** loop and the assignment, substituting in the indexing acceptor for the acceptor in the loop body:

$$
\begin{array}{ll}
 & \textsc{CodeGen}_{\mathsf{comm}}(\mathbf{parfor}\ \underline{n}\ out\ (\lambda i\ o.o := \mathbf{fst}\ (\mathbf{idx}\ (\mathbf{zip}\ xs\ ys)\ i) * \mathbf{snd}\ (\mathbf{idx}\ (\mathbf{zip}\ xs\ ys)\ i)), \eta) \\
= & \texttt{parfor(int i = 0; i < } n\texttt{; i+=1) \{} \\
 & \quad \textsc{CodeGen}_{\mathsf{comm}}(\mathbf{idxAcc}\ out\ i := \mathbf{fst}\ (\mathbf{idx}\ (\mathbf{zip}\ xs\ ys)\ i) * \mathbf{snd}\ (\mathbf{idx}\ (\mathbf{zip}\ xs\ ys)\ i)), \eta[i \mapsto \texttt{i}]) \\
 & \texttt{\}} \\
= & \texttt{parfor(int i = 0; i < } n\texttt{; i+=1) \{} \\
 & \quad \textsc{CodeGen}_{\mathsf{acc[num]}}(\mathbf{idxAcc}\ out\ i, \eta[i \mapsto \texttt{i}], []) = \\
 & \qquad \textsc{CodeGen}_{\mathsf{exp[num]}}(\mathbf{fst}\ (\mathsf{idx}\ (\mathbf{zip}\ xs\ ys)\ i) * \mathbf{snd}\ (\mathbf{idx}\ (\mathbf{zip}\ xs\ ys)\ i)), \eta[i \mapsto \texttt{i}], []); \\
 & \texttt{\}}
\end{array}
$$

The acceptor part of the assignment is translated as follows:

$$
\begin{array}{lcl}
\textsc{CodeGen}_{\mathsf{acc[num]}}(\mathbf{idxAcc}\ out\ i, \eta[i \mapsto \texttt{i}], []) & = & \textsc{CodeGen}_{\mathsf{acc}[\underline{n}.\mathsf{num}]}(out, \eta[i \mapsto \texttt{i}], [\texttt{i}]) \\
 & = & \texttt{out[i]}
\end{array}
$$

The expression part of the assignment is translated as follows, where we only spell out the left-hand side of the multiplication in detail; the right hand side is similar.

$$
\begin{array}{ll}
 & \textsc{CodeGen}_{\mathsf{exp[num]}}(\mathbf{fst}\ (\mathbf{idx}\ (\mathbf{zip}\ xs\ ys)\ i), \eta[i \mapsto \texttt{i}], []) \\
= & \textsc{CodeGen}_{\mathsf{exp[num\times num]}}(\mathbf{idx}\ (\mathbf{zip}\ xs\ ys)\ i, \eta[i \mapsto \texttt{i}], [\texttt{.x1}]) \\
= & \textsc{CodeGen}_{\mathsf{exp}[\underline{n}.(\mathsf{num}\times\mathsf{num})]}(\mathbf{zip}\ xs\ ys, \eta[i \mapsto \texttt{i}], [\texttt{i}, \texttt{.x1}]) \\
= & \textsc{CodeGen}_{\mathsf{exp}[\underline{n}.\mathsf{num}]}(xs, \eta[i \mapsto \texttt{i}], [\texttt{i}]) \\
= & \texttt{xs[i]}
\end{array}
$$

Putting everything together, we get the following translation of the original **parfor** loop, which has had all the data layout combinators translated away.

```
parfor(int i = 0; i < n; i+=1) { out[i] = xs[i] * ys[i]; }
```

A similar translation was recently presented in a more informal style by Steuwer et al. (2017). Their experimental results show that in practice it is important to keep the indices concise and short for generating efficient OpenCL code and discuss how to simplify index expressions making use of range information of the indices involved.

5 CORRECTNESS OF THE TRANSLATION

We now justify the translation process described in Section 4.1 and Section 4.2. We have not yet formalised a correctness proof for the final translation to C (Section 4.3); this is future work.

As we stated in Section 4, the goal of the translation process was to generate a "purely imperative" command $\mathcal{A}(\!|E|\!)_\delta(A)$ that is equivalent to the assignment command $A :=_\delta E$. To make this statement formal, we must define equivalence of DPIA programs. We do this in Section 5.1. We then state the functional coincidence property that establishes that our correctness property means just what we intend (Section 5.2). Finally, we prove the correctness of our translation in Section 5.3.

5.1 Semantics and Observational Equivalence for DPIA

We use Reddy's relationally parametric automata-theoretic semantics of SCIR (Reddy 2013). In this model, the interpretations of phrase types are parameterised by automata describing the permitted state transitions. The model supports relationally parametric reasoning (Reynolds 1983), which enables reasoning about locality and information hiding (following (O'Hearn and Tennent 1995)) and the use of automata permits reasoning about phrases that do not affect the state (i.e. passive phrases).

Reddy's model does not include indexed types or compound data types, as we do in DPIA, but these are straightforward to add to the model. Indexed types are interpreted by set-theoretic functions whose codomain depends on the input (we do not interpret the data type polymorphism using parametricity because we are only

interested in parametric reasoning for local state). Compound data types are interpreted as the following sets when they appear in expressions:

$$[\![\mathsf{num}]\!] = \mathsf{num} \qquad [\![\delta_1 \times \delta_2]\!] = [\![\delta_1]\!] \times [\![\delta_2]\!] \qquad [\![\underline{n}.\delta]\!] = \{0, \ldots, n-1\} \to [\![\delta]\!]$$

where the set num is some set of number-like objects used for scalar values. This interpretation of data types allows us to straightforwardly interpret the functional primitives of Figure 4a in the model, using the standard interpretation of **map** and **reduce** explicitly given in (Steuwer et al. 2015). This yields a coincidence property that we state in Section 5.2 below. Acceptors for compound data types are interpreted using the separating product construction, which ensures that disjoint components of compound acceptors are always non-interfering. This allows us to interpret **parfor** n as n parallel transformations on n pieces of disjoint state.

Reddy's semantics assigns an interpretation to every DPIA phrase. For observational equivalence of DPIA programs, we are interested in closed programs. A closed program is a command phrase whose free identifiers are all have types of the form $\mathsf{var}[\delta]$ for possibly different δ. Our notion of observational equivalence is standard. Two well-typed phrases $\Delta \mid \Pi; \Gamma \vdash P_1, P_2 : \theta$ are equivalent iff for all closing contexts $C[-]$, the programs $C[P_1]$ and $C[P_2]$, when instantiated with the standard interpretation of variables ((Reddy 2013), Figure 2), describe the same mapping of initial to final values. We formally write $\Delta \mid \Pi; \Gamma \vdash P_1 \simeq P_2 : \theta$, or informally $P_1 \simeq P_2$. Note that this relation is automatically an equivalence and is congruent with all the constructs of DPIA.

For the purposes of compilation, this notion of equivalence is justifiable: we are only interested in the relationship between the initial and final values of each variable, not the intermediate states.

5.2 Functional Coincidence

Our correctness criterion will have no force if we do not first establish that the assignment $out :=_\delta E$ means what we think it means. We state our coincidence property formally as follows. Let $\cdot \mid x_1 : \mathsf{exp}[\delta_1], \ldots, x_n : \mathsf{exp}[\delta_n]; \cdot \vdash E : \mathsf{exp}[\delta]$ be some expression phrase of DPIA built from the functional primitives in Figure 4a. Let $[\![E]\!] : [\![\delta_1]\!] \times \ldots \times [\![\delta_n]\!] \to [\![\delta]\!]$ be the functional reference semantics of E. Use E to construct a closed phrase:

$$\cdot \mid \cdot; v_1 : \mathsf{var}[\delta_1], \ldots, v_n : \mathsf{var}[\delta_n], out : \mathsf{var}[\delta] \vdash out.1 :=_\delta E[v_1.2/x_1, \ldots, v_n.2/x_n] : \mathsf{comm}$$

Then for all $a_1 \in [\![\delta_1]\!], \ldots, a_n \in [\![\delta_n]\!], a \in [\![\delta]\!]$, the interpretation of this command maps the store $(v_1 \mapsto a_1, \ldots, v_n \mapsto a_n, out \mapsto a)$ to the store $(v_1 \mapsto a_1, \ldots, v_n \mapsto a_n, out \mapsto [\![E]\!](a_1, \ldots, a_n))$. In other words, this program updates the variable *out* with the result of the expression, and leaves every other variable unaffected. We use Steuwer et al. (2015)'s interpretation of the functional primitives in our DPIA semantics, so this property is immediate.

5.3 Correctness of the Translation from Functional to Imperative

We structure our proof by first stating a collection of equivalences that can be proved in Reddy's model, and then use them to prove that the translation of Section 4.1 and Section 4.2 is correct (Theorem 5.1). The properties of contextual equivalences that we use in our proof, in addition to the fact that $\simeq$ is a congruent equivalence relation, are as follows.

(1) $\beta\eta$-equality for non-dependent and dependent functions:

$$(\lambda x.\ P)Q \simeq P[Q/x] \qquad P \simeq (\lambda x.\ Px) \qquad (\Lambda x.\ P)e \simeq P[e/x] \qquad P \simeq (\Lambda x.\ Px)$$

Full $\beta\eta$-equality for functions is one of the defining features of Idealised Algol and its descendents (Reynolds 1997). These are all justified by Reddy's model (and indeed almost all models of Idealised Algol-like languages).

(2) The **parfor**-based implementation of **mapI** (Section 4.2) satisfies the following equivalence:

$$\mathbf{mapI}\ n\ \delta_1\ \delta_2\ (\lambda x\ o.\ o :=_{\delta_2} F\ x)\ E\ A \quad \simeq \quad A :=_{n.\delta_2} \mathbf{map}\ n\ \delta_1\ \delta_2\ F\ E \tag{4}$$

By the definition of array assignment given in Section 4.1, this property is equivalent to:

$$\mathbf{mapI}\ n\ \delta_1\ \delta_2\ (\lambda x\ o.\ o :=_{\delta_2} F\ x)\ E\ A \quad \simeq \quad \mathbf{mapI}\ n\ \delta_2\ \delta_2\ (\lambda x\ o.\ o :=_{\delta_2} x)\ (\mathbf{map}\ \delta_1\ \delta_2\ F\ E)\ A$$

Expanding the definition of **mapI**, and β-reducing, we must show:

$$\textbf{parfor}\ n\ A\ (\lambda i\ o.\ o :=_{\delta_2} F\ (\textbf{idx}\ n\ \delta_1\ E\ i)) \simeq \textbf{parfor}\ n\ A\ (\lambda i\ o.\ o :=_{\delta_2} \textbf{idx}\ n\ \delta_2\ (\textbf{map}\ n\ \delta_1\ \delta_2\ F\ E)\ i)$$

which is immediate from the way that array data types are interpreted as functions from indices to values.

(3) Reddy's model validates the following equivalence involving the use of temporary storage. For all expressions E and continutations C that are non-interfering, we have:

$$\textbf{new}\ \delta\ (\lambda tmp.\ tmp.1 :=_{\delta} E; C(tmp.2)) \quad \simeq \quad C(E) \tag{5}$$

This equivalence relies crucially on the fact that C and E cannot interfere, so we can take a complete copy of E before invoking C. If C were able to write to storage that is read by E, then it would not be safe to cache E before invoking C. In Reddy's model, we use parametricity to relate the two uses of C: one in a store that contains the state that E reads, and one in a store that contains the result of evaluating E. Using parametricity and restriction to only the identity state transition on E's portion of the store further ensures that C does not interfere with E.

(4) The **for**-loop based implementation of **reduceI** should satisfy the following equivalence.

$$\textbf{reduceI}\ n\ \delta_1\ \delta_2\ (\lambda x\ y\ o.\ o :=_{\delta_2} F\ x\ y)\ I\ E\ C \quad \simeq \quad C(\textbf{reduce}\ n\ \delta 1\ \delta_2\ F\ I\ E) \tag{6}$$

Substituting in the implementation of **reduceI**, and β-reducing, this is equivalent to showing:

$$\textbf{new}\ \delta_2\ (\lambda v.\ v.1 :=_{\delta_2} I; \textbf{for}\ n\ (\lambda i.\ v.1 :=_{\delta_2} F\ v.2\ (\textbf{idx}\ E\ i)); C(v.2)) \simeq C(\textbf{reduce}\ n\ \delta 1\ \delta_2\ F\ I\ E)$$

Because the acceptor-expression pair v has been freshly allocated, it acts like a so-called "good variable" in Idealised Algol terminology. This means that the following equivalence holds, using the fact that neither F nor E interfere with v:

$$\textbf{for}\ n\ (\lambda i.\ v.1 :=_{\delta_2} F\ v.2\ (\textbf{idx}\ E\ i)) \simeq v.1 :=_{\delta_2} \textbf{reduce}\ n\ \delta_1\ \delta_2\ F\ v.2\ E$$

Now Equation 6 follows from Equation 5.

(5) Finally, we need agreement between the data layout combinators and their acceptor counterparts:

$$\begin{array}{lcl}
A :=_{\delta_1 \times \delta_2} \textbf{pair}\ \delta_1\ \delta_2\ E_1\ E_2 & \simeq & (\textbf{pairAcc}_\textbf{1}\ \delta_1\ \delta_2\ A :=_{\delta_1} E_1; \textbf{pairAcc}_\textbf{2}\ \delta_1\ \delta_2\ A :=_{\delta_2} E_2) \\
A :=_{n.(\delta_1 \times \delta_2)} \textbf{zip}\ n\ \delta_1\ \delta_2\ E_1\ E_2 & \simeq & (\textbf{zipAcc}_\textbf{1}\ n\ \delta_1\ \delta_1\ A :=_{n.\delta_1} E_1; \textbf{zipAcc}_\textbf{2}\ n\ \delta_1\ \delta_1\ A :=_{n.\delta_2} E_2) \\
A :=_{n.m.\delta} \textbf{split}\ n\ m\ \delta\ E & \simeq & \textbf{splitAcc}\ n\ m\ \delta\ A :=_{nm.\delta} E \\
A :=_{nm.\delta} \textbf{join}\ n\ m\ \delta\ E & \simeq & \textbf{joinAcc}\ n\ m\ \delta\ A :=_{n.m.\delta} E
\end{array}$$

The first equivalence follows directly from the definition of assignment at pair type, and β-reduction for pairs. The others are all straightforwardly justified in Reddy's model, given the interpretation of acceptors for compound data types using separating products, described above.

Theorem 5.1. *The translations $\mathcal{A}(\!|-|\!)_-(-)$ and $C(\!|-|\!)_-(-)$ defined in Figure 5a and Figure 5b satisfy the following observational equivalences for all acceptors A and functional expressions E with disjoint sets of active identifiers:*

$$\mathcal{A}(\!|E|\!)_\delta(A) \simeq A :=_\delta E \qquad\qquad C(\!|E|\!)_\delta(C) \simeq C(E)$$

Proof. By mutual induction on the steps of the translation process. The cases for variables in both translations are immediate. The cases for the first-order combinators on numbers follow from the induction hypotheses and β-reduction. For example, for **negate**:

$$\mathcal{A}(\!|\textbf{negate}\ E|\!)_{\mathsf{num}}(A) = C(\!|E|\!)_{\mathsf{num}}(\lambda x.\ A := \textbf{negate}\ x) \simeq (\lambda x.\ A := \textbf{negate}\ x)(E) \simeq A := \textbf{negate}\ E$$

The case for the first-order combinators in the continuation-passing translation are similar.

The acceptor-passing translation of **map** uses the induction hypothesis to establish the correctness of the translations of the subterms, β-equality, and then the correctness property of **mapI** (Equation 4):

$$\begin{array}{lcl}
\mathcal{A}(\!|\textbf{map}\ n\ \delta_1\ \delta_2\ F\ E|\!)_{n.\delta_2}(A) & = & C(\!|E|\!)_{n.\delta_1}(\lambda x.\ \textbf{mapI}\ n\ \delta_1\ \delta_2\ (\lambda x\ o.\ \mathcal{A}(\!|F\ x|\!)_{\delta_2}(o))\ x\ A) \\
& \simeq & \textbf{mapI}\ n\ \delta_1\ \delta_2\ (\lambda x\ o.\ \mathcal{A}(\!|F\ x|\!)_{\delta_2}(o))\ E\ A) \\
& \simeq & \textbf{mapI}\ n\ \delta_1\ \delta_2\ (\lambda x\ o.\ o :=_{\delta_2} F\ x)\ E\ A \\
& \simeq & A :=_{n.\delta_2} \textbf{map}\ n\ \delta_1\ \delta_2\ F\ E
\end{array}$$

The continuation-passing translation of **map** relies on the acceptor-passing translation and additionally Equation 5 that using temporary storage is unobservable. The acceptor-passing and contination-passing translations for **reduce** both rely on Equation 6 establishing the correctness of **reduceI**.

The acceptor-passing translations of the data layout combinators rely on the corresponding properties for **zip**, **split**, **join** and **pair**. The acceptor-passing cases for **fst** and **snd** follow from the induction hypothesis and β-equality. The correctness of the contination-passing translations for the data layout combinators also follow by applying the induction hypothesis and using β-equality. □

6 FROM DATA PARALLEL IDEALISED ALGOL TO OPENCL

In Section 4 we discussed the translation of higher-order functional array programs to imperative combinators **mapI** and **reduceI** which are then expanded into for-loops by substitution. In Section 5 we have shown that this translation is semantics preserving. In this section, we discuss the process of OpenCL code generation. OpenCL (Khronos OpenCL Working Group 2012) is the leading standard for programming GPUs and accelerators. These devices offer tremendous compute power for many application domains, including mathematical finance and deep learning, which makes GPUs an important hardware target.

6.1 A Short Introduction to OpenCL

The OpenCL programming model distinguishes between the managing *host program* and the *kernel programs* which are executed on parallel on an OpenCL enabled *device*. Kernel programs are special functions written in the OpenCL C programming language which is a dialect of C with parallel-specific restrictions and extensions. Our work focuses purely on the generation of the OpenCL kernel. A kernel function is executed in parallel on an OpenCL device by multiple *work-items* (threads) which can optionally be organised in *work-groups*. Each work-item is uniquely identified by a *global id*, or a combination of a *group id* and a *local id* internal to the group. These ids are used to determine which part of the data is accessed by each threads.

OpenCL also defines different *memory spaces* which correspond to memories with distinct performance characteristics. The *global memory* is visible by all the threads and is usually the largest, but also the slowest memory on an OpenCL device. The *local memory* is shared among the work-items of a work-group and is order of magnitudes faster than global memory (comparable to cache performance). Finally, the *private memory* is the fastest memory, but very small and can not be used for data shared among work-items (private memory usually corresponds to registers).

On some architectures vector instructions are crucial for achieving high performance. OpenCL supports special *vector types* such as `float4` where operations on a value of this type are performed by the vector units in the processor. Vector types are only available for a small number of underlying numerical data types (*e.g.*, `int`, `float`) and a fixed number of sizes: 2, 3, 4, 8, and 16.

6.2 OpenCL Specific Data Parallel Programming Primitives

Following the work of Steuwer et al. (2015) we have designed a set of parallel programming primitives reflecting the OpenCL programming model in an extension of DPIA. Their work has shown that the design presented below allows generation of efficient OpenCL code with performance comparable to expert written code.

Parallelism Hierarchy. To exploit the different parallelism levels of the OpenCL thread hierarchy with global work-items, local work-items organised in work-groups, and sequential execution inside a single work-item, we introduce four variants of the functional **map** primitive, all with the same type as the original:

$$\left.\begin{array}{l}\textbf{mapGlobal}\\ \textbf{mapWorkgroup}\\ \textbf{mapLocal}\\ \textbf{mapSeq}\end{array}\right\} : (n : \mathsf{nat}) \to (\delta_1\ \delta_2 : \mathsf{data}) \to (\mathsf{exp}[\delta_1] \to \mathsf{exp}[\delta_2]) \to \mathsf{exp}[n.\delta_1] \to \mathsf{exp}[n.\delta_2]$$

We also add four corresponding intermediate imperative combinators, specialising the **mapI** used above:

$$\left.\begin{array}{l}\textbf{mapIGlobal}\\ \textbf{mapIWorkgroup}\\ \textbf{mapILocal}\\ \textbf{mapISeq}\end{array}\right\} : (n : \mathsf{nat}) \to (\delta_1\ \delta_2 : \mathsf{data}) \to (\mathsf{exp}[\delta_1] \to \mathsf{acc}[\delta_2] \to_{\mathrm{p}} \mathsf{comm}) \to \mathsf{exp}[n.\delta_1] \to \mathsf{acc}[n.\delta_2] \to \mathsf{comm}$$

Finally, we add three OpenCL-specific variations of the **parfor** imperative primitive:

$$\left.\begin{array}{l}\textbf{parforGlobal}\\ \textbf{parforWorkgroup}\\ \textbf{parforLocal}\end{array}\right\} : (n : \mathsf{nat}) \to (\delta : \mathsf{data}) \to \mathsf{acc}[n.\delta] \to (\mathsf{exp}[\mathsf{idx}(n)] \to \mathsf{acc}[\delta] \to_{\mathrm{p}} \mathsf{comm}) \to \mathsf{comm}$$

We reuse the sequential **for** primitive for the translation of **mapSeq**. The specification of the translation of the specialised **map**$*$ functional primitives down to the corresponding variations of **parfor** via their intermediate imperative counterparts is defined exactly as for the **map** → **mapI** → **parfor** translation in Section 4. Semantically, all these variants of **map** and **parfor** are equivalent to the originals, so the correctness proof in Section 5 is unaffected. The additional information present in the names is only used by the OpenCL code generator. However, in future work, we want to formalise the OpenCL model to ensure by construction that we always generate valid OpenCL kernels that respect the parallelism hierarchy.

Address Spaces. To account for OpenCL's multiple address spaces, we add three primitives which wrap a function, i. e., take a function as its argument and return a function of the same type:

$$\textbf{toGlobal}, \textbf{toLocal}, \textbf{toPrivate} : (\delta_1\ \delta_2 : \mathsf{data}) \to (\mathsf{exp}[\delta_1] \to \mathsf{exp}[\delta_2]) \to \mathsf{exp}[\delta_1] \to \mathsf{exp}[\delta_2]$$

Semantically, these functions are all the identity. As above, the additional information is only used by the OpenCL code generator. During the translation these functions are replaced by specialised **new** primitives parameterised with the OpenCL memory space and perform the memory allocation in the indicated memory space:

$$\textbf{newGlobal}, \textbf{newLocal}, \textbf{newPrivate} : (\delta : \mathsf{data}) \to (\mathsf{var}[\delta] \to \mathsf{comm}) \to \mathsf{comm}$$

By default, **map** allocates memory in global memory for its output during the continuation-passing translation. When **map** is wrapped in, e.g., **toLocal** this will perform the memory allocation and trigger the acceptor-passing translation of **map** where it does not allocate memory itself, but rather writes to the provided acceptor.

As for the parallelism hierarchy, in future work we plan to extend our formal treatment to include the OpenCL memory model and track address space use with an effect system. This will allow us to ensure that the address spaces are only used correctly.

Vectorisation. To support the OpenCL vector types we extended DPIA's type system with an additional vector data type. This is defined similar to the array data type, but more restricted so that the element data type has to be num and the length must be one of the legal choices defined by OpenCL. Arrays of non-vector type can be turned into an array of vector type using the **asVector** primitive which behaves similar to the **split** primitive:

$$\textbf{asVector}_{\underline{n}} : (m : \mathsf{nat}) \to (\delta : \mathsf{data}) \to \mathsf{exp}[m\underline{n}.\delta] \to \mathsf{exp}[m.\mathsf{num}\langle\underline{n}\rangle] \quad (\text{where } \mathsf{num}\langle\underline{n}\rangle \text{ is a vector type})$$

Similarly to **join** which flattens a two dimensional array, **asScalar** turns an array of vector type into an array of non-vector type:

$$\textbf{asScalar}_{\underline{n}} : (m : \mathsf{nat}) \to (\delta : \mathsf{data}) \to \mathsf{exp}[m.\mathsf{num}\langle\underline{n}\rangle] \to \mathsf{exp}[m\underline{n}.\delta] \quad (\text{where } \mathsf{num}\langle\underline{n}\rangle \text{ is a vector type})$$

6.3 Translating Dot-product to OpenCL

We pick up the dot product example (2) given in Section 2 to show how a mild variation which makes use of the OpenCL-specific primitives is translated to real OpenCL. The example shown here uses the **mapWorkgroup** and **mapLocal** primitives together with the vectorisation primitives **asVector** and **asScalar**.

$$\textbf{asScalar}_4\ (\textbf{join}\ (\textbf{mapWorkgroup}\ (\lambda zs_1.\ \textbf{mapLocal}\ (\lambda zs_2.\ \textbf{reduce}\ (\lambda x\ a.\ (\textbf{fst}\ x * \textbf{snd}\ x) + a)\ 0\ (\textbf{split}\ 8192\ zs_2))\ zs_1)\ (\textbf{split}\ 8192\ (\textbf{zip}\ (\textbf{asVector}_4\ xs)\ (\textbf{asVector}_4\ ys))))))$$

This is the code used in the experimental evaluation (Section 7) and shows excellent performance on an Intel CPUs compared to the reference MKL implementation. Vectorisation is crucial on Intel CPUs for achieving high performance.

This purely functional program with OpenCL-specific primitives is translated to the following imperative program. The translation largely follows the steps explained in Section 4 extended to cover the OpenCL-specific primitives, as explained above.

parforWorkgroup $(N/8192)$ (**joinAcc** $(N/8192)$ 64 (**asScalarAcc**$_4$ $(N/128)$ *out*)) (λ *gid o*.
  parforLocal 64 *o* (λ *lid o*.
    newPrivate num$\langle 4\rangle$ *accum*.
      accum.1 := 0;
      for 2048 (λ *i*.
        accum.1 := *accum*.2 +
          (**fst** (**idx** (**idx** (**split** 2048 (**idx** (**split** $(8192 * 4)$ (**zip** (**asVector**$_4$ *xs*) (**asVector**$_4$ *ys*))) *gid*)) *lid*) *i*)) $*$
          (**snd** (**idx** (**idx** (**split** 2048 (**idx** (**split** $(8192 * 4)$ (**zip** (**asVector**$_4$ *xs*) (**asVector**$_4$ *ys*))) *gid*)) *lid*) *i*)));
      out := *accum.2*))

We generate the following OpenCL kernel where each line corresponds to a line of the imperative DPIA program.

```
1  kernel void KERNEL(global float *out, const global float *restrict xs,
2                     const global float *restrict ys, int N) {
3    for (int g_id = get_group_id(0); g_id < N / 8192; g_id += get_num_groups(0)) {
4      for (int l_id = get_local_id(0); l_id < 64; l_id += get_local_size(0)) {
5        float4 accum;
6        accum = (float4)(0.0, 0.0, 0.0, 0.0);
7        for (int i = 0; i < 2048; i += 1) {
8          accum = (accum +
9                   (vload4(((2048 * l_id) + (8192 * 4 * g_id) + i), xs) *
10                   vload4(((2048 * l_id) + (8192 * 4 * g_id) + i), ys))); }
11       vstore4(accum, ((64 * g_id) + l_id), out); } } }
```

The **parforWorkgroup** and **parforLocal** primitives have been translated into **for** loops in line 3 and 4 which use the OpenCL functions `get_group_id` and `get_local_id` for distributing iterations across parallel executing work-groups and work-items. Loading elements as vector data types from the `float` arrays `xs` and `ys` requires using the OpenCL provided function `vload4` in lines 9 and 10. Similarly, storing the computed value with vector data type in the output array uses the `vstore4` function in line 11.

6.4 Memory allocation in Data Parallel Idealised Algol for OpenCL

Our translation from functional to imperative programs leaves us with programs which perform statically bounded memory allocation. The lifetime of every memory allocation is known because it is bounded by the scope of the **new** primitive. Nevertheless, the memory allocation occurs dynamically as part of the execution of the program. In C these allocations can be performed with `malloc` on the heap or `alloca` on the stack. However, OpenCL does not support dynamic memory allocation. Furthermore, OpenCL demands that all temporary buffers in global and local memory – even with statically known size – have to be allocated prior to the kernel execution

and passed as pointers to the kernel function. In order to generate valid OpenCL, we perform an additional translation step to hoist all **newGlobal** and **newLocal** primitives to the very top of the program where we will eventually turn them into kernel arguments. **new** primitives can be nested inside parallel for loops, so when hoisting memory allocations out of the loop the amount of memory has to be multiplied by the number of loop iterations, so that every loop iteration has its distinct location to write to.

To hoist the allocations we traverse the imperative program and for each parallel for loop we encounter we remember the number of iterations and the loop variable. Once we reach a **newGlobal** or **newLocal** primitive, we replace it with its body and substitute the appropriate acceptor-expression pair for its variable that correctly points to the right place in the globally allocated data structure.

The following imperative DPIA program implements dot-product with two memory allocations nested in the **parforGlobal** loop. The allocation in global memory has to be hoisted out while the nested allocation in private memory (**newPrivate**) is permitted in OpenCL, and will translate to the allocation of a scalar stack variable.

parforGlobal *n out* ($\lambda i\ o$.
  newGlobal 1024.num *tmp*.
    for 1024 (λj. **idx** *tmp*.1 *j* := (**idx** (**idx** (**split** 1024 *xs*) *i*) *j*) $*$ (**idx** (**idx** (**split** 1024 *ys*) *i*) *j*));
    newPrivate num *accum*.
      accum.1 := 0; **for** 1024 (λj. *accum*.1 := *accum*.2 + (**idx** *tmp*.2 *j*)); *out* := *accum*.2)

To hoist out the allocation in global memory we first visit **parforGlobal**, remember the number of iterations *n* and the loop variable *i*. Then, we replace the **newGlobal** with its body in which we have replaced *tmp* with (**idx** *tmp*′ *i*). We indicate the places where uses of *tmp* have been replaced by shaded backgrounds:

newGlobal ($n \times 1024$).num *tmp*′.
  parforGlobal *n out* ($\lambda i\ o$.
    for 1024 (λj. **idx** (**idx** *tmp*′.1 *i*) *j* := (**idx** (**idx** (**split** 1024 *xs*) *i*) *j*) $*$ (**idx** (**idx** (**split** 1024 *ys*) *i*) *j*));
    newPrivate num *accum*.
      accum.1 := 0; **for** 1024 (λj. *accum*.1 := *accum*.2 + (**idx** (**idx** *tmp*′.2 *i*) *j*)); *out* := *accum*.2)

A **newGlobal** primitive is introduced at the very top of the program with the adjusted type.

7 EXPERIMENTAL RESULTS

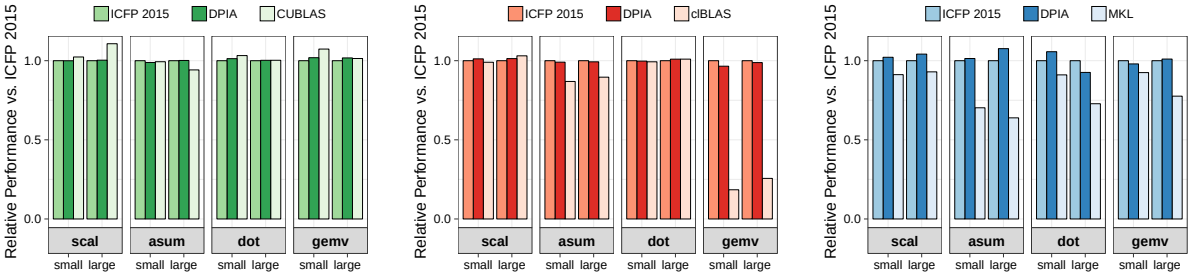
This section evaluates the quality of the OpenCL code generated from DPIA following the translation described in Section 4. We are interested to see if this formal translation introduces overheads compared to manual written OpenCL code and to the informal translations from functional programs to OpenCL used in (Steuwer et al. 2015), where semantic preserving rewrite rules were used purely at the functional level to explore different implementations. We start by describing our experimental setup and the benchmarks used.

7.1 Experimental Setup

With the help of the original authors, (Steuwer et al. 2015), we reproduced their results using the same methodology as them. We used three three different OpenCL platforms: 1) an Nvidia GeForce GTX TITAN X with CUDA 8 and driver 375.26 installed; 2) an AMD Radeon HD 7970 GPU with AMD-APP 3.0 and driver 15.300 installed; 3) an Intel Xeon E5530 CPU with 8 physical cores distributed across two sockets and hyper-threading enabled.

We used the same set of benchmarks with two input sizes. For *scal*, *asum*, and *dot*, we used vectors of 16 (small) and 128 (large) millions elements. For *gemv*, input matrices of 4096^2 (small) and 8192^2 (large) elements were used.

We used the OpenCL profiling API for measuring OpenCL kernel runtime and the CPU runtime was measured using the *gettimeofday* function. We did not measure data transfer time, as we were only interested in the quality of the generated OpenCL kernel. Each experiment was repeated 1000 times and we report median runtimes. We compare against the manually written and optimised code from the vendor-provided libraries: CUBLAS version 8.0 from Nvidia, clBLAS version 2.12 from AMD, and MKL version 11.1 from Intel.



(a) Nvidia GPU (b) AMD GPU (c) Intel CPU

Fig. 7. Performance comparison of code compiled via the formal translation from DPIA to OpenCL vs. informal translation of ICFP 2015 (cf. Steuwer et al. 2015) and vs. platform-specific libraries. The formal translation from DPIA to OpenCL introduces no performance overhead compared to ICFP 2015 and matches or outperforms highly tuned libraries on all three platforms.

7.2 Overhead of Formal Translation

Figure 7 shows the runtime performance of the OpenCL kernels generated via the formal translation described in Section 4. The graphs are normalised by the performance of the OpenCL code generated from the technique described by Steuwer et al. (2015) (labelled ICFP 2015). Bars lower than 1.0 indicate a performance loss and bars higher than 1.0 a performance gain.

The performance of the OpenCL code generated by the method of Steuwer et al. and the code generated from DPIA is almost identical in all cases with less than 5% difference. This demonstrates that our formal translation process does not introduce significant overheads.

7.3 Performance Comparison vs. Platform-Specific Libraries

The performance results comparing DPIA generated OpenCL kernels against platform-specific libraries provided by Nvidia, AMD, and Intel show that for most benchmarks and input sizes the generated code matches the library performance. For some cases, such as gemv on AMD or asum on Intel we even clearly outperform the library implementations by a factor of up to five times. These performance results are similar to the results published by Steuwer et al. (2015) and show that by exploring parallelisation strategies using semantics preserving rewrite rules it is possible to outperform manually written code. In this paper, we have extended the formal rewriting from the purely functional to the imperative level while achieving the same impressive performance results.

8 RELATED WORK

Idealised Algol and Syntactic Control of Interference. We have heavily relied on Reynolds' insightful design of Idealised Algol (IA) in this work, originally spelt out in (Reynolds 1997). IA's orthogonal combination of typed λ-calculus and imperative programming has given us the ideal language in which to formalise compilation from functional to imperative code. Moreover, Reynolds' Syntactic Control of Interference (SCI) (Reynolds 1978) enabled us to ensure that we always produce deterministic data race free programs. Brookes describes a concurrent version of Idealised Algol (Brookes 2002) that he calls "parallel", but is intentionally a non-deterministic concurrent language that allows threads to communicate through shared memory.

Reynolds presents SCI as a series of principles for a language design, which are formulated as a substructural type system by O'Hearn et al. (1999). We have used O'Hearn et al.'s formulation in our design of DPIA. We do not know of any other work using IA or SCI as an intermediate language for compilation, although Ghica has used interference controlled Idealised Algol as a high-level language for hardware synthesis (Ghica 2007).

Reynolds' original presentation of IA describes its semantics in terms of a functor categories (Oles 1982). This semantics models the stack discipline of IA (which Reynolds uses to systematically derive a compilation strategy for IA (Reynolds 1995)), but does not model non-interference or the locality of fresh state, both of which we rely upon in Section 5. O'Hearn and Tennent were the first to use relational parametricity to model locality (O'Hearn and Tennent 1995). Models of non-interference are given by O'Hearn (1993),Tennent (1990), and, O'Hearn and Tennent (1993), refining the original Reynolds/Oles functor category semantics. The semantic insights in this work later fed into Separation Logic (Ishtiaq and O'Hearn 2001). Reddy's object-space semantics (Reddy 1994, 1996) also modelled non-interference, but took an intensional viewpoint based on modelling interactions with the store, rather than transformations of the store. Reddy later synthesised the relationally parametric and intensional approaches in his automata-theoretic model (Reddy 2013), which we used in Section 5.

Functional Compilation Approaches Targeting Heterogeneous Architectures. There exist multiple functional approaches for generating code for heterogeneous hardware. Steuwer et al. (2015) use a data parallel language similar to the functional subset of DPIA. Semantics preserving rewrite rules are used to explore the space of possible implementations showing that achieving high performance across multiple architectures from functional code is possible. Obsidian (Svensson et al. 2016) is a functional low-level GPU programming language which gives programmers flexibility over how to write efficient GPU code while still providing functional abstractions. Accelerate is a Haskell embedded domain specific language providing higher level abstractions with the aim of generating efficient GPU code (Chakravarty et al. 2011; McDonell et al. 2013). LiquidMetal (Dubach et al. 2012) targets Field-Programmable Gate Array (FPGAs) and GPUs by extending Java with a data-flow programming model with built-in functional map and reduce operators. Bergstrom and Reppy (2012) compile Nesl, which is a first-order dialect of ML supporting nested data-parallelism and introduced by Blelloch (1993), to GPU code. Nvidia implement NOVA (Collins et al. 2014), a functional language targeted at code generation for GPUs, and Copperhead (Catanzaro et al. 2011), a data parallel language for GPUs embedded in Python. Delite (Sujeeth et al. 2014) is a system that enables the creation of domain-specific languages using functional parallel patterns and targets multi-core CPUs and GPUs.

Our work is the first to formally explain the translation of a functional data parallel language to imperative code and to demonstrate that this does not introduce overhead compared to the existing state-of-the-art.

9 CONCLUSIONS AND FUTURE WORK

This paper has introduced the new data parallel programming language DPIA, an interference controlled dialect of Idealised Algol. We showed how this language is used as a foundation for a formal translation of data parallel functional programs to imperative code for parallel machines. This approach offers strong guarantees about the absence of data-races in the generated programs and offers a straightforward translation strategy for generating efficient OpenCL code. Moreover, our approach is predictable and parallelism strategy preserving, allowing us to reuse Steuwer et al. (2015)'s automatic functional approach to generating high performance parallel strategies. Our experimental results show that this formalised approach is able to produce high performance code on a par with existing ad hoc techniques without introducing any overheads.

Although we have captured aspects of the OpenCL parallelism and memory hierarchies in DPIA (as described in Section 6, we are not guaranteed by construction to have generated legal programs that respect the hierarchy. For example, nesting a **mapWorkgroup** inside a **mapLocal** should not be permitted. In future work, we intend to capture this kind of "hardware paradigm" information in the type system of DPIA. The applicability of such a system goes beyond OpenCL. Heterogeneous parallel architectures abound: from data centre sized clusters of machines to FPGAs. A language based approach to describing such hierarchies will provide a framework for building robust strategy preserving functional to imperative compilation methods. We also plan to push our formalisation future to prove correctness of the final translation from DPIA to C and OpenCL.

REFERENCES

Andrew Barber. 1996. *Dual Intuitionistic Linear Logic.* Technical Report ECS-LFCS-96-347. LFCS, University of Edinburgh.

Lars Bergstrom and John H. Reppy. 2012. Nested data-parallelism on the gpu. In *ICFP*. ACM, 247–258.

Guy E. Blelloch. 1993. *NESL: A nested data-parallel language (version 2.6).* Technical Report CMU-CS-93-129. School of Computer Science, Carnegie Mellon University.

Stephen D. Brookes. 2002. The Essence of Parallel Algol. *Inf. Comput.* 179, 1 (2002), 118–149.

Bryan Catanzaro, Michael Garland, and Kurt Keutzer. 2011. Copperhead: Compiling an Embedded Data Parallel Language *(PPoPP)*. ACM, 10.

Manuel M. T. Chakravarty, Gabriele Keller, Sean Lee, Trevor L. McDonell, and Vinod Grover. 2011. Accelerating Haskell array codes with multicore GPUs. In *DAMP*. ACM, 3–14.

Alexander Collins, Dominik Grewe, Vinod Grover, Sean Lee, and Adriana Susnea. 2014. NOVA: A Functional Language for Data Parallelism. In *ARRAY@PLDI*. ACM, 8–13.

Olivier Danvy, Kevin Millikin, and Lasse R. Nielsen. 2007. On one-pass CPS transformations. *J. Funct. Program.* 17, 6 (2007), 793–812.

Christophe Dubach, Perry Cheng, Rodric Rabbah, David F. Bacon, and Stephen J. Fink. 2012. Compiling a High-level Language for GPUs: (via Language Support for Architectures and Compilers) *(PLDI)*. ACM, 12.

G. Gentzen. 1935. Untersuchungen über das logische Schließen. I. *Mathematische zeitschrift* 39, 1 (1935).

Dan R. Ghica. 2007. Geometry of synthesis: a structured approach to VLSI design. In *POPL*. ACM, 363–375.

Jean-Yves Girard. 1987. Linear Logic. *Theor. Comput. Sci.* 50 (1987), 1–102.

Samin S. Ishtiaq and Peter W. O'Hearn. 2001. BI as an Assertion Language for Mutable Data Structures. In *POPL*. ACM, 14–26.

Khronos OpenCL Working Group. 2012. The OpenCL Specification. (2012). Version 1.2.

Trevor L. McDonell, Manuel M. T. Chakravarty, Gabriele Keller, and Ben Lippmeier. 2013. Optimising purely functional GPU programs. In *ICFP*. ACM, 49–60.

Shayan Najd, Sam Lindley, Josef Svenningsson, and Philip Wadler. 2016. Everything old is new again: quoted domain-specific languages. In *PEPM*. ACM, 25–36.

Peter W. O'Hearn. 1993. A Model for Syntactic Control of Interference. *Mathematical Structures in Computer Science* 3, 4 (1993), 435–465.

Peter W. O'Hearn, John Power, Makoto Takeyama, and Robert D. Tennent. 1999. Syntactic Control of Interference Revisited. *Theor. Comput. Sci.* 228, 1-2 (1999), 211–252.

Peter W. O'Hearn and Robert D. Tennent. 1993. Semantical Analysis of Specification Logic, 2. *Inf. Comput.* 107, 1 (1993), 25–57.

Peter W. O'Hearn and Robert D. Tennent. 1995. Parametricity and Local Variables. *J. ACM* 42, 3 (1995), 658–709.

Frank J. Oles. 1982. *A Category Theoretic Approach To Semantics of Programming Languages.* Ph.D. Dissertation. Syracuse University, Syracuse, New York.

D. Prawitz. 1965. *Natural Deduction: A Proof-Theoretical Study.* Almqvist and Wiksell.

Uday Reddy. 1994. Passivity and independence. In *Proceedings of the Ninth Annual IEEE Symp. on Logic in Computer Science, LICS 1994*, Samson Abramsky (Ed.). IEEE Computer Society Press, 342–352.

Uday S. Reddy. 1996. Global State Considered Unnecessary: An Introduction to Object-Based Semantics. *Lisp and Symbolic Computation* 9, 1 (1996), 7–76.

Uday S. Reddy. 2013. Automata-Theoretic Semantics of Idealized Algol with Passive Expressions. *Electr. Notes Theor. Comput. Sci.* 298 (2013), 325–348.

John C. Reynolds. 1978. Syntactic Control of Interference. In *POPL*, Alfred V. Aho, Stephen N. Zilles, and Thomas G. Szymanski (Eds.). ACM Press, 39–46.

J. C. Reynolds. 1983. Types, Abstraction and Parametric Polymorphism. In *Proc. IFIP 9th World Computer Congress (Information Processing)*, Vol. 83. 513–523.

John C. Reynolds. 1995. Using Functor Categories to Generate Intermediate Code. In *POPL*. ACM Press, 25–36.

John C. Reynolds. 1997. The Essence of Algol. In *Algol-like Languages*, Peter W. O'Hearn and Robert D. Tennent (Eds.). Birkhäuser Boston, 67–88.

John C. Reynolds. 1998. Definitional Interpreters for Higher-Order Programming Languages. *Higher-Order and Symbolic Computation* 11, 4 (1998), 363–397.

G. .L. Steele. 1978. *Rabbit: A Compiler for Scheme.* Master's thesis. MIT.

Michel Steuwer, Christian Fensch, Sam Lindley, and Christophe Dubach. 2015. Generating performance portable code using rewrite rules: from high-level functional expressions to high-performance OpenCL code *(ICFP 2015)*. ACM, 205–217.

Michel Steuwer, Toomas Remmelg, and Christophe Dubach. 2017. Lift: A Functional Data-Parallel IR for High-Performance GPU Code Generation *(CGO 2017)*. IEEE, 74–85.

Christopher Strachey. 2000. Fundamental Concepts in Programming Languages. *Higher-Order and Symbolic Comp.* 13, 1/2 (2000), 11–49.

Arvind K. Sujeeth, Kevin J. Brown, Hyoukjoong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. 2014. Delite: A Compiler Architecture for Performance-Oriented Embedded Domain-Specific Languages. *ACM TECS* 13, 4s, Article 134 (2014), 25 pages.

Bo Joel Svensson, Ryan R. Newton, and Mary Sheeran. 2016. A language for hierarchical data parallel design-space exploration on GPUs. *J. Funct. Program.* 26 (2016), e6.

Robert D. Tennent. 1990. Semantical Analysis of Specification Logic. *Inf. Comput.* 85, 2 (1990), 135–162.