Data Parallel Idealised Algol

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

Parallel hardware has the potential for accelerating a wide class of algorithms. However, expert programming skills are required to achieve maximum performance. Typical parallel architectures expose low-level hardware details through imperative programming interfaces where programmers explicitly encode device-specific optimisation strategies. This inevitably results in non-performance-portable programs that deliver suboptimal performance on other devices.

Functional programming models have recently seen a renaissance in the systems community as they offer solutions for tackling the performance-portability challenge. Recent work has demonstrated how to automatically choose high-performance parallelisation strategies for a wide range of hardware architectures encoded in a functional representation. However, the translation of such functional representations to the imperative program demanded by the hardware interface is typically performed ad hoc with no correctness guarantees and no guarantee to preserve the intended parallelisation strategy.

We introduce Data Parallel Idealised Algol (DPIA), a parallel dialect of Reynolds' Idealised Algol, which supports both high-level functional code and low-level imperative code. We define and prove correct a translation from parallel high-level functional code to low-level data-race-free imperative code. Our performance results show that the translation yields performance on a par with hand-written low-level code.

CCS Concepts: • **Software and its engineering** → *Parallel programming languages*; *Functional languages*; *Imperative languages*; *Multiparadigm languages*;

ACM Reference format:
Robert Atkey, Michel Steuwer, Sam Lindley, and Christophe Dubach. 2018. Data Parallel Idealised Algol. *Proc. ACM Program. Lang.* 0, 0, Article 0 (October 2018), 27 pages.
https://doi.org/Unpublished

1 INTRODUCTION

Modern parallel hardware promises unprecedented performance, but only for those who can program it correctly. A single program must simultaneously describe the functional process for generating the result, as well as how to exploit the hardware's resources to deliver it. These programs are commonly written in imperative languages like OpenMP, OpenCL, and CUDA, which are difficult to reason about, and difficult to refactor to try different parallelisation strategies.

To mitigate this problem, there has been an old, but now accelerating, trend towards functional languages for parallel computation. The programmer specifies the program they want in a high level language, using higher order functions and compositional abstractions. The compiler then takes responsibility for generating efficient low level imperative code. 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. Since the "Sufficiently Smart Compiler" that always produces the highest performance code is yet to be fully realised, many modern systems have ways of describing the parallelisation strategy to be used either alongside or within the functional code. Such a strategy describes matters such as when and where parallelism ought to be used, and how

2018. 2475-1421/2018/10-ART0 $X
https://doi.org/Unpublished

data ought to be chunked for parallel processing. Ideally, we separate the concerns of functional behaviour from strategy and reason about each separately. Halide [Ragan-Kelley et al. 2018] is an example of a system where the strategy is given "on the side". Lift [Steuwer et al. 2015, 2017] and parallel Haskell [Trinder et al. 1998] are examples of systems where strategies are embedded within the functional code. The latter approach has the advantage that we cannot write strategies that do not make sense for the computation that we wish to perform. We will primarily concern ourselves with this embedded strategy approach in this paper.

Given such a strategic functional programming system, the question is how to compile the combination down to the imperative code required by the hardware, preserving the given strategy, and maintaining correctness. In previous works, this question has been treated as a mere matter of code generation from the high level functional code. However, code generation can be subtle. It is easy to formulate code generation procedures that do not work for all inputs, or produce code that does not compile, or has race conditions. Moreover, it would be beneficial to know that the code generation process not only produces functionally correct code, but also strategically correct code.

In this paper, we formalise a *semantics and strategy preserving translation* from functional to 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 into OpenMP annotated C code that can be executed in parallel (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 use of 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 generality of our approach in Section 6 where we augment the basic design of DPIA with vectorisation and double buffering support. Our primary conceptual contribution is to promote Idealised Algol and its variants as expressive and flexible intermediate languages for compilation to low level imperative code that efficiently and correctly use the facilities provided by the hardware. In particular, syntactic interference control allows us to given types to the primitive hardware operations that stops us from incorrectly applying them. We explain the process of designing new primitives for DPIA in Section 3, when we introduce the type system.

Contributions. This paper makes three major contributions:

(1) We introduce *Data Parallel Idealised Algol* (DPIA). This is an extension of O'Hearn et al.'s [1999] *Syntactic Control of Inteference Revisited* with indexed types, array and tuple data types, and primitives for data parallel programming. These extensions are fundamental for data parallel imperative programs. We describe DPIA in Section 3.
(2) 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 functionally correct in Section 5.
(3) We have implemented our framework using OpenMP as the backend and evaluate the performed achieved by our generated code in Section 7.
(4) To demonstrate the flexibility of the DPIA framework, we extend our whole approach in Section 6 to include vectorisation and iterative computations implemented by double buffering.

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 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:

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

This expression is a declarative specification of the dot product which does not specify how the computation should be performed. There are many possible strategies to execute the dot product computation in parallel. A naive strategy is to first perform the pairwise multiplication of xs and ys in parallel before sequentially summing up the intermediate results. We rewrite our expression to make this "parallel map, sequential reduce" strategy explicit:

$$\textbf{reduceSeq}\ (+)\ 0\ (\textbf{mapPar}\ (\lambda x.\ \textbf{fst}\ x * \textbf{snd}\ x)\ (\textbf{zip}\ xs\ ys)) \tag{2}$$

Such a naive strategy is not always best. If we try to execute one parallel job per element of the input arrays, the runtime might generate so many threads that coordination of them will dominate the runtime. The overall strategy of "parallel, then sequential" is likely not the most efficient, either.

We can give a more refined strategy better exploiting the target architecture. For instance, modern CPUs support SIMD vector instructions which perform arithmetic operations in parallel on multiple scalar values in a single clock cycle. We would also like to trade off the amount of work that every thread performs with the number of threads launched to best exploit the available hardware cores while minimising the coordination overhead between the threads. The following expression groups the input data as vectors of 4 scalar elements before distributing the work across multiple threads using **mapPar**. Each thread then performs multiple sequential reductions in parallel on the vectorised data using SIMD multiplication and addition instructions, before a final sequential reduction adds up all remaining results.

$$\begin{aligned}&\textbf{reduceSeq}\ (+)\ 0\ (\textbf{asScalar}\ (\textbf{join}\ (\textbf{mapPar}\\ &\qquad(\textbf{mapSeq}\ (\lambda zs.\ \textbf{reduceSeq}\ (\lambda x\ a.\ (\textbf{fst}\ x * \textbf{snd}\ x) + a)\ 0\ (\textbf{split}\ 2048\ zs)))\\ &\qquad(\textbf{split}\ (2048 * 64)\ (\textbf{zip}\ (\textbf{asVector}\ 4\ xs)\ (\textbf{asVector}\ 4\ ys)))\ )))\end{aligned} \tag{3}$$

Although this expression gives much more information about how to perform the computation in parallel, we have not left the functional paradigm. **mapPar**, **mapSeq**, and **reduceSeq** are just annotated variations of the standard **map** and **reduce** functions which still allows the straightforward mathematical reading of this expression. We can use equational reasoning to prove that this expression is semantically equivalent to (1). Equational reasoning can even be used to generate (2) and (3) from (1). Indeed Steuwer et al. [2016] 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 OpenMP or OpenCL 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 naive "parallel map, sequential reduce" strategy of dot-product (2), we can turn it into an imperative program simply by assigning its result to an output variable *out*:

$$\mathit{out} := \mathbf{reduceSeq}\ (+)\ 0\ (\mathbf{mapPar}\ (\lambda x.\ \mathbf{fst}\ x * \mathbf{snd}\ x)\ (\mathbf{zip}\ \mathit{xs}\ \mathit{ys}))$$

Unfortunately, this is not suitable for compilation targets like OpenMP or OpenCL. 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 $\mathcal{A}(\!|E|\!)_\delta(\mathit{out})$. The key idea is that for any expression E producing data of type δ, the translation $\mathcal{A}(\!|E|\!)_\delta(\mathit{out})$ is an imperative program that has the same effect as the assignment $\mathit{out} := E$ and is free from higher-order combinators. This translation is mutually defined with a continuation passing translation $C(\!|E|\!)_\delta(C)$ 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 **reduceSeq** by a corresponding imperative combinator **reduceSeqI**. We will see below that **reduceSeqI** is straightforwardly implemented in terms of variable allocation and a for-loop.

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

The **mapPar** is translated, by the continuation passing translation, into allocation of a temporary array and an imperative **mapParI** combinator. As with **reduceSeqI**, the **mapParI** combinator is straightforwardly implementable in terms of a parallel for-loop. The operator **new** δ *var* declares a new storage cell of type δ named *var*, where storage cells are represented as pairs of an acceptor (i.e., "writer", "l-value") part *var*.1 and an expression (i.e. "reader", "r-value") part *var*.2. Our language, which we introduce in Section 3, is a variant of Reynolds' Idealised Algol [Reynolds 1978].

$$\begin{aligned}
=\ & \mathbf{new}\ (n.\mathsf{num})\ (\lambda \mathit{tmp}.\ C(\!|\mathbf{zip}\ \mathit{xs}\ \mathit{ys}|\!)_{n.(\mathsf{num}\times\mathsf{num})}(\lambda x.\ \mathbf{mapParI}\ n\ (\lambda x\ o.\ \mathcal{A}(\!|\mathbf{fst}\ x * \mathbf{snd}\ x|\!)_{\mathsf{num}}(o))\ x\ \mathit{tmp}.1);\\
& \qquad C(\!|0|\!)_{\mathsf{num}}(\lambda y.\ \mathbf{reduceSeqI}\ n\ (\lambda x\ y\ o.\ \mathcal{A}(\!|x+y|\!)_{\mathsf{num}}(o))\ y\ \mathit{tmp}.2\ (\lambda r.\ \mathcal{A}(\!|r|\!)_{\mathsf{num}}(\mathit{out}))))
\end{aligned}$$

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 compiler 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 strategy of the functional program, as seen earlier in example (3). The parallelism strategy described by the functional code here precisely states "parallel map, followed by sequential reduce".

Continuing the translation process, we substitute *out*, the arithmetic expressions and the **zip**, leaving two uses of the "intermediate-level" macros **mapParI** and **reduceSeqI**:

$$\begin{aligned}
=\ & \mathbf{new}\ (n.\mathsf{num})\ (\lambda \mathit{tmp}.\ \mathbf{mapParI}\ n\ (\lambda x\ o.\ o := \mathbf{fst}\ x * \mathbf{snd}\ x)\ (\mathbf{zip}\ \mathit{xs}\ \mathit{ys})\ \mathit{tmp}.1;\\
& \qquad \mathbf{reduceSeqI}\ n\ (\lambda x\ y\ o.\ o := x + y)\ 0\ \mathit{tmp}.2\ (\lambda r.\ \mathit{out} := r))
\end{aligned}$$

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

$$\begin{aligned}
=\ & \mathbf{new}\ (n.\mathsf{num})\ (\lambda \mathit{tmp}.\ \mathbf{parfor}\ n\ \mathit{tmp}.1\ (\lambda i\ o.\ o := \mathbf{fst}\ (\mathbf{idx}\ (\mathbf{zip}\ \mathit{xs}\ \mathit{ys})\ i) * \mathbf{snd}\ (\mathbf{idx}\ (\mathbf{zip}\ \mathit{xs}\ \mathit{ys})\ i));\\
& \qquad \mathbf{new}\ \mathsf{num}\ (\lambda \mathit{accum}.\ \mathit{accum}.1 := 0;\\
& \qquad\qquad \mathbf{for}\ n\ (\lambda i.\ \mathit{accum}.1 := \mathit{accum}.2 + \mathbf{idx}\ \mathit{tmp}.2\ i);\\
& \qquad\qquad \mathit{out} := \mathit{accum}.2))
\end{aligned} \tag{4}$$

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 C and OpenMP (Section 4.3) resolves these data layout expressions into explicit indexing computations:

```
1 float tmp[N];
2 #pragma omp parallel for
3 for (int i = 0; i < N; i += 1)
4   tmp[i] = xs[i] * ys[i];
5 float accum = 0.0;
6 for (int i = 0; i < N; i += 1)
7   accum = accum + tmp[i];
8 out = accum;
```

This resulting low level imperative code precisely implements the strategy "parallel map (lines 2–4), followed by sequential reduce (lines 5–8)" described by our functional expression (2).

Our naive dot-product code does not produce particularly complex code, but our translation method scales to more detailed parallelism strategies. The alternative dot-product execution strategy in (3), which rearranges the abstract *map* and *reduce* combinators in order to better exploit parallel hardware, yields the following code using OpenMP pragmas for thread level parallelism (`omp parallel for`) as well as for SIMD vector instructions (`omp simd`):

```
1  float tmp[N/(4*131072/64)];
2  #pragma omp parallel for
3  for (int i = 0; i < (N / (4 * 131072)); i += 1) {
4    for (int j = 0; j < 64; j += 1) {
5      float accum[4];
6      #pragma omp simd
7      for (int k = 0; k < 4; k += 1) { accum[k] = ((float[4]){0.0f})[k]; }
8      for (int l = 0; l < 2048; l += 1) {
9        #pragma omp simd
10       for (int k = 0; k < 4; k += 1) {
11         accum[k] = (xs[(524288 * i) + (8192 * j) + (4 * l) + k]
12                     * ys[(524288 * i) + (8192 * j) + (4 * l) + k]) + accum[k];}}
13     #pragma omp simd
14     for (int k = 0; k < 4; k += 1) { tmp[k + (4 * j) + (256 * i)] = accum[k];}
15   }
16 }
17 float accum = 0.0;
18 for (int i = 0; i < N/(4*131072/64); i += 1)
19   accum = accum + tmp[i];
20 out = accum;
```

As we shall see in Section 7, given a target-architecture optimised parallelisation strategy defined in functional code, our translation process produces OpenMP code with performance on a par 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 a dialect of Reynolds's [1997] *Idealised Algol*, extended with interference control via a substructural type system [O'Hearn et al. 1999]. We saw some 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), typing (Section 3.2), and data parallel programming primitives (Section 3.3). We discuss the formal semantics of DPIA in Section 5.

Idealised Algol is built around Reynolds' observation that we can use typed λ-calculus as a powerful macro language for imperative programming. The base types of the system are what would normally be the syntactic categories of a first order imperative language: comm is the type of commands, or statements; exp is the type of expressions, or r-values; and acc is the type of acceptors or l-values (entities that can be assigned to). The syntactic constructs of the language are expressed as constants in the language. For imperative programming, these include: skip : comm for doing nothing; $(;) : \mathsf{comm} \times \mathsf{comm} \to \mathsf{comm}$ for sequencing; and $(:=) : \mathsf{acc} \times \mathsf{exp} \to \mathsf{comm}$ for assignment. Declaration of new variables has a higher order type, $\mathbf{new} : (\mathsf{acc} \times \mathsf{exp} \to \mathsf{comm}) \to \mathsf{comm}$: we supply a command that requires an additional piece of storage to work with, and **new** allocates the variable, runs the command, and deallocates, following a block structured stack discipline.

The typing of new indicates a useful programming pattern in Idealised Algol. We can think of terms with types of the form $Y \to (X \to \mathsf{comm}) \to \mathsf{comm}$ as implementing "objects" with interfaces X, using existing objects with interface Y. A canonical example is the counter object:

$$\begin{aligned}&\mathsf{counter} : \mathsf{acc} \to (\mathsf{comm} \to \mathsf{comm}) \to \mathsf{comm}\\ &\mathsf{counter}\ o\ \mathit{body} = \mathbf{new}\ (\lambda\langle v_{wr}, v_{rd}\rangle.\ \mathit{body}\ (v_{wr} := v_{rd} + 1); o := v_{rd})\end{aligned}$$

Given a client, *body*, and a place to write the final count to, *o*, counter allocates a new variable using new and hands *body* the restricted capability to only affect this variable by incrementing it. Once *body* has finished, the final value of the counter is written to *o*.

This implementation of counter uses λ-abstraction to hide its internal state from *body*, preventing any programming errors that would occur if *body* were able to interfere with counter's internal state. However, in original Idealised Algol, counter can only encapsulate what it creates. It is possible for both counter and *body* to write to *o*, potentially producing confusing results. Reynolds's [1978] *Syntactic Control of Interference* proposed a discipline that prevents such interference from taking place. Under interference control, *body* and *o* may only share identifiers that are used *passively* (i.e., read only). In use, interference control acts like Rust's *borrow checker* [Klabnik and Nichols 2018]. In a term of the form counter $o\ (\lambda \mathit{inc}.\ P); Q$, the variable o may not be used in P, except through *inc*. We may say that *inc*, the interface to the counter, has borrowed the resources owned by *o*. In the continuation command Q, this borrow ends and *o*'s resources may be used directly again.

We use this *safe resource reinterpretation* pattern multiple times in DPIA to provide abstracted access to low level parallelisation capabilities. At the lowest level, we have the **parfor** primitive:

$$\mathbf{parfor} : (n : \mathsf{nat}) \to (t : \mathsf{data}) \to \mathsf{acc}[n.t] \to (\mathsf{exp}[\mathsf{idx}[n]] \to \mathsf{acc}[t] \to_{\mathrm{p}} \mathsf{comm}) \to \mathsf{comm}$$

The type of **parfor** is more elaborate than before to allow for datatype and size polymorphism, but the essence is this: the latter two of **parfor**'s arguments are an acceptor for writing results to, and a command for implementing the body of the loop. The same interference control discipline as above ensures that the body of the loop cannot write to the array chosen for output, except through the acceptor it is give, eliminating some potential data races. The p subscript on the type of the body indicates that this command must be passive, meaning that it cannot write to any state which it is not explicitly handed. Thus, it is possible to safely implement **parfor** as a parallel for loop. We use this same pattern to safely implement our intermediate level data parallel functions in Section 4.2 and our vectorisation and double buffering extensions in Section 6.

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

(a) Kinds

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

(b) Kinding Structural Rules

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

(c) Type Equality

$$\frac{}{\Delta \vdash \underline{\ell} : \text{nat}} \qquad \frac{\Delta \vdash M : \text{nat} \qquad \Delta \vdash N : \text{nat}}{\Delta \vdash M + N : \text{nat}} \qquad \frac{\Delta \vdash M : \text{nat} \qquad \Delta \vdash N : \text{nat}}{\Delta \vdash M * N : \text{nat}}$$

(d) Natural numbers

$$\frac{}{\Delta \vdash \text{num} : \text{data}} \qquad \frac{\Delta \vdash N : \text{nat}}{\Delta \vdash \text{idx}[N] : \text{data}} \qquad \frac{\Delta \vdash N : \text{nat} \quad \Delta \vdash \delta : \text{data}}{\Delta \vdash N.\delta : \text{data}} \qquad \frac{\Delta \vdash \delta_1 : \text{data} \quad \Delta \vdash \delta_2 : \text{data}}{\Delta \vdash \delta_1 \times \delta_2 : \text{data}}$$

(e) Data Types

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

$$\frac{\Delta \vdash \theta_1 : \text{phrase} \qquad \Delta \vdash \theta_2 : \text{phrase}}{\Delta \vdash \theta_1 \times \theta_2 : \text{phrase}} \qquad \frac{\Delta \vdash \theta_1 : \text{phrase} \quad \Delta \vdash \theta_2 : \text{phrase}}{\Delta \vdash \theta_1 \to \theta_2 : \text{phrase}} \qquad \frac{\Delta \vdash \theta_1 : \text{phrase} \quad \Delta \vdash \theta_2 : \text{phrase}}{\Delta \vdash \theta_1 \to_{\text{p}} \theta_2 : \text{phrase}}$$

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

(f) Phrase Types

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

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

(g) Passive Types

Fig. 1. Well-formed Types

3.1 The Types of DPIA

The type system of DPIA refines the exp, acc, comm *phrase types* described above by parameterising expressions and acceptors by *data types*, which classify data (integers, floats, arrays, *etc.*). 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 numeric literals $\underline{\ell}$, addition $M + N$, or multiplication $M * N$ (where M and N range over terms of kind nat). We write MN as syntactic sugar for $M * N$.

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 $\mathsf{id}\times[M]$, parameterised by the size of the array. There are two compound types of data. For any data type δ and natural number term M, $M.\delta$ is the data type of homogeneous arrays of δs of size M. 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". 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. We find it convenient to allow lowercase letters to range over both term variables and type variables.

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 1g 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 $\mathsf{exp}[\delta]$ 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 $\theta_1 \to_{\mathrm{p}} \theta_2$ 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:

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

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. The contexts Δ and Π allow contraction and weakening; context Γ does not.

The typing rules of DPIA are given in Figure 2. These rules define how variable phrases are formed, how parameterised and compound phrases are introduced and eliminated, and how passive

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

(a) Structural Rules

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

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

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

(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. 2. Typing Rules: Indexed Affine λ-Calculus with Passivity [O'Hearn et al. 1999]

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, and extend it in Section 6.

Figure 2a 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 2b 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 2c 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 $\text{exp}[\delta]$, 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.

3.3 Data Parallel Programming Primitives

The SCIR framework may be instantiated in arbitrary ways by specifying a collection of primitives. Here we instantiate it with typed primitive operations for data parallel programming, outlined in Figure 3. Our primitives fall into two principal categories: high-level functional primitives and low-level imperative primitives. Source programs include abstract high-level functional primitives **map** and **reduce**. Programs that are the input to our translation process are composed of concrete high-level functional primitives. All uses of **map** must be replaced by **mapSeq** (a sequential implementation of **map**) or **mapPar** (a parallel implementation of **map**), and all uses of **reduce** by **reduceSeq** (a sequential implementation of **reduce**). (An OpenCL backend might have other concrete implementations of **map** corresponding to the parallelisation hierarchy and a parallel implementation of **reduce** makes sense for some backends.) Uses of **mapPar**, **mapSeq**, and **reduceSeq** have no immediate 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 **mapParI**).

Functional Primitives. Figure 3a lists the type signatures of the primitives used for constructing purely functional data parallel programs. These fit into four groups. The first group consists of numeric literals ($\underline{\ell}$) and first-order operations on scalars (**negate**, $(+)$, $(-)$, $(*)$, $(/)$). The second group contains the abstract higher-order functional combinators for constructing array processing programs: **map** and **reduceSeq**. These have (the Idealised Algol renditions of) the standard types for map and reduce, extended with size information. The third group contains the concrete higher-order functional combinators for constructing array processing programs: **mapPar**, **mapSeq** and **reduceSeq**. The fourth 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 dimensional array, **pair** constructs a pair, **fst** and **snd** deconstruct a pair, and **idx** looks up a value in an array. All of these primitives are data type indexed and those that operate on arrays are also size indexed.

Example: dot-product. The naive strategy for 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):

$$\begin{aligned}&\textbf{reduceSeq}\ n\ \text{num}\ \text{num}\ (\lambda x\ y.\ x + y)\ \underline{0}\ (\textbf{mapPar}\ n\ (\text{num} \times \text{num})\ \text{num}\\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\lambda x.\ \textbf{fst}\ \text{num}\ \text{num}\ x * \textbf{snd}\ \text{num}\ \text{num}\ x)\\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\textbf{zip}\ n\ \text{num}\ \text{num}\ xs\ ys))\end{aligned}$$

$\underline{\ell}$: exp[num]
negate : exp[num] → exp[num]
(+, ∗, /, −) : exp[num] × exp[num] → exp[num]

map : $(n : \text{nat}) \to (s\ t : \text{data}) \to (\text{exp}[s] \to \text{exp}[t]) \to \text{exp}[n.s] \to \text{exp}[n.t]$
reduce : $(n : \text{nat}) \to (s\ t : \text{data}) \to (\text{exp}[s] \to \text{exp}[t] \to \text{exp}[t]) \to \text{exp}[t] \to \text{exp}[n.s] \to \text{exp}[t]$

mapSeq : $(n : \text{nat}) \to (s\ t : \text{data}) \to (\text{exp}[s] \to \text{exp}[t]) \to \text{exp}[n.s] \to \text{exp}[n.t]$
mapPar : $(n : \text{nat}) \to (s\ t : \text{data}) \to (\text{exp}[s] \to \text{exp}[t]) \to \text{exp}[n.s] \to \text{exp}[n.t]$
reduceSeq : $(n : \text{nat}) \to (s\ t : \text{data}) \to (\text{exp}[s] \to \text{exp}[t] \to \text{exp}[t]) \to \text{exp}[t] \to \text{exp}[n.s] \to \text{exp}[t]$

zip : $(n : \text{nat}) \to (s\ t : \text{data}) \to \text{exp}[n.s] \to \text{exp}[n.t] \to \text{exp}[n.(s \times t)]$
split : $(n\ m : \text{nat}) \to (t : \text{data}) \to \text{exp}[nm.t] \to \text{exp}[m.n.t]$
join : $(n\ m : \text{nat}) \to (t : \text{data}) \to \text{exp}[n.m.t] \to \text{exp}[nm.t]$
pair : $(s\ t : \text{data}) \to \text{exp}[s] \to \text{exp}[t] \to \text{exp}[s \times t]$
fst : $(s\ t : \text{data}) \to \text{exp}[s \times t] \to \text{exp}[s]$
snd : $(s\ t : \text{data}) \to \text{exp}[s \times t] \to \text{exp}[t]$
idx : $(n : \text{nat}) \to (t : \text{data}) \to \text{exp}[n.t] \to \text{exp}[\text{idx}[n]] \to \text{exp}[t]$

(a) Functional primitives

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

zipAcc$_1$: $(n : \text{nat}) \to (s\ t : \text{data}) \to \text{acc}[n.s \times t] \to \text{acc}[n.s]$
zipAcc$_2$: $(n : \text{nat}) \to (s\ t : \text{data}) \to \text{acc}[n.s \times t] \to \text{acc}[n.t]$
splitAcc : $(n\ m : \text{nat}) \to (t : \text{data}) \to \text{acc}[m.n.t] \to \text{acc}[nm.t]$
joinAcc : $(n\ m : \text{nat}) \to (t : \text{data}) \to \text{acc}[nm.t] \to \text{acc}[n.m.t]$
pairAcc$_1$: $(s\ t : \text{data}) \to \text{acc}[s \times t] \to \text{acc}[s]$
pairAcc$_2$: $(s\ t : \text{data}) \to \text{acc}[s \times t] \to \text{acc}[t]$
idxAcc : $(n : \text{nat}) \to (t : \text{data}) \to \text{acc}[n.t] \to \text{exp}[\text{idx}[n]] \to \text{acc}[t]$

(b) Imperative primitives

mapSeqI : $(n : \text{nat}) \to (s\ t : \text{data}) \to (\text{exp}[s] \to \text{acc}[t] \to \text{comm}) \to \text{exp}[n.s] \to \text{acc}[n.t] \to \text{comm}$
mapParI : $(n : \text{nat}) \to (s\ t : \text{data}) \to (\text{exp}[s] \to \text{acc}[t] \to_{\text{p}} \text{comm}) \to \text{exp}[n.s] \to \text{acc}[n.t] \to \text{comm}$
reduceSeqI : $(n : \text{nat}) \to (s\ t : \text{data}) \to (\text{exp}[s] \to \text{exp}[t] \to \text{acc}[t] \to \text{comm}) \to$
$\text{exp}[t] \to \text{exp}[n.s] \to (\text{exp}[t] \to \text{comm}) \to \text{comm}$

(c) Intermediate imperative macros

Fig. 3. Data Parallel Programming Primitives, Functional and Imperative

Likewise, the more sophisticated strategy 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 3b 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 δ A B differ from normal for-loops by taking an additional acceptor argument $A : \mathrm{acc}[B.\delta]$ that is used for the output of each iteration. As described in the introduction to this section, this allows an implementation of **parfor** that can safely be implemented using actual parallelism. We will see how the acceptor-transforming behaviour of our **parfor** primitive is translated into a normal, potentially racy, parallel for loop in OpenMP C 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 **zip**, **split**, **join**, **pair** and **idx** functional primitives. The acceptor primitives transform acceptors of compound data into acceptors of their components. They are 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 are translated into l-values with explicit index computations.

Intermediate Imperative Macros. Figure 3c gives type signatures for the intermediate imperative counterparts of **mapPar**, **mapSeq** and **reduceSeq**. These macros 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

In this section we spell out the three stages of translation from higher-order functional array programs to parallel C-like code (as sketched in Section 2). First, the higher-order functional combinators **mapPar**, **mapSeq** and **reduceSeq** are translated into the higher-order imperative macros **mapParI**, **mapSeqI** and **reduceSeqI** using an acceptor-passing translation defined mutually recursively with a continuation-passing translation (Section 4.1). Second, the higher-order imperative macros are expanded into for-loops (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.

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 $E : \mathrm{exp}[\delta]$, constructed from the functional primitives in Figure 3a, and an acceptor $\mathit{out} : \mathrm{acc}[\delta]$ and to produce a comm phrase that has the same semantics as the command

$$\mathit{out} :=_\delta E$$

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 **mapPar**, **mapSeq** and **reduceSeq** 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

$$
\begin{array}{lcl}
\mathcal{A}(\!|x|\!)_\delta(A) & = & A :=_\delta x \\
\mathcal{A}(\!|\underline{\ell}|\!)_{\mathsf{num}}(A) & = & A := \underline{\ell} \\
\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{mapPar}\ N\ \delta_1\ \delta_2\ F\ E|\!)_{N.\delta_2}(A) & = & C(\!|E|\!)_{N.\delta_1}(\lambda x.\ \mathbf{mapParI}\ N\ \delta_1\ \delta_2\ (\lambda x\ o.\mathcal{A}(\!|F\ x|\!)_{\delta_2}(o))\ x\ A) \\
\mathcal{A}(\!|\mathbf{mapSeq}\ N\ \delta_1\ \delta_2\ F\ E|\!)_{N.\delta_2}(A) & = & C(\!|E|\!)_{N.\delta_1}(\lambda x.\ \mathbf{mapSeqI}\ N\ \delta_1\ \delta_2\ (\lambda x\ o.\mathcal{A}(\!|F\ x|\!)_{\delta_2}(o))\ x\ A) \\
\mathcal{A}(\!|\mathbf{reduceSeq}\ 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{reduceSeqI}\ 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{\ell}|\!)_{\mathsf{num}}(C) & = & C(\underline{\ell}) \\
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{mapPar}\ N\ \delta_1\ \delta_2\ F\ E|\!)_{N.\delta_2}(C) & = & \mathbf{new}\ (N.\delta_2)\ (\lambda tmp.\ \mathcal{A}(\!|\mathbf{mapPar}\ N\ \delta_1\ \delta_2\ F\ E|\!)_{N.\delta_2}(tmp.2);\ C(tmp.1)\ ) \\
C(\!|\mathbf{mapSeq}\ N\ \delta_1\ \delta_2\ F\ E|\!)_{N.\delta_2}(C) & = & \mathbf{new}\ (N.\delta_2)\ (\lambda tmp.\ \mathcal{A}(\!|\mathbf{mapSeq}\ N\ \delta_1\ \delta_2\ F\ E|\!)_{N.\delta_2}(tmp.2);\ C(tmp.1)\ ) \\
C(\!|\mathbf{reduceSeq}\ 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{reduceSeqI}\ 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. 4. Acceptor and Continuation-passing Translations

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:

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

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 4a) and the continuation-passing translation $C(\!|-|\!)_\delta$ (Figure 4b). 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 avoid producing 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 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 first-order operations on numeric data. In all cases, we defer to the continuation-passing translation for sub-expressions, with appropriate continuations. The third group is the most interesting: the translations of the higher order **mapPar**, **mapSeq** and **reduceSeq** primitives. For **mapPar**/**mapSeq**: in the acceptor-passing case, we can immediately translate to **mapParI**/**mapSeq** which already takes an acceptor to place its output into; in the continuation-passing case, we must create a temporary array as storage, invoke **mapParI**/**mapSeqI**, 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 **reduceSeq**, in both cases we translate to the **reduceSeqI** combinator. The fourth group handles the translations of the functional data layout combinators. The CPS translation, passes them straight through; they are handled by the final translation to low-level C-like code in Section 4.3. The acceptor-passing translation translates the combinators that construct data into 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.

Limits of Strategy-Preservation. Our translation is almost, but not quite, strategy-preserving. There are two places where the strategy is hard-wired. First, array assignments are always computed in parallel using **mapParI**. Second, the CPS translations of **mapPar** and **mapSeq** each introduce a temporary array, and there is no choice as to where in the memory hierarchy the array is allocated. If we were targetting a heterogeneous backend such as a GPU then we would likely require more fine-grained control over these choices. We defer study of such systems as well as a formal treatment of precisely what strategy-preservation means to future work.

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

The next stage in the translation expands the intermediate level imperative macros **mapParI**, **mapSeqI** and **reduceSeqI** as lower-level implementations in terms of (parallel and sequential) for-loops. This is accomplished by substitution and β-reduction (DPIA includes full $\beta\eta$ reasoning

principles). The combinator **mapParI** is implemented as a parallel loop:

$$\textbf{mapParI} = \Lambda n\ s\ t.\lambda f\ xs\ out.\ \textbf{parfor}\ n\ t\ out\ (\lambda i\ o.\ f\ (\textbf{idx}\ n\ s\ xs\ i)\ o)$$

mapSeqI is implemented as a sequential loop:

$$\textbf{mapSeqI} = \Lambda n\ s\ t.\lambda f\ xs\ out.\ \textbf{for}\ n\ (\lambda\ i.\ f\ (\textbf{idx}\ n\ s\ xs\ i)\ (\textbf{idxAcc}\ n\ t\ out\ i)\ )$$

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:

$$\begin{aligned}\textbf{reduceSeqI} = \Lambda n\ s\ t.\lambda f\ init\ xs\ c.\ \textbf{new}\ t\ (\lambda acc.\ & acc.2 :=_s init;\\ & \textbf{for}\ n\ (\lambda i.\ f\ (\textbf{idx}\ n\ s\ xs\ i)\ (acc.2)\ (acc.1)\ );\\ & c\ (acc.1)\ )\end{aligned}$$

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 **mapPar**, **mapSeq** and **reduceSeq**, but still contains uses of the data layout combinators **zip**, **split** etc., and their acceptor counterparts. We now define a translation to OpenMP C, an extension of the C language adding parallel for loops. This translation resolves the data layout expressions into explicit indexing.

The translation in Figure 5 is split into three parts: the translation of DPIA commands into OpenMP 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 5a 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 $\textsc{CodeGen}_{\text{data}}(\delta, \mathsf{v})$ 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 by translating **new**, **for** and **parfor** are all assumed to be fresh.

The translation of acceptors (Figure 5b) 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 5b all proceed

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

(a) DPIA commands to OpenMP C statements

$$
\begin{aligned}
&\textsc{CodeGen}_{\text{acc}[\delta]}(x, \eta, ps) && = \eta(x)(\mathit{reverse}\ ps)\\
&\textsc{CodeGen}_{\text{acc}[\delta]}(\textbf{idxAcc}\ N\ \delta\ A\ I, \eta, ps) && = \textsc{CodeGen}_{\text{acc}[N.\delta]}(A, \eta,\\
&&& \quad \textsc{CodeGen}_{\text{exp[idx}[n]\text{]}}(I, \eta, []) :: ps))\\
&\textsc{CodeGen}_{\text{acc}[NM.\delta]}(\textbf{splitAcc}\ N\ M\ \delta\ A, \eta, \texttt{i} :: ps) && = \textsc{CodeGen}_{\text{acc}[N.M.\delta]}(A, \eta, \texttt{i/}M :: \texttt{i\%}M :: ps)\\
&\textsc{CodeGen}_{\text{acc}[N.M.\delta]}(\textbf{joinAcc}\ n\ M\ \delta\ A, \eta, \texttt{i} :: \texttt{j} :: ps) && = \textsc{CodeGen}_{\text{acc}[NM.\delta]}(A, \eta, \texttt{i*}M\texttt{+j} :: ps)\\
&\textsc{CodeGen}_{\text{acc}[\delta_1]}(\textbf{pairAcc}_1\ \delta_1\ \delta_2\ A, \eta, ps) && = \textsc{CodeGen}_{\text{acc}[\delta_1 \times \delta_2]}(A, \eta, \texttt{.x1} :: ps)\\
&\textsc{CodeGen}_{\text{acc}[\delta_2]}(\textbf{pairAcc}_2\ \delta_1\ \delta_2\ A, \eta, ps) && = \textsc{CodeGen}_{\text{acc}[\delta_1 \times \delta_2]}(A, \eta, \texttt{.x2} :: ps)\\
&\textsc{CodeGen}_{\text{acc}[N.\delta_1]}(\textbf{zipAcc}_1\ N\ \delta_1\ \delta_2\ A, \eta, \texttt{i} :: ps) && = \textsc{CodeGen}_{\text{acc}[N.(\delta_1 \times \delta_2)]}(A, \eta, \texttt{i} :: \texttt{.x1} :: ps)\\
&\textsc{CodeGen}_{\text{acc}[N.\delta_2]}(\textbf{zipAcc}_2\ N\ \delta_1\ \delta_2\ A, \eta, \texttt{i} :: ps) && = \textsc{CodeGen}_{\text{acc}[N.(\delta_1 \times \delta_2)]}(A, \eta, \texttt{i} :: \texttt{.x2} :: ps)
\end{aligned}
$$

(b) DPIA acceptors to C l-values

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

(c) DPIA expressions to C r-values

Fig. 5. Translation of Purely Imperative DPIA to parallel OpenMP C

by manipulating the access path appropriately until an 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 5c) 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 (4) of the simple dot product in Section 2. We use the environment $\eta = [out \mapsto \texttt{out}, xs \mapsto \texttt{xs}, ys \mapsto \texttt{ys}]$. 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:

$$
\begin{aligned}
& \textsc{CodeGen}_{\textsf{comm}}(\textbf{parfor}\ \underline{\ell}\ out\ (\lambda i\ o.o := \textbf{fst}\ (\textbf{idx}\ (\textbf{zip}\ xs\ ys)\ i) * \textbf{snd}\ (\textbf{idx}\ (\textbf{zip}\ xs\ ys)\ i)), \eta) \\
= \quad & \texttt{\#pragma omp parallel for} \\
& \texttt{for(int i = 0; i < } \ell \texttt{; i+=1) \{} \\
& \quad \textsc{CodeGen}_{\textsf{comm}}(\textbf{idxAcc}\ out\ i := \textbf{fst}\ (\textbf{idx}\ (\textbf{zip}\ xs\ ys)\ i) * \textbf{snd}\ (\textbf{idx}\ (\textbf{zip}\ xs\ ys)\ i)), \eta[i \mapsto \texttt{i}]) \\
& \texttt{\}} \\
= \quad & \texttt{\#pragma omp parallel for} \\
& \texttt{for(int i = 0; i < } \ell \texttt{; i+=1) \{} \\
& \quad \textsc{CodeGen}_{\textsf{acc[num]}}(\textbf{idxAcc}\ out\ i, \eta[i \mapsto \texttt{i}], []) = \\
& \qquad \textsc{CodeGen}_{\textsf{exp[num]}}(\textbf{fst}\ (\textbf{idx}\ (\textbf{zip}\ xs\ ys)\ i) * \textbf{snd}\ (\textbf{idx}\ (\textbf{zip}\ xs\ ys)\ i)), \eta[i \mapsto \texttt{i}], [])\texttt{;} \\
& \texttt{\}}
\end{aligned}
$$

The acceptor part of the assignment is translated as follows:

$$
\begin{aligned}
\textsc{CodeGen}_{\textsf{acc[num]}}(\textbf{idxAcc}\ out\ i, \eta[i \mapsto \texttt{i}], []) &= \textsc{CodeGen}_{\textsf{acc}[\underline{\ell}.\textsf{num}]}(out, \eta[i \mapsto \texttt{i}], [\texttt{i}]) \\
&= \texttt{out[i]}
\end{aligned}
$$

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.

$$
\begin{aligned}
& \textsc{CodeGen}_{\textsf{exp[num]}}(\textbf{fst}\ (\textbf{idx}\ (\textbf{zip}\ xs\ ys)\ i), \eta[i \mapsto \texttt{i}], []) \\
= \quad & \textsc{CodeGen}_{\textsf{exp[num}\times\textsf{num]}}(\textbf{idx}\ (\textbf{zip}\ xs\ ys)\ i, \eta[i \mapsto \texttt{i}], [\texttt{.x1}]) \\
= \quad & \textsc{CodeGen}_{\textsf{exp}[\underline{\ell}.(\textsf{num}\times\textsf{num})]}(\textbf{zip}\ xs\ ys, \eta[i \mapsto \texttt{i}], [\texttt{i, .x1}]) \\
= \quad & \textsc{CodeGen}_{\textsf{exp}[\underline{\ell}.\textsf{num}]}(xs, \eta[i \mapsto \texttt{i}], [\texttt{i}]) \\
= \quad & \texttt{xs[i]}
\end{aligned}
$$

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

```
#pragma omp parallel for
for(int i = 0; i < ℓ; 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 imperative 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 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]). 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. 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:

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

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 3a 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 $\Delta \mid \Pi; \Gamma \vdash P, Q : \theta$ are equivalent iff for all closing contexts $C[-]$, the programs $C[P]$ and $C[Q]$, when instantiated with the standard interpretation of variables ([Reddy 2013], Figure 2), describe the same mapping of initial to final values. We formally write $\Delta \mid \Pi; \Gamma \vdash P \simeq Q : \theta$, or informally $P \simeq Q$. 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 3a. 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:

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

Full $\beta\eta$-equality for functions is one of the defining features of Idealised Algol and its descendants [Reynolds 1997]. These are all justified by Reddy's model.

(2) The **parfor** implementation of **mapParI** (Section 4.2) satisfies the following equivalence:

$$\textbf{mapParI}\ N\ \delta_1\ \delta_2\ (\lambda x\ o.\ o :=_{\delta_2} F\ x)\ E\ A \quad \simeq \quad A :=_{N.\delta_2} \textbf{mapPar}\ N\ \delta_1\ \delta_2\ F\ E \tag{5}$$

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

$$\textbf{mapParI}\ N\ \delta_1\ \delta_2\ (\lambda x\ o.\ o :=_{\delta_2} F\ x)\ E\ A \quad \simeq \quad \textbf{mapParI}\ N\ \delta_2\ \delta_2\ (\lambda x\ o.\ o :=_{\delta_2} x)\ (\textbf{mapPar}\ \delta_1\ \delta_2\ F\ E)\ A$$

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

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

which is immediate from the interpretation of arrays 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 continuations C that are non-interfering, we have:

$$\textbf{new}\ \delta\ (\lambda tmp.\ tmp.1 :=_{\delta} E; C(tmp.2)) \quad \simeq \quad C(E) \tag{6}$$

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 read by E, then it would be unsafe 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 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 **reduceSeqI** should satisfy the following equivalence.

$$\textbf{reduceSeqI}\ N\ \delta_1\ \delta_2\ (\lambda x\ y\ o.\ o :=_{\delta_2} F\ x\ y)\ I\ E\ C \quad \simeq \quad C(\textbf{reduceSeq}\ N\ \delta 1\ \delta_2\ F\ I\ E) \tag{7}$$

Expanding **reduceSeqI**, and β-reducing, this is equivalent to showing:

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

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:

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

Now Equation 7 follows from Equation 6.

(5) Finally, we need agreement between 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 above interpretation of acceptors for compound data types using separating products.

Theorem 5.1. *The translations $\mathcal{A}(\!|-|\!)_-(-)$ and $\mathcal{C}(\!|-|\!)_-(-)$ defined in Figure 4a and Figure 4b satisfy the following observational equivalences for all acceptors A and functional expressions E with disjoint sets of active identifiers:*

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

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**:

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

$$\frac{\ell \in \{2, 4, 8, 16\}}{\Delta \vdash \underline{\ell} : \mathsf{width}} \qquad \frac{\Delta \vdash W : \mathsf{width}}{\Delta \vdash \langle W \rangle \mathsf{num} : \mathsf{data}}$$

Fig. 6. Extension of DPIA Types with a Vector Data Type

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

The acceptor-passing translation of **mapPar** uses the induction hypothesis to establish the correctness of the translations of the subterms, β-equality, and then the correctness property of **mapParI** (Equation 5):

$$\begin{aligned}
\mathcal{A}(\!|\mathbf{mapPar}\ N\ \delta_1\ \delta_2\ F\ E|\!)_{N.\delta_2}(A) &= C(\!|E|\!)_{N.\delta_1}(\lambda x.\ \mathbf{mapParI}\ N\ \delta_1\ \delta_2\ (\lambda x\ o.\ \mathcal{A}(\!|F\ x|\!)_{\delta_2}(o))\ x\ A) \\
&\simeq \mathbf{mapParI}\ N\ \delta_1\ \delta_2\ (\lambda x\ o.\ \mathcal{A}(\!|F\ x|\!)_{\delta_2}(o))\ E\ A) \\
&\simeq \mathbf{mapParI}\ N\ \delta_1\ \delta_2\ (\lambda x\ o.\ o :=_{\delta_2} F\ x)\ E\ A \\
&\simeq A :=_{N.\delta_2} \mathbf{mapPar}\ N\ \delta_1\ \delta_2\ F\ E
\end{aligned}$$

The continuation-passing translation of **mapPar** relies on the acceptor-passing translation and additionally Equation 6 (temporary storage is unobservable). The acceptor-passing and contination-passing translations for **reduceSeq** both rely on Equation 7 establishing correctness of **reduceSeqI**.

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 EXTENSIONS

This section introduces two extensions to our core DPIA language: *vectorisation* and support for performing *iterative computations*. These extensions show the extensibility of DPIA for expressing more applications as well as execution strategies.

6.1 Vectorisation

Modern processors provide special SIMD vector instructions to perform computations on multiple scalar values in a single clock cycle. These instructions can significantly increase the performance of programs. To be able to explicitly encode strategies for vector operations in DPIA we extend the type system (Figure 6), introduce new primitives to operate on vectors (Figure 7), and extend the acceptor-passing, CPS, and code generation translations (Figure 8 and Figure 9).

We extend the definition of data types by adding a new vector type $\langle w \rangle \mathsf{num}$ (Figure 6) where w denotes vector width, *i.e.*, the number of elements grouped in the vector which can only be one of the sizes for which hardware instructions are available (here we assume 2, 4, 8, or 16). Widths have kind width, which is a subkind of nat. Thus a width w can be used as part of an array index.

To operate on vector values we introduce vector-specific primitives which resemble corresponding primitives defined over arrays. The **mapVec** primitive maps a scalar function over each component of a vector. Similarly to the **split** and **join** primitives, **asVector** and **asScalar** transform arrays of scalar values into arrays of vectors and vice versa. The **idxVec** primitive is used to access the individual components of a vector.

On the imperative side we add a vectorised for loop with the **forVec** primitive as well as the acceptor version of the aforementioned data layout primitives. Finally, we added the intermediate macro **mapVecI** which is expanded in our translation as follows:

$$\mathbf{mapVecI} = \Lambda n.\ \lambda f\ xs\ out.\ \mathbf{forVec}\ n\ out\ (\lambda i\ o.\ f\ (\mathbf{idxVec}\ n\ xs\ i)\ o)$$

$$
\begin{array}{lll}
\textbf{mapVec} & : & (w : \text{width}) \to (\text{exp}[\text{num}] \to \text{exp}[\text{num}]) \to \text{exp}[\langle w \rangle \text{num}] \to \text{exp}[\langle w \rangle \text{num}] \\
\textbf{asVector} & : & (w : \text{width}) \to (n : \text{nat}) \to \text{exp}[nw.\text{num}] \to \text{exp}[n.\langle w \rangle \text{num}] \\
\textbf{asScalar} & : & (w : \text{width}) \to (n : \text{nat}) \to \text{exp}[n.\langle w \rangle \text{num}] \to \text{exp}[nw.\text{num}] \\
\textbf{idxVec} & : & (w : \text{width}) \to \text{exp}[\langle w \rangle \text{num}] \to \text{exp}[\text{idx}[w]] \to \text{exp}[\text{num}] \\
\textbf{forVec} & : & (w : \text{width}) \to \text{acc}[\langle w \rangle \text{num}] \to (\text{exp}[\text{idx}[w]] \to \text{acc}[\text{num}] \to_{\text{p}} \text{comm}) \to \text{comm} \\
\textbf{asVectorAcc} & : & (w : \text{width}) \to (n : \text{nat}) \to \text{acc}[n.\langle w \rangle \text{num}] \to \text{acc}[nw.\text{num}] \\
\textbf{asScalarAcc} & : & (w : \text{width}) \to (n : \text{nat}) \to \text{acc}[nw.\text{num}] \to \text{acc}[n.\langle w \rangle \text{num}] \\
\textbf{idxVecAcc} & : & (w : \text{width}) \to \text{acc}[\langle w \rangle \text{num}] \to \text{exp}[\text{idx}[w]] \to \text{acc}[\text{num}] \\
\textbf{mapVecI} & : & (w : \text{width}) \to (\text{exp}[\text{num}] \to \text{acc}[\text{num}] \to_{\text{p}} \text{comm}) \to \text{exp}[\langle w \rangle \text{num}] \to \text{acc}[\langle w \rangle \text{num}] \to \text{comm}
\end{array}
$$

Fig. 7. Primitives for Vectorisation

$$
\begin{array}{lcl}
\mathcal{A}(\!|\textbf{mapVec}\ W\ F\ E|\!)_{\langle W \rangle \text{num}}(A) & = & C(\!|E|\!)_{\langle W \rangle \text{num}}(\lambda x.\ \textbf{mapVecI}\ W\ (\lambda x\ o. \mathcal{A}(\!|F\ x|\!)_{\text{num}}(o))\ x\ A) \\
\mathcal{A}(\!|\textbf{asVector}\ W\ N\ E|\!)_{N.\langle W \rangle \text{num}}(A) & = & \mathcal{A}(\!|E|\!)_{NW.\text{num}}(\textbf{asVectorAcc}\ W\ N\ A) \\
\mathcal{A}(\!|\textbf{asScalar}\ W\ N\ E|\!)_{NW.\text{num}}(A) & = & \mathcal{A}(\!|E|\!)_{N.\langle W \rangle \text{num}}(\textbf{asScalarAcc}\ W\ N\ A)
\end{array}
$$

$$
\begin{array}{lcl}
C(\!|\textbf{mapVec}\ W\ F\ E|\!)_{\langle W \rangle \text{num}}(C) & = & \textbf{new}\ (\langle W \rangle \text{num})\ (\lambda tmp.\ \mathcal{A}(\!|\textbf{mapVec}\ W\ F\ E|\!)_{\langle W \rangle \text{num}}(tmp.2);\ C(tmp.1)\ ) \\
C(\!|\textbf{asVector}\ W\ N\ E|\!)_{N.\langle W \rangle \text{num}}(C) & = & C(\!|E|\!)_{NW.\text{num}}(\lambda x.\ C(\textbf{asVector}\ W\ N\ x)\ ) \\
C(\!|\textbf{asScalar}\ W\ N\ E|\!)_{WN.\text{num}}(C) & = & C(\!|E|\!)_{N.\langle W \rangle \text{num}}(\lambda x.\ C(\textbf{asScalar}\ W\ N\ x)\ )
\end{array}
$$

Fig. 8. Acceptor and Continuation-passing Translations for Vectorisation

$$
\begin{array}{l}
\textsc{CodeGen}_{\text{comm}}(\textbf{forVec}\ N\ A\ (\lambda i\ o.\ P)\ E, \eta) = \texttt{\#pragma omp simd} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \texttt{for(int i = 0; i < } n\texttt{; i += 1) \{} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \textsc{CodeGen}_{\text{comm}}(P[\textbf{idxVecAcc}\ N\ A\ i/o], \eta[i \mapsto \texttt{i}])\ \texttt{\}}
\end{array}
$$

$$
\begin{array}{l}
\textsc{CodeGen}_{\text{acc}[WN.\text{num}]}(\textbf{asVectorAcc}\ W\ N\ A, \eta, \texttt{i} :: ps) = \textsc{CodeGen}_{\text{acc}[N.\langle W \rangle \text{num}]}(A, \eta, \texttt{i}/W :: ps) \\
\textsc{CodeGen}_{\text{acc}[N.\langle W \rangle \text{num}]}(\textbf{asScalarAcc}\ W\ N\ A, \eta, \texttt{i} :: \texttt{j} :: ps) = \textsc{CodeGen}_{\text{acc}[NW.\text{num}]}(A, \eta, \texttt{i}{*}N{+}\texttt{j} :: ps) \\
\textsc{CodeGen}_{\text{acc}[\text{num}]}(\textbf{idxVecAcc}\ W\ A\ I, \eta, ps) = \textsc{CodeGen}_{\text{acc}[\langle W \rangle \text{num}]}(A, \eta, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \textsc{CodeGen}_{\text{exp}[\text{idx}[W]]}(I, \eta, []) :: ps))
\end{array}
$$

$$
\begin{array}{l}
\textsc{CodeGen}_{\text{exp}[M.\langle N \rangle \text{num}]}(\textbf{asVector}\ N\ M\ E, \eta, \texttt{i} :: \texttt{j} :: ps) = \textsc{CodeGen}_{\text{exp}[MN.\text{num}]}(E, \eta, \texttt{i}{*}N{+}\texttt{j} :: ps) \\
\textsc{CodeGen}_{\text{exp}[NW.\text{num}]}(\textbf{asScalar}\ W\ N\ E, \eta, \texttt{i} :: ps) = \textsc{CodeGen}_{\text{exp}[N.\langle W \rangle \text{num}]}(E, \eta, \texttt{i}/W :: \texttt{i}\%W :: ps) \\
\textsc{CodeGen}_{\text{exp}[\text{num}]}(\textbf{idxVec}\ N\ E\ I, \eta, ps) = \textsc{CodeGen}_{\text{exp}[\langle W \rangle \text{num}]}(E, \eta, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \textsc{CodeGen}_{\text{exp}[\text{idx}[W]]}(I, \eta, []) :: ps))
\end{array}
$$

Fig. 9. Translation of Imperative Vector Primitives DPIA to Parallel OpenMP C

Figure 8 extends the acceptor and continuation-passing translations to account for the vector primitives. The translations are similar to those of **mapPar**, **split**, **join**, **splitAcc**, and **joinAcc**.

Code generation for vectors is defined in Figure 9. The **forVec** primitive is translated into a for loop annotated with OpenMP's `simd` pragma which indicates to the underlying C compiler that the instructions in the loop should be vectorised. Code generation for the other primitives is similar to code generation for the corresponding array primitives.

The assignment operation is extended to support vectors as follows:

$$A :=_{\langle W \rangle \text{num}} E \quad = \quad \textbf{mapVecI}\ W\ (\lambda x\ a.a :=_{\text{num}} x)\ E\ A$$

$$
\begin{array}{lcl}
\textbf{iterate} & : & (n\ m\ k : \mathsf{nat}) \to (t : \mathsf{data}) \to \\
& & \quad ((l : \mathsf{nat}) \to \mathsf{exp}[ln.t] \to \mathsf{exp}[l.t]) \to \mathsf{exp}[n^k m.t] \to \mathsf{exp}[m.t] \\
\textbf{newDoubleBuffer} & : & (n\ m\ k : \mathsf{nat}) \to (t : \mathsf{data}) \to \mathsf{exp}[n.t] \to \mathsf{acc}[m.t] \to \\
& & \quad (\mathsf{var}[k.t] \to \mathsf{comm} \to \mathsf{comm} \to \mathsf{comm}) \to \mathsf{comm} \\
\textbf{truncExp} & : & (n\ m : \mathsf{nat}) \to (t : \mathsf{data}) \to \mathsf{exp}[n.t] \to \mathsf{exp}[m.t] \\
\textbf{truncAcc} & : & (n\ m : \mathsf{nat}) \to (t : \mathsf{data}) \to \mathsf{acc}[n.t] \to \mathsf{acc}[m.t] \\
\textbf{iterateIAcc} & : & (n\ m\ k : \mathsf{nat}) \to (t : \mathsf{data}) \to \mathsf{acc}[m.t] \to \\
& & \quad ((l : \mathsf{nat}) \to \mathsf{acc}[n.t] \to \mathsf{exp}[ln.t] \to \mathsf{comm}) \to \mathsf{exp}[n^k m.t] \to \mathsf{comm}
\end{array}
$$

Fig. 10. Primitives for Iterative Computations

The vector extension allows us to specify strategies for which OpenMP code with vector instructions is generated. An example with the vectorised dot-product is given in Section 2 (Equation 3) and its translation into the OpenMP code is shown in Listing 2.2.

6.2 Iterative Computations

Our high-level functional language is deliberately restrictive. For example, it does not allow us to define recursive functions. While this severely limits the expressiveness of our language, the restrictiveness allows the generation of high performance code.

Here we show how to extend the expressiveness of the language by adding a limited form of iterative computations for which we provide an efficient implementation using a double buffering scheme. We believe that other similar extensions can be designed to extend the expressiveness without losing the ability to generate high performance code.

Figure 10 introduces the **iterate** primitive which applies a given function a fixed number of times to an array, by supplying the output of one function application as input to the next. We take our formulation of iterate from Steuwer et al. [2015] with a type which allows the size of the array to shrink by a fixed factor n in each iteration, allowing **iterate** to be used for implementing a parallel tree-based reduction by iteratively performing multiple sequential reductions in parallel — an implementation strategy suggested by Nvidia to be used on their GPUs [Harris 2008].

We implement **iterate** by double buffering, that is, we allocate two intermediate buffers and switch between which is read from and written to after each iteration. For this we introduce the **newDoubleBuffer** primitive:

$$\textbf{newDoubleBuffer}\ N\ M\ K\ \delta\ E\ A\ (\lambda v\ swap\ done.\ P)$$

Similar to the **new** primitive it binds a variable v which can be read from and written to, but in contrast to **new** this variable provides an interface to two different buffers in memory: reading from one and writing to the other. Besides the variable v the primitive also takes two commands: *swap* and *done*. The intention is that when *swap* is invoked, it swaps which buffer is read from and which buffer is written to and when *done* is invoked, the reading buffer is swapped as usual but the writing buffer is set directly to the output.

Figure 12 shows the code generation for **newDoubleBuffer** showing its efficient implementation. After allocation of two temporary buffers, two pointers are initialised: one pointing to the input E and the other to the first temporary buffer `tmp1`. The implementation of *swap* and *done* are added to an extended environment η which as well as mapping from DPIA identifiers to C variable names, now also maps identifiers of type comm to C statements. Every time *swap* or *done* are used inside of P, the statements from the environment will be emitted. We implement *swap* efficiently by swapping the two pointers between the two temporary buffers; a flag is toggled every time to

$$\mathcal{A}(\!|\textbf{iterate}\ N\ M\ K\ \delta\ F\ E|\!)_{N.\delta}(A) \quad = \quad C(\!|E|\!)_{N^K N.\delta}(\lambda x.\ \textbf{iterateIAcc}\ N\ M\ K\ \delta\ A\ (\Lambda l.\lambda o\ x.\mathcal{A}(\!|F\ l\ x|\!)_{l.\delta}(o))\ x)$$
$$C(\!|\textbf{iterate}\ N\ M\ K\ \delta\ F\ E|\!)_{N.\delta}(C) \quad = \quad \textbf{new}\ (N.\delta)\ (\lambda tmp.\ \mathcal{A}(\!|\textbf{iterate}\ N\ M\ K\ \delta\ F\ E|\!)_{N.\delta}(tmp.2);\ C(tmp.1)\ )$$

Fig. 11. Acceptor and Continuation-passing Translations for the **iterate** primitive

```
CodeGen_comm(newDoubleBuffer N M K δ E A (λp. P), η) =
  {
    CodeGen_data(K.δ) tmp1;   CodeGen_data(K.δ) tmp2;
    CodeGen_data(δ)* in_ptr = &(CodeGen_exp[N.δ](E, η, [0]));
    CodeGen_data(δ)* out_ptr = tmp1;
    char flag = 1;
    CodeGen_comm(P[(((v_a, v_e), swap), done)/p],
      η[ v_a ↦ out_ptr, v_e ↦ in_ptr,
        swap ↦ {in_ptr = flag ? tmp1 : tmp2;
                out_ptr = flag ? tmp2 : tmp1;
                flag = flag ^ 1;},
        done ↦ {in_ptr = flag ? tmp1 : tmp2;
                out_ptr = &(CodeGen_acc[M.δ](A, η, [0]));} ])
  }
```

Fig. 12. Translation of the Imperative **newDoubleBuffer** primitive to Parallel OpenMP C

indicate how to configure the buffers. We implement *done* by setting the pointer used for writing to the output acceptor A.

The **iterate** primitive is implemented in terms of **newDoubleBuffer** via a translation to the **iterateIAcc** macro given in Figure 11. **iterateIAcc** is then defined as follows:

$$\begin{aligned}\textbf{iterateIAcc} = \Lambda n\ m\ k\ \delta\ (\lambda a\ f\ e.\ &\textbf{newDoubleBuffer}\ (n^k m)\ m\ (n^k m)\ \delta\ e\ a\ (\lambda v\ swap\ done.\\ &\textbf{for}\ k\ (\lambda i.\ f\ (n^{k-1}m)\ (\textbf{truncAcc}\ (n^k m.\delta)\ (n^{k-i-1}m.\delta)\ \delta\ v.1)\\ &\qquad (\textbf{truncExp}\ (n^k m.\delta)\ (n^{k-i}m.\delta)\ \delta\ v.2)\ ;\\ &\qquad \textbf{if}\ (i < k-1)\ \textbf{then}\ swap\ \textbf{else}\ done)\ )\ )\end{aligned}$$

We assume a standard extension to support booleans. The **for** loop performs the iterative application of the function F. After each iteration *swap* is invoked until the second to last iteration when *done* is invoked. This implementation ensures that no unnecessary copies of input or output arrays are made: the first iteration reads straight from the input and the last writes directly to the output.

The size of the arrays change in each iteration but only two buffers of a fixed size are allocated. To ensure that the program typechecks we introduce **truncExp** and **truncAcc** primitives, which are no-ops that restrict the size of a larger array. For the implementation of **iterateIAcc** we also extend the arithmetic expressions with a subtraction operation. All nat expressions in the implementation are guaranteed to be non-negative as i is the iteration variable and thus always smaller than k.

7 EXPERIMENTAL RESULTS

We implemented DPIA and the compilation to OpenMP code via the translation described in Section 4. This section evaluates the quality of the OpenMP code generated with our implementation. We are interested to see if this formal translation introduces overheads compared to hand-written OpenMP code. We start by describing our experimental setup and the benchmarks used.

Experimental Setup. We used an Intel Core i7-7700 CPU with 4 physical cores and hyper-threading enabled. We measured the runtime of 10 runs and report the median runtime. We compare against

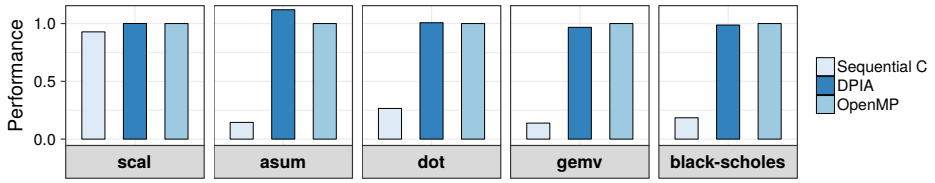

Fig. 13. Performance of DPIA generated code for five benchmarks compared to sequential C and OpenMP.

hand-written OpenMP code on four simple linear algebra benchmarks as well as the Black-Scholes application from financial stochastics.

Performance Results. Figure 13 shows the relative performance of sequential C code on the left, the code generated from out DPIA language in the middle and hand-written OpenMP code on the right. As can be seen, parallelism is beneficial for most of our benchmarks and our implementation does not introduce any significant overhead compared to the hand-written OpenMP code. For the absolute sum benchmark (asum) our DPIA implementation even slightly outperforms the hand-written OpenMP implementation as we use a more complex strategy using vectorisation along the lines of the example shown in Section 2.

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 Parallel Architectures. There exist multiple functional approaches for generating code for parallel 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 code while still providing functional abstractions. Accelerate is a Haskell embedded domain specific language providing higher level abstractions with the aim of generating efficient parallel 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. Anderson et al. [2017] present an approach to compile array-code written in Julia to parallel code. Delite [Sujeeth et al. 2014] enables the creation of domain-specific languages using functional parallel patterns targeting parallel hardware.

9 CONCLUSIONS AND FUTURE WORK

We introduced a novel data parallel programming language DPIA, an interference controlled dialect of Idealised Algol. We showed how DPIA is used as a foundation for a formal translation of data parallel functional programs to imperative code for parallel machines. The approach offers strong guarantees about the absence of data-races in the generated programs and a straightforward translation strategy for generating efficient OpenMP code. Our experimental results show that our approach yields efficient code on a par with hand-coded low-level parallel code.

While this paper has presented an implementation targeting OpenMP we are eager to extend this work to target a diverse set of heterogeneous devices such as GPUs and FPGAs where the choice of strategy is more challenging than on traditional CPUs. We envision such an extended version of DPIA as a unifying language for generating efficient code for diverse parallel hardware from a single high-level portable program, thus obtaining performance portability. To pursue our vision we intend to model parallelisation and memory hierarchies present in modern parallel devices such as GPUs in the type system which is for example required to guarantee deadlock freedom. While OpenMP follows the sequential-by-default fork-join parallelism model, the dominant low-level GPU programming models OpenCL and CUDA follow a parallelism-by-default model which requires changes to the presented translation to continue to guarantee race-freedom. Finally, we plan to push our formalisation further to prove the correctness of the final translation from DPIA to C, OpenMP, and in the future OpenCL.

REFERENCES

Todd A. Anderson, Hai Liu, Lindsey Kuper, Ehsan Totoni, Jan Vitek, and Tatiana Shpeisman. 2017. Parallelizing Julia with a Non-Invasive DSL. In *31st European Conference on Object-Oriented Programming, ECOOP 2017, June 19-23, 2017, Barcelona, Spain (LIPIcs)*, Peter Müller (Ed.), Vol. 74. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 4:1–4:29. https://doi.org/10.4230/LIPIcs.ECOOP.2017.4

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.

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.

Mark Harris. 2008. *Optimizing Parallel Reduction in CUDA*. Technical Report. nVidia. http://developer.download.nvidia.com/assets/cuda/files/reduction.pdf

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

Steve Klabnik and Carol Nichols. 2018. *The Rust Programming Language* (2nd ed.). No Starch Press.

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.

Jonathan Ragan-Kelley, Andrew Adams, Dillon Sharlet, Connelly Barnes, Sylvain Paris, Marc Levoy, Saman P. Amarasinghe, and Frédo Durand. 2018. Halide: decoupling algorithms from schedules for high-performance image processing. *Commun. ACM* 61, 1 (2018), 106–115. https://doi.org/10.1145/3150211

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. 2016. Matrix multiplication beyond auto-tuning: rewrite-based GPU code generation. In *2016 International Conference on Compilers, Architectures and Synthesis for Embedded Systems, CASES 2016, Pittsburgh, Pennsylvania, USA, October 1-7, 2016*. ACM, 15:1–15:10. https://doi.org/10.1145/2968455.2968521

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.

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.
Philip W. Trinder, Kevin Hammond, Hans-Wolfgang Loidl, and Simon L. Peyton Jones. 1998. Algorithms + Strategy = Parallelism. *J. Funct. Program.* 8, 1 (1998), 23–60. http://journals.cambridge.org/action/displayAbstract?aid=44147