Destination-Passing Style for Efficient Memory Management

Amir Shaikhha[*]
EPFL
Lausanne, Switzerland
amir.shaikhha@epfl.ch

Andrew Fitzgibbon
Microsoft HoloLens
Cambridge, United Kingdom
awf@microsoft.com

Simon Peyton Jones
Microsoft Research
Cambridge, United Kingdom
simonpj@microsoft.com

Dimitrios Vytiniotis
Microsoft Research
Cambridge, United Kingdom
dimitris@microsoft.com

Abstract

We show how to compile high-level functional array-processing programs, drawn from image processing and machine learning, into C code that runs as fast as hand-written C. The key idea is to transform the program to *destination-passing style*, which in turn enables a highly-efficient stack-like memory allocation discipline.

CCS Concepts • **Software and its engineering** → **Memory management**; **Functional languages**;

Keywords Destination-Passing Style, Array Programming

ACM Reference Format:
Amir Shaikhha, Andrew Fitzgibbon, Simon Peyton Jones, and Dimitrios Vytiniotis. 2017. Destination-Passing Style for Efficient Memory Management. In *Proceedings of 6th ACM SIGPLAN International Workshop on Functional High-Performance Computing, Oxford, UK, September 7, 2017 (FHPC'17),* 12 pages.
https://doi.org/10.1145/3122948.3122949

1 Introduction

Applications in computer vision, robotics, and machine learning [35, 38] may need to run in memory-constrained environments with strict latency requirements, and have high turnover of small-to-medium-sized arrays. For these applications the overhead of most general-purpose memory management, for example malloc/free, or of a garbage collector, is unacceptable, so programmers often implement custom memory management directly in C.

In this paper we propose a technique that automates a common custom memory-management technique, which we call *destination passing style* [23, 24] (DPS), as used in efficient C and Fortran libraries such as BLAS. We allow the programmer to code in a high-level functional style, while guaranteeing efficient stack allocation of all intermediate arrays. Fusion techniques for such languages are absolutely essential to eliminate intermediate arrays, and are well established. But fusion leaves behind an irreducible core of intermediate arrays that *must* exist to accommodate multiple or random-access consumers.

The key idea behind DPS is that every function is given the storage in which to store its result. The caller of the function is responsible for allocating the destination storage, and deallocating it as soon as it is no longer needed. This incurs a burden at the call site of computing the size of the callee result, but we will show how a surprisingly rich input language can nevertheless allow these computations to be done statically, or in negligible time. Our contributions are:

- We propose a new destination-passing style intermediate representation that captures a stack-like memory management discipline and ensures there are no leaks (Section 3). This is a good compiler intermediate language because we can perform transformations on it and reason about how much memory a program will take. It also allows efficient C code generation with bump-allocation. Although it is folklore to compile functions in this style when the result size is known, we have not seen DPS used as an actual compiler intermediate language, despite the fact that DPS has been used for other purposes (cf. Section 6).
- DPS requires to know at the call site how much memory a function will need. We design a carefully-restricted higher-order functional language, $\widetilde{\mathrm{F}}$ (Section 2) which is a subset of F#, and a *compositional* shape translation (Section 3.3) that guarantees to compute the result size of any $\widetilde{\mathrm{F}}$ expression, either statically or at runtime, with no allocation, and a run-time cost independent of the data or its size (Section 3.6). Other languages with similar properties [20] expose shape concerns intrusively at the language level, while $\widetilde{\mathrm{F}}$ programs are just F#.
- We present the implementation of of the technique (Section 4) and evaluate the runtime and memory performance of both micro-benchmarks and real-life computer vision and machine-learning workloads written in our high-level language and compiled to C via DPS (as shown in Section 5). We show that our approach gives performance comparable to, and sometimes better than, idiomatic C++.

2 $\widetilde{\mathrm{F}}$

$\widetilde{\mathrm{F}}$ (we pronounce it F smooth) is a subset of F#, an ML-like functional programming language (the syntax in this paper is slightly different from F# for presentation reasons). It is designed to be *expressive enough* to make it easy to write array-processing workloads, while simultaneously being *restricted enough* to allow it to be compiled to code that is as efficient as hand-written C, with very simple and efficient memory management. We are willing to sacrifice some expressiveness to achieve higher performance. As presented here,

[*]This work was done while the author was doing an internship at Microsoft Research, Cambridge.

FHPC'17, September 7, 2017, Oxford, UK

ACM ISBN 978-1-4503-5181-2/17/09…$15.00
https://doi.org/10.1145/3122948.3122949

e	::=	$e\ \overline{e}$	– Application
	|	$\lambda\overline{x}.e$	– Abstraction
	|	x	– Variable Access
	|	n	– Scalar Value
	|	i	– Index Value
	|	N	– Cardinality Value
	|	c	– Constants (see below)
	|	`let` x = e `in` e	– (Non-Rec.) Let Binding
	|	`if` e `then` e `else` e	– Conditional
T	::=	M	– Matrix Type
	|	$\overline{T} \Rightarrow M$	– Function Types (No Currying)
	|	`Card`	– Cardinality Type
	|	`Bool`	– Boolean Type
M	::=	Num	– Numeric Type
	|	`Array<M>`	– Vector, Matrix, ... Type
Num	::=	`Double` | `Index`	– Scalar and Index Type

Scalar Function Constants:

`+` | `-` | `*` | `/`	:	Num, Num ⇒ Num
`%`	:	`Index`, `Index` ⇒ `Index`
`>` | `<` | `==`	:	Num, Num ⇒ `Bool`
`&&` | `||`	:	`Bool`, `Bool` ⇒ `Bool`
`!`	:	`Bool` ⇒ `Bool`
$+^c \mid -^c \mid *^c \mid /^c \mid \%^c$	:	`Card`, `Card` ⇒ `Card`

Vector Function Constants:

`build` *n f*	:	`Card` , (`Index` ⇒ M) ⇒ `Array<M>`
`ifold` *f* m_0 *n*	:	(M , `Index` ⇒ M) , M , `Card` ⇒ M
`get` *a i*	:	`Array<M>` , `Index` ⇒ M
`length` *a*	:	`Array<M>` ⇒ `Card`

Syntactic Sugar:

$e_0[e_1]$ = `get` $e_0\ e_1$

e_1 *bop* e_2 = *bop* $e_1\ e_2$ – For binary operators *bop*

Figure 1. The core $\widetilde{\text{F}}$ syntax and function constants.

$\widetilde{\text{F}}$ strictly imposes its language restrictions, rejecting programs for which shape inference is not efficient. Of course it would also be possible to emit compilation warnings for inefficient constructs, and defer shape calculation to runtime, and also to add heap allocation using F#'s explicit "new".

2.1 Syntax and Types of $\widetilde{\text{F}}$

In addition to the usual λ-calculus constructs (abstraction, application, and variable access), $\widetilde{\text{F}}$ supports let binding and conditionals. The syntax and several built-in functions are shown in Figure 1, while the type system is shown in Figure 2. Note that Figure 1 shows an abstract syntax and parentheses can be used as necessary. Also, $\overline{x}$ and $\overline{e}$ denote one or more variables and expressions, respectively, which are separated by spaces, whereas, $\overline{T}$ represents one or more types which are separated by commas.

In support of array programming, the language has several built-in functions defined: `build` for producing arrays; `ifold` for iteration for a particular number of times (from `0` to `n-1`) while maintaining a state across iterations; `length` to get the size of an array; and `get` to index an array.

Although $\widetilde{\text{F}}$ is a higher-order functional language, it is carefully restricted in order to make it efficiently compilable:

$$\text{(T-If)}\ \frac{e_1 : \texttt{Bool} \qquad e_2 : M \qquad e_3 : M}{\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : M} \qquad \text{(T-Var)}\ \frac{x : T \in \Gamma}{\Gamma \vdash x : T}$$

$$\text{(T-App)}\ \frac{e_0 : \overline{T} \Rightarrow M \qquad \overline{e} : \overline{T}}{e_0\ \overline{e} : M} \qquad \text{(T-Abs)}\ \frac{\Gamma \cup \overline{x} : \overline{T} \vdash e : M}{\Gamma \vdash \lambda\overline{x}.e : \overline{T} \Rightarrow M}$$

$$\text{(T-Let)}\ \frac{\Gamma \vdash e_1 : T_1 \qquad \Gamma, x : T_1 \vdash e_2 : T_2}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : T_2}$$

Figure 2. The type system of $\widetilde{\text{F}}$

- $\widetilde{\text{F}}$ does not support arbitrary recursion, hence is not Turing Complete. Instead one can use `build` and `ifold` for producing and iterating over arrays.
- The type system is monomorphic. The only polymorphic functions are the built-in functions of the language, such as `build` and `ifold`, which are best thought of as language constructs rather than first-class functions.
- An array, of type `Array<M>`, is one-dimensional but can be nested. If arrays are nested they are expected to be rectangular, which is enforced by defining the specific `Card` type for dimension of arrays, which is used as the type of the first parameter of the `build` function.
- No partial application is allowed as an expression in this language. Additionally, an abstraction cannot return a function value. These two restrictions are enforced by (T-App) and (T-Abs) typing rules, respectively (cf. Figure 2).

As an example, Figure 3 shows a linear algebra library defined using $\widetilde{\text{F}}$. First, there are vector mapping operations (vectorMap and vectorMap2) which *build* vectors using the size of the input vectors. The i^{th} element (using a zero-based indexing system) of the output vector is the result of the application of the given function to the i^{th} element of the input vectors. Using the vector mapping operations, one can define vector addition, vector element-wise multiplication, and vector-scalar multiplication. Then, there are several vector operations which consume a given vector by *folding* over its elements. For example, vectorSum computes the sum of the elements of the given vector, which is used by the vectorDot and vectorNorm operations. Similarly, several matrix operations are defined using these vector operations. More specifically, matrix-matrix multiplication is defined in terms of vector dot product and matrix transpose. Finally, vector outer product is defined in terms of matrix multiplication of the matrix form of the two input vectors.

2.2 Fusion

Fusion is essential for array programs, without it they cannot be efficient. However fusion is also extremely well studied [6, 10, 32, 41], and we simply take it for granted in this paper. Let us work through one example which illustrates how fusion can be applied to an $\widetilde{\text{F}}$ program.

Consider this function, which returns the norm of the vector resulting from the addition of its two input vectors.

f = λ vec1 vec2. vectorNorm (vectorAdd vec1 vec2)

Executing this program, as is, involves constructing two vectors in total: one intermediate vector which is the result of adding the two vectors `vec1` and `vec2`, and another intermediate vector which is used in the implementation of vectorNorm (vectorNorm

```
let vectorRange = λn. build n (λi. i)
let vectorMap = λv f.
  build (length v) (λi. f v[i])
let vectorMap2 = λv1 v2 f.
  build (length v1) (λi. f v1[i] v2[i])
let vectorAdd = λv1 v2. vectorMap2 v1 v2 (+)
let vectorEMul = λv1 v2. vectorMap2 v1 v2 (×)
let vectorSMul = λv s. vectorMap v (λa. a × s)
let vectorSum = λv.
  ifold (λsum idx. sum + v[idx]) 0 (length v)
let vectorDot = λv1 v2.
  vectorSum (vectorEMul v1 v2)
let vectorNorm = λv. sqrt (vectorDot v v)
let vectorSlice = λv s e.
  build (e -^c s +^c 1) (λi. v[i + s])
let matrixRows = λm. length m
let matrixCols = λm. length m[0]
let matrixMap = λm f. build (length m) (λi. f m[i])
let matrixMap2 = λm1 m2 f.
build (length m1) (λi. f m1[i] m2[i])
let matrixAdd = λm1 m2. matrixMap2 m1 m2 vectorAdd
let matrixTranspose = λm.
  build (matrixCols m) (λi.
    build (matrixRows m) (λj. m[j][i])   )
let matrixMul = λm1 m2.
  let m2T = matrixTranspose m2
  build (matrixRows m1) (λi.
    build (matrixCols m2) (λj.
      vectorDot (m1[i]) (m2T[j])   ) )
let vectorOutProd = λv1 v2.
  let m1 = build 1 (λi. v1)
  let m2 = build 1 (λi. v2)
  let m2T = matrixTranspose m2
  matrixMul m1 m2T
```

Figure 3. Several Linear Algebra and Matrix operations defined in $\widetilde{\text{F}}$.

$$
\begin{array}{lcl}
(\texttt{build}\ e_0\ e_1)[e_2] & \rightsquigarrow & e_1\ e_2 \\
\texttt{length}\,(\texttt{build}\ e_0\ e_1) & \rightsquigarrow & e_0
\end{array}
$$

Figure 4. Fusion rules of $\widetilde{\text{F}}$.

invokes vectorDot, which invokes vectorEMul in order to perform the element-wise multiplication between two vectors). After using the rules presented in Figure 4, the fused function is as follows:

```
f = λvec1 vec2.
  ifold (λsum idx.
    let tmp = vec1[idx]+vec2[idx] in
      sum + tmp * tmp
  ) 0 (length vec1)
```

This is better because it does not construct the intermediate vectors. Instead, the elements of the intermediate vectors are consumed as they are produced.

However, our focus is on efficient allocation and de-allocation of the arrays that fusion cannot remove. For example: the array might be passed to a foreign library function; or it might be passed to a library function that is too big to inline; or it might be consumed by multiple consumers, or by a consumer with a random (non-sequential) access pattern. In these cases there are good reasons to build an intermediate array, but we want to allocate, fill, use, and de-allocate it extremely efficiently. In particular, we do not want to rely on a garbage collector.

3 Destination-Passing Style

Thus motivated, we define a new intermediate language, DPS-$\widetilde{\text{F}}$, in which memory allocation and deallocation is explicit. DPS-$\widetilde{\text{F}}$ uses *destination-passing style*: every array-returning function receives as its first parameter a pointer to memory in which to store the result array. No function allocates the storage needed for its result; instead the responsibility of allocating and deallocating the output storage of a function is given to the caller of that function. Similarly, all the storage allocated inside a function can be deallocated as soon as the function returns its result.

```
t   ::= t t̄ | λx̄. t | n | i | x | c | let x = t in t
      | P                          – Shape Value
      | r                          – Reference Access
      | •                          – Empty Memory Location
      | if t then t else t         – Conditional
      | alloc t (λr. t)            – Memory Allocation
P   ::= ∘                          – Zero Cardinality
      | N                          – Cardinality Value
      | (N, P)                     – Vector Shape Value
c   ::= [See Figure 6]
D   ::= M | D̄ ⇒ M | Bool
      | Shp                        – Shape Type
      | Ref                        – Machine Address
M   ::= Num | Array<M>
Num ::= Double | Index
Shp ::= Card                       – Cardinality Type
      | (Card * Shp)               – Vector Shape Type
```

Figure 5. The core DPS-$\widetilde{\text{F}}$ syntax.

Destination passing style is a standard programming idiom in C. For example, the C standard library procedures that return a string (e.g. `strcpy`) expect the caller to provide storage for the result. This gives the programmer full control over memory management for string values. Other languages have exploited destination-passing style during compilation [15, 16].

3.1 The DPS-$\widetilde{\text{F}}$ Language

The syntax of DPS-$\widetilde{\text{F}}$ is shown in Figure 5, while its type system is in Figure 6. The main additional construct in this language is the one for allocating a particular amount of storage space `alloc` t1 (λr. t2). In this construct t1 is an expression that evaluates to the size (in bytes) that is required for storing the result of evaluating t2. This storage is available in the lexical scope of the lambda parameter, *and is deallocated outside this scope*. The previous example can be written in the following way in DPS-$\widetilde{\text{F}}$:

Typing Rules:

$$\text{(T-Alloc)}\ \frac{\Gamma \vdash t_0 : \texttt{Card} \qquad \Gamma, r : \texttt{Ref} \vdash t_1 : M}{\texttt{alloc}\ t_0\ (\lambda\, r.\ t_1) : M}$$

Vector Function Constants:

```
build  : Ref, Card, (Ref, Index ⇒ M ),
             Card, (Card ⇒ Shp )
                ⇒ Array<M>
ifold  : Ref, (Ref, M, Index ⇒ M ), M, Card,
             (Shp, Card ⇒ Shp ), Shp, Card
                ⇒ M
get    : Ref, Array<M>, Index,
             Shp, Card ⇒ M
length : Ref, Array<M>, Shp ⇒ Card
copy   : Ref, Array<M> ⇒  Array<M>
```

Scalar Function Constants:

DPS version of $\widetilde{F}$ Scalar Constants (See Figure 1).

```
stgOff  :  Ref, Shp ⇒ Ref
vecShp  :  Card, Shp ⇒ (Card * Shp)
fst     :  (Card * Shp) ⇒ Card
snd     :  (Card * Shp) ⇒ Shp
bytes   :  Shp ⇒ Card
```

Syntactic Sugar:

$t_0.[t_1]\{r\}$ = get r t_0 t_1 length t = length • t

(t_0, t_1) = vecShp t_0 t_1

for all binary ops bop: e_1 *bop* e_2 = *bop* • e_1 e_2

Figure 6. The type system and built-in constants of DPS-$\widetilde{F}$

f = λ r_1 vec1 vec2. alloc (vecBytes vec1) (λ r_2.
 vectorNorm_dps • (vectorAdd_dps r_2 vec1 vec2))

Each lambda abstraction typically takes an additional parameter which specifies the storage space that is used for its result. Furthermore, every application should be applied to an additional parameter which specifies the memory location of the return value in the case of an array-returning function. However, a scalar-returning function is applied to a dummy empty memory location, specified by •. In this example, the memory location r_1 can be ignored, whereas the number of bytes allocated for the memory location r_2 is specified by the expression (vecBytes vec1) which computes the number of bytes of the array vec1.

3.2 Translation from $\widetilde{F}$ to DPS-$\widetilde{F}$

We now turn present the translation from $\widetilde{F}$ to DPS-$\widetilde{F}$. Before translating $\widetilde{F}$ expressions to their DPS form, the expressions should be transformed into a normal form similar to ANF [7]. In this representation, each subexpression of an application is either a constant value or a variable. This greatly simplifies the translation rules, specially the (D-App) rule.[1] The representation of our working example in ANF is as follows:

f = λ vec1 vec2.
 let tmp = vectorAdd vec1 vec2 in
 vectorNorm tmp

Figure 7 shows the translation from $\widetilde{F}$ to DPS-$\widetilde{F}$, where $\mathcal{D}[\![e]\!]r$ is the translation of a $\widetilde{F}$ expression e into a DPS-$\widetilde{F}$ expression that stores e's value in memory r. Rule (D-Let) is a good place to start. It uses alloc to allocate enough space for the value of e_1, the right hand side of the let — but how much space is that? We use an auxiliary translation $\mathcal{S}[\![e_1]\!]$ to translate e_1 to an expression that computes e_1's *shape* rather than its *value*. The shape of an array expression specifies the cardinality of each dimension. We will discuss why we need shape (what goes wrong with just using bytes) and the shape translation in Section 3.3. This shape is bound to x^{shp}, and used in the argument to alloc. The freshly-allocated storage r_2 is used as the destination for translating the right hand side e_1, while the original destination r is used as the destination for the body e_2.

[1] In a true ANF, *every* subexpression is a constant value or a variable, whereas in our case, we only care about the subexpressions of an application. Hence, our representation is *almost* ANF.

In general, every variable x in $\widetilde{F}$ becomes a *pair* of variables x (for x's value) and x^{shp} (for x's shape) in DPS-$\widetilde{F}$. You can see this same phenomenon in rules (D-App) and (D-Abs), which deal with lambdas and application: we turn each lambda-bound argument x into *two* arguments x and x^{shp}.

Finally, in rule (D-App) the context destination memory r is passed on to the function being called, as its additional first argument; and in (D-Abs) each lambda gets an additional argument, which is used as the destination when translating the body of the lambda. Figure 7 also gives a translation of an $\widetilde{F}$ type T to the corresponding DPS-$\widetilde{F}$ type D.

For variables there are two cases. In rule (D-VarScalar) a scalar variable is translated to itself, while in rule (D-VarVector) we must copy the array into the designated result storage using the copy function. The copy function copies the array elements as well as the header information (the second argument) into the given storage (the first argument).

3.3 Shape Translation

As we have seen, rule (D-Let) relies on the *shape translation* of the right hand side. This translation is given in Figure 8. If e has type T, then $\mathcal{S}[\![e]\!]$ is an expression of type $\mathcal{S}_{\mathcal{T}}[\![T]\!]$ that gives the shape of e. This expression can always be evaluated without allocation.

A *shape* is an expression of type Shp (Figure 5), whose values are given by P in that figure. There are three cases to consider. First, a scalar value has shape $\circ$ (rules (S-ExpNum), (S-ExpBool)). Second, when e is an array, $\mathcal{S}[\![e]\!]$ gives the shape of the array as a nested tuple, such as $(3, (4, \circ))$ for a 3-vector of 4-vectors. So the "shape" of an array specifies the cardinality of each dimension. Finally, when e is a function, $\mathcal{S}[\![e]\!]$ is a function that takes the shapes of its arguments and returns the shape of its result. You can see this directly in rule (S-App): to compute the shape of (the result of) a call, apply the shape-translation of the function to the shapes of the arguments. This is possible because $\widetilde{F}$ programs do not allow

	$\mathcal{D}[\![\mathrm{e}]\!]\mathrm{r}$	=	t
(D-App)	$\mathcal{D}[\![\mathrm{e_0\ x_1\ ...\ x}_k]\!]\mathrm{r}$	=	$(\mathcal{D}[\![\mathrm{e_0}]\!]\bullet)\ \mathrm{r\ x_1\ ...\ x}_k\ \mathrm{x_1}^{shp}\ ...\ \mathrm{x}_k{}^{shp}$
(D-Abs)	$\mathcal{D}[\![\lambda\,\mathrm{x_1\ ...\ x}_k.\ \mathrm{e_1}]\!]\bullet$	=	$\lambda\,\mathrm{r_2\ x_1\ ...\ x}_k\ \mathrm{x_1}^{shp}\ ...\ \mathrm{x}_k{}^{shp}.\ \mathcal{D}[\![\mathrm{e_1}]\!]\mathrm{r_2}$
(D-VarScalar)	$\mathcal{D}[\![\mathrm{x}]\!]\bullet$	=	x
(D-VarVector)	$\mathcal{D}[\![\mathrm{x}]\!]\mathrm{r}$	=	`copy r x`
(D-Let)	$\mathcal{D}[\![$`let x =` $\mathrm{e_1}$ `in` $\mathrm{e_2}]\!]\mathrm{r}$	=	`let` $\mathrm{x}^{shp} = \mathcal{S}[\![\mathrm{e_1}]\!]$ `in` `alloc (bytes` x^{shp}`)` $(\lambda\,\mathrm{r_2}.$ `let x =` $\mathcal{D}[\![\mathrm{e_1}]\!]\mathrm{r_2}$ `in` $\mathcal{D}[\![\mathrm{e_2}]\!]\mathrm{r})$
(D-If)	$\mathcal{D}[\![$`if` $\mathrm{e_1}$ `then` $\mathrm{e_2}$ `else` $\mathrm{e_3}]\!]\mathrm{r}$	=	`if` $\mathcal{D}[\![\mathrm{e_1}]\!]\bullet$ `then` $\mathcal{D}[\![\mathrm{e_2}]\!]\mathrm{r}$ `else` $\mathcal{D}[\![\mathrm{e_3}]\!]\mathrm{r}$
	$\mathcal{D}_\mathcal{T}[\![\mathrm{T}]\!]$	=	D
(DT-Fun)	$\mathcal{D}_\mathcal{T}[\![\mathrm{T_1, ..., T}_k \Rightarrow \mathrm{M}]\!]$	=	`Ref`, $\mathcal{D}_\mathcal{T}[\![\mathrm{T_1}]\!], ..., \mathcal{D}_\mathcal{T}[\![\mathrm{T}_k]\!], \mathcal{S}_\mathcal{T}[\![\mathrm{T_1}]\!], ..., \mathcal{S}_\mathcal{T}[\![\mathrm{T}_k]\!] \Rightarrow \mathcal{D}_\mathcal{T}[\![\mathrm{M}]\!]$
(DT-Mat)	$\mathcal{D}_\mathcal{T}[\![\mathrm{M}]\!]$	=	M
(DT-Bool)	$\mathcal{D}_\mathcal{T}[\![$`Bool`$]\!]$	=	`Bool`
(DT-Card)	$\mathcal{D}_\mathcal{T}[\![$`Card`$]\!]$	=	`Card`

Figure 7. Translation from $\widetilde{\mathrm{F}}$ to DPS-$\widetilde{\mathrm{F}}$

the programmer to write a function whose result size depends on the contents of its input array.

What is the shape-translation of a function f? Remembering that every in-scope variable f has become a pair of variables—one for the value and one for the shape—we can simply use the latter, f^{shp}, as we see in rule (S-Var).

For arrays, could the shape be simply the number of bytes required for the array, rather than a nested tuple? No. Consider the following function, which returns the first row of its argument matrix:

firstRow = λ m: `Array<Array<Double>>`. m[0]

The shape translation of firstRow, namely $\mathrm{firstRow}^{shp}$, is given the shape of m, and must produce the shape of m's first row. It cannot do that given only the number of bytes in m; it must know how many rows and columns it has. But by defining shapes as a nested tuple, it becomes easy: see rule (S-Get).

The shape of the result of the iteration construct (`ifold`) requires the shape of the state expression to remain the same across iterations, which is by checking the beta equivalence of the initial shape and the shape of each iteration. Otherwise the compiler produces an error, as shown in rule (S-Ifold).

The other rules are straightforward. *The key point is that by translating every in-scope variable, including functions, into a pair of variables, we can give a* compositional *account of shape translation, even in a higher order language.*

3.4 An Example

Using this translation, the running example at the beginning of Section 3.2 is translated as follows:

```
f = λ r₀ vec1 vec2 vec1^shp vec2^shp.
  let tmp^shp = vectorAdd^shp vec1^shp vec2^shp in
  alloc (bytes tmp^shp) (λ r₁.
    let tmp =
      vectorAdd r₁ vec1 vec2 vec1^shp vec2^shp in
    vectorNorm r₀ tmp tmp^shp
  )
```

The shape translations of some $\widetilde{\mathrm{F}}$ functions from Figure 3 are as follows:

```
let vectorRange^shp = λ n^shp. (n^shp, (λ i^shp. ∘) ∘)
let vectorMap2^shp = λ v1^shp v2^shp f^shp.
  (fst v1^shp, (λ i^shp. ∘) ∘)
let vectorAdd^shp = λ v1^shp v2^shp.
  vectorMap2^shp v1^shp v2^shp (λ a^shp b^shp. ∘)
let vectorNorm^shp = λ v^shp. ∘
```

3.5 Simplification

As is apparent from the examples in the previous section, code generated by the translation has many optimisation opportunities. This optimisation, or simplification, is applied in three stages: 1) $\widetilde{\mathrm{F}}$ expressions, 2) translated Shape-$\widetilde{\mathrm{F}}$ expressions, and 3) translated DPS-$\widetilde{\mathrm{F}}$ expressions. In the first stage, $\widetilde{\mathrm{F}}$ expressions are simplified to exploit fusion opportunities that remove intermediate arrays entirely. Furthermore, other compiler transformations such as constant folding, dead-code elimination, and common-subexpression elimination are also applied at this stage.

In the second stage, the Shape-$\widetilde{\mathrm{F}}$ expressions are simplified. The simplification process for these expressions mainly involves partial evaluation. By inlining all shape functions, and performing β-reduction and constant folding, shapes can often be computed at compile time, or at least can be greatly simplified. For example, the shape translations presented in Section 3.3 after performing simplification are as follows:

```
let vectorRange^shp = λ n^shp. (n^shp, ∘)
let vectorMap2^shp = λ v1^shp v2^shp f^shp. v1^shp
let vectorAdd^shp = λ v1^shp v2^shp. v1^shp
let vectorNorm^shp = λ v^shp. ∘
```

The final stage involves both partially evaluating the shape expressions in DPS-$\widetilde{\mathrm{F}}$ and simplifying the storage accesses in the DPS-$\widetilde{\mathrm{F}}$ expressions. Figure 9 demonstrates simplification rules for storage accesses. The first two rules remove empty allocations and

$$\mathcal{S}[\![\mathrm{e}]\!] = \mathrm{s}$$

(S-App)	$\mathcal{S}[\![\mathrm{e_0\ e_1\ ...\ e}_k]\!]$	=	$\mathcal{S}[\![\mathrm{e_0}]\!]\ \mathcal{S}[\![\mathrm{e_1}]\!]\ ...\ \mathcal{S}[\![\mathrm{e}_k]\!]$
(S-Abs)	$\mathcal{S}[\![\lambda\, x_1{:}\ T_1, ..., x_k{:}\ T_k.\ \mathrm{e}]\!]$	=	$\lambda\, {x_1}^{shp}{:}\ \mathcal{S_T}[\![T_1]\!], ..., {x_k}^{shp}{:}\ \mathcal{S_T}[\![T_k]\!].\ \mathcal{S}[\![\mathrm{e}]\!]$
(S-Var)	$\mathcal{S}[\![\mathrm{x}]\!]$	=	x^{shp}
(S-Let)	$\mathcal{S}[\![\texttt{let}\ \mathrm{x = e_1}\ \texttt{in}\ \mathrm{e_2}]\!]$	=	$\texttt{let}\ \mathrm{x}^{shp} = \mathcal{S}[\![\mathrm{e_1}]\!]\ \texttt{in}\ \mathcal{S}[\![\mathrm{e_2}]\!]$
(S-If)	$\mathcal{S}[\![\texttt{if}\ \mathrm{e_1}\ \texttt{then}\ \mathrm{e_2}\ \texttt{else}\ \mathrm{e_3}]\!]$	=	$\begin{cases}\mathcal{S}[\![\mathrm{e_2}]\!] & \mathcal{S}[\![\mathrm{e_2}]\!]\cong\mathcal{S}[\![\mathrm{e_3}]\!]\\ \text{Compilation Error!} & \mathcal{S}[\![\mathrm{e_2}]\!]\ncong\mathcal{S}[\![\mathrm{e_3}]\!]\end{cases}$
(S-ExpNum)	$\mathrm{e{:}\ Num} \vdash \mathcal{S}[\![\mathrm{e}]\!]$	=	$\circ$
(S-ExpBool)	$\mathrm{e{:}}\ \texttt{Bool} \vdash \mathcal{S}[\![\mathrm{e}]\!]$	=	$\circ$
(S-ValCard)	$\mathcal{S}[\![\mathrm{N}]\!]$	=	N
(S-AddCard)	$\mathcal{S}[\![\mathrm{e_0} +^c \mathrm{e_1}]\!]$	=	$\mathcal{S}[\![\mathrm{e_0}]\!] +^c \mathcal{S}[\![\mathrm{e_1}]\!]$
(S-MulCard)	$\mathcal{S}[\![\mathrm{e_0} *^c \mathrm{e_1}]\!]$	=	$\mathcal{S}[\![\mathrm{e_0}]\!] *^c \mathcal{S}[\![\mathrm{e_1}]\!]$
(S-Build)	$\mathcal{S}[\![\texttt{build}\ \mathrm{e_0\ e_1}]\!]$	=	$(\mathcal{S}[\![\mathrm{e_0}]\!], (\mathcal{S}[\![\mathrm{e_1}]\!]\ \circ))$
(S-Get)	$\mathcal{S}[\![\mathrm{e_0}[\mathrm{e_1}]]\!]$	=	$\texttt{snd}\ \mathcal{S}[\![\mathrm{e_0}]\!]$
(S-Length)	$\mathcal{S}[\![\texttt{length}\ \mathrm{e_0}]\!]$	=	$\texttt{fst}\ \mathcal{S}[\![\mathrm{e_0}]\!]$
(S-Ifold)	$\mathcal{S}[\![\ \texttt{ifold}\ \mathrm{e_1\ e_2\ e_3}\ ]\!]$	=	$\begin{cases}\mathcal{S}[\![\mathrm{e_2}]\!] & \forall n.\mathcal{S}[\![\mathrm{e_1\ e_2\ n}]\!]\cong\mathcal{S}[\![\mathrm{e_2}]\!]\\ \text{Compilation Error!} & \text{otherwise}\end{cases}$

$$\mathcal{S_T}[\![\mathrm{T}]\!] = \mathrm{S}$$

(ST-Fun)	$\mathcal{S_T}[\![\mathrm{T_1, T_2, ..., T}_k \Rightarrow \mathrm{M}\ ]\!]$	=	$\mathcal{S_T}[\![\mathrm{T_1}]\!], \mathcal{S_T}[\![\mathrm{T_2}]\!], ..., \mathcal{S_T}[\![\mathrm{T}_k]\!] \Rightarrow \mathcal{S_T}[\![\mathrm{M}]\!]$
(ST-Num)	$\mathcal{S_T}[\![\texttt{Num}]\!]$	=	`Card`
(ST-Bool)	$\mathcal{S_T}[\![\texttt{Bool}]\!]$	=	`Card`
(ST-Card)	$\mathcal{S_T}[\![\texttt{Card}]\!]$	=	`Card`
(ST-Vector)	$\mathcal{S_T}[\![\texttt{Array<M>}]\!]$	=	$(\texttt{Card}\ \texttt{*}\ \mathcal{S_T}[\![\mathrm{M}]\!])$

Figure 8. Shape Translation of $\widetilde{\mathrm{F}}$

merge consecutive allocations, respectively. The third rule removes a dead allocation, i.e. an allocation for which its storage is never used. The fourth rule hoists an allocation outside an abstraction whenever possible. The benefit of this rule is amplified more in the case that the storage is allocated inside a loop (`build` or `ifold`). Note that none of these transformation rules are available in $\widetilde{\mathrm{F}}$, due to the lack of explicit storage facilities.

After applying the presented simplification process, our working example is translated to the following program:

$\mathrm{f} = \lambda\, \mathrm{r_0\ vec1\ vec2\ vec1}^{shp}\ \mathrm{vec2}^{shp}.$
  $\texttt{alloc}\ (\texttt{bytes}\ \mathrm{vec1}^{shp})\ (\lambda\, \mathrm{r_1}.$
    $\texttt{let}\ \mathrm{tmp = vectorAdd\ r_1\ vec1\ vec2}$
        $\mathrm{vec1}^{shp}\ \mathrm{vec2}^{shp}\ \texttt{in}$
    $\mathrm{vectorNorm\ r_0\ tmp\ vec1}^{shp}$
)

In this program, there is no shape computation at runtime.

Empty Allocation:
$\texttt{alloc}\ \circ\ (\lambda\, \mathrm{r.\ t_1}) \rightsquigarrow \mathrm{t_1}[\mathrm{r} \mapsto \bullet]$

Allocation Merging:
$\texttt{alloc}\ \mathrm{t_1}\ (\lambda\, \mathrm{r_1}.\ \texttt{alloc}\ \mathrm{t_2}\ (\lambda\, \mathrm{r_2}.\ \mathrm{t_3}\ )) \rightsquigarrow \texttt{alloc}\ (\mathrm{t_1} +^c \mathrm{t_2})\ (\lambda\, \mathrm{r_1}.\ \texttt{let}\ \mathrm{r_2} = \texttt{stgOff}\ \mathrm{r_1\ t_1}\ \texttt{in}\ \mathrm{t_3}\ )$

Dead Allocation:
$\texttt{alloc}\ \mathrm{t_1}\ (\lambda\, \mathrm{r.\ t_2}) \rightsquigarrow \mathrm{t_2}$ if $\mathrm{r} \notin FV(\mathrm{t_2})$

Allocation Hoisting:
$\lambda x.\ \texttt{alloc}\ \mathrm{t_1}\ (\lambda\, \mathrm{r.\ t_2}) \rightsquigarrow \texttt{alloc}\ \mathrm{t_1}\ (\lambda\, \mathrm{r}.\ \lambda x.\ \mathrm{t_2})$ if $x \notin FV(\mathrm{t_1})$

Cardinality Simpl.:
$\texttt{bytes}\ \circ \rightsquigarrow \circ$
$\texttt{bytes}\ (\circ, \circ) \rightsquigarrow \circ$
$\texttt{bytes}\ (\mathrm{N}, \circ) \rightsquigarrow \texttt{NUM\_BYTES} *^c \mathrm{N} +^c \texttt{HDR\_BYTES}$
$\texttt{bytes}\ (\mathrm{N, s}) \rightsquigarrow (\texttt{bytes}\ \mathrm{s}) *^c \mathrm{N} +^c \texttt{HDR\_BYTES}$

Figure 9. Simplification rules of DPS-$\widetilde{\mathrm{F}}$

3.6 Properties of Shape Translation

The target language of shape translation is a subset of DPS-$\widetilde{\mathrm{F}}$ called Shape-$\widetilde{\mathrm{F}}$. The syntax of the subset is given in Figure 10. It includes nested pairs, of statically-known depth, to represent shapes, but it does not include vectors. That provides an important property for Shape-$\widetilde{\mathrm{F}}$ as follows:

Theorem 1. *All expressions resulting from shape translation, do not require any heap memory allocation.*

Proof. All the non-shape expressions have either scalar or function type. As shown in Figure 8 all scalar type expressions are translated into zero cardinality ($\circ$), which can be stack-allocated. On the other hand, the function type expressions can also be stack allocated. This is because functions are not allowed to return functions. Hence, the captured environment in a closure does not escape its scope. Hence, the closure environment can be stack allocated. Finally, the last case consists of expressions which are the result of shape translation for vector expressions. As we know the number

$$
\begin{array}{rcl}
\mathrm{s} & ::= & \mathrm{s}\,\overline{\mathrm{s}} \mid \lambda\overline{\mathrm{x}}.\,\mathrm{s} \mid \mathrm{x} \mid \mathrm{P} \mid \mathrm{c} \mid \texttt{let}\ \mathrm{x} = \mathrm{s}\ \texttt{in}\ \mathrm{s} \\
\mathrm{P} & ::= & \circ \mid \mathrm{N} \mid (\mathrm{N}, \mathrm{P}) \\
\mathrm{c} & ::= & \texttt{vecShp} \mid \texttt{fst} \mid \texttt{snd} \mid +^c \mid *^c \\
\mathrm{S} & ::= & \overline{\mathrm{S}} \Rightarrow \mathrm{Shp} \mid \mathrm{Shp} \\
\mathrm{Shp} & ::= & \texttt{Card} \mid (\texttt{Card} * \mathrm{Shp})
\end{array}
$$

Figure 10. Shape-$\widetilde{\mathrm{F}}$ syntax, which is a subset of the syntax of DPS-$\widetilde{\mathrm{F}}$ presented in Figure 5.

of dimensions of the original vector expressions, the translated expressions are tuples with a known-depth, which can be easily allocated on stack.

Next, we show the properties of our translation algorithm. First, let us investigate the impact of shape translation on $\widetilde{\mathrm{F}}$ types. For array types, we need to represent the shape in terms of the shape of each element of the array, and the cardinality of the array. We encode this information as a tuple. For scalar type and cardinality type expressions, the shape is a cardinality expression. This is captured in the following theorem:

Theorem 2. *If the expression e in $\widetilde{\mathrm{F}}$ has the type* T*, then* $\mathcal{S}[\![e]\!]$ *has type* $\mathcal{S}_{\mathcal{T}}[\![\mathrm{T}]\!]$.

Proof. Can be proved by induction on the translation rules from $\widetilde{\mathrm{F}}$ to Shape-$\widetilde{\mathrm{F}}$.

In order to have a simpler shape translation algorithm as well as better guarantees about the expressions resulting from shape translation, two important restrictions are applied on $\widetilde{\mathrm{F}}$ programs.

1. The accumulating function used in the `ifold` operator should preserve the shape of the initial value. Otherwise, converting the result shape into a closed-form polynomial expression requires solving a recurrence relation.
2. The shape of both branches of a conditional should be the same.

These two restrictions simplify the shape translation as is shown in Figure 8.

Theorem 3. *All expressions resulting from shape translation require linear computation time with respect to the size of terms in the original $\widetilde{F}$ program.*

Proof. This can be proved in two steps. First, translating a $\widetilde{\mathrm{F}}$ expression into its shape expression, leads to an expression with smaller size. This can be proved by induction on translation rules. Second, the run time of a shape expression is linear in terms of its size. An important case is the `ifold` construct, which by applying the mentioned restrictions, we ensured their shape can be computed without any need for recursion.

Finally, we believe that our translation is correct based on our successful implementation. However, we leave a formal semantics definition and the proof of correctness of the transformation as future work.

3.7 Discussion

One possible question is whether the DPS technique can go beyond the $\widetilde{\mathrm{F}}$ language. In other words, is it possible to support programs which require an arbitrary recursion, such as filtering an array, changing the size while recursing, or computing a Fibonacci-size array?

The answer is yes; instead of producing compilation errors (cf. Figure 8), the compiler produces warnings and postpones the shape computation until the run time. However, this can cause a massive run time overhead, as it is no longer possible to benefit from the performance guarantees mentioned in Section 3.6. More specifically, the shape computation could be as time consuming as the original array expressions [18], which can cause massive computation and space overheads. As an example, the computation complexity of a Fibonacci-size array will be $O(2.7^n)$ instead of $O(1.6^n)$ (the former is the closed form of $f(n) = 2f(n-1) + 2f(n-2)$, while the latter is the closed form of $f(n) = f(n-1) + f(n-2)$).

4 Implementation

4.1 $\widetilde{\mathrm{F}}$ Language

We implemented $\widetilde{\mathrm{F}}$ as a subset of F#. Hence $\widetilde{\mathrm{F}}$ programs are normal F# programs. Furthermore, the built-in constants (presented in Figure 2) are defined as a library in F# and all library functions (presented in Figure 3) are implemented using these built-in constants. If a given expression is in the subset supported by $\widetilde{\mathrm{F}}$, the compiler accepts it.

For implementing the transformations presented in the previous sections, instead of modifying the F# compiler, we use F# quotations [34]. Note that there is no need for the user to use F# quotations in order to implement a $\widetilde{\mathrm{F}}$ program. The F# quotations are only used by the compiler developer in order to implement transformation passes.

Although $\widetilde{\mathrm{F}}$ expressions are F# expressions, it is not possible to express memory management constructs used by DPS-$\widetilde{\mathrm{F}}$ expressions using the F# runtime. Hence, after translating $\widetilde{\mathrm{F}}$ expressions to DPS-$\widetilde{\mathrm{F}}$, we compile down the result program into a programming language which provides memory management facilities, such as C. The generated C code can either be used as kernels by other C programs, or invoked in F# as a native function using interoperatorability facilities provided by Common Language Runtime (CLR).

Next, we discuss why we choose C and how the C code generation works.

4.2 C Code Generation

There are many programming languages which provide manual memory management. Among them we are interested in the ones which give us full control on the runtime environment, while still being easy to debug. Hence, low-level imperative languages such as C and C++ are better candidates than LLVM mainly because of debugging purposes.

One of the main advantages of DPS-$\widetilde{\mathrm{F}}$ is that we can generate idiomatic C from it. More specifically, the generated C code is similar to a handwritten C program as we can manage the memory in a stack fashion. The translation from DPS-$\widetilde{\mathrm{F}}$ programs into C code is quite straightforward.

As our DPS encoded programs are using the memory in a stack fashion, the memory could be managed more efficiently. More specifically, we first allocate a specific amount of buffer in the beginning. Then, instead of using the standard `malloc` function, we bump-allocate from our already allocated buffer. Hence, in most cases allocating memory is only a pointer arithmetic operation to advance the pointer to the last allocated element of the buffer. In the cases that the user needs more than the amount which is allocated in the buffer, we need to double the size of the buffer. Furthermore, memory deallocation is also very efficient in this scheme. Instead of invoking the `free` function, we need to only decrement the pointer to the last allocated storage.

We compile lambdas by performing closure conversion. As functions in DPS-$\widetilde{F}$ do not return functions, the environment captured by a closure can be stack allocated.

As mentioned in Section 2, polymorphism is not allowed except for some built-in constructs in the language (e.g. `build` and `ifold`). Hence, all the usages of these constructs are monomorphic, and the C code generator knows exactly which code to generate for them. Furthermore, the C code generator does not need to perform the closure conversion for the lambdas passed to the built-in constructs. Instead, it can generate an efficient for-loop in place. As an example, the generated C code for a running sum function of $\widetilde{F}$ is as follows:

```
double vector_sum(vector v) {
  double sum = 0;
  for (index idx = 0; idx < v->length; idx++) {
    sum = sum + v->elements[idx];
  }
  return sum;
}
```

Finally, for the `alloc` construct in DPS-$\widetilde{F}$, the generated C code consists of three parts. First, a memory allocation statement is generated which allocates the given amount of storage. Second, the corresponding body of code which uses the allocated storage is generated. Finally, a memory deallocation statement is generated which frees the allocated storage. The generated C code for our working example is as follows:

```
double f(storage r0, vector vec1, vector vec2,
         vec_shape vec1_shp, vec_shape vec2_shp) {
  storage r1 = malloc(vector_bytes(vec1_shp));
  vector tmp = vector_add_dps(r1, vec1, vec2, vec1_shp,
     vec2_shp);
  double result = vector_norm_dps(r0,tmp,vec1_shp);
  free(r1);
  return result;
}
```

We use our own implementation of `malloc` and `free` for bump allocation.

5 Experimental Results

For the experimental evaluation, we use an iMac machine equipped with an Intel Core i5 CPU running at 2.7GHz, 32GB of DDR3 RAM at 1333Mhz. The operating system is OS X 10.10.5. We use Mono 4.6.1 as the runtime system for F# programs and CLang 700.1.81 for compiling the C++ code and generated C.[2]

Throughout this section, we compare the performance and memory consumption of the following alternatives:

- **F#:** Using the array operations (e.g. map) provided in the standard library of F# to implement vector operations.
- **CL: Leaky C code**, which is the generated C code from $\widetilde{F}$, using malloc to allocate vectors, never calling free.
- **CG: C code using Boehm GC**, which is the generated C code from $\widetilde{F}$, using `GC_malloc` of Boehm GC to allocate vectors.
- **CLF: CL + Fused Loops**, performs deforestation and loop fusion before CL.
- **D: DPS C code using system-provided malloc/free**, translates $\widetilde{F}$ programs into DPS-$\widetilde{F}$ before generating C code. Hence, the generated C code frees all allocated vectors. In this variant, the malloc and free functions are used for memory management.
- **DF: D + Fused Loops**, which is similar to the previous one, but performs deforestation before translating to DPS-$\widetilde{F}$.
- **DFB: DF + Buffer Optimizations**, which performs the buffer optimizations described in Section 3.5 (such as allocation hoisting and merging) on DPS-$\widetilde{F}$ expressions.
- **DFBS: DFB using stack allocator**, same as DFB, but using bump allocation for memory management, as previously discussed in Section 4.2. This is the best C code we generate from $\widetilde{F}$.
- **C++: Idiomatic C++**, which uses an handwritten C++ vector library, depending on C++14 move construction and copy elision for performance, with explict programmer indication of fixed-size (known at compile time) vectors, permitting stack allocation.
- **E++: Eigen C++**, which uses the Eigen [13] library which is implemented using C++ expression templates to effect loop fusion and copy elision. Also uses explicit sizing for fixed-size vectors.

First, we investigate the behavior of several variants of generated C code for two micro benchmarks. More specifically we see how DPS improves both run-time performance and memory consumption (by measuring the maximum resident set size) in comparison with an F# version. The behavior of the generated DPS code is very similar to manually handwritten C++ code and the Eigen library.

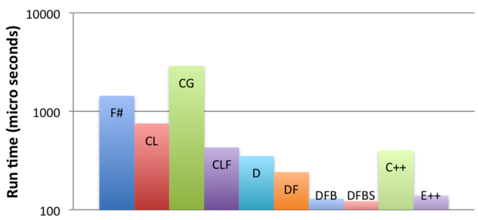
Then, we demonstrate the benefit of using DPS for some real-life computer vision and machine learning workloads motivated in [30]. Based on the results for these workloads, we argue that using DPS is a great choice for generating C code for numerical workloads, such as computer vision algorithms, running on embedded devices with a limited amount of memory available.

5.1 Micro Benchmarks

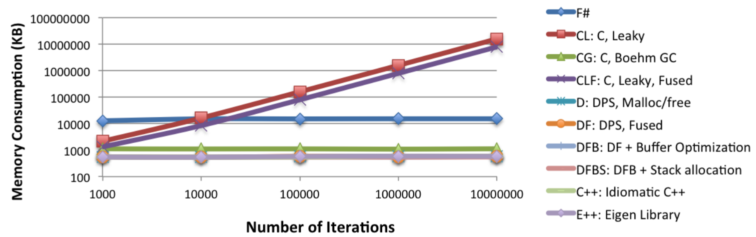
Figure 11 shows the experimental results for micro benchmarks, one adding three vectors, the second cross product of two vectors.

add3 : vectorAdd(vectorAdd(vec1, vec2), vec3)
in which all the vectors contain 100 elements. This program is run one million times in a loop, and timing results are shown in Figure 11a. In order to highlight the performance differences, the figure uses a logarithmic scale on its Y-axis. Based on these results we make the following observations. First, we see that all C and C++ programs are outperforming the F# program, except the one which uses the Boehm GC. This shows the overhead of garbage collection in the F# runtime environment and Boehm GC. Second, loop fusion has a positive impact on performance. This is because this program involves creating an intermediate vector (the one resulting from addition of vec1 and vec2). Third, the generated DPS C code which uses buffer optimizations (DFB) is faster than the one without this optimization (DF). This is mainly because the result vector is allocated only once for DFB whereas it is allocated once per iteration in DF. Finally, there is no clear advantage for C++ versions. This is mainly due to the fact that the vectors have sizes not known at compile time, hence the elements are not stack allocated. The Eigen version partially compensates this limitation by using vectorized operations, making the performance comparable to our best generated DPS C code.

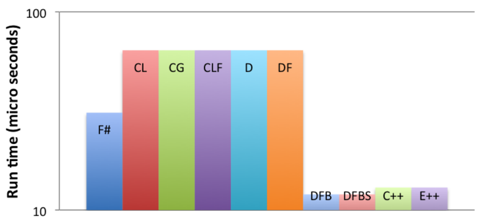
[2] All code and outputs are available at `http://github.com/awf/Coconut`.



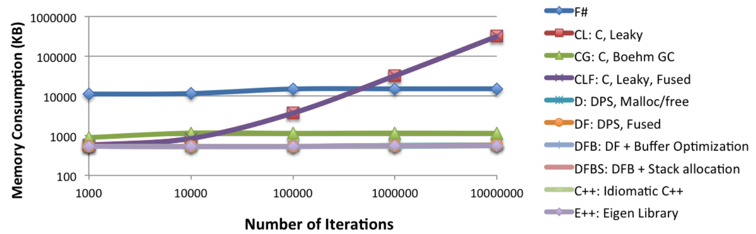
(a) Runtime performance comparison of different approaches on adding three vectors of 100 elements for one million times.



(b) Memory consumption comparison of different approaches on adding three vectors of 100 elements by varying the number of iterations. All the invisible lines are hidden under the bottom line.



(c) Runtime performance comparison of different approaches on cross product of two vectors of three elements for one million times.



(d) Memory consumption comparison of different approaches on cross product of two vectors of three elements by varying the number of iterations. All the invisible lines are hidden under the bottom line.

Figure 11. Experimental Results for Micro Benchmarks

The peak memory consumption of this program for different approaches is shown in Figure 11b. This measurement is performed by running this program by varying number of iterations. Both axes use logarithmic scales to better demonstrate the memory consumption difference. As expected, F# uses almost the same amount of memory over the time, due to GC. However, the runtime system sets the initial amount to 15MB by default. Also unsurprisingly, leaky C uses memory linear in the number of iterations, albeit from a lower base. The fused version of leaky C (CLF) decreases the consumed memory by a constant factor. Finally, DPS C, and C++ use a constant amount of space which is one order of magnitude less than the one used by the F# program, and half the amount used by the generated C code using Boehm GC.

cross : vectorCross(vec1, vec2)
This micro-benchmark is 1 million runs in which the two vectors contain 3 elements. Timing results are in Figure 11c. We see that the F# program is faster than the generated leaky C code, perhaps because garbage collection is invoked less frequently than in *add3*. Overall, in both cases, the performance of F# program and generated leaky C code is very similar. In this example, loop fusion does not have any impact on performance, as the program contains only one operator. As in the previous benchmark, all variants of generated DPS C code have a similar performance and outperform the generated leaky C code and the one using Boehm GC, for the same reasons. Finally, both handwritten and Eigen C++ programs have a similar performance to our generated C programs. For the case of this program, both C++ libraries provide fixed-sized vectors, which results in stack allocating the elements of the two vectors. This has a positive impact on performance. Furthermore, as there is no SIMD version of the cross operator, we do not observe a visible advantage for Eigen.

Finally, we discuss the memory consumption experiments of the second program, which is shown in Figure 11d. This experiment leads to the same observation as the one for the first program. However, as the second program does not involve creating any intermediate vector, loop fusion does not improve the peak memory consumption.

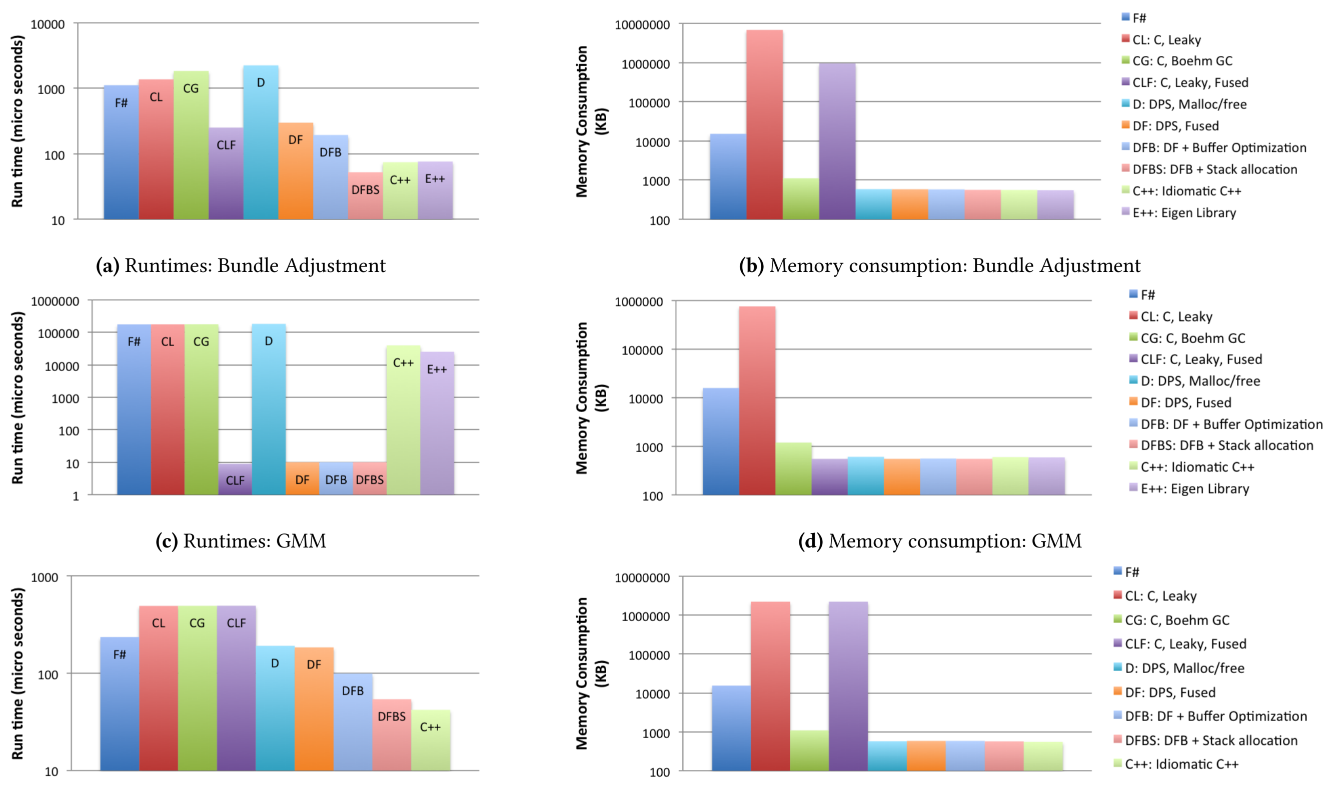
The presented micro benchmarks show that our DPS generated C code improves both performance and memory consumption by an order of magnitude in comparison with an equivalent F# program. Also, the generated DPS C code promptly deallocates memory which makes the peak memory consumption constant over the time, as opposed to a linear increase of memory consumption of the generated leaky C code. In addition, by using bump allocators the generated DPS C code can improve performance as well. Finally, we see that the generated DPS C code behaves very similarly to both handwritten and Eigen C++ programs.

5.2 Computer Vision and Machine Learning Workloads

In this section, we investigate the performance and memory consumption of real-life workloads.

Bundle Adjustment [38] is a computer vision problem which has many applications. In this problem, the goal is to optimize several parameters in order to have an accurate estimate of the projection of a 3D point by a camera. This is achieved by minimizing an objective function representing the reprojection error. This objective function is passed to a nonlinear minimizer as a function handle, and is typically called many times during the minimization.

One of the core parts of this objective function is the *project* function which is responsible for finding the projected coordinates



(a) Runtimes: Bundle Adjustment

(b) Memory consumption: Bundle Adjustment

(c) Runtimes: GMM

(d) Memory consumption: GMM

(e) Runtimes: Hand Tracking

(f) Memory consumption: Hand Tracking

Figure 12. Experimental Results for Computer Vision and Machine Learning Workloads

of a 3D point by a camera, including a model of the radial distortion of the lens. The $\widetilde{\mathrm{F}}$ implementation of this method is shown in Figure 13.

Figure 12a shows the runtime of different approaches after running *project* ten million times. First, the F# program performs similarly to the leaky generated C code and the C code using Boehm GC. Second, loop fusion improves speed fivefold. Third, the generated DPS C code is slower than the generated leaky C code, mainly due to costs associated with intermediate deallocations. However, this overhead is reduced by using bump allocation and performing loop fusion and buffer optimizations. Finally, we observe that the best version of our generated DPS C code marginally outperforms both C++ versions.

The peak memory consumption of different approaches for Bundle Adjustment is shown in Figure 12b. First, the F# program uses three orders of magnitude less memory in comparison with the generated leaky C code, which remains linear in the number of calls. This improvement is four orders of magnitude in the case of the generated C code using Boehm GC. Second, loop fusion improves the memory consumption of the leaky C code by an order of magnitude, due to removing several intermediate vectors. Finally, all generated DPS C variants as well as C++ versions consume the same amount of memory. The peak memory consumption of is an order of magnitude better than the F# baseline.

The Gaussian Mixture Model is a workhorse machine learning tool, used for computer vision applications such as image background modelling and image denoising, as well as semi-supervised learning.

In GMM, loop fusion can successfully remove all intermediate vectors. Hence, there is no difference between CL and CLF, or between DS and DSF, in terms of both performance and peak memory consumption as can be observed in Figure 12c and Figure 12d. Both C++ libraries behave three orders of magnitude worse than our fused and DPS generated code, due to the lack of support for fusion needed for GMM.

Due to the cost for performing memory allocation (and deallocation for DPS) at each iteration, the F# program, the leaky C code, and the generated DPS C code exhibit a worse performance than the fused and stack allocated versions. Furthermore, as the leaky C code does not deallocate the intermediate vectors, the consumed memory is increasing.

Hand tracking is a computer vision/computer graphics workload [35] that includes matrix-matrix multiplies, and numerous combinations of fixed- and variable-sized vectors and matrices. Figure 12e shows performance results of running one of the main functions of hand-tracking for 1 million times. As in the *cross* micro-benchmark we see no advantage for loop fusion, because in this function the intermediate vectors have multiple consumers. As above, generating DPS C code improves runtime performance,

```
let radialDistort = λ (radical: Vector) (proj: Vector).
  let rsq = vectorNorm proj
  let L = 1.0 + radical.[0] * rsq + radical.[1] * rsq * rsq
  vectorSMul proj L
let rodriguesRotate = λ (rotation: Vector) (x: Vector).
  let sqtheta = vectorNorm rotation
  if sqtheta != 0. then
    let theta = sqrt sqtheta
    let thetaInv = 1.0 / theta
    let w = vectorSMul rotation thetaInv
    let wCrossX = vectorCross w x
    let tmp = (vectorDot w x) * (1.0 - (cos theta))
    let v1 = vectorSMul x (cos theta)
    let v2 = vectorSMul wCrossX (sin theta)
    vectorAdd (vectorAdd v1 v2) (vectorSMul w tmp)
  else
    vectorAdd x (vectorCross rotation x)
let project = λ (cam: Vector) (x: Vector).
  let Xcam = rodriguesRotate (vectorSlice cam 0 2) (
    vectorSub x (vectorSlice cam 3 5) )
  let distorted = radialDistort (vectorSlice cam 9 10) (
      vectorSMul (vectorSlice Xcam 0 1) (1.0/Xcam.[2]) )
  vectorAdd (vectorSlice cam 7 8) (
    vectorSMul distorted cam.[6] )
```

Figure 13. Bundle Adjustment functions in $\widetilde{\mathrm{F}}$.

which is improved even more by using bump allocation and performing loop fusion and buffer optimizations. However, in this case the idiomatic C++ version outperforms the generated DPS C code. Figure 12f shows that DPS generated programs consume an order of magnitude less memory than the F# baseline, equal to the C++ versions.

6 Related Work

6.1 Programming Languages without GC

Functional programming languages without garbage collection dates back to Linear Lisp [2]. However, most functional languages (dating back to Lisp in around 1959) use garbage collection for managing memory.

Region-based memory management [37] was first introduced in ML and then in an extended version of C, called Cyclone [12], as an alternative or complementary technique to in order to remove the need for runtime garbage collection. This is achieved by allocating memory regions based on the liveness of objects. This approach improves both performance and memory consumption in many cases. However, in many cases the size of the regions is not known, whereas in our approach the size of each storage location is computed using the shape expressions. Also, in practice there are cases in which one needs to combine this technique with garbage collection [14], as well as cases in which the performance is still not satisfying [3, 36]. Furthermore, the complexity of region inference hinders the maintenance of the compiler, in addition to the overhead it causes for compilation time.

Safe [25, 26] suggests a simpler region inference algorithm by restricting the language to a first-order functional language. Also, linear regions [8] relax the stack discipline restriction on region-based memory management, due to certain usecases which use recursion and need an unbounded amount of memory. A Haskell implementation of this approach is given in [22]. The situation is similar for the linear types employed in Rust; due to loops it is not possible to enforce stack discipline for memory management. However, $\widetilde{\mathrm{F}}$ offers a restricted form of recursion, which always enforces a stack discipline for memory management.

6.2 Array Languages and Push-Arrays

APL [19] can be considered as the first array programming language. Futhark [16, 17] and SAC [11] are functional array programming languages. One interesting property of such languages is the support for fusion, which is achieved in $\widetilde{\mathrm{F}}$ by certain rewrite rules (cf. Figure 4). However, as this topic is out of the scope of this paper, we leave more discussion for future.

There is a close connection between so-called *push arrays* [1, 5, 33] and destination-passing style. A push-array is represented by an effectful function that, given an index and a value, will write the value into the array. This function closure captures the destination, so a program using push arrays is also using a form of destination-passing style. There are many differences, however. Our *functions* are transformed to destination-passing style, rather than our *arrays*. Our transformation is not array-specific, and can apply to any large object. Even though our basic array primitives are based on explicit indices, they are referentially transparent and may be read purely functionally. Our focus is on efficient allocation and freeing of array memory, which is not mentioned in the push-array literature. It may not be clear when the memory backing a push-array can be freed, whereas it is clear by construction in our work, and we guarantee to run without a garbage collector. Unsurprisingly, this guarantee comes with a limitation on expressiveness: we cannot handle operations such as filter, whose result size is data-dependent (cf. Section 3.7). Happily a large class of important applications can be expressed in our language, and enjoy its benefits.

There are many domain-specific languages (DSLs) for numerical workloads such as Halide [28], Diderot [4], and OptiML [31]. All these DSLs generate parallel code from their high-level programs. Furthermore, Halide [28] exploits the memory hierarchy by making tiling and scheduling decisions, similar to Spiral [27] and LGen [29]. Although both parallelism and improving the usage of a memory hierarchy are orthogonal concepts to translation into DPS, they are still interesting directions for $\widetilde{\mathrm{F}}$.

6.3 Estimation of Memory Consumption

One can use type systems for estimating memory consumption. Hofmann and Jost [18] enrich the type system with certain annotations and uses linear programming for the heap consumption inference. Another approach is to use sized types [39] for the same purpose.

Size slicing [15] uses a technique similar to ours for inferring the shape of arrays in the Futhark programming language. However, in $\widetilde{\mathrm{F}}$ we guarantee that shape inference is simplified and is based only on size computation, whereas in their case, they rely on compiler optimizations for its simplification and in some cases it can fall back to inefficient approaches which in the worst case could be as expensive as evaluating the original expression [18]. The FISh programming language [20] also makes shape information explicit in programs, and resolves the shapes at compilation time by using partial evaluation, which can also be used for checking shape-related errors [21]. Our shape translation (Section 3.3) is very similar to their shape analysis, but their purposes differ: theirs is an analysis, while ours generates for every function f a companion shape function that (without itself allocating) computes f's space needs; these companion functions are called at runtime to compute memory needs.

6.4 Optimizing Tail Calls

Destination-passing style was originally introduced in [23], then was encoded functionally in [24] by using linear types [42]. Walker and Morrisett [43] use extensions to linear type systems to support aliasing which is avoided in vanilla linear type systems. The idea of destination-passing style has many similarities to *tail-recursion modulo cons* [9, 40].

References

[1] Johan Anker and Josef Svenningsson. 2013. An EDSL approach to high performance Haskell programming. In *ACM Haskell Symposium*. 1–12.

[2] Henry G Baker. 1992. Lively linear lisp: 'look ma, no garbage!'. *ACM Sigplan notices* 27, 8 (1992), 89–98.

[3] Lars Birkedal, Mads Tofte, and Magnus Vejlstrup. 1996. From Region Inference to Von Neumann Machines via Region Representation Inference *(POPL '96)*. ACM, NY, USA, 171–183.

[4] Charisee Chiw, Gordon Kindlmann, John Reppy, Lamont Samuels, and Nick Seltzer. 2012. Diderot: A Parallel DSL for Image Analysis and Visualization *(PLDI '12)*. ACM, 111–120.

[5] Koen Claessen, Mary Sheeran, and Bo Joel Svensson. 2012. Expressive Array Constructs in an Embedded GPU Kernel Programming Language *(DAMP '12)*. ACM, NY, USA, 21–30.

[6] Duncan Coutts, Roman Leshchinskiy, and Don Stewart. Stream Fusion. From Lists to Streams to Nothing at All *(ICFP '07)*.

[7] Cormac Flanagan, Amr Sabry, Bruce F Duba, and Matthias Felleisen. 1993. The essence of compiling with continuations. In *ACM Sigplan Notices*, Vol. 28. ACM, 237–247.

[8] Matthew Fluet, Greg Morrisett, and Amal Ahmed. 2006. Linear regions are all you need *(ESOP '06)*. Springer, 7–21.

[9] D Friedman and S Wise. 1975. Unwinding stylized recursions into iterations. *Comput. Sci. Dep., Indiana University, Bloomington, IN, Tech. Rep* 19 (1975).

[10] Andrew Gill, John Launchbury, and Simon L Peyton Jones. 1993. A short cut to deforestation *(FPCA)*. ACM, 223–232.

[11] Clemens Grelck and Sven-Bodo Scholz. 2006. SAC—A Functional Array Language for Efficient Multi-threaded Execution. *Int. Journal of Parallel Programming* 34, 4 (2006), 383–427.

[12] Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. 2002. Region-based Memory Management in Cyclone *(PLDI '02)*. ACM, NY, USA, 282–293.

[13] Gaël Guennebaud, Benoit Jacob, and others. 2010. Eigen. *URl: http://eigen. tuxfamily. org* (2010).

[14] Niels Hallenberg, Martin Elsman, and Mads Tofte. 2002. Combining Region Inference and Garbage Collection *(PLDI '02)*. ACM, NY, USA, 141–152.

[15] Troels Henriksen, Martin Elsman, and Cosmin E. Oancea. 2014. Size Slicing: A Hybrid Approach to Size Inference in Futhark *(FHPC '14)*. ACM, New York, NY, USA, 31–42.

[16] Troels Henriksen and Cosmin E. Oancea. 2014. Bounds Checking: An Instance of Hybrid Analysis *(ARRAY '14)*. ACM, NY, USA.

[17] Troels Henriksen, Niels G. W. Serup, Martin Elsman, Fritz Henglein, and Cosmin E. Oancea. 2017. Futhark: Purely Functional GPU-programming with Nested Parallelism and In-place Array Updates *(PLDI 2017)*. ACM, New York, NY, USA, 556–571.

[18] Martin Hofmann and Steffen Jost. 2003. Static Prediction of Heap Space Usage for First-order Functional Programs *(POPL '03)*. ACM, New York, NY, USA, 185–197.

[19] Kenneth E Iverson. 1962. A Programming Language. In *Proceedings of the May 1-3, 1962, spring joint computer conference*. ACM, 345–351.

[20] C Barry Jay. 1999. Programming in FISh. *International Journal on Software Tools for Technology Transfer* 2, 3 (1999), 307–315.

[21] C. Barry Jay and Milan Sekanina. 1997. *Shape Checking of Array Programs*. Technical Report. In Computing: the Australasian Theory Seminar, Proceedings.

[22] Oleg Kiselyov and Chung-chieh Shan. 2008. Lightweight monadic regions. In *ACM Sigplan Notices*, Vol. 44. ACM, 1–12.

[23] James R Larus. 1989. *Restructuring symbolic programs for concurrent execution on multiprocessors*. Ph.D. Dissertation.

[24] Yasuhiko Minamide. 1998. A Functional Representation of Data Structures with a Hole *(POPL '98)*. 75–84.

[25] Manuel Montenegro, Ricardo Peña, and Clara Segura. 2008. A type system for safe memory management and its proof of correctness *(PPDP '08)*. ACM, 152–162.

[26] Manuel Montenegro, Ricardo Peña, and Clara Segura. 2009. A simple region inference algorithm for a first-order functional language. In *International Workshop on Functional and Constraint Logic Programming*. Springer, 145–161.

[27] Markus Puschel, José MF Moura, Jeremy R Johnson, David Padua, Manuela M Veloso, Bryan W Singer, Jianxin Xiong, Franz Franchetti, Aca Gacic, Yevgen Voronenko, and others. 2005. SPIRAL: Code generation for DSP transforms. *Proc. IEEE* 93, 2 (2005), 232–275.

[28] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines *(PLDI '13)*.

[29] Daniele G Spampinato and Markus Püschel. A basic linear algebra compiler for structured matrices. In *CGO '16*. ACM.

[30] Filip Srajer, Zuzana Kukelova, and Andrew Fitzgibbon. 2016. A Benchmark of Selected Algorithmic Differentiation Tools on Some Problems in Machine Learning and Computer Vision. (2016).

[31] Arvind Sujeeth, HyoukJoong Lee, Kevin Brown, Tiark Rompf, Hassan Chafi, Michael Wu, Anand Atreya, Martin Odersky, and Kunle Olukotun. 2011. OptiML: An Implicitly Parallel Domain-Specific Language for Machine Learning *(ICML '11)*. 609–616.

[32] Josef Svenningsson. 2002. Shortcut Fusion for Accumulating Parameters & Zip-like Functions *(ICFP '02)*. ACM, 124–132.

[33] Bo Joel Svensson and Josef Svenningsson. 2014. Defunctionalizing Push Arrays *(FHPC '14)*. ACM, NY, USA, 43–52.

[34] Don Syme. 2006. Leveraging .NET Meta-programming Components from F#: Integrated Queries and Interoperable Heterogeneous Execution *(ML '06)*. ACM, 43–54.

[35] Jonathan Taylor, Richard Stebbing, Varun Ramakrishna, Cem Keskin, Jamie Shotton, Shahram Izadi, Aaron Hertzmann, and Andrew Fitzgibbon. 2014. User-specific hand modeling from monocular depth sequences *(CVPR '14)*. 644–651.

[36] Mads Tofte, Lars Birkedal, Martin Elsman, and Niels Hallenberg. 2004. A Retrospective on Region-Based Memory Management. *Higher Order Symbol. Comput.* 17, 3 (Sept. 2004), 245–265.

[37] Mads Tofte and Jean-Pierre Talpin. 1997. Region-Based Memory Management. *Information and Computation* 132, 2 (1997).

[38] Bill Triggs, Philip F McLauchlan, Richard I Hartley, and Andrew W Fitzgibbon. 1999. Bundle adjustment—a modern synthesis. In *Inter. workshop on vision algorithms*. Springer, 298–372.

[39] Pedro B Vasconcelos. 2008. *Space cost analysis using sized types*. Ph.D. Dissertation. University of St Andrews.

[40] Philip Wadler. 1984. Listlessness is better than laziness: Lazy evaluation and garbage collection at compile-time. In *Proc. of ACM Symp. on LISP and functional programming*. 45–52.

[41] Philip Wadler. 1988. Deforestation: Transforming programs to eliminate trees. In *ESOP'88*. Springer, 344–358.

[42] Philip Wadler. 1990. Linear types can change the world. In *IFIP TC*, Vol. 2. Citeseer, 347–359.

[43] David Walker and Greg Morrisett. 2000. Alias types for recursive data structures. In *Inter. Workshop on Types in Compilation*. Springer, 177–206.