Using Destination-Passing Style to Compile a Functional Language into Efficient Low-Level Code

In submission

AMIR SHAIKHHA*, EPFL
ANDREW FITZGIBBON, Microsoft Research
SIMON PEYTON-JONES, Microsoft Research
DIMITRIOS VYTINIOTIS, Microsoft Research

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.

ACM Reference format:
Amir Shaikhha, Andrew Fitzgibbon, Simon Peyton-Jones, and Dimitrios Vytiniotis. 2017. Using Destination-Passing Style to Compile a Functional Language into Efficient Low-Level Code. *PACM Progr. Lang.* 1, 1, Article 1 (January 2017), 21 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Applications in computer vision, robotics, and machine learning [32, 35**?**] 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* (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.

That is where DPS takes over. The key idea 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 not longer needed. Of course 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 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 be able to

*This work was done while the author was at Microsoft Research, Cambridge.

$$
\begin{array}{rcll}
\mathrm{e} & ::= & \mathrm{e}\ \overline{\mathrm{e}} & \text{– Application} \\
& | & \lambda\overline{\mathrm{x}}.\mathrm{e} & \text{– Abstraction} \\
& | & \mathrm{x} & \text{– Variable Access} \\
& | & \mathrm{n} & \text{– Scalar Value} \\
& | & \mathrm{i} & \text{– Index Value} \\
& | & \mathrm{N} & \text{– Cardinality Value} \\
& | & \mathrm{c} & \text{– Constants} \\
& | & \texttt{let}\ \mathrm{x} = \mathrm{e}\ \texttt{in}\ \mathrm{e} & \text{– (Non-Rec.) Let Binding} \\
& | & \texttt{if}\ \mathrm{e}\ \texttt{then}\ \mathrm{e}\ \texttt{else}\ \mathrm{e} & \text{– Conditional} \\
\mathrm{c} & ::= & \texttt{build} \mid \texttt{reduce} \mid \texttt{length} \mid \texttt{get} & \\
& | & \textit{[More on Figure 2]} & \\
\mathrm{T} & ::= & \mathrm{M} & \text{– Matrix Type} \\
& | & \overline{\mathrm{T}} \Rightarrow \mathrm{M} & \text{– Function Types (No Currying)} \\
& | & \texttt{Card} & \text{– Cardinality Type} \\
& | & \texttt{Bool} & \text{– Boolean Type} \\
\mathrm{M} & ::= & \mathrm{Num} & \text{– Numeric Type} \\
& | & \texttt{Array<M>} & \text{– Vector, Matrix, ... Type} \\
\mathrm{Num} & ::= & \texttt{Double} & \text{– Scalar Type} \\
& | & \texttt{Index} & \text{– Index Type}
\end{array}
$$

Fig. 1. The core $\widetilde{\text{F}}$ syntax.

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 (c.f. 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, $\widetilde{\text{F}}$ (Section 2), and a *compositional* shape translation (Section 3.3) that guarantee to compute the result size of any $\widetilde{\text{F}}$ expression, either statically or at runtime, with no allocation, and a run-time cost independent of the data or its size (Section 3.6). We do not know any other language stack with these properties.
- We 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++.[1]

2 $\widetilde{\text{F}}$

$\widetilde{\text{F}}$ is a subset of F#, an ML-like functional programming language (the syntax 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.

[1]Keen C++ programmers may wonder what advantage we anticipate over C++. Primarily this is the future possibility of program transformations such as automatic differentiation, which are easily expressed in $\widetilde{\text{F}}$, but remain slow in C++ [28], and would be expected to generate code as efficient as our benchmarks indicate.

Typing Rules:

$$\text{(T-If)}\ \frac{\mathrm{e}_1 : \texttt{Bool} \quad \mathrm{e}_2 : \mathrm{M} \quad \mathrm{e}_3 : \mathrm{M}}{\texttt{if}\ \mathrm{e}_1\ \texttt{then}\ \mathrm{e}_2\ \texttt{else}\ \mathrm{e}_3 : \mathrm{M}} \qquad \text{(T-App)}\ \frac{\mathrm{e}_0 : \overline{\mathrm{T}} \Rightarrow \mathrm{M} \quad \overline{\mathrm{e}} : \overline{\mathrm{T}}}{\mathrm{e}_0\ \overline{\mathrm{e}} : \mathrm{M}}$$

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

$$\text{(T-Let)}\ \frac{\Gamma \vdash \mathrm{e}_1 : \mathrm{T}_1 \quad \Gamma, \mathrm{x} : \mathrm{T}_1 \vdash \mathrm{e}_2 : \mathrm{T}_2}{\Gamma \vdash \texttt{let}\ \mathrm{x} = \mathrm{e}_1\ \texttt{in}\ \mathrm{e}_2 \text{:}\ \mathrm{T}_2}$$

Scalar Function Constants:

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

Vector Function Constants:

`build`	:	`Card` , (`Index` ⇒ M) ⇒ `Array<M>`
`reduce`	:	(M , `Index` ⇒ M) , M , `Card` ⇒ M
`get`	:	`Array<M>` , `Index` ⇒ M
`length`	:	`Array<M>` ⇒ `Card`

Syntactic Sugars:

$\mathrm{e}_0[\mathrm{e}_1] = \texttt{get}\ \mathrm{e}_0\ \mathrm{e}_1$

for all binary operators bop: $\mathrm{e}_1\ \mathit{bop}\ \mathrm{e}_2 = \mathit{bop}\ \mathrm{e}_1\ \mathrm{e}_2$

Fig. 2. The type system and built-in constants of $\widetilde{\mathrm{F}}$

2.1 Syntax and types of $\widetilde{\mathrm{F}}$

In addition to the usual λ-calculus constructs (abstraction, application, and variable access), $\widetilde{\mathrm{F}}$ supports let binding and conditionals. In support of array programming, the language has several built-in functions defined: `build` for producing arrays; `reduce` 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.

The syntax of $\widetilde{\mathrm{F}}$ is shown in Figure 1, while the type system and several other built-in functions are shown in Figure 2. Note that Figure 1 shows an abstract syntax and parentheses can be used as necessary. Also, $\overline{\mathrm{x}}$ and $\overline{\mathrm{e}}$ denote one or more variables and expressions, respectively, which are separated by spaces, whereas, $\overline{\mathrm{T}}$ represents one or more types which are separated by commas.

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

- $\widetilde{\mathrm{F}}$ does not support arbitrary recursion, hence is not Turing Complete. Instead one can use `build` and `reduce` 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 `reduce`, 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 (c.f. Figure 2).

As an example, Figure 3 shows a linear algebra library defined using $\widetilde{\mathrm{F}}$. 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

```
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.
  reduce (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
```

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

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 *reducing* them. 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

Consider this function, which accepts two vectors and returns the norm of the vector resulting from the addition of these two vectors.

$$\text{f} = \lambda\text{vec1 vec2.}\;\; \text{vectorNorm (vectorAdd vec1 vec2)}$$

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 invokes vectorDot, which invokes vectorEMul in order to perform the element-wise multiplication between two vectors). In this example one can remove the intermediate vectors by *fusion* (or *deforestation*) [5, 10, 30, 39]. After fusion the function might look like this:

```
f = λvec1 vec2.
  reduce (λsum idx. sum + (vec1[idx]+vec2[idx]) * (vec1[idx]+vec2[idx])) 0 (length vec1)
```

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

Fusion is well studied, and we take it for granted in this paper. However, there are plenty of cases in which the intermediate array cannot be removed. For example: the intermediate array is passed to a foreign library function; it is passed to a library function that is too big to inline; or it is 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.

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 4, while its type system is in Figure 5. 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-$\widetilde{\text{F}}$:

$$
\begin{array}{rcll}
\mathrm{t} & ::= & \mathrm{t}\ \overline{\mathrm{t}} & \text{– Application} \\
& | & \lambda\overline{\mathrm{x}}.\ \mathrm{t} & \text{– Abstraction} \\
& | & \bullet & \text{– Empty Memory Location} \\
& | & \mathrm{n} & \text{– Scalar Value} \\
& | & \mathrm{i} & \text{– Index Value} \\
& | & \mathrm{P} & \text{– Shape Value} \\
& | & \mathrm{x} & \text{– Variable Access} \\
& | & \mathrm{r} & \text{– Reference Access} \\
& | & \mathrm{c} & \text{– Constants} \\
& | & \texttt{let}\ \mathrm{x} = \mathrm{t}\ \texttt{in}\ \mathrm{t} & \text{– (Non-Rec.) Let Binding} \\
& | & \texttt{if}\ \mathrm{t}\ \texttt{then}\ \mathrm{t}\ \texttt{else}\ \mathrm{t} & \text{– Conditional} \\
& | & \texttt{alloc}\ \mathrm{t}\ (\lambda\mathrm{r}.\ \mathrm{t}) & \text{– Memory Allocation} \\
\mathrm{P} & ::= & \circ & \text{– Zero Cardinality} \\
& | & \mathrm{N} & \text{– Cardinality Value} \\
& | & (\mathrm{N}, \mathrm{P}) & \text{– Vector Shape Value} \\
\mathrm{c} & ::= & \textit{[See Figure 5]} & \\
\mathrm{D} & ::= & \mathrm{M} \mid \overline{\mathrm{D}} \Rightarrow \mathrm{M}\ \mid \texttt{Bool} & \\
& | & \mathrm{Shp} & \text{– Shape Type} \\
& | & \texttt{Ref} & \text{– Machine Address} \\
\mathrm{M} & ::= & \mathrm{Num} \mid \texttt{Array<M>} & \\
\mathrm{Num} & ::= & \texttt{Double} \mid \texttt{Index} & \\
\mathrm{Shp} & ::= & \texttt{Card} & \text{– Cardinality Type} \\
& | & (\texttt{Card} * \mathrm{Shp}) & \text{– Vector Shape Type}
\end{array}
$$

Fig. 4. The core DPS-$\widetilde{\mathrm{F}}$ syntax.

$$
\begin{array}{l}
\mathrm{f} = \lambda \mathrm{r}_1\ \mathrm{vec1}\ \mathrm{vec2}.\ \ \texttt{alloc}\ (\mathrm{vecBytes\ vec1})\ (\lambda \mathrm{r}_2. \\
\quad \mathrm{vectorNorm\_dps}\ \bullet\ (\mathrm{vectorAdd\_dps}\ \mathrm{r}_2\ \mathrm{vec1}\ \mathrm{vec2}) \\
\ )
\end{array}
$$

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 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 $\widetilde{\mathrm{F}}$ to DPS-$\widetilde{\mathrm{F}}$

We now turn our attention to the translation from $\widetilde{\mathrm{F}}$ to DPS-$\widetilde{\mathrm{F}}$. Before translating $\widetilde{\mathrm{F}}$ expressions to their DPS form, the expressions should be transformed into a normal form similar to Administrative-Normal Form [7] (ANF). 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.[2] The representation of our working example in ANF is as follows:

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

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:

$\texttt{build}$: $\texttt{Ref}$, $\texttt{Card}$, $(\texttt{Ref}, \texttt{Index} \Rightarrow M)$, $\texttt{Card}$, $(\texttt{Card} \Rightarrow \text{Shp})$ $\Rightarrow \texttt{Array<}M\texttt{>}$

$\texttt{reduce}$: $\texttt{Ref}$, $(\texttt{Ref}, M, \texttt{Index} \Rightarrow M)$, M, $\texttt{Card}$, $(\text{Shp}, \texttt{Card} \Rightarrow \text{Shp})$, Shp, $\texttt{Card}$ $\Rightarrow M$

$\texttt{get}$: $\texttt{Ref}$, $\texttt{Array<}M\texttt{>}$, $\texttt{Index}$, Shp, $\texttt{Card} \Rightarrow M$

$\texttt{length}$: $\texttt{Ref}$, $\texttt{Array<}M\texttt{>}$, $\text{Shp} \Rightarrow \texttt{Card}$

$\texttt{copy}$: $\texttt{Ref}$, $\texttt{Array<}M\texttt{>} \Rightarrow \texttt{Array<}M\texttt{>}$

Scalar Function Constants:

DPS version of $\widetilde{F}$ Scalar Constants (See Figure 2).

$\texttt{stgOff}$	:	$\texttt{Ref}, \text{Shp} \Rightarrow \texttt{Ref}$
$\texttt{vecShp}$	:	$\texttt{Card}, \text{Shp} \Rightarrow (\texttt{Card} * \text{Shp})$
$\texttt{fst}$	:	$(\texttt{Card} * \text{Shp}) \Rightarrow \texttt{Card}$
$\texttt{snd}$	:	$(\texttt{Card} * \text{Shp}) \Rightarrow \text{Shp}$
$\texttt{bytes}$	:	$\text{Shp} \Rightarrow \texttt{Card}$

Syntactic Sugars:

$t_0.[t_1]\{r\} = \texttt{get}\ r\ t_0\ t_1 \quad \texttt{length}\, t = \texttt{length} \bullet t$

$(t_0, t_1) = \texttt{vecShp}\ t_0\ t_1$

for all binary ops bop: $e_1\ \textit{bop}\ e_2 = \textit{bop} \bullet e_1\ e_2$

Fig. 5. The type system and built-in constants of DPS-$\widetilde{F}$

$$\mathcal{D}[\![e]\!]r = t$$

(D-App)	$\mathcal{D}[\![e_0\ x_1\ ...\ x_k]\!]r$	$=$	$(\mathcal{D}[\![e_0]\!]\bullet)\ r\ x_1\ ...\ x_k\ {x_1}^{shp}\ ...\ {x_k}^{shp}$
(D-Abs)	$\mathcal{D}[\![\lambda x_1\ ...\ x_k.\ e_1]\!]\bullet$	$=$	$\lambda r_2\ x_1\ ...\ x_k\ {x_1}^{shp}\ ...\ {x_k}^{shp}.\ \ \mathcal{D}[\![e_1]\!]r_2$
(D-VarScalar)	$\mathcal{D}[\![x]\!]\bullet$	$=$	x
(D-VarVector)	$\mathcal{D}[\![x]\!]r$	$=$	$\texttt{copy}\ r\ x$
(D-Let)	$\mathcal{D}[\![\texttt{let}\ x = e_1\ \texttt{in}\ e_2]\!]r$	$=$	$\texttt{let}\ x^{shp} = \mathcal{S}[\![e_1]\!]\ \texttt{in}$ $\texttt{alloc}\ (\texttt{bytes}\ x^{shp})\ (\lambda r_2.$ $\texttt{let}\ x = \mathcal{D}[\![e_1]\!]r_2\ \texttt{in}\ \mathcal{D}[\![e_2]\!]r)$
(D-If)	$\mathcal{D}[\![\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3]\!]r$	$=$	$\texttt{if}\ \mathcal{D}[\![e_1]\!]\bullet\ \texttt{then}\ \mathcal{D}[\![e_2]\!]r\ \texttt{else}\ \mathcal{D}[\![e_3]\!]r$

$$\mathcal{D}_{\mathcal{T}}[\![T]\!] = D$$

(DT-Fun)	$\mathcal{D}_{\mathcal{T}}[\![T_1, ..., T_k \Rightarrow M]\!]$	$=$	$\texttt{Ref}, \mathcal{D}_{\mathcal{T}}[\![T_1]\!], ..., \mathcal{D}_{\mathcal{T}}[\![T_k]\!], \mathcal{S}_{\mathcal{T}}[\![T_1]\!], ..., \mathcal{S}_{\mathcal{T}}[\![T_k]\!] \Rightarrow \mathcal{D}_{\mathcal{T}}[\![M]\!]$
(DT-Mat)	$\mathcal{D}_{\mathcal{T}}[\![M]\!]$	$=$	M
(DT-Bool)	$\mathcal{D}_{\mathcal{T}}[\![\texttt{Bool}]\!]$	$=$	$\texttt{Bool}$
(DT-Card)	$\mathcal{D}_{\mathcal{T}}[\![\texttt{Card}]\!]$	$=$	$\texttt{Card}$

Fig. 6. Translation from $\widetilde{F}$ to DPS-$\widetilde{F}$

```
f = λvec1 vec2.
  let tmp = vectorAdd vec1 vec2 in
  vectorNorm tmp
```

Figure 6 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 $\mathcal{S}[\![\mathrm{e}_1]\!]$ 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.

In general, every variable x in $\widetilde{\mathrm{F}}$ becomes a *pair* of variables x (for x's value) and x^{shp} (for x's shape) in DPS-$\widetilde{\mathrm{F}}$. 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 destination memory r for the context is passed on to the function being called, as its additional first argument; and in (D-Abs) each lambda gets an additional first argument, which is used as the destination when translating the body of the lambda. Figure 6 also gives a translation of an $\widetilde{\mathrm{F}}$ type T to the corresponding DPS-$\widetilde{\mathrm{F}}$ 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 into the given storage.

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 7. If e has type T, then $\mathcal{S}[\![\mathrm{e}]\!]$ is an expression of type $\mathcal{S}_{\mathcal{T}}[\![\mathrm{T}]\!]$ that gives the shape of e. This expression can always be evaluated without allocation.

A *shape* is an expression of type Shp (Figure 4), 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, $\mathcal{S}[\![\mathrm{e}]\!]$ 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, $\mathcal{S}[\![\mathrm{e}]\!]$ 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 $\widetilde{\mathrm{F}}$ programs do not allow 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 = λm: `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 (`reduce`) requires the shape of the state expression to remain the same across iterations. Otherwise the compiler produce an error, as it is shown in rule (S-Reduce).

$$
\begin{array}{lrcl}
 & \mathcal{S}[\![\mathrm{e}]\!] & = & \mathrm{s} \\
 & & & \\
\text{(S-App)} & \mathcal{S}[\![\mathrm{e}_0\ \mathrm{e}_1\ ...\ \mathrm{e}_k\ ]\!] & = & \mathcal{S}[\![\mathrm{e}_0]\!]\ \mathcal{S}[\![\mathrm{e}_1]\!]\ ...\ \mathcal{S}[\![\mathrm{e}_k]\!] \\
\text{(S-Abs)} & \mathcal{S}[\![\lambda x_1\colon T_1,\ ...,\ x_k\colon T_k.\ \mathrm{e}\ ]\!] & = & \lambda {x_1}^{shp}\colon \mathcal{S}_{\mathcal{T}}[\![T_1]\!],\ ...,\ {x_k}^{shp}\colon \mathcal{S}_{\mathcal{T}}[\![T_k]\!].\ \mathcal{S}[\![\mathrm{e}]\!] \\
\text{(S-Var)} & \mathcal{S}[\![\mathrm{x}]\!] & = & \mathrm{x}^{shp} \\
\text{(S-Let)} & \mathcal{S}[\![\texttt{let}\ \mathrm{x} = \mathrm{e}_1\ \texttt{in}\ \mathrm{e}_2]\!] & = & \texttt{let}\ \mathrm{x}^{shp} = \mathcal{S}[\![\mathrm{e}_1]\!]\ \texttt{in}\ \mathcal{S}[\![\mathrm{e}_2]\!] \\
\text{(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} \\
 & & & \\
\text{(S-ExpNum)} & \mathrm{e}\colon \mathrm{Num} \vdash \mathcal{S}[\![\mathrm{e}]\!] & = & \circ \\
\text{(S-ExpBool)} & \mathrm{e}\colon \texttt{Bool} \vdash \mathcal{S}[\![\mathrm{e}]\!] & = & \circ \\
\text{(S-ValCard)} & \mathcal{S}[\![\mathrm{N}]\!] & = & \mathrm{N} \\
\text{(S-AddCard)} & \mathcal{S}[\![\mathrm{e}_0 +^c \mathrm{e}_1]\!] & = & \mathcal{S}[\![\mathrm{e}_0]\!] +^c \mathcal{S}[\![\mathrm{e}_1]\!] \\
\text{(S-MulCard)} & \mathcal{S}[\![\mathrm{e}_0 *^c \mathrm{e}_1]\!] & = & \mathcal{S}[\![\mathrm{e}_0]\!] *^c \mathcal{S}[\![\mathrm{e}_1]\!] \\
\text{(S-Build)} & \mathcal{S}[\![\texttt{build}\ \mathrm{e}_0\ \mathrm{e}_1]\!] & = & (\mathcal{S}[\![\mathrm{e}_0]\!], (\mathcal{S}[\![\mathrm{e}_1]\!]\ \circ)) \\
\text{(S-Get)} & \mathcal{S}[\![\mathrm{e}_0[\mathrm{e}_1]]\!] & = & \texttt{snd}\ \mathcal{S}[\![\mathrm{e}_0]\!] \\
\text{(S-Length)} & \mathcal{S}[\![\texttt{length}\,\mathrm{e}_0]\!] & = & \texttt{fst}\ \mathcal{S}[\![\mathrm{e}_0]\!] \\
\text{(S-Reduce)} & \mathcal{S}[\![\ \texttt{reduce}\ \mathrm{e}_1\ \mathrm{e}_2\ \mathrm{e}_3\ ]\!] & = & \begin{cases} \mathcal{S}[\![\mathrm{e}_2]\!] & \forall n.\mathcal{S}[\![\mathrm{e}_1\ \mathrm{e}_2\ \mathrm{n}]\!]\cong\mathcal{S}[\![\mathrm{e}_2]\!] \\ \text{Compilation Error!} & \text{otherwise} \end{cases} \\
 & & & \\
 & \mathcal{S}_{\mathcal{T}}[\![\mathrm{T}]\!] & = & \mathrm{S} \\
 & & & \\
\text{(ST-Fun)} & \mathcal{S}_{\mathcal{T}}[\![\mathrm{T}_1,\ \mathrm{T}_2,\ ...,\ \mathrm{T}_k \Rightarrow \mathrm{M}\ ]\!] & = & \mathcal{S}_{\mathcal{T}}[\![\mathrm{T}_1]\!],\ \mathcal{S}_{\mathcal{T}}[\![\mathrm{T}_2]\!],\ ...,\ \mathcal{S}_{\mathcal{T}}[\![\mathrm{T}_k]\!] \Rightarrow \mathcal{S}_{\mathcal{T}}[\![\mathrm{M}]\!] \\
\text{(ST-Num)} & \mathcal{S}_{\mathcal{T}}[\![\mathrm{Num}]\!] & = & \texttt{Card} \\
\text{(ST-Bool)} & \mathcal{S}_{\mathcal{T}}[\![\texttt{Bool}]\!] & = & \texttt{Card} \\
\text{(ST-Card)} & \mathcal{S}_{\mathcal{T}}[\![\texttt{Card}]\!] & = & \texttt{Card} \\
\text{(ST-Vector)} & \mathcal{S}_{\mathcal{T}}[\![\texttt{Array<M>}]\!] & = & (\texttt{Card}, \mathcal{S}_{\mathcal{T}}[\![\mathrm{M}]\!])
\end{array}
$$

Fig. 7. Shape Translation of $\widetilde{\mathsf{F}}$

The other rules are straightforward. *The key point is this: 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:

$$
\begin{array}{llll}
\texttt{alloc}\ \circ\ (\lambda \mathrm{r}.\ \mathrm{t}_1) & \leadsto & \mathrm{t}_1[\mathrm{r} \mapsto \bullet] & \text{Empty Allocation} \\
\texttt{alloc}\ \mathrm{t}_1\ (\lambda \mathrm{r}_1. & \leadsto & \texttt{alloc}\ (\mathrm{t}_1 +^c \mathrm{t}_2)\ (\lambda \mathrm{r}_1. & \\
\quad \texttt{alloc}\ \mathrm{t}_2\ (\lambda \mathrm{r}_2. & & \quad \texttt{let}\ \mathrm{r}_2 = \texttt{stgOff}\ \mathrm{r}_1\ \mathrm{t}_1\ \texttt{in} & \text{Allocation Merging} \\
\qquad \mathrm{t}_3\ )) & & \quad \mathrm{t}_3\ ) & \\
\texttt{alloc}\ \mathrm{t}_1\ (\lambda \mathrm{r}.\ \mathrm{t}_2) & \leadsto & \mathrm{t}_2 \qquad\qquad\qquad \text{if } \mathrm{r} \notin FV(\mathrm{t}_2) & \text{Dead Allocation} \\
\lambda x.\ \texttt{alloc}\ \mathrm{t}_1\ (\lambda \mathrm{r}.\ \mathrm{t}_2) & \leadsto & \texttt{alloc}\ \mathrm{t}_1\ (\lambda \mathrm{r}.\ \lambda x.\ \mathrm{t}_2)\ \text{if } x \notin FV(\mathrm{t}_1) & \text{Allocation Hoisting} \\
\texttt{bytes}\ \circ & \leadsto & \circ & \\
\texttt{bytes}\ (\circ, \circ) & \leadsto & \circ & \\
\texttt{bytes}\ (\mathrm{N}, \circ) & \leadsto & \texttt{NUM\_BYTES} *^c \mathrm{N} +^c \texttt{HDR\_BYTES} & \\
\texttt{bytes}\ (\mathrm{N}, \mathrm{s}) & \leadsto & (\texttt{bytes}\ \mathrm{s}) *^c \mathrm{N} +^c \texttt{HDR\_BYTES} &
\end{array}
$$

Fig. 8. Simplification rules of DPS-$\widetilde{\mathrm{F}}$

$$
\begin{array}{l}
\mathrm{f} = \lambda \mathrm{r}_0\ \mathrm{vec1}\ \mathrm{vec2}\ \mathrm{vec1}^{shp}\ \mathrm{vec2}^{shp}. \\
\quad \texttt{let}\ \mathrm{tmp}^{shp} = \mathrm{vectorAdd}^{shp}\ \mathrm{vec1}^{shp}\ \mathrm{vec2}^{shp}\ \texttt{in} \\
\quad \texttt{alloc}\ (\texttt{bytes}\ \mathrm{tmp}^{shp})\ (\lambda \mathrm{r}_1. \\
\qquad \texttt{let}\ \mathrm{tmp} = \\
\qquad\quad \mathrm{vectorAdd}\ \mathrm{r}_1\ \mathrm{vec1}\ \mathrm{vec2} \\
\qquad\qquad\quad \mathrm{vec1}^{shp}\ \mathrm{vec2}^{shp}\ \texttt{in} \\
\qquad \mathrm{vectorNorm}\ \mathrm{r}_0\ \mathrm{tmp}\ \mathrm{tmp}^{shp} \\
\quad)
\end{array}
$$

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

$$
\begin{array}{l}
\texttt{let}\ \mathrm{vectorRange}^{shp} = \lambda \mathrm{n}^{shp}.\ \ (\mathrm{n}^{shp}, (\lambda \mathrm{i}^{shp}.\ \circ)\ \ \circ) \\
\texttt{let}\ \mathrm{vectorMap2}^{shp} = \lambda \mathrm{v1}^{shp}\ \mathrm{v2}^{shp}\ \mathrm{f}^{shp}. \\
\quad (\texttt{fst}\ \mathrm{v1}^{shp}, (\lambda \mathrm{i}^{shp}.\ \circ)\ \ \circ) \\
\texttt{let}\ \mathrm{vectorAdd}^{shp} = \lambda \mathrm{v1}^{shp}\ \mathrm{v2}^{shp}. \\
\quad \mathrm{vectorMap2}^{shp}\ \mathrm{v1}^{shp}\ \mathrm{v2}^{shp}\ (\lambda \mathrm{a}^{shp}\ \mathrm{b}^{shp}.\ \circ) \\
\texttt{let}\ \mathrm{vectorNorm}^{shp} = \lambda \mathrm{v}^{shp}.\ \circ
\end{array}
$$

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$^{shp}$ = $\lambda$n$^{shp}$. (n$^{shp}$, $\circ$)
let vectorMap2$^{shp}$ = $\lambda$v1$^{shp}$ v2$^{shp}$ f$^{shp}$.  v1$^{shp}$
let vectorAdd$^{shp}$ = $\lambda$v1$^{shp}$ v2$^{shp}$.  v1$^{shp}$
let vectorNorm$^{shp}$ = $\lambda$v$^{shp}$. $\circ$
```

The final stage involves both partially evaluating the shape expressions in DPS-$\widetilde{\text{F}}$ and simplifying the storage accesses in the DPS-$\widetilde{\text{F}}$ expressions. Figure 8 demonstrates simplification rules for storage accesses. The first two rules remove empty allocations and 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 `reduce`). Note that none of these transformation rules are available in $\widetilde{\text{F}}$, due to the lack of explicit storage facilities.

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

```
f = $\lambda$r$_0$ vec1 vec2 vec1$^{shp}$ vec2$^{shp}$.
  alloc (bytes vec1$^{shp}$) ($\lambda$r$_1$.
    let tmp =
      vectorAdd r$_1$ vec1 vec2
          vec1$^{shp}$ vec2$^{shp}$ in
    vectorNorm r$_0$ tmp vec1$^{shp}$
  )
```

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

3.6 Properties of shape translation

The target language of shape translation is a subset of DPS-$\widetilde{\text{F}}$ called Shape-$\widetilde{\text{F}}$. The syntax of the subset is given in Figure 9. It includes nested pairs, of statically-known depth, to represent shapes, but it does not include vectors. That provides an important property for Shape-$\widetilde{\text{F}}$ 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 it is shown in Figure 7 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 we avoid partial application. 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 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 $\widetilde{\text{F}}$ 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:

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

Fig. 9. Shape-$\widetilde{\mathrm{F}}$ syntax, which is a subset of the syntax of DPS-$\widetilde{\mathrm{F}}$ presented in Figure 4.

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 which is used in the `reduce` 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 7.

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

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 [31]. 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 inter-operatorability 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-$\widetilde{\text{F}}$ is that we can generate idiomatic C from it. More specifically, the generated C code is similar to a handwritten C program. This is because, we can manage the memory in a stack fashion. The translation from DPS-$\widetilde{\text{F}}$ 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. Because DPS-$\widetilde{\text{F}}$ does not allow partial application, the environment captured by a closure can be stack allocated.

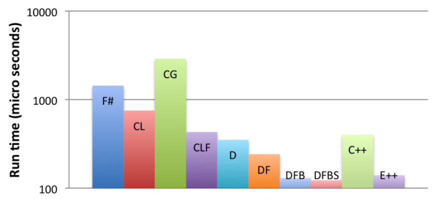
As mentioned in Section 2, polymorphism is not allowed except for some built-in constructs in the language (e.g. `build` and `reduce`). Hence, all the usages of these constructs are monomorphic, and the C code generator knows exactly which code to generate for them. Furthermore, the C code generator 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 $\widetilde{\text{F}}$ is:

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

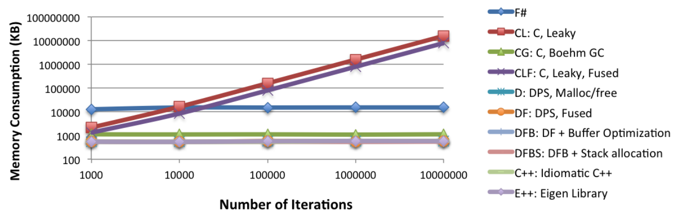
Finally, for the `alloc` construct in DPS-$\widetilde{\text{F}}$, the generated C code consists of the following three parts. First, a memory allocation statement is generated which allocates the given amount of storage. Second, the corresponding body of code which uses the allocated storage is generated.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

```
double f(storage r0, vector vec1_dps, vector vec2_dps,
            vec_shape vec1_shp, vec_shape vec2_shp) {
  storage r1 = malloc(vector_bytes(vec1_shp));
  vector tmp_dps =
    vector_add_dps(r1, vec1_dps, vec2_dps, vec1_shp, vec2_shp);
  double result = vector_norm_dps(r0, tmp_dps, vec1_shp);
  free(r1);
  return result;
}
```

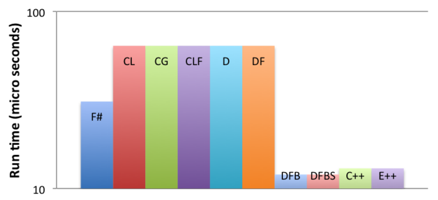
We use our own implementation of `malloc` and `free` for bump allocation.



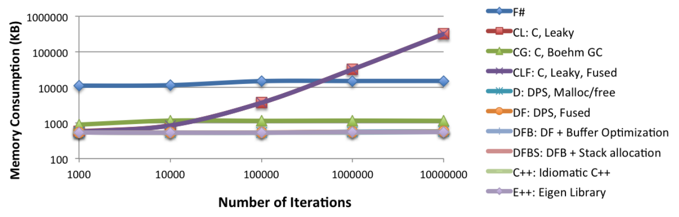
(a) Runtime performance comparison of different approaches on adding three vectors of 100 elements for one million times.



(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.

Fig. 10. Experimental Results for Micro Benchmarks

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.

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 $\widetilde{\text{F}}$, using malloc to allocate vectors, never calling free.
- **CG: C code using Boehm GC**, which is the generated C code from $\widetilde{\text{F}}$, 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 $\widetilde{\text{F}}$ programs into DPS-$\widetilde{\text{F}}$ 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-$\widetilde{\text{F}}$.
- **DFB: DF + Buffer Optimizations**, which performs the buffer optimizations described in Section 3.5 (such as allocation hoisting and merging) on DPS-$\widetilde{\text{F}}$ 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 $\widetilde{\mathrm{F}}$.
- **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 performance and memory consumption 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.

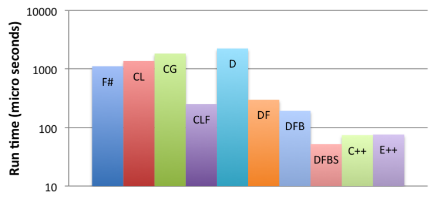
Then, we demonstrate the benefit of using DPS for some real-life computer vision and machine learning workloads motivated in [28]. Based on the results for these workloads, we argue that using DPS is a great choice for generating C code for numerical workloads, such as computer vision algorithms, running on embedded devices with a limited amount of memory available.

5.1 Micro Benchmarks

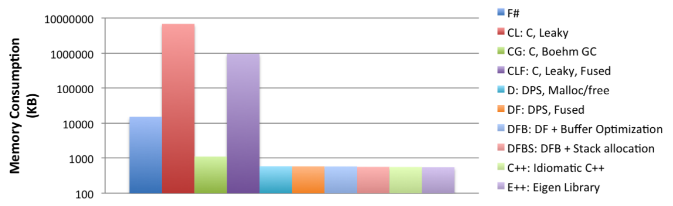
Figure 10 shows the experimental results for micro benchmarks, one adding three vectors, the second using vector cross product.

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 10a. 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.

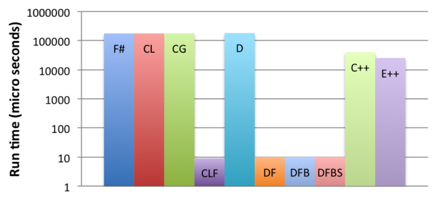
The peak memory consumption of this program for different approaches is shown in Figure 10b. This measurement is performed by running this program by varying number of iterations. Both axes use logarithmic scales to better demonstrate the memory consumption difference. As expected, F# uses almost the same amount of memory over the time, due to garbage collection. 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.



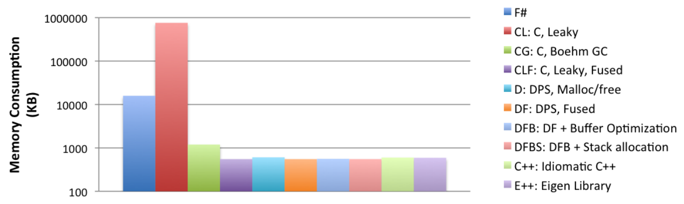
(a) Runtimes: Bundle Adjustment



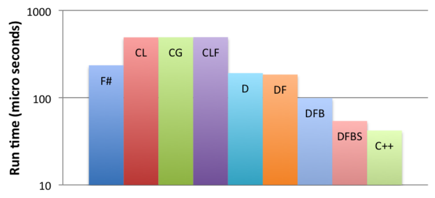
(b) Memory consumption: Bundle Adjustment



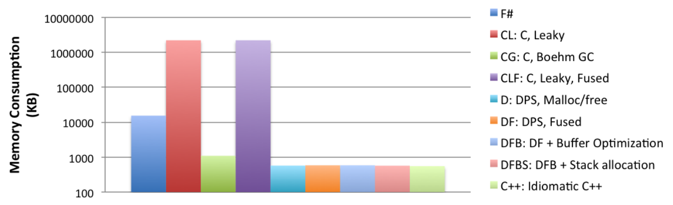
(c) Runtimes: GMM



(d) Memory consumption: GMM



(e) Runtimes: Hand Tracking



(f) Memory consumption: Hand Tracking

Fig. 11. Experimental Results for Computer Vision and Machine Learning Workloads

cross. : vectorCross(vec1, vec2)
This micro-benchmark is 1 million runs in which the two vectors contain 3 elements. Timing results are in Figure 10c. 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 10d. 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.

Next, we investigate the performance and memory consumption of real-life workloads.

5.2 Computer Vision and Machine Learning Workloads

Bundle Adjustment. [35] 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 of a 3D point by a camera, including a model of the radial distortion of the lens. The $\widetilde{\text{F}}$ implementation of this method is given in Figure 12.

Figure 11a 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 11b. 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 11c and Figure 11d. Both C++ libraries do not support the loop fusion needed for GMM. Hence, they behave three orders of magnitude worse than our fused and DPS generated C code.

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, it monotonically increases the consumed memory.

Hand tracking. is a computer vision/computer graphics workload [32] that includes matrix-matrix multiplies, and numerous combinations of fixed- and variable-sized vectors and matrices. Figure 11e 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. Similar to previous cases generating DPS C code improves runtime performance, which is improved even more by using bump allocation and performing loop fusion and

```
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 rotation = vectorSlice cam 0 2
  let center = vectorSlice cam 3 5
  let radical = vectorSlice cam 9 10
  let Xcam =
    rodriguesRotate rotation (vectorSub x center)
  let distorted =
    radialDistort radical (
      vectorSMul (
        vectorSlice Xcam 0 1
      ) (1.0/Xcam.[2]))
  vectorAdd (vectorSlice cam 7 8) (
    vectorSMul distorted cam.[6]
  )
```

Fig. 12. Bundle Adjustment functions in $\widetilde{\text{F}}$.

buffer optimizations. However, in this case the idiomatic C++ version outperforms the generated DPS C code. Figure 11f 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 using garbage collection dates back to Linear Lisp [1]. However, most functional languages (dating back to the Lisp around 1959) use garbage collection for managing memory.

Region-based memory management was first introduced in ML [33] 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 [2, 34]. Furthermore, the complexity of region inference, hinders the maintenance of the compiler, in addition to the overhead it causes for compilation time.

Safe [23, 24] 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. This is because of certain usecases, which use unbounded amount of memory due to recursion. A Haskell implementation of this approach is given in [20]. The restricted form of recursion allowed by $\widetilde{\mathrm{F}}$ means that we never face similar issues. Hence, we choose to always follow the stack discipline for memory management.

6.2 Estimation of Memory Consumption

One can use type systems for estimating memory consumption. Hofmann and Jost [17] enrich the type system with certain annotations and uses linear programming for the heap consumption inference. Another approach is to use sized types [37] for the same purpose.

Size slicing [16] 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 [17]. The FISh programming language [19] also makes shape information explicit in programs, and resolves the shapes at compilation time by using partial evaluation.

Clinger [4] explores different space efficiency classes. Based on the proposed formalism he defines formally what it means for a language to *properly* handle tail recursion. Next, we see related work on optimizing tail recursive calls.

6.3 Optimizing Tail Calls

Destination-passing style was originally introduced in [21], then was encoded functionally in [22] by using linear types [36, 40]. Walker and Morrisett [41] 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, 38].

6.4 Array Programming Languages

APL [18] can be considered as the first array programming language. Futhark [15, 16] 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. However, as this topic is out of the scope of this paper, we leave more discussion for the future work.

There are many domain-specific languages (DSLs) for numerical workloads such as Opt [6], Halide [26], Diderot [3], and OptiML [29]. All these DSLs generate parallel code from their high-level programs. Furthermore, Halide [26] exploits the memory hierarchy by making tiling and scheduling decisions, similar to Spiral [25] and LGen [27]. Although both parallelism and improving use of a memory hierarchy are orthogonal concepts to translation into DPS, they are still interesting directions for $\widetilde{\mathrm{F}}$.

REFERENCES

[1] H. G. Baker. Lively linear lisp: 'look ma, no garbage!'. *ACM Sigplan notices*, 27(8):89–98, 1992.
[2] L. Birkedal, M. Tofte, and M. Vejlstrup. From region inference to Von Neumann machines via region representation inference. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '96, pages 171–183, New York, NY, USA, 1996. ACM. ISBN 0-89791-769-3.
[3] C. Chiw, G. Kindlmann, J. Reppy, L. Samuels, and N. Seltzer. Diderot: A parallel DSL for Image Analysis and Visualization. In *Acm sigplan notices*, volume 47, pages 111–120. ACM, 2012.
[4] W. D. Clinger. Proper tail recursion and space efficiency. pages 174–185. ACM Press, 1998.
[5] D. Coutts, R. Leshchinskiy, and D. Stewart. Stream Fusion. From Lists to Streams to Nothing at All. In *ICFP '07*, 2007.
[6] Z. DeVito, M. Mara, M. Zollhöfer, G. Bernstein, J. Ragan-Kelley, C. Theobalt, P. Hanrahan, M. Fisher, and M. Nießner. Opt: A Domain Specific Language for Non-linear Least Squares Optimization in Graphics and Imaging. *arXiv preprint arXiv:1604.06525*, 2016.
[7] C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen. The essence of compiling with continuations. In *ACM Sigplan Notices*, volume 28, pages 237–247. ACM, 1993.
[8] M. Fluet, G. Morrisett, and A. Ahmed. Linear regions are all you need. In *European Symposium on Programming*, pages 7–21. Springer, 2006.
[9] D. Friedman and S. Wise. Unwinding stylized recursions into iterations. *Comput. Sci. Dep., Indiana University, Bloomington, IN, Tech. Rep*, 19, 1975.
[10] A. Gill, J. Launchbury, and S. L. Peyton Jones. A short cut to deforestation. FPCA, pages 223–232. ACM, 1993.
[11] C. Grelck and S.-B. Scholz. SAC—A functional array language for efficient multi-threaded execution. *International Journal of Parallel Programming*, 34(4):383–427, 2006. ISSN 1573-7640.
[12] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney. Region-based memory management in cyclone. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI '02, pages 282–293, New York, NY, USA, 2002. ACM. ISBN 1-58113-463-0.
[13] G. Guennebaud, B. Jacob, et al. Eigen. *URl: http://eigen. tuxfamily. org*, 2010.
[14] N. Hallenberg, M. Elsman, and M. Tofte. Combining Region Inference and Garbage Collection. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI '02, pages 141–152, New York, NY, USA, 2002. ACM. ISBN 1-58113-463-0. doi: 10.1145/512529.512547. URL http://doi.acm.org/10.1145/512529.512547.
[15] T. Henriksen and C. E. Oancea. Bounds checking: An instance of hybrid analysis. In *Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*, ARRAY'14, pages 88:88–88:94, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2937-8.
[16] T. Henriksen, M. Elsman, and C. E. Oancea. Size Slicing: A hybrid approach to size inference in Futhark. In *Proceedings of the 3rd ACM SIGPLAN Workshop on Functional High-performance Computing*, FHPC '14, pages 31–42, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-3040-4.
[17] M. Hofmann and S. Jost. Static prediction of heap space usage for first-order functional programs. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '03, pages 185–197, New York, NY, USA, 2003. ACM. ISBN 1-58113-628-5.
[18] K. E. Iverson. A Programming Language. In *Proceedings of the May 1-3, 1962, spring joint computer conference*, pages 345–351. ACM, 1962.
[19] C. B. Jay. Programming in FISh. *International Journal on Software Tools for Technology Transfer*, 2(3):307–315, 1999.
[20] O. Kiselyov and C.-c. Shan. Lightweight monadic regions. In *ACM Sigplan Notices*, volume 44, pages 1–12. ACM, 2008.
[21] J. R. Larus. *Restructuring symbolic programs for concurrent execution on multiprocessors.* PhD thesis, 1989.
[22] Y. Minamide. A functional representation of data structures with a hole. In *In Conference Record of the 25th Symposium on Principles of Programming Languages (POPL '98*, pages 75–84, 1998.
[23] M. Montenegro, R. Peña, and C. Segura. A type system for safe memory management and its proof of correctness. In *Proceedings of the 10th international ACM SIGPLAN conference on Principles and practice of declarative programming*, pages 152–162. ACM, 2008.
[24] M. Montenegro, R. Peña, and C. Segura. A simple region inference algorithm for a first-order functional language. In *International Workshop on Functional and Constraint Logic Programming*, pages 145–161. Springer, 2009.
[25] M. Puschel, J. M. Moura, J. R. Johnson, D. Padua, M. M. Veloso, B. W. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, et al. SPIRAL: code generation for DSP transforms. *Proceedings of the IEEE*, 93(2):232–275, 2005.

[26] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *ACM SIGPLAN Notices*, 48(6): 519–530, 2013.

[27] D. G. Spampinato and M. Püschel. A basic linear algebra compiler for structured matrices. In *CGO '16*. ACM.

[28] F. Srajer, Z. Kukelova, A. Fitzgibbon, A. G. Schwing, M. Pollefeys, and T. Pajdla. A benchmark of selected algorithmic differentiation tools on some problems in machine learning and computer vision. *Automatic Differentiation*, 2016.

[29] A. Sujeeth, H. Lee, K. Brown, T. Rompf, H. Chafi, M. Wu, A. Atreya, M. Odersky, and K. Olukotun. OptiML: An Implicitly Parallel Domain-Specific Language for Machine Learning. In *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, pages 609–616, 2011.

[30] J. Svenningsson. Shortcut fusion for accumulating parameters & zip-like functions. ICFP '02, pages 124–132. ACM, 2002. ISBN 1-58113-487-8.

[31] D. Syme. Leveraging .NET Meta-programming Components from F#: Integrated Queries and Interoperable Heterogeneous Execution. In *Proceedings of the 2006 Workshop on ML*, ML '06, pages 43–54, New York, NY, USA, 2006. ACM. ISBN 1-59593-483-9. doi: 10.1145/1159876.1159884. URL http://doi.acm.org/10.1145/1159876.1159884.

[32] J. Taylor, R. Stebbing, V. Ramakrishna, C. Keskin, J. Shotton, S. Izadi, A. Hertzmann, and A. Fitzgibbon. User-specific hand modeling from monocular depth sequences. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 644–651, 2014.

[33] M. Tofte and J.-P. Talpin. Region-based memory management. *Information and Computation*, 132(2):109 – 176, 1997. ISSN 0890-5401.

[34] M. Tofte, L. Birkedal, M. Elsman, and N. Hallenberg. A retrospective on region-based memory management. *Higher Order Symbol. Comput.*, 17(3):245–265, Sept. 2004. ISSN 1388-3690.

[35] B. Triggs, P. F. McLauchlan, R. I. Hartley, and A. W. Fitzgibbon. Bundle adjustment—a modern synthesis. In *International workshop on vision algorithms*, pages 298–372. Springer, 1999.

[36] D. N. Turner, P. Wadler, and C. Mossin. Once upon a type. In *Proceedings of the seventh international conference on Functional programming languages and computer architecture*, pages 1–11. ACM, 1995.

[37] P. B. Vasconcelos. *Space cost analysis using sized types*. PhD thesis, University of St Andrews, 2008.

[38] P. Wadler. Listlessness is better than laziness: Lazy evaluation and garbage collection at compile-time. In *Proceedings of the 1984 ACM Symposium on LISP and functional programming*, pages 45–52. ACM, 1984.

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

[40] P. Wadler. Linear types can change the world. In *IFIP TC*, volume 2, pages 347–359. Citeseer, 1990.

[41] D. Walker and G. Morrisett. Alias types for recursive data structures. In *International Workshop on Types in Compilation*, pages 177–206. Springer, 2000.