Destination-Passing Style for Efficient Memory Management

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

Simon Peyton Jones
Microsoft Research, UK
simonpj@microsoft.com

Andrew Fitzgibbon
Microsoft HoloLens, UK
awf@microsoft.com

Dimitrios Vytiniotis
Microsoft Research, UK
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:

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 [20, 21] (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 (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, 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 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 [17] expose shape concerns intrusively at the language level, while F programs are just F#.

• We present the implementation 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 $\tilde{F}$
$\tilde{F}$ (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

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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

FHPC’17, September 7, 2017, Oxford, UK
© 2017 Association for Computing Machinery.
ACM ISBN 978-1-4503-5181-2/17/09...$15.00
https://doi.org/10.1145/3122948.3122949
e ::= e e – Application
| A 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
| T ⊃ M – Function Types (No Currying)
| Card – Cardinality Type
| Bool – Boolean Type
| Num – Numeric Type
| Array< M > – Vector, Matrix, ... Type

M ::= Num | Index – Scalar and Index Type

Scalar Function Constants:
+ | | * | / : Num, Num ⇒ Num
% : Index, Index ⇒ Index
| < | == : Num, Num ⇒ Bool
& & | || : Bool, Bool ⇒ Bool
! : Bool ⇒ Bool
+ | | | * | / | % : Card, Card ⇒ Card

Vector Function Constants:
build n f : Card, (Index ⇒ M) ⇒ Array<M>
ifold f m0 n : (M, Index ⇒ M), M, Card ⇒ M
get a i : Array<M>, Index ⇒ M
length a : Array<M> ⇒ Card

e0[e1] = get e0 e1
e1 bop e2 = bop e1 e2 – For binary operators bop

e ::= e e – Application
| A 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
| T ⊃ M – Function Types (No Currying)
| Card – Cardinality Type
| Bool – Boolean Type
| Num – Numeric Type
| Array< M > – Vector, Matrix, ... Type

M ::= Num | Index – Scalar and Index Type

Scalar Function Constants:
+ | | * | / : Num, Num ⇒ Num
% : Index, Index ⇒ Index
| < | == : Num, Num ⇒ Bool
& & | || : Bool, Bool ⇒ Bool
! : Bool ⇒ Bool
+ | | | * | / | % : Card, Card ⇒ Card

Vector Function Constants:
build n f : Card, (Index ⇒ M) ⇒ Array<M>
ifold f m0 n : (M, Index ⇒ M), M, Card ⇒ M
get a i : Array<M>, Index ⇒ M
length a : Array<M> ⇒ Card

e0[e1] = get e0 e1
e1 bop e2 = bop e1 e2 – For binary operators bop


Although \( \tilde{F} \) is a higher-order functional language, it is carefully restricted in order to make it efficiently compilable:
- \( \tilde{F} \) 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 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 operators. 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.
Thus motivated, we define a new intermediate language, 

\[
\text{let vectorRange = } \lambda n. \ build n (\lambda i. i)
\]

\[
\text{let vectorMap = } \lambda v. f v
\]

\[
\text{build (length v) (\lambda i. f v[i])}
\]

\[
\text{let vectorMap2 = } \lambda v1 v2 f
\]

\[
\text{build (length v1) (\lambda i. f v1[i] v2[i])}
\]

\[
\text{let vectorEMul = } \lambda v1 v2. \ vectorMap2 v1 v2 (+)
\]

\[
\text{let vectorSMul = } \lambda v s. \ vectorMap v (\lambda a. a \times s)
\]

\[
\text{let vectorSum = } \lambda v.
\]

\[
\text{ifold (\lambda sum idx. sum + v[idx]) 0 (length v)}
\]

\[
\text{let vectorNorm = } \lambda v. \ \sqrt{\text{vectorDot } v v}
\]

\[
\text{let vectorSlice = } \lambda v s e.
\]

\[
\text{build (e - c s + c 1) (\lambda i. v[i + s])}
\]

\[
\text{let matrixRows = } \lambda m. \ \text{length m}
\]

\[
\text{let matrixCols = } \lambda m. \ \text{length m[0]}
\]

\[
\text{let matrixMap = } \lambda m. \ build (\text{length m}) (\lambda i. f m[i])
\]

\[
\text{let matrixMap2 = } \lambda m1 m2 f
\]

\[
\text{build (\text{length m1}) (\lambda i. f m1[i] m2[i])}
\]

\[
\text{let matrixAdd = } \lambda m1 m2. \ matrixMap2 m1 m2 \text{ vectorAdd}
\]

\[
\text{let matrixTranspose = } \lambda m.
\]

\[
\text{build (matrixCols m) (\lambda i. m[i][i])}
\]

\[
\text{let matrixMul = } \lambda m1 m2.
\]

\[
\text{let m2T = matrixTranspose m2}
\]

\[
\text{build (matrixRows m1) (\lambda i. m1[i][j])}
\]

\[
\text{let matrixOutProd = } \lambda v1 v2.
\]

\[
\text{let m1 = build 1 (\lambda i. v1)}
\]

\[
\text{let m2 = build 1 (\lambda i. v2)}
\]

\[
\text{let m2T = matrixTranspose m2}
\]

\[
\text{matrixMul m1 m2T}
\]

Adding the two vectors \(\text{vec1}\) and \(\text{vec2}\), 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). After using the rules presented in Figure 4, the fused function is as follows:

\[
f = \lambda \text{vec1 vec2}.
\]

\[
\text{ifold (\lambda sum idx. sum + \text{vec2[idx]} \text{in}}
\]

\[
\text{sum + tmp * tmp}
\]

\[
0 (\text{length vec1})
\]

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-F, in which memory allocation and deallocation is explicit. DPS-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. \text{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 [14, 15].

3.1 The DPS-F Language

The syntax of DPS-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...
Typing Rules:

\[
(T-\text{Alloc}) \quad \Gamma \vdash t_0 : \text{Card} \quad \Gamma, r : \text{Ref} \vdash t_1 : M \\
\frac{}{\text{alloc} t_0 (\lambda r. t_1) : M}
\]

Vector Function Constants:

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

Scalar Function Constants:

DPS version of \( F \) 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:

- \( t_0, [t_1] \{r\} = \text{get} \ r \ t_0 \ t_1 \ length \ t = \text{length} \ t \cdot t \)
- \( t_0, t_1 = \text{vecShp} \ t_0 \ t_1 \)

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-\( F \).

of storage space alloc \( t_1 \) (\( \lambda r. t_2 \)). In this construct \( t_1 \) is an expression that evaluates to the size (in bytes) that is required for storing the result of evaluating \( t_2 \). 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-\( F \):

\[
f = \lambda r_1 \ \text{vec1} \ \text{vec2}. \ \text{alloc} (\text{vecBytes} \ \text{vec1}) (\lambda r_2. \ \text{vectorNorm_dps} \bullet (\text{vectorAdd_dps} \ r_2 \ \text{vec1} \ \text{vec2}) )
\]

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 \( \bullet \). 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 \( F \) to DPS-\( F \)

We now turn present the translation from \( F \) to DPS-\( F \). Before translating \( F \) 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:

\[
f = \lambda \text{vec1} \ \text{vec2}.
\]

\[
\text{let} \ \text{tmp} = \text{vectorAdd} \ \text{vec1} \ \text{vec2} \ \text{in}
\]

\[
\text{vectorNorm_dps} \ \text{tmp}
\]

Figure 7 shows the translation from \( F \) to DPS-\( F \), where \( \mathcal{D}[e] \) is the translation of a \( F \) expression \( e \) into a DPS-\( F \) expression that stores \( e \)'s value in memory \( r \). Rule (D-Let) is a good place to start. It uses \( \text{alloc} \) 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 \( \mathcal{S}[e] \) 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 \( \text{alloc} \). 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 \( F \) becomes a pair of variables \( x \) (for \( x \)'s value) and \( x^{shp} \) (for \( x \)'s shape) in DPS-\( F \). 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 \( F \) type \( T \) to the corresponding DPS-\( F \) 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 \( \text{copy} \) function. The \( \text{copy} \) 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 \( \mathcal{S}[T] \) that gives the shape of \( e \). This expression can always be evaluated without allocation.

A shape is an expression of type \( \text{Shp} \) (Figure 5), whose values are given by \( \mathcal{P} \) 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

\[1\] 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.
3.4 An Example

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

\[ f = \lambda r_0 \text{vec1 vec2 vec1}^{shp} \text{vec2}^{shp}. \]

\[ \text{let tmp}^{shp} = \text{vectorAdd}^{shp} \text{vec1}^{shp} \text{vec2}^{shp} \text{ in} \]

\[ \text{alloc} (\text{bytes} \text{tmp}^{shp}) (\lambda r_1. \]

\[ \text{let tmp} = \]

\[ \text{vectorAdd} r_1 \text{vec1 vec2 vec1}^{shp} \text{vec2}^{shp} \text{ in} \]

\[ \text{vectorNorm} r_0 \text{tmp} \text{tmp}^{shp} \]

\[ ) \]

The shape translations of some \( \tilde{F} \) functions from Figure 3 are as follows:

\[ \text{let vectorRange}^{shp} = \lambda n^{shp}. \text{n}^{shp}, (\lambda i^{shp}. \text{o}) \text{ o} \]

\[ \text{let vectorMap2}^{shp} = \lambda v1^{shp} v2^{shp} p^{shp}. \]

\[ \text{fst} v1^{shp}, (\lambda i^{shp}. \text{o}) \text{ o} \]

\[ \text{let vectorAdd}^{shp} = \lambda v1^{shp} v2^{shp}. \]

\[ \text{vectorMap2}^{shp} v1^{shp} v2^{shp} (\lambda a^{shp} b^{shp}. \text{o}) \]

\[ \text{let vectorNorm}^{shp} = \lambda a^{shp}. \text{o} \]

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) \( \tilde{F} \) expressions, 2) translated Shape-\( \tilde{F} \) expressions, and 3) translated DPS-\( \tilde{F} \) expressions. In the first stage, \( \tilde{F} \) 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-\( \tilde{F} \) expressions are simplified. The simplification process for these expressions mainly involves partial evaluation. By inlining all shape functions, and performing \( \beta \)-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:
$S[e] = s$

(S-App) $S[e_0 	ext{ } e_1 ... e_k] = S[e_0] \cdot S[e_1] ... \cdot S[e_k]$

(S-Abs) $S[\lambda x_1:T_1, ..., x_k:T_k. e] = \lambda x_1^{shp}. S_T[T_1], ..., x_k^{shp}. S_T[T_k], S[e]$

(S-Var) $S[x] = x^{shp}$

(S-Let) $S[\text{let } x = e_1 \text{ in } e_2] = \text{let } x^{shp} = S[e_1] \text{ in } S[e_2]$

(S-If) $S[\text{if } e_1 \text{ then } e_2 \text{ else } e_3] = \begin{cases} S[e_2] & S[e_2] \neq S[e_3] \\ \text{Compilation Error!} & \text{otherwise} \end{cases}$

(S-ExpNum) $c: \text{Num} + S[e] = 0$

(S-ExpBool) $c: \text{Bool} + S[e] = 0$

(S-ValCard) $S[N] = N$

(S-AddCard) $S[e_0 +^c e_1] = S[e_0] + S[e_1]$

(S-MulCard) $S[e_0 *^c e_1] = S[e_0] * S[e_1]$

(S-Build) $S[\text{build } e_0 e_1] = S[e_0]. S[e_1]$

(S-Get) $S[e_0[e_1]] = \text{snd } S[e_0]$

(S-Length) $S[\text{length } e_0] = \text{fst } S[e_0]$

(S-Ifold) $S[\text{ifold } e_1 \text{ } e_2 \text{ } e_3] = \begin{cases} S[e_2] & \forall n.S[e_1] e_2 n \neq S[e_2] \\ \text{Compilation Error!} & \text{otherwise} \end{cases}$

$S_T[T] = S$

(St-Fun) $S_T[T_1, T_2, ..., T_k ] = S_T[T_1], S_T[T_2], ..., S_T[T_k] \Rightarrow S_T[M]$

(St-Num) $S_T[\text{Num}] = \text{Card}$

(St-Bool) $S_T[\text{Bool}] = \text{Card}$

(St-Card) $S_T[\text{Card}] = \text{Card}$

(St-Vector) $S_T[\text{Array<}M \text{>}] = (\text{Card } S_T[M])$

The final stage involves both partially evaluating the shape expressions in DPS-$\tilde{F}$ and simplifying the storage accesses in the DPS-$\tilde{F}$ expressions. Figure 9 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 ifold). Note that none of these transformation rules are available in $\tilde{F}$, due to the lack of explicit storage facilities.

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

```latex
let vectorRange^{shp} = \lambda n^{shp}. n^{shp}, o
let vectorMap^{shp} = \lambda v^{shp}. v^{shp}, p^{shp}, v^{shp}
let vectorAdd^{shp} = \lambda v^{shp}. v^{shp}, v^{shp}
let vectorNorm^{shp} = \lambda v^{shp}. o

let vectorRange^{shp} = \lambda n^{shp}. n^{shp}, o
let vectorMap^{shp} = \lambda v^{shp}. v^{shp}, p^{shp}, v^{shp}
let vectorAdd^{shp} = \lambda v^{shp}. v^{shp}, v^{shp}
let vectorNorm^{shp} = \lambda v^{shp}. o

\begin{align*}
& f = \lambda r_0 \text{ vec1 vec2 vec1^{shp} vec2^{shp}}, \\
& \text{alloc bytes vec1^{shp} } (\lambda r_1, \\
& \text{let tmp = vectorAdd r_1 vec1 vec2 vec1^{shp} vec2^{shp} in} \\
& \text{vectorNorm r_0 tmp vec1^{shp}}
\end{align*}
```

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-$\tilde{F}$ called Shape-$\tilde{F}$. 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-$\tilde{F}$ as follows:

**Theorem 1.** All expressions resulting from shape translation, do not require any heap memory allocation.
terms in the original as well as better guarantees about the expressions resulting from Theorem 3. All expressions resulting from shape translation Theorem 2. Proof. Can be proved by induction on the translation rules from F to Shape-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 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 F program.

Proof. This can be proved in two steps. First, translating a 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 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 (c.f. 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 [16], 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 F Language

We implemented F as a subset of F#. Hence 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 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 F program. The F# quotations are only used by the compiler developer in order to implement transformation passes.

Although F expressions are F# expressions, it is not possible to express memory management constructs used by DPS-F expressions using the F# runtime. Hence, after translating F expressions to DPS-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 over 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-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-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-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 F is as follows:

```c
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-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:

```c
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.

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 F, using malloc to allocate vectors, never calling free.
- **CG**: C code using Boehm GC, which is the generated C code from 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 F programs into DPS-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-F.
- **DFB**: DF + Buffer Optimizations, which performs the buffer optimizations described in Section 3.5 (such as allocation hoisting and merging) on DPS-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 F.
- **C++**: Idiomatic C++, which uses an handwritten C++ vector library, depending on C++14 move construction and copy elision for performance, with explicit programmer indication of fixed-size (known at compile time) vectors, permitting stack allocation.
- **E++**: Eigen C++, which uses the Eigen [12] 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 [27]. 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.

\(^2\) All code and outputs are available at http://github.com/awf/Coconut.
Figure 11. Experimental Results for Micro Benchmarks

5.1 Micro Benchmarks

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

\( \text{add3} \) : \text{vectorAdd}(\text{vectorAdd}(\text{vec1}, \text{vec2}), \text{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 sizes 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 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.

\( \text{cross} \) : \text{vectorCross}(\text{vec1}, \text{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** [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 $\tilde{F}$ implementation of this method is partially 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, 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 $\tilde{F}$ implementation of this method is partially 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, 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 $\tilde{F}$ implementation of this method is partially 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, 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 $\tilde{F}$ implementation of this method is partially 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, 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 $\tilde{F}$ implementation of this method is partially 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, 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 $\tilde{F}$ implementation of this method is partially 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, 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 $\tilde{F}$ implementation of this method is partially 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, 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 $\tilde{F}$ implementation of this method is partially 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, 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 $\tilde{F}$ implementation of this method is partially 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, 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 $\tilde{F}$ implementation of this method is partially 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, 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 $\tilde{F}$ implementation of this method is partially 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, 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 $\tilde{F}$ implementation of this method is partially 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, 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 $\tilde{F}$ implementation of this method is partially 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, 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 $\tilde{F}$ implementation of this method is partially 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, 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.
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 [32] 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, 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 was first introduced in ML [34] and then in an extended version of C, called Cyclone [11], 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 [13], as well as cases in which the performance is still not satisfying [3, 33]. Furthermore, the complexity of region inference hinders the maintenance of the compiler, in addition to the overhead it causes for compilation time.

Safe [22, 23] 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 use-cases which use recursion and need an unbounded amount of memory. A Haskell implementation of this approach is given in [19]. 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, F offers a restricted form of recursion, which always enforces a stack discipline for memory management.

6.2 Array Languages and Push-Arrays

There is a close connection between so-called push arrays [1, 5, 30] 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 (c.f. 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 [25], Diderot [4], and OptiML [28]. All these DSLs generate parallel code from their high-level programs. Furthermore, Halide [25] exploits the memory hierarchy by making tiling and scheduling decisions, similar to Spiral [24] and LGen [26]. Although both parallelism and improving the usage of a memory hierarchy are
orthogonal concepts to translation into DPS, they are still interesting directions for F.

6.3 Estimation of Memory Consumption

One can use type systems for estimating memory consumption. Hofmann and Jost [16] enrich the type system with certain annotations and uses linear programming for the heap consumption inference. Another approach is to use sized types [36] for the same purpose.

Size slicing [14] uses a technique similar to ours for inferring the shape of arrays in the Futhark programming language. However, in 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 [16]. The FIsh programming language [17] also makes shape information explicit in programs, and resolves the shapes at compilation time by using partial evaluation, which can be also be used for checking shape-related errors [18].

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 [20], then was encoded functionally in [21] by using linear types [39]. Walker and Morrisett [40] 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, 37].

References


