Flattening Combinations of Arrays and Records

Reg Huijben, Jordy Aaldering[0009−0001−3018−5152], Peter Achten[0000−0002−3585−7165], and Sven-Bodo Scholz[0000−0002−8663−1043]

Radboud University, Nijmegen, Netherlands
{Reg.Huijben,Jordy.Aaldering,Peter.Achten,SvenBodo.Scholz}@ru.nl

Abstract. Flattening is known to be a performance-boosting technique to orchestrate parallel computations on arbitrarily deeply nested arrays. In this paper, we propose a flattening transformation that deals with nested data structures that are composed of combinations of arrays and records. We choose the functional array programming language SaC as basis for this work, as it already supports flattening of homogeneously nested arrays, i.e. arrays in which all elements have the same shape. We propose an extension of SaC's syntax for records that allows records and arrays to be used in homogeneously nested form, and provide an implementation of this record transformation in the SaC compiler. Based on that extension, we show how any legal program that operates with such data structures can be transformed into an equivalent one that does not require any records at runtime.

Keywords: nested data structures · records · array programming · program transformation

1 Introduction

Nested data structures usually impose a performance and memory management challenge. A straightforward implementation allocates memory for each level of such data structures, leading to situations where the components of a data structure are spread out over a wide region of memory. This introduces several challenges in terms of runtime efficiency. Many small allocations tax memory management and reference counting, and bulk operations on such data suffer from poor memory locality.

Functional programming languages such as Haskell, Clean, and ML all use algebraic data types as one of their main ways of constructing data. To keep the memory management overhead at bay, these languages typically use garbage collection schemes like mark-and-sweep [6] or more sophisticated versions thereof such as generational garbage collection [15]. More recently, work in the context of Koka [14,16] tackles the memory management overhead by statically identifying reuse opportunities. While these techniques help deal with the memory management overhead, they usually do not improve memory locality.

Dealing with locality, in particular when aiming at bulk-synchronous parallel operations, is most effective for data that is stored in a flat way in memory. To

achieve this, arrays are better suited: they lead to larger grain allocations and they allow for better data locality through the use of a flattened notation instead of a nested one. To further improve these aspects when creating nested arrays, performance oriented languages typically apply some form of flattening. Languages such as Accelerate [5], Futhark [9] and SaC [7,8] aim at purely flattened memory representations for nested arrays at runtime.

Once we add records to these languages, we run the risk of losing the advantages of a flattened representation, in particular when creating several layers of nested structures such as arrays of records that contain further arrays within their fields. A naive implementation would result in a large amount of small allocations, not to mention the loss of locality when accessing elements of fields in neighbouring records. Instead, we would like to reduce the number of allocations to only a single allocation for each field. The key idea is to enable flattening "through" records. To achieve this, we rewrite records into separate arrays; a vector of records that contain two vectors each thus turns into two two-dimensional arrays. Not only does this resolve the overhead due to allocations and the consequent loss of locality, it also implies that we no longer need to support records at runtime at all. The record extension that is described in this paper works for all uses of non-recursive records and in which all fields have known size. The extension has been implemented in SaC as of version 1.3.3-1175-1 and can be downloaded from https://sac-home.org/download:main.

The contributions of this paper are:

- A syntax for records, constructors and accessors that introduces immutable records while preserving compliance with an imperative specificational style.
- A transformation scheme that translates programs that operate on potentially nested record structures into programs that do without.
- Optimisations for eliding unused arguments and unmodified return values from function applications, respectively, and
- A real-world example showing how the presented techniques cooperatively transform nested arrays of records into record-free multi-dimensional arrays.

2 Single assignment C

Since we use the functional array language Single assignment C (SaC) [7,8] as implementation vehicle for our proposed flattening, we provide a quick overview of the key features that we use throughout the paper. For a more in-depth introduction we refer the reader to papers such as [7,8] or the online material available on https://sac-home.org/docs:main.

SaC combines a syntax very close to the imperative language C with a purely side-effect-free semantics. This is achieved by the addition of multi-dimensional, immutable arrays to the language core, paired with an exclusion of pointers.

2.1 Arrays in SaC

Arrays in SaC are multi-dimensional; each array has a dimensionality (also referred to as *rank*) and a *shape* which describes the legal index-space of the array.

This enables computations on multi-dimensional arrays to inspect and manipulate not only the data of an array but also its rank and shape. Table 1 shows a few examples for arrays in SaC. The left-most column provides an abstract / mathematical representation, the middle column contains a corresponding definition in SaC and in the right column we see the three conceptual components of these arrays: their rank (an integer value), their shape (a vector of natural numbers), and their data (a vector of the values of the array).

$\begin{pmatrix}1&2&3\\4&5&6\\7&8&9\end{pmatrix}$	`[[1,2,3], [4,5,6], [7,8,9]]`	rank: 2 shape: [3,3] data: [1,2,3,4,5,6,7,8,9]
$(1, 2, 3, -4, -5)$	`[1,2,3,-4,-5]`	rank: 1 shape: [5] data: [1,2,3,-4,-5]
$((true, false, true))$	`[[true, false, true]]`	rank: 2 shape: [1,3] data: [true,false,true]
0.5	`0.5`	rank: 0 shape: [ ] data: [0.5]

Table 1. Array representation in SaC

From this table, we can observe that all elements of any given array ultimately can be represented as a *flat* data vector in memory. While the SaC syntax in the middle column may suggest to the programmer that some arrays indeed are nested, in fact, they are not; in SaC, nesting inherently is indistinguishable from higher-dimensional arrays. This contrasts starkly with the idea of arrays prevalent in most non-array languages, where arrays are considered inherently one dimensional and where higher dimensional arrays are mimicked through nesting. As a consequence of SaC's choice to support higher-dimensional arrays in combination with a vector for representing the shape, nested notations in SaC are restricted to the homogeneous case, i.e. all inner expressions must have the same shape. An expression of the form `[[1], [2,3]]` is not admissible in SaC.

2.2 Tensor Comprehensions

The key language construct of SaC for defining arrays from other arrays is the *Tensor Comprehension* [19,22]. A tensor comprehension is essentially a mapping of index vectors to values. For example, the tensor comprehension

```
{ iv -> arr[iv] * 2  |  iv < shape(arr) }
```

defines an array of the same shape as array `arr`, whose value in each index position `iv` equates to twice the value of `arr` at the given position. Note here

that the index variable `iv` denotes an index vector with as many elements as the array `arr` has dimensions. The ability to express such operations that operate on arrays of arbitrary rank is referred to as *rank polymorphism*. If a fixed rank is expected, an explicit vector of indices can be used instead. For example,

```
{ [i,j] -> mat[j,i]  |  [i,j] < reverse(shape(mat)) }
```

transposes the two dimensional array `mat`. Despite having a fixed-rank index space, this tensor comprehension can be turned into a rank polymorphic operation by a small change into

```
{ [i,j] -> mat[j,i]  |  [i,j] < reverse(take([2], shape(mat))) }
```

The rank polymorphism of this expression is achieved through two measures. Firstly, selections in SaC select entire subarrays if there are fewer indices than the rank of the array to be selected from. Secondly, the specification of the upper bound through `reverse(take([2], shape(mat)))` ensures a two element vector as upper bound, even if the rank of the array `mat` is higher than 2.

2.3 Function Definitions, Types, and Type Patterns

Function definition in SaC look like their C counterparts. The main difference is the types that are supported in SaC. All types in SaC consist of an element type followed by a shape specification. If the shape specification is omitted, scalar arrays are expected. For example, the simple transpose can be defined as:

```
double[n,m] transpose(double[m,n] mat)
{
    return { [i,j] -> mat[j,i]  |  [i,j] < reverse(shape(mat)) };
}
```

Note here that the shape variables `m` and `n` define scalar integer values that can be referred to in the function body. This allows for a specification of the upper bound as `[n,m]` instead of `reverse(shape(mat))`.

```
double[n,m] transpose(double[m,n] mat)
{
    return { [i,j] -> mat[j,i]  |  [i,j] < [n,m] };
}
```

The dual use of shape variables for describing domain constraints as well as for capturing these values are referred to as *type patterns* [1].

3 A Syntax for Records in SaC

We try to match the syntax for records to that of C, following the overall syntactic approach of SaC. The required syntactic extensions are shown in Figure 1.

Record type declarations `<record-decl>` start with the `struct` keyword, followed by the name of the record type. Such a record can contain any positive number of fields, where each field can be of arbitrary type as long as the shapes of these field-types are statically fixed. This includes record types as well, provided that there is no recursive use.

```
<record-decl>     ::= 'struct' <id> '{' (<field-type> <id> ';')+ '}'
<field-type>      ::= <type> <static-shp>?
                    |  'struct' <id> <static-shp>?
<static-shp>      ::= '[' <int> (',' <int>)* ']'
<record-type>     ::= 'struct' <id> <shp>?
<record-field>    ::= <expr> '.' <id>
<full-constr>     ::= <id> '{' <expr> (',' <expr>)* '}'
<default-constr>  ::= <id> '{' '}'
<explicit-constr> ::= <id> '{' <explicit-set> (',' <explicit-set>)* '}'
<explicit-set>    ::= '.' <id> '=' <expr>
```

Fig. 1. EBNF for SaC record syntax.

Record types `<record-type>` extend the set of basic types[1] in SaC. They can occur in all positions where basic types are admissible such as function signatures, variable declarations, or type assertions. Similar to record type declarations, record types start with the `struct` keyword followed by the name of the record type. Note here that the restriction to statically fixed shapes only applies to record fields, not the construction of record types. A vector of statically unknown length of records `struct T` can be denoted by the type `struct T[.]`.

Fields of records `<record-field>` can be accessed by using a dot-symbol in infix notation. Similar to the modification of array elements, we support the use of such infix-dot-symbols on the left hand side of assignments as well.

We provide three ways for constructing record values as reflected by the last non-terminals in the syntax extension all of which constitute expressions in SaC. First, `<full-constr>` constructs a record value by enumerating a value for every field of the record, in the same order as those fields are defined in the corresponding record type declaration. Second, `<default-constr>` constructs a record value using default values for each of the fields. Examples of default values are 0 for numbers and *false* for booleans. The default value for an array is an array of the given shape, filled with the corresponding default values. Third, `<explicit-constr>` assigns values to selected fields and assigns the default value to the non mentioned fields.

4 Example

As a running real-world example we consider the *naive n-body problem*, which is concerned with predicting the individual motions of a group of celestial bodies. We define a record `Body` to keep track of these celestial bodies, as well as a `Vector3` record for the position of this body. Using these we define a function that computes the acceleration one body imposes on another, along with a function that computes the acceleration imposed on each body by all other bodies.

[1] `byte`, `short`, `int`, `long`, `longlong`, `ubyte`, `ushort`, `uint`, `ulong`, `ulonglong`, `bool`, `char`, `float`, `double`

```
struct Vector3 {
    double x;
    double y;
    double z;
};

struct Body {
    struct Vector3 pos;
    double mass;
};

struct Vector3
acc(struct Body b1, struct Body b2)
{
    dir = b2.pos - b1.pos;
    factor = l2norm(dir) == 0.0
        ? 0.0 : b2.mass / pow3(l2norm(dir));
    return factor * dir;
}

struct Body[n]
acc_v(struct Body[n] bodies)
{
    return { [i] -> rsum(1, { [j] -> acc(bodies[i], bodies[j])
                            | [j] < shape(bodies) })
           | [i] < shape(bodies) };
}
```

With these acceleration functions we can define a function that computes the updated positions of all bodies after some given delta time `dt`. For this each body additionally requires a velocity field, which we can add to the record without requiring modifications to the acceleration function, highlighting one of the benefits of using records.

```
struct Body {
    struct Vector3 pos;
    double[3] vel;
    double mass;
};

struct Body[n]
timestep(struct Body[n] bodies, double dt)
{
    acceleration = acc_v(bodies);
    bodies.pos += bodies.vel * dt;
    bodies.vel += acceleration * dt;
    return bodies;
}
```

5 Key Idea

In a typical naively nested memory layout for arrays of records (Fig. 2) each element in the array is a pointer to a single record object, where each of those objects has a single value for each field of that record. In the case that these fields are arrays or nested records, they are yet again pointers to that array or record object respectively. Such a naive nesting requires a large amount of memory allocations, and consequently results in a blowup in the amount of reference counting operations that is required. Furthermore this naive nesting has a negative impact on the memory locality of these objects; adjacent record

fields in an array of records are no longer adjacent in memory. This limits the applicability of vectorization of operations on arrays of records.

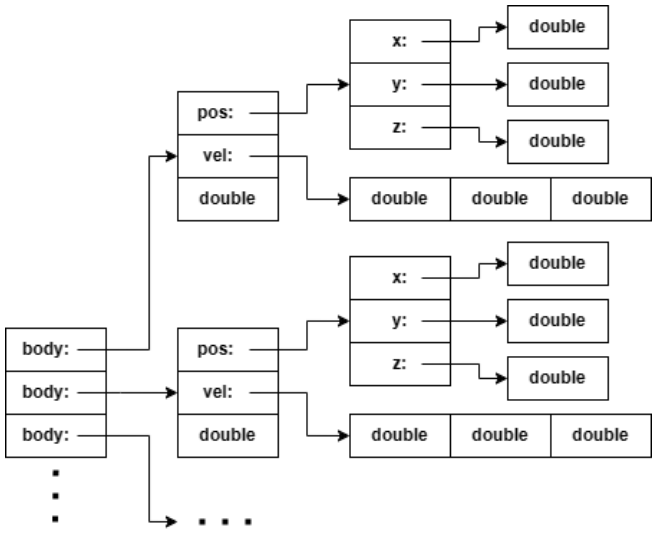

Fig. 2. Naively nested memory layout of the `Body` record.

In order to combat these drawbacks we rewrite records into a flattened array representation, where each base-type field is defined by a single array. Nested records are recursively flattened until only base-types remain. In the proposed flattened representation, only a single allocation is required for each base-type field of a record. Consequently this also decreases the amount of reference counting operations that is required, and opens the door to vectorization of operations on these arrays.

This transformation from arrays of records to flattened arrays additionally requires a rewrite of the rest of a program. Programs need to be rewritten to operate on multiple arrays, instead of on singular arrays of records. In the case of the n-body example, the record declaration is removed and the acceleration and timestep functions are rewritten to operate on arrays instead. These functions now require additional arguments and return multiple return values, instead of only a single record. Unused record fields are removed from these rewritten function definitions. Below we show the result of the rewriting process on the acceleration functions from the example.

```
double, double, double
acc(double x1, double y1, double z1,
    double x2, double y2, double z2,
    double mass2)
{
    xdir = x2 - x1;
    ydir = y2 - y1;
    zdir = z2 - z1;
```

```
    factor = l2norm(xdir, ydir, zdir) == 0.0
        ? 0.0 : m / pow3(l2norm(xdir, ydir, zdir));
    return (xdir * factor, ydir * factor, zdir * factor);
}

double[n], double[n], double[n]
acc_v(double[n] xs, double[n] ys, double[n] zs, double[n] masses)
{
    return { [i] -> rsum(1, { [j] -> acc(xs[i], ys[i], zs[i],
                                         xs[j], ys[j], zs[j],
                                         masses[j])
                                | [j] < [n] })
           | [i] < [n] };
}
```

The example shows that the transformation is not trivial. When arrays and records are combined and nested, the transformation to the required homogeneous arrays by SaC can become complex. To ensure that arrays of records can be transformed into homogeneous arrays we disallow (mutual) recursion and we require that array fields in records are of a statically known and fixed shape.

Additional effort is required for primitive functions, in order for those to be able to operate on record types. For example, it should be possible to get the dimensionality or shape of an array of records, or to select one of its elements, without the need for a special syntax. Similarly we need to ensure that tensor comprehensions and other array constructors are able to operate on multiple arguments. Already in the n-body example we see that often not all fields defined in a record are used by a function, and that not all fields of a record need to be returned. In order to limit the overhead introduced by an increase in the amount of arguments, and to maximise optimisation potential, two more optimisations are required to remove these unused arguments and return values.

6 Transformation

Similarly to earlier approaches (Homann et al. [11], Jubertie et al. [12], Kofler et al. [13]) we aim to improve the runtime performance of programs with records by transforming arrays of records into records of arrays. However, we go one step further and remove records from programs entirely. As discussed in Section 5 this transformation decreases the number of allocations and reference counting operations, as well as improving memory locality. An additional benefit is that removing records from programs in their entirety decreases the implementation effort of adding records to the language, since no modifications to the type system and other existing compiler phases are necessary. Especially in a large-scale project such as SaC, with many compiler phases that would require modifications to support records, such a transformation is paramount for a feasible implementation effort.

After the record transformation has been applied to a program, that program will no longer contain any record types. Instead, all record arguments and variable declarations are replaced by distinct arrays. Record constructors and field accessors and mutators are transformed into functions that operate on ar-

rays instead. Any user-defined functions and primitive operations on records are similarly transformed into operations on arrays.

In the case of SaC this record transformation is actually split up into two separate phases. During the parsing phase we replace records by a temporary "external" type, which is only fully removed after the type checking phase. This ensures that error messages generated by the type checker still pertain to record types without having to add support for records to the type checker, and that these error messages remain consistent with the pre-existing error messages. For example we get the following error message if we try to add a body to a string.

```
No definition found for a function "ArrayArith::+" that accepts
an argument of type "_MAIN::_struct_Body" as parameter no 1.
Full argument types are "( _MAIN::_struct_Body, String::string)".
```

However had records already been transformed into distinct arrays, the error message would look as follows and the relation between the error and the actual written code would be lost.

```
No definition found for a function "Array::+" that
expects 4 argument(s) and yields 1 return value(s)
```

For the sake of brevity however, we omit this additional step in the following transformation rules because it is specific to SaC and is not relevant to the actual transformation of records to arrays.

6.1 Constructors, Accessors, and Mutators

The first step in transforming programs with records into programs without records is to replace record constructors, accessors, and mutators by generated function definitions. For each record type in a program we generate a full constructor and a default constructor function, as well as accessor and mutator functions for every field of that record. Any record constructors, accessors, and mutators in the program are then replaced by applications of the corresponding generated functions.

Accessors and Mutators Normally an infix dot symbol is used to access or mutate the field of a record type, e.g. `bodies.pos`. In order to access or mutate a nested field, multiple accessors or mutators may be chained. However after programs have been transformed there will no longer be any records, and such a selection is no longer applicable. Instead we generate accessor and mutator functions for every field occurring in a record type. To access and mutate the position field of the body record, for example, these would be `body_set_pos` and `body_get_pos` respectively. For a record type with n fields, we generate an accessor and a mutator function for every field $i \in [1, n]$:

```
type1[shp1], ..., typen[shpn]
rt_get_idi(type1[shp1] id1, ..., typen[shpn] idn)
{
    return idi;
}
```

```
type1[shp1], ..., typen[shpn]
rt_set_idi(typei[shpi] value, type1[shp1] id1, ..., typen[shpn] idn)
{
    return (id1, ..., idi-1, value, idi+1, ..., idn);
}
```

Note that the argument and return types in these generated functions can still be records at this point. We expand these records at a later step, along with the expansion of records in user-defined functions.

Accessors and mutators through field selection can now be replaced by applications of these generated functions. For a field selection `id = x.y` on the right-hand-side of a let-expression we apply the accessor function `id = rt_get_y(x)`, whereas if on the left-hand-side there occurs a field selection `x.y = expr`, the mutator function is applied `rt_set_y(expr, x)`.

Constructors Syntactically there are three kinds of record constructors: a full constructor that expects a value for each field in order, a default constructor that takes no arguments and assigns a default (zero) value to each field, and an explicit constructor with only the values for some fields explicitly defined. For every record type `rt` in the program we generate a new function definition for both the full and the default constructor:

```
type1[shp1], ..., typen[shpn]
new_rt(type1[shp1] id1, ..., typen[shpn] idn)
{
    return (id1, ..., idn);
}

type1[shp1], ..., typen[shpn]
zero(type1[*] id1, ..., typen[*] idn)
{
    return new_rt(genarray([shp1], zero([:type1])
                  ...
                  genarray([shpn], zero([:typen]));
}
```

Where `[:type]` is an empty array of the given type. This ensures that the correct overload of the `zero` function is applied, returning the default value for that type.

Now whenever we encounter a full constructor of the form `rt{expr1, ..., exprn}` we replace it by `new_rt(expr1, ..., exprn)`, and whenever we encounter a default constructor of the form `rt{}` we replace it by `zero([:rt])`. Finally, if we encounter an explicit constructor of the form

```
rt{.fieldq = valueq, ..., .fieldr = valuer}
```

we first apply the default constructor, followed by a chain of mutator functions for the explicitly given fields.

```
rt_set_idr(valuer,
    ...
        rt_set_idq(valueq, zero([:rt]));
```

6.2 Expanding Records to Base Types

After record-specific syntax has been replaced by function applications, we must ensure that those and all other functions no longer contain any record types and

record variables. To achieve this we replace those record types and variables by the fields of those records instead. We look at the `l2norm` function as an example:

```
double
l2norm(struct Vector3 v)
{
    vx = vector3_get_x(v);
    vy = vector3_get_y(v);
    vz = vector3_get_z(v);
    return sqrt(vx * vx + vy * vy + vz * vz);
}
```

We aim to transform this function into one without any record types.

```
double
l2norm(double x, double y, double z)
{
    vx = vector3_get_x(x, y, z);
    vy = vector3_get_y(x, y, z);
    vz = vector3_get_z(x, y, z);
    return sqrt(vx * vx + vy * vy + vz * vz);
}
```

In this example we see how the record argument `struct Vector3 v` is separated into three distinct arguments x, y, and z; corresponding to the three fields of the `Vector3` record. Furthermore, all occurrences of this argument `v` are also replaced by the same three variables instead.

In the case where records and arrays are nested, this tranformation becomes non-trivial. Consider the signature of the `timestep` function from the example. Here the given record type is an array instead of a scalar value, furthermore it contains the nested `Vector3` record for the position, and an array `double[3]` for the velocity.

```
struct Body[n]
timestep(struct Body[n] bodies, double dt)
```

After expanding the `Body` record, and the nested `Vector3` record, we expect the function signature to look like:

```
double[n], double[n], double[n], /* pos  */
double[n,3],                     /* vel  */
double[n]                        /* mass */
timestep(double[n] x, double[n] y, double[n] z,
         double[n,3] vel, double[n] mass,
         double dt)
```

This example highlights multiple interesting cases. Firstly, although the `mass` field of the body record is a scalar `double`, after expansion we expect it to become an array of type `double[n]` since `struct Body[n]` denotes an array of records instead of a scalar record, which consequently applies to all fields of the record. Secondly, the body record contains a nested `Vector3` record, which itself is then expanded into its three distinct x, y, and z fields. Similarly to the mass field, here we must also ensure that these fields become arrays of type `double[n]`. Finally there is the velocity field, which itself is already an array of type `double[3]`. In this case, the shape of the bodies array must be concatenated with the shape of the velocity field, resulting in the type `double[n,3]`.

6.3 Denesting Fields of Nested Records

In order to be able to apply this transformation we need to have a mapping of record types to the fully denested fields and expanded shapes. We call this mapping σ, and allow it to be indexed by a record type to get the expanded fields of that record. In the case of the `Vector3` record, this would look as follows:

$$\sigma[\texttt{Vector3}] = \{\, \text{x}\colon \texttt{double},\ \text{y}\colon \texttt{double},\ \text{z}\colon \texttt{double} \,\}$$

Whereas for the `Body[n]` array of records we expect the following:

$$\begin{aligned}\sigma[\texttt{Body[n]}] = \{\, &\text{x}\colon \texttt{double[n]},\ \text{y}\colon \texttt{double[n]},\ \text{z}\colon \texttt{double[n]} \\ &\text{vel}\colon \texttt{double[n,3]},\ \text{mass}\colon \texttt{double[n]} \,\}\end{aligned}$$

Note that this environment does not contain the `pos` field, and instead has already expanded that record into its nested x, y, and z fields.

Denesting Records We populate this environment using a function called $\texttt{Denest}_{\texttt{rt}}$. This function expects a record declaration as its first argument, and the (initially empty) accumulated environment σ as its second argument. All fields of this record are then denested separately using a function $\texttt{Denest}_{\texttt{f}}$, whose resulting environments are combined to form the mapping σ of the record type. We apply this denesting function to all record types in a program, from top to bottom.

$$\texttt{Denest}_{\texttt{rt}}\left(\begin{array}{l}\texttt{struct rt \{} \\ \quad \texttt{type}_1\texttt{[shp}_1\texttt{]: id}_1\texttt{;} \\ \quad \ldots \\ \quad \texttt{type}_n\texttt{[shp}_n\texttt{]: id}_n\texttt{;} \\ \texttt{\};}\end{array},\ \sigma\right) = \begin{array}{l}\texttt{Denest}_{\texttt{f}}(\texttt{type}_1,\ \texttt{shp}_1,\ \texttt{id}_1,\ \sigma) \\ \cup\ \ldots \\ \cup\ \texttt{Denest}_{\texttt{f}}(\texttt{type}_n,\ \texttt{shp}_n,\ \texttt{id}_n,\ \sigma)\end{array}$$

Because from this point on this record may be used in a nested fashion in all following record declarations, we add this record to the environment.

Denesting Fields The main body of work lies in the $\texttt{Denest}_{\texttt{f}}$ function. Given a field name *id* and its corresponding type and shape, this function computes the environment σ' of that shape. Additionally this function requires the thus far accumulated environment σ, which is required when looking up the environment of a previously defined record type in the case that *type* is a record type. We distinguish between base-type fields and record type fields. In the case that we encounter a base-type field *id*, be it a scalar or an array, a mapping can immediately be added to the environment without additional work.

$$\begin{aligned}\texttt{Denest}_{\texttt{f}}(\mathit{basetype},\ [\,],\ \mathit{id},\ \sigma) &= \{\, \mathit{id} : \mathit{basetype} \,\} \\ \texttt{Denest}_{\texttt{f}}(\mathit{basetype},\ \mathit{shp},\ \mathit{id},\ \sigma) &= \{\, \mathit{id} : \mathit{basetype}[\mathit{shp}] \,\}\end{aligned}$$

If instead *id* is a scalar record type, we lookup the previously computed environment of that record type. Since this field does not have a shape, there is nothing

more to do and we can copy the environment as is. Because we require that records are defined top-to-bottom, this environment must exist at this point. Otherwise an incorrect program was provided and we can raise an error.

$$\texttt{Denest}_\texttt{f}(\mathit{recordtype},\ [\,],\ \mathit{id},\ \sigma) = \sigma[\mathit{recordtype}]$$

As we have seen in the example, we need to do some additional work in the case that *id* is an array of a record type. Not only does that record type need to be denested, but the shape of *id* must be prepended to all fields of the denested record type as well, for which we use a new function: `prepend`.

$$\texttt{Denest}_\texttt{f}(\mathit{recordtype},\ \mathit{shp},\ \mathit{id},\ \sigma) = \texttt{prepend}(\mathit{shp},\ \sigma[\mathit{recordtype}])$$

Following is the `prepend` function. Its first argument is the shape we want to prepend, and the second argument is the environment of the record type to which we want to prepend this shape. For every field in this environment, we then prepend the given shape to the previous shape, resulting in a new environment that has the same identifiers and types as the given argument, but now with expanded shapes.

$$\texttt{prepend}(\mathit{shp}_{rt},\ \sigma_{rt}) = \left\{ \begin{array}{l} \mathit{id}_1 : \mathit{type}_1[\mathit{shp}_{rt} :: \mathit{shp}_1], \\ \ldots, \\ \mathit{id}_n : \mathit{type}_n[\mathit{shp}_{rt} :: \mathit{shp}_n] \end{array} \right\}$$
$$\text{where } \sigma_{rt} = \{\, \mathit{id}_1 : \mathit{type}_1[\mathit{shp}_1],\ \ldots,\ \mathit{id}_n : \mathit{type}_n[\mathit{shp}_n] \,\}$$

No case distinction is needed for scalar fields, since their shape is the empty list (as seen in Section 2), and thus the concatenation $\mathit{shp} :: []$ will act as an identity on *shp*. Additionally we do not need to worry about nested records at this point, as they have already been denested and thus at this point we only have base-types.

Using this environment we can now actually apply the transformation proposed in Section 6.2. Whenever we encounter a record type, a record argument, or a record identifier we replace it by the expanded base-types and identifiers accordingly.

6.4 Primitive Functions

This record argument expansion applies to both the formal arguments of a function definition, and the actual arguments of applications of those functions. Consequently, the number of formal arguments and the number of actual arguments of user-defined functions will remain equal after the record transformation. Because of this no additional work is required for user-defined functions.

However in the case of primitive functions this leads to a problem. The actual record arguments of primitive function applications will have been expanded into multiple arguments, but since these functions are defined as compiler primitives they do not have a corresponding function definition in the program. As a result, the number of actual arguments and the number of expected arguments

for these primitive functions will no longer be the same. Since we expose record types to users as primitive types, we should also ensure that primitive operations on records are also possible. Namely, we must ensure that it is possible to get the dimensionality and shape of a record array, and it should be possible to select into this array. Additional work is required with regards to primitive function applications in order to ensure that they remain valid after the record transformation.

Consider the built-in `shape` primitive that we use in the running example to find the upper bound of the tensor comprehension. Given a single array, this primitive function returns the shape of that array. Such primitive functions should be applicable to arrays of records as well, for example to get the shape of an array of bodies. After the transformation, records arguments will have been expanded into multiple arguments, leading to incorrect applications of these primitive functions. For example,

```
shp = shape(bodies);
```

will be transformed into an application without records

```
shp = shape(bodies_pos, bodies_vel, bodies_mass);
```

This transformed code is no longer valid. The shape primitive expects only a single argument, however it now receives three arguments. To resolve this we might decide to arbitrarily take the first field of the record, in this case `bodies_pos`, and use only that value in primitive functions instead. However, this field might already have a shape within the record itself. Such is the case with `bodies_pos`, where `pos` itself is already a three-element integer vector. After transformation, this shape will then be `[n,3]`, whereas given the definition of `bodies` in the argument `struct Body[n] bodies`, we would expect its shape to be `[n]`.

Here we can rely on the fact that record fields are always arrays of a statically known shape. Because we know that `pos` is a one-dimensional vector, we can statically decide that the last element of the resulting shape vector (`[3]`) should be dropped from the resulting shape.

```
shp = drop(-1, shape(bodies_pos));
```

We apply a similar approach to the remaining primitive functions that require modification, such as `dim` (dimensionality) and `sel` (selection). However for the sake of brevity we omit those cases here.

6.5 Tensor Comprehension

Whereas tensor comprehensions on records previously operated on only that single record value, after the record expansion these tensor comprehensions operate on and return multiple values. For example, a tensor comprehension that generates a list of bodies

```
bodies = { iv -> Body{} | iv < [N] };
```

is transformed into a tensor comprehension that returns three values:

```
pos, vel, mass = { iv -> ([0,0,0], [0,0,0], 0) | iv < [N] };
```

This requires that tensor comprehensions, and similar constructions such as with-loops, are able to operate on and return multiple values. In the case of SaC, this is already supported [22].

7 Unused Argument Removal

After applying the record transformation to the acceleration function from the running example, the fields of all record arguments are expanded into separate arguments, which leads to the following intermediate function definition:

```
double[n], double[n], double[n]
acc_v(double[n] xs, double[n] ys, double[n] zs,
      double[n,3] vels, double[n] masses)
{
    return { [i] -> rsum(1, { [j] -> acc(xs[i], ys[i], zs[i],
                                         xs[j], ys[j], zs[j],
                                         masses[j])
                               | [j] < [n] })
           | [i] < [n] };
}
```

However the velocity field is not used within the body of this function. Typically the number of arguments quickly explodes when expanding records, which has a negative effect on the compilation time of programs, this would especially be the case for the timestep function for example. Furthermore, calling sites of these functions are not able to apply optimisations such as dead code removal on these unused arguments, limiting further optimisation potential. Removing these unused arguments from function signatures and corresponding function applications would not only improve compilation times, but also opens the door to better optimisation of those calling sites. In certain cases this may make it possible to remove some flattened fields entirely, avoiding the need for unnecessary memory allocation of those unused fields.

After applying this unused argument removal optimisation any calling sites of the accelerate function will only pass the required arguments instead of also including the velocity field, and the function signature becomes:

```
double, double, double
double[n], double[n], double[n]
acc_v(double[n] xs, double[n] ys, double[n] zs, double[n] masses)
```

Naively applying this unused argument removal optimisation only once is not sufficient. After applying the optimisation once, it may expose additional arguments that have become unused. Therefore this optimisation is applied iteratively in the optimisation cycle, which repeatedly applies a suite of program optimisations.

8 Unused Return Removal

Typically when a function expects a record and returns a modified version of that record, only some of the fields of that record will actually have been changed. This occurs in the `timestep` function from the running example, which returns a modified lists of bodies without changing their masses.

```
double[n], double[n], double[n], double[n,3], double[n]
timestep(double[n] x, double[n] y, double[n] z, double[n,3] vel,
         double[n] mass, double dt)
{
    ...
    return (x, y, z, vel, mass);
}
```

Here only the positions and velocities of the given bodies are changed, whereas the mass remains unchanged and is still equal to the value of the given argument after returning. Ideally, we would remove this mass from the return type to increase the potential for further optimisations. I.e., after removing the mass from the return types unused argument removal is able to remove this mass from the arguments as well, which as discussed previously opens the door for other optimisations.

Because overloaded functions must always have the same number of return types, we require a different approach compared to unused argument removal. This restriction however actually leads to a very simple implementation. During the optimisation cycle we annotate return types that remain unchanged with respect to one of the arguments, similarly to the annotation done by unused argument removal. However since we cannot change the return types of this function, we instead update calling sites of this function during the optimisation cycle. If we encounter an application that assigns an identifier whose corresponding return type is marked as unchanged, we replace any following occurrences of that identifier by the argument instead.

```
x2, y2, z2, vel2, mass2 = timestep(x, y, z, vel, mass, dt);
x3, y3, z3, vel3, mass3 = timestep(x2, y2, z2, vel2, mass2, dt);
x4, y4, z4, vel4, mass4 = timestep(x3, y3, z3, vel3, mass3, dt);
```

Since `timestep` does not change the value of `mass`, the returned value is thrown away and all following occurrences of `mass2` and `mass3` are replaced by `mass`.

```
x2, y2, z2, vel2, _ = timestep(x, y, z, vel, mass, dt);
x3, y3, z3, vel3, _ = timestep(x2, y2, z2, vel2, mass, dt);
x4, y4, z4, vel4, _ = timestep(x3, y3, z3, vel3, mass, dt);
```

9 Runtime Performance

We investigate whether the record flattening transformation introduces any significant overhead compared to hand-optimised code by benchmarking two implementations of the n-body algorithm: a hand-optimised version without records, and a version that makes use of records. In order to make optimal use of vectorization, the hand-optimised version operates on seven distinct arrays; three arrays for the x, y, and z coordinates, three arrays for the x, y, and z velocities, and a single array for the masses.

The record implementation from the example already does this through the use of the `Vector3` record, which highlights how without much effort we can play around with the memory layout. We do however modify the velocity field to also use a `Vector3` instead of a `double[3]`. The only necessary change then is to replace occurrences of `double[3]` types for velocities by `Vector3` types.

We benchmark these two implementations on 50,000 bodies, applying the timestep iteration 10 times. These benchmarks are repeated ten times on a Dell PowerEdge R7525 rack server containing two AMD EPYC 7313 CPUs, which have 16 cores running at 3.7GHz. Figure 3 plots the results of these benchmarks, which shows that the record version with `Vector3`s performs similarly to the hand-optimised version.

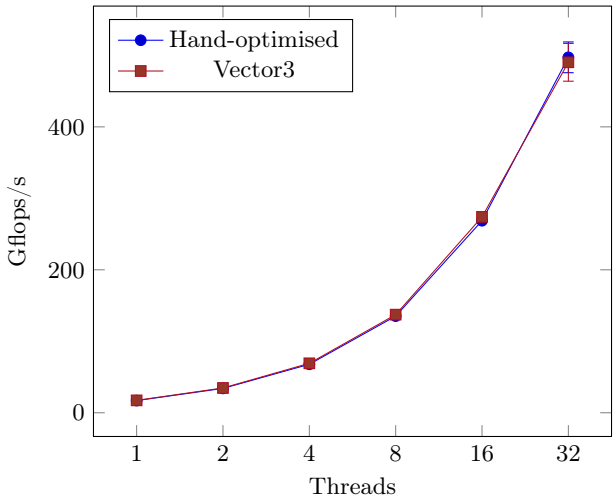

Fig. 3. Compute rate in GFLOP/s.

10 Related Work

This work expands upon previous work in the context of SaC [17], which explores how records can be added to the language. We extend on that work to fully support arbitrary non-recursive nesting of arrays and records, including all array operations on records as well as on tensor comprehensions and with-loops for constructing arrays of records.

The idea of converting the nesting structure of records and arrays is by no means new. Several papers investigate the performance gains of such rearrangements. In particular the performance benefits from records of arrays over arrays of records are well established. It has manifested itself in various software design strategies as, for example, the data-oriented design, which is applied in the context of game development [3].

However, it is also widely recognised that arrays of records for many application areas quite naturally lend themselves for specifying algorithms. For example, Homann et al. [11] discuss this ambiguity: they show the specificational benefits

of arrays of records and demonstrate the performance benefits of converting arrays of records into records of arrays. As a potential solution, they propose a new data structure based on C++ templates that provides a programming interface similar to arrays of records yet results in code with records of arrays. Jubertie et al. [12] and Kofler et al. [13] further improve on that idea. They suggest a more flexible interface that enables a decoupling of data specification and the layout that ultimately is being generated for the execution.

The main differences between that line of research and ours is that in our work the conversion from arrays of records is fully compiler-driven. There is no need for the programmer to create any specific scaffolding or to make decisions about the layout. There is not even a need for the programmer to be aware of the conversion. By converting all records to arrays, our approach goes even one step further: the resulting programs no longer contain any records.

Further related work suggests library [4,21] or DSL [20] based approaches for converting records to arrays. Similar to the template-based approaches, these require developers to actively consider the performance of their programs. By instead making this transformation a core part of the SaC language, we allow developers to focus on the "what" of their program instead of on the "how".

The two new optimisations this paper proposes, unused argument removal and unused return removal, bear quite some relation to standard optimisations found in the context of functional programming languages as well as optimising compilers in general. Unused Argument Removal can be seen as a dual to strictness analysis [18,23]. Instead of identifying function arguments that are guaranteed to be evaluated, we identify arguments that are guaranteed not to be needed. This allows us to eliminate these arguments, effectively avoiding the corresponding beta-reductions when applying the functions. Unused Return Removal also avoids redexes by forwarding values within the calling context. This can be seen as a special form of variable propagation, a standard optimisation in compilers (e.g. [2]). The main difference here is that we propagate the arguments to functions where we statically find out that the correponding parameter is directly propagated into one of the results. The identification of such functions, by itself, can be seen as a special form of program slicing [24]. However, instead of computing the slices, we only identify these trivial slices for applying the propagation.

11 Conclusion

This paper proposes an extensions of the array language SaC by records. As with any other basic type in SaC, all records are implicitly n-dimensional arrays of records; a scalar record has dimensionality 0 and an empty shape. As records are normal expressions, all array constructing operations of SaC can be applied to records as well.

The key contribution of this paper is a transformation that systematically transforms all arrays of records into records of arrays. As a consequence, any non-recursive nesting of arrays and records can be transformed into records that

appear as scalar records on the top level only and, thus, can be replaced by their fields entirely. Meaning that there is no need to support records at runtime at all. This not only reduces the implementation effort but it also avoids the overhead of nested memory allocations entirely.

The price that needs to be paid for getting rid of all records at runtime is an increase in function arguments and return values. To counter this effect, we also propose two optimisations that identify those arguments and return values that do not contribute to the actual computations within functions that operate on records. This reduces the increase of function signatures that results from our transformation to the necessary minimum.

To be able to perform the proposed transformation in all cases, we need to impose two restrictions. Firstly, we have to restrict arrays in record fields to be of a fixed shape. This restriction is a consequence of SaC's limitation to homogeneously nested arrays. In array languages without this restriction, arbitrarily shaped record fields would be directly possible, using the same transformation scheme as proposed in the paper.

Secondly, we have to restrict records to be defined in a non-recursive manner. This is a fundamental limitation of the proposed flattening approach. As the memory size of flattened arrays needs to be known prior to computing array values, recursively defined records cannot be flattened away as proposed in the paper. However, for most compute intensive applications such recursion is not needed. Language support for such recursion surely is possible and would constitute an extension of the language capabilities of SaC, but it would be orthogonal to the transformation work presented here. Our current implementation in the SaC compiler at www.sac-home.org does not support such recursion.

While our records do not support methods as part of the records, the function overloading of SaC provides the programmer with the opportunity to define record-type-specific functionalities. Even though no subtyping relation between record types is supported in SaC, it does support subtyping in array types [10] in general. Extending this to records as they are suggested in this paper might be interesting future work.

Acknowledgements

The authors thank Thomas Koopman for his help with optimising and benchmarking the record implementation in the SaC compiler, and the anonymous reviewers for their constructive feedback.

References

1. Aaldering, J., Scholz, S.B., Van Gastel, B.: Type patterns: Pattern matching on shape-carrying array types. In: Proceedings of the 35st Symposium on Implementation and Application of Functional Languages. IFL '23, Association for Computing Machinery, New York, NY, USA (2024). https://doi.org/10.1145/3652561.3652572

2. Aho, A., Sethi, R., Ullman, J.: Compilers – Principles, Techniques, and Tools. Addison-Wesley (1986), iSBN 0-201-10194-7
3. Bayliss, J.D.: Developing games with data-oriented design. In: Proceedings of the 6th International ICSE Workshop on Games and Software Engineering: Engineering Fun, Inspiration, and Motivation. p. 30–36. GAS '22, Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3524494.3527626
4. Bell, N., Hoberock, J.: Thrust: A productivity-oriented library for CUDA. In: mei W. Hwu, W. (ed.) GPU Computing Gems Jade Edition, pp. 359–371. Applications of GPU Computing Series, Morgan Kaufmann, Boston (2012). https://doi.org/10.1016/B978-0-12-385963-1.00026-5
5. Chakravarty, M.M., Keller, G., Lee, S., McDonell, T.L., Grover, V.: Accelerating Haskell array codes with multicore GPUs. p. 3–14. DAMP '11, Association for Computing Machinery, New York, NY, USA (2011). https://doi.org/10.1145/1926354.1926358
6. Dijkstra, E.W., Lamport, L., Martin, A.J., Scholten, C.S., Steffens, E.F.M.: On-the-fly garbage collection: An exercise in cooperation. Commun. ACM **21**(11), 966–975 (nov 1978). https://doi.org/10.1145/359642.359655
7. Grelck, C.: Single Assignment C (SaC) High Productivity Meets High Performance, pp. 207–278. Springer Berlin Heidelberg, Berlin, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32096-5
8. Grelck, C., Scholz, S.B.: SaC – A functional array language for efficient multithreaded execution. International Journal of Parallel Programming **34**(4), 383–427 (Aug 2006). https://doi.org/10.1007/s10766-006-0018-x
9. Henriksen, T., Serup, N.G.W., Elsman, M., Henglein, F., Oancea, C.E.: Futhark: Purely functional GPU-programming with nested parallelism and in-place array updates. SIGPLAN Not. **52**(6), 556–571 (jun 2017). https://doi.org/10.1145/3140587.3062354
10. Herhut, S., Scholz, S.B.: Generic programming on the structure of homogeneously nested arrays (2006), https://sac-home.org/_media/publications:pdf:gpotsohna.pdf
11. Homann, H., Laenen, F.: SoAx: A generic C++ structure of arrays for handling particles in HPC codes. Computer Physics Communications **224**, 325–332 (2018). https://doi.org/10.1016/j.cpc.2017.11.015
12. Jubertie, S., Masliah, I., Falcou, J.: Data layout and SIMD abstraction layers: Decoupling interfaces from implementations. In: 2018 International Conference on High Performance Computing & Simulation (HPCS). pp. 531–538 (2018). https://doi.org/10.1109/HPCS.2018.00089
13. Kofler, K., Cosenza, B., Fahringer, T.: Automatic data layout optimizations for GPUs. In: Träff, J.L., Hunold, S., Versaci, F. (eds.) Euro-Par 2015: Parallel Processing. pp. 263–274. Springer Berlin Heidelberg, Berlin, Heidelberg (2015). https://doi.org/10.1007/978-3-662-48096-0_21
14. Leijen, D.: Koka: Programming with row polymorphic effect types. Electronic Proceedings in Theoretical Computer Science **153**, 100–126 (jun 2014). https://doi.org/10.4204/eptcs.153.8
15. Lieberman, H., Hewitt, C.: A real-time garbage collector based on the lifetimes of objects. Commun. ACM **26**(6), 419–429 (jun 1983). https://doi.org/10.1145/358141.358147
16. Lorenzen, A., Leijen, D., Swierstra, W.: FP^2: Fully in-place functional programming. Proc. ACM Program. Lang. **7**(ICFP) (aug 2023). https://doi.org/10.1145/3607840

17. Luyat, H.: A lightweight implementation of records in functional array language SaC (2009), bachelor thesis, University of Amsterdam
18. Mycroft, A.: The theory and practice of transforming call-by-need into call-by-value. In: Robinet, B. (ed.) International Symposium on Programming. pp. 269–281. Springer Berlin Heidelberg, Berlin, Heidelberg (1980). https://doi.org/10.1007/3-540-09981-6_19
19. Scholz, S.B., Šinkarovs, A.: Tensor comprehensions in SaC. In: Proceedings of the 31st Symposium on Implementation and Application of Functional Languages. IFL '19, Association for Computing Machinery, New York, NY, USA (2021). https://doi.org/10.1145/3412932.3412947
20. Springer, M., Masuhara, H.: Ikra-cpp: A C++/CUDA DSL for object-oriented programming with structure-of-arrays layout. In: Proceedings of the 2018 4th Workshop on Programming Models for SIMD/Vector Processing. WPMVP'18, Association for Computing Machinery, New York, NY, USA (2018). https://doi.org/10.1145/3178433.3178439
21. Strzodka, R.: Abstraction for AoS and SoA layout in C++. In: mei W. Hwu, W. (ed.) GPU Computing Gems Jade Edition, pp. 429–441. Applications of GPU Computing Series, Morgan Kaufmann, Boston (2012). https://doi.org/10.1016/B978-0-12-385963-1.00031-9
22. Šinkarovs, A., Scholz, S.B.: Parallel scan as a multidimensional array problem. In: Proceedings of the 8th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming. p. 1–11. ARRAY 2022, Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3520306.3534500
23. Wadler, P., Hughes, R.J.M.: Projections for strictness analysis. In: Proceedings of the Functional Programming Languages and Computer Architecture. p. 385–407. Springer-Verlag, Berlin, Heidelberg (1987). https://doi.org/10.1007/3-540-18317-5_21
24. Weiser, M.: Program slicing. IEEE Transactions on Software Engineering **SE-10**(4), 352–357 (1984). https://doi.org/10.1109/TSE.1984.5010248