

Flattening, not only for Arrays alone, but for Combinations of Arrays and Records as well — Extended Abstract —

Reg Huijben¹, Jordy Aaldering¹[0009-0001-3018-5152], Peter
Achten¹[0000-0002-3585-7165], and Sven-Bodo Scholz¹[0000-0002-8663-1043]

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

Abstract. Flattening is known to be a performance-boosting technique to orchestrate parallel computations on nested arrays. In this paper, we propose a flattening transformation that deals with nested data structures that are composed 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. We propose an extension of SaC’s syntax for records that allows records and arrays to be used in nested form. 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 challenge. A straightforward implementation allocates memory for each level of such data structures, leading to situations where the components of a single data structure are spread out over a wide region of memory. This introduces several challenges in terms of runtime efficiency. Many small allocations place a challenge on the efficiency of memory management and bulk operations on such data suffer from poor memory locality.

Functional programming languages such as Haskell, Clean, or 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 [2] or more sophisticated versions thereof such as generational-garbage-collection [5]. More recently, work in the context of Koka [6] tackles the memory management overhead by statically identifying reuse opportunities. While these techniques help dealing with the memory management overhead, they usually do not improve memory locality.

Dealing with locality, in particular when aiming at bulk-synchronous parallel operations, requires the data to be stored in a flat way in memory. To achieve

this, arrays are better suited; they lead to larger grain allocations and they have better data locality. To further improve these aspects when creating nested arrays, performance oriented languages typically apply some form of flattening. Languages such as Accelerate [1], Futhark [4] or SaC [3] aim at purely flattened memory representations for nested arrays at runtime.

Once we add records to these languages, we run the risk of loosing the advantages of a flattened representation, in particular, when creating several layers of nested structures. As an example, consider a simple n-body simulation. A body is described by a record that consists of a position, a velocity, and a mass:

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

Using such a record definition, we can describe a system of bodies as a vector of bodies:

```
struct Body[10000] bodies;
```

A naive implementation would result in 30,001 allocations. We have one allocation for the 10,000 bodies, 10,000 allocations for the body records, and each body record would point to two separately allocated 3-element vectors of floating point values.

In particular the 20,000 allocations of 3-element vectors pose a major overhead, not to mention the loss of locality when accessing neighbouring positions. Instead, we would like to reduce the number of allocations to three. The key idea is to enable the flattening “through” the records. To achieve this, we rewrite our `bodies` into three separate arrays:

```
double[10000, 3] bodies_Body_pos;
double[10000, 3] bodies_Body_vel;
double[10000] bodies_Body_mass;
```

Not only does this resolve the overhead due to allocations and loss of locality, it also implies that we no longer need to support records at runtime at all.

The contributions of this paper are:

- A syntax for records, constructors and accessors that introduces immutable records while preserving compliance with an imperative specification style.
- A type system extension for type checking programs that use such records.
- A transformation scheme that translates programs that operate on potentially nested record structures into programs that do without.
- An optimisation for eliding unused arguments of function calls.
- An extended example, showing how the presented techniques cooperatively transform nested arrays of records into record-free multi-dimensional arrays.

2 Example

To demonstrate the interplay of record elimination and the Unused Argument Removal (UAR) optimisation, we look at a simple example, the *Naive n-body problem* shown in Fig. 1.

```

double[3] acc (struct Body b, struct Body b2)
{
    dir = b2.pos - b.pos;
    return all (dir == 0.0)
        ? [0.0, 0.0, 0.0]
        : b2.mass * dir / l2norm (dir) ^ 3;
}

struct Body[n] timeStep (struct Body[n] bs, double dt)
{
    acc = { [i] -> rsum (1, { [j] -> acc (bs[i], bs[j])
                                | [j] < shape (bs) })
            | [i] < shape (bs) };
    bs.pos += bs.vel * dt;
    bs.vel += acc * dt;

    return bs;
}

```

(a) Source code using records.

⇓

```

double acc (double b_Body_mass, double[3] b_Body_pos,
            double b2_Body_mass, double[3] b2_Body_pos)
{
    dir = b2_Body_pos - b_Body_pos;
    return all (dir == 0.0)
        ? [0.0, 0.0, 0.0]
        : b2_Body_mass * dir / l2norm (dir) ^ 3;
}

double[n],
double[n,3],
double[n,3] timeStep (double[n] bs_Body_mass,
                      double[n,3] bs_Body_pos,
                      double[n,3] bs_Body_vel,
                      double dt)
{
    acc = {[i] -> rsum (1, {[j] -> acc (bs_Body_mass[i],
                                        bs_Body_pos[i],
                                        bs_Body_mass[j],
                                        bs_Body_pos[j])
                                | [j] < take([1], shape (bs_Body_pos)))}
            | [i] < take([1], shape (bs_Body_pos))});
    bs_Body_pos += bs_Body_pos * dt;
    bs_Body_pos += acc * dt;

    return (bs_Body_mass, bs_Body_pos, bs_Body_vel);
}

```

(b) Transformed code after records and accessor-functions have been eliminated, unused function arguments have been stripped away, and applications of `shape` have been adjusted.

Fig. 1: Naive n-body simulation code in SaC using records.

In Fig. 1a, we see the original code making use of the record type `struct Body` that we use throughout the paper.

The function `acc` takes two bodies `b` and `b2` and computes the acceleration that body `b2` generates on body `b`. As long as the two bodies are not at the same position, the body `b2` creates an acceleration of the body `b` towards it which equates `b2`'s mass divided by the distance of the two bodies to the power of 3.

The function `timeStep` takes a vector of `n` bodies `bs` and a time delta `dt` and computes the gravitational effects after `dt` on those bodies. First, it sums up the acceleration between all pairs of bodies `i` and `j`. Then, it updates the positions of all bodies by using the current velocities, followed by an update of all velocities due to the initially computed accelerations.

In Fig. 1b, we see the code that results from applying the transformations described in this paper. Not only have all references to structures been replaced by records for the individual fields, but we can also see how the formal parameters and actual arguments to both functions have been adjusted accordingly. Note here, that only those arguments remain that are actually being used within the function body. The velocity component in the arguments to the function `acc` has been eliminated through `UAR`. Finally, the shape computations within the body of the function `timeStep` have been adjusted, reflecting the elimination of the record structure.

References

1. Chakravarty, M.M., Keller, G., Lee, S., McDonell, T.L., Grover, V.: Accelerating haskell array codes with multicore gpus. In: Proceedings of the Sixth Workshop on Declarative Aspects of Multicore Programming. p. 3–14. DAMP '11, Association for Computing Machinery, New York, NY, USA (2011). <https://doi.org/10.1145/1926354.1926358>
2. 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>
3. Grelck, C., Scholz, S.B.: Sac—a functional array language for efficient multi-threaded execution. *International Journal of Parallel Programming* **34**(4), 383–427 (Aug 2006). <https://doi.org/10.1007/s10766-006-0018-x>
4. Henriksen, T., Serup, N.G.W., Elsmann, 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>
5. 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>
6. Lorenzen, A., Leijen, D., Swierstra, W.: Fp²: Fully in-place functional programming. *Proc. ACM Program. Lang.* **7**(ICFP) (aug 2023). <https://doi.org/10.1145/3607840>