Verification of Data Layout Transformations

Ramon Fernández Mir

with Arthur Charguéraud

Inria

17/09/2018

Motivating example - initial code

```c
typedef struct {
  // Position
  float x, y, z;
  // Other fields
  float vx, vy, vz, c, m, v;
} particle;

particle data[NUM_PARTICLES];

for (int i = 0; i < NUM_PARTICLES; i++) {
  // Some calculation
}
```

Motivating example - splitting

Suppose that the calculation uses mainly the position.

```
typedef struct {
  float vx, vy, vz, c, m, v;
} cold_fields;

typedef struct {
  float x, y, z;
  cold_fields *other;
} particle;

particle data[NUM_PARTICLES];
```

Motivating example - peeling

Further suppose that the intial 'particle' record is not used as part of a dynamic data structure.

```
typedef struct {
  float vx, vy, vz, c, m, v;
} cold_fields;

typedef struct {
  float x, y, z;
} hot_fields;

cold_fields other_data[NUM_PARTICLES];
hot_fields pos_data[NUM_PARTICLES];
```

Motivating example - AoS to SoA

Now, say that we want to take advantage of vector instructions.

```
typedef struct {
  float x[NUM_PARTICLES];
  float y[NUM_PARTICLES];
  float z[NUM_PARTICLES];
} hot_fields;

hot_fields pos_data;
```

Motivating example - AoS to AoSoA

But without reducing too much the locality between accesses to fields of the original struct.

```c
typedef struct {
  float x[N];
  float y[N];
  float z[N];
} hot_fields;

hot_fields pos_data[NUM_PARTICLES / N];
```

Motivating example - summary

In short, the transformations we have seen are:

- Splitting.
- Peeling.
- AoS to SoA.
- AoS to AoSoA.

Note that after all these changes, where we wrote:

```
data[i].x
```

Now we have to write:

```
pos_data[i / N].x[i % N]
```

Project goals

- Formalize a subset of C that captures the essential for these transformations.
- Find the basic transformations that combined give rise to the ones we are interested in.
- Define them and prove their correctness.

Language overview - values and terms

```
Inductive val : Type :=
  (* High-level *)
  | val_error : val
  | val_unit : val
  | val_uninitialized : val
  | val_bool : bool → val
  | val_int : int → val
  | val_double : int → val
  | val_abstract_ptr : loc → accesses → val
  | val_array : typ → list val → val
  | val_struct : typ → map field val → val
  (* Low-level *)
  | val_concrete_ptr : loc → offset → val
  | val_words : list word → val.

Inductive trm : Type :=
  | trm_var : var → trm
  | trm_val : val → trm
  | trm_if : trm → trm → trm → trm
  | trm_let : bind → trm → trm → trm
  | trm_app : prim → list trm → trm
  | trm_while : trm → trm → trm
  | trm_for : var → val → val → trm → trm.
```

Language overview - primitive operations

```
Inductive prim : Type :=
  (* High-level *)
  | prim_binop : binop → prim
  | prim_get : typ → prim
  | prim_set : typ → prim
  | prim_new : typ → prim
  | prim_new_array : typ → prim
  | prim_struct_access : typ → field → prim
  | prim_array_access : typ → prim
  | prim_struct_get : typ → field → prim
  | prim_array_get : typ → prim
  (* Low-level *)
  | prim_ll_get : typ → prim
  | prim_ll_set : typ → prim
  | prim_ll_new : typ → prim
  | prim_ll_access : typ → prim.
```

Comparison of the semantics of our language with C:

```
get p   : *p                          array_access p i  : p + i
set p v : *p = v                      struct_access p f : &(p->f)
new T   : *p = malloc(sizeof(T))      struct_get s f    : s.f
```

Related Work

Structure	Memory	Time	Limitations
Arrays	$1\times$	$1\times$	concat/split/resize: $O(n)$
Vectors	$2-4\times$	$2\times$	concat/split: $O(n)$
Lists	$3\times$	$3\times$	concat/split/random access $O(n)$
Finger trees	$>3\times$	$>3\times$	Not transient
Ropes	?	?	More complex access to ends, not automatically balanced
Chunked Seq	$<1.2\times$	$<2\times$	

Interface

Chunks: fixed capacity arrays in which elements are stored
K = size of chunks

Operation	**Ephemeral**	**Persistent**
push/pop/front/back	$O(1 + \frac{1}{K} \log_K n)$	$O(K + \frac{1}{K} \log_K n)$
usual case	$O(1)$	$O(1)$
concat/split/get/set	$O(K \log_K n)$	$O(K \log_K n)$
iter/fold/...	$O(n)$	$O(n)$
Ephemeral $\rightarrow$Persistent		
destructive	$O(1)$	
nondestructive	$O(K)$	
Persistent $\rightarrow$Ephemeral	$O(K)$	

Tree Structure

[image]

Sequence Representation - persistent

Pchunk:

Fixed capacity persistent sequence

Implemented using a view on a shared “support” chunk

```
type 'a pchunk = {
  support : 'a chunk;
  mutable view : segment; }
```

```
type segment = int * int
```

The shared chunk is reusable when popping or when pushing past its bounds. Other push cases need copy-on-write.

Pops are always O(1), pushes are in amortized O(1) if iterated.

```
type 'a pseq =
| Empty
| Struct of 'a pchunk * ('a pchunk) seq * 'a pchunk
```

Versions

Goal: Work on pchunks with in-place updates in ephemeral sequences

Solution: Maintain whether a chunk is shared or uniquely possessed in ephemeral sequences

Invariants:
Persistent: all chunks are marked false
Ephemeral: some are false and shared, some are true and were created in this sequence

Ephemeral $\rightarrow$persistent = mark all chunks back to false
Version number trick enables this to be done in constant time.

```
type 'a pchunk = {
 version : version;
 support : 'a chunk;
 mutable view : segment; }
```

Transient Sequences - Types

```
type 'a seq = {
  mutable version : version;
  mutable front : 'a chunk;
  mutable middle : ('a pchunk) pseq;
  mutable back : 'a chunk;
}

type 'a pseq =
| Empty of 'a
| Struct of 'a pchunk * ('a pchunk) pseq * 'a pchunk
```

Note: Persistent sequence version number is stored in back chunk.

Summary and Additional Fields

```
type 'a chunk = {
  mutable head : int;
  mutable size : int;
  mutable data : 'a array;
  default : 'a; }
```

```
type 'a pchunk = {
  version : version;
  support : 'a chunk;
  mutable view : segment;
  mutable weight : weight; }
```

```
type 'a seq = {
  mutable version : version;
  mutable front : 'a chunk;
  mutable free_front : ('a chunk) option;
  mutable middle : ('a pchunk) PWSeq.t;
  mutable free_back : ('a chunk) option;
  mutable back : 'a chunk;
}
```

```
type 'a pseq =
    | Empty of 'a
    | Struct of weight * 'a pchunk * ('a pchunk) t * 'a pchunk
```