

# Verification of Data Layout Transformations

**Ramon Fernández Mir**

with Arthur Charguéraud

Inria

17/09/2018

# Motivating example

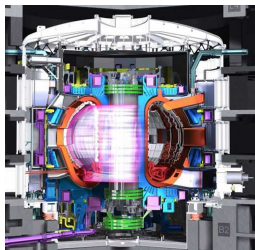


Figure: ITER tokamak

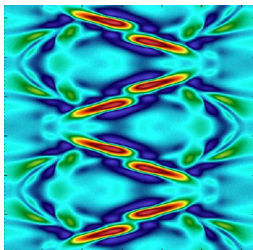


Figure: Plasma physics

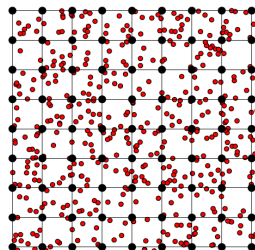


Figure: PIC simulation

## Challenges:

- Exploit data-level parallelism.
- Use domain-specific knowledge of the code.
- Do it without introducing any bugs.

# Motivating example - initial code

```
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 initial '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.

```
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

- Find the basic transformations that combined give rise to the ones we are interested in.
- Formalize a C-like language with arrays, structs and pointers.
  - On a high-level, to simplify the proofs.
  - On a low-level, to be closer to the semantics of C.
- Define the transformations and prove their correctness.

# Basic transformations

## 1. Field grouping

```
// Before
typedef struct {
    int a, b, c;
} s;
```

```
// After
typedef struct {
    int b, c;
} sg;
```

```
typedef struct {
    int a; sg fg;
} s';
```

## 2. Array tiling

```
// Before
int[N] a;
```

```
// After
int[B][N / B] a';
```

## 3. Adding indirection

```
// Before
typedef struct {
    int a, b;
} s;
```

```
// After
typedef struct {
    int a; int *b;
} s';
```

## 4. AoS to SoA

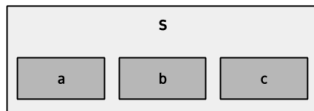
```
// Before
typedef struct {
    int a, b;
} s;
```

```
// After
typedef struct {
    int a[N]; int b[N];
} s;
```

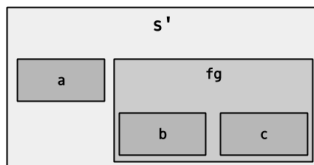
# Basic transformations - grouping

## 1. Field grouping

```
// Before  
typedef struct {  
    int a, b, c;  
} s;
```



```
// After  
typedef struct {  
    int b, c;  
} sg;  
  
typedef struct {  
    int a; sg fg;  
} s';
```



# Basic transformations - tiling

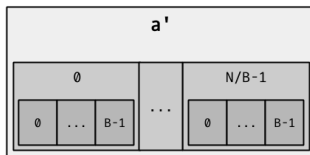
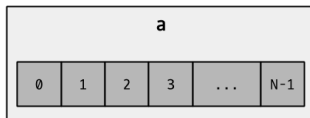
## 2. Array tiling

```
// Before
```

```
int[N] a;
```

```
// After
```

```
int[B][N / B] a';
```

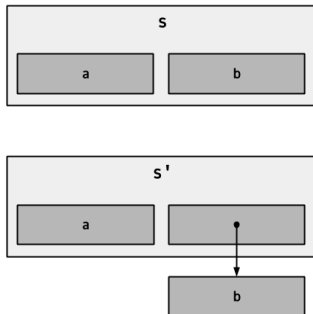


# Basic transformations - indirection

## 3. Adding indirection

```
// Before  
typedef struct {  
    int a, b;  
} s;
```

```
// After  
typedef struct {  
    int a; int *b;  
} s';
```



# Basic transformations - AoS to SoA

## 4. AoS to SoA

```
// Before  
typedef struct {  
    int a, b;  
} s;
```



```
// After  
typedef struct {  
    int a[N]; int b[N];  
} s';
```



# Basic transformations - justification

- **Peeling:** Field grouping twice.
- **Splitting:** Field grouping and then adding indirection on the field holding the group.
- **AoS to SoA:** AoS to SoA.
- **AoS to AoSoA:** Array tiling and then AoS to SoA on the tiles.

# Language overview - values and terms

Inductive val : Type :=

- | 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

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 :=  
| 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
```

Examples of the semantics of our language compared to C:

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

where pointers are represented as pairs:

(l, [access\_field T f, access\_array T' i])

which would correspond to the address:

$l + \text{field\_offset}(f) + i * \text{sizeof}(T')$

# Language overview - semantics

Some crucial definitions:

**Definition**  $\text{typdefctx} := \text{map typvar typ}.$

$\text{Record ll\_typdefctx} := \text{make\_ll\_typdefctx} \{$   
   $\text{typvar\_sizes} \quad : \text{map typvar size};$   
   $\text{fields\_offsets} : \text{map typvar (map field offset)};$   
   $\text{fields\_order} \quad : \text{map typvar (list field)} \}.$

**Definition**  $\text{stack} := \text{Ctx.ctx val}.$

**Definition**  $\text{state} := \text{map loc val}.$

And the relation that defines the big-step reduction rules:

$$\text{red} \subseteq \text{typdefctx} \times \text{ll\_typdefctx} \times \text{stack} \times \text{state} \times \text{trm} \times \text{state} \times \text{val}$$

# Language overview - typing

The allowed types are:

```
Inductive typ : Type :=  
  | typ_unit : typ  
  | typ_int : typ  
  | typ_double : typ  
  | typ_bool : typ  
  | typ_ptr : typ → typ  
  | typ_array : typ → option size → typ  
  | typ_struct : map field typ → typ  
  | typ_var : typvar → typ.
```

With their corresponding definitions (analogous to stack and state):

```
Definition gamma : Ctx.ctx typ.
```

```
Definition phi : map loc typ.
```

Typing is defined as the following relation:

$$\text{typing} \subseteq \text{typdefctx} \times \text{gamma} \times \text{phi} \times \text{trm} \times \text{typ}$$

# Language overview - properties

Need to think of something...

An approximation to type safety:

**Theorem** `type_soundness` :  $\forall C \text{ LLC } m \ t \ v \ T,$   
    `red C LLC nil empty t m v`  $\rightarrow$   
    `typing C nil empty t T`  $\rightarrow$   
     $\exists f, \text{typing\_val } C \ f \ v \ T$   
     $\wedge \text{state\_typing } C \ f \ m.$

# Transformations - grouping

group

# Transformations - tiling

tiling

# Transformations - AoS to SoA

AoS to SoA

# Transformations - proof

statement and proof



# High-to-low level transformation

A few slides on this.

# Project extent

what has been done and what hasn't quite and statistics

# Future work

for instance functions etc, combining them. Code realisations...

# Conclusion

conclusion