

Verification of Data Layout Transformations

Ramon Fernández Mir

with Arthur Charguéraud

Inria

17/09/2018

Motivating example

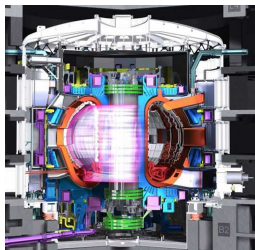


Figure: ITER tokamak

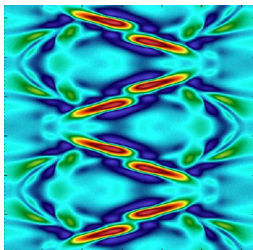


Figure: Plasma physics

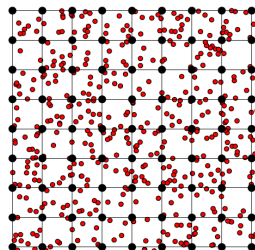


Figure: PIC simulation

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

- 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.
```

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 |

Language overview - typing

Maybe a couple or three slides for this.

Give overview of typing.

Use this theorem to explain the different parts of semantics and typing.

Theorem `type_soundess` : $\forall C \text{ LLC } F \text{ m } t \text{ v } T \text{ G } S \text{ m}',$
 `red C LLC S m t m' v` \rightarrow
 `typing C LLC F G t T` \rightarrow
 `state_typing C LLC F m` \rightarrow
 `stack_typing C LLC F G S` \rightarrow
 $\exists F', \text{ extends } F \text{ } F'$
 $\wedge \text{ typing_val } C \text{ LLC } F' \text{ v } T$
 $\wedge \text{ state_typing } C \text{ LLC } F' \text{ m}'.$

Some language properties.

Transformations - grouping

group

Transformations - tiling

tiling

Transformations - adding indirection

adding indirection

Transformations - AoS to SoA

AoS to SoA

Transformations - summary

Peeling = Splitting = AoS to SoA = AoS to SoA =

Transformations - proof

statement and proof

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