

Verification of Data Layout Transformations

Ramon Fernández Mir

with Arthur Charguéraud

Inria

17/09/2018

Motivating example

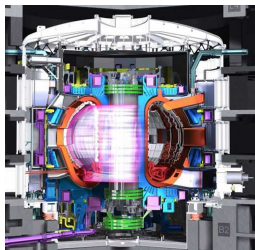


Figure: ITER tokamak

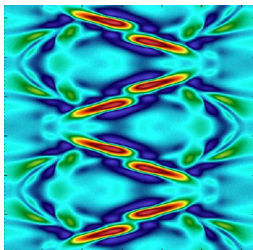


Figure: Plasma physics

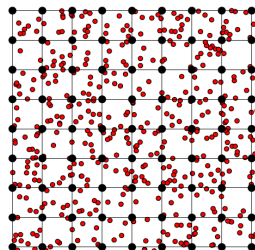


Figure: PIC simulation

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
typedef int a[N];
```

```
// After
typedef int a'[N_/_B][B];
```

3. Adding indirection

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

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

4. AoS to SoA

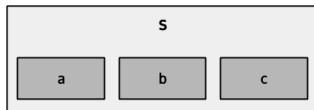
```
// Before
typedef struct {
    int a, b;
} s;
```

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

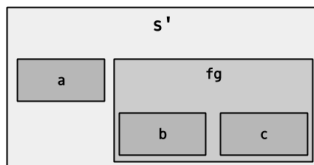
Basic transformations - grouping

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

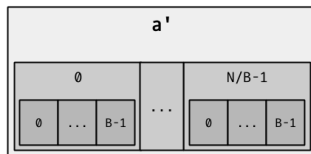
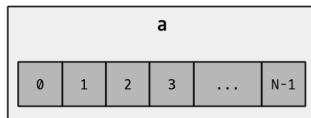
2. Array tiling

// Before

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

// After

```
typedef int a'[N_/_B][_B];
```

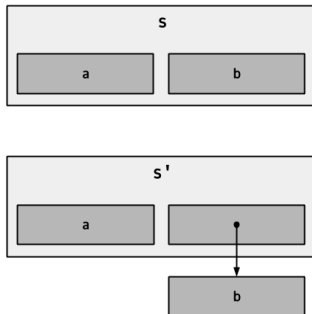


Basic transformations - indirection

3. Adding indirection

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

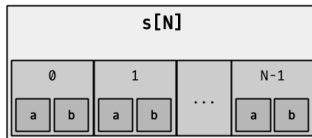
```
// After  
typedef struct {  
    int a; int *b;  
} s';
```



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

For memory accesses, we know the type of the data being manipulated:

```
Inductive typing_val (C:typdefctx) (f:phi) : val → typ → Prop :=  
  | typing_val_abstract_ptr : ∀ l p T,  
    read_phi C f l p T →  
    typing_val C f (val_abstract_ptr l p) (typ_ptr T)
```

```
Inductive typing (C:typdefctx) : gamma → phi → trm → typ → Prop :=  
  | typing_get : ∀ G f T t1,  
    typing C G f t1 (typ_ptr T) →  
    typing C G f (trm_app (prim_get T) (t1::nil)) T
```

Typing result for full execution:

```
Theorem type_soundness : ∀ C LLC m t v T,  
  red C LLC empty_stack empty_state t m v →  
  typing C empty_gamma empty_phi t T →  
  ~is_error v →  
  ∃ f, typing_val C f v T  
  ∧ state_typing C f m.
```

Field grouping

The arguments of the transformation are:

- The struct name `s`.
- The fields `b` and `c`.
- The new struct name `sg`.
- The new field `fg`.

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

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

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

These are used to define a transformation for:

- | | | |
|-----------|-------------|--------------|
| • terms, | • accesses, | • states and |
| • values, | • contexts, | • stacks. |

Field grouping - terms

We start with the transformation of the source code. In particular, we look at the struct access case:

```
Inductive tr_trm (gt:group_tr) : trm → trm → Prop :=
| tr_trm_struct_access_group : ∀fs Tt fg Tg f op0 t1 op2 op1 t1',
  gt = make_group_tr Tt fs Tg fg →
  f ∈ fs →
  (* The access s.f *)
  op0 = prim_struct_access (typ_var Tt) f →
  (* The access s'.fg.f *)
  op1 = prim_struct_access (typ_var Tt) fg →
  op2 = prim_struct_access (typ_var Tg) f →
  tr_trm gt t1 t1' →
  tr_trm gt (trm_app op0 (t1::nil)) (trm_app op2 ((trm_app op1 (t1'::nil))::nil))
```

Field grouping - values

Values need to be changed in the source code. For instance, if we look at the interesting case:

```
Inductive tr_val (gt:group_tr) : val → val → Prop :=
| tr_val_struct_group : ∀Tt Tg s s' fg fs sg,
  gt = make_group_tr Tt fs Tg fg →
  fs ⊆ dom s →
  fg ∉ dom s →
  dom s' = (dom s \ - fs) ∪ {fg} →
  dom sg = fs →
  (* Contents of the grouped fields. *)
  s'[fg] = val_struct (typ_var Tg) sg →
  (∀ f ∈ dom sg, tr_val gt s[f] sg[f]) →
  (* Contents of the rest of the fields. *)
  (∀ f ∈ dom s \ fs, tr_val gt s[f] s'[f]) →
  tr_val gt (val_struct (typ_var Tt) s) (val_struct (typ_var Tt) s')
```

And in the stack and the memory so, from `tr_val`, we naturally define `tr_stack` and `tr_state`.

Field grouping - accesses

For accesses, if we look at the interesting case:

```
Inductive tr_accesses (gt:group_tr) : accesses → accesses → Prop :=
| tr_accesses_field_group : ∀Tt fs fg Tg f a0 p a1 a2 p',
  gt = make_group_tr Tt fs Tg fg →
  f ∈ fs →
  (* The access s.f *)
  a0 = access_field (typ_var Tt) f →
  (* Becomes s'.fg.f *)
  a1 = access_field (typ_var Tt) fg →
  a2 = access_field (typ_var Tg) f →
  tr_accesses gt p p' →
  tr_accesses gt (a0::p) (a1::a2::p')
```

This is used in:

```
Inductive tr_val (gt:group_tr) : val → val → Prop :=
| tr_val_abstract_ptr : ∀l p p',
  tr_accesses gt p p' →
  tr_val gt (val_abstract_ptr l p) (val_abstract_ptr l p')
```


Field grouping - OK

We need a way of checking that the transformation is well-defined.

```
Inductive group_tr_ok : group_tr → typdefctx → Prop :=
| group_tr_ok_intros : ∀Tfs Tt fs fg Tg gt C,
  gt = make_group_tr Tt fs Tg fg →
  Tt ∈ dom C →
  (* The struct Tt can be transformed. *)
  C[Tt] = typ_struct Tfs →
  Tg ∉ dom C →
  fs ⊆ dom Tfs →
  fg ∉ dom Tfs →
  (* Tt doesn't appear anywhere else in the typdefctx. *)
  (∀ Tv,
    Tv ∈ dom C →
    Tv ≠ Tt →
    ~free_typvar C Tt C[Tv]) →
  group_tr_ok gt C.
```

Field grouping - typdefctx

We ‘update’ the type definitions context as follows:

```
Inductive tr_typdefctx (gt:group_tr) : typdefctx → typdefctx → Prop :=  
  | tr_typdefctx_intro : ∀Tfs Tfs' Tfsg Tt fs Tg fg C C',  
    gt = make_group_tr Tt fs Tg fg →  
    dom C' = dom C ∪ {Tg} →  
    (* The original map from fields to types. *)  
    C[Tt] = typ_struct Tfs →  
    (* The map for the new struct and for the grouped fields. *)  
    tr_struct_map gt Tfs Tfs' Tfsg →  
    C'[Tt] = typ_struct Tfs' →  
    C'[Tg] = typ_struct Tfsg →  
    (* The other type variables stay the same. *)  
    (∀ T,  
      T ∈ dom C →  
      T ≠ Tt →  
      C'[T] = C[T]) →  
    tr_typdefctx gt C C'.
```

Field grouping - main theorem

In the end the theorem that we prove is:

Theorem `red_tr`: $\forall \text{gt LLC } C \ C' \ t \ t' \ v \ m2,$
 `red C LLC empty_stack empty_state t m2 v` \rightarrow
 `group_tr_ok gt C` \rightarrow
 `tr_typdefctx gt C C'` \rightarrow
 `tr_trm gt t t'` \rightarrow
 `wf_typdefctx C` \rightarrow
 `wf_trm C t` \rightarrow
 `~is_error v` \rightarrow
 $\exists v' \ m2', \quad \text{tr_val gt } v \ v'$
 $\wedge \text{tr_state gt } m2 \ m2'$
 $\wedge \text{red } C' \text{ LLC } S' \ m1' \ t' \ m2' \ v'.$

Field grouping - induction

To make the proof work we strengthen it as follows:

Theorem `red_tr_ind`: $\forall \text{gt LLC C C' t t' v S S' m1 m1' m2,}$
`red C LLC S m1 t m2 v` \rightarrow
`group_tr_ok gt C` \rightarrow
`tr_typdefctx gt C C'` \rightarrow
`tr_trm gt t t'` \rightarrow
`tr_stack gt S S'` \rightarrow
`tr_state gt m1 m1'` \rightarrow
`wf_typdefctx C` \rightarrow
`wf_trm C t` \rightarrow
`wf_stack C S` \rightarrow
`wf_state C m1` \rightarrow
`~is_error v` \rightarrow
 $\exists v' m2', \quad \text{tr_val gt v } v'$
 $\wedge \quad \text{tr_state gt m2 } m2'$
 $\wedge \quad \text{red C' LLC S' m1' t' m2' v'.$

Array tiling

We need to know:

- The name of the array being changed (Ta).
- The new name for the tiles (Tt).
- The size of the tiles (K).

Similarly, we also define:

- `tiling_tr_ok`,
- `tr_typdefctx`,
- `tr_accesses`,
- `tr_val`,
- `tr_stack`,
- `tr_state` and
- `tr_trm`.

In this case, we change all the instances of `t[i]` to `t[i/K][i%K]` where `t` has type `typ_var Ta`.

Array tiling - some specifics

We use:

- I for the length of the original array,
- J for the length of the array of tiles and
- K for the length of the tile.

These are related by the definitions:

Definition $\text{nb_tiles } (K \ I \ J:\text{int}) : \text{Prop} :=$
 $J = I / K + \text{If } (I \bmod K = 0) \text{ then } 0 \text{ else } 1.$

Definition $\text{tiled_indices } (I \ J \ K \ i \ j \ k:\text{int}) : \text{Prop} :=$
 $i = j * K + k$
 $\wedge \text{index } I \ i$
 $\wedge \text{index } J \ j$
 $\wedge \text{index } K \ k.$

Array tiling - key components

The crucial case of `tr_val` from the array `aI` to `aJ` is captured by:

$$\begin{aligned} \forall i\ j\ k\ aK, \text{ tiled_indices } I\ J\ K\ i\ j\ k \rightarrow \\ aJ[j] = (\text{val_array } (\text{typ_var } Tt)\ aK) \rightarrow \\ \text{tr_val } tt\ aI[i]\ aK[k] \end{aligned}$$

For the translation accesses and primitive operations, the aim is for all the accesses

$$l1 ++ (\text{access_array } Ta\ i)::l2$$

to be transformed to

$$l1 ++ (\text{access_array } Ta\ (i/K))::(\text{access_array } Tt\ (i \bmod K))::l2.$$

AoS to SoA

For this transformation, we need to know:

- The name of the array being changed (T_a).
- The fields names and types of the struct being changed (T_f s).
- The size of the array (K).

This transformation is similar to array tiling in many ways. One key difference is that the accesses

```
l1 ++ (access_array Ta i)::(access_field Tfs[f] f)::l2
```

are transformed to

```
l1 ++ (access_field Ta f)::(access_field (typ_array Tfs[f] K) i)::l2.
```


High-to-low level transformation

The grammar is extended with:

```
Inductive val : Type :=  
  | val_concrete_ptr : loc → offset → val  
  | val_words : list word → val.
```

```
Inductive prim : Type :=  
  | prim_ll_get : typ → prim  
  | prim_ll_set : typ → prim  
  | prim_ll_new : typ → prim  
  | prim_ll_access : typ → prim.
```

There are two sides of this transformation:

- The memory.
- The programs.

High-to-low level transformation - OK

We need to ensure consistency between the type definition context (C) and the low-level context (LLC). In particular:

- The type variable sizes in LLC match with the types in C .
- The field offsets match with the order of the fields and the sizes of each of their types.

High-to-low level transformation - memory

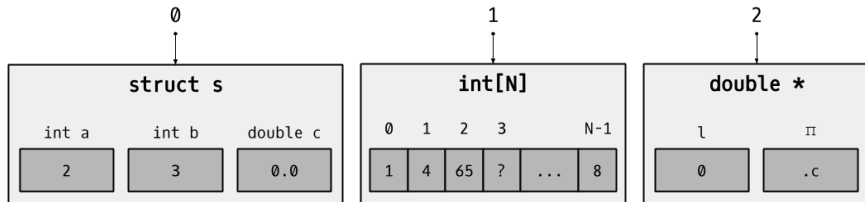


Figure: High-level memory.

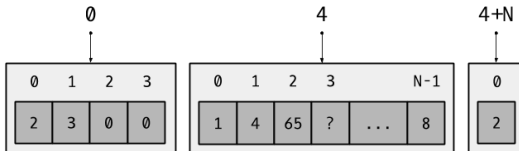


Figure: Low-level memory.

High-to-low level transformation - program

TODO: Details on the transformation of terms.

High-to-low level transformation - proof

TODO: Main theorem.

Project extent

Accomplished goals:

- Defined a high-level language convenient to argue about data-layout transformations.
- Found a way to connect it to realistic low-level semantics.
- Basically proved the correctness of:
 - Field grouping.
 - Array tiling.
 - AoS to SoA.

Some statistics:

lines of spec	lines of proof	lines of comments
2637	2623	606

Future work

Next steps:

- The transformation ‘adding indirection’.
- Realizations of the transformations as functions.
- Some arithmetic results in the tiling and low-level transformations.
- Work on loops and add loop transformations.
- Connect the low-level language with CompCert.

Verification of Data Layout Transformations

Ramon Fernández Mir

with Arthur Charguéraud

Inria

17/09/2018