Verification of Data Layout Transformations

Ramon Fernández Mir

with Arthur Charguéraud

Inria

24/09/2018

Software verification

Why do we care? Take for example GCC.

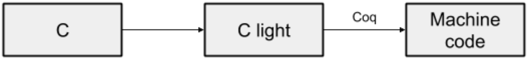
- Between 1999 and 2015, over 39.000 bugs were reported.
- Approximately 60% of the files have some sort of bug.
- The life span of a bug is $\sim$200 days.
- The most buggy file (as of 2015) had 817 different bugs.

Software verification

Why do we care? Take for example GCC.

- Between 1999 and 2015, over 39.000 bugs were reported.
- Approximately 60% of the files have some sort of bug.
- The life span of a bug is ~200 days.
- The most buggy file (as of 2015) had 817 different bugs.

Solution: The CompCert verified compiler.



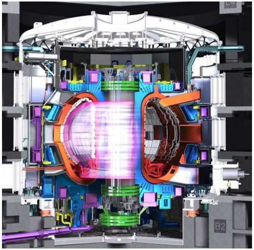
Software verification - principles

Coq provides a formal language to write mathematical definitions and an environment to write machine-checked proofs.

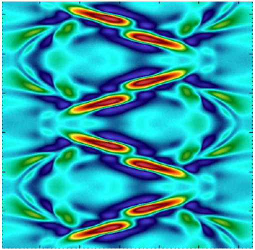
Key ideas:

- Language semantics can be expressed with mathematical rules.
- Language properties can be written as theorems.
- We can prove them!

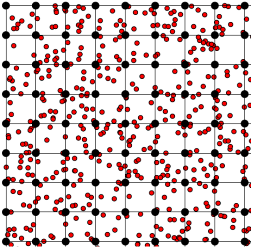
Motivating example



ITER tokamak



Plasma physics



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[N];

for (int i = 0; i < N; i++) {
  // Some calculation involving data[i]
}
```

Motivating example - peeling

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;
} hot_fields;

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

Motivating example - AoS to SoA

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

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

hot_fields pos_data[ceil(N/B)];
```

Motivating example - summary

In short, the transformations we have seen are:

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

Motivating example - summary

In short, the transformations we have seen are:

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

E.g., when applying all these transformations, an access of the form:

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

becomes:

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

Project goals

- Find the basic transformations that combined give rise to the ones we are interested in.

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.
  - Equipped with a high-level semantics, to simplify the proofs.
  - Equipped with a low-level semantics, to be closer to C.

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.
  - Equipped with a high-level semantics, to simplify the proofs.
  - Equipped with a low-level semantics, to be closer to C.
- Define the transformations and prove their correctness.

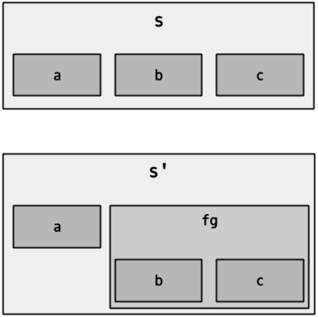
Basic transformations - grouping

1. Field grouping

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

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

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

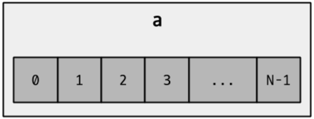


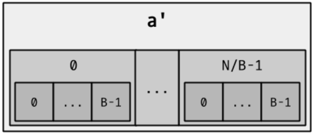
Basic transformations - tiling

2. Array tiling

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

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



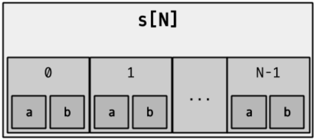


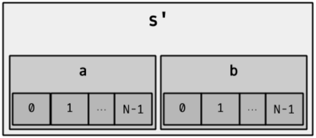
Basic transformations - AoS to SoA

3. 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.
- **AoS to SoA:** AoS to SoA.
- **AoS to AoSoA:** Array tiling and then AoS to SoA on the tiles.

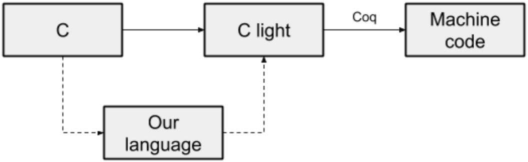
Language overview

The language includes:

- Pointers, structs and arrays.
- All the necessary memory operations:

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

In the big picture:



Language overview - rules

For example, the semantics of get is:

$$\frac{\langle C,\ S,\ m_1,\ t\rangle \Downarrow \langle m_2,\ (l,\pi)\rangle \qquad m_2[l]..\pi\ =\ v_r \qquad v_r \neq \varnothing}{\langle C,\ S,\ m_1,\ get_T\ t\rangle \Downarrow \langle m_2,\ v_r\rangle}$$

In Coq, this looks like:

```
Inductive red (C:typdefctx) : stack → state → trm → state → val → Prop :=
  | red_get : ∀l p S T v1 m1 m2 vr,
      red C S m1 t m2 (val_abstract_ptr l p) →
      read_state m2 l p vr →
      ~is_uninitialized vr →
      red C S m1 (trm_app (prim_get T) (t::nil)) m2 vr.
```

Field grouping - rules

Similarly, we define rules for our transformation:

$$\pi := \varnothing \mid [i] :: \pi \mid .f :: \pi$$

$$\begin{aligned}
\llbracket \varnothing \rrbracket &= \varnothing \\
\llbracket [i] :: \pi \rrbracket &= [i] :: \llbracket \pi \rrbracket \\
\llbracket\, .f :: \pi \rrbracket &= .f :: \llbracket \pi \rrbracket && \text{when } f \notin Fs \\
\llbracket\, .f :: \pi \rrbracket &= .f_g :: .f :: \llbracket \pi \rrbracket && \text{when } f \in Fs
\end{aligned}$$

Field grouping - Coq

In Coq, this looks like:

```
Inductive tr_accesses (gt:group_tr) : accesses → accesses → Prop :=
  | tr_accesses_nil :
      tr_accesses gt nil nil
  | tr_accesses_array : ∀p p' T i,
      tr_accesses gt p p' →
      tr_accesses gt (access_array T i::p) (access_array T i::p')
  | tr_accesses_field_other : ∀T Tt Fs Tg fg p p' f,
      gt = make_group_tr Tt Fs Tg fg →
      tr_accesses gt p p' →
      T ≠Tt ∨f ∉Fs →
      tr_accesses gt (access_field T f::p) (access_field T f::p').
  | tr_accesses_field_group : ∀Tt Fs Tg fg p p' f,
      gt = make_group_tr Tt Fs Tg fg →
      tr_accesses gt p p' →
      f ∈Fs →
      tr_accesses gt (access_field Tt f::p) (access_field Tt fg::access_field Tg f::p')
```

Field grouping - simulation

With a similar pattern we define:

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

The property that we require from the transformation is:

$$\begin{tikzcd}
t \arrow[r, "tr"] \arrow[d, Rightarrow] & \llbracket t \rrbracket \arrow[d, Rightarrow, dashed] \\
v \arrow[r, dashed, "tr"] & \llbracket v \rrbracket
\end{tikzcd}$$

Field grouping - theorem

In the end the theorem that we prove for full executions is:

```
Theorem red_tr: ∀gt C C' t t' v m,
  red C empty_stack empty_state t m v →
  ~is_error v →
  group_tr_ok gt C →
  tr_typdefctx gt C C' →
  tr_trm gt t t' →
  wf_typdefctx C →
  wf_trm C t →
  ∃v' m', tr_val gt v v'
       ∧ tr_state gt m m'
       ∧ red C' empty_stack empty_state t' m' v'.
```

Field grouping - induction

To make the proof work we strengthen it as follows:

```
Theorem red_tr_ind: ∀gt C C' t t' v S S' m1 m1' m2,
  red C S m1 t m2 v →
  ~is_error v →
  group_tr_ok gt C →
  tr_typdefctx gt C C' →
  tr_trm gt t t' →
  tr_stack gt S S' →
  tr_state gt m1 m1' →
  wf_typdefctx C →
  wf_trm C t →
  wf_stack C S →
  wf_state C m1 →
  ∃v' m2',    tr_val gt v v'
           ∧  tr_state gt m2 m2'
           ∧  red C' S' m1' t' m2' v'.
```

Demo

Array tiling and AoS to SoA

Array tiling

- Takes as arguments:
  - The name of the array being changed (`Ta`).
  - The name of the tiles (`Tt`).
  - The size of the tiles (`K`).
- All the instances of `t[i]` where `t` has type `Ta` become `t[i/K][i%K]`.

AoS to SoA

- Takes as arguments:
  - The name of the array being changed (`Ta`).
  - The fields names and types of the struct being changed (`Tfs`).
  - The size of the array (`K`).
- All the instances of `t[i].f` where `t` has type `Ta` become `t.f[i]`.

High-level transformations - summary

So far we have presented:

- Field grouping.
- Array tiling.
- AoS to SoA.

High-level transformations - summary

So far we have presented:

- Field grouping.
- Array tiling.
- AoS to SoA.

The correctness of these is proved!
(up to a couple axioms, e.g., results on the modulo operation)

High-level transformations - summary

So far we have presented:

- Field grouping.
- Array tiling.
- AoS to SoA.

The correctness of these is proved!
(up to a couple axioms, e.g., results on the modulo operation)

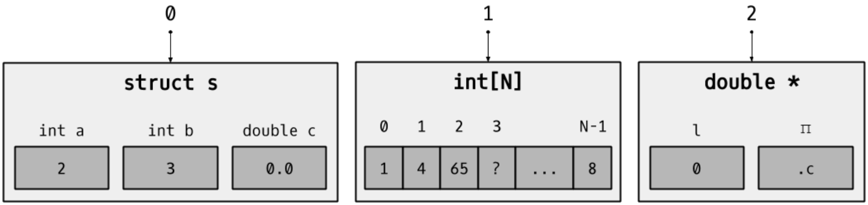
Problem: This might all be just a hack if we don't link it with a more concrete, CompCert-style semantics...

High-level to low-level transformation

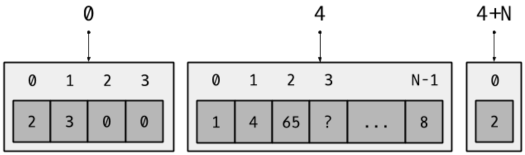
The grammar is extended with:

- Low-level pointers.
  `(l, p) => (l, offset(p))`
- Low-level heap operations.
  `struct_access (l, p) f => struct_ll_access (l, offset(p)) field_offset(f)`
- A special kind of value that consists of a list of words.

High-level to low-level transformation - memory



High-level memory.



Low-level memory.

High-level to low-level transformation - theorem

The goal is to prove:

```
Theorem red_tr_warmup : ∀C LLC T m a v t' m' v',
  red C LLC empty_stack empty_state t m v →
  typing C empty_gamma empty_phi t T →
  ~is_error v →
  ll_typdefctx_ok C LLC →
  tr_trm C LLC a t t' →
  wf_typdefctx C →
  wf_trm C t →
  wf_typ C T →
  ∃v' m',  tr_state C LLC a m m'
        ∧ tr_val C LLC a v v'
        ∧ red C LLC empty_stack empty_state t' m' v'.
```

Project extent

Accomplished goals:

- Defined a high-level language convenient to argue about data-layout transformations.

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.

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.
- Proved the correctness of:
  - Field grouping.
  - Array tiling.
  - AoS to SoA.

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.
- Proved the correctness of:
  - Field grouping.
  - Array tiling.
  - AoS to SoA.

Some statistics:

lines of spec	lines of proof	lines of comments
2721	3113	707

Future work

Next steps:

- 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 (at which level?)

Thanks!