Verification of Data Layout Transformations

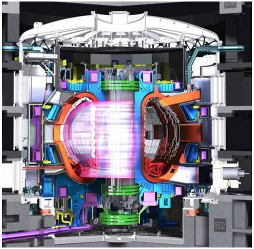
Ramon Fernández Mir

with Arthur Charguéraud

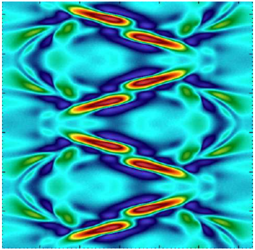
Inria

17/09/2018

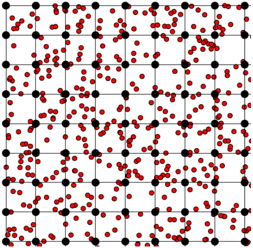
Motivating example



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 - 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[N];
```

Motivating example - peeling

Typically, cold fields are stored in a different array.

```
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:

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

Motivating example - summary

In short, the transformations we have seen are:

- Splitting.
- 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 – overview

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. AoS to SoA

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

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

4. Adding indirection

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

// After
typedef struct {
  int a; T *b;
} s';
```

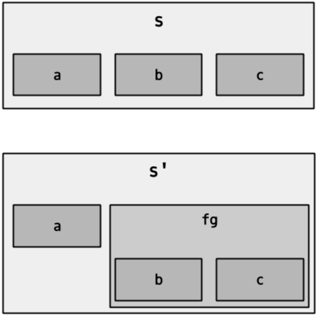
Basic transformations - grouping

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';
```

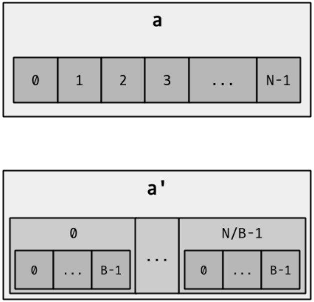


Basic transformations - tiling

2. Array tiling

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

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

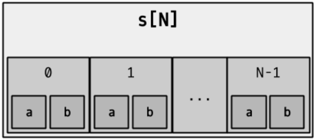


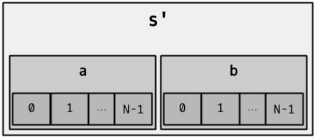
Basic transformations - AoS to SoA

3. AoS to SoA

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

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



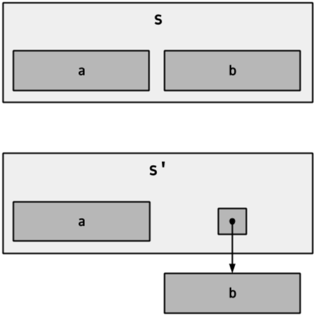


Basic transformations - indirection

4. Adding indirection

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

// After
typedef struct {
  int a; T *b;
} s';
```



Basic transformations - justification

- **Splitting:** Field grouping and then adding indirection on the field holding the group.
- **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 - 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 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
```

where pointers are represented as pairs:

```
(l, (access_field T f)::(access_array T' i)::nil)
```

which would correspond to the address:

```
l + field_offset(f) + i * sizeof(T')
```

Language overview - semantics

Some crucial definitions:

```
Definition typdefctx := map typvar typ.

Definition stack := Ctx.ctx val.

Definition state := map loc val.
```

Language overview - semantics

Some crucial definitions:

```
Definition typdefctx := map typvar typ.

Definition stack := Ctx.ctx val.

Definition state := map loc val.
```

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

```
Inductive red : typdefctx → stack → state → trm → state → val → Prop
```

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.
```

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.
```

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:

```
Inductive typing : typdefctx → gamma → phi → trm → typ → Prop
```

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 m t v T,
  red C 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 `Ts`.
- The fields `b` and `c` (`fs`).
- The new struct name `Tg`.
- The new field `fg`.

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

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

typedef struct {
  int a; Tg fg;
} Ts;
```

Field grouping

The arguments of the transformation are:

- The struct name `Ts`.
- The fields `b` and `c` (`fs`).
- The new struct name `Tg`.
- The new field `fg`.

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

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

typedef struct {
  int a; Tg fg;
} Ts;
```

These are used to define a transformation for:

- terms,
- values,
- accesses,
- contexts,
- states and
- 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 Ts fg Tg f op0 t1 op2 op1 t1',
      gt = make_group_tr Ts fs Tg fg →
      f ∈ fs →
      (* The access s.f *)
      op0 = prim_struct_access (typ_var Ts) f →
      (* The access s'.fg.f *)
      op1 = prim_struct_access (typ_var Ts) 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 : ∀Ts Tg s s' fg fs sg,
      gt = make_group_tr Ts 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 Ts) s) (val_struct (typ_var Ts) 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 : ∀Ts fs fg Tg f a0 p a1 a2 p',
      gt = make_group_tr Ts fs Tg fg →
      f ∈ fs →
      (* The access s.f *)
      a0 = access_field (typ_var Ts) f →
      (* Becomes s'.fg.f *)
      a1 = access_field (typ_var Ts) 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 - typdefctx

We ‘update’ the type definitions context as follows:

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

Field grouping - sanity checks

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

Field grouping - sanity checks

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

Regardless of `group_tr`, we need to check that everything is well-formed:

- The `typdefctx` is well-formed if the type definitions are productive.

Field grouping - sanity checks

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

Regardless of `group_tr`, we need to check that everything is well-formed:

- The `typdefctx` is well-formed if the type definitions are productive.
- Terms, values, stacks and states are well-formed if all the types that appear in them exist.

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'.
```

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`).

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 nb_tiles (K I J:int) : Prop :=
  J = I / K + If (I mod K = 0) then 0 else 1.

Definition tiled_indices (I J K i j k:int) : Prop :=
     i = j * K + k
  ∧  index I i
  ∧  index J j
  ∧  index K k.
```

Array tiling - key components

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

$$\forall \texttt{i j k aK},\quad \begin{array}{l}\texttt{tiled\_indices I J K i j k} \rightarrow \\ \texttt{aJ[j]} = (\texttt{val\_array (typ\_var Tt) aK}) \rightarrow \\ \texttt{tr\_val tt aI[i] aK[k]}\end{array}$$

Array tiling - key components

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

$$\forall \texttt{i j k aK}, \quad \begin{array}{l} \texttt{tiled\_indices I J K i j k} \rightarrow \\ \texttt{aJ[j]} = (\texttt{val\_array (typ\_var Tt) aK}) \rightarrow \\ \texttt{tr\_val tt aI[i] aK[k]} \end{array}$$

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

$\texttt{l1} +\!+ (\texttt{access\_array (typ\_var Ta) i})::\texttt{l2}$

to be transformed to:

$\texttt{l1} +\!+ (\texttt{access\_array (typ\_var Ta) (i/K)})::(\texttt{access\_array (typ\_var Tt) (i mod K)})::\texttt{l2}.$

AoS to SoA

For this transformation, we need to know:

- 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`).

AoS to SoA

For this transformation, we need to know:

- 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`).

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

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

are transformed to:

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

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:

```
Inductive val : Type :=
  | val_concrete_ptr : loc → 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.
```

High-level to low-level transformation

The grammar is extended with:

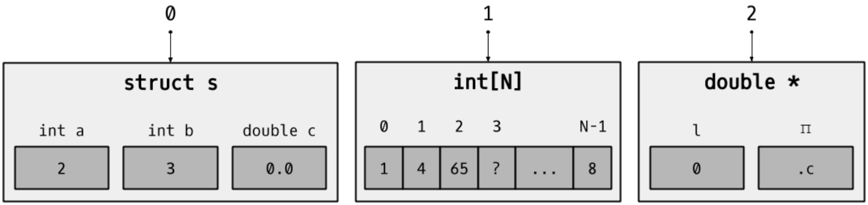
```
Inductive val : Type :=
  | val_concrete_ptr : loc → val
  | val_words : list word → val.

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

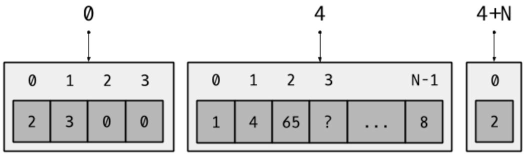
There are two sides of this transformation:

- The memory.
- The programs.

High-level to low-level transformation - memory



High-level memory.



Low-level memory.

High-level to low-level transformation - program

The values in the source code are all kept the same except for pointers:

```
Inductive tr_val (C:typdefctx) (LLC:ll_typdefctx) (a:alpha) : val → val → Prop :=
  | tr_val_abstract_ptr : ∀p l o,
      tr_ll_accesses C LLC p o →
       tr_val C LLC a (val_abstract_ptr l p) (val_concrete_ptr (a[l] + o)).
```

High-level to low-level transformation - program

The values in the source code are all kept the same except for pointers:

```
Inductive tr_val (C:typdefctx) (LLC:ll_typdefctx) (a:alpha) : val → val → Prop :=
  | tr_val_abstract_ptr : ∀p l o,
      tr_ll_accesses C LLC p o →
       tr_val C LLC a (val_abstract_ptr l p) (val_concrete_ptr (a[l] + o)).
```

For terms, as an example, a term:

```
trm_app (prim_struct_access T f) (t::nil)
```

gets translated to:

```
trm_app (prim_ll_access T[f]) (t'::(field_offset T f)::nil).
```

High-level to low-level transformation - program

The values in the source code are all kept the same except for pointers:

```
Inductive tr_val (C:typdefctx) (LLC:ll_typdefctx) (a:alpha) : val → val → Prop :=
  | tr_val_abstract_ptr : ∀p l o,
      tr_ll_accesses C LLC p o →
       tr_val C LLC a (val_abstract_ptr l p) (val_concrete_ptr (a[l] + o)).
```

For terms, as an example, a term:

```
trm_app (prim_struct_access T f) (t::nil)
```

gets translated to:

```
trm_app (prim_ll_access T[f]) (t'::(field_offset T f)::nil).
```

The semantics of `prim_ll_access` is, in fact, that of addition.

High-level to low-level transformation - LLC

The low-level context is defined as follows:

```
Record ll_typdefctx := make_ll_typdefctx {
  typvar_sizes   : map typvar size;
  fields_offsets : map typvar (map field offset);
  fields_order   : map typvar (list field) }.
```

We need to ensure coherency 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-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.
- Basically 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.
- Basically proved the correctness of:
  - Field grouping.
  - Array tiling.
  - AoS to SoA.

Some statistics:

lines of spec	lines of proof	lines of comments
2723	3103	668

Future work

Next steps:

- Formalization of 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 (at which level?)

Thanks!