Through the Steps of Programmable Refactoring of a Large Scientific Code

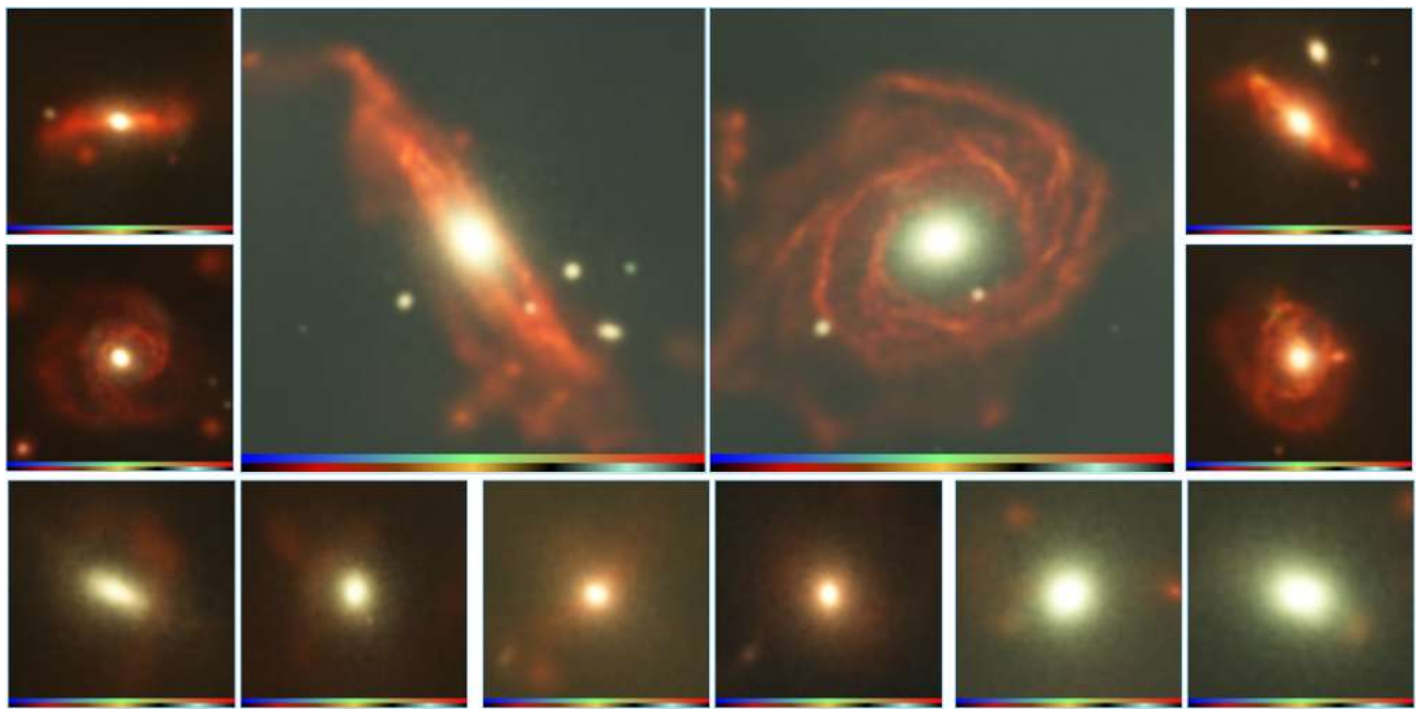
Michele MARTONE

Leibniz Supercomputing Centre
Garching bei München, Germany

Università degli studi di Verona
November 25, 2019

The GADGET simulation code[1]

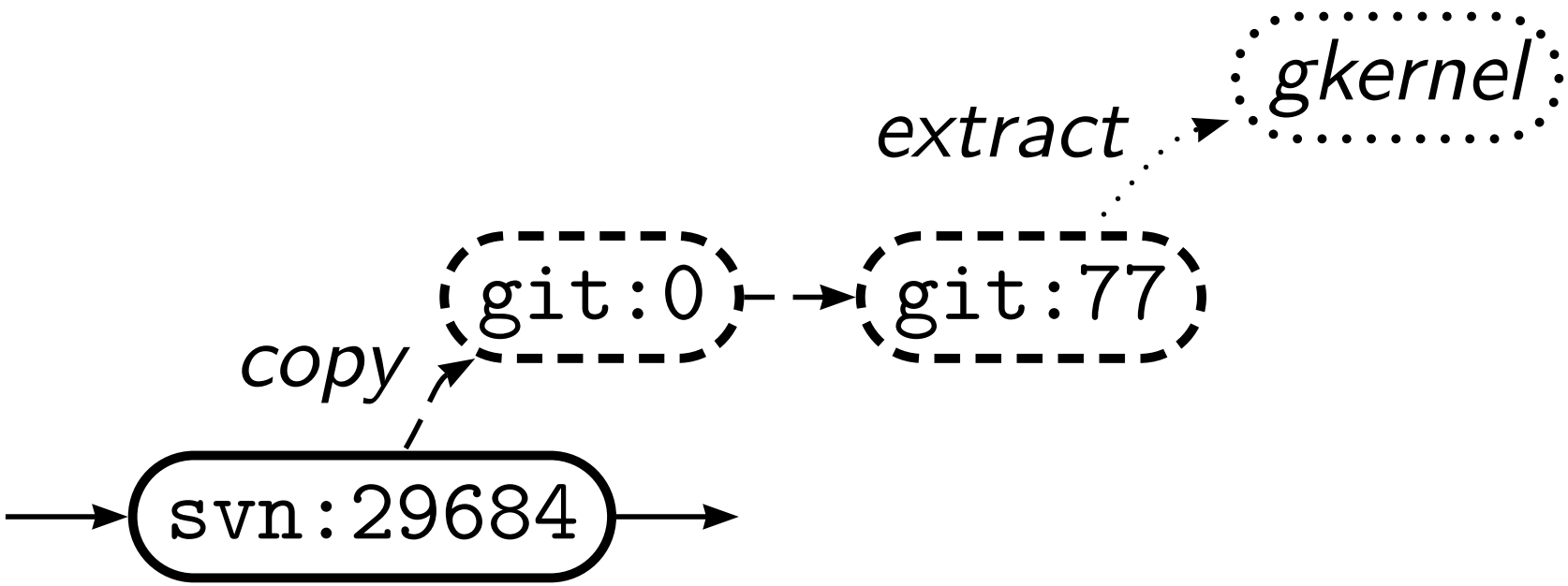
- Cosmological large-scale structure formation (galaxies and clusters)
- Highly scalable ($O(100k)$ Xeon cores on SuperMUC@LRZ)
- Several teams and versions (>100 kLoC each)



[1] V. Springel, "The cosmological simulation code GADGET-2", MNRAS, vol. 364, pp. 1105–1134, 2005

Performance pilot study at LRZ

- L.Iapichino, V. Karakasis, F.Baruffa, N.Hammer
- spanned over one year
- focused on 1kLoC extract
- identified changes meant for whole GADGET



Speedup requires data layout change [2]

main header:

```
1 struct particle {
2   double  Mass, ...
3 #if defined(BLACK_HOLES)...
4   double  Hsml, ...
5   ...
6 };
```

⇒

```
1 struct particle_soa_t {
2   double *Mass, ...
3 #if defined(BLACK_HOLES)...
4   double *Hsml, ...
5   ...
6 };
```

init source file:

```
1 // Array of Structures:
2 struct particle *P;
3 // allocate one global array:
4 P = mymalloc(...
5
6
7
8
```

⇒

```
1 // Structure of Arrays
2 struct particle_soa_t P_SoA;
3 // allocate one global array
4 // for each field:
5 P_SoA.Mass = mymalloc(...
6 #if defined(BLACK_HOLES)...
7 P_SoA.Hsml = mymalloc(...
8 ...
```

favour auto-vectorization in *.c:

```
1 ...
2 // may not vectorize
3 P[i].Mass + P[i]...
```

⇒

```
1 ...
2 // vectorizes better
3 P_SoA.Mass[i] + P_SoA...
```

[2] F.Baruffa et al. https://arxiv.org/abs/1612.06090

Speedup *suggests* major code change steps

1. each type, $10 \approx 100$ fields, `#ifdefs`:

```
1 struct particle {
2   double  Mass, ...
3 #if defined(BLACK_HOLES)...
4   double  Hsml, ...
5   ...
6 };
```

$\Rightarrow$

```
1 struct particle_soa_t {
2   double *Mass, ...
3 #if defined(BLACK_HOLES)...
4   double *Hsml, ...
5   ...
6 };
```

2. (almost) each field an allocation:

```
1 // Array of Structures:
2 struct particle *P;
3 // allocate one global array:
4 P = mymalloc(...
5
6
7
8
```

$\Rightarrow$

```
1 // Structure of Arrays
2 struct particle_soa_t P_SoA;
3 // allocate one global array
4 // for each field:
5 P_SoA.Mass = mymalloc(...
6 #if defined(BLACK_HOLES)...
7 P_SoA.Hsml = mymalloc(...
8 ...
```

3. $\gg$10KLoC change in `*.c`!

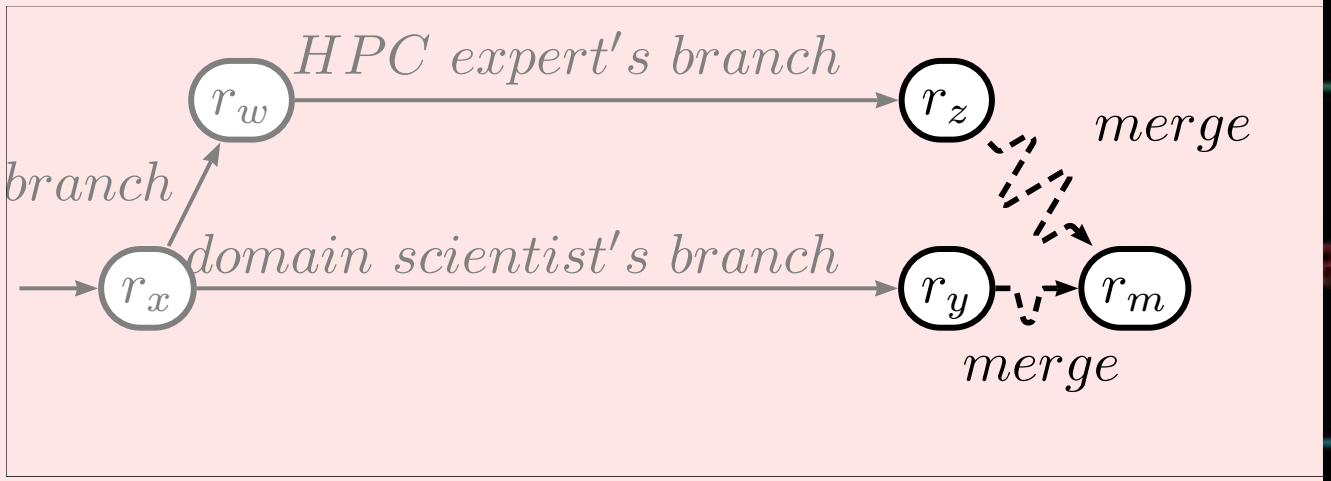
```
1 ...
2 // may not vectorize
3 P[i].Mass + P[i]...
```

$\Rightarrow$

```
1 ...
2 // vectorizes better
3 P_SoA.Mass[i] + P_SoA...
```

$\gg$10KLoC *diff* =⚠

- ~~by hand~~ ?
- custom script?
- and ...all-at-once?



A C code matching and transformation engine

- A project from Inria (France)

[2] https://git.kernel.org/pub/scm/linux/kernel/git/backports/backports.git/tree/patches

[3] https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/scripts/coccinelle

A C code matching and transformation engine

- A project from Inria (France)
- originally to
  - update Linux kernel drivers[2]
  - **smash bugs** (hence the name)[3]

[2] https://git.kernel.org/pub/scm/linux/kernel/git/backports/backports.git/tree/patches

[3] https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/scripts/coccinelle

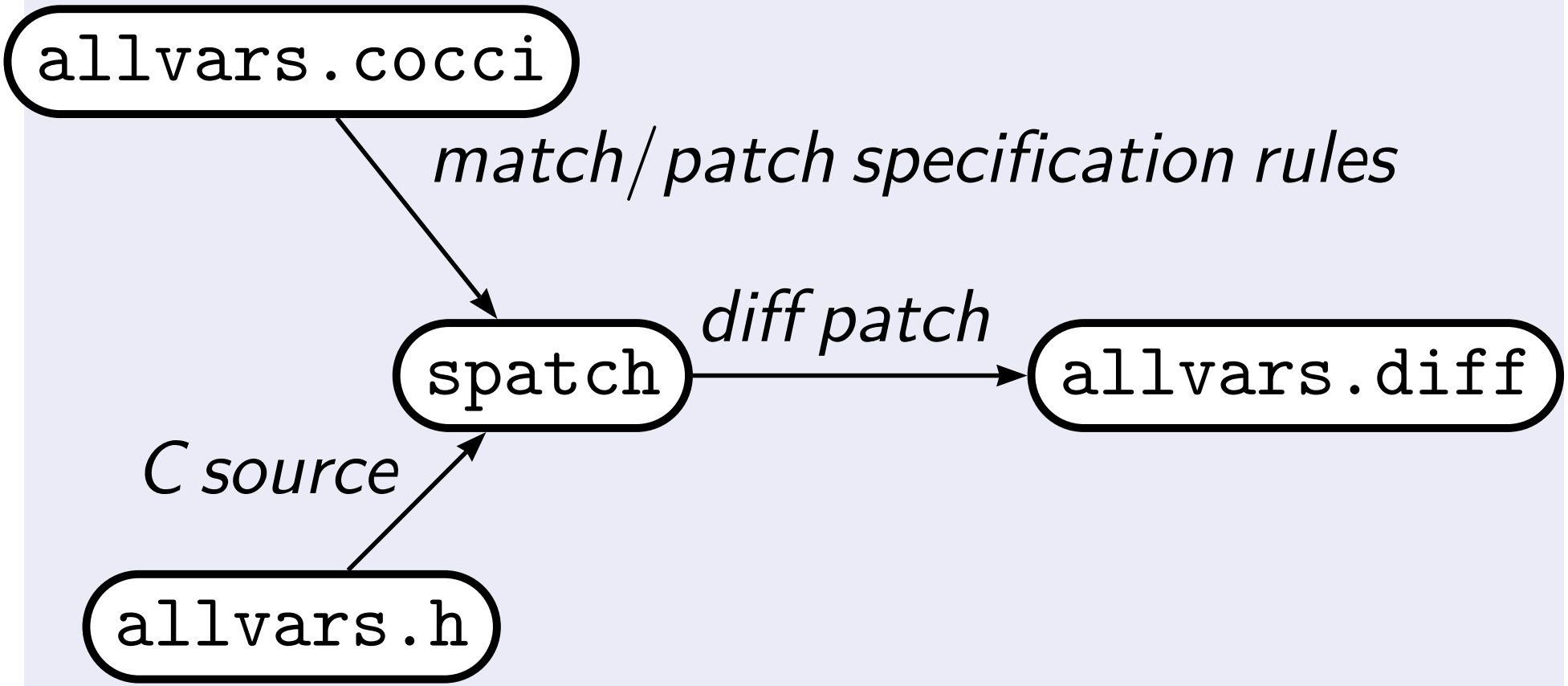
A C code matching and transformation engine

- A project from Inria (France)
- originally to
  - update Linux kernel drivers[2]
  - **smash bugs** (hence the name)[3]
- seemingly **underutilized** in other contexts

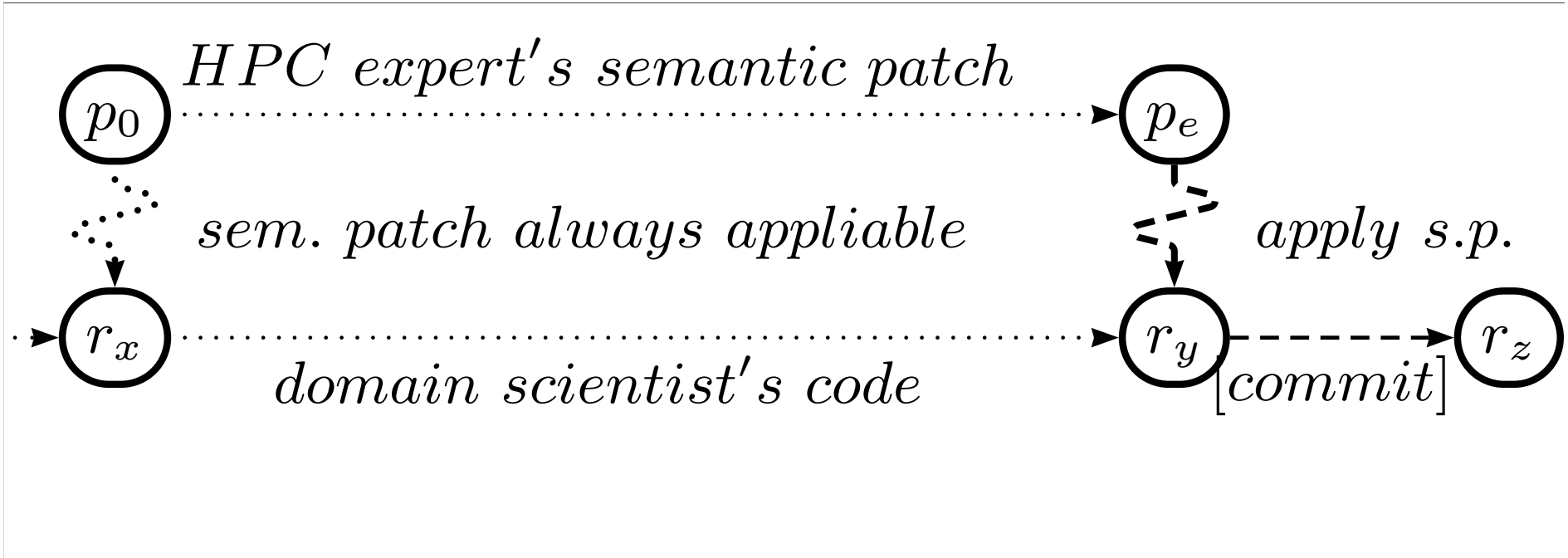
[2] https://git.kernel.org/pub/scm/linux/kernel/git/backports/backports.git/tree/patches

[3] https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/scripts/coccinelle

Patch what and how ?

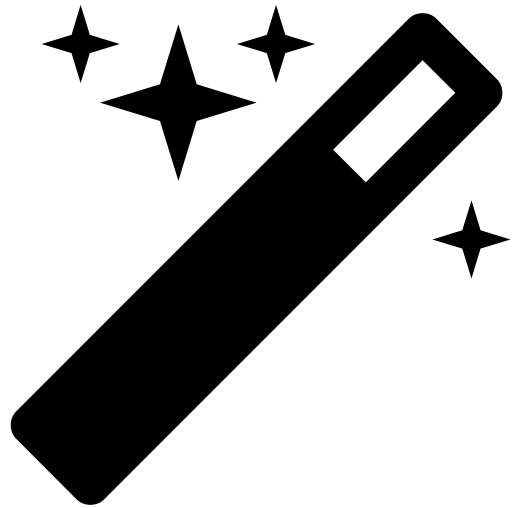


Transformations *encoded* in *semantic patches*



- produces usable patched (e.g. refactored) code
- no requirement to commit patched code

Step 1:
modify data structures into SoA



Match main data structure

```
@prtcl_str@
identifier id = {particle_data,sph_particle_data};
field list fs;
identifier I;
declaration d;
type ST;
@@

(
 struct id { fs } *I;
&
 ST        { fs } *I;
&
 d
)
```

- pattern language to *match* C
- *metavariables* `id`, `I`, ... to match C language elements

Create new identifier

```
1 @script:python new_prtcl_str_id@
2 id << prtcl_str.id;
3 id1;
4 @@
5
6 coccinelle.id1="%s_soa_t"%(id)
```

- reuse `identifier` `id` from rule `prtcl_str`
- new string using Python

Clone main data structure

```
1 @insert_new_prtcl_str depends on prtcl_str@
2 identifier new_prtcl_str_id.id1;
3 field list prtcl_str.fs;
4 type T;
5 @@
6
7 extern int maxThreads;
8 ++struct id1 { fs };
```

- pattern language to *patch* C
- reminiscent of `GNU patch`

Filter out unwanted fields

```
1  @match_anon_union_in_struct@
2  identifier id;
3  identifier J;
4  field list[n] fs;
5  identifier new_prtcl_str_id.id1;
6  @@
7
8  struct id1 { fs
9   union { ... } J;
10    ...
11 };
12
13 @rm_union_from_struct depends on match_anon_union_in_struct@
14 field list[match_anon_union_in_struct.n] fs;
15 identifier new_prtcl_str_id.id1;
16 field fld;
17 @@
18
19 struct id1 { fs
20 - fld
21   ...
22 };
```

Whitelisted types to pointers ♻

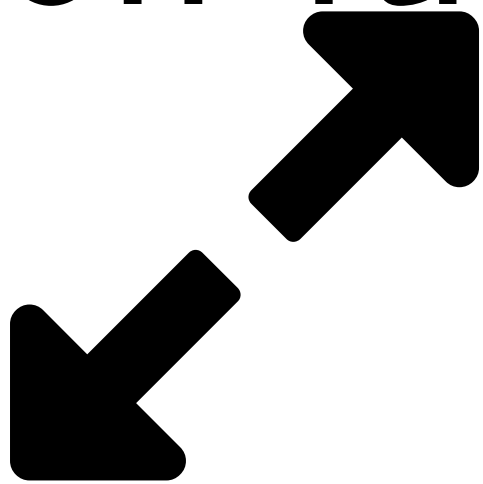
```
@make_ptr@
identifier new_prtcl_str_id.id1;
identifier M;
typedef MyDouble;
typedef MyFloat;
typedef MyLongDouble;
typedef MyDoublePos;
typedef MyBigFloat;
type MT = { double,float,MyDouble,MyFloat,MyLongDouble,
    MyDoublePos,MyBigFloat};


@@

struct id1 { ...
- MT  M;
+ MT *M;
  ...
};
```

Remove non-pointer fields ✖

```
1  @del_non_ptr@
2  identifier new_prtcl_str_id.id1;
3  identifier J;
4  type T;
5  type P != {T*};
6  @@
7
8  struct id1 { ...
9  - P J;
10   ...
11 };
```

- *for each field* `J` *of type* `P` *which is not a pointer to another type...*

Step 2:
allocation functions

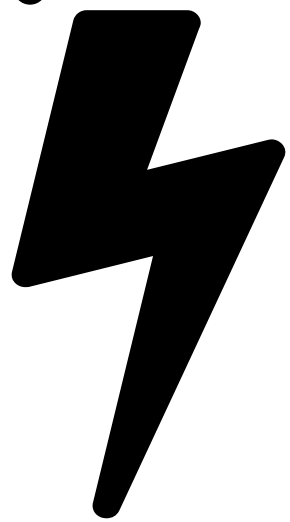


Populate allocation functions ⤢

```
1  @per_type_soa_alloc@
2  identifier new_prtcl_str_id.id1;
3  identifier prtcl_str_mmbrs.M;
4  type prtcl_str_mmbrs.MT;
5  symbol P;
6  identifier insert_per_type_soa_functions.soa_alloc_fid;
7  fresh identifier si = "#ifdef "##"HAVE__"##id1##"__"##M##" //";
8
9  identifier str_from_id.str;
10 @@
11 void soa_alloc_fid(...)
12 { ...
13 ++si;
14 ++       P->M = (MT*) mymalloc(str, sizeof(*(P->M)) * N);
15 ++       if(!P->M)soa_abort(/*"allocating "*/ str);
16 ++#endif
17 }
```

Step 3:

AoS → SoA



Match old structure defs[4]

```
@ostr@
identifier id = {particle_data,sph_particle_data};
type ST;
identifier P;
@@

(
 struct id { ... } *P;
&
 ST { ... } *P;
)
```

[4] Subset of rule `prtcl_str` from p. 10.

Match fields of new types[5] 📌

```
@nt@
identifier pps.id1;
identifier I;
type T;
@@

struct  id1 {
  ...
 T I;
  ...
};
```

[5] Relies on rule pps, duplicate of new_prtcl_str_id, p. 11.

Mold new identifiers[6]

```
@script:python pid@
id1 << pps.id1;
P << ostr.P;
S;
@@

coccinelle.S="%s_soa"%(P)
```

[6] Same mechanism as `new_prtcl_str_id` at p. 11.

Patch many expressions (≫10KLoC diff) 

```
@soa_access@
identifier ostr.P;
identifier pid.S;
identifier nt.I;
expression E;
@@

- P[E].I
+ S.I[E]
```

∀ `identifier` `P` previously matched in rule `ostr`
∀ `identifier` `S` previously matched in rule `pid`
∀ `identifier` `I` previously matched in rule `nt`
∀ `expression` `E` matching rule `soa_access`

substitute `P[E].I` with `S.I[E]`

Outcome ✔

- GADGET can evolve further
- its semantic patches...
  - ...can be stored 💾
  - ...and applied at anytime ▶
  - ...ease performance experiments 🧪
  - ...serve also as a Coccinelle test[7] 🔄

Thanks to Dr. Julia Lawall (Inria) for successful collaboration!

[7] https://github.com/coccinelle/coccinelle/commit/ad5a94

One-day training: 24.03.2020 at LRZ

Introduction to Semantic Patching of C programs with Coccinelle

Register online

https://www.lrz.de/services/compute/courses/2020-03-24_hspc2w19/