Towards thread-safe GAP library

Alexander Konovalov
Centre of Interdisciplinary Research in Computational Algebra
University of St Andrews

2nd GAP Days
Aachen, 16-20 March 2015

Frequently asked question

- **Q**: If I will construct a group in a thread, will it be thread-local?
- **A**: No. It is an atomic component object, so it will be located in the public region

```
gap> G:=SymmetricGroup(3);
Sym( [ 1 .. 3 ] )
gap> RegionOf(G);
<region: public region>
gap> t:=RunTask(
> function() G:=SymmetricGroup(4);end);;
gap> G;
Sym( [ 1 .. 4 ] )
gap> RegionOf(G);
<region: public region>
```

```
gap> RegionOf([1,2,3]);
<region: thread region #0>
gap> RegionOf(rec(a:=1));
<region: thread region #0>
```

```
gap> PARAM:=1;;
gap> t:=RunTask(
> function() PARAM:=2;end);;
gap> PARAM;
2
```

- Lists and records will be thread-local
- OTOH, immutable global variables may be changed by any thread

Exercise

- List all global variables that wrongly stay thread-local:
- `Filtered( NamesSystemGVars(),`
  `x -> IsBoundGlobal(x) and IsThreadLocal(ValueGlobal(x) ) );`
- E.g. `fam!.Zcache` in `ffecoway.gi` keeps known primitive elements of finite fields of characteristic *p*
  - Discoverable only at runtime and only when accessed by another thread
  - `!.` syntax is used in ~5K lines in the GAP library and in 23K+ lines in 600+ files in GAP packages

```
FFECONWAY.ZNC := function(p,d)
  local   fam,  zc,  v;
  fam := FFEFamily(p);
  if not IsBound(fam!.ZCache) then
    fam!.ZCache := [];
  fi;
  zc := fam!.ZCache;
  ...
  return zc[d];
end;
```

Some common recipes

Global lists or records, which are not changed after their creation	make read-only
Global lists or records, which may change atomically at runtime, possibly not losing or changing anything already stored	make atomic, possibly with write-once or even strict write-once semantics
Global lists used as caches, which may be modified in a non-atomic way	put in a shared region to ensure that only one thread may modify them at a time
Data which make sense only within some limited scope	make thread-local

Making objects read-only

Search for calls of **MakeReadOnly** in the GAP library to see examples:

- Making read-only global function records like GAPTCENUM, GAPKB_REW, IEEE754FLOAT, SMTX.
- Making read-only global lists such as e.g. CompareCyclotomicCollectionHelper_Semirings.
- Making read-only special objects e.g. infinity, NullMapMatrix, Info classes.
- Making read-only type objects such as e.g. TypeOfTypes, TypeOfFamilyOfTypes, TypeOfFamilyOfFamilies.
- Making read-only ZmodnZObj objects (see zmodnz.gi).
- Organising global data in read-only lists of write-once atomic lists, e.g. TYPES_VEC8BIT and TYPES_MAT8BIT.
- Making read-only thread-local lists before storing in the shared cache, e.g. fam!.ConwayPolCoeffs[d] := MakeReadOnly(cp) from ffeconway.gi

Atomic records

Examples of global records which were made atomic include:

- GAPInfo record and its components, e.g. GAPInfo.UserPreferences, GAPInfo.CommandLineEditFunctions, GAPInfo.PackagesLoaded etc.
- CONWAYPOLYNOMIALSINFO.cache used for p > 110000 (with write-once semantics)
- Such records may change at runtime so they can not be made immutable, but we assume that there is no need to lock such data if part of them is being modified.

Using named regions

If a region contains many objects that should be locked simultaneously, it may be convenient to use named regions. Named regions may be created using e.g.

```
TERMINAL_REGION := ShareObj("TERMINAL_REGION");
```

or

```
HELP_REGION:=NewRegion("HELP_REGION");
```

Then instead of listing variable(s) that need to be locked, an atomic statement may use the name of the region, e.g.

```
atomic readonly HELP_REGION do
  ...
od;
```

Other precautions

- Careful about using external representation of an object (will produce thread-local data)
- Beware of using CLONE_OBJ - it is a horrible hack
- Careful about in-place conversions (more on this later)

Changes with major impact on the foreground test

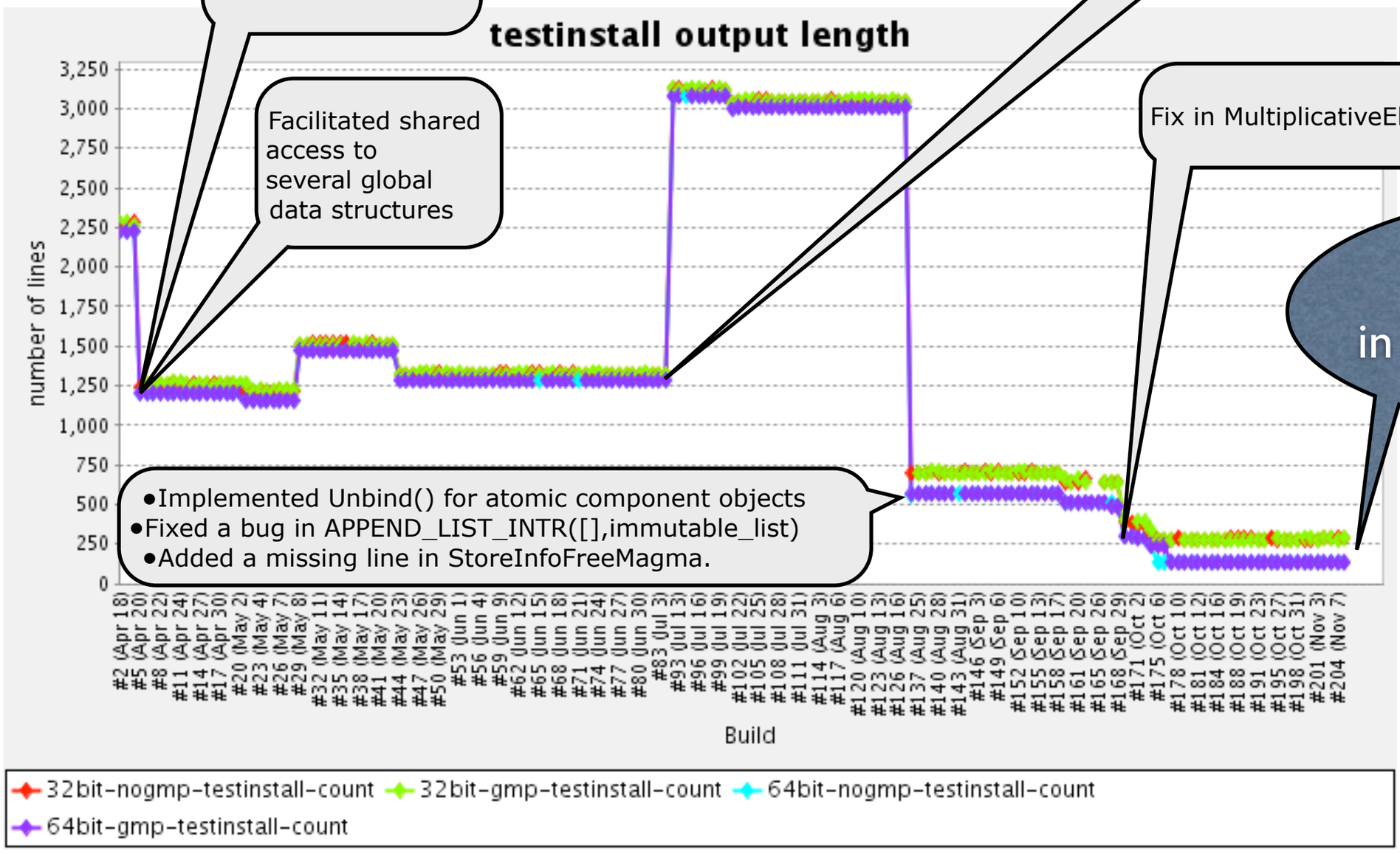
Fixed and simplified extended type construction

Facilitated shared access to several global data structures

A series of cleanups in the kernel and library revealed a number of issues which did not show up before

Fix in MultiplicativeElementsWithInversesFamilyByRws

in the meantime,
in the underground ...



Changes with major impact on the background test

Fixed and simplified extended type construction

Facilitated shared access to several global data structures

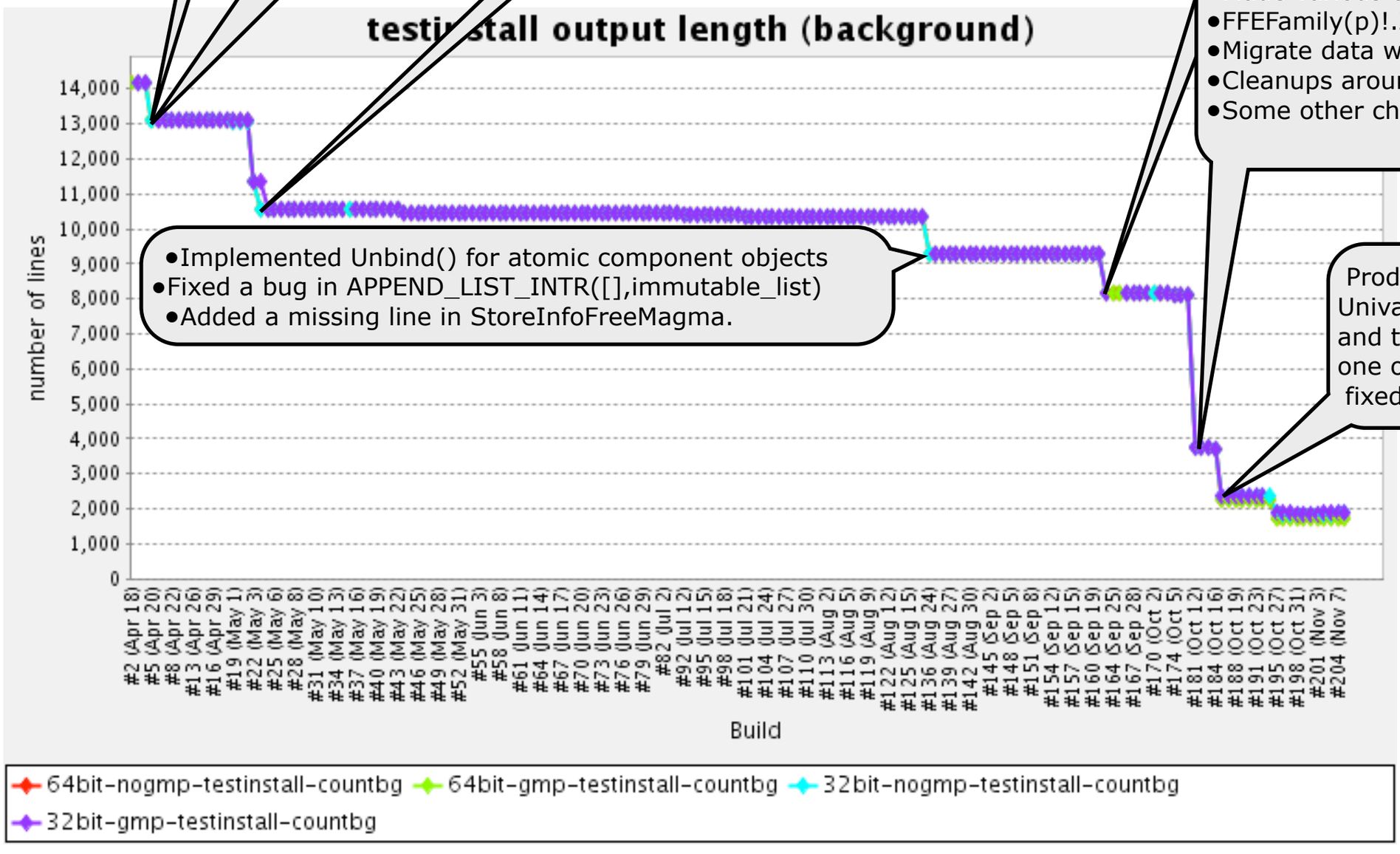
•TYPES_VEC8BIT shared (later was made an atomic list of atomic lists with write once semantics
• Finite fields kernel code thread-safe
•fam!.threeLaurentPolynomialTypes made immutable

•TYPES_MAT8BIT made an atomic list of atomic lists with write once semantics
•Make ffe to integer conversion tables immutable when we create them

•Called MakeWriteOnceAtomic for FFEFamily(p) and sorted out its caches.
•Made various objects in zmodnz.gi read-only.
•FFEFamily(p)!.ZCache is MadeWriteOnceAtomic list.
•Migrate data when extending PrimesProofs.
•Cleanups around ConwayPolynomials
•Some other changes

•Implemented Unbind() for atomic component objects
•Fixed a bug in APPEND_LIST_INTR([],immutable_list)
•Added a missing line in StoreInfoFreeMagma.

ProdCoeffUnivfunc calls UnivariateRationalFunctionByExtRepNC and this involves calling CLONE_OBJ on one of its read-only arguments. This was fixed using ShallowCopy.



GAP standard test suite

(Currently observed numbers for the HPC-GAP compiled in 64-bit mode with GMP. Alpha release also uses packages: AutPGrp, AtlasRep, CTblLib, GAPDoc, GrpConst, irredsol, TomLib)

Test	Number of diffs	
	Main execution thread (**alpha-release**)	Background thread (no packages)
testinstall	0	12
testall	1 (just different degree perm.rep)	119 (some from missing pkgs)
Tutorial examples	8 (all harmless e.g. randomness)	
Reference manual examples	58 (all harmless e.g. randomness)	
Global variables remaining thread-local	86	

Debugging toolkit

- May be disruptive: no nice recovery from the break loop
- ReadGuards and WriteGuards - special assertions
- `LastInaccessible;`
- `RegionOf(LastInaccessible);`
- `ViewShared(LastInaccessible);`
- `MAKE_PUBLIC(LastInaccessible);`
- `UNSAFE_VIEW(LastInaccessible);`
- `FindAllGVarsHolding(LastInaccessible);`

Debugging toolkit

- With ‘make debug’ or ‘make gapdebug’:
  - `CREATOR_OF(LastInaccessible);`
  - `List(CREATOR_OF(LastInaccessible), NAME_FUNC);`
- `OBJ_HANDLE(...)`
- `DISABLE_GUARDS(n)`:
  - disable read guards with n=1
  - disable read and write guards with n=2
  - enable with n=0

Old, old tale about an error

```
gap> TaskResult(RunTask(SmallGroup,[8,3]));
Error, No read access to object 4398002176 of type list (plain,cyc)
in gen/vars.c, line 1419, function EvalElmList(), accessing list in
  if IsIdenticalObj( LETTER_WORD_EREP_CACHE[i], w )  then
    return LETTER_WORD_EREP_CACHEVAL[i];
fi; called from
ERepLettWord( w ) called from
NumberSyllables( gens[i] ) called from
SingleCollectorByRelators( efam, gens, rels, conflicts ) called from
PolycyclicFactorGroupByRelators( ElementsFamily( FamilyObj( fgrp ) ),
GeneratorsOfGroup( fgrp ),
 rels ) called from
PolycyclicFactorGroup( FreeGroupOfFpGroup( F ), RelatorsOfFpGroup( F )
) called from
...  at line 0 of *defin*
brk> OBJ_HANDLE(4398002176);
<obj 4398002176 inaccessible in region: thread region #0>
brk> UNSAFE_VIEW(OBJ_HANDLE(4398002176));
[ 1, 1, 1 ]
```

- This points to `LETTER_WORD_EREP_CACHEVAL`

Using thread-local caches

- **Indeed,** `ERepLettWord` function from `wordlett.gi` uses a cache of the last three external representations, so we made them thread-local by adding three lines:

```
##  We cache the last 3 external representations to use them for syllable access.
LETTER_WORD_EREP_CACHE:=[1,1,1];    # initialization with dummys
LETTER_WORD_EREP_CACHEVAL:=[1,1,1]; # initialization with dummys
LETTER_WORD_EREP_CACHEPOS:=1;

MakeThreadLocal( "LETTER_WORD_EREP_CACHE" );
MakeThreadLocal( "LETTER_WORD_EREP_CACHEVAL" );
MakeThreadLocal( "LETTER_WORD_EREP_CACHEPOS" );

BindGlobal("ERepLettWord",function(w)
local  i,r,elm,len,g,h,e;
  for i in [1..3] do
    if IsIdenticalObj(LETTER_WORD_EREP_CACHE[i],w) then
      return LETTER_WORD_EREP_CACHEVAL[i];
    fi;
  od;
  ...
  LETTER_WORD_EREP_CACHE[LETTER_WORD_EREP_CACHEPOS]:=w;
  LETTER_WORD_EREP_CACHEVAL[LETTER_WORD_EREP_CACHEPOS]:=Immutable(r);
  LETTER_WORD_EREP_CACHEPOS:=(LETTER_WORD_EREP_CACHEPOS mod 3)+1;
  return r;
end);
```

Using shared caches

- As an example, we consider `APPROXROOTS` which is used by `ApproximateRoot`
- First, `APPROXROOTS` must be shared upon its creation:

```
APPROXROOTS:=[]; ShareSpecialObj(APPROXROOTS);
```

- Its usage in `ApproximateRoot` follows the "check-compute-recheck" pattern:
  - check if it's already known and return if so (only read lock is required)
  - if not known, compute it. Do not hold the lock while doing it.
  - acquire the write-lock, and check again if the result is already known, because it may be computed by another thread in the meantime:
    - if known, abandon your result and return what is already stored
    - if not known, save and return your result.
- It is important to remember that the cache is in the shared region, so any data which are written to the cache, should also be migrated to its region or make immutable, whatever is appropriate.
- See next slide for an example.
- Another examples of a shared caches (which may be flushed) are `ABELIAN_NUMBER_FIELDS` and `Z_MOD_NZ`.
- For more examples, look for calls of `ShareSpecialObj` in the library.

Shared caches - example

```
BindGlobal("ApproximateRoot",function(arg)
local r,e,f,x,nf,lf,c,store;
  r:=arg[1]; e:=arg[2];
  # CHECK
  store:= e<=10 and IsInt(r) and 0<=r and r<=100;
  if store then
      atomic readonly APPROXROOTS do
          if IsBound(APPROXROOTS[e]) and IsBound(APPROXROOTS[e][r+1])
          then return APPROXROOTS[e][r+1];
          fi;
      od;
  fi;
  # COMPUTE x
  ...
  # RECHECK
  if store then
      atomic readwrite APPROXROOTS do
         if not IsBound(APPROXROOTS[e]) then
             APPROXROOTS[e]:=MigrateObj([],APPROXROOTS);
             APPROXROOTS[e][r+1]:=x;
         elif IsBound(APPROXROOTS[e][r+1]) then
             return APPROXROOTS[e][r+1];
         else
             APPROXROOTS[e][r+1] := x;
         fi;
      od;
  fi;
  return x;
end);
```

Write-once atomic lists for accumulating caches

```
InstallMethod( LargeGaloisField, [IsPosInt, IsPosInt], function(p,d)
    ...
    fam := FFEFamily(p);
    if not IsBound(fam!.ConwayFieldCache) then
       fam!.ConwayFieldCache := MakeWriteOnceAtomic([]);
    fi;
    if not IsBound(fam!.ConwayFieldCache[d]) then
       fam!.ConwayFieldCache[d] :=
         FieldByGenerators(GF(p,1),[FFECONWAY.ZNC(p,d)]);
    fi;
    return fam!.ConwayFieldCache[d];
end);
```

- `fam!.ConwayFieldCache` only accumulates the data. It is never being reset. Each field GF(p^d) is stored in a fixed *d*-th position. Since `fam!.ConwayFieldCache` is an atomic lists, different threads may read it without explicit locking, and making it write-once ensures that only the 1st thread writing to the *d*-th position will actually write it, others will be using stored value.
- `TYPES_VEC8BIT, TYPES_MAT8BIT` are read-only lists of write-once atomic lists
- `_TransformationFamiliesDatabase` is a write-once atomic list to store transformation types and families.

Storing additional information in the type

- Types are created in read-only region
- Sometimes, type is used to store additional information, e.g. in `MultiplicativeElementsWithInversesFamilyByRws` methods in `rwspcgrp.gi`
- Recommended solution to call `StrictBindOnce`

```
InstallMethod( MultiplicativeElementsWithInversesFamilyByRws,
               "16 bits family",  true,
    ...
    # create the default type for the elements
    fam!.defaultType := NewType( fam, IsPackedElementDefaultRep );
    # create the special 16 bits type
    fam!.16BitsType := NewType( fam, Is16BitsPcWordRep );

    for i  in [ AWP_FIRST_ENTRY+1 .. AWP_FIRST_FREE-1 ]  do
     StrictBindOnce( fam!.16BitsType, i, sc![SCP_DEFAULT_TYPE]![i] );
    od;
    StrictBindOnce( fam!.16BitsType, AWP_PURE_TYPE, fam!.16BitsType );
    StrictBindOnce( fam!.16BitsType, PCWP_NAMES, FamilyObj(ReducedOne(sc))!.names );
    sc := ShallowCopy(sc);
    sc![SCP_DEFAULT_TYPE] := fam!.16BitsType;
    StrictBindOnce( fam!.16BitsType, PCWP_COLLECTOR, sc );

    SetOne( fam, ElementByRws( fam, ReducedOne(fam!.rewritingSystem) ) );
    pcs := List( GeneratorsOfRws(sc), x -> ElementByRws(fam,x) );
    SetDefiningPcgs( fam, PcgsByPcSequenceNC( fam, pcs ) );
    return fam;
end );
```

New: CopyToVectorRep

- May eventually replace **CopyToVectorRep**(**v** , **q**) and associates
- **CopyToVectorRep** returns a compressed vector **r** which is equal to the list **v** but is given in the internal row vector representation over the field of size **q**, if this is possible. If v can not be written in such representation, it returns **fail**.
- The output of **CopyToVectorRep** will have the same mutability as **v**. One can use **CopyToVectorRepMutable** and **CopyToVectorRepImmutable** to ensure that the output will be mutable or immutable for any **v**.
- The first argument of these functions may already be a compressed vector. In this case, if **q** is different from size of the field over which the vector **v** is represented, then **r** will be the result of rewriting **v** over the field of size **q**. If v is already represented over the field of size **q**, then the result will be a vector **r** identical to v, if both **v** and r are immutable, and a new vector equal to **v** otherwise.
- **CopyToVectorRepNC** etc. are NC-versions of **CopyToVectorRep** etc. It is forbidden to call the NC-version unless **v** is a plain list or a row vector, **q**<=256 is a valid size of a finite field, and all elements of **v** lie in this field. Violation of this condition can lead to unpredictable behaviour or a system crash. (Setting the assertion level to at least 2 might catch some violations before a crash). The NC-version will never return **fail**, but may enter a break loop in some cases.

CopyToVectorRep - example

Create a row vector and then ask GAP to rewrite it over GF(2) and then over GF(4)

```
gap> v := [Z(2)^0,Z(2),Z(2),0*Z(2)];
[ Z(2)^0, Z(2)^0, Z(2)^0, 0*Z(2) ]
gap> RepresentationsOfObject(v);
[ "IsPlistRep", "IsInternalRep" ]
gap> w:=CopyToVectorRepNC(v,2);
<a GF2 vector of length 4>
gap> RepresentationsOfObject(w);
[ "IsDataObjectRep", "IsGF2VectorRep" ]
gap> u:=CopyToVectorRepNC(w,4);
[ Z(2)^0, Z(2)^0, Z(2)^0, 0*Z(2) ]
gap> RepresentationsOfObject(u);
[ "IsDataObjectRep", "Is8BitVectorRep" ]
gap> t:=CopyToVectorRepNC(v,4);
[ Z(2)^0, Z(2)^0, Z(2)^0, 0*Z(2) ]
gap> RepresentationsOfObject(t);
[ "IsDataObjectRep", "Is8BitVectorRep" ]
```

Concluding remarks

- Small Groups Library: SmallGroup works for may orders, including everything for |G| <= 512
- Polycyclic collectors - difficult problem

Popular wisdom
(from electrical engineering)

Any problem may be explained by either

- missing contact when you need it

or

- shortcut, i.e. contact when you don’t need it

Popular wisdom
(from parallel programming)

Any problem may be explained by either

- *no access to the object* when you need it

or

- *access to the object* when you don't need it

Everything else is a bug