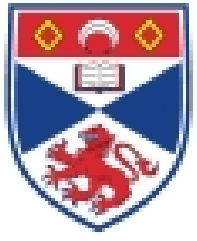
# *Connecting Coq theorem prover to GAP*

Vladimir Komendantsky Alexander Konovalov
Steve Linton

University of St Andrews, UK



SCIEnce/CICM'10

*Motivation*

What can interactive TP and CAS do for each other

*Related work*

*Brief background on interactive theorem proving*

Type systems

The SSReflect hierarchy

*Our development*

Our development regarding the SSReflect hierarchy

# *Possible development tracks of a Coq–CAS interface*

1. CAS as a hint engine for Coq
2. CAS as a proof engine for Coq
3. prove in Coq correctness of CAS algorithms

The particular CAS we are working with:

**GAP** – Groups, Algorithms, Programs

`http://www.gap-system.org/`

# *Motivation 1: Hint engine*

Mechanisms, introduced in proof assistants to improve the type inference algorithm:

- canonical structures,
- type classes,
- pullbacks,
- unification hints (Asperti): $\dfrac{?_x^{\vec{}} := \vec{H}}{P \equiv Q}$

*Problem*

Provide suitable hints for the unification procedure underlying type inference. How useful would be to employ the CAS as a search tool to generate hints?

# *Motivation 2: Proof engine*

- **Admit GAP results.** (Good scenario, although with limitations.)

*Example*

Automatically generate an axiom, e.g.,

```
Axiom isoG : G \iso E_n(35).
```

if we asked GAP to find a group isomorphic to a given Coq group `G` and the result was the Coq group `E_n(35)`.

- **Prove GAP results** (something Coq does not know in advance but can prove if a hint is received from GAP).

*Example*

As above, but ask Coq user to prove the generated statements, i.e.,

```
Lemma isoG : G \iso E_n(35).
```

followed by user or automated proof and `Qed`, or proof is impossible.

# *Motivation 3: Correctness of GAP algorithms*

Frequently, GAP standard library functions rely on side effects produced by other functions:

```
##############################################################################
##
#M  Size( <G> ) . . . . .  for cyclotomic matrix group not known to be finite
##
InstallMethod( Size,
    "cyclotomic matrix group not known to be finite",
    [ IsCyclotomicMatrixGroup ],
    function( G )
    if IsFinite( G ) then
        return Size( G );  # now G knows it is finite
    else
        return infinity;
    fi;
end );
```

# *Motivation 3: Correctness of GAP algorithms*

Frequently, GAP standard library functions rely on side effects produced by other functions:

```
##############################################################################
##
#M  Size( <G> ) . . . . .  for cyclotomic matrix group not known to be finite
##
InstallMethod( Size,
    "cyclotomic matrix group not known to be finite",
    [ IsCyclotomicMatrixGroup ],
    function( G )
    if IsFinite( G ) then
        return Size( G );  # now G knows it is finite
    else
        return infinity;
    fi;
end );
```

Let's use side effects!

# *A formal model of algorithm implementation* ***would*** *imply a formal model of its specification*

```
##
#M  IsFinite( G ) . . . . . . . . . . .  IsFinite for cyclotomic matrix group
##
InstallMethod( IsFinite, "cyclotomic matrix group", [ IsCyclotomicMatrixGroup ],
function( G )
    local lat, ilat, grp, mat;
    # if not rational, use the nice monomorphism into a rational matrix group
    if not IsRationalMatrixGroup( G ) then
        return IsFinite( Image( NiceMonomorphism( G ) ) );
    fi;
    # if not integer, choose basis in which it is integer
    if not IsIntegerMatrixGroup( G ) then
        lat := InvariantLattice( G );
        if lat = fail then return false; fi;
        ilat := lat^-1; grp := G^(ilat); IsFinite( grp );
        # IsFinite may have set the size, so we save it;
        # <code omitted>
        # IsFinite may have set an invariant quadratic form
        # <code omitted>
        return IsFinite( grp );
    else
        return IsFinite( G );  # now G knows it is integer
    fi;
end );
```

# *Related work*

- J. Harrison and L. Théry: HOL–Maple
- H. Herbelin, M. Mayero, D. Delahaye: "Maple mode" for Coq.
- S. Ould-Biha: prototype external tactic coq_gap in C.
- ...
- Calculus of explicit substitutions (Cézar Muñoz):

  "proofs as terms" $\Longrightarrow$ "partial proofs as partial terms"

  typed partially specified $\lambda$-terms are built stepwise at the same time as partial proofs.

# *Coq-like interactive theorem provers*

- Theorem proving can be done using procedural or declarative scripting languages.
- Based on higher-order Curry–Howard correspondence in the Calculus of Inductive Constructions (an extension of $F_\omega$):
  - Procedural and declarative proof scripts produce $\lambda$-terms (also called *proof terms*).
  - Incomplete proofs are $\lambda$-terms with typed placeholders denoting missing subproofs.
- Library is made of definitions and proofs, both being $\lambda$-terms.
- Mathematical formulas can be typeset using ambiguous notations (especially complete implementation in **Matita**).

# *Terms and types*

*λ-terms*

$$\Lambda ::= \mathcal{V} \mid (\Lambda\ \Lambda) \mid (\lambda \mathcal{V}.\Lambda)$$

*Types*

$$\mathbf{T} ::= \mathbf{V} \mid (\mathbf{T} \to \mathbf{T}) \mid (\forall \mathbf{V}.\mathbf{T})$$

# *Type inference rules of System F (polymorphism)*

Var $$\frac{}{\Gamma \vdash x : \Gamma(x)}$$

App $$\frac{\Gamma \vdash t : A \to B \qquad \Gamma \vdash u : A}{\Gamma \vdash t\ u : B}$$

Abs $$\frac{\Gamma;\ x : A \vdash t : B}{\Gamma \vdash (\lambda x.t) : A \to B} \qquad x \notin \mathit{Dom}(\Gamma)$$

Inst $$\frac{\Gamma \vdash t : \forall x.A}{\Gamma \vdash t : A[x := u]}$$

$\mathsf{Gen}_x$ $$\frac{\Gamma \vdash t : A}{\Gamma \vdash t : \forall x.A} \qquad x \notin \mathit{FTV}(\Gamma)$$

# *Types and typability*

System F (Girard and Reynolds): types with universal quantifiers, *explicitly* typed (Church-style).

*Undecidability result: J. Wells, 1994*

System F, Curry-style: types are derived for pure lambda terms. Typability and type checking for System F (Curry-style) are equivalent and undecidable.

*Corollary*

*Any System F (Curry-style) type inference algorithm is non-terminating or incomplete.*

# *System $F^\omega$ (polymorphism + type operators)*

System $F_\omega$ is an inductively defined family of systems where, for $n \geq 0$, $F_n$ admits

- the type Type of types, that is, Type $\in F_n$,
- the type of functions from types to types $A \to B$ where $A \in F_{n-1}$ and $B \in F_n$.

Then, $F_\omega$ is defined to be

$$\bigcup_{0 \leq i} F_i$$

# *Term levels in Coq-like proof assistants*

1. fully specified terms: CIC terms

$$\forall a{:}\mathbb{Z},\ eq\ \mathbb{Z}\ a\ (Zplus\ a\ Z0)$$

2. partially specified terms: CIC terms with omitted subterms

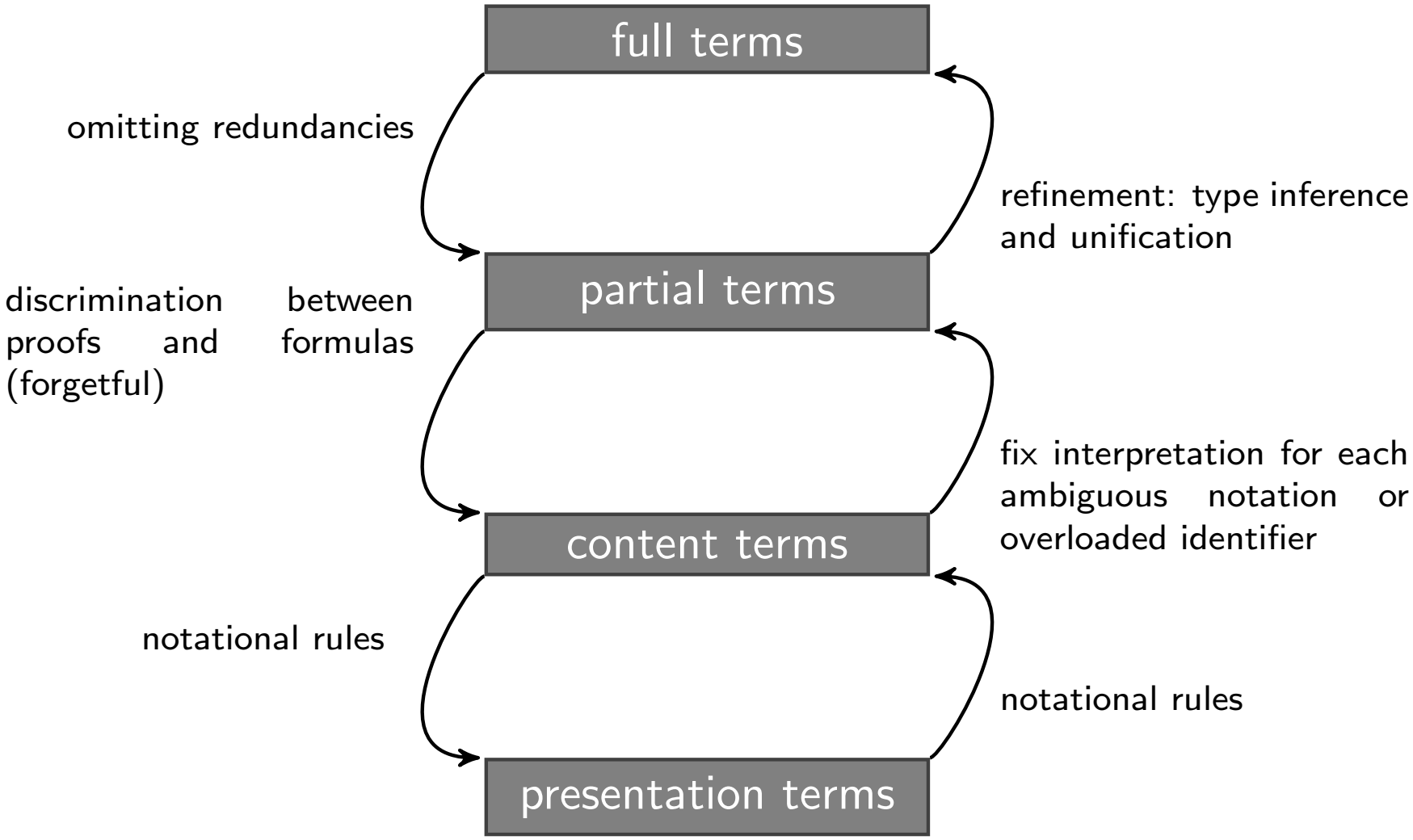
$$\forall a{:}\mathbb{Z},\ eq\ ?_1\ a\ (Zplus\ ?_2\ Z0)$$

3. content level terms: compact, overloaded human-oriented encoding with abstract syntactic structure

$$\forall a,\ a = \ \_ + 0$$

4. presentation level terms: formatting structure; finite, non-extendible language (e.g., MathML for printing, TeX-like for editing)

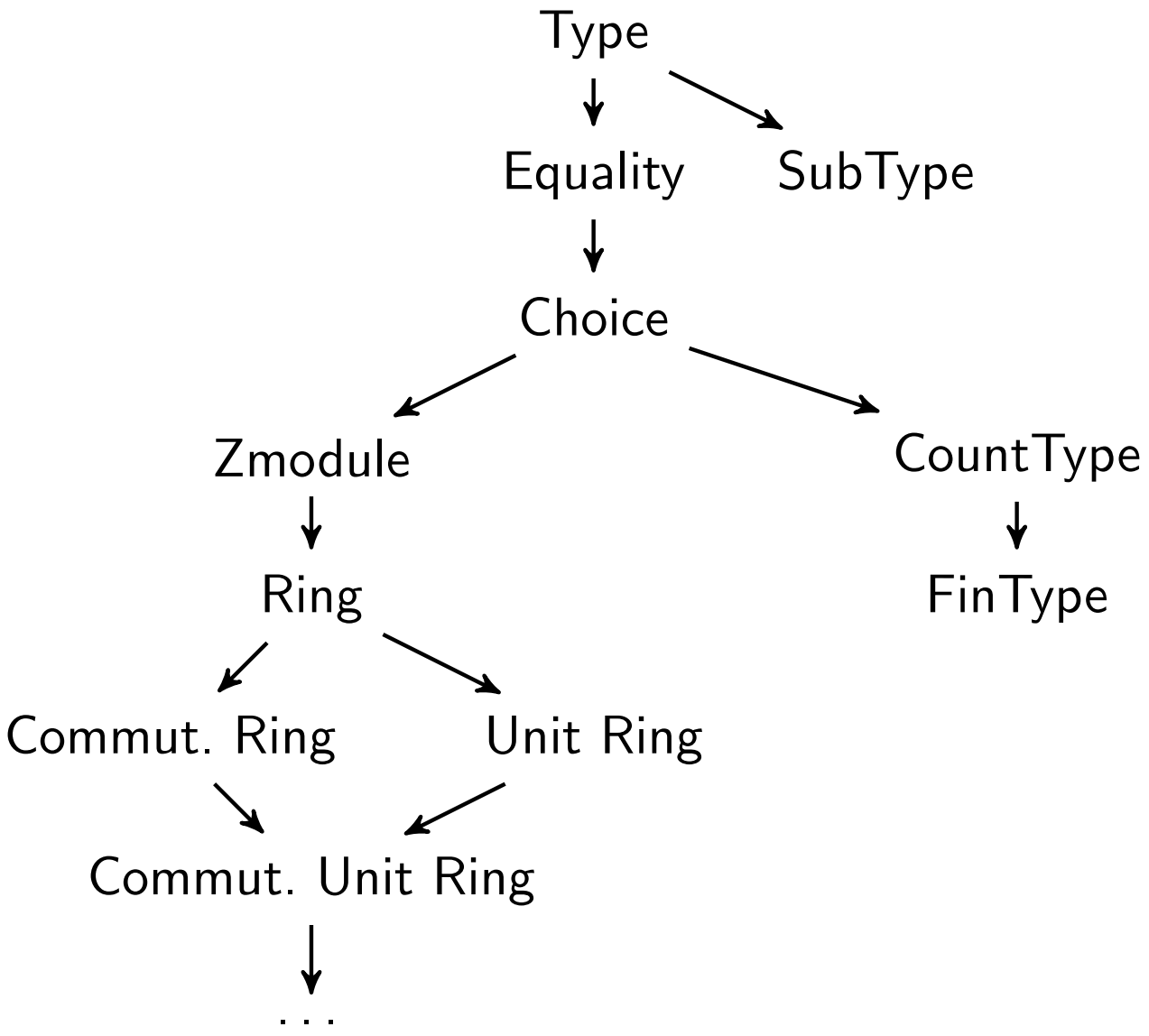
# *Term level translation*



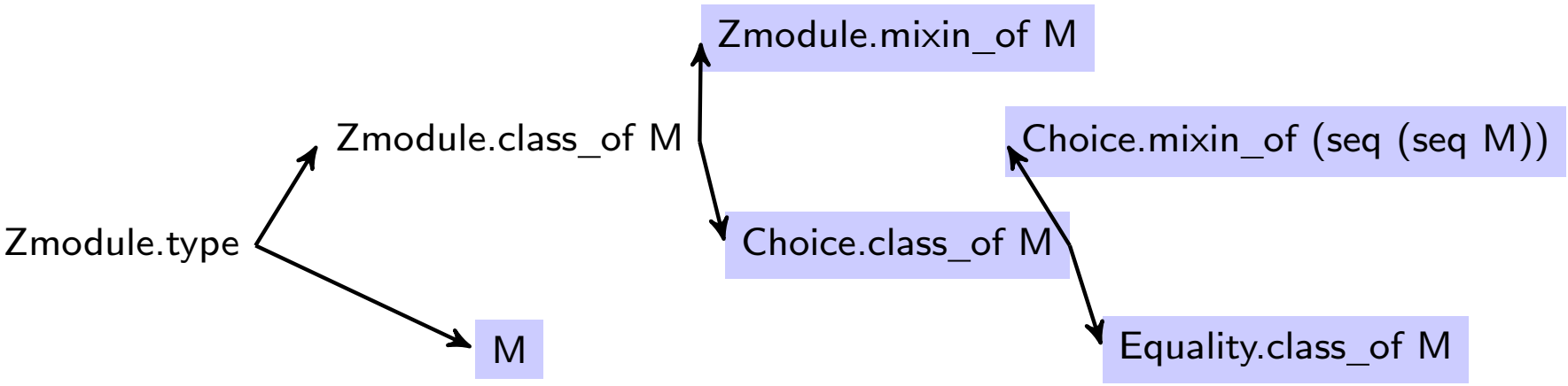
# *The choice of formalised mathematical structures: SSReflect's packed classes*

- Formalisation of mathematical concepts in type theory is not straightforward!
- There may be several ways to formalise a concept, and one is required to choose the most appropriate way.
- *De Bruijn factor*: formalised proof / paper proof (in lines of text)
- Formalise informal maths without making explicit too much information.
- Structures with inheritance; shared carrier and unification.

# *Algebraic hierarchy in SSReflect*



# *Packed classes and mixins (Zmodule example)*




*Legend*

$T_1$ *inherits* from (*coerces* to) $T_2$

# *Packed classes and mixins (Zmodule example) cont.*

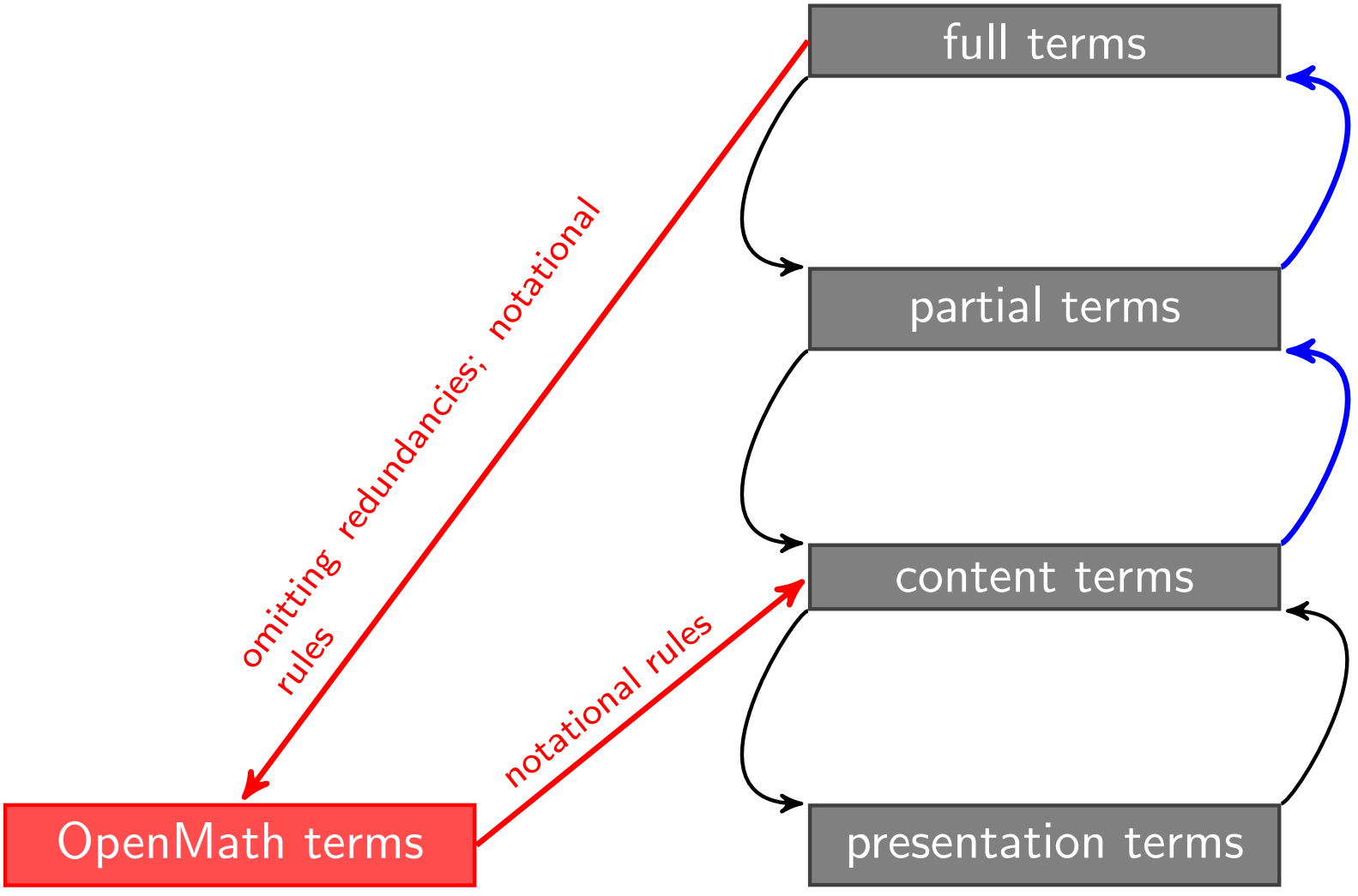
| | |
|---|---|
| Zmodule | basic algebraic structure |
| type | type for Zmodule structure |
| Mixin | constructs the *mixin* containing the definition of a Zmodule |
| pack M | packs the mixin M and builds a Zmodule of type type. |

```
Record mixin_of (M : Type) : Type := Mixin {
  zero : M;
  opp : M -> M;
  add : M -> M -> M;
  _ : associative add;
  _ : commutative add;
  _ : left_id zero add;
  _ : left_inverse zero opp add
}.

Record class_of (M : Type) : Type :=
  Class { base :> Choice.class_of M; ext :> mixin_of M }.
Record type : Type := Pack {sort :> Type; _ : class_of sort; _ : Type}.
Definition class cT := let: Pack _ c _ := cT return class_of cT in c.

Definition pack := let k T c m := Pack (@Class T c m) T in Choice.unpack k.
```

# *Term encoding of foreign data*



# *Implementation: Use a communication protocol*

**SCSCP** – Symbolic Computation Software Composability Protocol

`http://www.symbolic-computation.org/scscp`

- A computer algebra system may offer services over network and a client may employ them.
- All messages in the protocol are represented as OpenMath objects, using the new Content Dictionaries scscp1 and scscp2, developed for this purpose.

# *Possible implementation of a proof engine*

Implementation by

- Ltac (tactic language of Coq),
- reflection (that is, definition of a model of GAP in Coq),
- possibly, GAP instrumentation.

# *Summary*

We are. . .

- working with a particular kind of mathematical structures in Type Theory: the packed classes of SSReflect (constructive Finite Group Theory library);
- currently implementing a prototype hint engine for Coq that employs GAP as a source of background knowledge:
  - higher-level (content-oriented) interaction with GAP;
  - support for packed classes is being developed.

# *Summary*

We are. . .

- working with a particular kind of mathematical structures in Type Theory: the packed classes of SSReflect (constructive Finite Group Theory library);
- currently implementing a prototype hint engine for Coq that employs GAP as a source of background knowledge:
  - higher-level (content-oriented) interaction with GAP;
  - support for packed classes is being developed.

Thank you!