The Coq proof assistant : principles and practice

J.-F. Monin

Université Grenoble Alpes

2016

Lecture 1

Outline

Outline

Overview of Formal Methods

Understanding Formal Methods, J.F. Monin, Springer, 2003

- Static analysis
- Model Checking
- Deductive techniques
- Soundness: LCF architecture, proof terms (can be checked independantly)

Trade off

- pencil-paper / tool support
- automatization / generality
- ...

In summary

- Describe a model
- Explain it
- Reason about it
- Be clean and precise

Use math and logic... and make it funny!

Coq teaching in the world

Now routinely taught in many highly ranked universities

- France: Paris, Grenoble, Lyon, Bordeaux, Strasbourg…
- Europe: UK, Italy,…
- USA: Harvard, Yale, U. Pennsyvania, MIT, Princeton…
- Australia
- China: Coq Summer School Tsinghua, Suzhou, Shanghai

Some industrial uses

Spacecrafts, airplanes (Airbus, Boing)

Microsoft
Intel
French railways
Telecom Operators
Nuclear power plants
Banks
Cryptography

Contents of this course

Discover 3 aspects of Coq

1. Coq as a proof assistant
   - write precise and clear definitions
   - how to state meaningful theorems
   - how to prove them in a perfectly rigorous way
     this task is interactive: tedious parts can be discharged by the machine but creative part need input from a human.

Contents of this course

Discover 3 aspects of Coq

2. Coq as a challenging programming language
   - many applications of Coq to problems arising in computer science

Contents of this course

Discover 3 aspects of Coq

3. Applications to reasoning about non-trivial programs
   - lists, trees...
   - data-structures implemented with pointers

Outline

Firsts steps to Coq

Key idea: abstraction

- take a concrete expression
- make some value (repeated or not) a parameter
- that's it

Simple but far reaching

The abstract thing can be

- a data, a function, a program, a type
- a family of them
- subtle combinations
  e.g. a program may depend on a previously abstracted value; programs may depend on pgms, or on types, or conversely.

A very powerful logic

- Statements
- Proofs: concrete data
- More powerful than Peano arithmetic:
  Goodstein sequences

A way to compute proofs for given statements

⇒ Programming comes first

A strange programming language

- Without state!!
- Called functional programming
- Components:

Components

- lambda-calculus (pure functions)
- inductive types

Remarks

- States can be simulated
- Actually lambda-calculus has the power of Turing machines

A strange programming language

State is a burden for reasoning

Immutable values are much more convenient

All proof assistants are related to a functional programming language

In the case of Coq (and others e.g. Agda, Matita, Lego, Nuprl) the relationship is very tight

Outline

Introduction

Coq

Lambda-calculus

Inductive types (graphically: trees)

Graphical syntax

Composition of trees

Trees with variables

More general trees
- Several constructors
- Polymorphic trees

Lambda-calculus

Receipe (part 1)

- Take your preferred programming language (C, Python, Ocaml, Java, Javascript,...)
- Remove objects, classes,...
- Remove state variables (global, static, local, ...)
- Remove assignments
- Remove `goto` statements
- Remove `if` statements, loops
- Remove all side effects

Lambda-calculus

What is left?

- expressions
- constants
- function calls
- (possibly recursive) function definitions

Receipe (part 2)

- Remove recursion

What you get is essentially lambda-calculus with constants

Lambda-calculus with constants

- Built-in computations
  on integers, Booleans, characters,...
- Function calls :
  replacement of formal parameters by actual parameters

Lambda-calculus

Receipe (part 3)

- Remove constants and built-in computations

What you get is essentially pure lambda-calculus

Pure lambda-calculus

- Function calls :
  replacement of formal parameters by actual parameters

Pure lambda-calculus

Just 3 things

- Variables: x, y, …
- Application: $U\ V$
- Abstraction: $\lambda x.U$

Just 1 computation rule: β-reduction

- Variables: x, y, …
- Application: $U\ V$
- $(\lambda x.U)V$ β-reduces to $U[x := V]$
  (in any context, i.e., at any position inside a λ-term)

In Coq: lambda-calculus + definitions

Math notation

$a \stackrel{\text{def}}{=\!=} 3$

$f \stackrel{\text{def}}{=\!=} \lambda x.\, x + 2$

Coq notation

```
Definition a := 3.
Definition f := fun x => x + 2.
```

Expansion of a definition to its body

Called δ-reduction

$f\ a$ reduces (in two δ steps) to $(\lambda x.\, x + 2)\, 3$

Then $(\lambda x.\, x + 2)\, 3$ β-reduces to $3 + 2$

Functions with two (or more) arguments

A function of x and y is a function of x which returns a function of y

- Example: $\lambda x.\,(\lambda y.\ x + 2 * y)$
- Shorthands:
  $\lambda x.(\lambda y.\ x + 2 * y)$
  $\lambda xy.\ x + 2 * y$
- Application:

$$\begin{aligned} & (\lambda x.\,(\lambda y.\ x + 2 * y)\ 5)\ 1 \\ \xrightarrow{\beta}\ & (\lambda y.\ 5 + 2 * y)\ 1 \\ \xrightarrow{\beta}\ & 5 + 2 * 1 \end{aligned}$$

Functions everywhere

Pure λ-calculus deals only with functions

- Variables actually stand for functions
- Functions return functions
- Function take functions as arguments
- Such functions are called higher-order functions

Pure λ-calculus has the power of Turing machines

- Constants (numbers, Booleans, etc.) can be encoded by functions
- Data-structures (pairs, tuples, lists, trees) can be encoded by functions
- Loops (iteration, recursion) can be encoded by functions

Main properties of lambda-calculus (1)

Confluence (Church-Rosser)

- A redex is a position in a λ-term where a β-reduction is possible.
- A λ-term may contain several redexes
- Reducing a redex may produce 0, 1 or several new redexes
- Therefore, there are in general many ways to compute (reduce and reduce) a given term
- However, the final result (if any) is always the same: we say that pure λ-calculus has the Church-Rosser property

Main properties of lambda-calculus (2)

Termination (Normalization)

- A term without redex is said to be normal (end of computations)
- We say that a term U is weakly (respectively strongly) normalizing if, respectively
  - there exist a reduction sequence $T = T_0 \xrightarrow{\beta} T_1 \xrightarrow{\beta} \ldots T_n$ such that T_n is normal
  - all reduction sequences $T = T_0 \xrightarrow{\beta} T_1 \xrightarrow{\beta} \ldots$ eventually end with a normal term T_n
- (Pure) λ-calculus contains non-normalizing terms, e.g., $\Omega \stackrel{\text{def}}{=\!=} \Delta\Delta$ with $\Delta \stackrel{\text{def}}{=\!=} \lambda x.\, xx$
- However, typed versions of λ-calculus, including Coq, don't allow such terms – actually all terms are strongly normalizing

Main properties of lambda-calculus (3)

The version of pure typed λ-calculus used in Coq is called the Calculus of Constructions (CoC, or CC).
The full λ-calculus used in Coq also contains inductive types; it is called the Calculus of Inductive Constructions (CIC).

Alltogether, confluence and normalization ensure that functions do provide a unique result for any input.
That is, functions are total (defined everywhere).

Examples of λ-calculi with this feature include

- simply typed λ-calculus, contained in
- CoC, itself contained in
- CIC

We will see that these typed λ-calculi have a logical interpretation. Totality is mandatory for the underlying logic to be consistent, and then to be usable in a proof assistant!

A glance at typed lambda-calculus (1)

Simply typed λ-calculus

- Types are atomic types or arrow types $A \to B$
- All variables are provided such a type
- If U has type $A \to B$ and V has type A, then $U\ V$ has type B
- If x has type A and U has type B, then $\lambda x.U$ has type $A \to B$
- Example: $\lambda x.x$ has many types such as $A \to A$, $(A \to B) \to (A \to B)$, etc.

A glance at typed lambda-calculus (2)

Polymorphic typed λ-calculus

- Simple types + universally quantified types, e.g. $\forall X, X \to X$ (a satisfactory type for $\lambda x.x$)
- Such types are called polymorphic types

CoC (Calculus of Constructions)

- simple types
- polymorphic types
- dependent types (see later)

CIC (Calculus of Inductive Constructions)

- COC + inductive types

Expressive power of untyped lambda-calculus

Terms similar to Ω can be used to define general recursion. E.g. $\mathrm{Y} \stackrel{\text{def}}{=\!=} \lambda f.\, (\lambda x.\, f(xx))(\lambda x.\, f(xx))$.

Exercise: check that $\mathrm{Y}\ f$ is a fixed point of f, that is, it β-reduces to a term which is equivalent to $f(\mathrm{Y}\ f)$.

Any "recursive" definition of a function can be defined using Y. Therefore, untyped pure λ-calculus has the power of Turing machines.

However, strong normalization is lost, since such a computation contains infinite sequences of reductions (Hint : look a the redex inside Y).
Untyped pure λ-calculus is logically inconsistent
(Technically, Y could be used to prove `False`).

Expressive power of typed lambda-calculi

Y is not typable even in CoC

- Good news: the underlying logic is consistent
- Bad news: general recursion is lost; is it serious?

Limited forms of recursion are typable:
(higher order) iteration and primitive recursion.

Expressive power of some typed λ-calculi

- simply typed λ-calculus: very weak (polynomials), moreover unconvenient
- CoC: very powerful – any practically provably total function can be represented
  (reminder: Goodstein sequences);
  however, still not very convenient
- CIC: very powerful (similar to CoC)
  but much more convenient

Live Demo

Outline

Introduction

Coq

Lambda-calculus

Inductive types (graphically: trees)

Graphical syntax

Composition of trees

Trees with variables

More general trees
- Several constructors
- Polymorphic trees

Types everywhere

Very powerful types
Everything has a type, even types
We can compute on types and on values at the same time.

Examples: families of types.

- Example: n-tuples, with $n = 1, 2 \ldots$ even 0.

… So it will become complex…

We start with a graphical syntax

Types having finitely many values

The simplest are called an enumeration

Example

```
Red : color                Red_f : rgb
Orange : color
Yellow : color
Green : color              Green_f : rgb
Blue : color               Blue_f: rgb
Indigo : color
Violet : color
```

Warning: a value has only one type

What is the type of `color` and `rgb`?

```
color : Set
rgb : Set
```

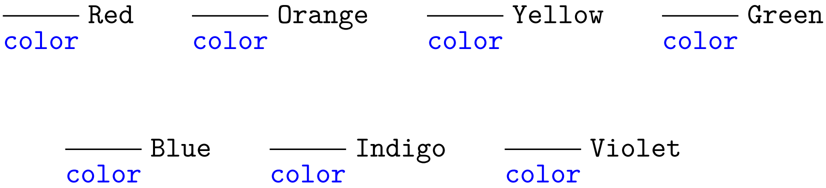
What is the type of `Set`?

```
Set : Type
```

What is the type of `Type`?

`Type`(i) : `Type`$(i+1)$

Graphical syntax



`Red`
`color`

`Orange`
`color`

`Yellow`
`color`

`Green`
`color`

`Blue`
`color`

`Indigo`
`color`

`Violet`
`color`

Outline

Introduction

Coq

Lambda-calculus

Inductive types (graphically: trees)

Graphical syntax

Composition of trees

Trees with variables

More general trees
  - Several constructors
  - Polymorphic trees

Graphical syntax

$$\frac{}{\mathtt{color}}\ \mathtt{Red} \qquad \frac{}{\mathtt{color}}\ \mathtt{Orange} \qquad \frac{}{\mathtt{color}}\ \mathtt{Yellow} \qquad \frac{}{\mathtt{color}}\ \mathtt{Green}$$

$$\frac{}{\mathtt{color}}\ \mathtt{Blue} \qquad \frac{}{\mathtt{color}}\ \mathtt{Indigo} \qquad \frac{}{\mathtt{color}}\ \mathtt{Violet}$$

The horizontal bar means: MAKES

`Red`, `Orange`,. . . are called CONSTRUCTORS

At the same time we have $\frac{}{\mathtt{Set}}\ \mathtt{color}$

Intermezzo: definitions

In order to save space, we use definitions.
E.g. *(Coq syntax)*

```
Definition R := Red.
```

means that `R` is definitionally the same as `Red`.

```
Definition co := color.
```

means that `co` is definitionally the same as `color`.

Hence
`Red:color`, `Red:co`, `R:color` and `R:co`
are all the same judgement

Graphical syntax

```
Definition O := Orange.   Definition Y := Yellow.
Definition G := Green.    Definition B := Blue.
Definition I := Indigo.   Definition V := Violet.
```

$$\frac{}{\texttt{co}}\,\texttt{R} \qquad \frac{}{\texttt{co}}\,\texttt{O} \qquad \frac{}{\texttt{co}}\,\texttt{Y} \qquad \frac{}{\texttt{co}}\,\texttt{G} \qquad \frac{}{\texttt{co}}\,\texttt{B} \qquad \frac{}{\texttt{co}}\,\texttt{I} \qquad \frac{}{\texttt{co}}\,\texttt{V}$$

```
Definition Rf := Red_f.   Definition Gf := Green_f.
Definition Bf := Blue_f.
```

$$\frac{}{\texttt{rgb}}\,\texttt{Rf} \qquad \frac{}{\texttt{rgb}}\,\texttt{Gf} \qquad \frac{}{\texttt{rgb}}\,\texttt{Bf}$$

Outline

Introduction

Coq

Lambda-calculus

Inductive types (graphically: trees)

Graphical syntax

Composition of trees

Trees with variables

More general trees
- Several constructors
- Polymorphic trees

Using a value

We know how to make (or construct) a value in `color` or in `rgb`.

Next issue: how to use a value

- use a given value
- use a (still) unknown value

Composition of n-tuples

Making a 4-tuple of rgb

$$\frac{\texttt{rgb}\quad\texttt{rgb}\quad\texttt{rgb}\quad\texttt{rgb}}{\texttt{tuple4}}\,\texttt{Mk4}$$

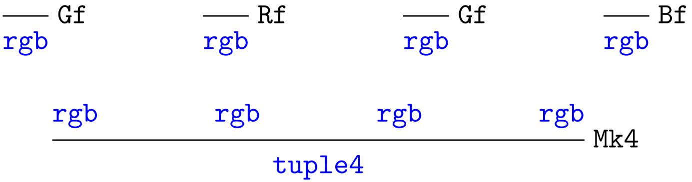
The constructor `Mk4` makes a tuple4 from

- a `rgb`
- a `rgb`
- a `rgb`
- a `rgb`

At the same time we have $\frac{}{\texttt{Set}}\,\texttt{tuple4}$

Pluging rgb into Mk4

Building blocks



Gf rgb

Rf rgb

Gf rgb

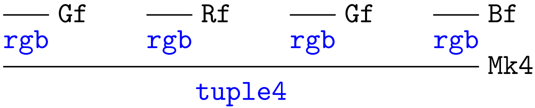
Bf rgb

rgb rgb rgb rgb

Mk4

tuple4

Connecting them yields the concrete 4-tuple of rgb



Gf rgb

Rf rgb

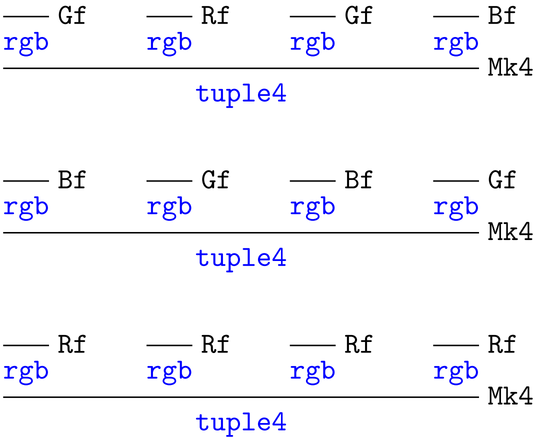
Gf rgb

Bf rgb

Mk4

tuple4

Others trees for 4-tuples



Gf Rf Gf Bf

rgb rgb rgb rgb

Mk4

tuple4

Bf Gf Bf Gf

rgb rgb rgb rgb

Mk4

tuple4

Rf Rf Rf Rf

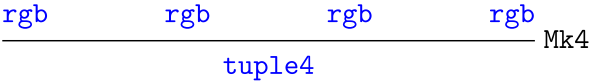
rgb rgb rgb rgb

Mk4

tuple4

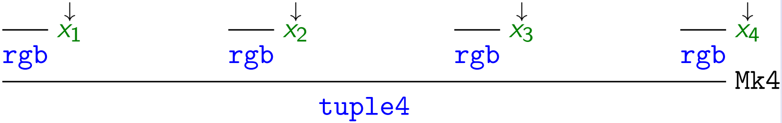
Another view on Mk4

As a building block



$$\frac{\mathtt{rgb}\qquad\mathtt{rgb}\qquad\mathtt{rgb}\qquad\mathtt{rgb}}{\mathtt{tuple4}}\ \mathtt{Mk4}$$

As a tree



$$\frac{\dfrac{}{\mathtt{rgb}}\ \overset{\downarrow}{x_1}\qquad\dfrac{}{\mathtt{rgb}}\ \overset{\downarrow}{x_2}\qquad\dfrac{}{\mathtt{rgb}}\ \overset{\downarrow}{x_3}\qquad\dfrac{}{\mathtt{rgb}}\ \overset{\downarrow}{x_4}}{\mathtt{tuple4}}\ \mathtt{Mk4}$$

This is called an open tree

Outline

- Introduction
- Coq
- Lambda-calculus
- Inductive types (graphically: trees)
- Graphical syntax
- Composition of trees
- Trees with variables
- More general trees
  - Several constructors
  - Polymorphic trees

Closed and open trees

The meaning (or value) of

$$\dfrac{\dfrac{}{\texttt{rgb}}\,\texttt{Gf} \qquad \dfrac{}{\texttt{rgb}}\,\texttt{Rf} \qquad \dfrac{}{\texttt{rgb}}\,\texttt{Gf} \qquad \dfrac{}{\texttt{rgb}}\,\texttt{Bf}}{\texttt{tuple4}}\,\texttt{Mk4}$$

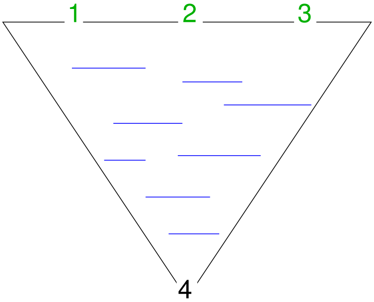
is completely defined: this is called a closed tree.

In contrast, the meaning of the open tree

$$\dfrac{\dfrac{}{\texttt{rgb}}\,\texttt{Gf} \qquad \dfrac{}{\texttt{rgb}}\,\overset{\downarrow}{x_2} \qquad \dfrac{}{\texttt{rgb}}\,\texttt{Rf} \qquad \dfrac{}{\texttt{rgb}}\,\overset{\downarrow}{x_4}}{\texttt{tuple4}}\,\texttt{Mk4}$$

depends on x_2 and x_4.

General shape: trees

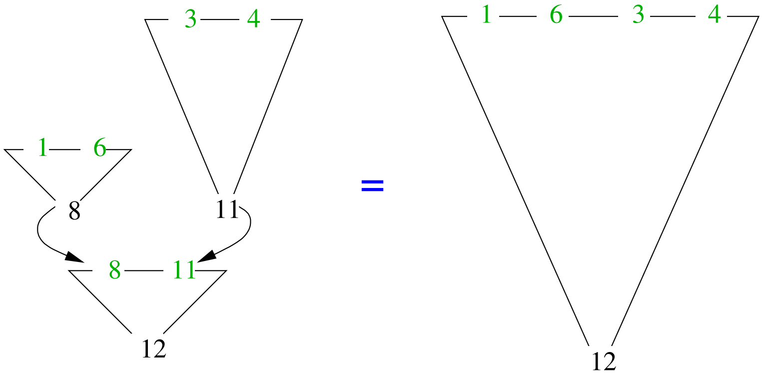


Interpretation

- At positions 1, 2, 3, 4: types
- 1, 2, 3: inputs
- 4: output (or result)

Makes the output from the inputs

Pluging trees



Environment

The tree

$$\dfrac{\dfrac{}{\texttt{rgb}}\,\texttt{Gf} \qquad \dfrac{}{\texttt{rgb}}\,\overset{\downarrow}{x_2} \qquad \dfrac{}{\texttt{rgb}}\,\texttt{Rf} \qquad \dfrac{}{\texttt{rgb}}\,\overset{\downarrow}{x_4}}{\texttt{tuple4}}\,\texttt{Mk4}$$

has a meaning for all trees plugged into x_2 and x_4.

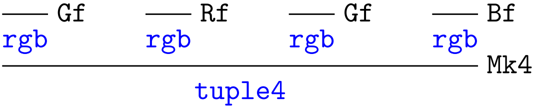
The variables $x_2 : \texttt{rgb}$ and $x_4 : \texttt{rgb}$ make up the environment of this tree

Outline

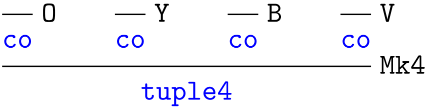
More 4-tuples

WRONG

4-tuple of rgb



4-tuple of color



Mk4 must be applied to arguments of a given type

How to make 4-tuples more general

Solution 1: have different constructors

$$\dfrac{\dfrac{}{\texttt{rgb}}\,\overset{\downarrow}{x_1} \qquad \dfrac{}{\texttt{rgb}}\,\overset{\downarrow}{x_2} \qquad \dfrac{}{\texttt{rgb}}\,\overset{\downarrow}{x_3} \qquad \dfrac{}{\texttt{rgb}}\,\overset{\downarrow}{x_4}}{\texttt{tuple4}}\ \texttt{Mk4rgb}$$

$$\dfrac{\dfrac{}{\texttt{color}}\,\overset{\downarrow}{x_1} \qquad \dfrac{}{\texttt{color}}\,\overset{\downarrow}{x_2} \qquad \dfrac{}{\texttt{color}}\,\overset{\downarrow}{x_3} \qquad \dfrac{}{\texttt{color}}\,\overset{\downarrow}{x_4}}{\texttt{tuple4}}\ \texttt{Mk4co}$$

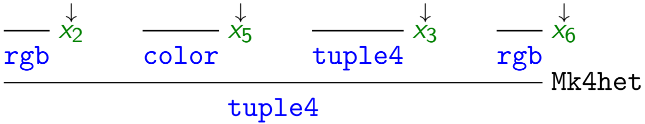
$$\dfrac{\dfrac{}{\texttt{tuple4}}\,\overset{\downarrow}{x_1} \qquad \dfrac{}{\texttt{tuple4}}\,\overset{\downarrow}{x_2} \qquad \dfrac{}{\texttt{tuple4}}\,\overset{\downarrow}{x_3} \qquad \dfrac{}{\texttt{tuple4}}\,\overset{\downarrow}{x_4}}{\texttt{tuple4}}\ \texttt{Mk4t4}$$

At the same time we have $\dfrac{}{\texttt{Set}}\ \texttt{tuple4}$

How to make 4-tuples more general

Remark

Beyond `Mk4rgb`, `Mk4co`, `Mk4t4`, we can imagine hererogeneous 4-tuples, for instance:



$$\dfrac{\dfrac{}{\texttt{rgb}}\overset{\downarrow}{x_2} \qquad \dfrac{}{\texttt{color}}\overset{\downarrow}{x_5} \qquad \dfrac{}{\texttt{tuple4}}\overset{\downarrow}{x_3} \qquad \dfrac{}{\texttt{rgb}}\overset{\downarrow}{x_6}}{\texttt{tuple4}}\ \texttt{Mk4het}$$

Many possibilities... to be considered again later.

How to make 4-tuples more general

Solution 2: only one constructor, but more general

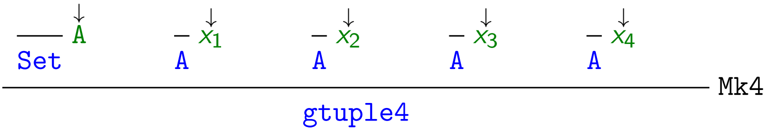
$$\frac{\texttt{A}\qquad\texttt{A}\qquad\texttt{A}\qquad\texttt{A}}{\texttt{gtuple4}}\;\texttt{Mk4}$$

But where does `A` come from?

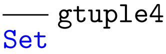
We want the previous tree for all `A`...

The magic

Solution 2: only one constructor, but more general



As usual, at the same time we have



Intermezzo: a shorthand for trees

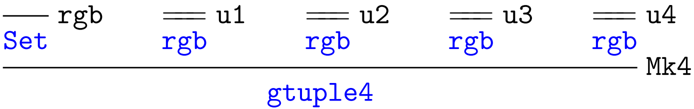
$$\dfrac{\overline{\overline{\texttt{tuple4}}}\ \texttt{t1} \qquad \overline{\overline{\texttt{tuple4}}}\ \texttt{t2} \qquad \overline{\overline{\texttt{tuple4}}}\ \texttt{t3} \qquad \overline{\overline{\texttt{tuple4}}}\ \texttt{t4}}{\texttt{tuple4}}\ \texttt{Mk4t4}$$

Where `t1` is for example defined as

$$\dfrac{\dfrac{}{\texttt{rgb}}\ \texttt{Gf} \qquad \dfrac{}{\texttt{rgb}}\ \texttt{Rf} \qquad \dfrac{}{\texttt{rgb}}\ \texttt{Gf} \qquad \dfrac{}{\texttt{rgb}}\ \texttt{Bf}}{\texttt{tuple4}}\ \texttt{Mk4rgb}$$

And so on for `t2`, etc.

Concrete example



rgb Set

u1 rgb

u2 rgb

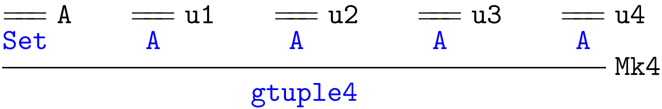
u3 rgb

u4 rgb

Mk4

gtuple4

General homogeneous 4-tuples



`A` `Set` `u1` `A` `u2` `A` `u3` `A` `u4` `A` `Mk4`

`gtuple4`

The type `A` can be many things beyond `rgb`

- `gtuple4`
- a complex tree

The trees `u1`, `u2`, `u3`, `u4` and `A` can be open
(they can depend on variables).

Exercises

1) Write trees for examples of 4-tuples of 4-tuples using `tuple4` and `gtuple4`.
Some of them, closed, some of them open
E.g. $\langle\langle \mathtt{R}, \mathtt{Y}, \mathtt{B}, \mathtt{B}\rangle, \langle \mathtt{B}, \mathtt{O}, x_4, \mathtt{R}\rangle, \langle x_7, x_7, x_7, \mathtt{V}\rangle, \langle \mathtt{V}, \mathtt{Y}, \mathtt{O}, \mathtt{R}\rangle\rangle$

2) Trees for heterogeneous pairs (2-tuples) and for heterogeneous triples.

3) Trees for homogeneous *n*-tuples, where *n* can be 1, 2 or 3.

4) Trees for heterogeneous *n*-tuples, where *n* can be 0, 1 or 2.