Course 1: Extraction of Terminating Algorithms in Coq

Dominique Larchey-Wendling
https://github.com/DmxLarchey/PC19

LORIA (Nancy), TU & WPI (Vienna), CNRS

PC'19, Herrsching, September 20, 2019

Extraction in Coq

- `Extraction` = Coq command
  - auto. maps a Coq term to a program (OCaml)
  - captures the Computational Content (CC)

Extraction in Coq

- `Extraction` = Coq command
  - auto. maps a Coq term to a program (OCaml)
  - captures the Computational Content (CC)
- Consider a fully specified term t:

$$t : \forall x : X, \mathbb{D}\ x \to \{y : Y \mid \mathbb{G}\ x\ y\}$$

$\mathbb{D} : X \to \texttt{Prop}$	Domain	Pre-condition
$\mathbb{G} : X \to Y \to \texttt{Prop}$	Specification	Post-condition

Extraction in Coq

- `Extraction` = Coq command
  - auto. maps a Coq term to a program (OCaml)
  - captures the Computational Content (CC)
- Consider a fully specified term t:

$$t : \forall x : X, \mathbb{D}\ x \to \{y : Y \mid \mathbb{G}\ x\ y\}$$

$\mathbb{D} : X \to \texttt{Prop}$	Domain	Pre-condition
$\mathbb{G} : X \to Y \to \texttt{Prop}$	Specification	Post-condition

- $\mathbb{D}\, x$ (domain) and $\mathbb{G}\, x\, y$ (spec)
  - are erased at extraction
  - $\textsc{extr}(t) : \textsc{extr}(X) \to \textsc{extr}(Y)$

Extraction in Coq

- `Extraction` = Coq command
  - auto. maps a Coq term to a program (OCaml)
  - captures the Computational Content (CC)
- Consider a fully specified term t:

$$t : \forall x : X, \mathbb{D}\ x \to \{y : Y \mid \mathbb{G}\ x\ y\}$$

$\mathbb{D} : X \to \texttt{Prop}$	Domain	Pre-condition
$\mathbb{G} : X \to Y \to \texttt{Prop}$	Specification	Post-condition

- $\mathbb{D}\,x$ (domain) and $\mathbb{G}\,x\,y$ (spec)
  - are erased at extraction
  - $\text{EXTR}(t) : \text{EXTR}(X) \to \text{EXTR}(Y)$
- What do $\mathbb{D}$ and $\mathbb{G}$ become?
  - it depends...
  - now: they are just erased
  - ideally (shortly ?): correctness of $\text{EXTR}(t)$

Dominique Larchey-Wendling
https://github.com/DmxLarchey/PC19

Certification by Extraction

- How to certify by extraction ?
  - From a given OCaml algo. $\varphi : \alpha \to \beta$
  - Get $\varphi = \mathrm{EXTR}(t_\varphi) : \mathrm{EXTR}(X_\alpha) \to \mathrm{EXTR}(X_\beta)$

$\mathbb{D}_\varphi : X_\alpha \to \texttt{Prop}$	Domain
$\mathbb{G}_\varphi : X_\alpha \to X_\beta \to \texttt{Prop}$	Specification
$t_\varphi : \forall x : X_\alpha,\ \mathbb{D}_\varphi\ x \to \{y : X_\beta \mid \mathbb{G}_\varphi\ x\ y\}$	Implementation

Certification by Extraction

- How to certify by extraction ?
  - From a given OCaml algo. $\varphi : \alpha \to \beta$
  - Get $\varphi = \textsc{extr}(t_\varphi) : \textsc{extr}(X_\alpha) \to \textsc{extr}(X_\beta)$

$\mathbb{D}_\varphi : X_\alpha \to \texttt{Prop}$	Domain
$\mathbb{G}_\varphi : X_\alpha \to X_\beta \to \texttt{Prop}$	Specification
$t_\varphi : \forall x : X_\alpha,\ \mathbb{D}_\varphi\ x \to \{y : X_\beta \mid \mathbb{G}_\varphi\ x\ y\}$	Implementation

- $\mathbb{D}_\varphi\, x$ (domain) and $\mathbb{G}_\varphi\, x\, y$ (spec)
  - erased at extraction
  - but contain the statement of correctness

Certification by Extraction

- How to certify by extraction ?
  - From a given OCaml algo. $\varphi : \alpha \to \beta$
  - Get $\varphi = \text{EXTR}(t_\varphi) : \text{EXTR}(X_\alpha) \to \text{EXTR}(X_\beta)$

$\mathbb{D}_\varphi : X_\alpha \to \texttt{Prop}$	Domain
$\mathbb{G}_\varphi : X_\alpha \to X_\beta \to \texttt{Prop}$	Specification
$t_\varphi : \forall x : X_\alpha,\ \mathbb{D}_\varphi\ x \to \{y : X_\beta \mid \mathbb{G}_\varphi\ x\ y\}$	Implementation

- $\mathbb{D}_\varphi\, x$ (domain) and $\mathbb{G}_\varphi\, x\, y$ (spec)
  - erased at extraction
  - but contain the statement of correctness
- Problem: how to define such a t_φ in Coq ?
  - no `let rec`, only restricted `Fixpoints` (`struct`)
  - How to control the CC ?

Dominique Larchey-Wendling
https://github.com/DmxLarchey/PC19

Why in Coq?

- Pros:
  - Constructive means implicit CC
  - Expressive Type theory, spec and proof language
  - Build-in `Extraction` (correct but no yet certified)

Why in Coq?

- Pros:
  - Constructive means implicit CC
  - Expressive Type theory, spec and proof language
  - Build-in `Extraction` (correct but no yet certified)
- Cons:
  - Coq programs have to terminate

$$\texttt{Fixpoint}\ f\ x\ y\ \{\texttt{struct}\ y\} := \ldots f\ y' \ldots$$

  - must ensure $y' <_{\text{struct}} y$ is a strict sub-term
  - $n <_{\text{struct}} \texttt{S}\ n$ but $n/2 \not<_{\text{struct}} n$ (for $n : \texttt{nat}$)
  - look very restrictive (at least to beginners)

Dominique Larchey-Wendling
https://github.com/DmxLarchey/PC19

Why in Coq?

- Pros:
  - Constructive means implicit CC
  - Expressive Type theory, spec and proof language
  - Build-in `Extraction` (correct but no yet certified)
- Cons:
  - Coq programs have to terminate

$$\texttt{Fixpoint}\ f\ x\ y\ \{\texttt{struct}\ y\} := \ldots f\ y' \ldots$$

  - must ensure $y' <_{\text{struct}} y$ is a strict sub-term
  - $n <_{\text{struct}} \mathtt{S}\ n$ but $n/2 \not<_{\text{struct}} n$ (for $n : \texttt{nat}$)
  - look very restrictive (at least to beginners)
- What about?

$$\begin{array}{l} \texttt{let rec itl}\ l\ m := \texttt{match}\ l\ \texttt{with} \\ \quad \mid\ [] \quad\ \ \rightarrow m \\ \quad \mid\ x :: l \rightarrow x :: \texttt{itl}\ m\ l \end{array}$$

Some background material

- Non-constructive recursion:
  - *Termination of Nested and Mutually Recursive Algorithms* (Giesl 97)
  - *Partial and Nested Recursive Function Definitions in Higher-Order Logic* (Krauss 09)
  - *Partiality and Recursion in Interactive Theorem Provers - An Overview* (Bove&Krauss&Sozeau 15)

Some background material

- Non-constructive recursion:
  - *Termination of Nested and Mutually Recursive Algorithms* (Giesl 97)
  - *Partial and Nested Recursive Function Definitions in Higher-Order Logic* (Krauss 09)
  - *Partiality and Recursion in Interactive Theorem Provers - An Overview* (Bove&Krauss&Sozeau 15)
- Constructive recursion:
  - *Modelling general recursion in type theory* (Bove&Capretta 05)
  - *Ten Years of Partiality and General Recursion in Type Theory* (Bove 10)
  - the `Equations` package (Sozeau)
  - our work ITP'18, TYPES'18 (with JF. Monin), MPC'19 (with R. Matthes)

Some background material

- Non-constructive recursion:
  - *Termination of Nested and Mutually Recursive Algorithms* (Giesl 97)
  - *Partial and Nested Recursive Function Definitions in Higher-Order Logic* (Krauss 09)
  - *Partiality and Recursion in Interactive Theorem Provers - An Overview* (Bove&Krauss&Sozeau 15)
- Constructive recursion:
  - *Modelling general recursion in type theory* (Bove&Capretta 05)
  - *Ten Years of Partiality and General Recursion in Type Theory* (Bove 10)
  - the `Equations` package (Sozeau)
  - our work ITP'18, TYPES'18 (with JF. Monin), MPC'19 (with R. Matthes)
- Extraction related:
  - Extraction in Coq (P. Letousey's thesis)
  - MetaCoq and Œuf (CPP'18)

What you will learn, and what you will not

- Techniques in Coq with standard tools:
  - implement spec while controlling CC
  - separate defs. from correctness proofs
  - measure based induction
  - non-terminating algo.
  - nested&mutual non-terminating algo
  - but no co-recursion

What you will learn, and what you will not

- Techniques in Coq with standard tools:
  - implement spec while controlling CC
  - separate defs. from correctness proofs
  - measure based induction
  - non-terminating algo.
  - nested&mutual non-terminating algo
  - but no co-recursion
- We do not use Coq extensions:
  - `Program Fixpoint` for measure induction
  - `Equations` (great to define)
  - not so great to control CC

What you will learn, and what you will not

- Techniques in Coq with standard tools:
  - implement spec while controlling CC
  - separate defs. from correctness proofs
  - measure based induction
  - non-terminating algo.
  - nested&mutual non-terminating algo
  - but no co-recursion
- We do not use Coq extensions:
  - `Program Fixpoint` for measure induction
  - `Equations` (great to define)
  - not so great to control CC
- Practical courses:
  - no constructive recursion theory, no general results
  - no certification of extraction itself
  - but examples, tutorials, exercices
  - control of CC and separation of LC from CC

Take home ideas

- Separate tasks
  - definition of the function in Coq
  - prove its partial correctness
  - prove (partial) termination

Take home ideas

- Separate tasks
  - definition of the function in Coq
  - prove its partial correctness
  - prove (partial) termination
- The algorithm is enough
  - to define the function
  - no need to know why it terminates
  - no need to know what it computes

Take home ideas

- Separate tasks
  - definition of the function in Coq
  - prove its partial correctness
  - prove (partial) termination
- The algorithm is enough
  - to define the function
  - no need to know why it terminates
  - no need to know what it computes
- Extraction
  - erases the Logical Content (LC)
  - keeps the Computational Content (CC)
  - give access to partial algorithms

Content of the courses

- This course: termination & non-structural recursion
  - well-founded and measure based recursion
  - extraction of ∞-loop
  - simple interleaving `itl`
  - merge sort
- 2nd course: non-termination
  - constructive epsilon (Hilbert), unbounded min.
  - cycle detection
  - depth-first search (with ∞ domain)
  - inductive/recursive schemes
- 3rd course: nesting and mutual induction
  - McCarthy's F91, Knuth F91
  - Paulson's normalization
  - Unification

WF recursion via Accessibility predicates

- Structural recursion

$$
\begin{array}{l}
\texttt{Fixpoint fact } n := \texttt{match } n \texttt{ with} \\
\quad \mid \; 0 \quad \Rightarrow 1 \\
\quad \mid \; \texttt{S } n \Rightarrow (\texttt{S } n) * \texttt{fact } n \\
\texttt{end.}
\end{array}
$$

WF recursion via Accessibility predicates

- Structural recursion

$$
\begin{array}{l}
\texttt{Fixpoint fact } n := \texttt{match } n \texttt{ with} \\
\quad \mid \; 0 \quad \Rightarrow 1 \\
\quad \mid \; \texttt{S } n \Rightarrow (\texttt{S } n) * \texttt{fact } n \\
\texttt{end.}
\end{array}
$$

- WF recursion for $R : X \to X \to \texttt{Prop}$ (module Wf)

$$
\begin{array}{l}
\texttt{Inductive Acc } R\ x : \texttt{Prop} := \\
\quad \mid \texttt{Acc\_intro} : (\forall y, R\ y\ x \to \texttt{Acc } R\ y) \to \texttt{Acc } R\ x.
\end{array}
$$

$$
\begin{array}{l}
\texttt{Fixpoint } f\ x\ (A_x : \texttt{Acc } R\ x)\ \{\texttt{struct } A_x\} := \\
\quad body(\ \ldots\ \ f\ y\ A_y\ \ \ldots\ )
\end{array}
$$

WF recursion via Accessibility predicates

- Structural recursion

$$
\begin{array}{l}
\texttt{Fixpoint fact } n := \texttt{match } n \texttt{ with} \\
\quad \mid \texttt{0} \quad \Rightarrow \texttt{1} \\
\quad \mid \texttt{S } n \Rightarrow (\texttt{S } n) * \texttt{fact } n \\
\texttt{end.}
\end{array}
$$

- WF recursion for $R : X \to X \to \texttt{Prop}$ (module `Wf`)

$$
\begin{array}{l}
\texttt{Inductive Acc } R\ x : \texttt{Prop} := \\
\quad \mid \texttt{Acc\_intro} : (\forall y,\ R\ y\ x \to \texttt{Acc } R\ y) \to \texttt{Acc } R\ x.
\end{array}
$$

$$
\begin{array}{l}
\texttt{Fixpoint } f\ x\ (A_x : \texttt{Acc } R\ x)\ \{\texttt{struct } A_x\} := \\
\quad body(\ \ldots \quad f\ y\ A_y \quad \ldots\ )
\end{array}
$$

- Requires:
  - definition of R *before* f, proof of $\forall x, \texttt{Acc } R\ x$
  - ensure $\boxed{A_y <_{\text{struct}} A_x}$

Well-founded fixpoints (module Wf)

- Given a relation R and a predicate P

$$R : X \to X \to \texttt{Prop} \quad \text{and} \quad P : X \to \texttt{Type}$$

- Given F, a functor (or IH) for P wrt R

$$F : \forall x, (\forall y, R\ y\ x \to P\ y) \to P\ x$$

Dominique Larchey-Wendling
https://github.com/DmxLarchey/PC19

Well-founded fixpoints (module Wf)

- Given a relation R and a predicate P

$$R : X \to X \to \texttt{Prop} \quad \text{and} \quad P : X \to \texttt{Type}$$

- Given F, a functor (or IH) for P wrt R

$$F : \forall x, (\forall y, R\ y\ x \to P\ y) \to P\ x$$

- Define the fixpoint of F on $\texttt{Acc}\ R$ (R-wf part of X)

Well-founded fixpoints (module Wf)

- Given a relation R and a predicate P

$$R : X \to X \to \texttt{Prop} \quad \text{and} \quad P : X \to \texttt{Type}$$

- Given F, a functor (or IH) for P wrt R

$$F : \forall x, (\forall y, R\ y\ x \to P\ y) \to P\ x$$

- Define the fixpoint of F on $\texttt{Acc}\ R$ (R-wf part of X)
- Prove $\texttt{Fix\_F} : \forall x, \texttt{Acc}\ R\ x \to P\ x$ by

$$\texttt{Fix\_F}\,x\,A_x := F\ x\ \big(\texttt{fun}\,y\,H_y \Rightarrow \texttt{Fix\_F}\,y\,(\texttt{Acc\_inv}\,A_x\,H_y)\big)$$

Well-founded fixpoints (module Wf)

- Given a relation R and a predicate P

$$R : X \to X \to \texttt{Prop} \quad \text{and} \quad P : X \to \texttt{Type}$$

- Given F, a functor (or IH) for P wrt R

$$F : \forall x, (\forall y, R\ y\ x \to P\ y) \to P\ x$$

- Define the fixpoint of F on $\texttt{Acc}\ R$ (R-wf part of X)
- Prove $\texttt{Fix\_F} : \forall x, \texttt{Acc}\ R\ x \to P\ x$ by

$$\texttt{Fix\_F}\ x\ A_x := F\ x\ \big(\texttt{fun}\ y\ H_y \Rightarrow \texttt{Fix\_F}\ y\ (\texttt{Acc\_inv}\ A_x\ H_y)\big)$$

- Measure: $R_m\ x\ y := m\ x < m\ y$ for $m : X \to \texttt{nat}$
  - then $\texttt{Rwf} : \forall x, \texttt{Acc}\ R_m\ x$ (inverse image of $<$)
- $\texttt{measure\_rect}\ x : P\ x := \texttt{Fix\_F}\ x\ (\texttt{Rwf}\ x)$

Extraction of infinite loops

- The empty proposition `False`:
  - `Inductive False` : `Prop` := .
  - `False_rect` : $\forall X,$ `False` $\rightarrow X$
  - extracts to

```
let false_rect _ = assert false
```

Extraction of infinite loops

- The empty proposition False:
  - Inductive False : Prop := .
  - False_rect : $\forall X$, False $\to X$
  - extracts to

```
let false_rect _ = assert false
```

- Proof $\texttt{False\_elim}_0 : \forall X, \texttt{False} \to X$ using Fix_F
  - define R (_ _ : unit) := True
  - show $A_{\texttt{tt}}$: False $\to$ Acc R tt
  - $\texttt{False\_elim}_0\ X\ C : X := \texttt{Fix\_F} \ldots (A_{\texttt{tt}}\ C)$
  - extracts to

```
let false_elim0 _ =
    let rec fix_F _ = fix_F ()
    in fix_F ()
```

Binary measures for recursion

- Given a measure $m : X \to Y \to \texttt{nat}$
- Given a predicate $P : X \to Y \to \texttt{Type}$
- Given F a functor for P (induction hyp)

$$F : \forall x\, y,\ (\forall x'\, y',\ m\, x'\, y' < m\, x\, y \to P\, x'\, y') \to P\, x\, y$$

Binary measures for recursion

- Given a measure $m : X \to Y \to \mathtt{nat}$
- Given a predicate $P : X \to Y \to \mathtt{Type}$
- Given F a functor for P (induction hyp)

$$F : \forall x\, y,\, (\forall x'\, y',\, m\, x'\, y' < m\, x\, y \to P\, x'\, y') \to P\, x\, y$$

- We define $\mathtt{measure\_rect}_2 : \forall x\, y, P\ x\ y$ and show

$$\begin{aligned}&\mathtt{measure\_rect}_2\ x\ y =\\ &\qquad F\ x\ y\ (\mathtt{fun}\ x'\ y'\ \_ \Rightarrow \mathtt{measure\_rect}_2\ x'\ y')\end{aligned}$$

Binary measures for recursion

- Given a measure $m : X \to Y \to \texttt{nat}$
- Given a predicate $P : X \to Y \to \texttt{Type}$
- Given F a functor for P (induction hyp)

$$F : \forall x\, y,\ (\forall x'\, y',\ m\, x'\, y' < m\, x\, y \to P\, x'\, y') \to P\, x\, y$$

- We define $\texttt{measure\_rect}_2 : \forall x\, y, P\ x\ y$ and show

$$\begin{aligned}&\texttt{measure\_rect}_2\ x\ y = \\ &\qquad F\ x\ y\ (\texttt{fun}\ x'\ y'\ \_ \Rightarrow \texttt{measure\_rect}_2\ x'\ y')\end{aligned}$$

- $\texttt{measure\_rect}_2$ is to be *inlined* at extraction

Binary measures for recursion

- Given a measure $m : X \to Y \to \texttt{nat}$
- Given a predicate $P : X \to Y \to \texttt{Type}$
- Given F a functor for P (induction hyp)

$$F : \forall x\, y,\ (\forall x'\, y',\ m\, x'\, y' < m\, x\, y \to P\, x'\, y') \to P\, x\, y$$

- We define $\texttt{measure\_rect}_2 : \forall x\, y, P\ x\ y$ and show

$$\begin{aligned}&\texttt{measure\_rect}_2\ x\ y = \\ &\qquad F\ x\ y\ (\texttt{fun}\ x'\ y'\ \_ \Rightarrow \texttt{measure\_rect}_2\ x'\ y')\end{aligned}$$

- $\texttt{measure\_rect}_2$ is to be *inlined* at extraction
- To apply $\texttt{measure\_rect}_2$, a user-friendly tactic

$$\boxed{\texttt{induction on}\ x\ y\ \texttt{as}\ IH\ \texttt{with measure}\ m}$$

Simple interleaving

- Goal is to extract

$$
\begin{array}{l}
\texttt{let rec itl}\ l\ m := \texttt{match}\ l\ \texttt{with} \\
\quad \begin{array}{l|lcl}
 & [\,] & \to & m \\
 & x :: l & \to & x :: \texttt{itl}\ m\ l
\end{array}
\end{array}
$$

Simple interleaving

- Goal is to extract

$$\begin{array}{l}\texttt{let rec itl}\ l\ m := \texttt{match}\ l\ \texttt{with} \\ \quad \mid\ [\,]\quad\ \ \to m \\ \quad \mid\ x :: l \to x :: \texttt{itl}\ m\ l\end{array}$$

- We can use $\texttt{itl}_s$ as spec: $\boxed{\texttt{itl}\ l\ m = \texttt{itl}_s\ l\ m}$

$$\begin{array}{l}\texttt{Fixpoint itl}_s\ l\ m := \\ \quad \texttt{match}\ l, m\ \texttt{with} \\ \qquad \mid\ [\,],\quad\ \ \_\qquad \Rightarrow m \\ \qquad \mid\ \_ :: \_, [\,]\quad\ \ \Rightarrow l \\ \qquad \mid\ x :: l, y :: m \Rightarrow x :: y :: \texttt{itl}_s\ l\ m \\ \quad \texttt{end.}\end{array}$$

Simple interleaving

- Goal is to extract

$$\begin{array}{l} \texttt{let rec itl}\ l\ m := \texttt{match}\ l\ \texttt{with} \\ \quad \mid\ [] \quad\ \ \rightarrow m \\ \quad \mid\ x :: l \rightarrow x :: \texttt{itl}\ m\ l \end{array}$$

- We can use $\texttt{itl}_s$ as spec: $\boxed{\texttt{itl}\ l\ m = \texttt{itl}_s\ l\ m}$

$$\begin{array}{l} \texttt{Fixpoint itl}_s\ l\ m := \\ \quad \texttt{match}\ l, m\ \texttt{with} \\ \quad\quad \mid\ [],\ \ \_ \qquad\ \ \Rightarrow m \\ \quad\quad \mid\ \_ :: \_, [] \qquad \Rightarrow l \\ \quad\quad \mid\ x :: l, y :: m \Rightarrow x :: y :: \texttt{itl}_s\ l\ m \\ \quad \texttt{end.} \end{array}$$

- For $\texttt{itl}$, we proceed by induction on $|l| + |m|$

Merge sort (halving)

- Halving a list in two list (structural)

$$
\begin{array}{l}
\texttt{let rec halve } m = \\
\quad \texttt{match } m \texttt{ with} \\
\qquad \mid [\,] \quad \to [\,], [\,] \\
\qquad \mid x :: k \to \texttt{let } l, r = \texttt{halve } k \texttt{ in } (x :: r), l
\end{array}
$$

Merge sort (halving)

- Halving a list in two list (structural)

$$
\begin{array}{l}
\texttt{let rec halve } m = \\
\quad \texttt{match } m \texttt{ with} \\
\qquad \mid [\,] \quad\; \to [\,], [\,] \\
\qquad \mid x :: k \to \texttt{let } l, r = \texttt{halve } k \texttt{ in } (x :: r), l
\end{array}
$$

- What is the spec $\mathbb{G}_{\texttt{halve}}$?

Merge sort (halving)

- Halving a list in two list (structural)

$$
\begin{array}{l}
\texttt{let rec halve } m = \\
\quad \texttt{match } m \texttt{ with} \\
\qquad \mid [\,] \quad \rightarrow [\,], [\,] \\
\qquad \mid x :: k \rightarrow \texttt{let } l, r = \texttt{halve } k \texttt{ in } (x :: r), l
\end{array}
$$

- What is the spec $\mathbb{G}_{\texttt{halve}}$?
  - (l, r) is permutable with m, $m \sim_p l +\!\!+\, r$

Merge sort (halving)

- Halving a list in two list (structural)

$$
\begin{array}{l}
\texttt{let rec halve } m = \\
\quad \texttt{match } m \texttt{ with} \\
\qquad \mid [\,] \quad\;\; \to [\,], [\,] \\
\qquad \mid x :: k \to \texttt{let } l, r = \texttt{halve } k \texttt{ in } (x :: r), l
\end{array}
$$

- What is the spec $\mathbb{G}_{\texttt{halve}}$?
  - (l, r) is permutable with m, $m \sim_p l +\!\!+\, r$
  - l and r have similar size $|r| \leqslant |l| \leqslant 1 + |r|$

Merge sort (halving)

- Halving a list in two list (structural)

$$\begin{array}{l} \texttt{let rec halve } m = \\ \quad \texttt{match } m \texttt{ with} \\ \qquad \mid [\,] \quad \to [\,], [\,] \\ \qquad \mid x :: k \to \texttt{let } l, r = \texttt{halve } k \texttt{ in } (x :: r), l \end{array}$$

- What is the spec $\mathbb{G}_{\texttt{halve}}$?
  - (l, r) is permutable with m, $m \sim_p l +\!\!+ r$
  - l and r have similar size $|r| \leqslant |l| \leqslant 1 + |r|$
  - proved also by structural induction with `Fixpoint`

Merge sort (merging)

- Merging is not structural (induction on $|l| + |m|$)

$$
\begin{array}{l}
\texttt{let rec merge } cmp\ l\ m = \\
\quad \texttt{match } l \texttt{ with} \\
\qquad \mid [\,] \quad\ \ \rightarrow m \\
\qquad \mid x :: l' \rightarrow \\
\qquad\quad \texttt{match } m \texttt{ with} \\
\qquad\qquad \mid [\,] \qquad\ \ \rightarrow l \\
\qquad\qquad \mid y :: m' \rightarrow \texttt{if } cmp\ x\ y \\
\qquad\qquad\qquad\qquad\quad \texttt{then } x :: \texttt{merge } cmp\ l'\ m \\
\qquad\qquad\qquad\qquad\quad \texttt{else } y :: \texttt{merge } cmp\ l\ m'
\end{array}
$$

Merge sort (merging)

- Merging is not structural (induction on $|l| + |m|$)

```
let rec merge cmp l m =
  match l with
   | []      → m
   | x :: l' →
      match m with
       | []      → l
       | y :: m' → if cmp x y
                   then x :: merge cmp l' m
                   else y :: merge cmp l m'
```

- What is the spec $\mathbb{G}_{\texttt{merge}}$?

Dominique Larchey-Wendling
https://github.com/DmxLarchey/PC19

Merge sort (merging)

- Merging is not structural (induction on $|l| + |m|$)

$$
\begin{array}{l}
\texttt{let rec merge}\ cmp\ l\ m = \\
\quad \texttt{match}\ l\ \texttt{with} \\
\qquad |\ []\ \ \ \ \ \to m \\
\qquad |\ x :: l' \to \\
\qquad\quad \texttt{match}\ m\ \texttt{with} \\
\qquad\qquad |\ []\ \ \ \ \ \ \to l \\
\qquad\qquad |\ y :: m' \to \texttt{if}\ cmp\ x\ y \\
\qquad\qquad\qquad\qquad\quad \texttt{then}\ x :: \texttt{merge}\ cmp\ l'\ m \\
\qquad\qquad\qquad\qquad\quad \texttt{else}\ y :: \texttt{merge}\ cmp\ l\ m'
\end{array}
$$

- What is the spec $\mathbb{G}_{\texttt{merge}}$?
  - if l and m are sorted then so is $\texttt{merge}\ cmp\ l\ m$

Merge sort (merging)

- Merging is not structural (induction on $|l| + |m|$)

$$
\begin{array}{l}
\texttt{let rec merge } cmp\ l\ m = \\
\quad \texttt{match } l \texttt{ with} \\
\qquad \mid [\,] \quad \to m \\
\qquad \mid x :: l' \to \\
\qquad\quad \texttt{match } m \texttt{ with} \\
\qquad\qquad \mid [\,] \quad \to l \\
\qquad\qquad \mid y :: m' \to \texttt{if } cmp\ x\ y \\
\qquad\qquad\qquad\qquad \texttt{then } x :: \texttt{merge } cmp\ l'\ m \\
\qquad\qquad\qquad\qquad \texttt{else } y :: \texttt{merge } cmp\ l\ m'
\end{array}
$$

- What is the spec $\mathbb{G}_{\texttt{merge}}$?
  - if l and m are sorted then so is $\texttt{merge}\ cmp\ l\ m$
  - $\texttt{merge}\ cmp\ l\ m \sim_p l +\!\!+\, m$

Merge sort

- Sorting not structural (induction on $|l| + |m|$)

$$
\begin{array}{l}
\texttt{let rec merge\_sort } cmp\ m = \\
\quad \texttt{let } l, r = \texttt{halve } m \\
\quad \texttt{in match } r \texttt{ with} \\
\qquad \mid [\,] \qquad \to m \\
\qquad \mid \_ :: \_ \to \texttt{merge } cmp \\
\qquad\qquad\qquad\qquad (\texttt{merge\_sort } cmp\ l) \\
\qquad\qquad\qquad\qquad (\texttt{merge\_sort } cmp\ m)
\end{array}
$$

Merge sort

- Sorting not structural (induction on $|l| + |m|$)

$$
\begin{array}{l}
\texttt{let rec merge\_sort}\ cmp\ m =\\
\quad \texttt{let}\ l, r = \texttt{halve}\ m\\
\quad \texttt{in match}\ r\ \texttt{with}\\
\qquad |\ []\qquad \to m\\
\qquad |\ \_ :: \_ \to \texttt{merge}\ cmp\\
\qquad\qquad\qquad\qquad (\texttt{merge\_sort}\ cmp\ l)\\
\qquad\qquad\qquad\qquad (\texttt{merge\_sort}\ cmp\ m)
\end{array}
$$

- What is the spec $\mathbb{G}_{\texttt{merge\_sort}}$?

Merge sort

- Sorting not structural (induction on $|l| + |m|$)

$$
\begin{array}{l}
\texttt{let rec merge\_sort } cmp\ m = \\
\quad \texttt{let } l, r = \texttt{halve } m \\
\quad \texttt{in match } r \texttt{ with} \\
\qquad \mid [\,] \quad \to m \\
\qquad \mid \_ :: \_ \to \texttt{merge } cmp \\
\qquad\qquad\qquad\qquad (\texttt{merge\_sort } cmp\ l) \\
\qquad\qquad\qquad\qquad (\texttt{merge\_sort } cmp\ m)
\end{array}
$$

- What is the spec $\mathbb{G}_{\texttt{merge\_sort}}$?
  - $\texttt{merge\_sort}\ cmp\ m$ is sorted

Merge sort

- Sorting not structural (induction on $|l| + |m|$)

$$
\begin{array}{l}
\texttt{let rec merge\_sort}\ cmp\ m = \\
\quad \texttt{let}\ l, r = \texttt{halve}\ m \\
\quad \texttt{in match}\ r\ \texttt{with} \\
\qquad \mid [\,] \quad\ \ \rightarrow m \\
\qquad \mid \_ :: \_ \rightarrow \texttt{merge}\ cmp \\
\qquad\qquad\qquad\qquad (\texttt{merge\_sort}\ cmp\ l) \\
\qquad\qquad\qquad\qquad (\texttt{merge\_sort}\ cmp\ m)
\end{array}
$$

- What is the spec $\mathbb{G}_{\texttt{merge\_sort}}$?
  - $\texttt{merge\_sort}\ cmp\ m$ is sorted
  - $m \sim_p \texttt{merge\_sort}\ cmp\ m$