VERIFIED MEMOIZATION AND DYNAMIC PROGRAMMING

SIMON WIMMER, SHUWEI HU, AND TOBIAS NIPKOW

FAKULTÄT FÜR INFORMATIK,
TECHNISCHE UNIVERSITÄT MÜNCHEN

INTRODUCTION

WHAT WE WANT TO ACHIEVE

INPUT

$$fib\,0 = 1 \qquad fib\,1 = 1$$

$$fib\,(n+2) = fib\,(n+1) + fib\,n$$

PUSH BUTTON

$$fib_{\mathrm{m}}\,0 =_{\mathrm{m}} \langle 0 \rangle \qquad fib_{\mathrm{m}}\,1 =_{\mathrm{m}} \langle 1 \rangle$$

$$fib_{\mathrm{m}}\,(n+2) =_{\mathrm{m}} \textsf{do}\ \{a \leftarrow fib_{\mathrm{m}}\,(n+1);\, b \leftarrow fib_{\mathrm{m}}\,n;\, \langle a+b \rangle\}$$

AUTOMATIC PROOF

$$run\_state\,(fib_{\mathrm{m}}\,n)\,empty = (v, m) \longrightarrow fib\,n = v$$

State monad: functional (red-black tree) or imperative (arrays)

HOW DO WE GET THERE?

HIGH-LEVEL VIEW

1. Monadification: define equation in the state monad

2. Termination: replay termination proof from original function

3. Correspondence: automatically prove that fib and fib_{m} do the same thing via relational parametricity

4. State monads: purely functional and Imperative-HOL heap monad

5. Application: dynamic programming

MONADIFICATION

BASIC CONCEPTS

STATE MONAD

$$\textbf{datatype}\ (\sigma, \alpha)\ state = State\ (run\_state :\ \sigma \to \alpha \times \sigma)$$

from now: σ implicitly fixed $\rightsquigarrow \alpha\ s$

COMBINATORS

$\langle - \rangle$ $\quad$ $return :: \alpha \to \alpha\ s$

$\gg=$ $\quad$ $bind :: \alpha\ s \to (\alpha \to \beta\ s) \to \beta\ s$

$$\langle a \rangle = State\ (\lambda M.\ (a, M))$$

$$s \gg= f = State\ (\lambda M.\ \text{case}\ run\_state\ s\ M\ \text{of}\ (a, M') \Rightarrow run\_state\ (f\ a)\ M')$$

LIFTED FUNCTION APPLICATION

$$f_{\mathsf{m}} \bullet x_{\mathsf{m}} = f_{\mathsf{m}} \gg= (\lambda f.\ x_{\mathsf{m}} \gg= f) = \textbf{do}\ \{f \leftarrow f_{\mathsf{m}}; x \leftarrow x_{\mathsf{m}}; f\ x\}$$

$$fib_{\mathsf{m}}\ (n+2) =_{\mathsf{m}} \langle \lambda x.\ \langle \lambda y.\ \langle x + y \rangle \rangle \rangle \bullet fib_{\mathsf{m}}\ (n+1) \bullet fib_{\mathsf{m}}\ n$$

MONADIFICATION

TYPES

LIFTED TYPE

$$M(\tau) = M'(\tau)\, s$$

$$M'(\tau_1 \to \tau_2) = M'(\tau_1) \to M(\tau_2)$$

$$M'(\tau) = \tau \qquad \text{otherwise}$$

EXAMPLE

$$M((\alpha \to \beta) \to (\alpha\ \mathit{list} \to \beta\ \mathit{list})) =$$

$$((\alpha \to \beta\ s) \to (\alpha\ \mathit{list} \to \beta\ \mathit{list}\ s)\ s)\ s$$

MONADIFICATION

EXAMPLE

INPUT

$$\begin{aligned} map\ f\ [\,] &= [\,] \\ map\ f\ (Cons\ x\ xs) &= Cons\ (f\ x)\ (map\ f\ xs) \end{aligned}$$

RESULT

$$\begin{aligned} map_{\mathsf{m}} &= \langle \lambda f'_{\mathsf{m}}.\ \langle \lambda xs.\ map'_{\mathsf{m}}\ f'_{\mathsf{m}}\ xs \rangle \rangle \\ map'_{\mathsf{m}}\ f'_{\mathsf{m}}\ [\,] &= \langle [\,] \rangle \\ map'_{\mathsf{m}}\ f'_{\mathsf{m}}\ (Cons\ x\ xs) &= Cons_{\mathsf{m}} \bullet (\langle f'_{\mathsf{m}} \rangle \bullet \langle x \rangle) \bullet (map'_{\mathsf{m}} \bullet \langle f'_{\mathsf{m}} \rangle \bullet \langle xs \rangle) \\ Cons_{\mathsf{m}} &= \langle \lambda x.\ \langle \lambda xs.\ \langle Cons\ x\ xs \rangle \rangle \rangle \end{aligned}$$

MONADIFICATION

IN DETAIL

- Set of rewrite rules, applied in a top-down manner (with priorities)
- Maintain mapping from terms to their monadified versions
  Initially: $\Gamma_0 = \{f \mapsto \langle f'_{\mathsf{m}} \rangle, x \mapsto \langle x'_{\mathsf{m}} \rangle\}$ for $f\, x = t \hookrightarrow f'_{\mathsf{m}}\, x'_{\mathsf{m}} = t_{\mathsf{m}}$

$$\frac{e :: \tau_0 \to \tau_1 \to \ldots \to \tau_n \qquad e \text{ is } \Gamma\text{-pure} \qquad \forall i.\ M'(\tau_i) = \tau_i}{\Gamma \vdash e \rightsquigarrow \langle \lambda t_0.\ \langle \lambda t_1.\ \cdots \langle \lambda t_{n-1}.\ e\ t_0\ t_1 \cdots t_{n-1} \rangle \cdots \rangle \rangle}\ \textsc{Pure}$$

$$\frac{\Gamma[x \mapsto \langle x'_{\mathsf{m}} \rangle] \vdash t \rightsquigarrow t_{\mathsf{m}}}{\Gamma \vdash (\lambda x :: \tau.\ t) \rightsquigarrow \langle \lambda x'_{\mathsf{m}} :: M'(\tau).\ t_{\mathsf{m}} \rangle}\ \lambda \qquad \frac{\Gamma \vdash e \rightsquigarrow e_{\mathsf{m}} \qquad \Gamma \vdash x \rightsquigarrow x_{\mathsf{m}}}{\Gamma \vdash (e\ x) \rightsquigarrow (e_{\mathsf{m}} \bullet x_{\mathsf{m}})}\ \textsc{App}$$

$$\frac{g \in \mathsf{dom}\,\Gamma}{\Gamma \vdash g \rightsquigarrow \Gamma(g)}\ \Gamma$$

CORRESPONDENCE PROOF

RELATIONAL PARAMETRICITY

- Memory m consistent with $f :: \alpha \to \beta$:
  if m maps x to r then $f\ x = r$

- Consistency relation

CONSISTENT

$$\Downarrow_R v\ s\ =\ \forall m.\ \mathit{cmem}\ m \longrightarrow$$
$$\text{(case } \mathit{run\_state}\ s\ m \text{ of } (v',\ m') \Rightarrow R\ v\ v' \wedge \mathit{cmem}\ m'\text{)}$$

- Parametricity theorems for the monad combinators, e.g. bind

$$\mathit{bind} :: \alpha\ s \to (\alpha \to \beta\ s) \to \beta\ s$$

$$(\Downarrow_R \dashrightarrow (R \dashrightarrow \Downarrow_S) \dashrightarrow \Downarrow_S)\ (\lambda v\ g.\ g\ v)\ (\gg\!=)$$

$$R \dashrightarrow S = \lambda f\ g.\ \forall x\ y.\ R\ x\ y \longrightarrow S\ (f\ x)\ (g\ y)$$

CORRESPONDENCE PROOF

INDUCTION

- Prove e.g. $((=) \dashrightarrow \Downarrow_{(=)})\ \mathit{fib}\ \mathit{fib}'_{\mathsf{m}}$
  induction (following recursion structure) + parametricity reasoning
- Use rules for elementary combinators + parametricity theorems for previously monadified combinators
- Challenge: automation!
- Main problem: congruence reasoning
  - Function definition command extracts surrounding context for recursive function calls → encoded in congruence rules

CONGRUENCE RULES

EXAMPLE

EXAMPLE

$$\mathit{fib}\, n = 1 + \mathit{sum}\, ((\lambda f.\ \mathit{map}\ f\ [0..n-2])\ \mathit{fib})$$

CONG. RULE

$$\frac{xs = ys \qquad \forall x.\, x \in \mathit{set}\, ys \longrightarrow\ f\, x = g\, x}{(\mathit{map}\ f\ xs) = (\mathit{map}\ g\ ys)}$$

TRANSFER RULE

$$\frac{\mathit{list\_all2}\ \ R\ xs\ ys \qquad R \dashrightarrow \Downarrow_S\ f\ f'_{\mathsf{m}}}{\Downarrow_{\mathit{list\_all2}\ \ S}\ (\mathit{map}\ f\ xs)\ (\mathit{map}_{\mathsf{m}} \bullet \langle f'_{\mathsf{m}} \rangle \bullet \langle ys \rangle)}$$

OUR RULE

$$\frac{xs = ys \qquad \forall x.\ x \in \mathit{set}\, ys \longrightarrow \Downarrow_S\ (f\ x)\ (f'_{\mathsf{m}}\ x)}{\Downarrow_{\mathit{list\_all2}\ \ S}\ (\mathit{map}\ f\ xs)\ (\mathit{map}_{\mathsf{m}} \bullet \langle f'_{\mathsf{m}} \rangle \bullet \langle ys \rangle)}$$

MEMOIZATION
FUNCTIONAL

INTERFACE

$$lookup :: \alpha \to \beta \ option \ s$$
$$update :: \alpha \to \beta \to unit \ s$$

MEMOIZATION

$$\big(f'_{\mathsf{m}}\ x =_{\mathsf{m}} t\big) \ = \ \big(f'_{\mathsf{m}}\ x = retrieve\_or\_run\ x\ t\big)$$

$$\begin{aligned} retrieve\_or\_run\ x\ t \ = \ & lookup\ x \gg\!= \big(\lambda r.\ \text{case } r \text{ of} \\ & Some\ v \Rightarrow \langle v \rangle \\ & \mid None \quad \Rightarrow t \gg\!= (\lambda v.\ update\ x\ v \gg\!= \lambda\_.\ \langle v \rangle)\big) \end{aligned}$$

MEMOIZATION

IMPERATIVE HOL

- Imperative HOL (Bulwahn et al.): Isabelle/HOL framework for reasoning about imperative programs with arrays and references

- Code generation to Haskell, ML, OCaml, and Scala

- Provides **heap monad** to shallowly embed imperative programs = state monad on heaps with failure

  $$\textbf{datatype}\ 'a\ Heap = Heap\ (execute : heap \to ('a \times heap)\ option)$$

- Define consistency relation $\Downarrow_R'$ analogously to $\Downarrow_R$
  Computations may never fail

- Same parametricity theorems for basic combinators
  → correspondence proof still the same

DYNAMIC PROGRAMMING

DYNAMIC PROGRAMMING

AS AN APPLICATION OF MEMOIZATION

- Define dynamic programming algorithms as recursive functions, prove them correct, memoize them
- Choice on:
  - Memory implementation
  - Computation order

DYNAMIC PROGRAMMING

BEYOND MEMOIZATION

- Define dynamic programming algorithms as recursive functions, prove them correct, memoize them
- Choice for space-efficient memoization
  - Memory implementation → LRU cache for last two rows
  - Computation order → Bottom-up computation via iterator
- Case studies: **Bellman-Ford**, CYK, minimum edit distance, …

DYNAMIC PROGRAMMING

BELLMAN-FORD ALGORITHM

- Single-destination shortest path
- Nodes $1, \ldots, n$, target $t \in \{1, \ldots, n\}$, weights $W :: nat \to nat \to int$
- Consider paths in order of increasing path length

$$OPT\ i\ v = Min\ \{weight\ (v \cdot xs)\ t \mid |xs| \leq i \wedge xs \subseteq \{0..n\}\}$$

LENGTH OF SHORTEST PATH FROM v TO t USING AT MOST i EDGES

- No negative cycle → $OPT\ n$ represents shortest path lengths
- Recursion equation

$$OPT\ (Suc\ i)\ v = min\ (OPT\ i\ v)\ (Min\ \{OPT\ i\ w + W\ v\ w \mid w.\ w \leq n\})$$

DYNAMIC PROGRAMMING

BELLMAN-FORD ALGORITHM

$$OPT\ (Suc\ i)\ v = min\ (OPT\ i\ v)\ (Min\ \{OPT\ i\ w + W\ v\ w \mid w.\ w \leq n\})$$

RECURSIVE PROGRAM

$$\begin{aligned} BF\ 0\ j &= (\text{if } t = j \text{ then } 0 \text{ else } \infty) \\ BF\ (Suc\ k)\ j &= min\_list\ (BF\ k,\ j \cdot [W\ j\ i + BF\ k,\ i\ .\ i \leftarrow [0..n]]) \end{aligned}$$

MEMOIZATION

$$BF_m{}'\ 0\ j =_{\mathrm{m}} if_{\mathrm{m}}\ \langle t = j \rangle\ \langle 0 \rangle\ \langle \infty \rangle$$

$$BF_m{}'\ (Suc\ k)\ j =_{\mathrm{m}} \langle \lambda xs.\ \langle min\_list\ xs \rangle \rangle \bullet (\langle \lambda x.\ \langle \lambda xs.\ \langle x \cdot xs \rangle \rangle \rangle \bullet BF_m{}'\ k\ j \bullet (map_{\mathrm{m}} \bullet \langle \lambda i.\ \langle \lambda x.\ \langle W\ j\ i\ +\ x \rangle \rangle \bullet BF_m{}'\ k\ i \rangle \bullet \langle [0..n] \rangle))\ .$$

BOTTOM UP

$$BF\ i\ j = fst\ (run\_state\ (iter\_BF\ (i,\ j) \gg\!= (\lambda\_.\ BF_m{}'\ i\ j))\ Mapping.empty)$$

RELATED WORK

- Monadification
  - Initial inspiration from Haskell (Erwig & Ren '04)
  - Imperative Refinement Framework (Lammich '15)
- Parametric reasoning (Reynolds '83)
  - Isabelle's parametricity reasoner (Huffman & Kuncar '13)
  - Imperative Refinement Framework
  - Schneider & Lochbihler in unpublished work
- Dynamic Programming
  - Manual memoization for e.g. CYK (Bortin '16)
  - Framework for optimising DP algorithms (Itzhaky et al. '16)

CONCLUSION

ONGOING WORK

- More case studies in the AFP: optimal binary search tree, Viterbi algorithm
- Not complete but sufficiently complete for what we encountered
- Future: generalize monadification
  - more monads: reader, writer, ….
    what is the “consistency” relation?
  - insert monadic effects more freely
  - memoize multiple functions at once
  - prove runtime complexity automatically

THANK YOU!

isa-afp.org/entries/Monad_Memo_DP.html