Adjustable References

Viktor Vafeiadis

Max Planck Institute for Software Systems (MPI-SWS)

Introduction

- Reasoning about imperative programs
- How to represent mutable state?
- Two standard approaches
  - Deep embedding
  - Monadic
- What if state is only used as an optimisation?

Memoization

```
let memo f =
    let h = Hashtbl.create() in
    λa. match Hashtbl.get h a with
            | Some b → b
            | None →
                let b = f a in
                Hashtbl.add h a b; b
```

Memoization

```
let memo f =
    let h = Hashtbl.create() in
    λa. match Hashtbl.get h a with
            | Some b → b
            | None →
                let b = f a in
                Hashtbl.add h a b; b
```

Intuitively, memo f = f.

Memoization

```
let memo f =
    let h = Hashtbl.create() in
    λa. match Hashtbl.get h a with
            | Some b → b
            | None →
                let b = f a in
                Hashtbl.add h a b; b
```

But with monads, f : A → B and
memo f : A → State B.

Adjustable References

- Like mutable references,
  but mutation is not observable.
- Internal representation type, R.
- Observation type, T.
- Observation function, $f : R \to T$.

Parameter aref : $\forall R\ T,\ (R \to T) \to \text{Type}$.

Adjustable References (2)

Parameter aval : $\forall R\ T\ (f : R \to T), R \to \text{aref}\ f.$

Parameter aget : $\forall R\ T\ (f : R \to T), \text{aref}\ f \to T.$

Axiom aref_inhabited :
$\forall R\ T\ (f : R \to T)\ (r : \text{aref}\ f),\ \exists v,\ r = \text{aval}\ f\ v.$

Axiom agetval :
$\forall R\ T\ (f : R \to T)\ v,\ \text{aget}(\text{aval}\ f\ v) = f\ v.$

Unobservable Updates

Naively,

$$adjupd : \forall R\ T\ (f : R \to T)\ (r : \text{aref}\ f)\ (v : R),$$
$$f\ v = \text{aget}\ r \to \text{unit}\,.$$

Problems:

1. Evaluation order unknown & Coq can discard unused results
2. The new internal value cannot depend on the old internal value.
3. Not useful unless T is a function type.

Adjusting Reads

Parameter agetu :

$$\forall R\ A\ B\ (f : R \to A \to B)\ (upd : R \to A \to R \times B)$$
$$(PF : \forall x\ a,\ f\ (fst\ (upd\ x\ a)) = f\ x)$$
$$(PF' : \forall x\ a,\ snd\ (upd\ x\ a) = f\ x\ a),$$
$$\text{aref}\ f \to A \to B.$$

Axiom agetuE :

$$\forall R\ A\ B\ (f : R \to A \to B)\ upd\ PF\ PF',$$
$$\text{agetu}\ upd\ PF\ PF' = \text{aget}\ (f := f).$$

Parameter anew :

$$\forall R_1\ T_1\ (f_1 : R_1 \to T_1)\ (r : \text{aref}\ f_1)$$
$$R_2\ T_2\ (f_2 : R_2 \to T_2)$$
$$(g : \forall v, \text{aget}\ r = f_1\ v \to R_2)$$
$$(PF : \forall x\ p_x\ y\ p_y, f_2\ (g\ x\ p_x) = f_2\ (g\ y\ p_y)),$$
$$\text{aref}\ f_2.$$

Axiom anewval :

$$\forall R_1\ T_1\ (f_1 : R_1 \to T_1)\ v\ R_2\ T_2\ (f_2 : R_2 \to T_2)\ g\ PF,$$
$$\text{anew}\ (\text{aval}\ f_1\ v)\ g\ PF$$
$$= \text{aval}\ f_2\ (g\ v\ (\text{agetval}\ f_1\ v)).$$

Trivial; just ignore the adjustments.

Definition aref $R\ T\ f := R$.
Definition aval $R\ T\ f\ v := v$.
Definition aget $R\ T\ f\ r := f\ r$.
Definition agetu $R\ A\ B\ f\ upd\ p\ p'\ r := f\ r$.
Definition anew $R_1\ T_1\ f_1\ r\ R_2\ T_2\ f_2\ g\ p :=$
$g\ r$ eq_refl.

Imperative Implementation

```
type ('r, 't) aref = 'r ref
let aval x = ref x
let aget f r = f !r
let agetu u r a =
    let (v, b) = u !r a in
    r := v;
    b
let anew r g = ref (g !r ())
```

Memoization Using Adjustable References (1)

Section *Memo*.

Variables $(AB : \mathrm{Type})\ (f : A \to B)$.
Variable $eqA : \forall x\ y : A, \{x = y\} + \{x \neq y\}$.
Variable $hash : A \to int$.

Definition $cache :=$
$\{c : Parray.t\ (option(A \times B)) \mid$
$\quad \forall x\ a\ b, Parray.get\ c\ x = Some(a, b) \to b = f\ a\}$.

Memoization Using Adjustable References (2)

```
Program Definition mupd (c : cache) (a : A) :=
  let h := hash a in
  match Parray.get c h with
  | None ⇒
    let b := f a in (Parray.set c h (Some(a, b)), b)
  | Some(a', b) ⇒
    if eqA a a' then (c, b) else
    let b := f a in (Parray.set c h (Some(a, b)), b)
  end.⟨...⟩
Program Definition memo :=
  let r := aval (λc, f) (Parray.create 100 None) in
  agetu mupd _ _ r.⟨...⟩
```

Memoization Using Adjustable References (3)

Lemma *memo_eq* : *memo* = *f*.
Proof.
  by unfold *memo*; **rewrite** arefgetuE, agetval.
Qed.

End *Memo*.

Other examples

Data structures that optimize their shape even when performing read-only operations.

- Union-find path compression (in the paper)
- Splay trees

Limitation

- Cannot have any observable updates.
- But observable updates are fine...
  ...provided that nobody observes them.
- In a sequential setting, you can do a sequence of updates, whose total effect is unobservable.
- cf. Conchon and Filliâtre persistent arrays.

Future work

- Overcome the limitation!

$$\mathsf{runST} : \forall c : \mathsf{AdjStateMonad}\ A.\ c \text{ is logically pure} \rightarrow A\,.$$

- Formally justify the imperative implementation
- Develop applications of adjustable references