CCL – an approach to verifiable multi-valued computations on $\mathbb{R}$

Franz Brauße[1] Norbert Th. Müller[1] Robert Rettinger[2]

Universität Trier, Germany

Hochschule Dortmund, Germany

iRRAM/MPFR/MPC workshop, Schloß Dagstuhl, 2018-04-20

Observations

- C++ not formally verifiable (yet?).
- Programs in `iRRAM` need to run in its C++ loop and throw C++ exceptions.

$\leadsto$ `iRRAM`-algorithms on reals not verifiable.

- tiny imperative language[1], built-in types '`Boolean`', '`Kleenean`', '`Integer`', '`Real`'.
- don't want to think in convergent sequences!
- aim twofold:
  - compile to `iRRAM` $\leadsto$ fast execution
  - compile to Coq $\leadsto$ verified exact real algorithm
- computable constructs $\leadsto$ operations have multi-valued semantics, especially *composition*!
- semantics of expressions e defined through multi-valued function $\vdash_e\colon \Gamma \rightrightarrows \Gamma \times \operatorname{dom} \operatorname{type} e$ directly in Coq

[1] Based on Clerical by Bauer, Park, Simpson,: `https://github.com/andrejbauer/clerical`

Multi-valued functions on $\mathbb{R}$

When computing $f : \mathbb{R} \to \mathbb{R}$, in order to decide on an approximation of $f(x)$, only finitely many bits of x are known. Thus, continuity:

$$\forall x, \varepsilon > 0\, \exists \delta > 0 : f((x \pm \delta)) \subseteq (f(x) \pm \varepsilon)$$

(Computable) way around discontinuities: multi-valued 'functions':

- $f : \mathbb{R} \rightrightarrows A$ is a *relation* on $\mathbb{R} \times A$, really.
- but: composition $(f \circ g)(x)$ only defined iff $\varnothing \neq g(x) \subseteq \operatorname{dom} f$.

Example: Test for x close to zero

E.g. compute $x \leq 2^{-n}$ multi-valued as $x \mapsto \begin{cases} \{1\} & x < 2^{-n}/2 \\ \{0\} & x > 2^{-n} \\ \{0,1\} & x \in [2^{-n}/2, 2^{-n}] \end{cases}$

During computation, just one of the values $\in f(x)$ is selected in an inherently implementation-defined manner.

CCL

- intro variables: ‘`let`’ x ‘`:=`’ e ‘`in`’ f
- loops: ‘`while`’ b ‘`do`’ e
- assignments: x ‘`:=`’ e
- $+$ std arithmetic

- ‘`lim`’ n ‘`=>`’ e
  - n new variable, e an expression depending on n
  - e computes an approx. to accuracy 2^{-n} of limit

- ‘`case`’ b_1 ‘`=>`’ e_1 ‘`||`’ b_2 ‘`=>`’ e_2
  - at least one of b_1, b_2 must be ‘`True`’, otherwise undefined
  - more than one branch ‘`True`’ $\rightsquigarrow$ multi-valued execution semantics
  - every state transition is potentially multi-valued

- Function contracts / expression invariants as annotations
  - translated to proof obligations
  - documentation to users

Limit expression – example: Heron's method

```
{∀x : Coq.Reals.R, abs x = Coq.Reals.Rabs x}
external abs(Real) -> Real
external bounded(Real, Int) -> Bool

{∀x : Coq.Reals.R, x ≥ 0 ⇒ sqrt x = Coq.Reals.Rsqrt x}
function sqrt(Real x):
    lim n =>
        var y := 1.0 in (
        var z := x/y in (
            while !bounded(abs(y-z), n)
            do (
                y := (y+z)/2.0;
                z := x/y
            )
        ); y)

do
    sqrt(2.0)
```

Case expression – intro multi-valuedness

- Multi-valued test for x bounded by 2^{-n}:

  {$\forall\ x\ n,\ x < (/\ 2\ n)\ \Rightarrow\ \texttt{bounded}\ x\ n = \text{‘}\texttt{True}\text{’}$}

```
function bounded(Real x, Int n):
    var eps := 2.0^real(-n) in
        case  x < eps     => True
        ||    x > eps/2.0 => False
        end
```

- single-valued absolute value via limit of a multi-valued sequence

  {$\forall\ x : \texttt{Coq.ZArith.Z}, \texttt{real}\ x = \texttt{Coq.Reals.IZR}\ x$}

```
external real(Int) -> Real
```

  {$\forall\ x,\ \texttt{abs}\ x = \texttt{Coq.Reals.Rabs}\ x$}

```
function abs(Real x):
    lim n =>
        var eps := 2.0^real(-n) in
        case x <  eps => -x
        ||   x > -eps =>  x
        end
```

[2] IZR injects Coq's integers into Coq's axiomatic Reals

Thank you!

Questions? Remarks? Comments? Weird approach? Suggestions?

CCL $\rightarrow$ iRRAM: `https://github.com/fbrausse/cclerical`

```
(** Multi-valued functions **)
Definition mv_f (A B : Type) := A -> B -> Prop.

Definition mv_compose {A B C : Type} (f : mv_f A B) (g : mv_f B C)
  : mv_f A C
  := fun a c => (forall b : B, f a b -> exists c : C, g b c) /\
               exists b : B, f a b /\ g b c.

(** vdash relates "previous" contexts to
    "later" contexts and a result value **)
Definition vdash {sigma : scope} {t : ccl_type} :=
  context sigma -> context sigma -> ccl_dom t -> Prop.

Definition case {sigma : scope} {t : ccl_type}
               (b0 b1 : @vdash sigma Kleenean)
               (e0 e1 : @vdash sigma t)
  : vdash := fun g g' v => b0 g g (kl_bool true) /\ e0 g g' v \/
                          b1 g g (kl_bool true) /\ e1 g g' v.

Definition lim {sigma : scope} {t : ccl_type} (id : string)
              (e : @vdash (mk_scope_record id Integer :: sigma) t)
  := fun g g' v
     => g = g' /\
        let (d,_) := ccl_met t in
        let s := fun n w => let h := (g,Z.of_nat n) in e h h w in
        mv_total s /\ mv_converges_fast (ccl_dom t) d s v.
```