Type-Based Termination, Inflationary Fixed-Points, and Mixed Inductive-Coinductive Types

Andreas Abel

Department of Computer Science
Ludwig-Maximilians-University Munich

Fixed Points in Computer Science (FICS 2012)
ETAPS 2012, Tallinn, Estonia
24 March 2012

Aspects of Termination

What the talk is about:

- ✓ foundational approach to termination
- ✓ types and semantics
- ✓ compositional termination
- ✓ communicating termination (across function/module boundaries)
- ✓ `MiniAgda`

What the talk is not about:

- ✗ smart termination orders
- ✗ automatic termination inference

Well-typed programs don't go wrong

- Desired: absence of run-time errors (don't go wrong).
- Property not compositional!
- If f and a don't go wrong, $f\ a$ might still!
- Milner: introduce types to "strengthen induction hypothesis".
- Polymorphic types allow to abstract out code $t[u] \rightsquigarrow \text{let } x = u \text{ in } t$.
- Types make error-freeness compositional!

Well-typed programs terminate

- Desired: termination (or stream productivity).
- Termination is not compositional!
- If f and a terminate, $f\ a$ might still diverge!
  (E.g. $f = a = \lambda x.\, x\, x$)
- Welltyped programs terminate!?
  - ✓ Simply-typed lambda-calculus
  - ✓ Polymorphic lambda-calculus (System F)
  - ✗ Haskell (has recursion)
  - ✗ Agda, Coq (have separate termination checking)
- What is the problem with a separate termination check?

A simple, terminating function

- Picks every other element from a list.

```
fun everyOther : List A → List A
{ everyOther nil                    = nil
; everyOther (cons a nil)           = nil
; everyOther (cons a (cons a' as))  = cons a (everyOther as)
}
```

- Terminates, since `as` $<$ `cons a (cons a' as)`.
- Abstract out 0-1-many case distinction:

```
fun zeroOneMany : List A → C → (A → C) → ...
{ zeroOneMany nil                    z o m = z
; zeroOneMany (cons a nil)           z o m = o a
; zeroOneMany (cons a (cons a' as))  z o m = m a a' as
}
```

Abstractions not supported

- Function using combinator `zeroOneMany`.

```
fun everyOther : List A → List A
{ everyOther l = zeroOneMany l
    nil
    (λ a        → nil)
    (λ a a' as → cons a (everyOther as))
}
```

- Terminating? Relation between `as` and `l` lost.
- Inlining `zeroOneMany`?
  - ✗ Bad performance of checker.
  - ✗ Source code might not be available.
- Trouble with abstraction? Types to the rescue!

Type-based termination

- Refine types: `List A i` contains lists up to length `i`.
- A precise type with bounded universal `[j < i] → ...`

```
fun zeroOneMany : List A i →
  ...
  (many : [j < i] → A → A → List A j → C) → C
```

- Relation between `as : List A j` and `l : List A i` tracked by types!

```
fun everyOther : List A i → List A i
{ everyOther l = zeroOneMany l
    ...
    (λ a a' as → cons a (everyOther as))
}
```

Summary: Type-based termination

- Type-based termination is compositional.
- Need only type, not code, of used functions.
- Module-wise termination check.
- Little overhead to classic type checking.
- Strength depends on language of sizes.
- Here: foundational concept of size...

Sizes as iteration stages

- Inductive types are least fixed points.
- $\mathsf{List}\, A \cong \mu F$ with $F\, X = \top + A \times X$.
- Approximating the fixed-point from below:

$$\begin{array}{lcl} \mu^0 \; F & = & \bot \\ \mu^{\alpha+1} \; F & = & F\, (\mu^\alpha F) \\ \mu^\lambda \; F & = & \bigcup_{\alpha<\lambda} \mu^\alpha F \end{array}$$

- $\mathsf{List}\, A\, i = \mu^i F$ gives lists of length $< i$.
- For monotone F it holds that

$$\mu^\alpha F = \bigcup_{\beta<\alpha} F\, (\mu^\beta F)$$

Recursion principle

- Transfinite recursion on sizes:

$$\frac{f : \forall i.\, A\, i \to A\,(i+1)}{\mathsf{fix}\, f : \forall i.\, A\, i}$$

- Base case: $A\,0 = \top$.
- Limit case: $\bigcap_{\alpha<\lambda} A\,\alpha \subseteq A\,\lambda$.
- Typical use: $A\,i = \mu^i F \to C\,i$.
- Basis of almost all work on type-based termination.
- Can we avoid the side conditions?

Inflationary least fixed points

- Take proven equation as definition of $\mu^\alpha F$!

$$\mu^\alpha F = \bigcup_{\beta<\alpha} F\ (\mu^\beta F)$$

- Irrelevant: Reaches fixed point also for non-monotone F.
- Relevant: No case on 0, $\_ + 1$, and λ (limit).
- Sizes α, β need not be classical ordinals.
- Allows recursive definition of inductive types:

```
List A i = [j < i] & Maybe (A & List A j)
```

Bounded existential `[j < i] & ...` and cartesian product `A & B`.

Recursion principle

- Well-founded recursion on sizes:

$$\frac{f : \forall i.\, (\forall j < i.\, A\, j) \to A\, i}{\mathrm{fix}\, f : \forall i.\, A\, i}$$

- No conditions on A!
- Definable in `MiniAgda`:

```
cofun fix : ([i : Size] → ([j < i] → A j) → A i) →
            [i : Size] → A i
{ fix f i = f i (λ j → fix f j)
}
```

Inflationary greatest fixed points

- Coinductive types are greatest fixed points.

$$\nu^{\alpha} F = \bigcap_{\beta < \alpha} F\ (\nu^{\beta} F)$$

- Stream $A\,i = \nu^{i} F$ with $F\,X = A \times X$
- Stream $A\,i$ are streams of depth i.
- Can be unrolled safely up to i times.

```
Stream A i = [j < i] → A & Stream A j
```

Programming streams

- Deconstructing and constructing streams:

```
Stream A i = [j < i] → A & Stream A j

let tail [i : Size] (s : Stream A (i+1)) : Stream A i
  = case (s i) { (a, as) → as }

cofun repeat (a : A) [i : Size] → Stream A i
{ repeat a i = λj → (a, repeat a j)
}
```

- **repeat** is productive because $j < i$.

The famous Fibonacci stream

- Zipping two streams with function f.

```
cofun zipWith : [i : Size] → (A → B → C) →
                Stream A i → Stream B i → Stream C i
{ zipWith i f sa sb = λj →
    case (sa j, sb j)
    { ((a, as), (b, bs)) → (f a b, zipWith j f as bs)
    }
}
```

- Fibonacci stream $0, 1, 1, 2, 3, 5, 8, 13, \ldots$

```
cofun fib : [i : Size] → Stream Nat i
{ fib i = λ j → (0,
          λ k → (1,
          zipWith k add
            (fib k)
            (tail k (fib j))))
}
```

Mixed Induction-Coinduction

- Classification of recursive data types
  - Inductive μ (lists, trees, Brouwer ordinals)
  - Coinductive ν (streams, processes)
  - Coinductive-inductive $\nu\mu$ (stream processors)
  - Other mixes...
- How do mixed types fit into our framework?

Stream processors

- Stream processors [Ghani, Hancock, Patterson] code continuous maps on streams.

```
data SP a b = Get (a → SP a b)
            | Put b (SP a b)
```

- `Get`: We can either read an `a` from the input stream and enter a new state depending on `a`, or
- `Put`: write a `b` on the output stream and enter a new state.
- `run` executes a `SP`.

```
run :: SP a b → [a] → [b]
run (Get f) (a : as)  = run (f a) as
run (Put b sp)   as   = b : run sp as
```

Stream processors (cont.)

- Continuity: An output must appear after finite input.
  - ✗ No infinite succession of `Gets`.
  - ✓ Infinite sequence of `Puts` possible.

$$\frac{f : A \to \mathrm{SP}}{\mathrm{get}\ f : \mathrm{SP}} < \omega \qquad \frac{b : B \qquad sp : \mathrm{SP}}{\mathrm{put}\ b\ sp : \mathrm{SP}} \leq \omega$$

- Model `SP` by nesting μ into ν.

$$\mathrm{SP} = \nu X.\mu Y.\,(A \to Y) \times (B \times X)$$

- We can restart getting after a put.

$$\mathrm{SP} = \mu Y.\,(A \to Y) \times (B \times \mathrm{SP})$$

Lexicographic recursion

- Nested inflationary fixed-points:

$$\mathsf{SP}\ \alpha\ \beta = \bigcap_{\alpha'<\alpha} \bigcup_{\beta'<\beta} (A \to \mathsf{SP}\ \alpha\ \beta') + (B \times \mathsf{SP}\ \alpha'\ \infty)$$

- ∞ is closure ordinal.
- Type defined by lexicographic recursion.
- Pushing quantifiers in:

$$\mathsf{SP}\ \alpha\ \beta = (A \to \bigcup_{\beta'<\beta} \mathsf{SP}\ \alpha\ \beta') + (B \times \bigcap_{\alpha'<\alpha} \mathsf{SP}\ \alpha'\ \infty)$$

- Coinductive occurrence prefixed by universal/existential.
- Sizing scheme ((α', ∞) vs. (α, β')) represents nesting $\nu\mu$.

Stream Processors in MiniAgda

- Type def. and pattern synonyms:

```
SP i j = Either (A → [j' < j] & SP i j')
                (B & ([i' < i] → SP i' #))
pattern get f    = left f
pattern put b sp = right (b, sp)
```

- run defined by lexicographic recursion over i, j.

```
cofun run : [i, j : Size] → SP i j →
            Stream A # → Stream B i
{ run i j (get f)     as = case f (head # as)
                          { (j', sp) → run i j' sp (tail # as) }
; run i j (put b sp)  as = λi' → (b, run i' # (sp i') as)
}
```

Wrapping up

- Type-based termination is compositional and local.
- Inflationary iteration provides simple foundation.
- (Co)induction is replaced by well-founded recursion on size.
- Mixed types just fall in our lap.

There is more to it

- Size language has $0, +1, \infty, \max$ and possibly $+$.
- Bounded quantification induces subtyping, e.g.:

$$\mathsf{List}\, A\, i = \bigcup_{j<i} (\top + A \times \mathsf{List}\, A\, j) \text{ covariant in } A \text{ and } i$$

- Size is tree height/depth.
  Other size assignments!?
- Termination measures are lexicographic products of sizes.
  What else do we need?
- Most sizes are inferable.
  Integrate size solving with higher-order unification!

Termination and Metavariables

- Agda: a dependently typed language with interactive proof/program development.
- Metavariables stand for missing code
- ... filled in by Agda or the user.
- Only well-typed solutions accepted.
- Easy, because type checking is local.
- Global properties like positivity and termination not checked.
- Agda refuses to solve recursive metas; solution might be diverging.
  $\Longrightarrow$ Integrate all static checks into type system!
  $\Longrightarrow$ Smaller implementation, no corner cases, orthogonality.

Related Work

- Type-based termination (transfinite recursion): see paper.
- Circular proofs (well-founded recursion):
  Dam, Sprenger, Simpson, Schoepp
- Certifying termination proofs
- Coinduction a la Nakano: Atkey, Birkedal et al.