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 $t[u] ⇝ \text{let } x = u \text{ in } t$.
- 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. \( f = a = \lambda x. x x \))
- Well-typed 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.
  
  ```haskell
  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:
  
  ```haskell
  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`.

```plaintext
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: \texttt{List A i} contains lists up to length \texttt{i}.
- A precise type with bounded universal \([ j < i ] \rightarrow \ldots \)

  \[
  \textbf{fun} \; \text{zeroOneMany} : \text{List A i} \rightarrow \ldots \\
  \quad (\text{many} : [j < i] \rightarrow A \rightarrow A \rightarrow \text{List A j} \rightarrow C) \rightarrow C
  \]

- Relation between \texttt{as} : \text{List A j} and \texttt{l} : \text{List A i} tracked by types!

  \[
  \textbf{fun} \; \text{everyOther} : \text{List A i} \rightarrow \text{List A i} \\
  \quad \{ \text{everyOther} \; l = \text{zeroOneMany} \; l \\
  \quad \quad \ldots \\
  \quad \quad (\lambda \; a \; a' \; \text{as} \rightarrow \text{cons} \; a \; (\text{everyOther} \; \text{as}) ) \}
  \]
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.
- List $A \cong \mu F$ with $F X = \top + A \times X$.
- Approximating the fixed-point from below:
  \[
  \begin{align*}
  \mu^0 F & = \perp \\
  \mu^{\alpha+1} F & = F (\mu^\alpha F) \\
  \mu^\lambda F & = \bigcup_{\alpha < \lambda} \mu^\alpha F 
  \end{align*}
  \]
- 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:
  \[
  f : \forall i. A i \rightarrow A(i + 1) \\
  \text{fix } f : \forall i. A i
  \]

- Base case: \( A 0 = \top \).
- Limit case: \( \bigcap_{\alpha < \lambda} A \alpha \subseteq A \lambda \).
- Typical use: \( A i = \mu^i F \rightarrow C i \).
- 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 $\lambda$ (limit).
- Sizes $\alpha, \beta$ 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:
  \[
  f : \forall i. (\forall j < i. A j) \rightarrow A i \\
  \text{fix } f : \forall i. A i
  \]

- No conditions on \( A \)!

- Definable in MiniAgda:

```agda
cofun fix : ([i : Size] \rightarrow ([j < i] \rightarrow A j) \rightarrow A i) \rightarrow 
  [i : Size] \rightarrow A i

{ fix f i = f i (\lambda j \rightarrow \text{fix } f j) }
```

Andreas Abel (LMU)
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.

\[ \text{Stream } A \, i = [j < i] \to A \& \text{Stream } A \, j \]
Programming streams

- Deconstructing and constructing streams:

  \[ \text{Stream } A \ i = [j < i] \rightarrow A \ & \text{Stream } A \ j \]

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

  \[
  \text{cofun repeat } (a : A) [i : \text{Size}] \rightarrow \text{Stream } A \ i \\
  \{ \text{repeat } a \ i = \lambda j \rightarrow (a, \text{repeat } a \ j) \}
  \]

- \text{repeat} is \text{productive} because \( j < i \).
The famous Fibonacci stream

- Zipping two streams with function $f$.
  
  \[
  \text{cofun} \ \text{zipWith} : [i : \text{Size}] \to (A \to B \to C) \to \\
  \quad \text{Stream A i} \to \text{Stream B i} \to \text{Stream C i}
  \]
  
  \[
  \{
  \text{zipWith} i f sa sb = \lambda j \to \\
  \quad \text{case} \ (sa j, sb j) \\
  \quad \{ \ ((a, as), (b, bs)) \to (f a b, \text{zipWith} j f as bs) \\
  \quad \}
  \}
  \]

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

  \[
  \text{cofun} \ \text{fib} : [i : \text{Size}] \to \text{Stream Nat i}
  \]
  
  \[
  \{ \ \text{fib} i = \lambda j \to (0, \\
  \quad \lambda k \to (1, \\
  \quad \text{zipWith} k \text{add} \\
  \quad \ (fib k) \\
  \quad \ (\text{tail} k \ (\text{fib} j)))
  \}
  \]
Mixed Induction-Coinduction

- Classification of recursive data types
  - Inductive $\mu$ (lists, trees, Brouwer ordinals)
  - Coinductive $\nu$ (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.

```haskell
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`.

```haskell
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.

\[
\begin{align*}
f : A &\rightarrow SP < \omega \\
g &\text{get } f : SP < \omega \\
b : B &\rightarrow sp : SP < \omega \\
 &\text{put } b sp : SP \leq \omega
\end{align*}
\]

- Model \(SP\) by nesting \(\mu\) into \(\nu\).

\[
SP = \nu X. \mu Y. (A \rightarrow Y) \times (B \times X)
\]

- We can restart getting after a put.

\[
SP = \mu Y. (A \rightarrow Y) \times (B \times SP)
\]
Lexicographic recursion

- Nested inflationary fixed-points:

\[
\text{SP } \alpha \beta = \bigcap_{\alpha' < \alpha, \beta' < \beta} \bigcup (A \rightarrow \text{SP } \alpha \beta') + (B \times \text{SP } \alpha' \infty)
\]

- $\infty$ is closure ordinal.
- Type defined by lexicographic recursion.
- Pushing quantifiers in:

\[
\text{SP } \alpha \beta = (A \rightarrow \bigcup_{\beta' < \beta} \text{SP } \alpha \beta') + (B \times \bigcap_{\alpha' < \alpha} \text{SP } \alpha' \infty)
\]

- Coinductive occurrence prefixed by universal/existential.
- Sizing scheme $((\alpha', \infty) \text{ vs. } (\alpha, \beta'))$ represents nesting $\nu\mu$. 
Stream Processors in MiniAgda

- **Type def. and pattern synonyms:**

  \[ SP \ i \ j = \textsf{Either} \ (A \rightarrow \ [j' < j] \ & \ SP \ i \ j') \]
  \[ \quad (B \ & \ ([i' < i] \rightarrow \ SP \ i' \ #)) \]

  \textbf{pattern get} \ f \ = \ \textsf{left} \ f \\
  \textbf{pattern put} \ b \ sp \ = \ \textsf{right} \ (b, \ sp) \\

- **run** defined by lexicographic recursion over \(i,j\).

  \textbf{cofun} \ run : [i, j : \textsf{Size}] \rightarrow \ SP \ i \ j \rightarrow \\  \quad \text{Stream A #} \rightarrow \text{Stream B i} \\
  \{ \ \text{run \ i \ j \ (get \ f)} \ as = \ \textbf{case} \ f \ (\text{head} \ # \ as) \\
  \quad \{ \ (j', \ sp) \rightarrow \ \text{run \ i \ j' \ sp} \ (\text{tail} \ # \ as) \} \}
  ; \ \text{run \ i \ j \ (put \ b \ sp)} \ as = \ \lambda i' \rightarrow \ (b, \ \text{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 $\oplus$.
- Bounded quantification induces subtyping, e.g.:

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

- 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 Metavariabls

- Agda: a dependently typed language with interactive proof/program development.
- Metavariabls 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.
  ➞ Integrate all static checks into type system!
  ➞ 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.