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

Andreas Abel
Department of Computer Science
Ludwig-Maximilians-University Munich, Germany
andreas.abel@ifi.lmu.de

Type systems certify program properties in a compositional way. From a bigger program one can abstract out a part and certify the properties of the resulting abstract program by just using the type of the part that was abstracted away. *Termination* and *productivity* are non-trivial yet desired program properties, and several type systems have been put forward that guarantee termination, compositionally. These type systems are intimately connected to the definition of least and greatest fixed-points by ordinal iteration. While most type systems use "conventional" iteration, we consider inflationary iteration in this article. We demonstrate how this leads to a more principled type system, with recursion based on well-founded induction. The type system has a prototypical implementation, MiniAgda, and we show in particular how it certifies productivity of corecursive and mixed recursive-corecursive functions.

1 Introduction: Types, Compositionality, and Termination

While basic types like *integer*, *floating-point number*, and *memory address* arise on the machine-level of most current computers, higher types like function and tuple types are abstractions that classify values. Higher types serve to guarantee certain good program behaviors, like the classic "don't go wrong" absence of runtime errors [Mil78]. Such properties are usually not compositional, i. e., while a function f and its argument a might both be well-behaved on their own, their application $f\,a$ might still go wrong. This issue also pops up in termination proofs: take $f = a = \lambda x.xx$, then both are terminating, but their application loops. To be compositional, the property *terminating* needs to be strengthened to what is often called *reducible* [Gir72] or *strongly computable* [Tai67], leading to a semantic notion of type. While the bare properties are not compositional, *typing* is.

Type *polymorphism* [Rey74, Gir72, Mil78] has been invented for compositionality in the opposite direction: We want to decompose a larger program into smaller parts such that the well-typedness of the parts imply the well-typedness of the whole program. Consider $(\lambda x.x)\,(\lambda x.x)\,\mathsf{true}$, a simply-typed program which can be abstracted to $\mathsf{let\ id} = \lambda x.x\ \mathsf{in\ id\ id\ true}$. The two occurrences of id have different type, namely $\mathsf{Bool} \to \mathsf{Bool}$ and $(\mathsf{Bool} \to \mathsf{Bool}) \to \mathsf{Bool} \to \mathsf{Bool}$, and the easiest way to type check the new program is to just inline the definition of id. This trick does not scale, however, making type checking infeasible and separate compilation of modules impossible. The accepted solution is to give id the polymorphic type $\forall X.\, X \to X$ which can be instantiated to the two required types of id.

Termination checking, if it is to scale to software development with powerful abstractions, needs to be compositional. Just like for other non-standard analyses, e. g., strictness, resource consumption and security, type-based termination promises to be a model of success. Current termination checkers, however, like foetus [AA02, Wah00, AD10], the one of Agda [Nor07], and Coq's guardedness check [Gim95, Bar10b] are not type-based, but syntactic. Let us see how this affects compositionality. Consider the following recursive program defined by pattern matching. We use the syntax of MiniAgda [Abe10], in this and all following examples.

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

The polymorphic function `everyOther` returns a list consisting of every second element of the input list. Since the only recursive call happens on sublist `as` of the input list `cons a (cons a' as)`, termination is evident. We say that the call argument decreases in the *structural order*; this order, plus lexicographic extensions, is in essence the termination order accepted by the proof assistants Agda, Coq, and Twelf [Pie01].

The function distinguishes on the empty list, the singleton list, and lists with at least 2 elements. Such a case distinction is used in list sorting algorithms, too, so we may want to abstract it from `everyOther`.

```
fun zeroOneMany : [A : Set] → List A → [C : Set] →
  (zero : C) →
  (one  : A → C) →
  (many : A → A → List A → C) →
  C
{ zeroOneMany A nil                    C zero one many = zero
; zeroOneMany A (cons a nil)           C zero one many = one a
; zeroOneMany A (cons a (cons a' as))  C zero one many = many a a' as
}
```

After abstracting away the case distinction, termination is no longer evident; the program is rejected by Agda's termination checker foetus.

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

Whether the recursive call argument `as` is structurally smaller than the input `l` depends on the definition of `zeroOneMany`. In such situations, Coq's guardedness check may inline the definition of `zeroOneMany` and succeed. Yet in general, as we have discussed in the context of type checking, inlining definitions is expensive, and in case of recursive definitions, incomplete and brittle. Current Coq [INR10] may spend minutes on checking a single definition, and fail nevertheless.

Type-based termination can handle abstraction as in the above example, by assigning a more informative type to `zeroOneMany` that guarantees that the list passed to `many` is structurally smaller than the list analyzed by `zeroOneMany`. Using this restriction, termination of `everyOther` can be guaranteed. To make this work, we introduce a purely administrative type **`Size`** and let variables `i`, `j`, and `k` range over **`Size`**. The type of lists is refined as `List A i`, meaning lists of length $< i$. We also add bounded size quantification $\bigcap_{j<i} T(j)$, in concrete syntax `[j < i] → T j`, which lets `j` only be instantiated to sizes strictly smaller than `i`. The refined type of `zeroOneMany` thus becomes:

FICS 2010
EPTCS ??, 20??, pp. 1–11, doi:10.4204/EPTCS.??.??

```
fun zeroOneMany : [A : Set] → [i : Size] → List A i → [C : Set] →
  (zero : C) →
  (one  : A → C) →
  (many : [j < i] → A → A → List A j → C) →
  C
```

The list passed to `many` is bounded by size `j`, which is strictly smaller than `j`. This is exactly the information needed to make `everyOther` termination-check.

Barthe et. al. [BGP06] study type-based termination as an automatic analysis "behind the curtain", with no change to the user syntax of types. Size quantification is restricted to rank-1 quantifiers, known as ML-style quantification [Mil78]. This excludes the type of `zeroOneMany`, which has a rank-2 (bounded) quantification. Higher-rank polymorphism is not inferable automatically, yet without it we fall short of our aim: compositional termination. Anyway, the prerequisite for inference is the availability of the source code, which fails for abstract interfaces (such as parametrized modules in Agda, Coq, or ML). Thus, we advocate a type system with explicit size information based on the structural order. It will be presented in the remainder of this article.

2 Sizes, Iteration, and Fixed-Points

In the following, rather than syntactic we consider semantic types such as sets of terminating terms. We assume that types form a complete lattice $(\mathscr{T}, \subseteq, \bigcap, \bigcup)$ with least element $\bot$ and greatest element $\top$. Further, let the usual type operators $+$ (disjoint sum), $\times$ (Cartesian product), and $\to$ (function type) have a sensible definition.

Inductive types μF, such as $\mathsf{List}\,A$, are conceived as least fixed points of monotone type constructors F, for lists this being $FX = \top + A \times X$. Constructively [CC79], least fixed points are obtained on a $\cup$-semilattice by ordinal iteration up to a sufficiently large ordinal γ. Let $\mu^\alpha F$ denote the αth *iterate* or *approximant*, which is defined by transfinite recursion on α:

$$\begin{array}{lcll} \mu^0 F & = & \bot & \text{zero ordinal: least element of the lattice} \\ \mu^{\alpha+1} F & = & F(\mu^\alpha F) & \text{successor ordinal: iteration step} \\ \mu^\lambda F & = & \bigcup_{\alpha<\lambda} \mu^\alpha F & \text{limit ordinal: upper limit} \end{array}$$

For monotone F, iteration is monotone, i. e., $\mu^\alpha F \subseteq \mu^\beta F$ for $\alpha \le \beta$. At some ordinal γ, which we call *closure ordinal* of this inductive type, we have $\mu^\alpha F = \mu^\gamma F$ for all $\alpha \ge \gamma$—the chain has become stationary, the least fixed point has been reached. For polynomial F, i. e., those expressible without a function space, the closure ordinal is ω. The index α to the approximant $\mu^\alpha F$ is a strict upper bound on the *height* of the well-founded trees inhabiting this type; in the case of lists (which are linear trees) it is a strict upper bound on the length.

Dually, coinductive types νF are constructed on a $\cap$-semilattice by iteration from above.

$$\begin{array}{lcll} \nu^0 F & = & \top & \text{zero ordinal: greatest element of the lattice} \\ \nu^{\alpha+1} F & = & F(\nu^\alpha F) & \text{successor ordinal: iteration step} \\ \nu^\lambda F & = & \bigcap_{\alpha<\lambda} \nu^\alpha F & \text{limit ordinal: lower limit} \end{array}$$

Iteration from above is antitone, i. e., $\nu^\alpha F \supseteq \nu^\beta F$ for $\alpha \le \beta$. The chain of approximants starts with the all-type $\top$ and descends towards the greatest fixed-point νF. In case of the above F this would be $\mathsf{CoList}\,A$, the type of possibly infinite lists over element type A. The index α in the approximant $\nu^\alpha F$

could be called the *depth* of the non-well-founded trees inhabiting this type. It is a lower bound on how deep we can descend into the tree before we hit undefined behavior ($\top$).

The central idea of type-based termination, going all the way back to Mendler [Men87], Hughes, Pareto, and Sabry [HPS96], Giménez [Gim98], and Amadio and Coupet-Grimal [ACG98] is to introduce syntax to speak about approximants in the type system. Common to the more expressible systems, such as Barthe et. al. [BGR08a] and Blanqui [Bla04] is syntax for ordinal variables i, ordinal successor $\mathtt{s}a$ (MiniAgda: `$a`), closure ordinal ∞ (MiniAgda: `#`) and data type approximants D^a (MiniAgda: e. g., `List A i`). Hughes et. al. and the author [Abe08b] have also quantifiers $\forall i.\, T$ over ordinals (MiniAgda: `[i : Size] → T`).

How do we get a recursion principle from approximants? Consider the simplest example: constructing an infinite repetition r of a fixed element a by corecursion. After assembling the colist-constructor $\mathsf{cons} : A \to \mathsf{CoList}\,A\,i \to \mathsf{CoList}\,A\,(i+1)$ on approximants, we give a recursive equation $r = \mathsf{cons}\,a\,r$ with the following typing of the r.h.s.

$$i : \mathsf{Size},\ r : \mathsf{CoList}\,A\,i \vdash \mathsf{cons}\,a\,r : \mathsf{CoList}\,A\,(i+1)$$

The types certify that each unfolding of the recursive definition of r increases the number of produced colist elements by one, hence, in the limit we obtain an infinite sequence and, in particular, r is productive. Our example is a special instance of the recursion principle of type-based termination, expressible as type assignment for the fixpoint combinator:

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

(Take $T = \mathsf{CoList}\,A$ and $f = \lambda r.\,\mathsf{cons}\,a\,r$ to reconstruct the example.) The fixed-point rule can be justified by transfinite induction on ordinal index i. While the successor case is covered by the premise of the rule, for zero and limit case the size-indexed type T must satisfy two conditions: $T\,0 = \top$ (*bottom check*) and $\bigcap_{\alpha<\lambda} T\,\alpha \subseteq T\,\lambda$ for limit ordinals λ [HPS96]. The latter condition is non-compositional, but has a compositional generalization, *upper semi-continuity* $\bigcap_{\alpha<\lambda} \bigcup_{\alpha\le\beta<\lambda} T\,\beta \subseteq T\,\lambda$ [Abe08b].

The soundness of type-based termination in different variants for different type systems has been assessed in at least 5 PhD theses: Barras [Bar99] (CIC), Pareto [Par00] (lazy ML), Frade [Fra03] (STL), the author [Abe06] (F^ω), and Sacchini [Sac11] (CIC). Recently, Barras [Bar10a] has completed a comprehensive formal verification in Coq, by implementing a set-theoretical model of the CIC with type-based termination.

However, type-based termination has not been integrated into bigger systems like Agda and Coq. There are a number of reasons:

1. Subtyping.
   The inclusion relation between approximants gives rise to subtyping, and for dependent types, subtyping has not been fully explored. While there are basic theory [AC01, Che97], substantial work on coercive subtyping [Che03, LA08] and new results on Pure Subtype Systems [Hut10], no theory of higher-order polarized subtyping [Ste98, Abe08a] has been formulated for dependent types yet. In practice, the introduction of subtyping means that already complicated higher-order unification has to be replaced by preunification [QN94].
2. Erasure.
   Mixing sizes into types and expressions means that one also needs to erase them after type checking, since they have no computational significance. The type system must be able to distinguish

relevant from irrelevant parts. This is also work in progress, partial solutions have been given, e. g., by Barras and Bernardo [BB08] and the author [Abe11].

3. Semi-continuity.
   A technical condition like semi-continuity can kill a system as a candidate for the foundation of logics and programming. It seems that it even deters the experts: Most systems for type-based termination replace semi-continuity by a rough approximation, trading expressivity for simplicity—Pareto and the author being notable exceptions.

4. Pattern matching.
   The literature on type-based termination is a bit thin when it comes to pattern matching. Pattern matching on sized inductive types has only been treated by Blanqui [Bla04]. Pattern matching on coinductive types is known to violate subject reduction in dependent type theory (detailed analysis by McBride [McB09]). Deep matching on sized types can lead to a surprising paradox [Abe10].

While items 1 and 2 require more work, items 3 and 4 can be addressed by switching to a different style of type-based termination, which we study in the next section.

3 Inflationary Iteration and Bounded Size Quantification

Sprenger and Dam [SD03] note that for monotone F,

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

and base their system of *circular proofs in the μ-calculus* on this observation. They introduce syntax for unbounded $\exists i$ and bounded $\exists j < i$ ordinal existentials and for approximants μ^i (cf. Dam and Gurov [DG02] and Schöpp and Simpson [SS02]). Induction is well-founded induction on ordinals, and no semi-continuity is required.

A first thing to note is that if we take above equation as the *definition* for $\mu^\alpha F$, the chain $\alpha \mapsto \mu^\alpha F$ is monotone regardless of monotonicity of F. This style of iteration from below is called *inflationary iteration* and the dual, *deflationary iteration*,

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

always produces a descending chain. While inflationary iteration of F becomes stationary at some closure ordinal γ, the limit $\mu^\gamma F$ is only a pre-fixed point of F, i. e., $F\ (\mu^\gamma F) \subseteq \mu^\gamma F$. This means we can construct elements in a inflationary fixed-point as usual, but not necessarily analyze them sensibly. Unless F is monotone, destructing an element of $\mu^\gamma F$ yields only an element of $F\,(\mu^\beta F)$ for some $\beta < \gamma$ and not one of $F\,(\mu^\gamma F)$. Dually, deflationary iteration reaches a post-fixed point $\nu^\gamma F \subseteq F\ (\nu^\gamma F)$ giving the usual destructor, but the constructor has type $(\forall \beta < \gamma.\ F\ (\nu^\beta F)) \to \nu^\gamma F$.

While we have not come across a useful application of negative inflationary fixed points in programming, inflationary iteration leads to "cleaner" type-based termination. Inductive data constructors have type $(\exists j < i.\ F\,(\mu^j F)) \to \mu^i F$, meaning that when we pattern match at inductive type $\mu^i F$, we get a fresh size variable $j < i$ and a rest of type $F\,(\mu^j F)$. This is the "good" way of matching that avoids paradoxes [Abe10]; find it also in Barras [Bar10a]. Coinductive data has type $\nu^i F \cong \forall j < i.\ F\,(\nu^j F)$,

akin to a dependent function type. We cannot match on it, only apply it to a size, preventing subject reduction problems mentioned in the previous section. Finally, recursion becomes well-founded recursion on ordinals,

$$\frac{f : \forall i.\ (\forall j < i.\ T\ j) \to T\ i}{\mathsf{fix}\ f : \forall i.\ T\ i}$$

with no condition on T. Also, just like in PiSigma [ADLO10], we can dispose of inductive and coinductive types in favor of recursion. We just define approximants recursively using bounded quantifiers; for instance, sized streams are $\mathsf{Stream}\ A\ i = \forall j < i.\ A \times \mathsf{Stream}\ A\ j$, and in MiniAgda:

```
cofun Stream : +(A : Set) -(i : Size) → Set
{ Stream A i = [j < i] → A & Stream A j
}
```

MiniAgda checks that $\mathsf{Stream}\ A\ i$ is monotone in element type A and antitone in depth i, as specified by the polarities + and - in the type signature. If we erase sizes to () and Size to the non-informative type $\top$, we obtain $\mathsf{Stream}\ A\ () = \top \to A \times \mathsf{Stream}\ A\ ()$ which is a possible representation of streams in call-by-value languages. Thus, size quantification can be considered as type *lifting*, size application as *forcing* and size abstraction as *delaying*.

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

Taking the tail requires a stream of non-zero depth $i+1$. Since $s : \forall j < (i+1).\ A \times \mathsf{Stream}\ A\ j$, we can apply it to i (*force* it) and then take its second component.

Zipping two streams $sa = a_0, a_1, \ldots$ and $sb = b_0, b_1, \ldots$ with a function f yields a stream $sc = f(a_0, b_0), f(a_1, b_1), \ldots$ whose depth is the minimum of the depths of sa and sb. Since depths are lower bounds, we can equally state that all three streams have a common depth i.

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

Forcing the recursively defined stream zipWith $A\ B\ C\ f\ i\ sa\ sb$ by applying it to $j < i$ yields a head-tail pair $(f\ a\ b,\ \mathsf{zipWith}\ A\ B\ C\ f\ j\ as\ bs)$ which is computed from heads a and b and tails as and bs of the forced input streams $sa\ j$ and $sb\ j$. The recursion is well-founded since $j < i$.

The famous Haskell one-line definition `fib = 0 : 1 : zipWith (+)fib (tail fib)` of the Fibonacci stream `0 : 1 : 1 : 2 : 3 : 5 : 8 : 13...` can now be replayed in MiniAgda.

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

The `|i|` in the type explicitly states that ordinal i shall serve as termination measure (syntax due to Xi [Xi02]). Note the two delays $\lambda j < i$ and $\lambda k < j$ and the two recursive calls, both at smaller depth $j, k < i$. Such a definition is beyond the guardedness check [Coq93] of Agda and Coq, but here the type

system communicates that zipWith preserves the stream depth and, thus, productivity.

While our type system guarantees termination and productivity at run-time, *strong* normalization, in particular when reducing under λ-abstractions, is lost when coinductive types are just defined recursively. Thus, equality testing of functions has to be very intensional (α-equality [ADLO10]), since testing η-equality may loop. McBride [McB09] suggests an extensional propositional equality [AMS07] as cure.

Having explained away inductive and coinductive types, mixing them does not pose a problem anymore, as we will see in the next section.

4 Mixing Induction and Coinduction

A popular mixed coinductive-inductive type are stream processors [GHP06] given recursively by the equation $\mathsf{SP}\ A\ B = (A \to \mathsf{SP}\ A\ B) + (B \times \mathsf{SP}\ A\ B)$. The intention is that $\mathsf{SP}\ A\ B$ represents continuous functions from $\mathsf{Stream}\ A$ to $\mathsf{Stream}\ B$, meaning that only finitely many A's are taken from the input stream before a B is emitted on the output stream. This property can be ensured by nesting a least fixed-point into a greatest one: $\mathsf{SP}\ A\ B = \nu X.\, \mu Y.\, (A \to Y) + (B \times X)$ [Abe07, GHP09]. The greatest fixed-point unfolds to $\mu Y.\, (A \to Y) + (B \times \mathsf{SP}\ A\ B)$, hence, whenever we chose the second alternative, the least fixed-point is "restarted". Thus, we can conceive $\mathsf{SP}\ A\ B$ by a *lexicographic* ordinal iteration

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

where ∞ represents the closure ordinal. The nesting is now defined by the lexicographic recursion pattern, so we do not need to represent it in the order of quantifiers. Pushing them in maximally yields an alternative definition:

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

This variant is close to the mixed data types of Agda [DA10], where recursive occurrences are inductive unless marked with ∞:

```
data SP (A B : Set) : Set where
  get : (A → SP A B) → SP A B
  put : B → ∞ (SP A B) → SP A B
```

In Agda, one cannot specify the nesting order, it always considers the greatest fixed-point to be on the outside [AD10].

Let us program with mixed types via bounded quantification in MiniAgda! The type of stream processors is defined recursively, with lexicographic termination measure `|i,j|`. The bounded existential $\exists j' < j.T$ has concrete syntax `[j' <j] & T`, and `Either X Y` with constructors `left: X → Either X Y` and `right : Y → Either X Y` is the (definable) disjoint sum type. We directly code the "mixed" definition of SP:

```
cofun SP : -(A : Set) +(B : Set) -(i : Size) +(j : Size) → |i,j| → Set
{ SP A B i j = Either (A → [j' < j] & SP A B i j')
                     (B & ([i' < i] → SP A B i' #))
}
pattern get f    = left f
pattern put b sp = right (b , sp)
```

We can *run* a stream processor of depth i and height j on an A-stream of unbounded depth (∞) to yield a

B-stream of depth i (this is also called stream *eating* [GHP09]). If the stream processor is a get f, we feed the head of the stream to f, getting an new stream processor of smaller height (index j), and continue running on the stream tail. If the stream processor is a put b sp, we produce a $\lambda i' < i$ delayed stream whose head is b and tail is computed by running sp, which has smaller depth (index i) but unbounded height (index j).

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

A final note on quantifier placement: For monotone F and $\overline{\mu}^\alpha = F\ (\bigcup_{\beta<\alpha} \overline{\mu}^\beta)$ we have $\overline{\mu}^\alpha F = \mu^{\alpha+1} F$. In particular $\overline{\mu}^0 F = F\,\bot$, thus for the list generator $F\ X = \top + A \times X$ the first approximant $\overline{\mu}^0 F$ is not empty but contains exactly the empty list. Type $\overline{\mu}^\alpha F$ contains the lists of maximal length α. This encoding of data type approximants is more suitable for size arithmetic and has been advocated by Barthe, Grégoire, and Riba [BGR08b]; in practice, it might be superior—time will tell.

5 Conclusions

We have given a short introduction into a type system for termination based on ordinal iteration. Bounded size quantification, inspired by inflationary fixed points, and recursion with ordinal lexicographic termination measures are sufficient to encode inductive and coinductive types and recursive and corecursive definitions and all mixings thereof. The full power of classical ordinals is not needed to justify our recursion schemes: We only need a well-founded order $<$ that is "long enough" and has a successor operation. I conjecture that set induction or constructive ordinals (Aczel and Rathjen [AR08]) can play this role, leading to a constructive justification of type-based termination.

While our prototype MiniAgda lacks type reconstruction needed for an enjoyable programming experience, it is evolving into a core language for dependent type theory with termination certificates. Our long-term goal is to extend Agda with type-based termination in a way that most termination certificates will be constructed automatically. MiniAgda could serve as an intermediate language that double-checks proofs constructed by Agda, erases static code, and feeds the rest into a compiler back-end.

Acknowledgements. I am grateful for discussions with Cody Roux which exposed a problem with MiniAgda's pattern matching and set me on the track towards bounded quantification as basic principle for type-based termination. Thanks to Brigitte Pientka for many discussions on sized types and the invitation to McGill, where some ideas of this paper prospered. Finally, I thank the MiniAgda users, especially Nils Anders Danielsson and David Thibodeau, who have coped with the user-unfriendliness of the system and kept me busy fixing bugs.

References

[AA02] Andreas Abel & Thorsten Altenkirch (2002): *A Predicative Analysis of Structural Recursion*. *J. Func. Program.* 12(1), pp. 1–41, doi:10.1017/S0956796801004191.

[Abe06] Andreas Abel (2006): *A Polymorphic Lambda-Calculus with Sized Higher-Order Types*. Ph.D. thesis, Ludwig-Maximilians-Universität München.

[Abe07] Andreas Abel (2007): *Mixed Inductive/Coinductive Types and Strong Normalization*. In Zhong Shao, editor: *Proc. of the 5th Asian Symp. on Programming Languages and Systems, APLAS 2007*, *Lect. Notes in Comput. Sci.* 4807, Springer, pp. 286–301, doi:10.1007/978-3-540-76637-7_19.

[Abe08a] Andreas Abel (2008): *Polarized Subtyping for Sized Types*. *Math. Struct. in Comput. Sci.* 18, pp. 797–822, doi:10.1017/S0960129508006853. Special issue on subtyping, edited by Healfdene Goguen and Adriana Compagnoni.

[Abe08b] Andreas Abel (2008): *Semi-continuous Sized Types and Termination*. *Logical Meth. in Comput. Sci.* 4(2), doi:10.2168/LMCS-4(2:3)2008. CSL'06 special issue.

[Abe10] Andreas Abel (2010): *MiniAgda: Integrating Sized and Dependent Types*. In Ana Bove, Ekaterina Komendantskaya & Milad Niqui, editors: *Wksh. on Partiality And Recursion in Interactive Theorem Provers (PAR 2010)*, *Electr. Proc. in Theor. Comp. Sci.* 43, pp. 14–28, doi:10.4204/EPTCS.43.2.

[Abe11] Andreas Abel (2011): *Irrelevance in Type Theory with a Heterogeneous Equality Judgement*. In Martin Hofmann, editor: *Proc. of the 14th Int. Conf. on Foundations of Software Science and Computational Structures, FOSSACS 2011*, *Lect. Notes in Comput. Sci.* 6604, Springer, pp. 57–71, doi:10.1007/978-3-642-19805-2_5.

[AC01] David Aspinall & Adriana B. Compagnoni (2001): *Subtyping dependent types*. *Theor. Comput. Sci.* 266(1-2), pp. 273–309, doi:10.1016/S0304-3975(00)00175-4.

[ACG98] Roberto M. Amadio & Solange Coupet-Grimal (1998): *Analysis of a Guard Condition in Type Theory (Extended Abstract)*. In Maurice Nivat, editor: *Proc. of the 1st Int. Conf. on Foundations of Software Science and Computation Structure, FoSSaCS'98*, *Lect. Notes in Comput. Sci.* 1378, Springer, pp. 48–62, doi:10.1007/BFb0053541.

[AD10] Thorsten Altenkirch & Nils Anders Danielsson (2010): *Termination Checking in the Presence of Nested Inductive and Coinductive Types*. Short note supporting a talk given at PAR 2010, Workshop on Partiality and Recursion in Interactive Theorem Provers, FLoC 2010. Available at http://www.cse.chalmers.se/~nad/publications/altenkirch-danielsson-par2010.pdf.

[ADLO10] Thorsten Altenkirch, Nils Anders Danielsson, Andres Löh & Nicolas Oury (2010): *PiSigma: Dependent Types without the Sugar*. In Matthias Blume, Naoki Kobayashi & Germán Vidal, editors: *Proc. of the 10th Int. Symp. on Functional and Logic Programming, FLOPS 2010*, *Lect. Notes in Comput. Sci.* 6009, Springer, pp. 40–55, doi:10.1007/978-3-642-12251-4_5.

[AMS07] Thorsten Altenkirch, Conor McBride & Wouter Swierstra (2007): *Observational equality, now!* In Aaron Stump & Hongwei Xi, editors: *Proc. of the Wksh. Programming Languages meets Program Verification, PLPV 2007*, ACM Press, pp. 57–68, doi:10.1145/1292597.1292608.

[AR08] Peter Aczel & Michael Rathjen (2008): *Notes on Constructive Set Theory*. Available at http://www.maths.manchester.ac.uk/logic/mathlogaps/workshop/CST-book-June-08.pdf. Draft.

[Bar99] Bruno Barras (1999): *Auto-validation d'un système de preuves avec familles inductives*. Ph.D. thesis, Université Paris 7.

[Bar10a] Bruno Barras (2010): *Sets in Coq, Coq in Sets*. *J. Formalized Reasoning* 3(1). Available at http://jfr.cib.unibo.it/article/view/1695.

[Bar10b] Bruno Barras (2010): *The syntactic guard condition of Coq*. Talk at the Journée "égalité et terminaison" du 2 février 2010 in conjunction with JFLA 2010. Available at http://coq.inria.fr/files/adt-2fev10-barras.pdf.

[BB08] Bruno Barras & Bruno Bernardo (2008): *The Implicit Calculus of Constructions as a Programming Language with Dependent Types*. In Roberto M. Amadio, editor: *FoSSaCS*, *Lect. Notes in Comput. Sci.* 4962, Springer, pp. 365–379, doi:10.1007/978-3-540-78499-9_26.

[BGP06] Gilles Barthe, Benjamin Grégoire & Fernando Pastawski (2006): *CIC^: Type-Based Termination of Recursive Definitions in the Calculus of Inductive Constructions*. In Miki Hermann & Andrei Voronkov, editors: *Proc. of the 13th Int. Conf. on Logic for Programming, Artificial Intelligence, and Reasoning, LPAR 2006*, *Lect. Notes in Comput. Sci.* 4246, Springer, pp. 257–271, doi:10.1007/11916277_18.

[BGR08a] Gilles Barthe, Benjamin Grégoire & Colin Riba (2008): *A Tutorial on Type-Based Termination*. In Ana Bove, Luís Soares Barbosa, Alberto Pardo & Jorge Sousa Pinto, editors: *LerNet ALFA Summer School*, *Lect. Notes in Comput. Sci.* 5520, Springer, pp. 100–152, doi:10.1007/978-3-642-03153-3_3.

[BGR08b] Gilles Barthe, Benjamin Grégoire & Colin Riba (2008): *Type-Based Termination with Sized Products*. In Michael Kaminski & Simone Martini, editors: *Computer Science Logic, 22nd Int. Wksh., CSL 2008, 17th Annual Conf. of the EACSL*, *Lect. Notes in Comput. Sci.* 5213, Springer, pp. 493–507, doi:10.1007/978-3-540-87531-4_35.

[Bla04] Frédéric Blanqui (2004): *A Type-Based Termination Criterion for Dependently-Typed Higher-Order Rewrite Systems*. In Vincent van Oostrom, editor: *Rewriting Techniques and Applications (RTA 2004), Aachen, Germany*, *Lect. Notes in Comput. Sci.* 3091, Springer, pp. 24–39, doi:10.1007/978-3-540-25979-4_2.

[CC79] Patrick Cousot & Radhia Cousot (1979): *Constructive Versions of Tarski's Fixed Point Theorems*. *Pacific Journal of Mathematics* 81(1), pp. 43–57.

[Che97] Gang Chen (1997): *Subtyping Calculus of Construction (Extended Abstract)*. In Igor Prívara & Peter Ruzicka, editors: *Proc. of the 22nd Int. Symb. on Mathematical Foundations of Computer Science, MFCS'97*, *Lect. Notes in Comput. Sci.* 1295, Springer, pp. 189–198, doi:10.1007/BFb0029962.

[Che03] Gang Chen (2003): *Coercive subtyping for the calculus of constructions*. In: *Proc. of the 30st ACM Symp. on Principles of Programming Languages, POPL 2003*, *ACM SIGPLAN Notices* 38, ACM Press, pp. 150–159, doi:10.1145/640128.604145.

[Coq93] Thierry Coquand (1993): *Infinite Objects in Type Theory*. In H. Barendregt & T. Nipkow, editors: *Types for Proofs and Programs (TYPES '93)*, *Lect. Notes in Comput. Sci.* 806, Springer, pp. 62–78, doi:10.1007/3-540-58085-9_72.

[DA10] Nils Anders Danielsson & Thorsten Altenkirch (2010): *Subtyping, Declaratively*. In Claude Bolduc, Jules Desharnais & Béchir Ktari, editors: *Proc. of the 10th Int. Conf. on Mathematics of Program Construction, MPC 2010*, *Lect. Notes in Comput. Sci.* 6120, Springer, pp. 100–118, doi:10.1007/978-3-642-13321-3_8.

[DG02] Mads Dam & Dilian Gurov (2002): *μ-Calculus with Explicit Points and Approximations*. *J. Log. Comput.* 12(2), pp. 255–269, doi:10.1093/logcom/12.2.255.

[Fra03] Maria João Frade (2003): *Type-Based Termination of Recursive Definitions and Constructor Subtyping in Typed Lambda Calculi*. Ph.D. thesis, Universidade do Minho, Departamento de Informática.

[GHP06] Neil Ghani, Peter Hancock & Dirk Pattinson (2006): *Continuous Functions on Final Coalgebras*. *Electr. Notes in Theor. Comp. Sci.* 164(1), pp. 141–155, doi:10.1016/j.entcs.2006.06.009.

[GHP09] Neil Ghani, Peter Hancock & Dirk Pattinson (2009): *Representations of Stream Processors Using Nested Fixed Points*. *Logical Meth. in Comput. Sci.* 5(3), doi:10.2168/LMCS-5(3:9)2009.

[Gim95] Eduardo Giménez (1995): *Codifying Guarded Definitions with Recursive Schemes*. In Peter Dybjer, Bengt Nordström & Jan Smith, editors: *Types for Proofs and Programs, Int. Wksh., TYPES'94*, *Lect. Notes in Comput. Sci.* 996, Springer, pp. 39–59, doi:10.1007/3-540-60579-7_3.

[Gim98] Eduardo Giménez (1998): *Structural Recursive Definitions in Type Theory*. In K. G. Larsen, S. Skyum & G. Winskel, editors: *Int. Colloquium on Automata, Languages and Programming (ICALP'98), Aalborg, Denmark*, *Lect. Notes in Comput. Sci.* 1443, Springer, pp. 397–408, doi:10.1007/BFb0055070.

[Gir72] Jean-Yves Girard (1972): *Interprétation fonctionnelle et élimination des coupures dans l'arithmétique d'ordre supérieur*. Thèse de Doctorat d'État, Université de Paris VII.

[HPS96] John Hughes, Lars Pareto & Amr Sabry (1996): *Proving the Correctness of Reactive Systems Using Sized Types*. In: *Proc. of the 23rd ACM Symp. on Principles of Programming Languages, POPL'96*, pp. 410–423, doi:10.1145/237721.240882.

[Hut10] DeLesley S. Hutchins (2010): *Pure subtype systems*. In Manuel V. Hermenegildo & Jens Palsberg, editors: *Proc. of the 37th ACM Symp. on Principles of Programming Languages, POPL 2010*, ACM Press, pp. 287–298, doi:10.1145/1706299.1706334.

[INR10] INRIA (2010): *The Coq Proof Assistant Reference Manual*, version 8.3 edition. INRIA. Available at http://coq.inria.fr/.

[LA08] Zhaohui Luo & Robin Adams (2008): *Structural subtyping for inductive types with functorial equality rules*. *Math. Struct. in Comput. Sci.* 18(5), pp. 931–972, doi:10.1017/S0960129508006956.

[McB09] Conor McBride (2009): *Let's See How Things Unfold: Reconciling the Infinite with the Intensional*. In Alexander Kurz, Marina Lenisa & Andrzej Tarlecki, editors: *3rd Int. Conf. on Algebra and Coalgebra in Computer Science, CALCO 2009*, *Lect. Notes in Comput. Sci.* 5728, Springer, pp. 113–126, doi:10.1007/978-3-642-03741-2_9.

[Men87] Nax Paul Mendler (1987): *Recursive Types and Type Constraints in Second-Order Lambda Calculus*. In: *Proc. of the 2nd IEEE Symp. on Logic in Computer Science (LICS'87)*, IEEE Computer Soc. Press, pp. 30–36.

[Mil78] Robin Milner (1978): *A Theory of Type Polymorphism in Programming*. *J. Comput. Syst. Sci.* 17, pp. 348–375, doi:10.1016/0022-0000(78)90014-4.

[Nor07] Ulf Norell (2007): *Towards a Practical Programming Language Based on Dependent Type Theory*. Ph.D. thesis, Dept of Comput. Sci. and Engrg., Chalmers, Göteborg, Sweden.

[Par00] Lars Pareto (2000): *Types for Crash Prevention*. Ph.D. thesis, Chalmers University of Technology.

[Pie01] Brigitte Pientka (2001): *Termination and Reduction Checking for Higher-Order Logic Programs*. In Rajeev Goré, Alexander Leitsch & Tobias Nipkow, editors: *Automated Reasoning, First International Joint Conference, IJCAR 2001*, *Lect. Notes in Art. Intell.* 2083, Springer, pp. 401–415, doi:10.1007/3-540-45744-5_32.

[QN94] Zhenyu Qian & Tobias Nipkow (1994): *Reduction and Unification in Lambda Calculi with a General Notion of Subtype*. *J. of Autom. Reasoning* 12(3), pp. 389–406, doi:10.1007/BF00885767.

[Rey74] John C. Reynolds (1974): *Towards a Theory of Type Structure*. In B. Robinet, editor: *Programming Symposium*, *Lect. Notes in Comput. Sci.* 19, Springer, Berlin, pp. 408–425, doi:10.1007/3-540-06859-7_148.

[Sac11] Jorge Luis Sacchini (2011): *On Type-Based Termination and Pattern Matching in the Calculus of Inductive Constructions*. Ph.D. thesis, INRIA Sophia-Antipolis and École des Mines de Paris.

[SD03] Christoph Sprenger & Mads Dam (2003): *On the Structure of Inductive Reasoning: Circular and Tree-Shaped Proofs in the μ-Calculus*. In Andrew D. Gordon, editor: *Proc. of the 6th Int. Conf. on Foundations of Software Science and Computational Structures, FoSSaCS 2003*, *Lect. Notes in Comput. Sci.* 2620, Springer, pp. 425–440, doi:10.1007/3-540-36576-1_27.

[SS02] Ulrich Schöpp & Alex K. Simpson (2002): *Verifying Temporal Properties Using Explicit Approximants: Completeness for Context-free Processes*. In Mogens Nielsen & Uffe Engberg, editors: *Proc. of the 5th Int. Conf. on Foundations of Software Science and Computational Structures, FoSSaCS 2002*, *Lect. Notes in Comput. Sci.* 2303, Springer, pp. 372–386, doi:10.1007/3-540-45931-6_26.

[Ste98] Martin Steffen (1998): *Polarized Higher-Order Subtyping*. Ph.D. thesis, Technische Fakultät, Universität Erlangen.

[Tai67] William W. Tait (1967): *Intensional Interpretations of Functionals of Finite Type I*. *J. Symb. Logic* 32(2), pp. 198–212.

[Wah00] David Wahlstedt (2000): *Detecting termination using size-change in parameter values*. Master's thesis, Göteborgs Universitet.

[Xi02] Hongwei Xi (2002): *Dependent Types for Program Termination Verification*. *J. Higher-Order and Symb. Comput.* 15(1), pp. 91–131, doi:10.1023/A:1019916231463.