EPTCS 77

Proceedings of the

8th Workshop on

Fixed Points in Computer Science

Tallinn, Estonia, 24th March 2012

Edited by: Dale Miller and Zoltán Ésik

Published: 14th February 2012
DOI: 10.4204/EPTCS.77
ISSN: 2075-2180
Open Publishing Association

Table of Contents

Preface

This volume contains the proceedings of the Eighth Workshop on Fixed Points in Computer Science which took place on 24 March 2012 in Tallinn, Estonia as an ETAPS-affiliated workshop. Past workshops have been held in Brno (1998, MFCS/CSL workshop), Paris (2000, LC workshop), Florence (2001, PLI workshop), Copenhagen (2002, LICS (FLoC) workshop), Warsaw (2003, ETAPS workshop), Coimbra (2009, CSL workshop), and Brno (2010, MFCS-CSL workshop).

Fixed points play a fundamental role in several areas of computer science and logic by justifying induction and recursive definitions. The construction and properties of fixed points have been investigated in many different frameworks such as: design and implementation of programming languages, program logics, and databases. The aim of this workshop is to provide a forum for researchers to present their results to those members of the computer science and logic communities who study or apply the theory of fixed points.

We wish to thank Andreas Abel (Ludwig-Maximilians-Universität) and Luke Ong (University of Oxford) for accepting our invitation to speak at this workshop and for their contributions to these proceedings. We also thank all those authors who have submitted extended abstracts for evaluation to the program committee. Thanks are also due to Keiko Nakata and Tarmo Uustalu for their support of this workshop, both organizational and financial, and to Zoltán L. Németh for his help with the workshop's web pages and with assembling these proceedings.

This workshop received support from the Estonian Centre of Excellence in Computer Science (EXCS), a project financed by the European Regional Development Fund (ERDF). We thank them for their support.

Zoltán Ésik and Dale Miller

D. Miller and Z. Ésik (Eds.): Fixed Points
in Computer Science 2012 (FICS 2012)
EPTCS 77, 2012, pp. iii–iv, doi:10.4204/EPTCS.77.0

The FICS 2012 Program Committee

David Baelde, University of Paris 11, France
Julian Bradfield, University of Edinburgh, UK
Arnaud Carayol, Institut Gaspard-Monge, France
Zoltán Ésik, University of Szeged, Hungary (co-Chair)
Wan Fokkink, Vrije Universiteit, Holland
Fabio Gadducci, University of Pisa, Italy
Irène Guessarian, University of Paris 7, France
Achim Jung, University of Birmingham, UK
Stephan Kreutzer, Technische Universität Berlin, Germany
Dale Miller, INRIA-Saclay & LIX/Ecole Polytechnique, France (co-Chair)
Ralph Matthes, IRIT Toulouse, France
Jan Rutten, CWI, Amsterdam, Radboud University, Holland
Luigi Santocanale, Université Aix-Marseille I, France
Tarmo Uustalu, Institute of Cybernetics, Estonia
Igor Walukiewicz, LaBRI Bordeaux, France

The FICS Steering Committee

Peter Dybjer, Chalmers University of Technology
Zoltán Ésik, University of Szeged
Anna Ingólfsdóttir, Reykjavík University
Ralph Matthes, IRIT, Toulouse, (Chair)
Damian Niwinski, University of Warsaw
Luigi Santocanale, LIF, Université Aix-Marseille I
Alex Simpson, University of Edinburgh
Tarmo Uustalu, Institute of Cybernetics, Tallinn
Igor Walukiewicz, LaBRI, Bordeaux

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 $\mathsf{let}\ \mathsf{id} = \lambda x.x\ \mathsf{in}\ \mathsf{id}\ \mathsf{id}\ \mathsf{true}$. 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

D. Miller and Z. Ésik (Eds.): Fixed Points
in Computer Science 2012 (FICS 2012)
EPTCS 77, 2012, pp. 1–11, doi:10.4204/EPTCS.77.1

[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:

```
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 $F\,X = \top + A \times X$. 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:

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

For monotone F, iteration is monotone, i. e., $\mu^\alpha F \subseteq \mu^\beta F$ for $\alpha \leq \beta$. At some ordinal γ, which we call *closure ordinal* of this inductive type, we have $\mu^\alpha F = \mu^\gamma F$ for all $\alpha \geq \gamma$—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.

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

Iteration from above is antitone, i. e., $\nu^\alpha F \supseteq \nu^\beta F$ for $\alpha \leq \beta$. 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 $\mathsf{zipWith}\ A\ B\ C\ f\ i\ sa\ sb$ 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.

High-Order Model Checking

C.-H. Luke Ong
University of Oxford
lo@cs.ox.ac.uk

Recursion schemes are in essence the simply-typed lambda calculus with recursion, generated from uninterpreted first-order symbols. An old model of computation much studied in the Seventies, there has been a revival of interest in recursion schemes as generators of infinite structures (such as infinite trees) with rich algorithmic properties. *Higher-order model checking*—the model checking of trees generated by higher-order recursion schemes—is a natural generalisation of finite-state and pushdown model checking; it can serve as a basis for software model checkers for functional languages such as ML and Haskell.

After a quick survey of expressivity and decidability results in higher-order model checking [6, 2, 5, 1], we present our recent application [7] to the model checking of higher-order functional programs with pattern-matching algebraic data types. We are concerned with the (undecidable) verification problem: given a correctness property ϕ, a functional program $\mathscr{P}$ and a regular input set I, does every term that is reachable from I under rewriting by $\mathscr{P}$ satisfy ϕ? Our solution is a sound semi-algorithm (i.e. given a no-instance of the verification problem, the method is guaranteed to terminate) which uses counterexample-guided abstraction refinement, and is based on a backend model checker.

Given a trivial automaton (i.e. Büchi tree automaton with a trivial acceptance condition) and a non-deterministic higher-order recursion scheme with case construct over finite date-types, the model checker decides if the language of trees generated by the scheme is accepted by the automaton. The model checking problem is characterised by an intersection type system [4, 5] extended with a carefully restricted form of union types; the decision procedure is based on the notion of traversal from game semantics [3, 6]. We demonstrate the effectiveness of an implementation of the algorithm on abstract models of functional programs obtained from an abstraction-refinement procedure.

This talk is based on joint work with Steven Ramsay and Robin Neatherway.

References

[1] Christopher H. Broadbent, Arnaud Carayol, C.-H. Luke Ong & Olivier Serre (2010): *Recursion Schemes and Logical Reflection*. In: *LICS*, pp. 120–129. Available at http://doi.ieeecomputersociety.org/10.1109/LICS.2010.40.

[2] Matthew Hague, Andrzej S. Murawski, C.-H. Luke Ong & Olivier Serre (2008): *Collapsible Pushdown Automata and Recursion Schemes*. In: *LICS*, pp. 452–461. Available at http://doi.ieeecomputersociety.org/10.1109/LICS.2008.34.

[3] J. M. E. Hyland & C.-H. Luke Ong (2000): *On Full Abstraction for PCF: I, II, and III*. *Inf. Comput.* 163(2), pp. 285–408. Available at http://dx.doi.org/10.1006/inco.2000.2917.

[4] Naoki Kobayashi (2009): *Types and higher-order recursion schemes for verification of higher-order programs*. In: *POPL*, pp. 416–428. Available at http://doi.acm.org/10.1145/1480881.1480933.

[5] Naoki Kobayashi & C.-H. Luke Ong (2009): *A Type System Equivalent to the Modal Mu-Calculus Model Checking of Higher-Order Recursion Schemes*. In: *LICS*, pp. 179–188. Available at http://dx.doi.org/10.1109/LICS.2009.29.

D. Miller and Z. Ésik (Eds.): Fixed Points
in Computer Science 2012 (FICS 2012)
EPTCS 77, 2012, pp. 13–14, doi:10.4204/EPTCS.77.2

[6] C.-H. Luke Ong (2006): *On Model-Checking Trees Generated by Higher-Order Recursion Schemes*. In: *LICS*, pp. 81–90. Available at http://doi.ieeecomputersociety.org/10.1109/LICS.2006.38. Long version (55 pp.) www.cs.ox.ac.uk/people/luke.ong/personal.

[7] C.-H. Luke Ong & Steven James Ramsay (2011): *Verifying higher-order functional programs with pattern-matching algebraic data types*. In: *POPL*, pp. 587–598. Available at http://doi.acm.org/10.1145/1926385.1926453.

Characteristic Formulae for Relations with Nested Fixed Points*

Luca Aceto Anna Ingólfsdóttir†
ICE-TCS, School of Computer Science
Reykjavik University
Reykjavik, Iceland
{luca,annai}@ru.is

A general framework for the connection between characteristic formulae and behavioral semantics is described in [2]. This approach does not suitably cover semantics defined by nested fixed points, such as the n-nested simulation semantics for n greater than 2. In this study we address this deficiency and give a description of nested fixed points that extends the approach for single fixed points in an intuitive and comprehensive way.

1 Introduction

In process theory it has become a standard practice to describe behavioural semantics in terms of equivalences or preorders. A wealth of such relations has been classified by van Glabbeek in his linear time/branching time spectrum [4]. Branching-time behavioural semantics are often defined as largest fixed points of monotonic functions over the complete lattice of binary relations over processes.

In [2] we give a general framework to reason about how this type of behavioral semantics can be characterized by a modal logic equipped with a greatest fixed point operator, or more precisely by characteristic formulae expressed in such a logic. In that reference we show that a behavioural relation that is derived as a greatest fixed point of a function of relations over processes is given by the greatest fixed point of the semantic interpretation of a logical declaration that expresses the function in a formal sense that is defined in present paper. Roughly speaking if a logical declaration describes a monotonic function over a complete lattice then its fixed point describes exactly the fixed point of the function. In [2] preorders and equivalences such as simulation preorder and bisimulation equivalence are characterized following this approach in a simple and constructive way. However, when the definition of a behavioural relation involves nested fixed points, i. e. when the monotonic function that defines the relation takes another fixed point as an argument, things get more complicated. The framework offered in [2] only deals with nesting on two levels and in a rather clumsy and unintuitive way. Furthermore it does not extend naturally to deeper nesting, like for the n-nested simulations for $n > 2$. In this study we address this deficiency and define a logical framework in which relations obtained as a chain of nested fixed points of monotonic functions can be characterized following general principles. This extends the approach for single fixed points in an intuitive and comprehensive way.

As the applications we present in the paper only deal with nesting of greatest fixed points, this study only focuses on greatest fixed points. However it is straightforward to extend it to deal with alternating nesting of both least and greatest fixed points. We also believe that our approach gives some idea about how fixed point theories in different domains can be compared in a structured way.

*Supported by the project Processes and Modal Logics' (project nr. 100048021) of the Icelandic Research Fund.
†Supported by the VELUX visiting professorship funded by the VILLUM FOUNDATION.

D. Miller and Z. Ésik (Eds.): Fixed Points
in Computer Science 2012 (FICS 2012)
EPTCS 77, 2012, pp. 15–22, doi:10.4204/EPTCS.77.3

The remainder of the paper is organized as follows. Section 2 presents some background on fixed points of monotone functions. Section 3 briefly introduces the model of labelled transition systems and some results on behavioural relations defined as greatest fixed points of monotonic functions over binary relations. The logic we shall use to define characteristic formulae in a uniform fashion is discussed in Section 4. The key notion of a declaration expressing a monotone function is also given in that section. Section 5 is devoted to an application of our framework to the logical characterization of the family of nested simulation semantics.

2 Posets, monotone functions and fixed points

In this section we introduce some basic concepts we need in the paper.

Definition 2.1

- *A* partially ordered set, *or* poset, $(A, \sqsubseteq_A)$ *(usually referred to simply as A) consists of a set A and a partial order* $\sqsubseteq_A$ *over it.*
- *If A is a poset and* $M \subseteq A$*, then* $a \in A$ *is an* upper bound *for M if* $m \sqsubseteq_A a$ *for all* $m \in M$*. a is a* least upper bound *(lub) for M if it is an upper bound for M and if whenever b is an upper bound for M then* $a \sqsubseteq_A b$*.*
- *A poset A is a* complete lattice *if the lub for M exists for all* $M \subseteq A$*.*
- *For posets A and B, a function* $\phi : A \to B$ *is monotone if it is order preserving; it is an* isomorphism *if it is bijective and both* ϕ *and its inverse* ϕ^{-1} *are monotone. We let* $A \to_{mono} B$ *denote the set of monotone functions from A to B.*
- *If A is a poset and* $f \in A \to_{mono} A$*, then* $x \in A$ *is a fixed point of* f *if* $f(x) = x$*. We write* νf *(or* $\nu x.\, f(x)$*) for the greatest fixed point of* f *if it exists.*
- *If A and B are posets,* $f \in A \to_{mono} A$ *and* $\phi \in A \to_{mono} B$ *is an isomorphism then we define* $\phi^* f : B \to B$ *as* $\phi^* f = \phi \circ f \circ \phi^{-1}$*.*

Note that the *lub* of a subset of a poset A is unique if it exists and the same holds for greatest fixed points of monotone functions over posets. It is well known, that if A and B are posets/complete lattices and I is some set, then the Cartesian product $A \times B$ and the function space $I \to A$ are a posets/complete lattices under the pointwise ordering. The following theorem is due to Tarski.

Theorem 2.2 ([10]) *If A is a complete lattice and* $f \in A \to_{mono} A$*, then* f *has a unique greatest fixed point.*

The theorem below is proved in [2] and is the key to the general theory we present in this paper.

Theorem 2.3 *Let A and B be posets,* $f \in A \to_{mono} A$ *and* $\phi : A \to B$ *be an isomorphism. Then* νf *exists iff* $\nu(\phi^* f)$ *exists. If these fixed points exist then* $\phi(\nu f) = \nu(\phi^* f)$*.*

3 Labelled transition systems and behavioural relations

It has become standard practice to describe behavioural semantics of processes by means of a *labelled transition system* as defined below.

Definition 3.1 ([7]) *A* labelled transition system (LTS) *is a triple* $P = (\mathbf{P}, \mathbf{A}, \to)$ *where*

- $\mathbf{A}$ *is a finite set (of actions),*

- $\mathbf{P}$ *is a finite set (of processes), and*
- $\rightarrow \subseteq \mathbf{P} \times \mathbf{A} \times \mathbf{P}$ *is a transition relation.*

As usual, we write $p \xrightarrow{a} p'$ for $(p,a,p') \in \rightarrow$. Throughout this paper we assume that the set $\mathbf{A}$ is fixed.

As LTSs are in general to concrete, processes are compared by preorders or equivalences. These are often obtained as the greatest fixed points to monotone endofunctions on the complete lattice $\mathscr{P}(\mathbf{P} \times \mathbf{P})$. We will show some example of such functions but first we state and prove some properties.

Definition 3.2 *If $\mathscr{F} \in \mathscr{P}(\mathbf{P} \times \mathbf{P}) \rightarrow_{mono} \mathscr{P}(\mathbf{P} \times \mathbf{P})$ and $A \in \mathscr{P}(\mathbf{P} \times \mathbf{P})$, we define*

- $\tilde{\mathscr{F}} : S \mapsto (\mathscr{F}(S^{-1}))^{-1}$, *and*
- $\mathscr{F} \cap A : S \mapsto \mathscr{F}(S) \cap A$.

The following lemma will be applied below.

Lemma 3.3 *Let $\mathscr{F} \in \mathscr{P}(\mathbf{P} \times \mathbf{P}) \rightarrow_{mono} \mathscr{P}(\mathbf{P} \times \mathbf{P})$ and $A \in \mathscr{P}(\mathbf{P} \times \mathbf{P})$. Then*

- $\tilde{\mathscr{F}}, \mathscr{F} \cap A \in \mathscr{P}(\mathbf{P} \times \mathbf{P}) \rightarrow_{mono} \mathscr{P}(\mathbf{P} \times \mathbf{P})$,
- $\nu\tilde{\mathscr{F}} = (\nu\mathscr{F})^{-1}$ *and*
- $\widetilde{\mathscr{F} \cap A} = \tilde{\mathscr{F}} \cap A^{-1}$.

Proof The first two statements are proved in [2]. To prove the third one we proceed follows:

$$(\widetilde{\mathscr{F} \cap A})(S) = ((\mathscr{F} \cap A)(S^{-1}))^{-1} = (\mathscr{F}(S^{-1}))^{-1} \cap A^{-1} = (\tilde{\mathscr{F}} \cap A^{-1})(S).$$

We will complete this section by giving some examples of endofunction that define some standard behavioural preorders and equivalences [4, 1].

Definition 3.4 *Let $\mathscr{F} : \mathscr{P}(\mathbf{P} \times \mathbf{P}) \rightarrow \mathscr{P}(\mathbf{P} \times \mathbf{P})$ be defined as follows:*

$$(p,q) \in \mathscr{F}(S) \text{ iff } \forall a \in \mathbf{A}, p' \in \mathbf{P}. p \xrightarrow{a} p' \Rightarrow \exists q' \in \mathbf{P}. q \xrightarrow{a} q' \wedge (p',q') \in S.$$

It is easy to check that $\mathscr{F}$ is monotonic and therefore it has a greatest fixed point.

Definition 3.5 *We define:*

- $\mathscr{F}_{sim} = \mathscr{F}$ *and* $\sqsubseteq_{sim} = \nu\mathscr{F}_{sim}$ *(simulation preorder),*
- $\mathscr{F}_{opsim} = \tilde{\mathscr{F}}$ *and* $\sqsubseteq_{opsim} = \nu\mathscr{F}_{opsim}$ *(inverse simulation preorder),*
- $\sim_{sim} = \sqsubseteq_{sim} \cap \sqsubseteq_{opsim}$ *(simulation equivalence) and*
- $\mathscr{F}_{bisim} = \mathscr{F}_{sim} \cap \mathscr{F}_{opsim}$ *and* $\sim_{bisim} = \nu\mathscr{F}_{bisim}$ *(bisimulation equivalence).*

4 Equational modal ν-calculi with nested fixed-points

In this section we introduce variants of the standard equational modal μ-calculus [8]. Like in [9] these variants only allow for nested fixed points, i. e. where the logical languages form a hierarchy where fixed points in a language on one level are allowed as constants in the logic on the level above. Our approach, however, differs from the original one in the sense that the fixed-point operator is explicit in the syntax and can therefore be used in logical expressions. In this study we only focus on greatest fixed points (which explains the title of this section) but the framework can easily be extended to involve nesting

of both greatest and least fixed points. The logical languages we introduce depend on the implicitly assumed fixed finite set $\mathbf{A}$.

Our basic logic $\mathscr{M}$ is the standard Hennessy-Milner Logic (HML) [6] without variables. This logic is generated by $\Sigma = (\Sigma_0, \Sigma_1, \Sigma_2)$ where $\Sigma_0 = \{tt, ff\}$ are the constants or the operators of arity 0, $\Sigma_1 = \{\langle a\rangle, [a], a \in \mathbf{A}\}$ are the operators of arity 1, and $\Sigma_2 = \{\wedge, \vee\}$ are the operators of arity 2.

The formulae in $\mathscr{M}$ are interpreted over an LTS $(\mathbf{P}, \mathbf{A}, \rightarrow)$ as the set of elements from $\mathbf{P}$ that satisfy them. Satisfaction is determined by a semantic function that is defined below. For $M \subseteq \mathbf{P}$ we let $\langle\cdot a\cdot\rangle M = \{p \in \mathbf{P} \mid \exists q \in M. p \xrightarrow{a} q\}$, and $[\cdot a\cdot]M = \overline{\langle\cdot a\cdot\rangle\overline{M}}$ where $\overline{M}$ is the complement of the set M.

Definition 4.1 *The semantic function $\mathscr{M}[\![\ ]\!]$ is defined as follows:*

1. $\mathscr{M}[\![tt]\!] = \mathbf{P}$, $\mathscr{M}[\![ff]\!] = \emptyset$,
2. $\mathscr{M}[\![F_1 \wedge F_2]\!] = \mathscr{M}[\![F_1]\!] \cap \mathscr{M}[\![F_2]\!]$, $\mathscr{M}[\![F_1 \vee F_2]\!] = \mathscr{M}[\![F_1]\!] \cup \mathscr{M}[\![F_2]\!]$,
3. $\mathscr{M}[\![\langle a\rangle F]\!] = \langle\cdot a\cdot\rangle\mathscr{M}[\![F]\!]$, $\mathscr{M}[\![[a]F]\!] = [\cdot a\cdot]\mathscr{M}[\![F]\!]$.

The logic $\mathscr{V}$ is the standard Hennessy-Milner logic with variables that was introduced in [9]. It assumes a finite index set I and an I-indexed set of variables $\mathscr{X}$. In what remains of this paper we assume a fixed pair of such I and $\mathscr{X}$, unless stated otherwise.

As the elements of $\mathscr{V}$ typically contain variables, they have to be interpreted with respect to a variable *interpretation* $\sigma \in \mathscr{P}(\mathbf{P})^I$ that associates to each $i \in I$ the set of processes in $\mathbf{P}$ that are assumed to satisfy the variable X_i. The semantic function $\mathscr{V}[\![\ ]\!]$ in this case takes a formula F and a $\sigma \in \mathscr{P}(\mathbf{P})^I$ and delivers an element of $\mathscr{P}(\mathbf{P})$.

Definition 4.2 *The semantic function $\mathscr{V}[\![\ ]\!]$ is defined as follows:*

1. $\mathscr{V}[\![F]\!]\sigma = \mathscr{M}[\![F]\!]$ *if* $F \in \Sigma_0$,
2. $\mathscr{V}[\![X_i]\!]\sigma = \sigma(i)$, $i \in I$,
3. $\mathscr{V}[\![F_1 \wedge F_2]\!]\sigma = \mathscr{V}[\![F_1]\!]\sigma \cap \mathscr{V}[\![F_2]\!]\sigma$, $\mathscr{V}[\![F_1 \vee F_2]\!]\sigma = \mathscr{V}[\![F_1]\!]\sigma \cup \mathscr{V}[\![F_2]\!]\sigma$,
4. $\mathscr{V}[\![\langle a\rangle F]\!]\sigma = \langle\cdot a\cdot\rangle\mathscr{V}[\![F]\!]\sigma$, $\mathscr{V}[\![[a]F]\!]\sigma = [\cdot a\cdot]\mathscr{V}[\![F]\!]\sigma$.

In [9] the meaning of the variables in the logic $\mathscr{V}$ is defined by means of a declaration, or a function $D : I \rightarrow \mathscr{V}$. Intuitively the syntactic function generates a monotonic endofunction $\mathscr{V}[\![D]\!]$ over $\mathscr{P}(\mathbf{P})^I$ defined by $(\mathscr{V}[\![D]\!])(i) = \mathscr{V}[\![D(i)]\!]$ for all $i \in I$. By Theorem 2.2, $\mathscr{V}[\![D]\!]$ has a unique largest fixed point $\nu\mathscr{V}[\![D]\!] \in \mathscr{P}(\mathbf{P})^I$ that can be used to give the semantics for the variables and the formulae that contain those in the logic $\mathscr{V}$. We can then use this to extend the logic $\mathscr{M}$ with $\{\nu D(i) | i \in I\}$ as constants interpreted as $\{\nu\mathscr{V}[\![D]\!](i) | i \in I\}$. By this we get a logic $\mathscr{M}'$ that is generated by $\Sigma' = (\Sigma_0 \cup \{\nu D(i) | i \in I\}, \Sigma_2, \Sigma_3)$. Then this procedure can be repeated for another declaration that possibly depends on νD as a constant and with $\mathscr{M}'$ as the basic logic. The following example shows how this construction works.

Example Let $I = \{1\}$, $\mathscr{X} = \{X_1\}$ and $\mathbf{A} = \{a, b\}$ and let the property "invariantly $\langle a\rangle tt$" be defined as the greatest fixed point corresponding to the declaration D_0 defined as $D_0(1) = \langle a\rangle tt \wedge [a]X_1 \wedge [b]X_1$. To interpret this we define $\mathscr{M} = \mathscr{M}_0$ and $\mathscr{V}_0 = \mathscr{V}$ where $\mathscr{M}$ and $\mathscr{V}$ have the meaning described above. The derived semantic function $\mathscr{V}_0[\![D_0]\!] : \mathscr{P}(\mathbf{P})^{\{1\}} \rightarrow \mathscr{P}(\mathbf{P})^{\{1\}}$ is easily shown to be monotonic and has the greatest fixed point $\nu\mathscr{V}_0[\![D_0]\!] \in \mathscr{P}(\mathbf{P})^{\{1\}}$. Now we define $\mathscr{M}_1$ as the extension of $\mathscr{M}_0$ that is generated by $\Sigma^1 = (\{tt, ff, \nu D_0(1)\}, \Sigma_1, \Sigma_2)$, i.e. has $\nu D_0(1)$ as a constant that is interpreted as $\nu\mathscr{V}_0[\![D_0]\!](1)$, i.e. $\mathscr{M}_1[\![\nu D_0(1)]\!] = \nu\mathscr{V}_0[\![D_0]\!](1)$.

Next let us assume that we have the declaration $D_1 : \{1\} \rightarrow \mathscr{V}_1$ where $\mathscr{V}_1$ is the variable logic generated by $(\{tt, ff, \nu D_0(1), X_1\}, \Sigma_2, \Sigma_3)$ and D_1 is defined as $D_1(1) = \langle b\rangle \nu D_0(1) \wedge [b]X_1$. As before the declaration is interpreted over $\mathscr{P}(\mathbf{P})^{\{1\}}$ but using $\mathscr{M}_1[\![\ ]\!]$ to interpret the constant $\nu D_0(1)$. Again D_1 is interpreted by using $\mathscr{V}_1[\![\ ]\!]$ which leads to a monotonic endofunction $\mathscr{V}_1[\![D_1]\!]$ over $\mathscr{P}(\mathbf{P})^{\{1\}}$ with a fixed point

$\nu\mathscr{V}_1[\![D_1]\!]$. The logic $\mathscr{M}_2$ is now defined as the one generated by $\Sigma^2 = (\{tt, ff, \nu D_1(1), \nu D_2(1)\}, \Sigma_2, \Sigma_3)$ where $\mathscr{M}_0[\![\ ]\!]$ and $\mathscr{M}_1[\![\ ]\!]$ are used to define the meaning of $\nu D_1(1)$ and $\nu D_2(1)$ respectively.

We will now generalize this procedure and define our hierarchy of nested fixed point logics, derived from a sequence of nested declarations $D_j, j = 1, 2, \ldots, N$, i.e. where for each $n < N$, D_{n+1} is allowed to depend on the constants tt, ff and $\nu D_j(i)$ for $j \leq n$ and $i \in I$. In the definition we assume a finite index set I and an I-indexed variable set $\mathscr{X}$. We use the notation $\mathscr{G}(\Sigma_0)$ for the logic generated by $(\Sigma_0, \Sigma_1, \Sigma_2)$ and $\mathscr{G}_I(\Sigma_0)$ for the logic generated by $(\Sigma_0 \cup \mathscr{X}, \Sigma_1, \Sigma_2)$.

Definition 4.3

- *Define*
  - $\Sigma_0^0 = \{tt, ff\}$,
  - $\mathscr{M}_0 = \mathscr{G}(\Sigma_0^0)$ *and*
  - $\mathscr{V}_0 = \mathscr{G}_I(\Sigma_0^0)$.
- *For* $n \geq 1$, *if* $D_n : I \to \mathscr{V}_n$, *define*
  - $\Sigma_0^{n+1} = \Sigma_0^n \cup \{\nu D_n(i) | i \in I\}$,
  - $\mathscr{M}_{n+1} = \mathscr{G}(\Sigma_0^{n+1})$ *and*
  - $\mathscr{V}_{n+1} = \mathscr{G}_I(\Sigma_0^{n+1})$.

To define the semantic functions associated with these logics we need the following lemma.

Lemma 4.4 *Assume that* $\mathscr{M} = \mathscr{G}(C)$ *and* $\mathscr{V} = \mathscr{G}_I(C)$ *for some set of constants* C *where* $\mathscr{M}[\![c]\!]$ *is well defined for all* $c \in C$. *Then for all* $D : I \to \mathscr{V}$, *the derived semantic function* $\mathscr{V}[\![D]\!]$ *defined by*

$$\forall i \in I.(\mathscr{V}[\![D]\!]\sigma)(i) = \mathscr{V}[\![D(i)]\!]\sigma$$

is in $\mathscr{P}(\mathbf{P})^I \to_{mono} \mathscr{P}(\mathbf{P})^I$ *and hence, by Theorem 2.2,* $\nu\mathscr{V}[\![D]\!] \in \mathscr{P}(\mathbf{P})^I$ *exists.*

Now we are ready to define the semantic functions for $\mathscr{M}_n$ and $\mathscr{V}_n$ for all $n \geq 0$.

Definition 4.5

- $\mathscr{M}_0 = \mathscr{M}$ *and* $\mathscr{V}_0 = \mathscr{V}$ *as defined in Definition 4.1 and 4.2 respectively.*
- *For* $n \geq 0$ *the semantic functions for* $\mathscr{M}_{n+1}$ *is defined as follows:*
  1. $\mathscr{M}_{n+1}[\![F]\!] = \mathscr{M}_n[\![F]\!]$ *if* $F \in \Sigma_0^n$,
  2. $\mathscr{M}_{n+1}[\![(\nu D_n)(i)]\!] = \nu\mathscr{V}_n[\![D_n]\!](i)$ *for* $i \in I$,
  3. $\mathscr{M}_{n+1}[\![F_1 \wedge F_2]\!] = \mathscr{M}_{n+1}[\![F_1]\!] \cap \mathscr{M}_{n+1}[\![F_2]\!]$, $\mathscr{M}_{n+1}[\![F_1 \vee F_2]\!] = \mathscr{M}_{n+1}[\![F_1]\!] \cup \mathscr{M}_{n+1}[\![F_2]\!]$,
  4. $\mathscr{M}_{n+1}[\![\langle a\rangle F]\!] = \langle\cdot a\cdot\rangle\mathscr{M}_{n+1}[\![F]\!]$, $\mathscr{M}_{n+1}[\![[a]F]\!] = [\cdot a\cdot]\mathscr{M}_{n+1}[\![F]\!]$.
- *For* $n \geq 0$ *the semantic function for* $\mathscr{V}_{n+1}$ *is defined as follows:*
  1. $\mathscr{V}_{n+1}[\![F]\!]\sigma = \mathscr{M}_{n+1}[\![F]\!]$ *if* $F \in \Sigma_n^0$,
  2. $\mathscr{V}_{n+1}[\![X_i]\!]\sigma = \sigma(i)$, $i \in I$,
  3. $\mathscr{V}_{n+1}[\![F_1 \wedge F_2]\!]\sigma = \mathscr{V}_{n+1}[\![F_1]\!]\sigma \cap \mathscr{V}_{n+1}[\![F_2]\!]\sigma$, $\mathscr{V}_{n+1}[\![F_1 \vee F_2]\!]\sigma = \mathscr{V}_{n+1}[\![F_1]\!]\sigma \cup \mathscr{V}_{n+1}[\![F_2]\!]\sigma$,
  4. $\mathscr{V}_{n+1}[\![\langle a\rangle F]\!]\sigma = \langle\cdot a\cdot\rangle\mathscr{V}_{n+1}[\![F]\!]\sigma$, $\mathscr{V}_{n+1}[\![[a]F]\!]\sigma = [\cdot a\cdot]\mathscr{V}_{n+1}[\![F]\!]\sigma$.

4.1 Characteristic Formulae by means of Declarations

The aim of this section is to show how each process $p \in \mathbf{P}$ can be characterized up to a binary relation $\bowtie$ over processes (such as an equivalence or a preorder) by a single formula, the so called characteristic formula for p up to $\bowtie$.

To achieve this, we take $I = \mathbf{P}$ in the definitions in the previous section. A declaration D for a variable logic $\mathscr{V}$ assigns exactly one formula $D(p)$ from $\mathscr{V}$ to each process $p \in \mathbf{P}$. We have seen that each such function induces an endofunction $\mathscr{V}[\![D]\!] \in \mathscr{P}(\mathbf{P})^{\mathbf{P}} \to_{mono} \mathscr{P}(\mathbf{P})^{\mathbf{P}}$ and therefore $\mathscr{V}[\![D]\!]$ exists. This leads to the following definition:

Definition 4.6 *A declaration D for the logic $\mathscr{V}$ characterizes $\bowtie \subseteq \mathbf{P} \times \mathbf{P}$ iff for each $p, q \in \mathbf{P}$,*

$$(p,q) \in \bowtie \text{ iff } q \in (\nu\mathscr{V}[\![D]\!])(p).$$

In what follows, we will describe how we can devise a characterizing declaration for a relation that is obtained as a fixed point, or a sequence of nested fixed points of monotone endofunctions, which can be expressed in the logic. In order to define this precisely we use the notation introduced in Definition 4.7 below.

Definition 4.7 *If $S \subseteq \mathbf{P} \times \mathbf{P}$ we define the variable interpretation $\sigma_S \in \mathscr{P}(\mathbf{P})^{\mathbf{P}}$ associated to S by*

$$\sigma_S(p) = \{q \in \mathbf{P} \mid (p,q) \in S\}, \text{ for each } p \in \mathbf{P}.$$

Thus σ_S assigns to p all those processes q that are related to it via S.

Definition 4.8 *A declaration D for $\mathscr{V}$* expresses *a monotone endofunction $\mathscr{F}$ on $\mathscr{P}(\mathbf{P} \times \mathbf{P})$ when*

$$(p,q) \in \mathscr{F}(S) \text{ iff } q \in \mathscr{V}[\![D(p)]\!]\sigma_S = (\mathscr{V}[\![D]\!]\sigma_S)(p),$$

for every relation $S \subseteq \mathbf{P} \times \mathbf{P}$ and every $p, q \in \mathbf{P}$.

We need the following to prove our main result.

Definition 4.9 *Let $\Phi : \mathscr{P}(\mathbf{P} \times \mathbf{P}) \to \mathscr{P}(\mathbf{P})^{\mathbf{P}}$ be defined by $\Phi(S) = \sigma_S$.*

Lemma 4.10

- *$\Phi : \mathscr{P}(\mathbf{P} \times \mathbf{P}) \to \mathscr{P}(\mathbf{P})^{\mathbf{P}}$ is an isomorphism.*
- *If $A_1, A_2 \in \mathscr{P}(\mathbf{P} \times \mathbf{P})$ and $\mathscr{F}_1, \mathscr{F}_2 \in \mathscr{P}(\mathbf{P} \times \mathbf{P}) \to_{mono} \mathscr{P}(\mathbf{P} \times \mathbf{P})$ then*
  - *$\Phi(A_1 \cap A_2) = \Phi(A_1) \cap \Phi(A_2)$,*
  - *$\Phi^*(\mathscr{F}_1 \cap A_1) = \Phi^*(\mathscr{F}_1) \cap \Phi(A_1)$ and*
  - *$\Phi^*(\mathscr{F}_1 \cap \mathscr{F}_2) = \Phi^*(\mathscr{F}_1) \cap \Phi^*(\mathscr{F}_2)$.*

Proof The first part is proved in [2] whereas the second part follows directly from the definition of Φ.

Corollary 4.11 *Assume that $D \in \mathbf{P} \to \mathscr{V}$ and $\mathscr{F} \in \mathscr{P}(\mathbf{P} \times \mathbf{P}) \to_{mono} \mathscr{P}(\mathbf{P} \times \mathbf{P})$. Then*

$$D \text{ expresses } \mathscr{F} \text{ iff } \Phi^*(\mathscr{F}) = \mathscr{V}[\![D]\!] \text{ iff } D \text{ characterizes } \nu\mathscr{F}.$$

5 Applications

Following the approach in [2], we define declarations D and $\tilde{D}$ that express the functions $\mathscr{F}$ and $\tilde{\mathscr{F}}$ that were defined in Section 3.

Definition 5.1 *Let*

- *Let $D : p \mapsto \bigwedge_{a \in \mathbf{A}} \bigwedge_{p' \in \mathbf{P}.\, p \xrightarrow{a} p'} \langle a \rangle X_{p'}$ and*
- *$\tilde{D} : p \mapsto \bigwedge_{a \in \mathbf{A}} [a] \bigvee_{p' \in \mathbf{P}.\, p \xrightarrow{a} p'} X_{p'}$.*

From [2] we have:

Lemma 5.2

- *D expresses $\mathscr{F}$ and characterizes $\nu\mathscr{F}$, and*
- *$\tilde{D}$ expresses $\tilde{\mathscr{F}}$ and characterizes $\nu\tilde{\mathscr{F}}$.*

Now we recall from [2] the declarations that characterize simulation equivalence and bisimulation equivalence.

Definition 5.3 *Define $D_{bisim} = D_{sim} \wedge D_{opsim}$ and $D_{simeq} = \nu D_{sim} \wedge \nu D_{opsim}$.*

Lemma 5.4 *D_{bisim} characterizes $\sim_{bisim}$ and D_{simeq} characterizes $\sim_{sim}$.*

Proof D_{bisim} does not contain nested fixed points and can therefore be interpreted directly over $\mathscr{V}_0 = \mathscr{V}$. Now we proceed as follows:

$$\Phi^*(\mathscr{F}_{bisim}) = \Phi^*(\mathscr{F}_{sim}) \cap \Phi^*(\mathscr{F}_{opsim}) = \mathscr{V}[\![D_{sim}]\!] \cap \mathscr{V}[\![D_{opsim}]\!] = \mathscr{V}[\![D_{sim} \wedge D_{opsim}]\!] = \mathscr{V}[\![D_{bisim}]\!].$$

To interpret D_{simeq} we define $\Sigma_1 = \{tt, ff\} \cup \{\nu D_{sim}(p) | p \in \mathbf{P}\}$ and $\Sigma_2 = \Sigma_1 \cup \{\nu D_{opsim}(p) | p \in \mathbf{P}\}$ and let $\mathscr{M}_0, \mathscr{M}_1, \mathscr{M}_2$ and $\mathscr{V}_0, \mathscr{V}_1$ be defined as before. Then $D_{simeq} : \mathbf{P} \to \mathscr{V}_1$. If we let $\mathscr{F}_{simeq} = \nu\mathscr{F}_{sim} \cap \nu\mathscr{F}_{opsim}$, we get

$$\Phi^*(\mathscr{F}_{simeq}) = \Phi(\nu\mathscr{F}_{sim}) \cap \Phi(\nu\mathscr{F}_{opsim)} = \nu\mathscr{V}_1[\![D_{sim}]\!] \cap \nu\mathscr{V}_1[\![D_{opsim}]\!] =$$

$$\mathscr{M}_2[\![\nu D_{sim}]\!] \cap \mathscr{M}_2[\![\nu D_{opsim}]\!] = \mathscr{M}_2[\![\nu D_{sim} \wedge \nu D_{opsim}]\!] = \mathscr{V}_1[\![D_{simeq}]\!].$$

The result now follows from Cor. 4.11.

Next we define the nested simulation preorders introduced in [5] by using the function $\mathscr{F}$. These definition involve nesting of fixed points and are defined recursively on the depth of the nesting. The 1-nested simulation $\sqsubseteq_{(1)sim}$ is just the simulation preorder $\sqsubseteq_{sim}$ as defined in Section 3 and the function $\mathscr{F}_{(1)sim}$ is therefore the function $\mathscr{F}$. As the preorder $\sqsubseteq_{(n+1)sim}$ depends on the inverse of the preorder $\sqsubseteq_{(n)sim}$, which we call $\sqsubseteq_{(n)opsim}$, we simultaneously define the nested simulations and their inverse in our recursive definition. The functions that define $\sqsubseteq_{(n)sim}$ and $\sqsubseteq_{(n)opsim}$ are called $\mathscr{F}_{(n)sim}$ and $\mathscr{F}_{(n)opsim}$ respectively.

Definition 5.5 (Nested simulations)

1. *$\mathscr{F}_{(1)sim} = \mathscr{F}$ and $\sqsubseteq_{(1)sim} = \nu\mathscr{F}_{(1)sim}$,*
2. *$\mathscr{F}_{(1)opsim} = \widetilde{\mathscr{F}}$ and $\sqsubseteq_{(1)opsim} = \nu\mathscr{F}_{(1)opsim}$,*
3. *$\mathscr{F}_{(n+1)sim} = \mathscr{F}_{(1)sim} \cap \nu\mathscr{F}_{(n)opsim}$ and $\sqsubseteq_{(n+1)sim} = \nu\mathscr{F}_{(n+1)sim}$.*
4. *$\mathscr{F}_{(n+1)opsim} = \mathscr{F}_{(1)opsim} \cap \nu\mathscr{F}_{(n)sim}$ and $\sqsubseteq_{(n+1)opsim} = \nu\mathscr{F}_{(n+1)opsim}$.*

We complete this note by defining a sequence of nested declarations and prove that they characterize the sequence of n-nested simulation preorders.

Theorem 5.6

1. *$D_{(1)sim} = D$ expresses $\mathscr{F}_{(1)sim}$ and characterizes $\sqsubseteq_{(1)sim}$,*
2. *$D_{(1)opsim} = \widetilde{D}$ expresses $\mathscr{F}_{(1)opsim}$ and characterizes $\sqsubseteq_{(1)opsim}$,*
3. *$D_{(n+1)sim} = D_{(1)sim} \wedge \nu D_{(n)opsim}$ expresses $\mathscr{F}_{(n+1)sim}$ and characterizes $\sqsubseteq_{(n+1)sim}$,*

4. $D_{(n+1)opsim} = D_{(1)opsim} \wedge \nu D_{(n)sim}$ *expresses* $\mathscr{F}_{(n+1)opsim}$ *and characterizes* $\sqsubseteq_{(n+1)opsim}$.

Proof We prove the statements simultaneously by induction on n. First we note that $D_1, D_2, \ldots,$ where $D_{2i-2} = D_{(i)sim}$ and $D_{2i-1} = D_{(i)opsim}$ for $i \geq 1$ is a sequence of nested declarations. For the case $n = 1$ we get from Lemma 5.2 that $\Phi^*(\mathscr{F}_{(1)sim}) = \mathscr{V}_0[\![D_{(1)sim}]\!]$ and $\Phi^*(\mathscr{F}_{(1)opsim}) = \mathscr{V}_1[\![D_{(1)opsim}]\!]$. Next assume that $\Phi^*(\mathscr{F}_{(n)sim}) = \mathscr{V}_{2n-2}[\![D_{(n)sim}]\!]$ and $\Phi^*(\mathscr{F}_{(n)opsim}) = \mathscr{V}_{2n-1}[\![D_{(n)opsim}]\!]$. To prove 3. we proceed as follows:

$$\Phi^*(\mathscr{F}_{(n+1)sim}) = \Phi^*(\mathscr{F}_{(1)sim}) \cap \Phi(\nu\mathscr{F}_{(n)opsim}) = \mathscr{V}_0[\![D_{(1)sim}]\!] \cap \nu\mathscr{V}_{2n-2}[\![D_{(n)opsim}]\!] = \mathscr{V}_{2n-2}[\![D_{(1)sim} \wedge \nu D_{(n)opsim}]\!] = \mathscr{V}_{2n}[\![D_{(n+1)sim}]\!].$$

Finally, to prove 4. we have:

$$\Phi^*(\mathscr{F}_{(n+1)opsim}) = \Phi^*(\mathscr{F}_{(1)opsem}) \cap \Phi(\nu\mathscr{F}_{(n)sim}) = \mathscr{V}_1[\![D_{(1)opsim}]\!] \cap \nu\mathscr{V}_{2n-1}[\![D_{(n)sim}]\!] = \mathscr{V}_{2n-1}[\![D_{(1)opsim} \wedge \nu D_{(n)sim}]\!] = \mathscr{V}_{2n+1}[\![D_{(n+1)opsim}]\!].$$

References

[1] L. Aceto, A. Ingolfsdottir, K.G. Larsen & J. Srba (2007): *Reactive Systems: Modelling, Specification and Verification*. Cambridge University Press, doi:10.1017/CBO9780511814105.

[2] L. Aceto, A. Ingolfsdottir, P. B. Levy & J. Sack (2012): *Characteristic Formulae for Fixed-Point Semantics: A General Framework*. *Mathematical Structures in Computer Science* doi:10.4204/EPTCS.8.1. Special issue devoted to selected papers from EXPRESS 2009, Cambridge University Press.

[3] Jan Bergstra, Alban Ponse & Scott A. Smolka, editors (2001): *Handbook of Process Algebra*. Elsevier.

[4] R. van Glabbeek (2001): *The linear time–branching time spectrum. I. The semantics of concrete, sequential processes*. In Bergstra et al. [3], pp. 3–99, doi:10.1016/B978-044482830-9/50019-9.

[5] Jan Friso Groote & Frits W. Vaandrager (1992): *Structured Operational Semantics and Bisimulation as a Congruence*. *Information and Computation* 100(2), pp. 202–260, doi:10.1016/0890-5401(92)90013-6.

[6] M. Hennessy & R. Milner (1985): *Algebraic laws for nondeterminism and concurrency*. *Journal of the ACM* 32(1), pp. 137–161, doi:10.1145/2455.2460.

[7] R.M. Keller (1976): *Formal verification of parallel programs*. *Communications of the ACM* 19(7), pp. 371–384, doi:10.1145/360248.360251.

[8] Dexter Kozen (1983): *Results on the Propositional mu-Calculus*. *Theoretical Computer Science* 27, pp. 333–354, doi:10.1016/0304-3975(82)90125-6.

[9] Kim Guldstrand Larsen (1990): *Proof Systems for Satisfiability in Hennessy–Milner Logic with Recursion*. *Theoretical Computer Science* 72(2–3), pp. 265–288, doi:10.1016/0304-3975(90)90038-J.

[10] A. Tarski (1955): *A Lattice-Theoretical Fixpoint Theorem and its Applications*. *Pacific Journal of Mathematics* 5(2), pp. 285–309. Available at `http://projecteuclid.org/euclid.pjm/1103044538`.

IO vs OI in Higher-Order Recursion Schemes

Axel Haddad
LIAFA (Université Paris 7 & CNRS) & LIGM (Université Paris Est & CNRS)

We propose a study of the modes of derivation of higher-order recursion schemes, proving that value trees obtained from schemes using innermost-outermost derivations (IO) are the same as those obtained using unrestricted derivations.

Given that higher-order recursion schemes can be used as a model of functional programs, innermost-outermost derivations policy represents a theoretical view point of call by value evaluation strategy.

1 Introduction

Recursion schemes have been first considered as a model of computation, representing the syntactical aspect of a recursive program [15, 2, 3, 4] . At first, (order-1) schemes were modelling simple recursive programs whose functions only take values as input (and not functions). Since, higher-order versions of recursion schemes [11, 5, 6, 7, 8, 9] have been studied.

More recently, recursion schemes were studied as generators of infinite ranked trees and the focus was on deciding logical properties of those trees [12, 8, 10, 1, 13, 14].

As for programming languages, the question of the evaluation policy has been widely studied. Indeed, different policies results in the different evaluation [8, 9, 7]. There are two main evaluations policy for schemes: outermost-innermost derivations (OI) and inner-outermost IO derivations, respectively corresponding to call by need and call by value in programming languages.

Standardization theorem for the lambda-calculus shows that for any scheme, outermost-innermost derivations (OI) lead to the same tree as unrestricted derivation. However, this is not the case for IO derivations. In this paper we prove that the situation is different for schemes. Indeed, we establish that the trees produced using schemes with IO policy are the same as those produced using schemes with OI policy. For a given a scheme of order n, we can use a simplified continuation passing style transformation, to get a new scheme of order $n+1$ in which IO derivations will be the same as OI derivations in the initial scheme (Section 3). Conversely, in order to turn a scheme into another one in which unrestricted derivations lead to the same tree as IO derivations in the initial scheme, we adapt Kobayashi's [13] recent results on HORS model-checking, to compute some key properties over terms (Section 4.1). Then we embed these properties into a scheme turning it into a self-correcting scheme of the same order of the initial scheme, in which OI and IO derivations produce the same tree (Section 4.2).

2 Preliminaries

Types are defined by the grammar $\tau ::= o \mid \tau \to \tau$; o is called the ***ground type***. Considering that $\to$ is associative to the right (i.e. $\tau_1 \to (\tau_2 \to \tau_3)$ can be written $\tau_1 \to \tau_2 \to \tau_3$), any type τ can be written uniquely as $\tau_1 \to ... \to \tau_k \to o$. The integer k is called the ***arity*** of τ. We define the ***order of a type*** by $\text{order}(o) = 0$ and $\text{order}(\tau_1 \to \tau_2) = \max(\text{order}(\tau_1)+1, \text{order}(\tau_2))$. For instance $o \to o \to o \to o$ is a type

D. Miller and Z. Ésik (Eds.): Fixed Points
in Computer Science 2012 (FICS 2012)
EPTCS 77, 2012, pp. 23–30, doi:10.4204/EPTCS.77.4

of order 1 and arity 3, $(o \to o) \to (o \to o)$, that can also be written $(o \to o) \to o \to o$ is a type of order 2. Let $\tau^\ell \to \tau'$ be a shortcut for $\underbrace{\tau \to ... \to \tau}_{\ell \text{ times}} \to \tau'$.

Let Γ be a finite set of symbols such that to each symbol is associated a type. Let Γ^τ denote the set of symbols of type τ. For all type τ, we define the set of ***terms*** of type $\mathscr{T}^\tau(\Gamma)$ as the smallest set satisfying: $\Gamma^\tau \subseteq \mathscr{T}^\tau(\Gamma)$ and $\bigcup_{\tau'}\{t\ s \mid t \in \mathscr{T}^{\tau' \to \tau}(\Gamma), s \in \mathscr{T}^{\tau'}(\Gamma)\} \subseteq \mathscr{T}^\tau(\Gamma)$. If a term t is in $\mathscr{T}^\tau(\Gamma)$, we say that t has type τ. We shall write $\mathscr{T}(\Gamma)$ as the set of terms of any type, and $t : \tau$ if t has type τ. The arity of a term t, $arity(t)$, is the arity of its type. Remark that any term t can be uniquely written as $t = \alpha\ t_1 ... t_k$ with $\alpha \in \Gamma$. We say that α is the ***head*** of the term t. For instance, let $\Gamma = \{F : (o \to o) \to o \to o\ ,\ G : o \to o \to o\ ,\ H : (o \to o)\ ,\ \mathtt{a} : o\}$: $F\ H$ and $G\ \mathtt{a}$ are terms of type $o \to o$; $F(G\ \mathtt{a})\ (H\ (H\ a))$ is a term of type o; $F\ \mathtt{a}$ is not a term since F is expecting a first argument of type $o \to o$ while $\mathtt{a}$ has type o.

Let $t : \tau$, $t' : \tau'$ be two terms, $x : \tau'$ be a symbol of type τ', then we write $t_{[x \mapsto t']} : \tau$ the term obtained by substituting all occurences of x by t' in the term t. A ***τ-context*** is a term $C[\bullet^\tau] \in \mathscr{T}(\Gamma \uplus \{\bullet^\tau : \tau\})$ containing exactly one occurrence of $\bullet^\tau$; it can be seen as an application turning a term into another, such that for all $t : \tau$, $C[t] = C[\bullet^\tau]_{[\bullet^\tau \mapsto t]}$. In general we will only talk about ground type context where $\tau = o$ and we will omit to specify the type when it is clear. For instance, if $C[\bullet] = F \bullet\ (H\ (H\ a))$ and $t' = G\ \mathtt{a}$ then $C[t'] = F\ (G\ \mathtt{a})\ (H\ (H\ \mathtt{a}))$.

Let Σ be a set of symbols of order at most 1 (i.e. each symbols has type o or $o \to ... \to o$) and $\bot : o$ be a fresh symbol. A ***tree*** t over $\Sigma \uplus \bot$ is a mapping $t : dom^t \to \Sigma \uplus \bot$, where dom^t is a prefix-closed subset of $\{1, ..., m\}^*$ such that if $u \in dom^t$ and $t(u) = a$ then $\{j \mid uj \in dom^t\} = \{1, ..., arity(a)\}$. Note that there is a direct bijection between ground terms of $\mathscr{T}^o(\Sigma \uplus \bot)$ and finite trees . Hence we will freely allow ourselves to treat ground terms over $\Sigma \uplus \bot$ as trees. We define the partial order $\sqsubseteq$ over trees as the smallest relation satisfying $\bot \sqsubseteq t$ and $t \sqsubseteq t$ for any tree t, and $a\ t_1 ... t_k \sqsubseteq a\ t'_1 ... t'_k$ iff $t_i \sqsubseteq t'_i$. Given a (possibly infinite) sequence of trees $t_0, t_1, t_2, ...$ such that $t_i \sqsubseteq t_{i+1}$ for all i, one can prove that the set of all t_i has a supremum that is called the ***limit tree*** of the sequence.

A ***higher order recursion scheme (HORS)*** $G = \langle \mathscr{V}, \Sigma, \mathscr{N}, \mathscr{R}, S \rangle$ is a tuple such that: $\mathscr{V}$ is a finite set of typed symbols called ***variables***; Σ is a finite set of typed symbols of order at most 1, called the ***set of terminals***; $\mathscr{N}$ is a finite set of typed symbols called ***set of non-terminals***; $\mathscr{R}$ is a set of ***rewrite rules***, one per non terminal $F : \tau_1 \to ... \to \tau_k \to o \in \mathscr{N}$, of the form $F\ x_1\ ...\ x_k\ \to\ e$ with $e : o \in \mathscr{T}(\Sigma \uplus \mathscr{N} \uplus \{x_1, ..., x_k\})$; $S \in \mathscr{N}$ is the ***initial non-terminal***.

We define the ***rewriting relation*** $\to_G \in\ \mathscr{T}(\Sigma \uplus \mathscr{N})^2$ (or just $\to$ when G is clear) as $t \to_G t'$ iff there exists a context $C[\bullet]$, a rewrite rule $F\ x_1 ... x_k \to e$, and a term $F\ t_1\ ...\ t_k : o$ such that $t = C[F\ t_1 ... t_k]$ and $t' = C[e_{[x_1 \mapsto t_1] ... [x_k \mapsto t_k]}]$. We call $F\ t_1\ ...\ t_k : o$ a ***redex***. Finally we define $\to_G^*$ as the reflexive and transitive closure of $\to_G$.

We define inductively the ***$\bot$-transformation*** $(\cdot)^\bot\ :\ \mathscr{T}^o(\mathscr{N} \uplus \Sigma) \to \mathscr{T}^o(\Sigma \uplus \{\bot : o\})$: $(F\ t_1\ ...\ t_k)^\bot\ =\ \bot\ \forall F \in \mathscr{N}$ and $(a\ t_1\ ...\ t_k)^\bot\ =\ a\ t_1^\bot ... t_k^\bot$ for all $a \in \Sigma$. We define a ***derivation***, as a possibly infinite sequence of terms linked by the rewrite relation. Let $t_0 = S \to_G t_1 \to_G t_2 \to_G ...$ be a derivation, then one can check that $(t_0)^\bot \sqsubseteq (t_1)^\bot \sqsubseteq (t_2)^\bot \sqsubseteq ...$, hence it admits a limit. One can prove that the set of all such limit trees has a greatest element that we denote $\|G\|$ and refer to as the ***value tree*** of G. Note that $\|G\|$ is the supremum of $\{t^\bot \mid S \to^* t\}$. Given a term $t : o$, we denote by G_t the scheme obtained by transforming G such that it starts derivations with the term t, formally, $G_t = \langle \mathscr{V}, \Sigma, \mathscr{N} \uplus \{S'\}, \mathscr{R} \uplus \{S' \to t\}, S' \rangle$. One can prove that if $t \to t'$ then $\|G_t\| = \|G_{t'}\|$.

Example. Let $G = \langle \mathscr{V}, \Sigma, \mathscr{N}, \mathscr{R}, S \rangle$ be the scheme such that: $\mathscr{V} = \{x : o, \phi : o \to o, \psi : (o \to o) \to o \to o\}$, $\Sigma = \{\mathtt{a} : o^3 \to o, \mathtt{b} : o \to o \to o, \mathtt{c} : o\}$, $\mathscr{N} = \{F : \big((o \to o) \to o \to o\big) \to (o \to o) \to o \to o, H : (o \to$

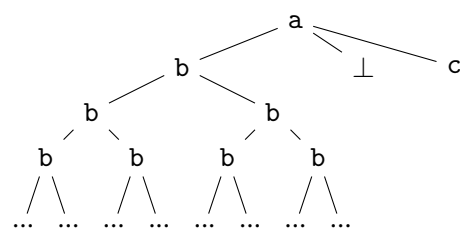
$o) \to o \to o, I, J, K : o \to o, S : o\}$, and $\mathscr{R}$ contains the following rewrite rules:

$$\begin{array}{lcl@{\qquad}lcl@{\qquad}lcl} F\ \psi\ \phi\ x & \to & \psi\ \phi\ x & I\ x & \to & x & H\ \phi\ x & \to & \mathtt{a}\ (J\ x)\ (K\ x)\ (\phi\ x) \\ J\ x & \to & \mathtt{b}\ (J\ x)\ (J\ x) & K\ x & \to & K\ (K\ x) & S & \to & F\ H\ I\ \mathtt{c} \end{array}$$

Here is an example of finite derivation:

$$\begin{array}{lllllll} S & \to & F\ H\ I\ \mathtt{c} & \to & H\ I\ \mathtt{c} & \to & \mathtt{a}\ (J\ \mathtt{c})\ (K\ \mathtt{c})\ (I\ \mathtt{c}) \\ & & & & & \to & \mathtt{a}\ (J\ \mathtt{c})\ (K\ (K\ \mathtt{c}))\ (I\ \mathtt{c}) \quad \to \quad \mathtt{a}\ (J\ \mathtt{c})\ (K\ (K\ (K\ \mathtt{c})))\ (I\ \mathtt{c}) \end{array}$$

If one extends it by always rewriting a redex of head K, its limit is the tree $\mathtt{a}\ \bot\ \bot\ \bot$, but this is not the value tree of G. The value tree $\|G\|$ is depicted below.



Evaluation Policies

We now put constraints on the derivations we allow. If there are no constraints, then we say that the derivations are unrestricted and we let $\mathrm{Acc}^G = \{t : o \mid S \to^* t\}$ be the set of accessible terms using unrestricted derivations. Given a rewriting $t \to t'$ such that $t = C[F\ s_1\ ...\ s_k]$ and $t' = C[e_{[\forall j\ x_j \mapsto s_j]}]$ with $F\ x_1 ... x_k \to e \in \mathscr{R}$.

- We say that $t \to t'$ is an ***outermost-innermost*** (OI) rewriting (written $t \to_{OI} t'$) there is no redex containing the occurrence of $\bullet$ as a subterm of $C[\bullet]$.
- We say that $t \to t'$ is an ***innermost-outermost*** (IO) rewriting (written $t \to_{IO} t'$), if for all j there is no redex as a subterm of s_j.

Let $\mathrm{Acc}^G_{OI} = \{t : o \mid S \to^*_{OI} t\}$ be the set of accessible terms using OI derivations and $\mathrm{Acc}^G_{IO} = \{t : o \mid S \to^*_{IO} t\}$ be the set of accessible terms using IO derivations. There exists a supremum of Acc^G_{OI} (resp. Acc^G_{IO}) which is the maximum of the limit trees of OI derivations(resp. IO derivations). We write it $\|G\|_{OI}$ (resp. $\|G\|_{IO}$). For all recursive scheme G, $(\mathrm{Acc}^G)^\bot = (\mathrm{Acc}^G_{OI})^\bot$, in particular $\|G\|_{OI} = \|G\|$. But $\|G\|_{IO} \sqsubseteq \|G\|$ and in general, the equality does not hold (see the example is the next section).

3 From OI to IO

Fix a recursion scheme $G = \langle \mathscr{V}, \Sigma, \mathscr{N}, \mathscr{R}, S\rangle$. Our goal is to define another scheme $\overline{G} = \langle \overline{\mathscr{V}}, \Sigma, \overline{\mathscr{N}}, \overline{\mathscr{R}}, I\rangle$ such that $\|\overline{G}\|_{IO} = \|G\|$. The idea is to add an extra argument (Δ) to each non terminal, that will be required to rewrite it (hence the types are changed). We feed this argument to the outermost non terminal, and duplicate it to subterms only if the head of the term is a terminal. Hence all derivations will be IO-derivations.

We define the $\overline{(\cdot)}$ ***transformation*** over types by $\overline{o} = o \to o$, and $\overline{\tau_1 \to \tau_2} = \overline{\tau_1} \to \overline{\tau_2}$. In particular, if $\tau = \tau_1 \to ... \to \tau_k \to o$ then $\overline{\tau} = \overline{\tau_1} \to ... \to \overline{\tau_k} \to o \to o$. Note that for all τ, $\mathrm{order}(\overline{\tau}) = \mathrm{order}(\tau) + 1$.

For all $x:\tau \in \mathscr{V}$ we define $\overline{x}:\overline{\tau}$ as a fresh variable. Let ar_{max} be the maximum arity of terminals, we define $\eta_1,...,\eta_{arity_{max}}:o\to o$ and $\delta:o$ as fresh variables, and we let $\overline{\mathscr{V}}=\{\overline{x}:\overline{\tau}\mid x\in\mathscr{V}\}\uplus\{\eta_1,...,\eta_{ar_{max}}\}\uplus\{\delta:o\}$. Note that δ is the only variable of type o. For all $a:\tau\in\Sigma$ define $\overline{a}:\overline{\tau}$ as a fresh **non-terminal** and for all $F:\tau\in\mathscr{N}$ define $\overline{F}:\overline{\tau}$ as a fresh non-terminal. Let $\overline{\mathscr{N}}=\{\overline{a}:\overline{\tau}\mid a\in\Sigma\}\uplus\{\overline{F}:\overline{\tau}\mid F\in\mathscr{N}\}\uplus\{\Delta:o,I:o\}$. Note that I and Δ are the only symbols in $\overline{\mathscr{N}}$ of type o.

Let $t:\tau\in\mathscr{T}(\mathscr{V}\uplus\Sigma\uplus\mathscr{N})$, we define inductively the term $\overline{t}:\overline{\tau}\in\mathscr{T}(\overline{\mathscr{V}}\uplus\overline{\mathscr{N}})$: If $t=x\in\mathscr{V}$ (resp. $t=a\in\Sigma$, $t=F\in\mathscr{N}$), we let $\overline{t}=\overline{x}\in\overline{\mathscr{V}}$ (resp. $\overline{t}=\overline{a}\in\overline{\Sigma}$, $\overline{t}=\overline{F}\in\mathscr{N}$), if $t=t_1\ t_2:\tau$ then $\overline{t}=\overline{t_1}\ \overline{t_2}$.

Let $F\ x_1\ ...\ x_k\to e$ be a rewrite rule of $\mathscr{R}$. We define the (valid) rule $\overline{F}\ \overline{x_1}\ ...\ \overline{x_k}\ \delta\ \to\ \overline{e}\ \Delta$ in $\overline{\mathscr{R}}$. Let $a\in\Sigma$ of arity k, we define the rule $\overline{a}\ \eta_1\ ...\ \eta_k\ \delta\ \to\ a\ (\eta_1\ \Delta)\ ...\ (\eta_k\ \Delta)$ in $\overline{\mathscr{R}}$. We also add the rule $I\ \to\ \overline{S}\ \Delta$ to $\overline{\mathscr{R}}$. Finally let $\overline{G}=\langle\overline{\mathscr{V}},\Sigma,\overline{\mathscr{N}},\overline{\mathscr{R}},I\rangle$.

Example. Let $G=\langle\mathscr{V},\Sigma,\mathscr{N},\mathscr{R},S\rangle$ be the order-1 recursion scheme with $\Sigma=\{\mathtt{a},\mathtt{c}:o\}$, $\mathscr{N}=\{S:o,F:o\to o\to o,H:o\to o\}$, $\mathscr{V}=\{x,y:o\}$, and the following rewrite rules:

$$S\ \to\ F\ (H\ \mathtt{a})\ \mathtt{c}\qquad F\ x\ y\ \to\ y\qquad H\ x\ \to\ H\ (H\ x)$$

Then we have $\|G\|_{OI}=\mathtt{c}$ while $\|G\|_{IO}=\bot$ (indeed, the only IO derivation is the following $S\to F\ (Ha)\ \mathtt{c}\to F\ (H\ (H\ a))\ \mathtt{c}\to F\ (H\ (H\ (H\ a)))\ \mathtt{c}\to ...$). The order-2 recursion scheme $\overline{G}=\langle\overline{\mathscr{V}},\Sigma,\overline{\mathscr{N}},\overline{\mathscr{R}},I\rangle$ is given by $\overline{\mathscr{N}}=\{I,\Delta:o,\overline{S},\overline{\mathtt{a}},\overline{\mathtt{c}}:o\to o,\overline{F}:(o\to o)\to(o\to o)\to o\to o,\overline{H}:(o\to o)\to o\to o\}$,$\overline{\mathscr{V}}=\{\delta:o,\overline{x},\overline{y}:o\to o\}$ and the following rewrite rules:

$$\begin{array}{lcl@{\qquad}lcl@{\qquad}lcl}
I & \to & \overline{S}\ \Delta & \overline{S}\ \delta & \to & \overline{F}\ (\overline{H}\ \overline{a})\ \overline{c}\ \Delta & \overline{F}\ \overline{x}\ \overline{y}\ \delta & \to & \overline{y}\ \Delta\\
\overline{H}\ \overline{x}\ \delta & \to & \overline{H}\ (\overline{H}\ \overline{x})\ \Delta & \overline{\mathtt{c}}\ \delta & \to & \mathtt{c} & \overline{\mathtt{a}}\ \delta & \to & \mathtt{a}
\end{array}$$

Note that in the term $\overline{F}\ (\overline{H}\ \overline{\mathtt{a}})\ \overline{\mathtt{c}}\ \Delta$, the subterm $\overline{H}\ \overline{\mathtt{a}}$ is no longer a redex since it lacks its last argument, hence it cannot be rewritten, then the only IO derivation, which is the only unrestricted derivation is $I\to\overline{S}\ \Delta\to\overline{F}\ (\overline{H}\ \overline{\mathtt{a}})\ \overline{\mathtt{c}}\ \Delta\to\overline{\mathtt{c}}\ \Delta\to\mathtt{c}$. Therefore $\|\bar{G}\|_{IO}=\|\bar{G}\|=\mathtt{c}=\|G\|$.

Lemma 1. *Any derivation of $\overline{G}$ is in fact an OI **and** an IO derivation. Hence that $\|\overline{G}\|_{IO}=\|\overline{G}\|$.*

Proof (Sketch). The main idea is that the only redexes will be those that have Δ as last argument of the head non-terminal. The scheme is constructed so that Δ remains only on the outermost non-terminals, that is why any derivation is an OI derivation. Furthermore, we have that if $t=\overline{F}\ t_1...t_k\Delta$ is a redex, then none of the t_i contains Δ, therefore they do not contain any redex, hence t is an innermost redex. □

Note that *OI* derivations in $\overline{G}$ acts like *OI* derivations in G, hence $\|G\|=\|\overline{G}\|$.

Theorem 2 (OI vs IO)**.** *Let G be an order-n scheme. Then one can construct an order-$(n+1)$ scheme $\overline{G}$ such that $\|G\|=\|\overline{G}\|_{IO}$.*

4 From IO to OI

The goal of this section is to transform the scheme G into a scheme G'' such that $\|G''\|=\|G\|_{IO}$. The main difference between IO and OI derivations is that some redex would lead to $\bot$ in IO derivation while OI derivations could be more productive. For example take $F:o\to o$ such that $F\ x\to c$, and $H:o$ such that $H\to a\ H$, with $a:o\to o$ and $c:o$ being terminal symbols. The term $F\ H$ has a unique *OI* derivation, $F\ H\to_{OI} c$, it is finite and it leads to the value tree assiocated. On the other hand, the (unique) IO derivation is the following $F\ H\to F(a\ H)\to F\ (a\ (a\ H))\to ...$ which leads to the tree $\bot$.

The idea of the transformation is to compute a tool (based on a type system) that decides if a redex would produce $\bot$ with *IO* derivations (Section 4.1); then we embed it into G and force any such redex to produce $\bot$ even with unrestricted derivations (Section 4.2).

4.1 The Type System

Given a term $t : \tau \in \mathscr{T}(\Sigma \uplus \mathscr{N})$, we define the two following properties on t: $\mathscr{P}_\perp(t) =$"The term t has type o and its associated IO valuation tree is $\perp$", and $\mathscr{P}_\infty(t) =$"the term t has not necessarily ground type, it contains a redex r such that any IO derivation from r producing it's IO valuation tree is infinite". Note that $\mathscr{P}_\infty(t)$ is equivalent to "the term t contains a redex r such that $\|G_r\|_{IO}$ is either infinite or contains $\perp$". In this section we describe a type system, inspired from the work of Kobayashi [13], that characterises if a term verifies these properties.

Let Q be the set $\{q_\perp, q_\infty\}$. Given a type τ, we define inductively the sets $(\tau)^{atom}$ and $(\tau)^\wedge$ called respectively set of atomic mappings and set of conjunctive mappings:
$(o)^{atom} = Q$, $(o)^\wedge = \{\bigwedge\{\theta_1,...,\theta_i\} \mid \theta_1,...,\theta_i \in Q\}$, $(\tau_1 \to \tau_2)^{atom} = \{q_\infty\} \uplus \{(\tau_1)^\wedge \to (\tau_2)^{atom}\}$
$(\tau_1 \to \tau_2)^\wedge = \{\bigwedge\{\theta_1,...,\theta_i\} \mid \theta_1,...,\theta_i \in (\tau_1 \to \tau_2)^{atom}\}$.

We will usually use the letter θ to represents atomic mappings, and the letter σ to represent conjunctive mappings. Given a conjunctive mapping σ (resp. an atomic mapping θ) and a type τ, we write $\sigma :: \tau$ (resp. $\theta ::_a \tau$) the relation $\sigma \in (\tau)^\wedge$ (resp. $\theta \in (\tau)^{atom}$). For the sake of simplicity, we identify the atomic mapping θ with the conjunctive mapping $\bigwedge\{\theta\}$.

Given a term t and a conjunctive mapping σ, we define a judgment as a tuple $\Theta \vdash t \triangleright \sigma$, pronounce "from the environment Θ, one can prove that t matches the conjunctive mapping σ", where the environment Θ is a partial mapping from $\mathscr{V} \uplus \mathscr{N}$ to conjunctive mapping. Given an environment Θ, $\alpha \in \mathscr{V} \uplus \mathscr{N}$ and a conjunctive mapping σ, we define the environment $\Theta' = \Theta, \alpha \triangleright \sigma$ as $Dom(\Theta') = Dom(\Theta) \cup \{\alpha\}$ and $\Theta'(\alpha) = \sigma$ if $\alpha \notin Dom(\Theta)$, $\Theta'(\alpha) = \sigma \wedge \Theta(\alpha)$ otherwise, and $\Theta'(\beta) = \Theta(\beta)$ if $\beta \neq \alpha$.

We define the following judgement rules:

$$\frac{\Theta \vdash t \triangleright \theta_1 \quad ... \quad \Theta \vdash t \triangleright \theta_n}{\Theta \vdash t \triangleright \bigwedge\{\theta_1,...,\theta_n\}}(Set) \qquad \frac{}{\Theta, \alpha \triangleright \bigwedge\{\theta_1,...,\theta_n\} \vdash \alpha \triangleright \theta_i}(At) \text{ (for all } i)$$

$$\frac{}{\Theta \vdash a \triangleright \sigma_1 \to ... \to \sigma_{i \leq arity(a)} \to q_\infty}(\Sigma) \text{ (for } a \in \Sigma \text{ and } \exists j\ \sigma_j = q_\infty \text{)}$$

$$\frac{\Theta \vdash t_1 \triangleright \sigma \to \theta \quad \Theta \vdash t_2 \triangleright \sigma}{\Theta \vdash t_1\ t_2 \triangleright \theta}(App) \quad \frac{}{\Theta \vdash t \triangleright q_\infty \to q_\infty}(q_\infty \to q_\infty\ I) \text{ (if } t : \tau_1 \to \tau_2) \quad \frac{\Theta \vdash t_1 \triangleright q_\infty}{\Theta \vdash t_1\ t_2 \triangleright q_\infty}(q_\infty\ I)$$

Remark that there is no rules that directly involves $q_\perp$, but it does not mean that no term matches $q_\perp$, since it can appear in Θ. Rules like (At) or (App) may be used to state that a term matches $q_\perp$.

We say that (G,t) matches the conjunctive mapping σ written $\vdash (G,t) \triangleright \sigma$ if there exists an environment Θ, called a witness environment of $\vdash (G,t) \triangleright \sigma$, such that (1) $Dom(\Theta) = \mathscr{N}$, (2) $\forall F : \tau \in \mathscr{N}\ \Theta(F) :: \tau$, (3) if $F\ x_1...x_k \to e \in \mathscr{R}$ and $\Theta \vdash F \triangleright \sigma_1 \to ... \to \sigma_{i \leq k} \to q$ then either there exists j such that $q_\infty \in \sigma_j$, or $i = k$ and $\Theta, x_1 \triangleright \sigma_1, ..., x_k \triangleright \sigma_k \vdash e \triangleright q$, (4) $\Theta \vdash t \triangleright \sigma$.

The following two results state that this type system matches the properties $\mathscr{P}_\perp$ and $\mathscr{P}_\infty$ and furthermore we can construct a universal environment, $\Theta^\star$, that can correctly judge any term.

Theorem 3 (Soundness and Completeness)**.** *Let G be an HORS, and t be term (of any type), $\vdash (G,t) \triangleright q_\infty$ (resp. $\vdash (G,t) \triangleright q_\perp$) if and only if $\mathscr{P}_\infty(t)$ (resp. $\mathscr{P}_\perp(t)$) holds.*

Proposition 4 (Universal Witness)**.** *There exists an environment $\Theta^\star$ such that for all term t, the judgment $\vdash (G,t) \triangleright \sigma$ holds if and only if $\Theta^\star \vdash t \triangleright \sigma$.*

Proof (Sketch). To compute $\Theta^\star$, we start with an environment Θ_0 satisfying Properties (1) and (2) ($Dom(\Theta_0) = \mathscr{N}$ and $\forall F : \tau \in \mathscr{N}\ \Theta_0(F) :: \tau$) that is able to judge any term $t : \tau$ with any conjunctive mapping $\sigma :: \tau$.

Then let $\mathscr{F}$ be the mapping from the set of environments to itself, such that for all $F : \tau_1 \to ... \to \tau_k \to o \in \mathscr{N}$, if $F\ x_1 ... x_k \to e \in \mathscr{R}$ then,

$$\begin{aligned}\mathscr{F}(\Theta)(F) = \{\sigma_1 \to ... \to \sigma_k \to q \mid q \in Q \wedge \forall i\ \sigma_i :: \tau_i \wedge \Theta, x_1 \triangleright \sigma_1, ..., x_k \triangleright \sigma_k \vdash e : q\} \\ \cup \{\sigma_1 \to ... \to \sigma_{i \leq k} \to q_\infty \mid \wedge \forall i\ \sigma_i :: \tau_i \wedge \exists j\ q_\infty \in \sigma_j\} \\ \cup \{\sigma_1 \to ... \to \sigma_k \to q_\bot \mid \forall i\ \sigma_i :: \tau_i \wedge \exists j\ q_\infty \in \sigma_j\}.\end{aligned}$$

We iterate $\mathscr{F}$ until we reach a fixpoint. The environment we get is $\Theta^\star$, it verifies properties (1) (2) and (3). Furthermore we can show that this is the maximum of all environment satisfying these properties, i.e. if $\vdash (G,t) \triangleright \sigma$ then $\Theta^\star \vdash t \triangleright \sigma$. □

4.2 Self-Correcting Scheme

For all term $t : \tau \in \mathscr{T}(\Sigma \uplus \mathscr{N})$, we define $[\![t]\!] \in (\tau)^\wedge$, called the semantics of t, as the conjunction of all atomic mappings θ such that $\Theta^\star \vdash t \triangleright \theta$ (recall that $\Theta^\star$ is the environment of Proposition 4). In particular $\mathscr{P}_\bot(t)$ (resp. $\mathscr{P}_\infty(t)$) holds if and only if $q_\bot \in [\![t]\!]$ (resp. $q_\infty \in [\![t]\!]$). Given two terms $t_1 : \tau_2 \to \tau$ and $t_2 : \tau_2$ the only rules we can apply to judge $\Theta^\star \vdash t_1\ t_2 \triangleright \theta$ are (App), $(q_\infty \to q_\infty\ I)$ and $(q_\infty\ I)$. We see that θ only depends on which atomic mappings are matched by t_1 and t_2. In other words $[\![t_1\ t_2]\!]$ only depends on $[\![t_1]\!]$ and $[\![t_2]\!]$, we write $[\![t_1]\!]\ [\![t_2]\!] = [\![t_1\ t_2]\!]$.

In this section, given a scheme $G = \langle \mathscr{V}, \Sigma, \mathscr{N}, \mathscr{R}, S\rangle$, we transform it into $G' = \langle \mathscr{V}', \Sigma, \mathscr{N}', \mathscr{R}', S\rangle$ which is basically the same scheme except that while it is producing an IO derivation, it evaluates $[\![t']\!]$ for any subterm t' of the current term and label t' with $[\![t']\!]$. Note that if $t \to_{IO} t'$, then $[\![t]\!] = [\![t']\!]$. Since we cannot syntactically label terms, we will label all symbols by the semantics of their arguments, *e.g.* if we want to label $F\ t_1 ... t_k$, we will label F with the k-tuple $([\![t_1]\!], ..., [\![t_k]\!])$.

A problem may appear if some of the arguments are not fully applied, for example imagine we want to label $F\ H$ with $H : o \to o$. We will label F with $[\![H]\!]$, but since H has no argument we do not know how to label it. The problem is that we cannot wait to label it because once a non-terminal is created, the derivation does not deal explicitly with it. The solution is to create one copy of H per possible semantics for its argument (here there are four of them: $\bigwedge\{\}, \bigwedge\{q_\bot\}, \bigwedge\{q_\infty\}, \bigwedge\{q_\bot, q_\infty\}$). This means that $F^{[\![H]\!]}$ would not have the same type as F: F has type $(o \to o) \to o$, but $F^{[\![G]\!]}$ will have type $(o \to o)^4 \to o$. Hence, $F\ H$ will be labelled the following way: $F^{[\![H]\!]}\ H^{\wedge\{\}} H^{\wedge\{q_\bot\}} H^{\wedge\{q_\infty\}} H^{\wedge\{q_\bot, q_\infty\}}$. Note that even if F has 4 arguments, it only has to be labelled with one semantics since all four arguments represent different labelling of the same term. We now formalize these notions.

Let us generalize the notion of semantics to deals with terms containing some variables. Given an environment on the variables $\Theta^{\mathscr{V}}$ such that $Dom(\Theta^{\mathscr{V}}) \subseteq \mathscr{V}$ and if $x : \tau$ then $\Theta^{\mathscr{V}}(x) :: \tau$, and given a term $t : \tau \in \mathscr{T}(\Sigma \uplus \mathscr{N} \uplus Dom(\Theta^{\mathscr{V}}))$, we define $[\![t]\!]_{\Theta^{\mathscr{V}}} \in (\tau)^\wedge$, as the conjunction of all atomic mappings θ such that $\Theta^\star, \Theta^{\mathscr{V}} \vdash t \triangleright \theta$. Given two terms $t_1 : \tau_2 \to \tau$ and $t_2 : \tau_2$ we still have that $[\![t_1\ t_2]\!]_{\Theta^{\mathscr{V}}}$ only depends on $[\![t_1]\!]_{\Theta^{\mathscr{V}}}$ and $[\![t_2]\!]_{\Theta^{\mathscr{V}}}$.

To a type $\tau = \tau_1 \to ... \to \tau_k \to o$ we associate the integer $\lceil \tau \rceil = Card(\{(\sigma_1, ..., \sigma_k) \mid \forall i\ \sigma_i \in (\tau_i)^\wedge\})$ and a complete ordering of $\{(\sigma_1, ..., \sigma_k) \mid \forall i\ \sigma_i \in (\tau_i)^\wedge\}$ denoted $\vec{\sigma}_1^\tau$, $\vec{\sigma}_2^\tau$, ... , $\vec{\sigma}_{\lceil\tau\rceil}^\tau$. We define inductively the type $\tau^+ = (\tau_1^+)^{\lceil\tau_1\rceil} \to ... \to (\tau_k^+)^{\lceil\tau_k\rceil} \to o$.

To a non terminal $F : \tau_1 \to ... \to \tau_k \to o$ (resp. a variable $x : \tau_1 \to ... \to \tau_k \to o$) and a tuple $\sigma_1 :: \tau_1, ..., \sigma_k :: \tau_k$, we associate the non-terminal $F^{\sigma_1,...,\sigma_k} : \tau_1^{\lceil \tau_1 \rceil} \to ... \to \tau_k^{\lceil \tau_k \rceil} \to o \in \mathscr{N}'$ (resp. a variable $x^{\sigma_1,...,\sigma_k} : \tau_1^{\lceil \tau_1 \rceil} \to ... \to \tau_k^{\lceil \tau_k \rceil} \to o \in \mathscr{V}'$).

Given a term $t : \tau = \tau_1 \to ... \to \tau_k \to o \in \mathscr{T}(\mathscr{V} \uplus \Sigma \uplus \mathscr{N})$ and an environment on the variables $\Theta^{\mathscr{V}}$ such that $Dom(\Theta^{\mathscr{V}}) \subseteq \mathscr{V}$ contains all variables in t, we define inductively the term $t^{+\sigma_1,...,\sigma_k}_{\Theta^{\mathscr{V}}} : \tau^+ \in \mathscr{T}(\mathscr{V}' \uplus \Sigma' \uplus \mathscr{N}')$ for all $\sigma_1 :: \tau_1, ..., \sigma_k :: \tau_k$. If $t = F \in \mathscr{N}$ (resp. $t = x \in \mathscr{V}$), $t^{+\sigma_1,...,\sigma_k}_{\Theta^{\mathscr{V}}} = F^{\sigma_1,...,\sigma_k}$ (resp. $t^{+\sigma_1,...,\sigma_k}_{\Theta^{\mathscr{V}}} = x^{\sigma_1,...,\sigma_k}$), if $t = a \in \Sigma$, $t^{+\sigma_1,...,\sigma_k}_{\Theta^{\mathscr{V}}} = a$. Finally consider the case where $t = t_1\ t_2$ with $t_1 : \tau' \to \tau$ and $t_2 : \tau'$. Let $\sigma = [\![t_2]\!]_{\Theta^{\mathscr{V}}}$. Remark that ${t_1}^{+\sigma,\sigma_1,...,\sigma_k}_{\Theta^{\mathscr{V}}} : (\tau'^+)^{\lceil \tau' \rceil} \to \tau^+$. We define $(t_1\ t_2)^{+\sigma_1,...,\sigma_k}_{\Theta^{\mathscr{V}}} = {t_1}^{+\sigma,\sigma_1,...,\sigma_k}_{\Theta^{\mathscr{V}}}\ {t_2}^{+\vec{\sigma}_1^{\tau'}}_{\Theta^{\mathscr{V}}} ...\ {t_2}^{+\vec{\sigma}^{\tau'}_{\lceil \tau' \rceil}}_{\Theta^{\mathscr{V}}}$. Note that since this transformation is only duplicating and anotating, given a term $t^{+\sigma_1,...,\sigma_k}$ we can uniquely find the unique term t associated to it.

Let $F : \tau_1 \to ... \to \tau_k \to o \in \mathscr{N}$, $\sigma_1 :: \tau_1, ..., \sigma_k :: \tau_k$, and $\Theta^{\mathscr{V}} = x_1 \triangleright \sigma_1, ..., x_k \triangleright \sigma_k$. If $F\ x_1 ... x_k \to e \in \mathscr{R}$, we define in $\mathscr{R}'$ the rule $F^{\sigma_1,...,\sigma_k}\ x_1^{+\vec{\sigma}_1^{\tau_1}} ...\ x_1^{+\vec{\sigma}^{\tau_1}_{\lceil \tau_1 \rceil}}\ ...\ x_k^{+\vec{\sigma}_1^{\tau_k}} ...\ x_k^{+\vec{\sigma}^{\tau_k}_{\lceil \tau_k \rceil}} \to e^+_{\Theta^{\mathscr{V}}}$. Finally, recall that $G' = \langle \mathscr{V}', \Sigma, \mathscr{N}', \mathscr{R}', S \rangle$.

The following theorem states that G' is just a labeling version of G and that it acts the same.

Theorem 5 (Equivalence between G and G')**.** *Given a term $t : o$, $\|G'_{t^+}\|_{IO} = \|G_t\|_{IO}$.*

We transform G' into the scheme G'' that will directly turn into $\perp$ a redex t such that $q_\perp \in [\![t]\!]$. For technical reason, instead of adding $\perp$ we add a non terminal $Void : o$ and a rule $Void \to Void$. $G' = \langle \mathscr{V}', \Sigma, \mathscr{N}' \uplus \{Void : o\}, \mathscr{R}'', S \rangle$ such that $\mathscr{R}''$ contains the rule $Void \to Void$ and for all $F \in \mathscr{N}$, if $q_\perp \in [\![F]\!]\ \sigma_1\ ...\ \sigma_k$ then $F^{\sigma_1,...,\sigma_k}\ x_1^{+\vec{\sigma}_1^{\tau_1}} ...\ x_1^{+\vec{\sigma}^{\tau_1}_{\lceil \tau_1 \rceil}} ...\ x_k^{+\vec{\sigma}_1^{\tau_k}} ...\ x_k^{+\vec{\sigma}^{\tau_k}_{\lceil \tau_k \rceil}} \to Void$ otherwise we keep the rule of $\mathscr{R}'$.

The following theorem concludes Section 4.

Theorem 6 (IO vs OI)**.** *Let G be a higher-order recursion scheme. Then one can construct a scheme G'' having the same order of G such that $\|G''\| = \|G\|_{IO}$.*

Proof (Sketch). First, given a term $t : o$, one can prove that $\|G''_{t^+}\|_{IO} = \|G'_{t^+}\|_{IO}$.

Then take a redex t such that $\|G''_t\|_{IO} = \perp$, i.e. $q_\perp \in [\![G_t]\!]$. There is only one OI derivation from t: $t \to Void \to Void \to ...$, then $\|G''_t\| = \perp$. We can extend this result saying that if there is the symbol $\perp$ at node u in $\|G''_t\|_{IO}$, then there is $\perp$ at node u in $\|G''_t\|$. Hence, since $\|G''_t\|_{IO} \sqsubseteq \|G''_t\|$, we have $\|G''\| = \|G''\|_{IO}$. Then $\|G''\| = \|G''\|_{IO} = \|G'\|_{IO} = \|G\|_{IO}$.

□

5 Conclusion

We have shown that value trees obtained from schemes using innermost-outermost derivations (IO) are the same as those obtained using unrestricted derivations. More precisely, given an order-n scheme G we create an order-$(n+1)$ scheme $\overline{G}$ such that $\|\overline{G}\|_{IO} = \|G\|$. However, the increase of the order seems unavoidable. We also create an order-n scheme G'' such that $\|\overline{G}''\| = \|G\|_{IO}$. In this case the order does not increase, however the size of the scheme deeply increases while it remains almost the same in $\overline{G}$.

References

[1] Klaus Aehlig (2006): *A Finite Semantics of Simply-Typed Lambda Terms for Infinite Runs of Automata*. In: *"Proc. of Computer Science Logic, 20th Annual Conference of the EACSL"*, *Lecture Notes in Comput. Sci.* 4207, Springer-Verlag, pp. 104–118, doi:10.1007/11874683_7.

[2] Bruno Courcelle (1978): *A Representation of Trees by Languages I*. *Theoret. Comput. Sci.* 6, pp. 255–279, doi:10.1016/0304-3975(78)90008-7.

[3] Bruno Courcelle (1978): *A Representation of Trees by Languages II*. *Theoret. Comput. Sci.* 7, pp. 25–55, doi:10.1016/0304-3975(78)90039-7.

[4] Bruno Courcelle & Maurice Nivat (1978): *The Algebraic Semantics of Recursive Program Schemes*. In: *Proc. 7th Symposium, Mathematical Foundations of Computer Science 1978*, *Lecture Notes in Comput. Sci.* 64, Springer-Verlag, pp. 16–30.

[5] Werner Damm (1977): *Higher type program schemes and their tree languages*. In: *Theoretical Computer Science, 3rd GI-Conference*, *Lecture Notes in Comput. Sci.* 48, Springer-Verlag, pp. 51–72.

[6] Werner Damm (1977): *Languages Defined by Higher Type Program Schemes*. In: *Proc. 4th Colloq. on Automata, Languages, and Programming (ICALP)*, *Lecture Notes in Comput. Sci.* 52, Springer-Verlag, pp. 164–179.

[7] Werner Damm (1982): *The IO- and OI-Hierarchies.* *Theoret. Comput. Sci.* 20, pp. 95–207, doi:10.1016/0304-3975(82)90009-3.

[8] Joost Engelfriet & Erik Meineche Schmidt (1977): *IO and OI. I.* *J. Comput. System Sci.* 15(3), pp. 328–353, doi:10.1016/S0022-0000(77)80034-2.

[9] Joost Engelfriet & Erik Meineche Schmidt (1978): *IO and OI. II.* *J. Comput. System Sci.* 16(1), pp. 67–99, doi:10.1016/0022-0000(78)90051-X.

[10] Matthew Hague, Andrzej S. Murawski, C.-H. Luke Ong & Olivier Serre (2008): *Collapsible Pushdown Automata and Recursion Schemes*. In: *Proceedings of the 23rd Annual IEEE Symposium on Logic in Computer Science (LICS)*, IEEE Computer Society, pp. 452–461.

[11] Klaus Indermark (1976): *Schemes with Recursion on Higher Types*. In: *Proc. 5th Symposium, Mathematical Foundations of Computer Science 1976*, *Lecture Notes in Comput. Sci.* 45, Springer-Verlag, pp. 352–358.

[12] Teodor Knapik, Damian Niwiński & Pawel Urzyczyn (2002): *Higher-Order Pushdown Trees Are Easy*. In: *Proceedings of the 5th International Conference on Foundations of Software Science and Computation Structures (FoSSaCS)*, *Lecture Notes in Comput. Sci.* 2303, Springer-Verlag, pp. 205–222, doi:10.1007/3-540-45931-6_15.

[13] Naoki Kobayashi (2009): *Types and higher-order recursion schemes for verification of higher-order programs*. In: *Proc. 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, ACM, pp. 416–428.

[14] Naoki Kobayashi & C.-H. Luke Ong (2009): *A Type System Equivalent to the Modal Mu-Calculus Model Checking of Higher-Order Recursion Schemes*. In: *Proceedings of the 24th Annual IEEE Symposium on Logic in Computer Science (LICS)*, IEEE Computer Society, pp. 179–188.

[15] M. Nivat (1972): *On the interpretation of recursive program schemes*. In: *Symposia Matematica.*

Initial Semantics for Strengthened Signatures

André Hirschowitz
Laboratoire J.-A. Dieudonné
Université de Nice - Sophia Antipolis
France
ah@unice.fr

Marco Maggesi
Dipartimento di Matematica "U. Dini"
Università degli Studi di Firenze
Italy
maggesi@math.unifi.it

We give a new general definition of arity, yielding the companion notions of signature and associated syntax. This setting is modular in the sense requested by [12]: merging two extensions of syntax corresponds to building an amalgamated sum. These signatures are too general in the sense that we are not able to prove the existence of an associated syntax in this general context. So we have to select arities and signatures for which there exists the desired initial monad. For this, we follow a track opened by Matthes and Uustalu [16]: we introduce a notion of strengthened arity and prove that the corresponding signatures have initial semantics (i.e. associated syntax). Our strengthened arities admit colimits, which allows the treatment of the λ-calculus with explicit substitution in the spirit of [12].

1 Introduction

Many programming or logical languages allow constructions which bind variables and this higher-order feature causes much trouble in the formulation, the understanding and the formalization of the theory of these languages. For instance, there is no universally accepted discipline for such formalizations: that is precisely why the POPLmark Challenge [4] offers benchmarks for testing old and new approaches. Although this problem may ultimately concern typed languages and their operational semantics, it already concerns untyped languages. In this work, we extend to new kinds of constructions our treatment of higher-order abstract syntax [13], based on modules and linearity.

First of all, we give a new general definition of arity, yielding the companion notion of signature. The notion is coined in such a way to induce a companion notion of representation of an arity (or of a signature) in a monad: such a representation is a morphism among modules over the given monad, so that an arity simply assigns two modules to each monad. There is a natural category of such representations of a signature and whenever it exists, the initial representation deserves the name of syntax associated with the given signature. This approach enjoys modularity in the sense introduced by [12]: in our category of representations, merging two extensions of a syntax corresponds to building an amalgamated sum.

Our notion of arity (or signature) is too general in the sense that we are not able to build, for each signature, a corresponding initial representation. Following a track opened in Matthes-Uustalu [16], we define a fairly general notion of *strengthened* arity, yielding the corresponding notion of strengthened signature. Our main result (Theorem 7.8) says that any strengthened signature yields the desired initial representation. As usual, this initial object is built as a minimal fixpoint.

Understanding the syntax of the lambda-calculus with explicit substitution was already done in [12], where the arity for this construction was identified as a coend, hence a colimit, of elementary arities (see Section 8). Our main motivation for the present work (and for our next one) was to propose a general approach to syntax (and ultimately to semantics) accounting for this example in the spirit of our previous work [14]. This is achieved thanks to our second main result (Theorem 4.3) which states the existence of colimits in the category of (strengthened) arities.

D. Miller and Z. Ésik (Eds.): Fixed Points
in Computer Science 2012 (FICS 2012)
EPTCS 77, 2012, pp. 31–38, doi:10.4204/EPTCS.77.5

In this extended abstract, we do not discuss proofs. A complete version is available on-line.[1]

2 Related and future work

The idea that the notion of monad is suited for modeling substitution concerning syntax (and semantics) has been retained by many recent contributions on the subject (see e.g. [5, 12, 16]) although some other settings have been considered. For instance in [15] the authors argue in favor of a setting based on Lawvere theories, while in [7] the authors work within a setting roughly based on operads (although they do not write this word down). The latter approach has been broadly extended, notably by M. Fiore [8, 9, 10]. Our main specificity here is the systematic use of the observation that the natural transformations we deal with are linear with respect to natural structures of module (a form of linearity had already been observed, in the operadic setting, see [11], Section 4).

The signatures we consider here are much more general than the signatures in [7], and cover the signatures appearing in [16, 12]. Note however that the latter works treat also non-wellfounded syntax, an aspect which we do not consider at all.

In our next work, we will propose a treatment of equational semantics for the present syntaxes. This approach should also be accommodated to deal with typed languages as done for elementary signatures in [17, 18, 2], or to model operational semantics as done for elementary signatures in [1].

3 The big category of modules

Modules over monads and the associated notion of linear natural transformation intend to capture the notion of "algebraic structure which is well-behaved with respect to substitution". An introduction on this subject can be found in our papers [13, 14]. Let us recall here the very basic idea.

Let R be a monad over a base category C. A module over R with range in a category D is a functor $M\colon \mathsf{C} \to \mathsf{D}$ endowed with an action of R, i.e., a natural "substitution" transformation $\rho\colon M \cdot R \longrightarrow M$ compatible with the substitution of R in the obvious sense. Given two modules M, N over the same monad and with the same range, a linear natural transformation $\phi\colon M \to N$ is a natural transformation of functors which is compatible with the actions in the obvious sense. This gives a category $\mathrm{Mod}^{\mathsf{D}}(R)$ of modules with fixed base R and range D.

It is useful for the present paper to consider a larger category which collects modules over different monads. For the following definition, we fix a range category D.

Definition 3.1 (The big module category)**.** We define the big module category $\mathrm{BMod}^{\mathsf{D}}_{\mathsf{C}}$ as follows:

- its objects are pairs (R, M) of a monad R on C and an R-module M with range in D.
- a morphism from (R, M) to (S, N) is a pair (f, m) where $f\colon R \longrightarrow S$ is a morphism of monads, and $m\colon M \longrightarrow f^*N$ is a morphism of R-modules (here f^*N is the functor N equipped with the obvious structure of R-module).

4 The category of arities

In this section, we give our new notion of arity. The destiny of an arity is to have representations in monads. A representation of an arity a in a monad R should be a morphism between two modules

[1] http://web.math.unifi.it/users/maggesi/strengthened/.

$\mathrm{dom}(a,R)$ and $\mathrm{codom}(a,R)$. For instance, in the case of the arity a of a binary operation, we have $\mathrm{dom}(a,R) := R^2$ and $\mathrm{codom}(a,R) := R$. Hence an arity should consist of two halves, each of which assigns to each monad R a module over R in a functorial way. However, in all our natural examples, we have $\mathrm{codom}(a,R) = R$ as above. Although this will no longer be the case in the typed case (which we do not consider here), we choose to restrict our attention to arities of this kind, where $\mathrm{codom}(a,R)$ is R.

From now on we will consider only monads over the category Set and modules with range Set. For technical reasons, see Section 7, we restrict our attention to the category of ω-cocontinuous endofunctors that we will denote $\mathrm{End}^{\omega}(\mathsf{Set})$. Analogously we will write Mon^{ω} (resp. BMod^{ω}) for the full subcategory of monads (resp. of modules over these monads) which are ω-cocontinuous.

We recall that finite limits commute with filtered colimits in Set. It follows that $\mathrm{End}^{\omega}(\mathsf{Set})$ has finite limits and arbitrary (small) colimits. This is the key ingredient in the proofs of ω-cocontinuity for most of our functors.

Definition 4.1 (Arities)**.** An *arity* is a right-inverse functor to the forgetful functor from the category BMod^{ω} to the category Mon^{ω}.

Now we give our basic examples of arities:

- Every monad R is itself a R-module. The assignment $R \mapsto R$ gives an arity which we denote by Θ.
- The assignment $R \mapsto *_R$, where $*_R$ denotes the final module over R is an arity which we denote by $*$.
- Given two arities a and b, the assignment $R \mapsto a(R) \times b(R)$ is an arity which we denote by $a \times b$. In particular $\Theta^2 = \Theta \times \Theta$ is the arity of any (first-order) binary operation and, in general Θ^n is the arity of n-ary operations.
- Given an endofunctor F of Set, we consider the *derived* functor given by $F' \colon X \mapsto F(X + *)$. It can be checked how when F is a module so is F'. Given an arity a, the assignment $R \mapsto a(R)'$ is an arity which we denote a' and is called *derivative* of a.
- Derivation can be iterated. We denote by $a^{(n)}$ the n-th derivative of a. Hence, in particular, we have $a^{(0)} = a$, $a^{(1)} = a'$, $a^{(2)} = a''$.
- For each sequence of non-negative integers $s = (s_1, \dots, s_n)$, the assignment $R \mapsto R^{(s_1)} \times \cdots \times R^{(s_n)}$ is an arity which we denote by $\Theta^{(s)}$. Arities of the form $\Theta^{(s)}$ are said *algebraic*. These algebraic arities are those which appear in [7].
- Given two arities a, b their composition $a \cdot b := R \mapsto a(R) \cdot b(R)$ is an arity.

Definition 4.2. A morphism among two arities $a_1, a_2 \colon \mathrm{Mon}^{\omega} \longrightarrow \mathrm{BMod}^{\omega}$ is a natural transformation $m \colon a_1 \longrightarrow a_2$ which, post-composed with the projection $\mathrm{BMod}^{\omega} \longrightarrow \mathrm{Mon}^{\omega}$, becomes the identity. We easily check that arities form a subcategory Ar of the category of functors from Mon^{ω} to BMod^{ω}.

Now we give two examples of morphisms of arities:

- The natural transformation $\mu \colon \Theta \cdot \Theta \longrightarrow \Theta$ induced by the structural composition of monads is a morphism of arities.
- The two natural transformations $\Theta \cdot \eta$ and $\eta \cdot \Theta$ from Θ to $\Theta \cdot \Theta$ are morphisms of arities.

Theorem 4.3. *The category of arities has finite limits and arbitrary (small) colimits.*

5 Categories of representations

Definition 5.1 (Signatures)**.** We define a signature $\Sigma = (O, \alpha)$ to be a family of arities $\alpha \colon O \longrightarrow \mathrm{Ar}$. A signature is said to be algebraic if it consists of algebraic arities.

Definition 5.2 (Representation of an arity, of a signature)**.** Given an ω-cocontinuous monad R over Set, we define a representation of the arity a in R to be a module morphism from $a(R)$ to R; a representation of a signature Σ in R consists of a representation in R for each arity in Σ.

Example 5.3. The usual $\mathsf{app}\colon \Lambda^2 \longrightarrow \Lambda$ is a representation of the arity Θ^2 into the monad Λ of λ-calculus 8.

Definition 5.4. Given a signature $\Sigma = (O, \alpha)$, we build the category Mon^Σ of representations of Σ as follows. Its objects are ω-cocontinuous monads equipped with a representation of Σ. A morphism m from (M, r) to (N, s) is a morphism of monads from M to N compatible with the representations in the sense that, for each o in O, the following diagram of M-modules commutes:

$$\begin{tikzcd} \alpha_o(M) \arrow[r, "r_o"] \arrow[d, "a_o(m)"'] & M \arrow[d, "m"] \\ m^*(\alpha_o(N)) \arrow[r, "m^* s_o"'] & m^*N \end{tikzcd}$$

where the horizontal arrows come from the representations and the left vertical arrow comes from the functoriality of arities and $m\colon M \longrightarrow m^*N$ is the morphism of monad seen as morphism of M-modules.

These morphisms, together with the obvious composition, turn Mon^Σ into a category which comes equipped with a forgetful functor to the category of monads.

We are primarily interested in the existence of an initial object in this category Mon^Σ.

Definition 5.5. A signature Σ is said representable if the category Mon^Σ has an initial object, which we denote $\hat{\Sigma}$.

Theorem 5.6. *Algebraic signatures are representable.*

For more details we refer to our paper [13] (Theorems 1 and 2). We give below a more general result (Theorem 7.8).

6 Modularity and the big category of representations

It has been stressed in [12] that the standard approach (via algebras) to higher-order syntax lacks modularity. In the present section we show in which sense our approach via modules enjoys modularity. The key for this modularity is what we call the big category of representations.

Suppose that we have a signature $\Sigma = (O, a)$ and two subsignatures Σ_1 and Σ_2 covering Σ in the obvious sense, and let Σ_0 be the intersection of Σ_1 and Σ_2. Suppose that these four signatures are representable (for instance because Σ is algebraic or strengthened in the sense defined below). Modularity would mean that the corresponding diagram of monads

$$\begin{tikzcd} \hat{\Sigma}_0 \arrow[r] \arrow[d] & \hat{\Sigma}_1 \arrow[d] \\ \hat{\Sigma}_2 \arrow[r] & \hat{\Sigma} \end{tikzcd}$$

is a pushout. The observation of [12] is that this diagram of raw monads is, in general, not a pushout. Since we do not want to change the monads, in order to claim for modularity, we will have to consider

a category of enhanced monads. Here by enriched monad, we mean a monad equipped with some additional structure, namely a representation of some signature.

Our solution to this problem goes through the following "big" category of representations, which we denote by RMon, where R may stand for representation or for rich:

- An object of RMon is a triple (R,Σ,r) where R is a monad, Σ a signature, and r is a representation of Σ in R.
- A morphism in RMon from $(R_1,(O_1,a_1),r_1)$ to $(R_2,(O_2,a_2),r_2)$ consists of an injective map $i := O_1 \longrightarrow O_2$ compatible with a_1 and a_2 and a morphism m from (R_1,r_1) to $(R_2,i^*(r_2))$, where $i^*(r_2)$ should be understood as the restriction of the representation r_2 to the subsignature (O_1,a_1) where we pose $i^*(r_2)(o) := r_2(i(o))$.
- It is easily checked that the obvious composition turns RMon into a category.

Now for each signature Σ, we have an obvious functor from Mon^Σ to RMon, through which we may see $\hat{\Sigma}$ as an object in RMon. Furthermore, an injection $i\colon \Sigma_1 \longrightarrow \Sigma_2$ obviously yields a morphism $i_* := \hat{\Sigma}_1 \longrightarrow \hat{\Sigma}_2$ in RMon. Hence our 'pushout' square of signatures as described above yields a square in RMon. The proof of the following statement is straightforward.

Modularity holds in RMon, in the sense that given a 'pushout' square of representable signatures as described above, the associated square in RMon is a pushout again.

As usual, we will denote by RMon^ω the full subcategory of RMon constituted by ω-cocontinuous functors. It is easy to check that the previous statement is equally valid in RMon^ω. Indeed, recall that, by our definition, the initial representation of representable signatures lies in RMon^ω.

7 Strengthening signatures

Guided by the ideas of Matthes and Uustalu [16] we introduce in our framework the notion of *strengthened arity*. For a category C, let us denote by $\mathrm{End}^\omega_*(\mathsf{C})$ the category of ω-cocontinuous *pointed endofunctors*, i.e., the category of pairs (F,η) of an ω-cocontinuous endofunctor F of C and a natural transformation $\eta\colon I \longrightarrow F$ from the identity endofunctor to F. A morphism of pointed endofunctors $f\colon (F_1,\eta_1) \longrightarrow (F_2,\eta_2)$ is a natural transformation $f\colon F_1 \longrightarrow F_2$ satisfying $f \circ \eta_1 = \eta_2$.

Definition 7.1. A *strengthened arity* is a pair (H,θ) where H is an ω-cocontinuous endofunctor of $\mathrm{End}^\omega(\mathsf{Set})$ (i.e., $H \in \mathrm{End}^\omega(\mathrm{End}^\omega(\mathsf{Set}))$) and θ is a natural transformation $\theta\colon H(-)\cdot\sim \longrightarrow H(-\cdot\sim)$ (where $H(-)\cdot\sim$ and $H(-\cdot\sim)$ have to be understood as functors from $\mathrm{End}^\omega(\mathsf{Set})\times\mathrm{End}^\omega_*(\mathsf{Set})$ to $\mathrm{End}^\omega(\mathsf{Set})$) satisfying $\theta_{X,(I,1_I)} = 1_{HX}$ and such that the following diagram is commutative

$$\begin{tikzcd}
H(X)\cdot Z_1\cdot Z_2 \arrow[rr, "\theta_{X,(Z_1\cdot Z_2,e_1\cdot e_2)}"] \arrow[dr, "\theta_{X,(Z_1,e_1)}Z_2"'] & & H(X\cdot Z_1\cdot Z_2) \\
& H(X\cdot Z_1)\cdot Z_2 \arrow[ur, "\theta_{X\cdot Z_1,(Z_2,e_2)}"'] &
\end{tikzcd} \tag{1}$$

for every endofunctor X and pointed endofunctors (Z_1,e_1), (Z_2,e_2). We refer to θ as the *strength* on H.

Our first task is to make clear that our wording is consistent in the sense that a strengthened arity H somehow yields a genuine arity $\tilde{H}$. For this task, for each monad R we pose $\tilde{H}(R) := H(R)$ and we exhibit on it a structure of R-module. We do even slightly more by upgrading H into a *module transformer* in the following sense:

Definition 7.2. A module transformer is an endofunctor of the big module category BMod^{ω} which commutes with the structural forgetful functor $\mathrm{BMod}^{\omega} \longrightarrow \mathrm{Mon}^{\omega}$.

Let (H,θ) be a strengthened arity. For every ω-cocontinuous monad R and ω-cocontinuous R-module M, we define the natural transformation $\rho^{H(M)}\colon H(M)\cdot R \longrightarrow H(M)$ as the composition $H(\rho^{M})\cdot\theta_{M,R}$. Then $(H(M),\rho^{H(M)})$ is an R-module, and this construction upgrades H into a module transformer denoted by $\hat{H}$.

We call the restricion $\tilde{H}$ of the module transformer $\hat{H}$ to the category of monads the arity associated to the strengthened arity H.

Our next task is to upgrade our favorite examples of arities into strengthened arities:

- The arity Θ comes from the strengthened arity (H,θ) where H and θ are the relevant identities.
- The arity $*$ comes from the strengthened arity (H,θ) where H is the final endofunctor and θ is the relevant identity. This is the final strengthened arity.
- The arity $\Theta\cdot\Theta$ comes from the strengthened arity (H,θ) where $H := X \mapsto X\cdot X$ and $\theta_{X,Y} : X\cdot X\cdot Y \longrightarrow X\cdot Y\cdot X\cdot Y := X\cdot\eta^{Y}\cdot X\cdot Y$; here we have written η^{Y} for the morphism from the identity functor to Y (remember that Y is pointed).
- If an arity comes from a strengthened arity, so does its derivative (see Proposition 7.4).

Then we show how our basic constructions in the category of arities carries over the category of strengthened arities. First we describe this category. Its objects are strengthened arities and we take for morphisms from (H_1,θ_1) to (H_2,θ_2) those natural transformations $m\colon H_1 \longrightarrow H_2$ which are compatible with θ_1 and θ_2, that is, the diagram

$$\begin{tikzcd} H_1(X)\cdot Z \arrow[r, "\theta_1"] \arrow[d, "m_X Z"'] & H_1(X\cdot Z) \arrow[d, "m_{X\cdot Z}"] \\ H_2(X)\cdot Z \arrow[r, "\theta_2"'] & H_2(X\cdot Z) \end{tikzcd}$$

is commutative for every endofunctor X and every pointed endofunctor Z.

Theorem 7.3. *The category of strengthened arities has finite limits and arbitrary colimits.*

Next, we take care of the derivation. We denote by D the endofunctor of Set given by $A \mapsto A+*$. For any other pointed endofunctor X over Set we have a natural transformation $w^{X}\colon D\cdot X \longrightarrow X\cdot D$ given by

$$w^{X}_{A}\colon X(A)+* \longrightarrow X(A+*) \qquad w^{X}_{A} := X(i_A)+\eta_{A+*}\cdot\underline{*}$$

where $i_A\colon A \longrightarrow A+*$ and $\underline{*}\colon * \longrightarrow A+*$ are the inclusion maps.

Proposition 7.4. *If (H,θ) is a strengthened arity, then the pair (H',θ'), where $H' := X \mapsto H(X)'$ and $\theta'_{X,Z} := \theta_{X,Z}D\cdot H(X)w^{Z}$, is a strengthened arity. We call it the* derivative *of (H,θ).*

Now we point out the possibility of composing strengthened arities.

Definition 7.5. If $H := (H,\rho)$ and $K := (K,\sigma)$ are two strengthened arities, their composition $H\cdot K$ is the pair $(H\cdot K,\theta)$ where θ is defined by $\theta_{X,(Z,e)} := H(\sigma_{X,(Z,e)})\cdot\rho_{K(X),(Z,e)}$.

Proposition 7.6. *This composition turns strengthened arities into a strict monoidal category.*

Next, we turn to the main interest of strengthened arities (or signatures) which is that the fixed point we are interested in inherits a structure of monad.

Lemma 7.7. *Let (H,θ) be a strengthened arity. Then the fixed point T of the functor $F := I+H$ is ω-cocontinuous and comes equipped with a structure of $\tilde{H}$-representation which is the initial object in the category of the $\tilde{H}$-representations.*

We say that a signature is strengthened if it is a family of strengthened arities. The previous lemma leads immediately to the following result.

Theorem 7.8. *Strengthened signatures are representable.*

8 Examples of strengthened syntax

Lambda-calculus modulo α-equivalence One paradigmatic example of syntax with binding is the λ-calculus. We denote by $\Lambda(X)$ the set of lambda-terms up to α-equivalence with free variables 'indexed' by the set X. It is well-known [6, 3, 13] that Λ has a natural structure of cocontinuous monad where the monad composition is given by variable substitution.

It can be easily verified that application $\mathsf{app}\colon \Lambda^2 \longrightarrow \Lambda$ and abstraction $\mathsf{abs}\colon \Lambda' \longrightarrow \Lambda$ are Λ-linear natural transformations, that is, Λ is a monad endowed with a representation ρ of the signature $\Sigma = \{\mathsf{app}\colon \Theta^2, \mathsf{abs}\colon \Theta'\}$. The monad Λ is initial in the category Mon^Σ of ω-cocontinuous monads equipped with a representation of the signature Σ.

This is an example of algebraic signature and thus already treated by other previous works [13, 14, 7]. Here we simply remark that our new theory covers such a classical case.

Explicit composition operator We now consider our first example of non-algebraic signature. On any monad R, we have the composition operator (also called join operator) $\mu^R\colon R \cdot R \longrightarrow R$ which has arity $\Theta \cdot \Theta$. We will refer to the μ^R operator as the *implicit* composition operator. An interesting problem is to see if this kind of operators admits a corresponding *explicit* version, i.e., if they can be implemented as a syntactic construction. As we have seen before $\Theta \cdot \Theta$ is a strengthened arity hence we can build syntaxes with explicit composition operator of kind

$$\mathsf{join}\colon \Theta \cdot \Theta \longrightarrow \Theta.$$

Of course, this is only a *syntactic* composition operator, in the sense that it does not enjoy several desirable conversion rules like associativity, two-side identity and the obvious compatibility rules with the other syntactic constructions present in the signature. In our next work we will show how to construct such kind of *semantic* composition operator.

Let us mention that given a monad R, the unit $\eta_R\colon I \longrightarrow R$ is not an R-linear morphism (in fact, I is not even an R-module in general). For this reason we cannot treat examples of syntax with explicit unit.

Syntax and semantics with explicit substitution On any monad R, we have a series of substitution operators $\sigma_n\colon R^{(n)} \cdot R^n \longrightarrow R$ which simultaneously replace n formal arguments in a term with n given terms. As observed by Ghani and Uustalu [12], these substitution morphisms satisfy a series of compatibility relations which mean that they come from a single morphism $\mathsf{subst}\colon C \longrightarrow \Theta$ where C is identified as the coend

$$C = \int^{A:\mathsf{Fin}} \Theta^{(A)} \times \Theta^A.$$

Here Fin stands for a skeleton of the category of finite sets, Θ^A denotes the cartesian power and $\Theta^{(A)}$ is defined by $\Theta^{(A)}(R,X) := R(X+A)$. Since coends are special colimits, and strengthened arities admit colimits, we just have to check that the bifunctorial arity $(A,B) \mapsto \Theta^{(A)} \times \Theta^B$ factors through the category of strengthened arities. As far as objects are concerned, this follows from our results in Section 7. The verification of the compatibility of the corresponding "renaming" and "projection" morphisms with the strengthened structures is straightforward.

References

[1] Benedikt Ahrens (2011): *Modules over relative monads for syntax and semantics*. *ArXiv e-prints* .

[2] Benedikt Ahrens & Julianna Zsidó (2011): *Initial Semantics for higher-order typed syntax in Coq*. *Journal of Formalized Reasoning* 4(1), pp. 25–69.

[3] Thorsten Altenkirch & Bernhard Reus (1999): *Monadic Presentations of Lambda Terms Using Generalized Inductive Types*. In: *CSL*, pp. 453–468.

[4] B. Aydemir, A. Bohannon, M. Fairbairn, J. Foster, B. Pierce, P. Sewell, D. Vytiniotis, G. Washburn, S. Weirich & S. Zdancewic (2005): *Mechanized metatheory for the masses: The POPLmark Challenge*. In: *Proceedings of the Eighteenth International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2005)*.

[5] Richard Bird & Ross Paterson (1999): *Generalised Folds for Nested Datatypes*. *Formal Aspects of Computing* 11(2), pp. 200–222.

[6] Richard S. Bird & Ross Paterson (1999): *De Bruijn Notation as a Nested Datatype*. *Journal of Functional Programming* 9(1), pp. 77–91.

[7] Marcelo Fiore, Gordon Plotkin & Daniele Turi (1999): *Abstract Syntax and Variable Binding*. In: *LICS '99: Proceedings of the 14th Annual IEEE Symposium on Logic in Computer Science*, IEEE Computer Society, Washington, DC, USA, p. 193.

[8] Marcelo P. Fiore (2008): *Second-Order and Dependently-Sorted Abstract Syntax*. In: *LICS*, IEEE Computer Society, pp. 57–68, doi:10.1109/LICS.2008.38.

[9] Marcelo P. Fiore & Chung-Kil Hur (2010): *Second-Order Equational Logic (Extended Abstract)*. In Anuj Dawar & Helmut Veith, editors: *CSL*, *Lecture Notes in Computer Science* 6247, Springer, pp. 320–335, doi:10.1007/978-3-642-15205-4_26.

[10] Marcelo P. Fiore & Ola Mahmoud (2010): *Second-Order Algebraic Theories - (Extended Abstract)*. In Petr Hlinený & Antonín Kucera, editors: *MFCS*, *Lecture Notes in Computer Science* 6281, Springer, pp. 368–380, doi:10.1007/978-3-642-15155-2_33.

[11] Marcelo P. Fiore & Daniele Turi (2001): *Semantics of Name and Value Passing*. In: *Logic in Computer Science*, pp. 93–104.

[12] Neil Ghani, Tarmo Uustalu & Makoto Hamana (2006): *Explicit substitutions and higher-order syntax*. *Higher-order and Symbolic Computation* 19(2–3), pp. 263–282.

[13] André Hirschowitz & Marco Maggesi (2007): *Modules over Monads and Linearity*. In Daniel Leivant & Ruy J. G. B. de Queiroz, editors: *WoLLIC*, *Lecture Notes in Computer Science* 4576, Springer, pp. 218–237, doi:10.1007/978-3-540-73445-1_16.

[14] André Hirschowitz & Marco Maggesi (2010): *Modules over monads and initial semantics*. *Information and Computation* 208(5), pp. 545–564, doi:10.1016/j.ic.2009.07.003. Special Issue: 14th Workshop on Logic, Language, Information and Computation (WoLLIC 2007).

[15] Martin Hyland & John Power (2007): *The category theoretic understanding of universal algebra: Lawvere theories and monads*. *Electronic Notes in Theoretical Computer Science* 172, pp. 437–458, doi:10.1016/j.entcs.2007.02.019.

[16] Ralph Matthes & Tarmo Uustalu (2004): *Substitution in non-wellfounded syntax with variable binding*. *Theor. Comput. Sci.* 327(1-2), pp. 155–174, doi:10.1016/j.tcs.2004.07.025.

[17] Julianna Zsidó (2005/06): *Le lambda calcul vu comme monade initiale*. Master's thesis, Université de Nice – Laboratoire J. A. Dieudonné. Mémoire de Recherche – master 2.

[18] Julianna Zsidó (2010): *Typed Abstract Syntax*. Ph.D. thesis, University of Nice, France. http://tel.archives-ouvertes.fr/tel-00535944/.

Model-Checking the Higher-Dimensional Modal μ-calculus

Martin Lange Etienne Lozes
School of Electr. Eng. and Computer Science, University of Kassel, Germany

The higher-dimensional modal μ-calculus is an extension of the μ-calculus in which formulas are interpreted in tuples of states of a labeled transition system. Every property that can be expressed in this logic can be checked in polynomial time, and conversely every polynomial-time decidable problem that has a bisimulation-invariant encoding into labeled transition systems can also be defined in the higher-dimensional modal μ-calculus. We exemplify the latter connection by giving several examples of decision problems which reduce to model checking of the higher-dimensional modal μ-calculus for some fixed formulas. This way generic model checking algorithms for the logic can then be used via partial evaluation in order to obtain algorithms for theses problems which may benefit from improvements that are well-established in the field of program verification, namely on-the-fly and symbolic techniques. The aim of this work is to extend such techniques to other fields as well, here exemplarily done for process equivalences, automata theory, parsing, string problems, and games.

1 Introduction

The Modal μ-Calculus $\mathscr{L}_\mu$ [6] is mostly known as a backbone for temporal logics used in program specification and verification. The most important decision problem in this domain is the model checking problem which is used to automatically prove correctness of programs. The model checking problem for $\mathscr{L}_\mu$ is well-understood by now. There are several algorithms and implementations for it. It is known that model checking $\mathscr{L}_\mu$ is equivalent under linear-time translations to the problem of solving a parity game [8] for which there also is a multitude of algorithms available. From a purely theoretical point of view, there is still the intriguing question of the exact computational complexity of model checking $\mathscr{L}_\mu$: the best known upper bound for finite models is UP$\cap$coUP [5], which is not entirely matched by the P-hardness inherited from model checking modal logic.

$\mathscr{L}_\mu$ can express exactly the bisimulation-invariant properties of tree or graph models which are definable in Monadic Second-Order Logic [4], i.e. are regular. This means that for every such set L of trees or graphs there is a fixed $\mathscr{L}_\mu$ formula φ_L s.t. a tree or graph G is a model of φ_L iff it belongs to L. Thus, any decision problem that has an encoding into regular and bisimulation-invariant sets of trees or graphs can in principle be solved using model checking technology. In detail, suppose there is a set M and a function f from the domain of M to graphs s.t. $\{f(x) \mid x \in M\}$ is regular and closed under bisimilarity. By the result above there is an $\mathscr{L}_\mu$ formula φ_M which defines (the encoding of) M. Now any model checking algorithm for $\mathscr{L}_\mu$ can be used in order to solve M.

Note that in theory this is just a reduction from M to the model checking problem for $\mathscr{L}_\mu$ on a fixed formula. Obviously reductions from any problem A to some problem B can be used to transfer algorithms from B to A, and the algorithm obtained for A can in general be at most as good as the algorithm for B unless it can be optimised for the fragment of B resulting from embedding A into it. However, there are two aspects that are worth noting in this context.

- A reduction to model checking for a fixed formula can lead to much more efficient algorithms. A model checking algorithm takes two inputs in general: a structure and a formula. If the formula is

D. Miller and Z. Ésik (Eds.): Fixed Points
in Computer Science 2012 (FICS 2012)
EPTCS 77, 2012, pp. 39–46, doi:10.4204/EPTCS.77.6

fixed then partial evaluation can be used in order to optimise the general scheme, throw away data structures, etc.

- Program verification is a very active research area which has developed many clever techniques for evaluating formulas in certain structures including on-the-fly [8] and symbolic methods [2], partial-order reductions, etc.

We refer to [1] for an example of this scheme of reductions to model checking for fixed formulas, there being done for problems that are at least PSPACE-hard. It also shows how this can be used to solve computation problems in this way. Since the data complexity (model checking with fixed formula) of $\mathscr{L}_\mu$ is in P, using this scheme for $\mathscr{L}_\mu$ is restricted to computationally simpler problems which can nevertheless benefit from developments in program verification. Furthermore, it is the presence of fixpoint operators in such a logic which makes it viable to this approach: fixpoint operators can be used to express inductive concepts—e.g. the derivation relation in a context-free grammar—and at the same time provide the foundation for algorithmic solutions via fixpoint iteration for instance.

Here we consider an extension of $\mathscr{L}_\mu$, the Higher-Dimensional Modal μ-Calculus $\mathscr{L}_\mu^\omega$, and investigate its usefulness regarding the possibility to obtain algorithmic solutions to various decision or computation problems which may benefit from techniques originally developed for program verification purposes only. It is known that $\mathscr{L}_\mu^\omega$ captures the bisimulation-invariant fragment of P. We will sketch how the $\mathscr{L}_\mu^\omega$ model checking problem can be reduced to $\mathscr{L}_\mu$ model checking via a simple product construction on transition systems. Thus we can obtain—in principle—an algorithm for every problem that admits a polynomial-time solution and a bisimulation-invariant encoding into graphs. The reduction from $\mathscr{L}_\mu^\omega$ to $\mathscr{L}_\mu$ is compatible with on-the-fly or BDD-based model checking techniques, thus transferring such algorithms from $\mathscr{L}_\mu$ first to $\mathscr{L}_\mu^\omega$ and then on to such decision problems.

2 The Higher-Dimensional Modal μ-Calculus

Labeled Transition Systems. A labeled transition system (LTS) is a graph whose vertices and edges are labeled with sets of propositional variables and labels respectively. Formally, an LTS over a set $\Sigma = \{a,b,\dots\}$ of edge labels and a set $\mathsf{P} = \{p,q,\dots\}$ of atomic propositions is a tuple $\mathfrak{M} = (S, s_0, \Delta, \rho)$ such that $s_0 \in S$, $\Delta \subseteq S \times \Sigma \times S$ and $\rho : S \to \mathscr{P}(\mathsf{P})$. Elements of S are called states, and we write $s \xrightarrow{a} s'$ when $(s,a,s') \in \Delta$. The state $s_0 \in S$ is called the initial state of $\mathfrak{M}$.

We will mainly consider *finite* transition systems, *i.e.* transition systems (S, s_0, Δ, ρ) such that S is a finite set. Infinite-state transition systems arising from program verification are also of interest, but their model checking techniques differ from the ones of finite LTS and cannot be handled by our approach (see more comments on that point in the conclusion).

Syntax. We assume infinite sets $\mathsf{Var} = \{x,y,\dots\}$ and $\mathsf{Var}_2 = \{X,Y,\dots\}$, of first-order and second-order variables respectively. For tuples of first-order variables $\overline{x} = (x_1,\dots,x_n)$ and $\overline{y} = (y_1,\dots,y_n)$, with all x_i distinct, $\overline{x} \leftarrow \overline{y}$, denotes the function $\kappa : \mathsf{Var} \to \mathsf{Var}$ such that $\kappa(x_i) = y_i$, and $\kappa(z) = z$ otherwise. It is called a *variable replacement*.

The syntax of the higher-dimensional modal μ-calculus $\mathscr{L}_\mu^\omega$ is reminiscent of that of the ordinary modal μ-calculus. However, modalities and propositions are relativized to a first-order variable, and it also features the *replacement* modality $\{\kappa\}$. Formulas of $\mathscr{L}_\mu^\omega$ are defined by the grammar

$$\varphi, \psi \;:=\; p(x) \mid X \mid \neg\varphi \mid \varphi \wedge \psi \mid \langle a \rangle_x \varphi \mid \mu X.\varphi \mid \{\overline{x} \leftarrow \overline{y}\}\varphi$$

where $x,y \in \mathsf{Var}$, $\kappa : \mathsf{Var} \to \mathsf{Var}$ is a variable replacement with finite domain, $a \in \Sigma$, and $X \in \mathsf{Var}_2$. We require that every second-order variable gets bound by a fixpoint quantifier μ at most once in a formula. Then for every formula φ there is a function fp_φ which maps each second-order variable X occurring in φ to its unique binding formula $fp_\varphi(X) = \mu X.\psi$. Finally, we allow occurrences of a second-order variable X only under the scope of an even number of negation symbols underneath $fp_\varphi(X)$.

A formula is of dimension n if it contains at most n distinct first-order variables; we write $\mathscr{L}_\mu^n$ to denote the set of formulas of dimension n. Note that $\mathscr{L}_\mu^1$ is equivalent to the standard modal μ-calulus: with a single first-order variable x, we have $p(x) \equiv p$, $\{x \leftarrow x\}\psi \equiv \psi$ and $\langle a \rangle_x \psi \equiv \langle a \rangle \psi$ for any ψ.

As usual, we write $\varphi \vee \psi$, $[a]_x \varphi$, and $\nu X.\varphi$ to denote $\neg(\neg\varphi \wedge \neg\psi)$, $\neg\langle a \rangle_x \neg\varphi$, $\neg\mu X.\neg\varphi'$ respectively where φ' is obtained from φ by replacing every occurrence of X with $\neg X$. Other Boolean operators like $\Rightarrow$ and $\Leftrightarrow$ are defined as usual.

Note that $\{\kappa\}$ is an operator in the syntax of the logic; it does not describe syntactic replacement of variables. Consider for instance the formula

$$\nu X. \bigwedge_{p \in \mathsf{P}} p(x) \Rightarrow p(y) \quad \wedge \quad \bigwedge_{a \in \Sigma} [a]_x \langle a \rangle_y X \quad \wedge \quad \{(x,y) \leftarrow (y,x)\} X.$$

As we will later see, this formula characterizes bisimilar states x and y. In this formula, the operational meaning of $\{x,y \leftarrow y,x\}X$ can be thought as "swapping the players' pebbles" in the bisimulation game.

We will sometimes require formulas to be in *positive normal form*. Such formulas are built from literals $p(x)$, $\neg p(x)$ and second-order variables X using the operators $\wedge$, $\vee$, $\langle a \rangle_x$, $[a]_x$, μ, ν, and $\{\kappa\}$. A formula is *closed* if all second-order variables are bound by some μ.

With $Sub(\varphi)$ we denote that set of all *subformulas* of φ. It also serves as a good measure for the *size* of a formula: $|\varphi| := |Sub(\varphi)|$. Another good measure of the complexity of the formula φ is its *alternation depth* ad_φ, *i.e* the maximal alternation of μ and ν quantifiers along any path in the syntactic tree of its positive normal form.

Semantics. A first-order valuation v over a LTS $\mathfrak{M}$ is a mapping from first-order variables to states, and a second order valuation is a mapping from second order variables to sets of first-order valuations:

$$\begin{array}{rcll} \mathsf{Val} & \triangleq & \mathsf{Var} & \to \quad S \\ \mathsf{Val}_2 & \triangleq & \mathsf{Var}_2 & \to \quad \mathscr{P}(\mathsf{Val}) \end{array}$$

We write $v[\bar{x} \mapsto \bar{s}]$ to denote the first-order valuation that coincides with v, except that $x_i \in \bar{x}$ is mapped to the corresponding $s_i \in \bar{s}$. We use the same notation $\mathscr{V}[\bar{X} \mapsto \bar{P}]$ for second-order valuations. The semantics of a formula φ of $\mathscr{L}_\mu^\omega$ for a LTS $\mathfrak{M}$ and a second-order valuation $\mathscr{V}$ is defined as a set of first-order valuations by induction on the formula:

$$\begin{array}{lcl} [\![p(x)]\!]_{\mathfrak{M}}^{\mathscr{V}} & \triangleq & \{v : p \in \rho(v(x))\} \\ [\![\neg\varphi]\!]_{\mathfrak{M}}^{\mathscr{V}} & \triangleq & \mathsf{Val} - [\![\varphi]\!]_{\mathfrak{M}}^{\mathscr{V}} \\ [\![\varphi \wedge \psi]\!]_{\mathfrak{M}}^{\mathscr{V}} & \triangleq & [\![\varphi]\!]_{\mathfrak{M}}^{\mathscr{V}} \cap [\![\psi]\!]_{\mathfrak{M}}^{\mathscr{V}} \\ [\![\langle a \rangle_x \varphi]\!]_{\mathfrak{M}}^{\mathscr{V}} & \triangleq & \{v : \exists s.\ v(x) \xrightarrow{a} s \text{ and } v[x \mapsto s] \in [\![\varphi]\!]_{\mathfrak{M}}^{\mathscr{V}}\} \\ [\![X]\!]_{\mathfrak{M}}^{\mathscr{V}} & \triangleq & \mathscr{V}(X) \\ [\![\mu X.\varphi]\!]_{\mathfrak{M}}^{\mathscr{V}} & \triangleq & LFP\ \ \lambda P \in \mathscr{P}(\mathsf{Val}).\ [\![\varphi]\!]_{\mathfrak{M}}^{\mathscr{V}[X \mapsto P]} \\ [\![\{\bar{x} \leftarrow \bar{y}\}\varphi]\!]_{\mathfrak{M}}^{\mathscr{V}} & \triangleq & \{v : v[\bar{x} \mapsto v(\bar{y})] \in [\![\varphi]\!]_{\mathfrak{M}}^{\mathscr{V}}\} \end{array}$$

We simply write $[\![\varphi]\!]_{\mathfrak{M}}$ to denote the semantics of a closed formula. We write $\mathfrak{M}, v \vDash \varphi$ if $v \in [\![\varphi]\!]_{\mathfrak{M}}$, and $\mathfrak{M} \vDash \varphi$ if $\mathfrak{M}, v_0 \vDash \varphi$, where v_0 is the constant function to s_0. Two formulas are equivalent, written

$\varphi \equiv \psi$, if $[\![\varphi]\!]_{\mathfrak{M}} = [\![\psi]\!]_{\mathfrak{M}}$ for any LTS $\mathfrak{M}$. As with the normal modal μ-calculus, it is a simple exercise to prove that every formula is equivalent to one in positive normal form.

Proposition 1. *For every $\varphi \in \mathscr{L}_\mu^\omega$ there is a ψ in positive normal form such that $\varphi \equiv \psi$ and $|\psi| \leq 2 \cdot |\varphi|$.*

Reduction to the Ordinary μ-Calculus. Here we consider $\mathscr{L}_\mu^\omega$ as a formal language for defining decision problems. Algorithms for these problems can be obtained from model checking algorithms for $\mathscr{L}_\mu$ on fixed formulas using partial evaluation. In order to lift all sorts of special techniques which have been developed for model checking in the area of program verification we show how to reduce the $\mathscr{L}_\mu^\omega$ model checking problem to that of $\mathscr{L}_\mu^1$, i.e. the ordinary μ-calculus.

Let us assume a fixed non-empty finite subset V of first-order variables. A formula φ of $\mathscr{L}_\mu^\omega$ with $fv(\varphi) \subseteq V$ can be seen as a formula $\hat{\varphi}$ of $\mathscr{L}_\mu^1$ over the set of the atomic propositions $\mathsf{P} \times V$ and the action labels $\Sigma \times V \cup (V \to V)$. We write p_x instead of (p,x) for elements of $\mathsf{P} \times V$, and equally a_x for elements from $\Sigma \times V$. Then $\varphi \mapsto \widehat{\varphi}$ can be defined as the homomorphism such that $\widehat{p(x)} \triangleq p_x$, $\widehat{\langle a \rangle_x \varphi} \triangleq \langle a_x \rangle \widehat{\varphi}$, and $\widehat{\{\overline{x \leftarrow y}\}\varphi} \triangleq \langle \overline{x \leftarrow y} \rangle \widehat{\varphi}$.

We call an LTS *higher-dimensional* when it interprets the extended propositions p_x and modalities $\langle a_x \rangle$ and $\langle \kappa \rangle$ introduced by the formulas $\widehat{\varphi}$, and *ground* when it interprets the standard propositions and modalities. For a ground LTS $\mathfrak{M}$ and a formula φ, we thus need to define the higher-dimensional LTS over which $\widehat{\varphi}$ should be interpreted: we call it the *V-clone* of $\mathfrak{M}$, and write it $\mathsf{clone}_V(\mathfrak{M})$. Roughly speaking, $\mathsf{clone}_V(\mathfrak{M})$ is the asynchronous product of $|V|$ copies of $\mathfrak{M}$. More formally, assume $\mathfrak{M} = (S, s_0, \Delta, \rho)$; then $\mathsf{clone}_V(\mathfrak{M}) = (S', s_0', \Delta', \rho')$ is defined as follows.

- The states are valuations of the variables in V by states in S, *e.g* $S' = V \to S$, and s_0' is the constant function $\lambda x \in V.s_0$.
- The atomic proposition p_x is true in those new states, which assign x to an original state that satisfies p, e.g. $\rho'(v) = \{p_x \ : \ p \in \rho(v(x))\}$.
- The transitions contain labels of two kinds. First, there is an a_x-edge between two valuations v and v', if there is an a-edge between $v(x)$ and $v'(x)$ in the original LTS $\mathfrak{M}$:

  $$v \xrightarrow{a_x} v' \quad \text{iff} \quad \exists t.v(x) \xrightarrow{a} t \text{ and } v' = v[x \mapsto t].$$

  For the other kind of transitions we need to declare the effect of applying a replacement to a valuation. Let $v : V \to S$ be a valuation of the first-order variables in V, and $\kappa : V \to V$ be a replacement operator. Let ${}^t\kappa(v)$ be the valuation such that ${}^t\kappa(v)(x) = v(\kappa(x))$. Then we add the following transitions to Δ'.

  $$v \xrightarrow{\kappa} v' \quad \text{iff} \quad v' = {}^t\kappa(v)$$

Note that the relation with label κ is functional for any such κ, i.e. every state in $\mathsf{clone}_V(\mathfrak{M})$ has exactly one κ-successor. Hence, we have $\langle \kappa \rangle \psi \equiv [\kappa] \psi$ over cloned LTS.

Theorem 2. *Let V be a finite set of first-order variables, let $\mathfrak{M} = (S, s_0, \Delta, \rho)$ be a ground LTS, and let φ be a $\mathscr{L}_\mu^\omega$ formula such that $fv(\varphi) \subseteq V$. Then*

$$\mathfrak{M} \models \varphi \quad \textit{iff} \quad \mathsf{clone}_V(\mathfrak{M}) \models \widehat{\varphi}.$$

The proof goes by straightforward induction on φ and is therefore ommitted – see also the chapter on descriptive complexity in [3] for similar results. The importance of Thm. 2 is based on the fact that it transfers many model checking algorithms for the modal μ-calculus to $\mathscr{L}_\mu^1$, for example on-the-fly model checking [8], symbolic model checking [2] with BDDs or via SAT, strategy improvement schemes [9], etc.

3 Various Problems as Model Checking Problems

The model checking algorithms we mentioned can be exploited to solve any polynomial-time problem that can be encoded as a model checking problem in $\mathscr{L}_\mu^\omega$. By means of examples, we now intend to show that these problems are quite numerous.

Process Equivalences. The first examples are process equivalences encountered in process algebras. We only consider here strong simulation equivalence and bisimilarity, and let the interested reader think about how to encode other process equivalences, like weak bisimilarity for instance.

Let us first recall some standard definitions. Let $\mathfrak{M} = (S, s_0, \Delta, \rho)$ be a fixed LTS. A *simulation* is a binary relation $R \subseteq S \times S$ such that for all (s_1, s_2) in R,

- for all $p \in \mathsf{P}$: $p \in \rho(s_1)$ iff $p \in \rho(s_2)$;
- for all $a \in \Sigma$ and $s_1' \in S$, if $s_1 \xrightarrow{a} s_1'$, then there is $s_2' \in S$ such that $s_2 \xrightarrow{a} s_2'$ and $(s_1', s_2') \in R$.

Two states s, s' are *simulation equivalent*, $s \backsimeq s'$, if there are simulations R, R' such that $(s, s') \in R$ and $(s', s) \in R'$. A simulation R is a *bisimulation* if $R = R^{-1}$; we say that s, s' are *bisimilar*, $s \sim s'$, if there is a bisimulation that contains (s, s'). We say that two valuations are bisimilar, $v \sim v'$, if for all $x \in \mathsf{Var}$, $v(x) \sim v'(x)$.

Proposition 3. *[7] $\mathscr{L}_\mu^\omega$ is closed under bisimulation: if $v \in [\![\varphi]\!]$ and $v \sim v'$, then $v' \in [\![\varphi]\!]$.*

Let us now explain how these process equivalences can be decided by the model checking algorithms: the following formula captures valuations v such that $v(x) \sim v(y)$

$$\nu X. \bigwedge_{p \in \mathsf{P}} p(x) \Leftrightarrow p(y) \;\wedge\; \bigwedge_{a \in \Sigma} [a]_x \langle a \rangle_y X \;\wedge\; \{(x,y) \leftarrow (y,x)\} X$$

whereas the following formula captures valuations v such that $v(x) \backsimeq v(y)$

$$\nu X \big(\nu Y. \bigwedge_{p \in \mathsf{P}} p(x) \Leftrightarrow p(y) \;\wedge\; \bigwedge_{a \in \Sigma} [a]_x \langle a \rangle_y Y \big) \;\wedge\; \{(x,y) \leftarrow (y,x)\} X.$$

Automata Theory. A second application of $\mathscr{L}_\mu^\omega$ is in the field of automata theory. To illustrate this aspect, we pick some language inclusion problems that can be solved in polynomial-time.

A non-deterministic Büchi automaton can be viewed as a finite LTS $A = (S, s_0, \Delta, \rho)$ where ρ interprets a predicate final. Remember that a run on an infinite word $w \in \Sigma^\omega$ in A is accepting if it visits infinitely often a final state. The set of words $L(A) \subseteq \Sigma^\omega$ that have an accepting run is called the language accepted by A.

The language inclusion problem $L(A) \subseteq L(B)$ is PSPACE-hard for arbitrary Büchi automata and therefore unlikely to be definable in $\mathscr{L}_\mu^\omega$. In the restricted case of B being deterministic, it becomes solvable in polynomial time. Remember that a Büchi automaton is called deterministic if for all $a \in \Sigma$, for all $s, s_1, s_2 \in S$, if $s \xrightarrow{a} s_1$ and $s \xrightarrow{a} s_2$, then $s_1 = s_2$.

Let us now encode the language inclusion problem $L(A) \subseteq L(B)$ as a $\mathscr{L}_\mu^\omega$ model checking problem. To shorten a bit the formula, we assume that B is moreover *complete*, *i.e.* for all $s \in S$, for all $a \in \Sigma$, there is at least one s' such that $s \xrightarrow{a} s'$. Let us introduce the modality $\langle synch \rangle \varphi \triangleq \bigvee_{a \in \Sigma} \langle a \rangle_x \langle a \rangle_y \varphi$. Consider the formula

$$\varphi_{incl} \;\triangleq\; \langle synch \rangle^* \nu Z_1. \Big(\mathsf{final}(x) \wedge \neg \mathsf{final}(y) \wedge \mu Z_2. \langle synch \rangle \big(Z_1 \vee (\neg \mathsf{final}(y) \wedge Z_2) \big) \Big)$$

Let $\mathfrak{M}_{A,B}$ be the LTS obtained as the disjoint union of A and B with initial states s_A of A and s_B of B respectively. Then $L(A)$ is included in $L(B)$ if and only if $\mathfrak{M}_{A,B}, v \not\models \varphi_{incl}$ where $v(x) = s_A$ and $v(y) = s_B$. Indeed, this formula is satisfied if there is a run r_A of A and a run r_B of B reading the same word $w \in \Sigma^\omega$ such that r_A visits a final state of A infinitely often, whereas r_B eventually stops visiting the final states of B. Since B is deterministic, no other run r'_B could read w, thus $w \in L(A) \backslash L(B)$.

The same ideas can be applied to parity automata. A parity automaton is a finite automaton where states are assigned priorities; it can be seen as an LTS (S, s_0, Δ, ρ) where ρ interprets *priority predicates* prty_k in such a way that $\rho(s)$ is a singleton $\{\mathsf{prty}_k\}$ for all $s \in S$. A word $w \in \Sigma^\omega$ is accepted by a parity automaton if there is a run of w such that the largest priority visited infinitely often is even. Consider the formulas $\mathsf{prty}_{\leq m}(x) = \mathsf{prty}_0(x) \vee \ldots \vee \mathsf{prty}_m(x)$ and

$$\varphi_{n,m} \;=\; \langle synch \rangle^* \nu Z. \langle synch' \rangle^+ \big(\mathsf{prty}_n(x) \wedge \langle synch' \rangle^+ (\mathsf{prty}_m(y) \wedge Z)\big)$$

where $\langle synch' \rangle^+ \varphi$ is a shorthand for $\mu Z. \langle synch \rangle \mathsf{prty}_{\leq n}(x) \wedge \mathsf{prty}_{\leq m}(y) \wedge (\varphi \vee Z)$. Then $\varphi_{n,m}$ asserts that there are two runs r_A and r_B of two parity automata A and B recognizing the same word w such that the highest priorities visited infinitely often by r_A and r_B are respectively n and m. Since $L(A) \not\subseteq L(B)$ if and only there is an even n and an odd m such that $\mathfrak{M}_{A,B} \models \varphi_{n,m}$, this gives us again a decision procedure for the language inclusion problem of parity automata when B is deterministic complete.

Parsing of Formal Languages. A third application of $\mathscr{L}_\mu^\omega$ is in the field of parsing for formal, namely context-free languages. To each finite word w, we may associate its linear LTS $\mathfrak{M}_w$. For instance, for $w = aab$, $\mathfrak{M}_w$ is the LTS $\bigcirc \xrightarrow{a} \bigcirc \xrightarrow{a} \bigcirc \xrightarrow{b} \bigcirc$. Let us now consider a context-free grammar G, and define a formula that describes the language of G. To ease the presentation, we assume that G is in Chomsky normal form, but a linear-size formula would be derivable for an arbitrary context-free grammar as well. The production rules of G are thus of the form either $X_i \to X_j X_k$ or $X_i \to a$, for $X_1, \ldots, X_n$ the non-terminals of G. Let us pick variables x,y and z, intended to represent respectively the initial the final, and an intermediate position in the (sub)word currently parsed. To every non-terminal X_i, we associate the recursive definition:

$$\varphi_i \;=_\mu\; \bigvee_{X_i \to a} \langle a \rangle_x \, x \sim y \;\vee\; \bigvee_{X_i \to X_j X_k} \{z \leftarrow x\} \langle - \rangle_z^* \big((\{y \leftarrow z\} \varphi_j) \wedge (\{x \leftarrow z\} \varphi_k) \big)$$

where $x \sim y$ is the formula characterizing bisimilarity and $\langle - \rangle_z^* \varphi$ is $\mu Z. \varphi \vee \bigvee_{a \in \Sigma} \langle a \rangle_z Z$. If $v(x)$ and $v(y)$ are respectively the initial and final states of $\mathfrak{M}_w$, then $\mathfrak{M}_w, v \models \varphi_i$ is equivalent to w being derivable in G starting with the symbol X_i.

String Problems. Model Checking for $\mathscr{L}_\mu^\infty$ can even be useful for computation (as opposed to decision) problems. Consider for example the Longest Common Subword problem: given words $w_1, \ldots, w_m$ over some alphabet Σ, find a longest v that is a subword of all w_i. This problem is NP-complete for an unbounded number of input words. Thus, we consider the problem restricted to some fixed m, and it is possible to define a formula $\varphi_{\mathrm{LCSW}}^m \in \mathscr{L}_\mu^m$ such that model checking this formula on a suitable representation of the w_i essentially computes such a common subword.

For the LTS take the disjoint union of all $\mathfrak{M}_{w_i}$ for $i = 1, \ldots, m$, and assume that each state in $\mathfrak{M}_{w_i}$ is labeled with a proposition p_i which makes it possible to define m-tuples of states in which the i-th component belongs to $\mathfrak{M}_{w_i}$. Now consider the formula

$$\varphi_{\mathrm{LCSW}}^m := \nu X. \bigwedge_{i=1}^m p_i(x_i) \wedge \bigvee_{a \in \Sigma} \langle a \rangle_1 \ldots \langle a \rangle_m X$$

Note that $\varphi^m_{\mathrm{LCSW}}$ is unsatisfiable for any $m \geq 1$. Thus, a symbolic model checking algorithm for instance would always return the empty set of tuples when called on this formula and any LTS. However, on an LTS representing $w_1, \ldots, w_m$ as described above it consecutively computes in the j-th round of the fixpoint iteration, all tuples of positions $h_1, \ldots, h_m$ such that the subwords in w_i from position $h_i - j$ to h_i are all the same for every $i = 1, \ldots, m$. Thus, it computes, in its penultimate round the positions inside the input words in which the longest common substrings end. Their starting points can easily be computed by maintaining a counter for the number of fixpoint iterations done in the model checking run.

In the same way, it is possible to compute the longest common subsequence of input words $w_1, \ldots, w_m$. A subsequence of w is obtained by deleting arbitrary symbols, whereas a subword is obtained by deleting an arbitrary prefix and suffix from w. The Longest Common Subsequence problem is equally known to be NP-complete for unbounded m. For any fixed m, however, the following formula can be used to compute all longest common subsequences of such input words using model checking technology in the same way as it is done in the case of the Longest Common Subword problem.

$$\varphi^m_{\mathrm{LCSS}} := \nu X. \bigwedge_{i=1}^{m} p_i(x_i) \wedge \bigvee_{a \in \Sigma} \langle a \rangle_{x_1} \langle - \rangle^*_{x_1} \ldots \langle a \rangle_{x_m} \langle - \rangle^*_{x_m} X$$

where $\langle - \rangle^*_{x_i} \psi$ stands for $\mu Y. \psi \vee \bigvee_{a \in \Sigma} \langle a \rangle_{x_i} Y$.

Games. The Cat and Mouse Game is played on a directed graph with three distinct nodes c, m and t as follows. Initially, the cat resides in node c, the mouse in node m. In each round, the mouse moves first. He can move along an edge to a successor node of the current one or stay on the current node, then the cat can do the same. If the cat reaches the mouse, she wins; otherwise, if the mouse reaches the target node t, he wins; otherwise, the mouse runs forever without being caught nor reaching the target node: in that case, the cat wins. The problem of solving the Cat and Mouse Game is to decide whether or not the mouse has a winning strategy for a given graph.

Note that this problem is not bisimulation-invariant under the straight-forward encoding of the directed graph as an LTS with a single proposition t to mark the target node. Consider for example the following two, bisimilar game arenas.

Clearly, if the cat and mouse start on the two separate leftmost nodes then the mouse can reach the target first. However, these nodes are bisimilar to the left node of the right graph, and if they both start on this one then the cat has caught the mouse immediately.

Thus, winning strategies cannot necessarily be defined in $\mathscr{L}^\infty_\mu$. However, it is possible to define them when a new atomic formula $eq(x,y)$ expressing that x and y evaluate to the same node, is being added to the syntax of $\mathscr{L}^\infty_\mu$ (standard model checking procedures can be extended to handle the equality predicate eq as well).

$$\varphi_{\mathrm{CMG}} := \mu X. (t(x) \wedge \neg eq(x,y)) \vee \langle - \rangle_x (\neg eq(x,y)) \wedge [-]_y X)$$

We have $v \models \varphi_{\mathrm{CMG}}$ if and only if the mouse can win from position $v(x)$ when the cat is on position $v(y)$ initially.

4 Conclusion

We have considered the modal fixpoint logic $\mathscr{L}_\mu^\omega$ for a potential use in algorithm design and given examples of problems which can be defined in $\mathscr{L}_\mu^\omega$. The combination of fixpoint quantifiers and modal operators has been proved to be very fruitful for obtaining algorithmic solutions for problems in automatic program verification. The examples boost the idea of using successful model checking technology in other areas too.

The use of model checking algorithms on fixed formulas does not provide a generic recipe that miraculously generates efficient algorithms, but it provides the potential to do so. The next step on this route towards an efficient algorithm for some problem P requires partial evaluation on a model checking algorithm and the formula φ_P defining P. This usually requires manual tweaking of the algorithm and is highly dependent on the actual φ_P. Thus, future work on this direction would consist of consequently optimising $\mathscr{L}_\mu^\omega$ model checking algorithms for certain definable problems and testing their efficiency in practice.

On a different note, $\mathscr{L}_\mu^\omega$ is an interesting fixpoint calculus for which the model checking problem over infinite-state transition systems has not been quite studied so far. The most prominent result in this area is the decidability of $\mathscr{L}_\mu^1$ over pushdown LTS [10]. However, model checking $\mathscr{L}_\mu^\omega$ — or even just $\mathscr{L}_\mu^k$ for some $k \geq 2$ — seems undecidable for pushdown LTS. It is questionable whether model checking of $\mathscr{L}_\mu^\omega$ is decidable for any popular class of infinite-state transition systems.

References

[1] R. Axelsson & M. Lange (2007): *Model Checking the First-Order Fragment of Higher-Order Fixpoint Logic*. In: *Proc. 14th Int. Conf. on Logic for Programming, Artificial Intelligence, and Reasoning, LPAR'07, LNCS* 4790, Springer, pp. 62–76, doi:10.1007/978-3-540-75560-9_7.

[2] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill & L. J. Hwang (1992): *Symbolic Model Checking:* 10^{20} *States and Beyond*. *Information and Computation* 98(2), pp. 142–170, doi:10.1016/0890-5401(92)90017-A.

[3] E. Grädel, P. G. Kolaitis, L. Libkin, M. Marx, J. Spencer, M. Y. Vardi, Y. Venema & S. Weinstein (2007): *Finite Model Theory and its Applications*. Springer-Verlag, doi:10.1007/3-540-68804-8.

[4] D. Janin & I. Walukiewicz (1996): *On the Expressive Completeness of the Propositional μ-Calculus with Respect to Monadic Second Order Logic*. In: *CONCUR*, pp. 263–277, doi:10.1007/3-540-61604-7_60.

[5] M. Jurdziński (1998): *Deciding the winner in parity games is in UP∩co-UP*. *Inf. Process. Lett.* 68(3), pp. 119–124, doi:10.1016/S0020-0190(98)00150-1.

[6] D. Kozen (1983): *Results on the Propositional μ-calculus*. *TCS* 27, pp. 333–354, doi:10.1007/BFb0012782.

[7] M. Otto (1999): *Bisimulation-invariant PTIME and higher-dimensional μ-calculus*. *Theor. Comput. Sci.* 224(1-2), pp. 237–265, doi:10.1016/S0304-3975(98)00314-4.

[8] C. Stirling (1995): *Local Model Checking Games*. In: *Proc. 6th Conf. on Concurrency Theory, CONCUR'95, LNCS* 962, Springer, pp. 1–11, doi:10.1007/3-540-60218-6_1.

[9] J. Vöge & M. Jurdziński (2000): *A Discrete Strategy Improvement Algorithm for Solving Parity Games*. In: *CAV*, pp. 202–215, doi:10.1007/10722167_18.

[10] Igor Walukiewicz (1996): *Pushdown Processes: Games and Model Checking*. In: *CAV*, pp. 62–74, doi:10.1007/3-540-61474-5_58.

Cut-elimination for the mu-calculus with one variable

Grigori Mints
Dept. of Philosophy
Stanford University
USA
gmints@stanford.edu

Thomas Studer
Inst. of Computer Science and Appl. Math.
University of Bern
Switzerland
tstuder@iam.unibe.ch

We establish syntactic cut-elimination for the one-variable fragment of the modal mu-calculus. Our method is based on a recent cut-elimination technique by Mints that makes use of Buchholz' Ω-rule.

1 Introduction

The propositional modal μ-calculus is a well-established modal fixed point logic that includes fixed points for arbitrary positive formulae. Thus it subsumes many temporal logics (with an always operator), epistemic logics (with a common knowledge operator), and program logics (with an iteration operator).

Making use of the finite model property, Kozen [10] introduces a sound and complete infinitary system for the modal μ-calculus. In this system greatest fixed points are introduced by means of the ω-rule that has a premise for each finite approximation of the greatest fixed point. Jäger et al. [8] show by *semantic* methods that the cut rule is admissible in this kind of infinitary systems. So far, however, there is no *syntactic* cut-elimination procedure available for the modal μ-calculus. It is our aim in this paper to present an effective cut-elimination method for the one-variable fragment of the μ-calculus.

There are already a few results available on syntactic cut-elimination for modal fixed point logics. Most of them make use of deep inference where rules may not only be applied to outermost connectives but also deeply inside formulae. The first result of this kind has been obtained by Pliuskevicius [12] who presents a syntactic cut-elimination procedure for linear time temporal logic. Brünnler and Studer [2] employ nested sequents to develop a cut-elimination procedure for the logic of common knowledge. Hill and Poggiolesi [7] use a similar approach to establish effective cut-elimination for propositional dynamic logic. A generalization of this method is studied in [3] where it is also shown that it cannot be extended to fixed points that have a $\Box$-operator in the scope of a μ-operator. Fixed points of this kind occur, for instance, in CTL in the form of universal path quantifiers.

Thus we need a more general approach to obtain syntactic cut-elimination for the modal μ-calculus. A standard proof-theoretic technique to deal with inductive definitions and fixed points is Buchholz' Ω-rule [4, 6]. Jäger and Studer [9] present a formulation of the Ω-rule for non-iterated modal fixed point logic and they obtain cut-elimination for positive formulae of this logic. In order to overcome this restriction to positive formulae, Mints [11] introduces an Ω-rule that has a wider set of premises, which enables him to obtain full cut-elimination for non-iterated modal fixed point logic.

Mints' cut-elimination algorithm makes use of, in addition to ideas from [5], a new tool presented in [11]. It is based on the distinction, see [13], between implicit and explicit occurrences of formulae in a derivation with cut. If an occurrence of a formula is traceable to the endsequent of the derivation, then it is called explicit. If it is traceable to a cut-formula, then it is an implicit occurrence.

Implicit and explicit occurrences of greatest fixed points are treated differently in the translation of the induction rule to the infinitary system. An instance of the induction rule that derives a sequent

D. Miller and Z. Ésik (Eds.): Fixed Points
in Computer Science 2012 (FICS 2012)
EPTCS 77, 2012, pp. 47–54, doi:10.4204/EPTCS.77.7

$\nu X.A, B$ goes to an instance of the ω-rule if $\nu X.A$ is explicit. Otherwise, if $\nu X.A$ is traceable to a cut-formula, the induction rule is translated to an instance of the Ω-rule that is preserved until the last stage of cut-elimination. At that stage, called collapsing, the Ω-rule is eliminated completely.

In the present paper we show that this method can be extended to a μ-calculus with iterated fixed points. Hence we obtain complete syntactic cut-elimination for the one-variable fragment of the modal μ-calculus. Our infinitary system is completely cut-free in the sense that there are not only no cut rules in the system but also no embedded cuts. Thus our cut-free system enjoys the subformula property. This is in contrast to the recent cut-elimination results by Baelde [1] and by Tiu and Momigliano [14] for the finitary systems $\mu\mathsf{MALL}$ and Linc^{-}, respectively, where the ν-introduction rule and the co-induction rule contain embedded cuts, which results in the loss of the subformula property.

2 Syntax and semantics

We first introduce the language $\mathscr{L}$. We start with a countable set PROP of atomic propositions p_i and their negations $\overline{p_i}$. We use P to denote an arbitrary element of PROP. Moreover, we will use a special variable X.

Definition 1. *Operator forms* $A, B, \ldots$ are given by the following grammar:

$$A :== p_i \mid \overline{p_i} \mid X \mid A \wedge A \mid A \vee A \mid \Box A \mid \Diamond A \mid \mu X.A \mid \nu X.A.$$

Formulae F are defined by:

$$F :== p_i \mid \overline{p_i} \mid F \wedge F \mid F \vee F \mid \Box F \mid \Diamond F \mid \mu X.A \mid \nu X.A.$$

The fixed point operators μ and ν bind the variable X and, therefore, we will talk of free and bound occurrences of X. Hence a formula is an operator form without free occurrences of X.

The negation of an operator form is inductively defined as follows.

1. $\neg p_i := \overline{p_i}$ and $\neg\overline{p_i} := p_i$
2. $\neg X := X$
3. $\neg(A \wedge B) := \neg A \vee \neg B$ and $\neg(A \vee B) := \neg A \wedge \neg B$
4. $\neg\Box A := \Diamond\neg A$ and $\neg\Diamond A := \Box\neg A$
5. $\neg\mu X.A := \nu X.\neg A$ and $\neg\nu X.A := \mu X.\neg A$

Note that negation is well-defined: the negation of an X-positive operator form is again X-positive since we have $\neg X := X$. Thus, for example,

$$\neg\mu X.\Box(p_i \wedge X) := \nu X.\neg\Box(p_i \wedge X) := \nu X.\Diamond\neg(p_i \wedge X) := \nu X.\Diamond(\neg p_i \vee \neg X) := \nu X.\Diamond(\overline{p_i} \vee X).$$

For an arbitrary but fixed atomic proposition p_i we set $\top := p_i \vee \overline{p_i}$. If A is an operator form, then we write $A(B)$ for the result of simultaneously substituting B for every free occurrence of X in A. We will also use finite iterations of operator forms, given as follows

$$A^0(B) := B \text{ and } A^{k+1}(B) := A(A^k(B)).$$

$$\Gamma, P, \neg P \qquad\qquad \Gamma, \mu X.A, \neg\mu X.A$$

$$\frac{\Gamma, A, B}{\Gamma, A \vee B}\,(\vee) \qquad \frac{\Gamma, A \qquad \Gamma, B}{\Gamma, A \wedge B}\,(\wedge) \qquad \frac{\Gamma, A}{\Diamond\Gamma, \Box A, \Sigma}\,(\Box)$$

$$\frac{\Gamma, A(\mu X.A)}{\Gamma, \mu X.A}\,(\mathsf{clo}) \qquad \frac{\neg A(B), B}{\neg\mu X.A, B}\,(\mathsf{ind}) \qquad \frac{\Gamma, A \qquad \Gamma, \neg A}{\Gamma}\,(\mathsf{cut})$$

Figure 1: System **M**

3 System M

System **M** derives sequents, that are finite sets of formulae. We denote sequents by Γ, Σ and use the following notation: if $\Gamma := \{A_1, \ldots, A_n\}$, then $\Diamond\Gamma := \{\Diamond A_1, \ldots, \Diamond A_n\}$, System **M** consists of the axioms and rules given in Figure 1.

4 System $\mathbf{M}^{\omega}$

System $\mathbf{M}^{\omega}$ is an infinitary cut-free system for the modal μ-calculus with one variable. It consists of the axioms and rules given in Figure 2.

$$\Gamma, P, \neg P$$

$$\frac{\Gamma, A, B}{\Gamma, A \vee B}\,(\vee) \qquad \frac{\Gamma, A \qquad \Gamma, B}{\Gamma, A \wedge B}\,(\wedge) \qquad \frac{\Gamma, A}{\Diamond\Gamma, \Box A, \Sigma}\,(\Box)$$

$$\frac{\Gamma, A(\mu X.A)}{\Gamma, \mu X.A}\,(\mathsf{clo}) \qquad \frac{\Gamma, A^i(\top) \text{ for all natural numbers } i}{\Gamma, \nu X.A}\,(\omega)$$

Figure 2: System $\mathbf{M}^{\omega}$

5 System $\mathbf{M}_k^{\omega,\Omega}$

In order to embed **M** into $\mathbf{M}^{\omega}$, we need a family of intermediate systems $\mathbf{M}_k^{\omega,\Omega}$ that include additional rules to derive greatest fixed points that later will be cut away.

The language $\mathscr{L}_\Omega$ extends $\mathscr{L}$ by a new connective ν' to denote those greatest fixed points. Formally, $\mathscr{L}_\Omega$ is given as follows. Operator forms of $\mathscr{L}_\Omega$ are defined like operator forms of $\mathscr{L}$ with the additional case

1. If A is an operator form, then $\nu' X.A$ is also an operator form.

A formula of $\mathscr{L}_\Omega$ is an $\mathscr{L}_\Omega$ operator form without free occurrence of X. A formula is a *greatest fixed point* if it has the form $\nu X.A$ or $\nu' X.A$.

Definition 2. The level $\mathsf{lev}(A)$ of an operator form A is the maximal nesting of fixed point operators in A. Formally we set:

1. $\mathsf{lev}(P) := \mathsf{lev}(X) := 0$ for all P in PROP
2. $\mathsf{lev}(A \wedge B) := \mathsf{lev}(A \vee B) := \max(\mathsf{lev}(A), \mathsf{lev}(B))$
3. $\mathsf{lev}(\Box A) := \mathsf{lev}(\Diamond A) := \mathsf{lev}(A)$
4. $\mathsf{lev}(\mu X.A) := \mathsf{lev}(\nu X.A) := \mathsf{lev}(\nu' X.A) := \mathsf{lev}(A) + 1$

The level of a sequent is the maximum of the levels of its formulae. We say a formula (sequent) is *k-positive* if for all $\nu' X.A$ occurring in it we have $\mathsf{lev}(\nu' X.A) < k$.

When working in $\mathbf{M}_k^{\omega,\Omega}$, we will use the following notation: the formula A' is obtained from A by replacing all occurrences of νX in A with $\nu' X$.

Let $k \geq 0$. System $\mathbf{M}_k^{\omega,\Omega}$ consists of the axioms and rules of $\mathbf{M}^\omega$ (formulated in $\mathscr{L}_\Omega$) and the additional rules: cut, Ω_h, and $\tilde{\Omega}_h$. The cut rule is given as follows

$$\frac{\Gamma, A' \qquad \Gamma, (\neg A)'}{\Gamma}\ (\mathsf{cut}),$$

where A is a formula with $\mathsf{lev}(A) \leq k$. The rules Ω_h and $\tilde{\Omega}_h$, where $1 \leq h \leq k$, are informally described as follows:

$$\frac{\cdots \qquad \dfrac{\mathbf{M}_{k-1}^{\omega,\Omega} \vdash_0 \Delta, (\mu X.A)'}{\Delta, \Gamma} \qquad \cdots}{\Gamma, (\neg \mu X.A)'}\ \Omega_h$$

and

$$\frac{\Gamma, (\mu X.A)' \qquad \cdots \qquad \dfrac{\mathbf{M}_{k-1}^{\omega,\Omega} \vdash_0 \Delta, (\mu X.A)'}{\Delta, \Gamma} \qquad \cdots}{\Gamma}\ \tilde{\Omega}_h$$

where $\mathsf{lev}((\neg \mu X.A)') = h$ and Δ ranges over h-positive sequents such that there is a cut-free proof of the sequent $\Delta, (\mu X.A)'$ in $\mathbf{M}_{k-1}^{\omega,\Omega}$.

Definition 3. We use $\mathbf{M}_k^{\omega,\Omega} \vdash_0 \Gamma$ to express that there is a cut-free derivation of Γ in $\mathbf{M}_k^{\omega,\Omega}$.

In a more formal notation we can state the Ω_h-rule as follows. If for every h-positive sequent Δ

$$\mathbf{M}_{k-1}^{\omega,\Omega} \vdash_0 \Delta, (\mu X.A)' \quad \Longrightarrow \quad \mathbf{M}_k^{\omega,\Omega} \vdash \Delta, \Gamma,$$

then

$$\mathbf{M}_k^{\omega,\Omega} \vdash \Gamma, (\neg \mu X.A)',$$

and similarly for $\tilde{\Omega}_h$.

Note that System $\mathbf{M}_0^{\omega,\Omega}$ does not include Ω_h- or $\tilde{\Omega}_h$-rules. Hence we immediately get the following lemma.

Lemma 4. *Let Γ be an $\mathscr{L}$ sequent. We have*

$$\mathbf{M}_0^{\omega,\Omega} \vdash_0 \Gamma \quad \Longrightarrow \quad \mathbf{M}^\omega \vdash \Gamma.$$

6 Embedding

In this section we present a translation from **M**-proofs into $\mathbf{M}_k^{\omega,\Omega}$-proofs. First we establish an auxiliary lemma.

Lemma 5. *For all natural numbers $h \leq k$ we have the following.*

1. *If* $\mathsf{lev}(\mu X.A) = h$, *then* $\mathbf{M}_k^{\omega,\Omega} \vdash_0 \mu X.A, \neg\mu X.A$.
2. *If* $\mathsf{lev}(A) = h$, *then* $\mathbf{M}_k^{\omega,\Omega} \vdash_0 \Gamma, A' \quad \Longrightarrow \quad \mathbf{M}_k^{\omega,\Omega} \vdash_0 \Gamma, A$.
3. *If* $\mathsf{lev}(\mu X.A) = h$, *then* $\mathbf{M}_k^{\omega,\Omega} \vdash_0 \mu X.A, (\neg\mu X.A)'$.
4. *If* $\mathsf{lev}(A) = h$, *then* $\mathbf{M}_k^{\omega,\Omega} \vdash_0 B, C \quad \Longrightarrow \quad \mathbf{M}_k^{\omega,\Omega} \vdash_0 (\neg A)(B), A(C)$.
5. *If* $\mathsf{lev}(A) = h$, *then* $\mathbf{M}_k^{\omega,\Omega} \vdash_0 B, C' \quad \Longrightarrow \quad \mathbf{M}_k^{\omega,\Omega} \vdash_0 (\neg A)(B), A'(C')$.

Proof. The five statements are shown simultaneously by induction on h. For space considerations we show only one particular case of the second statement, which is shown by induction on the derivation of Γ, A' and a case distinction on the last rule. Assume the last rule is an instance of Ω_h with main formula A'. We have $A' = (\nu X.A_0)'$ with $\mathsf{lev}(A_0) < h$. By the premise of the Ω_h-rule we have for all h-positive sequents Δ

$$\mathbf{M}_{k-1}^{\omega,\Omega} \vdash_0 \Delta, (\mu X.\neg A_0)' \quad \Longrightarrow \quad \mathbf{M}_k^{\omega,\Omega} \vdash_0 \Delta, \Gamma. \tag{1}$$

Trivially we have

$$\mathbf{M}_k^{\omega,\Omega} \vdash_0 \top, \Gamma. \tag{2}$$

We also have

$$\mathbf{M}_{k-1}^{\omega,\Omega} \vdash_0 \top, (\mu X.\neg A_0)'$$

from which we get by the induction hypothesis for the fifth claim of this lemma

$$\mathbf{M}_{k-1}^{\omega,\Omega} \vdash_0 A_0(\top), (\neg A_0)'((\mu X.\neg A_0)').$$

An application of clo yields

$$\mathbf{M}_{k-1}^{\omega,\Omega} \vdash_0 A_0(\top), (\mu X.\neg A_0)'.$$

By (1) we get

$$\mathbf{M}_k^{\omega,\Omega} \vdash_0 A_0(\top), \Gamma. \tag{3}$$

Note that (2) and (3) are the first two premises of an instance of ω. By further iterating this we obtain for all i

$$\mathbf{M}_k^{\omega,\Omega} \vdash_0 A_0^i(\top), \Gamma.$$

Hence an application of ω yields

$$\mathbf{M}_k^{\omega,\Omega} \vdash_0 \nu X.A_0, \Gamma.$$

□

We will need a certain form of the induction rule in $\mathbf{M}_k^{\omega,\Omega}$, which we are going to derive next. We write $\Sigma[(\mu X.A)' := B]$ for the result of simultaneously replacing in every formula in Σ every occurrence of $(\mu X.A)'$ with B.

Lemma 6. *Let A be an operator form with $\mathsf{lev}(\nu X.A) \leq k$. Let $\Delta, \Sigma_1, \Sigma_2$ be h-positive sequents and let B be a formula with $\mathsf{lev}(B) \leq k$. Assume that*

$$\mathbf{M}_k^{\omega,\Omega} \vdash (\neg A(B))', B \quad \textit{and} \quad \mathbf{M}_k^{\omega,\Omega} \vdash (\neg A(B))', B'.$$

Then we have, if

$$\mathbf{M}_{k-1}^{\omega,\Omega} \vdash_{\overline{0}} \Delta, \Sigma_1, \Sigma_2$$

then

$$\mathbf{M}_k^{\omega,\Omega} \vdash \Delta, \Sigma_1[(\mu X.A)' := B], \Sigma_2[(\mu X.A)' := B'].$$

Lemma 7. *Let A be an operator form with $\mathsf{lev}(\nu X.A) \leq k$. Further let B be an arbitrary formula with $\mathsf{lev}(B) \leq k$. Assume that*

$$\mathbf{M}_k^{\omega,\Omega} \vdash (\neg A(B))', B \quad \textit{and} \quad \mathbf{M}_k^{\omega,\Omega} \vdash (\neg A(B))', B'.$$

Then we have

$$\mathbf{M}_k^{\omega,\Omega} \vdash (\neg \mu X.A)', B \quad \textit{and} \quad \mathbf{M}_k^{\omega,\Omega} \vdash (\neg \mu X.A)', B'.$$

Proof. Let $h = \mathsf{lev}(\nu X.A)$. In view of our assumptions and the previous lemma we know that for all h-positive sequents Δ

$$\mathbf{M}_{k-1}^{\omega,\Omega} \vdash_{\overline{0}} \Delta, (\mu X.A)' \quad \Longrightarrow \quad \mathbf{M}_k^{\omega,\Omega} \vdash \Delta, B.$$

Hence by an application of the Ω_h-rule we conclude $\mathbf{M}_k^{\omega,\Omega} \vdash (\neg \mu X.A)', B$. Similarly, we can derive $\mathbf{M}_k^{\omega,\Omega} \vdash (\neg \mu X.A)', B'$. □

Theorem 8. *Let Γ be a sequent of $\mathscr{L}$. Assume $\mathbf{M} \vdash \Gamma$ and assume further for any sequent Δ occurring in that proof we have $\mathsf{lev}(\Delta) \leq k$. Then we have $\mathbf{M}_k^{\omega,\Omega} \vdash \Gamma$.*

Proof. An operation σ on sequents is called '-operation if $\sigma(\Gamma, A_1, \ldots, A_n) = \Gamma, A_1', \ldots, A_n'$. The result of applying σ to a sequent Γ is denoted Γ^σ.

To establish the theorem, we show by induction on the depth of the **M**-proof that for all '-operations σ, we have $\mathbf{M}_k^{\omega,\Omega} \vdash \Gamma^\sigma$. We distinguish the following cases for the last rule.

1. Γ is an axiom different from $\Gamma_0, \mu X.A, \neg \mu X.A$. Then Γ^σ is an axiom of $\mathbf{M}_k^{\omega,\Omega}$, too.
2. Γ is $\Gamma_0, \mu X.A, \neg \mu X.A$. Then Γ^σ follows either by the first or the third claim of Lemma 5 depending on whether $\neg \mu X.A$ is replaced by σ or not.
3. The last rule is an instance of $\wedge$, $\vee$, $\Box$ or clo. We can apply the same rule in $\mathbf{M}_k^{\omega,\Omega}$.
4. The last rule is a cut
$$\frac{\Gamma, A \qquad \Gamma, \neg A}{\Gamma}.$$
We extend the current '-operation σ to a '-operation τ such that $(\Gamma, A)^\tau = \Gamma^\sigma, A'$ and $(\Gamma, \neg A)^\tau = \Gamma^\sigma, (\neg A)'$ By the induction hypothesis for the '-operation τ we obtain $\mathbf{M}_k^{\omega,\Omega} \vdash \Gamma^\sigma, A'$ as well as $\mathbf{M}_k^{\omega,\Omega} \vdash \Gamma^\sigma, (\neg A)'$. With an instance of cut we get $\mathbf{M}_k^{\omega,\Omega} \vdash \Gamma^\sigma$.
5. The last rule is an instance of the induction rule. Then the endsequent has the form $\neg \mu X.A, B$ which is $\nu X.\neg A, B$. There are two possible cases.
   (a) The principal occurrence of $\nu X.\neg A$ is not changed by σ. By the induction hypothesis we can derive $(\neg A(B))', B^\sigma$ and $(\neg A(B))', B'$. We obtain our claim by the following proof.

$$\dfrac{\dfrac{}{(\neg A(B))', B'}\,\text{I.H.} \qquad \dfrac{\top, B'}{(\neg A)(\top), (A(B))'}\,\text{L. 5}}{(\neg A)(\top), B'}\,\mathsf{cut}$$

$$\vdots$$

$$\dfrac{\cdots \qquad \dfrac{\dfrac{}{(\neg A(B))', B^{\sigma}}\,\text{I.H.} \qquad \dfrac{(\neg A)^{i}(\top), B'}{(\neg A)^{i+1}(\top), (A(B))'}\,\text{L. 5}}{(\neg A)^{i+1}(\top), B^{\sigma}}\,\mathsf{cut} \qquad \cdots}{\nu X.\neg A, B^{\sigma}}\,\omega$$

(b) The principal occurrence of $\nu X.\neg A$ is changed by σ. Let τ_1, τ_2 be '-operations such that

$$(\neg A(B), B)^{\tau_1} = (\neg A(B))', B$$

and

$$(\neg A(B), B)^{\tau_2} = (\neg A(B))', B'.$$

By the induction hypothesis for τ_1 and τ_2 we obtain

$$\mathbf{M}_k^{\omega,\Omega} \vdash (\neg A(B))', B \quad \text{and} \quad \mathbf{M}_k^{\omega,\Omega} \vdash (\neg A(B))', B'.$$

We apply Lemma 7 and conclude $\mathbf{M}_k^{\omega,\Omega} \vdash (\neg \mu X.A)', B^{\sigma}$. □

7 Cut elimination

We eliminate instances of cut in the standard way, see for instance [5, 11], by pushing them up the derivation. When an instance of cut with cut formulae $(\mu X.A)'$ and $(\neg \mu X.A)'$ meets the instance of $\mathbf{\Omega}_h$ that introduces $(\neg \mu X.A)'$, this pair of inferences is replaced by $\tilde{\mathbf{\Omega}}_h$.

Lemma 9 (Cut-elimination)**.** *If* $\mathbf{M}_k^{\omega,\Omega} \vdash \Gamma$, *then* $\mathbf{M}_k^{\omega,\Omega} \vdash_{0} \Gamma$.

The cut-elimination process terminates in a formally cut-free derivation that may contain instances of $\tilde{\mathbf{\Omega}}_h$-rules. Now we show that these instances of $\tilde{\mathbf{\Omega}}_h$ also can be eliminated.

Lemma 10 (Collapsing)**.** *Let* Γ *be an* $(h+1)$*-positive sequent. If* $\mathbf{M}_k^{\omega,\Omega} \vdash_{0} \Gamma$, *then* $\mathbf{M}_h^{\omega,\Omega} \vdash_{0} \Gamma$.

Proof. By transfinite induction on the derivation in $\mathbf{M}_k^{\omega,\Omega}$. The only interesting case is when the last rule is an instance of $\tilde{\mathbf{\Omega}}_l$ for $h < l \leq k$ as follows

$$\dfrac{\Gamma, (\mu X.A)' \qquad \cdots \qquad \begin{array}{c} \mathbf{M}_{l-1}^{\omega,\Omega} \vdash_{0} \Delta, (\mu X.A)' \\ \hdashline \Delta, \Gamma \end{array} \qquad \cdots}{\Gamma}\,\tilde{\mathbf{\Omega}}_l$$

Note that $\Gamma, (\mu X.A)'$ is l-positive. Thus by the induction hypothesis we get

$$\mathbf{M}_{l-1}^{\omega,\Omega} \vdash_{0} \Gamma, (\mu X.A)'. \tag{4}$$

Moreover, also by the induction hypothesis we get for all $(h+1)$-positive Δ

$$\mathbf{M}_{l-1}^{\omega,\Omega} \vdash_{\overline{0}} \Delta, (\mu X.A)' \quad \Longrightarrow \quad \mathbf{M}_{h}^{\omega,\Omega} \vdash_{\overline{0}} \Delta, \Gamma. \tag{5}$$

Now we plug (4) in (5) and obtain $\mathbf{M}_{h}^{\omega,\Omega} \vdash_{\overline{0}} \Gamma$ as required. □

We now have all ingredients ready for our main result.

Corollary 11. *Let Γ be an $\mathscr{L}$-sequent. We have*

$$\mathbf{M} \vdash \Gamma \quad \Longrightarrow \quad \mathbf{M}^{\omega} \vdash \Gamma.$$

Proof. Assume $\mathbf{M} \vdash \Gamma$. By Theorem 8 we get $\mathbf{M}_{k}^{\omega,\Omega} \vdash \Gamma$ for some k. By cut-elimination we obtain $\mathbf{M}_{k}^{\omega,\Omega} \vdash_{\overline{0}} \Gamma$. Then collapsing yields $\mathbf{M}_{0}^{\omega,\Omega} \vdash_{\overline{0}} \Gamma$ which finally gives us $\mathbf{M}^{\omega} \vdash \Gamma$ by Lemma 4. □

References

[1] David Baelde (2009): *Least and greatest fixed points in linear logic*. *CoRR* abs/0910.3383v4. Available at `http://arxiv.org/abs/0910.3383v4`.

[2] Kai Brünnler & Thomas Studer (2009): *Syntactic cut-elimination for common knowledge*. *Annals of Pure and Applied Logic* 160(1), pp. 82–95, doi:10.1016/j.apal.2009.01.014.

[3] Kai Brünnler & Thomas Studer (preprint): *Syntactic cut-elimination for a fragment of the modal mu-calculus*.

[4] Wilfried Buchholz (1981): *The $\Omega_{\mu+1}$-rule*. In Wilfried Buchholz, Solomon Feferman, Wolfram Pohlers & Wilfried Sieg, editors: *Iterated Inductive Definitions and Subsystems of Analysis: Recent Proof Theoretic Studies, Lecture Notes in Mathematics* 897, Springer, pp. 189–233, doi:10.1007/BFb0091898.

[5] Wilfried Buchholz (2001): *Explaining the Gentzen-Takeuti reduction steps: a second-order system*. *Archive for Mathematical Logic* 40(4), pp. 255–272, doi:10.1007/s001530000064.

[6] Wilfried Buchholz & Kurt Schütte (1988): *Proof Theory of Impredicative Subsystems of Analysis*. Bibliopolis.

[7] Brian Hill & Francesca Poggiolesi (2010): *A Contraction-free and Cut-free Sequent Calculus for Propositional Dynamic Logic*. *Studia Logica* 94(1), pp. 47–72, doi:10.1007/s11225-010-9224-z.

[8] Gerhard Jäger, Mathis Kretz & Thomas Studer (2008): *Canonical completeness for infinitary μ*. *Journal of Logic and Algebraic Programming* 76(2), pp. 270–292, doi:10.1016/j.jlap.2008.02.005.

[9] Gerhard Jäger & Thomas Studer (2011): *A Buchholz rule for modal fixed point logics*. *Logica Universalis* 5, pp. 1–19, doi:10.1007/s11787-010-0022-1.

[10] Dexter Kozen (1988): *A finite model theorem for the propositional μ–calculus*. *Studia Logica* 47(3), pp. 233–241, doi:10.1007/BF00370554.

[11] Grigori Mints (to appear): *Effective Cut-elimination for a fragment of Modal mu-calculus*. *Studia Logica* .

[12] Regimantas Pliuskevicius (1991): *Investigation of Finitary Calculus for a Discrete Linear Time Logic by means of Infinitary Calculus*. In: *Baltic Computer Science, Selected Papers*, Springer, pp. 504–528, doi:10.1007/BFb0019366.

[13] Gaisi Takeuti (1987): *Proof Theory*. North-Holland.

[14] Alwen Tiu & Alberto Momigliano (2010): *Cut Elimination for a Logic with Induction and Co-induction*. *CoRR* abs/1009.6171v1. Available at `http://arxiv.org/abs/1009.6171v1`.

Structured general corecursion and coinductive graphs [extended abstract]

Tarmo Uustalu
Institute of Cybernetics at Tallinn University of Technology, Estonia
tarmo@cs.ioc.ee

Bove and Capretta's popular method for justifying function definitions by general recursive equations is based on the observation that any structured general recursion equation defines an inductive subset of the intended domain (the "domain of definedness") for which the equation has a unique solution. To accept the definition, it is hence enough to prove that this subset contains the whole intended domain.

This approach works very well for "terminating" definitions. But it fails to account for "productive" definitions, such as typical definitions of stream-valued functions. We argue that such definitions can be treated in a similar spirit, proceeding from a different unique solvability criterion. Any structured recursive equation defines a coinductive relation between the intended domain and intended codomain (the "coinductive graph"). This relation in turn determines a subset of the intended domain and a quotient of the intended codomain with the property that the equation is uniquely solved for the subset and quotient. The equation is therefore guaranteed to have a unique solution for the intended domain and intended codomain whenever the subset is the full set and the quotient is by equality.

Unique solutions to recursive equations General recursive definitions are commonplace in programming practice.

In particular, it is highly desirable to be able to define functions by some forms of controlled general recursion in type-theoretically motivated languages of total functional programming (in particular, proof assistants) that come with a set-theoretic rather than a domain-theoretic semantics. For an overview of this area, see Bove et al. [5].

In this paper, we are concerned with describing a function $f : A \to B$ definitely by an equation of the form:

$$\begin{tikzcd} FA \arrow[d, "Ff"'] & A \arrow[l, "\alpha"'] \arrow[d, "f"] \\ FB \arrow[r, "\beta"'] & B \end{tikzcd} \tag{1}$$

where A, B are sets (the intended domain and codomain), F is a functor (the branching type of recursive call [corecursive return] trees), α is an F-coalgebra structure on A (marshals arguments for recursive calls) and β is an F-algebra structure on B (collects recursive call results). We are interested in conditions under which the equation is guaranteed to have a unique solution (rather than a least solution in a domain-theoretic setting or some solution that is canonical in some sense). There are several important generalizations of this setting, but we will not treat them here.

There are some well-known good cases.

D. Miller and Z. Ésik (Eds.): Fixed Points
in Computer Science 2012 (FICS 2012)
EPTCS 77, 2012, pp. 55–61, doi:10.4204/EPTCS.77.8

Some good cases (1): Initial algebra The following equation has a unique solution for any B, β.

$$\begin{tikzcd}
1+\mathsf{El}\times\mathsf{List} \arrow[d, "1+\mathsf{El}\times f"'] & \mathsf{List} \arrow[l, "{[\mathsf{nil},cons]}^{-1}"'] \arrow[d, "f"] \\
1+\mathsf{El}\times B \arrow[r, "\beta"'] & B
\end{tikzcd}$$

E.g., for $B = \mathsf{List}$ (lists over El), $\beta = \mathsf{ins}$ (insertion of an element into a list assumed to be sorted), we get $f = \mathsf{isort}$ (insertion sort).

A unique f exists because $(\mathsf{List}, [\mathsf{nil}, \mathsf{cons}])$ is the *initial algebra* for the functor $FX = 1 + \mathsf{El} \times X$. It is the *fold* (the unique algebra map) determined by the algebra (B, β).

Some good cases (2): Recursive coalgebras A unique solution exists for any B, β also for the equation

$$\begin{tikzcd}
1+\mathsf{El}\times\mathsf{List}\times\mathsf{List} \arrow[d, "1+\mathsf{El}\times f\times f"'] & \mathsf{List} \arrow[l, "\mathsf{qsplit}"'] \arrow[d, "f"] \\
1+\mathsf{El}\times B\times B \arrow[r, "\beta"'] & B
\end{tikzcd}$$

where $\mathsf{qsplit}\,\mathsf{nil} = \mathsf{inl}\,*$ and $\mathsf{qsplit}\,(\mathsf{cons}\,(x, xs)) = \mathsf{inr}\,(x, xs|_{\leq x}, xs|_{>x})$. E.g., for $B = \mathsf{List}$, $\beta = \mathsf{concat}$ (concatenation of the first list, the element and the second list), we get $f = \mathsf{qsort}$ (quicksort).

$(\mathsf{List}, \mathsf{qsplit})$ is <u>not</u> the inverse of the initial algebra of $FX = 1 + \mathsf{El} \times X \times X$ (which is the algebra of binary node-labelled trees), but we still have a unique f for any (B, β).

For this property, $(\mathsf{List}, \mathsf{qsplit})$ is called a *recursive coalgebra* of F. Recursive F-coalgebras form a full subcategory of the category of all F-coalgebras. The inverse of the initial F-algebra is the final recursive F-coalgebra.

While recursiveness is a very useful property of a coalgebra, it is generally difficult to determine whether a given coalgebra is recursive. For more information on recursive coalgebras, see Taylor [8], Capretta et al. [6], Adámek et al. [1].

Some good cases (3): Final coalgebra This equation has a unique solution for any A, α.

$$\begin{tikzcd}
\mathsf{El}\times A \arrow[d, "1+\mathsf{El}\times f"'] & A \arrow[l, "\alpha"'] \arrow[d, "f"] \\
\mathsf{El}\times\mathsf{Str} \arrow[r, "{\langle\mathsf{hd},\mathsf{tl}\rangle}^{-1}"'] & \mathsf{Str}
\end{tikzcd}$$

E.g., for $A = \mathsf{Str}$ (streams), $\alpha = \langle \mathsf{hd}, \mathsf{tl} \circ \mathsf{tl} \rangle$ (the analysis of a stream into its head and the tail of its tail), we get $f = \mathsf{dropeven}$ (the function dropping every even-position element of a given stream).

A unique f exists for any (A, α) because $(\mathsf{Str}, \langle \mathsf{hd}, \mathsf{tl} \rangle)$ is the *final coalgebra* of $FX = \mathsf{El} \times X$. It is the *unfold* (the unique F-coalgebra map) given by the coalgebra (A, α).

Some good cases (4): Corecursive algebras This equation has a unique solution for any A, α:

$$\begin{tikzcd}
\mathsf{El} \times A \times A \arrow[d, "\mathsf{El} \times f \times f"'] & A \arrow[l, "\alpha"'] \arrow[d, "f"] \\
\mathsf{El} \times \mathsf{Str} \times \mathsf{Str} \arrow[r, "\mathsf{smerge}"'] & \mathsf{Str}
\end{tikzcd}$$

Here $\mathsf{hd}\,(\mathsf{smerge}(x, xs_0, xs_1)) = x$ and $\mathsf{tl}\,(\mathsf{smerge}(x, xs_0, xs_1)) = \mathsf{smerge}(\mathsf{hd}\,xs_0, xs_1, \mathsf{tl}\,xs_0)$.

$(\mathsf{Str}, \mathsf{smerge})$ is not the inverse of the final coalgebra of $FX = \mathsf{El} \times X \times X$, but a unique f still exists for any (A, α). We say that $(\mathsf{Str}, \mathsf{smerge})$ is a *corecursive algebra* of F, cf. Capretta et al. [7]. [The inverse of the final F-coalgebra is the initial corecursive F-algebra and thus a special case.] Similarly to recursiveness of a coalgebra, corecursiveness of an algebra is a useful property, but generally difficult to establish.

The equation 1 can of course have a unique solution also in other cases. In particular, it may well happen that neither is (A, α) corecursive nor is (B, β) recursive, but the equation still has exactly one solution.

General case (1): Inductive domain predicate Bove and Capretta [3, 4] put forward the following approach to recursive definitions in type theory (the idea has occurred in different guises in multiple places; it must go back to McCarthy): for a given recursive definition, work out its "domain of definition" and see if it contains the intended domain.

For given (A, α), define a predicate dom on A *inductively* by

$$\frac{a : A \quad (\tilde{F}\,\mathsf{dom})\,(\alpha\, a)}{\mathsf{dom}\, a}$$

(i.e., as the smallest/strongest predicate validating this rule), denoting by $\tilde{F}\,P$ the lifting of a predicate P from A to $F\,A$.

Write $A|_{\mathsf{dom}}$ for the subset of A determined by the predicate dom, the "domain of definedness". It is easily verified that, for any (B, β), there is $f : A|_{\mathsf{dom}} \to B$ uniquely solving

$$\begin{tikzcd}
F(A|_{\mathsf{dom}}) \arrow[d, "Ff"'] & A|_{\mathsf{dom}} \arrow[l, "\alpha|_{\mathsf{dom}}"'] \arrow[d, "f"] \\
FB \arrow[r, "\beta"'] & B
\end{tikzcd}$$

If $\forall a : A.\, \mathsf{dom}\, a$, which is the same as $A|_{\mathsf{dom}} \cong A$, then f is a unique solution of the original equation 1, i.e., the coalgebra (A, α) is recursive.

For $A = \mathsf{List}$, $\alpha = \mathsf{qsplit}$, dom is defined inductively by

$$\frac{}{\mathsf{dom}\,\mathsf{nil}} \qquad \frac{x : \mathsf{El} \quad xs : \mathsf{List} \quad \mathsf{dom}\,(xs|_{\leq x}) \quad \mathsf{dom}\,(xs|_{>x})}{\mathsf{dom}\,(\mathsf{cons}\,(x, xs))}$$

We can prove that $\forall xs : \mathsf{List}.\, \mathsf{dom}\, xs$. Hence $(\mathsf{List}, \mathsf{qsplit})$ is recursive.

If $A|_{\mathsf{dom}} \cong A$, the coalgebra (A,α) is said to be *wellfounded*. Wellfoundedness gives an induction principle on A: For any predicate P on A, we have

$$\frac{a : A \qquad \begin{array}{c} a' : A \quad (\tilde{F}\,P)\,(\alpha\,a') \\ \vdots \\ P\,a' \end{array}}{P\,a}$$

We have seen that wellfoundedness suffices for recursiveness. In fact, it is also necessary. While this equivalence is easy for polynomial functors on the category of sets, it becomes remarkably involved in more general settings, see Taylor [8].

For $FX = 1 + \mathsf{El} \times X \times X$, $A = \mathsf{List}$, $\alpha = \mathsf{qsplit}$, we get this induction principle:

$$\frac{xs : \mathsf{List} \quad P\,\mathsf{nil} \qquad \begin{array}{c} x : \mathsf{El} \quad xs' : \mathsf{List} \quad P\,(xs'|_{\leq x}) \quad P\,(xs'|_{>x}) \\ \vdots \\ P\,(\mathsf{cons}\,(x, xs')) \end{array}}{P\,xs}$$

General case (2): Inductive graph relation The original Bove-Capretta method separates determining the domain of definition of a function from determining its values. Bove [2] showed that this separation can be avoided.

For given (A,α), (B,β), define a relation $\downarrow$ between A, B *inductively* by

$$\frac{a : A \quad bs : F\,B \quad \alpha\,a\,(\tilde{F}\downarrow)\,bs}{a \downarrow \beta\,bs}$$

Further, define a predicate Dom on A by

$$\mathsf{Dom}\,a = \exists b : B.\, a \downarrow b$$

It is straightforward to verify that $\forall a : A, b, b_* : B. a \downarrow b \wedge a \downarrow b_* \to b = b_*$. Moreover, it is also the case that $\forall a : A.\, \mathsf{Dom}\,a \leftrightarrow \mathsf{dom}\,a$. So, Dom does not really depend on the given (B,β)!

From the last equivalence it is immediate that there is $f : A|_{\mathsf{Dom}} \to B$ uniquely solving

$$\begin{tikzcd} F\,(A|_{\mathsf{Dom}}) \arrow[d, "F\,f"'] & A|_{\mathsf{Dom}} \arrow[l, "\alpha|_{\mathsf{Dom}}"'] \arrow[d, "f"] \\ F\,B \arrow[r, "\beta"'] & B \end{tikzcd}$$

And, if $\forall a : A.\, \mathsf{Dom}\,a$, which is the same as $A|_{\mathsf{Dom}} \cong A$, then f is a unique solution of the original equation.

As a matter of fact, recursiveness and wellfoundedness are equivalent exactly because $\forall a : A.\, \mathsf{Dom}\,a \leftrightarrow \mathsf{dom}\,a$.

For $FX = 1 + \mathsf{El} \times X \times X$, $A = \mathsf{List}$, $\alpha = \mathsf{qsplit}$, $B = \mathsf{List}$, $\beta = \mathsf{concat}$, the relation $\downarrow$ is defined inductively by

$$\frac{}{\mathsf{nil} \downarrow \mathsf{nil}} \qquad \frac{x : \mathsf{El} \quad xs : \mathsf{List} \quad xs|_{\leq x} \downarrow ys_0 \quad xs|_{>x} \downarrow ys_1}{\mathsf{cons}\,(x, xs) \downarrow \mathsf{app}\,(ys_0, \mathsf{cons}(x, ys_1))}$$

Inductive domain and graph do not work for non-terminating productive definitions Unfortunately, for our dropeven example,

$$\begin{tikzcd}
\mathsf{El} \times \mathsf{Str} \arrow[d, "1+\mathsf{El}\times\mathsf{dropeven}"'] & \mathsf{Str} \arrow[l, "\langle \mathsf{hd}, \mathsf{tl} \circ \mathsf{tl} \rangle"'] \arrow[d, "\mathsf{dropeven}"] \\
\mathsf{El} \times \mathsf{Str} \arrow[r, "\langle \mathsf{hd}, \mathsf{tl} \rangle^{-1}"'] & \mathsf{Str}
\end{tikzcd}$$

we get $\forall xs : \mathsf{Str}.\, \mathsf{dom}\, xs \equiv \bot$! Now, surely there is a unique function from $0 \to \mathsf{Str}$. But this is uninteresting! We would like to learn that there is a unique function $\mathsf{Str} \to \mathsf{Str}$.

Intuitively, the reason why this equation has a unique solution lies not in how a given argument is consumed but in how the corresponding function value is produced. This is not a terminating but a productive definition.

General case (3): Coinductive bisimilarity relation The concept of the domain of definedness can be dualized [7]. Besides partial solutions that are defined only on a subset of the intended domain, it makes sense to consider "fuzzy" solutions that are defined everywhere but return values in a quotient of the intended codomain. But since the category of sets is not self-dual, the theory dualizes only to a certain extent and various mismatches arise.

For given (B, β), define a relation $\approx$ on B *coinductively* by

$$\frac{b, b_* : B \quad b \approx b_*}{\exists bs, bs_* : FB.\, b = \beta\, bs \wedge b_* = \beta\, bs_* \wedge bs\, (\tilde{F} \approx^*)\, bs_*}$$

(i.e., we take $\approx$ to be the largest/coarsest relation validating this rule).

There need not necessarily be a function f solving the equation

$$\begin{tikzcd}
FA \arrow[d, "Ff"'] & A \arrow[l, "\alpha"'] \arrow[d, "f"] \\
F(B/_{\approx^*}) \arrow[r, "\beta/_{\approx^*}"'] & B/_{\approx^*}
\end{tikzcd}$$

but, if such a function exists, it can easily checked to be unique. (See Capretta et al. [7, Thm. 1].)

If $\forall b, b_* : B.\, b \approx b_* \to b = b_*$, which is the same as $B/_{\approx^*} \cong B$ (where $B/_{\approx^*}$ is the quotient of B by the reflexive-transitive closure of $\approx$), we say that (B, β) is *antifounded*. If (B, β) is antifounded, solutions to equation 1 are the same as solutions to the equation above, and thus unique.

For $FX = \mathsf{El} \times X \times X$, $B = \mathsf{Str}$, $\beta = \mathsf{smerge}$, the relation $\approx$ is defined coinductively by

$$\frac{xs, xs_* : \mathsf{Str} \quad xs \approx xs_*}{\begin{array}{l} \exists x : \mathsf{El}, xs_0, xs_1, xs_{0*}, xs_{1*} : \mathsf{Str}. \\ \quad xs = \mathsf{smerge}(x, xs_0, xs_1) \wedge xs_* = \mathsf{smerge}(x, xs_{0*}, xs_{1*}) \wedge xs_0 \approx xs_{0*} \wedge xs_1 \approx xs_{1*} \end{array}}$$

It turns out that $\forall xs, xs' : \mathsf{Str}.\, xs \approx xs' \to xs = xs'$. Based on this knowledge, we may conclude that solutions are unique. (They do in fact exist as well for this example, but this has to be verified separately.)

Solutions need not exist for antifounded algebras. E.g., for $FX = X$, $B = \mathsf{Nat}$, $\beta = \mathsf{succ}$, we have that (B, β) is antifounded, but for A any set and $\alpha = \mathsf{id}_A$, the equation has the form $f\, a = \mathsf{succ}\,(f\ a)$ and has no solutions.

We have thus seen that antifoundedness of (B,β) does not guarantee that it is corecursive. The converse also fails: not every corecursive algebra (B,β) is antifounded [7, Prop. 5].

However, for an antifounded algebra (B,β), we do get an interesting coinduction principle on B: For any relation R on B, we have

$$\frac{b,b_* : B \quad b\,R\,b_* \quad \begin{array}{c} b',b'_* : B \quad b'\,R\,b'_* \\ \vdots \\ \exists bs',bs'_* : FB.\, b' = \beta\, bs' \wedge b'_* = \beta\, bs'_* \wedge bs'\,(\tilde{F}\,R^*)\,bs'_* \end{array}}{b = b_*}$$

For $FX = \mathsf{El} \times X \times X$, $B = \mathsf{Str}$, $\beta = \mathsf{smerge}$, we get this coinduction principle:

$$\frac{xs,xs_* : \mathsf{Str} \quad xs\,R\,xs_* \quad \begin{array}{c} xs',xs'_* : \mathsf{Str} \quad xs'\,R\,xs'_* \\ \vdots \\ \exists x' : \mathsf{El}, xs'_0, xs'_1, xs'_{0*}, xs'_{1*} : \mathsf{Str}. \\ xs' = \mathsf{smerge}(x', xs'_0, xs'_1) \wedge xs'_* = \mathsf{smerge}(x', xs'_{0*}, xs'_{1x*}) \wedge xs'_0\,R\,xs'_{0*} \wedge xs'_1\,R\,xs'_{1*} \end{array}}{xs = xs_*}$$

General case (4): Coinductive graph relation Could one also dualize the notion of the inductive graph? The answer is positive. Differently from the case of the coinductive concept of bisimilarity, this yields a criterion of unique solvability.

For given (A,α), (B,β), define a relation $\downarrow^\infty$ between A, B *coinductively* by

$$\frac{a : A \quad b : B \quad a \downarrow^\infty b}{\exists bs : FB.\, b = \beta\, bs \wedge \alpha\, a\, (\tilde{F} \downarrow^\infty)\, bs}$$

Define a predicate Dom^∞ on A by

$$\mathsf{Dom}^\infty a = \exists b : B.\, a \downarrow^\infty b$$

Now we can construct $f : A|_{\mathsf{Dom}^\infty} \to B/_{\approx^*}$ that we can prove to uniquely solve

$$\begin{tikzcd} F(A|_{\mathsf{Dom}^\infty}) \arrow[d, "Ff"'] & A|_{\mathsf{Dom}^\infty} \arrow[l, "\alpha|_{\mathsf{Dom}^\infty}"'] \arrow[d, "f"] \\ F(B/_{\approx^*}) \arrow[r, "\beta/_{\approx^*}"'] & B/_{\approx^*} \end{tikzcd}$$

If both $\forall a : A.\, \mathsf{Dom}^\infty a$ and $\forall b,b_* : B.\, b \approx b_* \to b = b_*$, which are the same as $A|_{\mathsf{Dom}^\infty} \cong A$ resp. $B/_{\approx^*} \cong B$, then f uniquely solves also the equation 1. Notice, however, that in this situation we have obtained a unique solution only for the given (A,α): we have <u>not</u> established that (B,β) is corecursive.

To formulate a further condition, we define a relation $\equiv$ on B by

$$b \equiv b_* = \exists a : A.\, a \downarrow^\infty b \wedge a \downarrow^\infty b_*$$

A unique solution to equation 1 also exists if $\forall a : A.\, \mathsf{Dom}^\infty a$ and $\forall b,b_* : B.\, b \equiv b_* \to b = b_*$.

This condition is weaker: while $\forall b,b_* : B.\, b \equiv b_* \to b \approx b_*$, the converse is generally not true.

For $FX = \mathsf{El} \times X \times X$, $B = \mathsf{Str}$, $\beta = \mathsf{smerge}$ and any fixed A, α, the relation $\downarrow^\infty$ is defined coinductively by

$$\frac{a : A \quad xs : \mathsf{Str} \quad a \downarrow^\infty xs}{\exists xs_0, xs_1 : \mathsf{Str}.\, xs = \mathsf{smerge}\,(\mathsf{fst}\,(\alpha\, a), xs_0, xs_1) \wedge \mathsf{fst}\,(\mathsf{snd}\,(\alpha\, a)) \downarrow^\infty xs_0 \wedge \mathsf{snd}\,(\mathsf{snd}\,(\alpha\, a)) \downarrow^\infty xs_1}$$

It turns out that $\forall a : A.\, \mathsf{Dom}^\infty\, a$ no matter what A, α are. So in this case we do have a unique solution f for any A, α, i.e., $(\mathsf{Str}, \mathsf{smerge})$ is corecursive.

Conclusion We have considered two flavors of partiality of a function: a function may be defined only on a subset of the intended domain and the values it returns may be underdetermined.

The Bove-Capretta method in its graph-based version scales meaningfully to equations where unique solvability is not due to termination, but productivity or a combination the two. But instead of one condition to check by ad-hoc means, there are two in the general case.

The theory of corecursion/coinduction is not as clean as that of recursion/induction—in particular, to admit coinduction is not the same as to admit corecursion. We would like like to study the coinductive graph approach further and to find out to what extent it proves useful in actual programming practice. The main pragmatic issue is the same as with Bove and Capretta's method: how to prove the conditions.

Acknowledgments This research was supported by Estonian Science Foundation grant no. 6940 and the ERDF funded Estonian Centre of Excellence in Computer Science, EXCS.

References

[1] A. Adámek, D. Lücke & S. Milius (2007): *Recursive coalgebras of finitary functors*. Theor. Inform. and Appl. 41(4), 447–462. doi:10.1051/ita:2007028

[2] A. Bove (2009): *Another look at function definitions*. In S. Abramsky, M. Mislove & C. Palamidessi, editors: *Proc. of 25th Conf. on Mathematical Foundations of Programming Semantics, MFPS-XXV (Oxford, Apr. 2009)*, *Electron. Notes in Theor. Comput. Sci.* 249, Elsevier, 61–74. doi:10.1016/j.entcs.2009.07.084

[3] A. Bove & V. Capretta (2005): *Modelling general recursion in type theory*. Math. Struct. in Comput. Sci. 15(4), 671–708. doi:10.1017/s0960129505004822

[4] A. Bove & V. Capretta (2008): *A type of partial recursive functions*. In O. Aït Mohamed, C. Muñoz & S. Tahar, editors: *Proc. of 21st Int. Conf. on Theorem Proving in Higher Order Logics TPHOLs 2008 (Montreal, Aug. 2008*, *Lect. Notes in Comput. Sci.* 5170, Springer, 102–117. doi:10.1007/978-3-540-71067-7_12

[5] A. Bove, A. Krauss & M. Sozeau (2011): *Partiality and recursion in interactive theorem provers: an overview*. Manuscript, submitted to *Math. Struct. in Comput. Sci.*.

[6] V. Capretta, T. Uustalu & V. Vene (2006): *Recursive coalgebras from comonads*. Inform. and Comput. 204(4), 437–468. doi:10.1016/j.ic.2005.08.005

[7] V. Capretta, T. Uustalu & V. Vene (2009): *Corecursive algebras: a study of general structured corecursion*. In M. V. M. Oliveira & J. Woodcock, editors: *Revised Selected Papers from 12th Brazilian Symp. on Formal Methods, SBMF 2009 (Gramado, Aug. 2009)*, *Lect. Notes in Comput. Sci.* 5902, Springer, 84–100. doi:10.1007/978-3-642-10452-7_7

[8] P. Taylor (1999): *Practical Foundations of Mathematics*, chapter VI. Cambridge University Press.