Clocks for Functional Programs *

Jörg Endrullis[1], Dimitri Hendriks[1],
Jan Willem Klop[1,2], and Andrew Polonsky[1,3]

[1] VU University Amsterdam, Department of Computer Science
[2] Centrum Wiskunde & Informatica (CWI)
[3] Radboud Universiteit Nijmegen

`j.endrullis@vu.nl` `r.d.a.hendriks@vu.nl`
`j.w.klop@vu.nl` `a.polonsky@vu.nl`

Dedicated to Rinus Plasmeijer for his 61st birthday:
to a clean functional programmer, in friendship and admiration.

1 Introduction

Of the current authors the oldest one remembers with fondness numerous meetings with Rinus from the ancient times of the European Basic Research Actions and from personal tutorials in Nijmegen about λ-terms, term graphs and processes on the one hand, and the practice of functional programming in the `Clean` environment on the other hand.

The youngest author as well remembers with gratitude the AFP 2008 summer school on functional programming that Rinus had helped organize. Taking place in the idyllic Center Parcs, in Boxmeer, this gratifying experience from his PhD years has left behind a number of wonderful memories.

Now that the clock for Rinus himself has arrived at 61 years, we like to offer him the present elaboration of an inherent clock mechanism in functional programs. A clock mechanism that is not only interesting from the perspective of curiosity, but that serves two very concrete goals.

The first goal is to *distinguish* between different functional programs, different in the sense that they are not convertible to each other by some canonical conversion rules, such as β-reduction and the ensuing convertibility. The usual procedure to establish such a discrimination is using the infinite unfolding, known as the Böhm Tree; if their respective Böhm Trees are different, then the programs are also inconvertible in the finite sense. But what if their Böhm Trees are identical? Then the classical Böhm Tree discrimination argument is not applicable. But here our clock method steps in: by means of an annotated version of Böhm Trees we often can observe a difference in the tempo in which the Böhm Trees are generated, and if this tempo is sufficiently different (in a sense to be made

* The research has been partially funded by the Netherlands Organization for Scientific Research (NWO) under grant numbers 612.000.934, 639.021.020, and 612.001.002.

precise), the original λ-terms (or functional programs) are inconvertible in the finite sense.

So the discrimination can often be done on the basis of a difference in clock velocity, but note that we do not mean the clock velocity in the actual computer implementation, but a clock on a much higher level, on the level of the λ-terms that ultimately encode the program.

The second concrete goal is to use the inherent clock phenomenon described below for *optimization* of programs, or rather to measure the extent of such an optimization. To give a quick example: the two simplest fixed point combinators are the one of Church, Y_0, and the one of Turing, Y_1. While they perform similarly, in the sense that $\mathsf{Y}_0 x$ and $\mathsf{Y}_1 x$ both reduce to the infinite iteration of x written as x^ω, the first one delivers its output, the fixed point, in a faster tempo than the second one.

In fact we will not go all the way back to good old λ-calculus to describe the functional programs. First, we will adopt the *simply typed* version, because this conforms much more to actual functional programming practice than pure, untyped λ-terms would.

The second adaptation is that we consider the *extension* with the well-known μ-operator for recursion. This is equivalent with using a fixed point combinator, but it is more direct, and it again conforms more to actual functional programming practice as it can be considered to be tantamount to the letrec operator.

In previous work [EHK10,EHKP12] we have worked out this inherent clock feature in pure λ-terms. As the ticks of the clock, *head reduction steps* were used, that lead one from one node in the Böhm Tree being developed to a successor node. It is a simple observation that contracting internal redexes in a term can only diminish the number of head redexes needed to reach the next node. In other words, reducing a term can only speed-up its internal clock.

In subsequent work [EHKP13], we have internalized this 'external' counting of the head steps, in favour of a τ-operator, like the silent step in process theory. In the present work we do the same, but now with an additional ι-step to count applications of the μ-rules. In fact, we work with *weak head reduction*.

We include some examples suggesting the use of the clock method for simply typed $\lambda\mu$-terms, and thereby for functional programs.

As a historical note, we mention that [NI89] already proposed to use the number of root steps used in evaluating a term in a term rewriting system as a measure of efficiency in comparing terms.

2 A Glossary of Requisites

We will start with a glossary of preliminary notions. For general reference to λ-calculus we refer to [Bar84], for typed versions of λ-calculus to [BDS13,HS08]. For a general reference to term rewriting systems we refer to [BN98,Ter03]. For an introduction to functional programming, see [Hut07,dMJB$^+$]. Rather than repeat in detail much of what these general references offer, we present several of the prerequisites for understanding this paper in the form of a somewhat informal glossary. Some basic familiarity with λ-calculus and term rewriting systems is assumed.

Lambda Calculus The kernel of all calculi figuring in this paper (except for the $\lambda\mu$-calculus introduced later) will be the $\lambda\beta$-calculus with as single reduction rule the β-reduction rule $(\lambda x.M)N \to_\beta M[x:=N]$. Here $x \in \mathcal{X}$, the set of variables. This rule may be applied in a context, a λ-term $C[]$ with a hole, resulting in one-step β-reduction $C[\lambda x.M] \to_\beta C[M[x:=M]]$. The transitive-reflexive closure of $\to_\beta$ is written as $\twoheadrightarrow_\beta$ and the equivalence relation generated by $\to_\beta$, also called β-convertibility, is $=_\beta$. The $\lambda\beta$-calculus has $Ter(\lambda)$ as set of terms. It has the Church–Rosser or confluence property (CR) stating that for $M =_\beta N$ there is a common reduct L such that $M \twoheadrightarrow_\beta L \twoheadleftarrow_\beta N$. A normal form is a term N that does not admit $\to_\beta$-steps. The property SN, strong normalization, stating that there are no infinite reduction sequences $M_0 \to_\beta M_1 \to_\beta \ldots$ does not hold in $\lambda\beta$, and neither does WN, weak normalization, stating that every $M \in Ter(\lambda\beta)$ has (reduces to) a normal form. Both $\neg$SN and $\neg$WN are witnessed by the 'unsolvable' term $\Omega \equiv \omega\omega$ with $\omega \equiv (\lambda x.xx)$ which has a β-loop to itself: $\Omega \to_\beta \Omega$. Here '$\equiv$' denotes syntactic identity, to be distinguished from $=_\beta$.

Fixed Point Combinators An fpc, fixed point combinator, is a term $Y \in Ter(\lambda\beta)$ such that $Yx =_\beta x(Yx)$. The two simplest fpc's are Curry's fpc $\mathsf{Y}_0 \equiv \lambda f.\omega_f\omega_f$ where $\omega_f \equiv \lambda x.f(xx)$, and Turing's fpc $\mathsf{Y}_1 \equiv \theta\theta$ where $\theta \equiv \lambda ab.b(aab)$. Using the term $\delta \equiv \lambda ab.b(ab)$, called the Owl in [Smu85], we have $\mathsf{Y}_0\delta =_\beta \mathsf{Y}_1$.
An fpc Y is called reducing if not only $Yx =_\beta x(Yx)$, but even $Yx \twoheadrightarrow_\beta x(Yx)$. So Y_1 is reducing, but Y_0 is not. If Y is a reducing fpc, also $Y\delta$ is one.
Of interest are also weak fpcs, or wfpcs. These are terms $Z \in Ter(\lambda\beta)$ such that $Zx = x(Z'x)$ where Z' is again a wfpc. (This is a coinductive definition.) So any fpc is a wfpc but not conversely. An example of such a 'proper' wfpc is $A(BAB)$ where $B = \lambda xyz.x(yz)$ and $A \equiv B\omega$ given by Statman, see [EHK12].

Unsolvable Terms are terms that cannot be evaluated to positive information, to a hnf. The ur-example is Ω as above. A $\lambda\beta$-term is unsolvable if it has no hnf, which is the case if it admits an infinite reduction containing infinitely many head reduction steps, i.e., where a head redex R is contracted (reduced); such a redex R occurs in a term $\lambda\boldsymbol{x}.R\boldsymbol{N}$ where $\boldsymbol{x}$ is a vector of variables, and $\boldsymbol{N}$ a vector of terms.

Böhm Trees are the infinite expansions of λ-terms, analogous to expansions in number theory such as $\pi = 3.1415926535\ldots$. For Böhm Trees (BTs) there are two kinds of building blocks; positive information carriers which have the form of a head normal form (hnf) $\lambda\boldsymbol{x}.y\Box\ldots\Box$, where $\boldsymbol{x}, y \in \mathcal{X}$, the set of variables, and $\Box\ldots\Box$ are empty places; and negative information carriers

(no information) of the form $\perp$ or Ω. Definition 13 gives a more precise description suitable for the present setting.
Next to the Böhm Tree semantics there is a slightly more refined semantics, based on Lévy–Longo Trees (LLTs), and a third and finest semantics based on Berarducci Trees (BeTs). See [EHK12].

Term Rewriting Systems This name is most often reserved for first-order rewrite systems, where by definition there is no binding of variables as in the β-reduction rule above, or the μ-reduction rule $\mu x.t \to_\mu t[x := \mu x.t]$. Sometimes the name is used in a generic sense to include all term rewrite systems, first-order but also higher-order; but not e.g. term graph rewrite systems where terms have been generalized to term graphs.

Some Notations As said, $\twoheadrightarrow$ or $\to^*$ denotes the transitive-reflexive closure of a rewrite relation $\to$; $\twoheadrightarrow\!\!\!\!\rightarrow$ always denotes infinitary reduction (of arbitrary ordinal length). For the substitution operation we use the notation $s[x := t]$ to indicate that in term s all free occurrences of x are replaced by the term t.

Types We will be very short about types and refer to the before mentioned standard reference works. Only this: we mostly use (as in [HS08]) the Church style of writing the simple types, as superscripts A of subterms s, so as s^A, rather than the judgments $s : A$. Only occasionally we will have to mention a typing as a judgment.

Infinitary Rewriting A development in term rewriting and λ-calculus which has been elaborated in a relatively late stage, is that of infinitary rewriting, which emerges naturally, and gives a domain where infinite Böhm Trees are at home. The fpc $\mathsf{Y}_1 \equiv \theta\theta$ where $\theta \equiv \lambda ab.b(aab)$ already gives the idea:

$$\mathsf{Y}_1 x \to^2 x(\mathsf{Y}_1 x) \to^2 x(x(\mathsf{Y}_1 x)) \to^2 x^3(\mathsf{Y}_1 x) \twoheadrightarrow x^n(\mathsf{Y}_1 x) \twoheadrightarrow \ldots$$

The natural extension here is to go on and continue rewriting to a limit $x^\omega \equiv x(x(x(\ldots)))$, so that we have $\mathsf{Y}_1 x \twoheadrightarrow\!\!\!\!\rightarrow x^\omega$. Here $\twoheadrightarrow\!\!\!\!\rightarrow$ stands for an infinite reduction, in this case of length ω; we therefore also write $\mathsf{Y}_1 x \to^\omega x^\omega$. In general we may have reductions $s \to^\alpha t$ for every countable ordinal α. Infinitary rewriting requires a limit notion, which is that of ordinary Cauchy convergence with respect to the usual metric distance d. That is to say, $\mathrm{d}(s, t) = 2^{-n}$ if n is the first level where the formation term trees of s and t differ. There is however one extra requirement that is put on top of Cauchy convergence: when approaching a limit ordinal such as ω, $\omega \cdot 2$, ω^2, ϵ_0, $\ldots$, the 'action' has to go down all the way, more precisely, the depth of the contracted redexes has to tend to ∞. Note that this is indeed the case in the example $\mathsf{Y}_1 x \twoheadrightarrow\!\!\!\!\rightarrow x^\omega$ above. But note also that we do not have $\Omega \to^\omega \Omega$, as here the action stays confined at the top, the root. (A trivial subtlety: we do have $\Omega \twoheadrightarrow\!\!\!\!\rightarrow \Omega$, because we have $\twoheadrightarrow\!\!\!\!\rightarrow \supseteq \twoheadrightarrow \supseteq \equiv$: infinitary rewriting comprises finitary rewriting which in turn comprises identity.)
Let us mention that in recent work [EHH+13] a coinductive definition of infinitary rewriting $\twoheadrightarrow\!\!\!\!\rightarrow$ is given that is 'coordinate-free', i.e., avoids all mention

of ordinals and depth of redex contractions, which has the virtue of making the notion of infinitary rewriting much more amenable to an automated treatment and a formalization in theorem provers.

Main Syntactic Properties For finite rewriting as in $\lambda\beta$-calculus, $\lambda\beta\mu$-calculus or their simply typed versions, but also in first-order term rewriting systems (without bound variables) we have some important syntactic properties of the rewrite or reduction relation. These are:

CR the confluence property, $\twoheadleftarrow \cdot \twoheadrightarrow \subseteq \twoheadrightarrow \cdot \twoheadleftarrow$;

PML Parallel Moves Lemma $\leftarrow \cdot \twoheadrightarrow \subseteq \twoheadrightarrow \cdot \twoheadleftarrow$;

WN weak normalization: every term has (reduces to) a normal form;

SN strong normalization: every reduction ends in a normal form when prolonged long enough; otherwise said, there are no infinite reductions;

UN every term has at most one normal form.

The infinitary counterparts of these properties are:

CR^∞ the infinitary confluence property, $\mathrel{\twoheadleftarrow\!\!\!\!\leftarrow} \cdot \mathrel{\rightarrow\!\!\!\!\twoheadrightarrow} \subseteq \mathrel{\rightarrow\!\!\!\!\twoheadrightarrow} \cdot \mathrel{\twoheadleftarrow\!\!\!\!\leftarrow}$;

PML^∞ the infinitary Parallel Moves Lemma, $\leftarrow \cdot \mathrel{\rightarrow\!\!\!\!\twoheadrightarrow} \subseteq \mathrel{\rightarrow\!\!\!\!\twoheadrightarrow} \cdot \mathrel{\twoheadleftarrow\!\!\!\!\leftarrow}$;

WN^∞ every term has a (possibly infinite) normal form;

SN^∞ every reduction sequence, when prolonged long enough, even infinitarily, will strongly converge to a (possibly infinite) normal form; in other words, there are no diverging reductions;

UN^∞ every term has at most one (possibly infinite) normal form. Here 'has' means 'reduces to' ($\mathrel{\rightarrow\!\!\!\!\twoheadrightarrow}$).

Miscellaneous Next to first-order rewrite systems (so without bound variables), and higher-order rewrite systems such as the calculi featuring in this paper which all involve bound variables (by λ or μ), we have a general notion of higher-order rewrite system, that unifies the afore-mentioned rewrite systems. These are the Combinatory Reduction Systems (CRSs), see [Ter03]. All rewrite systems in this paper belong to this general family. An important notion in such systems is 'orthogonality', meaning that the reduction rules do not overlap harmfully; there are no critical pairs (and moreover the rules must be left-linear, no duplicated variables in lefthand-sides of the rules). For such orthogonal CRSs (once called OCRSs in this paper), we have the confluence property CR, and hence also the property UN, unique normal forms. The family of orthogonal CRSs also includes by definition subcalculi, where (e.g.) a typing restriction is adopted. For such subcalculi the CR property also holds.

3 A Cube of Calculi

The $\lambda\beta$-calculus can be considered to be the mother of all term rewriting systems—it is the origin corner of our cube of calculi in Figure 1. But it is not

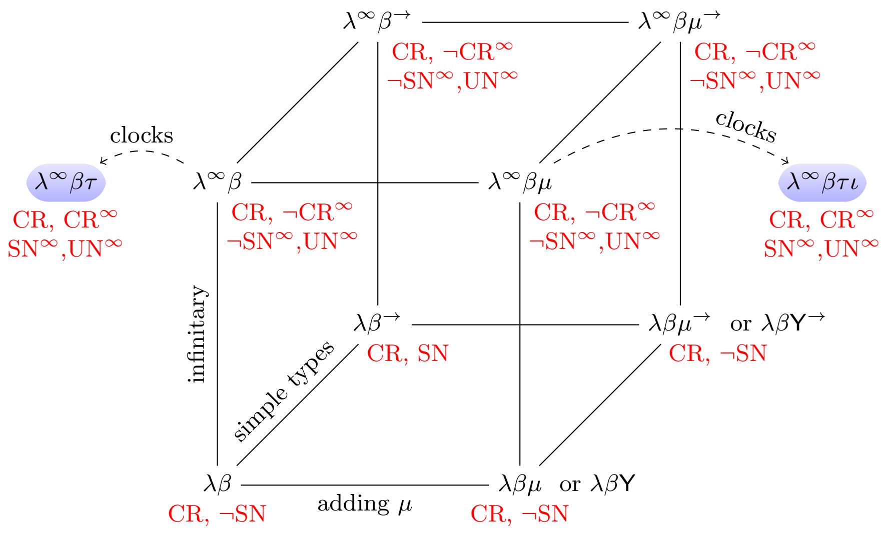
$\lambda^\infty\beta^\to$
CR, ¬CR$^\infty$ ¬SN$^\infty$,UN$^\infty$
$\lambda^\infty\beta\mu^\to$
CR, ¬CR$^\infty$ ¬SN$^\infty$,UN$^\infty$
clocks
$\lambda^\infty\beta\tau$
CR, CR$^\infty$ SN$^\infty$,UN$^\infty$
$\lambda^\infty\beta$
CR, ¬CR$^\infty$ ¬SN$^\infty$,UN$^\infty$
$\lambda^\infty\beta\mu$
CR, ¬CR$^\infty$ ¬SN$^\infty$,UN$^\infty$
clocks
$\lambda^\infty\beta\tau\iota$
CR, CR$^\infty$ SN$^\infty$,UN$^\infty$
infinitary
$\lambda\beta^\to$
CR, SN
$\lambda\beta\mu^\to$ or $\lambda\beta\mathsf{Y}^\to$
CR, ¬SN
simple types
$\lambda\beta$
CR, ¬SN
adding μ
$\lambda\beta\mu$ or $\lambda\beta\mathsf{Y}$
CR, ¬SN

Fig. 1. *Partial ordering of calculi with some main properties.*

the 'main calculus' of this paper, which is the simply typed λ-calculus with the β-rule and extended with the variable binding operator μ and the corresponding reduction rule. We use the notation $\lambda\beta\mu^\to$ for this λ-calculus. Writing $\mathit{Ter}(\lambda)$ for the set of λ-terms we first extend the terms to $\mathit{Ter}(\lambda\mu)$, and next restrict the terms to the typable ones, $\mathit{Ter}(\lambda\mu^\to)$, according to the definition below. Let us note that, for simplicity, in this paper we do not consider the η-reduction rule (except for a brief appearance in Section 7.3).

The interest of the $\lambda\beta\mu^\to$-calculus is that it has a clear relevance for actual functional programming, both by the discipline of simple typing, and by the inclusion of the μ-operator which provides an abstraction over particular implementations of the fixed point combinators, so that a term which is defined by recursive equations can be analyzed without reference to the particular fixed point combinator used to construct the solutions.

The $\lambda\beta\mu^\to$-calculus will be our platform for an extension with a clock mechanism (by τ and ι ticks of a clock) that will enable us to discriminate between many fixed point combinators and the recursive solutions they facilitate. Thus we can discriminate terms with general "periodic" behavior. For example, passing to the $\lambda\beta\mu^\to$-calculus will allow us to discriminate functional programs defined by `letrec` expressions that have the same extensional behavior but cannot be converted from one to another by a finite sequence of syntactic rewrite steps. As hinted at in the introduction, while the output of such programs could well be the same, their computation may often be distinguished by considering the time it takes to produce a value from one recursive step to the next one.

First, let us present the formal definition. The cube of calculi in Figure 1 displays the finitary and infinitary calculi that we define and study.

The following system is obtained by extending the simply typed lambda calculus ($\lambda\beta^{\to}$) with a term constructor $\mu x^A.t$ together with corresponding typing and reduction rules.

Types:

$$\mathbb{T} ::= \alpha \mid \mathbb{T} \to \mathbb{T}$$

Terms:

$$t ::= x \mid t\,t \mid \lambda x^{\mathbb{T}}.t \mid \mu x^{\mathbb{T}}.t$$

Typing:

$$\frac{(x^A) \in \Gamma}{\Gamma \vdash x^A} \qquad \frac{\Gamma \vdash s^{A\to B} \quad \Gamma \vdash t^A}{\Gamma \vdash (st)^B} \qquad \frac{\Gamma, x^A \vdash t^B}{\Gamma \vdash (\lambda x^A.t)^{A\to B}} \qquad \frac{\Gamma, x^A \vdash t^A}{\Gamma \vdash (\mu x^A.t)^A}$$

Reduction:

$$\begin{aligned} \beta : &\quad (\lambda x^A.s)t \to s[x := t] \\ \mu : &\quad \mu x^A.t \to t[x := \mu x^A.t] \end{aligned}$$

Having made a formal acquaintance with the main calculus $\lambda\beta\mu^{\to}$ of our paper, in the cube of Figure 1 located at the corner 110, let us look at the whole cube. In the origin 000 we find the $\lambda\beta$-calculus. From there new calculi are obtained in three directions:

(i) by adding μ and its reduction rule (x-direction);
(ii) by adopting the simple type discipline (y-direction);
(iii) and by making the calculus infinitary by a coinductive reading of all the definitions (z-direction).

Furthermore there are some related calculi outside of this cube. The result is a family of a dozen related λ-calculi as in Table 1.

Remark 1. (i) One could also consider the *untyped* $\lambda\beta\mu$-calculus as the main calculus of our exposition, but we prefer the typed version because it admits natural intuitive interpretation in terms of Scott domains [Plo77], and rules out pathological terms such as $\mu x.xx$, which has Böhm Tree

$$@(@(@..)(@..))(@..)$$

Its term tree is depicted in Figure 2.

position	notation	name
000	$\lambda\beta$	$\lambda\beta$-calculus
001	$\lambda^{\infty}\beta$	infinitary $\lambda\beta$-calculus
010	$\lambda\beta^{\rightarrow}$	simply typed $\lambda\beta$-calculus
011	$\lambda^{\infty}\beta^{\rightarrow}$	infinitary simply typed $\lambda\beta$-calculus
100	$\lambda\beta\mu$	$\lambda\beta\mu$-calculus
101	$\lambda^{\infty}\beta\mu$	infinitary $\lambda\beta\mu$-calculus
110	$\lambda\beta\mu^{\rightarrow}$	simply typed $\lambda\beta\mu$-calculus
111	$\lambda^{\infty}\beta\mu^{\rightarrow}$	infinitary simply typed $\lambda\beta\mu$-calculus
	$\lambda\beta\mathsf{Y}$	$\lambda\beta\mathsf{Y}$-calculus
	$\lambda\beta\mathsf{Y}^{\rightarrow}$	simply typed $\lambda\beta\mathsf{Y}$-calculus
	$\lambda^{\infty}\beta\tau^{(\rightarrow)}$	(simply typed) clocked $\lambda\beta$-calculus
	$\lambda^{\infty}\beta\mu\tau\iota^{(\rightarrow)}$	(simply typed) clocked $\lambda\beta\mu$-calculus
	$\lambda\mu$	$\lambda\mu$-calculus

Table 1. *Family of λ-calculi.*

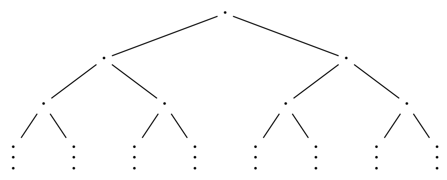

Fig. 2. *Term tree of the Böhm Tree of $\mu x.xx$.*

(ii) The simply typed $\lambda\beta\mu^{\rightarrow}$-calculus is very interesting, as it harmoniously combines some seemingly opposite features. On the one hand, the presence of the simple type discipline seems to forbid infinite reductions, as it does in the sub-calculus $\lambda\beta^{\rightarrow}$, the simply typed $\lambda\beta$-calculus. However, then we also loose fixed point combinators (fpc's), as these all have by definition an infinite reduction. Now this loss is cured by reinstating fpc's by virtue of the μ-operator and the corresponding reduction rule. This simultaneous restriction and extension is still harmonious, in that it has the confluence property (CR). This is so because $\lambda\beta\mu^{\rightarrow}$ is a sub-calculus of an orthogonal CRS, namely the $\lambda\beta\mu$-calculus.

(iii) It is a rewarding exercise to check where the usual SN-proofs for simply typed lambda-calculus fail in the presence of the μ-operator. For the proof using multisets of degrees of redexes, the reason is that created redexes do not have a degree which is less complicated. What is the reason for failure of the other main type of SN proof via computability?

(iv) Another interesting aspect of this restricted-extended calculus, and some of its related calculi, is that its meta-theory hovers on the brink of decidability.

The related calculus $\lambda\beta\mathsf{Y}$ has undecidable convertibility, but decidable unsolvability and normalizability [Sta02]. Presumably the same holds for the present calculus.

4 Variations of the Main Calculus

In this section we will describe some variations of the main calculus $\lambda\beta\mu^{\rightarrow}$, three finite calculi, and one infinitary extension. The three finite versions are well-known in practice; the infinite extension is less well-known, but provides a firm foundation for functional programming.

4.1 The $\boldsymbol{\lambda\beta\mathsf{Y}}$ Variant

Instead of the μ-constructor, we could instead assume the existence of a family of terms

$$\mathsf{Y}_A^{(A\to A)\to A}$$

together with the reduction rule

$$\mathsf{Y}: \qquad \mathsf{Y}_A f \to f(\mathsf{Y}_A f)$$

This system, when extended with a native type of natural numbers, is a Turing-complete programming language for functionals of higher types. It was introduced in 1966 by Platek [Pla84] in order to define higher-order computability in an "index-free" manner. Plotkin [Plo77] extensively studies the semantics of this calculus, culminating in the *full abstraction problem* for PCF. Bezem discusses $\lambda\beta\mathsf{Y}$ in [BDS13, Ch. 5].

Without the type of natural numbers, Irina Bercovici [Ber85] shows that having a head normal form is decidable. Similarly, normalization and compactness (Böhm Tree finiteness) are decidable. Despite these results, Statman [Sta02] shows that the full word problem (convertibility with respect to the reduction rules β and Y) remains undecidable.

However, restricting to the lowest type level, and admitting only Y's of type $(0 \to 0) \to 0$, the word problem is solvable. Statman employs in the short proof of this fact an auxiliary reduction which is just our μ-reduction:

$$\mathsf{Y}(\lambda x.s) \to s[x := \mathsf{Y}(\lambda x.s)]$$

Our main calculus $\lambda\beta\mu^{\rightarrow}$ has an interesting sub-calculus, namely the one consisting of the fragment of only μ-binders, variables and applications. In our current notation it can be called $\lambda\mu$. So there is no β-reduction, only μ-reduction. Now one can ask whether this calculus has a solvable word problem, or in other words, whether its convertibility relation is decidable. Indeed this is the case. There are two sources for a proof of this fact: first, it is a corollary of Statman's result [Sta02] mentioned above, and second, it was also proved for the untyped setting in [EGKvO11].

For the rest of the paper, we will stick with the μ-constructor formulation of the fixed point lambda calculus, which is our main calculus $\lambda\beta\mu^{\rightarrow}$.

4.2 The `letrec` Variant

Another system of equal expressive power is that obtained by postulating the existence of solutions to arbitrary systems of equations of the form

$$\begin{aligned} x_1 &= t_1[\boldsymbol{x}] \\ &\vdots \\ x_n &= t_n[\boldsymbol{x}] \end{aligned}$$

That is, instead of formally adding a unary fixed point constructor, the language provides the ability to solve multiple fixed point equations *simultaneously.*

This idea is implemented by the `letrec` construction commonly found in functional programming languages such as `Clean` [dMJB+]. Its syntax is given as follows:

$$t ::= x \mid t\,t \mid \lambda x^A.t \mid \texttt{let}\ x_1{:=}t, \ldots, x_n{:=}t\ \texttt{in}\ t$$

For convenience, we will write ($\texttt{let}\ \boldsymbol{x} := \boldsymbol{t}\ \texttt{in}\ u$) in place of

$$\texttt{let}\ x_1{:=}t_1, \ldots, x_n{:=}t_n\ \texttt{in}\ u$$

Similarly, in the typing rule below, we will write $\boldsymbol{x}^{\boldsymbol{A}}$ in place of

$$x_1^{A_1}, \ldots, x_n^{A_n}$$

The typing rule is

$$\frac{\Gamma, \boldsymbol{x}^{\boldsymbol{A}} \vdash t_1^{A_1} \qquad \cdots \qquad \Gamma, \boldsymbol{x}^{\boldsymbol{A}} \vdash t_n^{A_n} \qquad \Gamma, \boldsymbol{x}^{\boldsymbol{A}} \vdash u^B}{\Gamma \vdash (\texttt{let}\ \boldsymbol{x} := \boldsymbol{t}\ \texttt{in}\ u)^B}$$

The computation of ($\texttt{let}\ \boldsymbol{x} := \boldsymbol{t}\ \texttt{in}\ u$) returns u in which occurrences of x_i get replaced by t_i, possibly creating new occurrences, which can subsequently be replaced again, and so on. As a rewrite rule, this can be formalized as

$$\texttt{let}\ \boldsymbol{x} := \boldsymbol{t}\ \texttt{in}\ u \quad \to \quad u[x_i := (\texttt{let}\ \boldsymbol{x} := \boldsymbol{t}\ \texttt{in}\ t_i)]_{i=1..n}$$

For our purposes, this rule is much less convenient to work with than that of the μ-constructor, hence we will stick with the original formulation. The two systems are mutually interpretable, although there are some subtleties related to the possibility of "horizontal sharing" (as it was called in [AK95]) in the `letrec` system which we will not investigate here. In any case, the `letrec` syntax is precisely what we find in `Clean`-like functional languages.

4.3 Simultaneous Fixed Point Solutions Via The μ-Operator

Clearly, every $\lambda\mu$-term can be captured within the previous syntax: the expression $\mu x^A.t$ is represented by the single-variable expression $\texttt{let}\ x := t\ \texttt{in}\ x$.

Going the other way, we can represent any term M defined by a simultaneous recursion system ($\texttt{let}\ \boldsymbol{x} := \boldsymbol{t}(\boldsymbol{x})\ \texttt{in}\ u$) by a cascade of μ-expressions.

We show this by an example. Suppose we are a given a `letrec` expression

$$\begin{aligned} M = \texttt{let}\ & x_1 = t_1(x_1, x_2, x_3) \\ & x_2 = t_2(x_1, x_2, x_3) \\ & x_3 = t_3(x_1, x_2, x_3) \\ \texttt{in}\ & \\ & u(x_1, x_2, x_3) \end{aligned}$$

Define the terms

$$\begin{aligned} f_3(x_1, x_2) &= \mu x_3.t_3(x_1, x_2, x_3) \\ f_2(x_1) &= \mu x_2.t_2(x_1, x_2, f_3(x_1, x_2)) \\ f_1 &= \mu x_1.t_1(x_1, f_2(x_1), f_3(x_1, f_2(x_1))) \end{aligned}$$

Finally, put

$$M^\mu = u(f_1, f_2(f_1), f_3(f_1, f_2(f_1)))$$

and it can be easily seen that M^μ the same extensional behavior as M (i.e., M^μ is bisimilar to M).

Convention 2 Using the previous technique of solving simultaneous recursive systems using the μ-operator, we will sometimes write

$$\mu \boldsymbol{x}^{\boldsymbol{A}}.\boldsymbol{t}(x_1, \ldots, x_n)$$

for the corresponding μ-term solving the system given in the matrix. Here we take as given that $t_i(x_1^{A_1}, \ldots, x_n^{A_n})$ has type A_i.

Example 3. Here are some examples of $\lambda\mu^{\rightarrow}$-terms.

(i) The simplest meaningful example is the definition of a fixed point combinator by way of the μ-notation. Put

$$Y = \lambda f^{A \to A}.\mu y^A.fy$$

Then $Yf =_{\beta\mu} f(Yf)$. We have the head reduction

$$Yf \to \mu y.fy \to f(\mu y.fy) \to f(f(\mu y.fy)) \to \cdots \to f^n(\mu y.fy) \to \cdots$$

(Here and throughout, we spare annotation of the types of bound variables when they are unambiguously determined by the immediate context.)

(ii) Applying the above term to the identity yields the canonical unsolvable $\lambda\mu$-term for any type A:

$$\perp_A = \mu x^A.x$$

which has the head reduction

$$\perp \to \perp \to \cdots$$

See [EGKvO11] for a characterization of all unsolvables arising in the $\lambda\mu$-calculus, e.g., $\mu xyz.x$ and $\mu xyz.y$ are examples.

4.4 Infinitary Calculi

It is interesting to extend $\lambda\beta\mu^{\to}$ to include infinitary rewriting. We call the resulting infinitary simply typed calculus: $\lambda^{\infty}\beta\mu^{\to}$. Not only the calculus itself, but also its definition method is an interesting application of the recently developed method to define infinitary rewriting, both the terms and the reductions, using coinduction and coalgebraic techniques [CC96,EP13,EHH+13]. In the present case the new elements are the μ-construct and the simple types. Both can be lifted to the infinite setting by a straightforward coinductive reading of the defining clauses. Intuitively, the benefit is that in this way the somewhat coinductive flavour of the typing rule for μ-terms is elucidated. It follows straightforward by a consideration of the infinite normal form of the μ-term, and its obvious simple typing. Figure 3 contains an example.

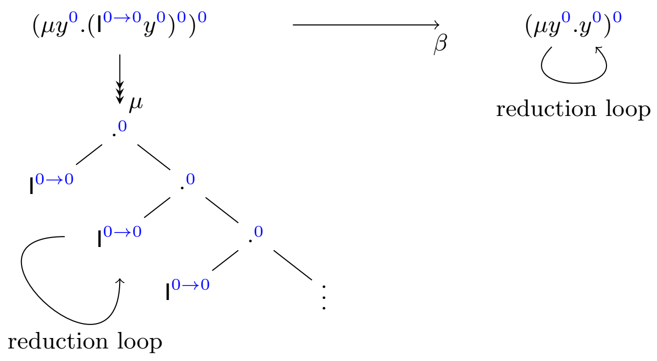

Fig. 3. *Failure of infinitary confluence (CR$^{\infty}$) in the main calculus $\lambda^{\infty}\beta\mu^{\to}$.*

How about the fundamental theorems for $\lambda^{\infty}\beta\mu^{\to}$? We have the failure of PML$^{\infty}$, the infinitary parallel moves lemma, as the following counterexample witnesses:

$$\mu y.\,\mathsf{I}y \twoheadrightarrow\!\!\!\!\!\rightarrow_{\mu} \mathsf{I}^{\omega} \qquad\qquad \text{and also} \qquad\qquad \mu y.\,\mathsf{I}y \to_{\beta} \mu y.\,y$$

Both reducts can only reduce to themselves, in one step. Hence $\neg$PML$^{\infty}$, and therefore also $\neg$CR$^{\infty}$. (See Figure 3.)

Note that the looping terms in Figure 3 are unsolvable. This observation suggest to restore infinitary confluence (CR$^{\infty}$) by quotienting out the unsolvable terms, see [EHK12]. In spite of the failure of CR$^{\infty}$, UN$^{\infty}$ holds, by an appeal on a theorem of Ketema and Simonsen [KS09], stating UN$^{\infty}$ for all infinitary OCRSs; here we have a substructure of such an iOCRS, which by its closure properties admits the same proof of UN$^{\infty}$.

Remark 4. Pure μ-terms $\mu x_1, \dots, x_n.x_i$ are unsolvable. The μ-reductions between theses terms constitute an interesting reduction graph, see [EGKvO11]. In particular, the terms $\mu x_1, \dots, x_n.x_1$ are looping terms. All these μ-unsolvables reduce to $\mu x.x$.

Question 5. Looping terms (admitting a one-step reduction cycle) are interesting as they constitute the difference between the canonical notion of convergence in infinitary rewriting, namely strong convergence (see Glossary), and mere Cauchy convergence:

(i) For the finite $\lambda\beta$-calculus the looping terms are easily classified: they are of the form $C[\Omega]$ for some context $C[]$. For the infinitary λ-calculus $\lambda^\infty\beta$ the full characterization of looping terms was given by Endrullis and Polonsky in [EP13]. Two questions arise at this point:
   (a) What are the looping terms in the main calculus $\lambda^\infty\beta\mu^{\to}$?
   (b) And without typing, so in $\lambda^\infty\beta\mu$?

(ii) For $\lambda\beta\mu$ the question is easy, using the remark above and item (i).

5 Adding Clocks

In this section we prove the main results, the clock theorems, of this paper. We introduce clocked Böhm Trees [EHK10,EHKP12,EHKP13] for the $\lambda\beta\mu$-calculus. For this purpose, we extend the $\lambda\beta\mu^{\to}$-calculus with unary constructors τ and ι that are witnesses of β-steps and μ-steps, respectively. The resulting calculus is orthogonal and infinitary normalizing. The unique infinitary normal forms are Böhm Trees enriched with τ and ι providing information on the speed at which the tree was formed (the number of steps needed to head normalize the corresponding subterm).

Definition 6. The set $\mathit{Ter}^\infty(\lambda\mu\tau\iota)$ of (finite and infinite) terms of the clocked $\lambda\mu$-calculus is coinductively defined[4] by the following grammar

$$M ::=^{\text{co}} x \mid MM \mid \lambda x.M \mid \mu x.M \mid \tau(M) \mid \iota(M) \qquad (x \in \mathcal{X})$$

The set of contexts is inductively defined by

$$C ::= \Box \mid \lambda x.C \mid CM \mid MC \mid \tau(C) \qquad (x \in \mathcal{X}, M \in \mathit{Ter}^\infty(\lambda\mu\tau\iota))$$

Definition 7. The rewrite rules of the clocked $\lambda\mu$-calculus are:

$$\begin{array}{llcll} \beta: & (\lambda x.t)s \to \tau(t[x := s]) & \qquad & \tau: & \tau(x)y \to \tau(xy) \\ \mu: & \mu x.t \to \iota(t[x := \mu x.t]) & & \iota: & \iota(x)y \to \iota(xy) \end{array}$$

The rewrite relation $\to_{\text{🕓}}$ is defined as the closure under contexts of these rules.

[4] This means that $\mathit{Ter}^\infty(\lambda\mu\tau\iota)$ is defined as the greatest fixed point of the underlying set functor.

The shift rules for τ and ι (on the right) are adopted for a better correspondence between the unclocked and the clocked version of the calculus. Without the shift rules, we could not lift reduction from the unclocked to the clocked calculus since the τ or ι may be in the way of a β-redex. For example, the unclocked reduction $\mathsf{II}x \to_\beta \mathsf{I}x \to x$ yields in the clocked calculus $\mathsf{II}x \to_\beta \tau(\mathsf{I})x \to_\tau \tau(\mathsf{I}x) \to_\beta \tau(\tau(x))$ where the shift step is needed to reveal the β-redex.

In this section we can do without the simple type discipline but it would be easy to adopt it. In that case we assume the trivial typing rules for τ and ι, that is to say: a τ-term has the type of its argument, and likewise for ι.

We write $\to$ for the usual (non-clocked) $\lambda\beta\mu$-rewrite relation. We write $\tau^n(t)$ for the term $\tau(\tau(\cdots\tau(t)))$ with n τ's, and likewise for $\iota^n(t)$.

Example 8. Consider $\mu x.x$. We have the reduction

$$\mu x.x \to_{\circledcirc} \iota(\mu x.x) \to_{\circledcirc} \iota(\iota(\mu x.x)) \to_{\circledcirc} \cdots \twoheadrightarrow_{\circledcirc} \iota^\omega$$

An infinite stack of τ's and ι's in the normal form indicates that the corresponding position in the term could not be evaluated to a weak head normal form. In the Böhm Tree such unsolvable subterms are replaced by $\perp$ or Ω.

The motivation for choosing different witnesses for β- and μ-steps is to extract more information from the reduction to the normal form. By distinguishing between τ's and ι's, we can extract information about the working of unsolvables. For example, let $\omega \equiv \lambda x.xx$, then we have:

$$\omega\omega \to_{\circledcirc} \tau(\omega\omega) \to_{\circledcirc} \tau(\tau(\omega\omega)) \to_{\circledcirc} \cdots \twoheadrightarrow_{\circledcirc} \tau^\omega$$

Note that τ^ω is a different form of undefined than ι^ω. We can also have infinite towers of alternating τ's and ι's as illustrated by the reduction:

$$\mu x.\mathsf{I}x \to_{\circledcirc} \mu x.\tau(x) \to_{\circledcirc} \iota(\tau(\mu x.\tau(x))) \to_{\circledcirc} \iota(\tau(\iota(\tau(\mu x.\tau(x))))) \to_{\circledcirc} \ldots \twoheadrightarrow_{\circledcirc} (\iota\tau)^\omega$$

This example has lead to failure of CR$^\infty$ in the $\lambda\beta\mu^{\to}$-calculus as shown in Figure 3. In the clocked setting, the infinitary confluence is restored as shown in Figure 4. For the elementary diagrams involved we refer to Figure 5.

Example 9. Let us have a look at the fixed point combinators of Curry and Turing. In the $\lambda\beta\mu^{\to}$-calculus these fpc's can be rendered as follows.

(i) Curry's fpc Y_0 corresponds to $\lambda f.\mu x.fx$; *par abus de langage* we also use Y_0 for the latter term. Then we have

$$\begin{aligned}
\mathsf{Y}_0 f &\equiv (\lambda f.\mu x.fx)f \\
&\to_{\circledcirc} \tau(\mu x.fx) \\
&\to_{\circledcirc} \tau(\iota(f\mu x.fx)) \\
&\to_{\circledcirc} \tau(\iota(f(\iota(f\mu x.fx)))) \\
&\to_{\circledcirc} \ldots \twoheadrightarrow_{\circledcirc} \tau((\iota f)^\omega) \quad (= \tau(\iota(f(\iota(f\ldots)))))
\end{aligned}$$

$$\begin{tikzcd}
\mu y.\mathsf{I}y \arrow[rrrrr, "\beta"] \arrow[dr, "\mu"] & & & & & \mu y.\tau(y) \arrow[dl, "\mu"'] \\
& \iota(\mathsf{I}\mu y.\mathsf{I}y) \arrow[dr, "\beta"] & & & \iota(\tau(\mu y.\tau(y))) \arrow[ddl, two heads, "\mu"'] & \\
& & \iota(\tau(\mu y.\mathsf{I}y)) \arrow[dr, two heads, "\beta\mu"] & & & \\
& & & (\iota\tau)^{\omega} & &
\end{tikzcd}$$

Fig. 4. *Restoring of infinitary confluence with clocks (compare with Figure 3).*

$$\begin{tikzcd}
\mu y.\mathsf{I}y \arrow[r, "\mu"'] \arrow[dd, "\beta"] & \mathsf{I}(\mu y.\mathsf{I}y) \arrow[d, "\beta"] \\
& {} \arrow[d, "\beta"] \\
\mu y.y \arrow[r, dashed, no head] & \mu y.y
\end{tikzcd}
\qquad
\begin{tikzcd}
\mu y.\mathsf{I}y \arrow[r, "\mu"'] \arrow[dd, "\beta"] & \iota(\mathsf{I}(\mu y.\mathsf{I}y)) \arrow[d, "\beta"] \\
& \iota(\tau(\mu y.\mathsf{I}y)) \arrow[d, "\beta"] \\
\mu y.\tau(y) \arrow[r, "\mu"'] & \iota(\tau(\mu y.\tau(y)))
\end{tikzcd}$$

Fig. 5. *Elementary diagrams: unclocked (left) and clocked (right).*

(ii) Turing's fpc Y_1 corresponds to $\mu x.\lambda f.f(xf)$. Again we also use Y_1 to denote this μ-term. Then we have

$$\begin{aligned}
\mathsf{Y}_1 f &\equiv (\mu x.\lambda f.f(xf))f \\
&\to_{\circlearrowleft} (\iota(\lambda f.f(\mathsf{Y}_1 f)))f \\
&\to_{\circlearrowleft} \iota((\lambda f.f(\mathsf{Y}_1 f))f) \\
&\to_{\circlearrowleft} \iota(\tau(f(\mathsf{Y}_1 f))) \\
&\to_{\circlearrowleft} \ldots \twoheadrightarrow\!\!\!\!\to_{\circlearrowleft} (\iota\tau f)^{\omega} \quad (= \iota(\tau(f(\iota(\tau(f \ldots))))))
\end{aligned}$$

The term Y_0 is more efficient than Y_1 in the sense that between the f's in the infinitary normal forms of Y_0 and Y_1, we have ι in contrast with $\iota\tau$, respectively.

Remark 10. The clocked $\lambda\mu$-calculus can be extended to *atomic clocks*, as they are called in [EHK10,EHKP13], as follows:

$$\begin{aligned}
\beta &: \quad (\lambda x.t)s \to \tau_{\varepsilon}(t[x := s]) & \tau &: \quad \tau_p(x)y \to \tau_{\mathrm{L}p}(xy) \\
\mu &: \quad \mu x.t \to \iota_{\varepsilon}(t[x := \mu x.t]) & \iota &: \quad \iota_p(x)y \to \iota_{\mathrm{L}p}(xy)
\end{aligned}$$

Positions are defined as words over the alphabet $\{\lambda, \mathrm{L}, \mathrm{R}\}$ in the obvious way; the letter L stands for 'left'; ε denotes the empty word. Then the symbols τ_p and μ_p witness not only the type of the rewrite step but also its (relative) position p. In order to keep the presentation simple, we stick to the non-atomic clocks.

Lemma 11. *The rewrite relation* $\to_{\circlearrowleft}$ *has the properties* UN^{∞}, SN^{∞} *and* CR^{∞}.

Proof. Observe that any contraction of a root redex will introduce a τ or ι at the root, hence every term admits at most one root step. We get SN^{∞} by the non-existence of root-active terms [KdV05]. Finally, UN^{∞} follows from orthogonality of the rules, see [KS09] and CR^{∞} immediately follows from UN^{∞} and SN^{∞}.

For terms M, we use $\mathrm{nf}_{\circledcirc}(M)$ to denote the unique infinitary normal forms of M with respect to $\to_{\circledcirc}$. We note that $\mathrm{nf}_{\circledcirc}(M)$ corresponds to the Lévy–Longo Tree [EHK12] of M enriched with symbols τ and ι that provide information about the speed in which this tree has been developed.

Definition 12. We define *τ, ι-removal* $\to_{\mathrm{acc}} \subseteq \mathit{Ter}^{\infty}(\lambda\mu\tau\iota)^2$ as the closure under contexts of the rules

$$\tau(M) \to M \qquad\qquad \iota(M) \to M$$

and use $=_{\mathrm{acc}}$ to denote the equivalence closure of $\to_{\mathrm{acc}}$ (the subscript "acc" abbreviates "acceleration"). For $M, N \in \mathit{Ter}^{\infty}(\lambda\mu)$, we define

(i) $M \succeq_{\circledcirc} N$, *$M$ is globally improved by N* iff $\mathrm{nf}_{\circledcirc}(M) \mathrel{\rightarrow\!\!\!\!\twoheadrightarrow}_{\mathrm{acc}} \mathrm{nf}_{\circledcirc}(N)$;
(ii) $M =_{\circledcirc\exists} N$, *$M$ eventually matches N* iff $\mathrm{nf}_{\circledcirc}(M) =_{\mathrm{acc}} \mathrm{nf}_{\circledcirc}(N)$.

So global improving means that we may drop everywhere in the normal form of M occurrences of τ and ι, even in infinitely many places; while eventual matching means that we may drop these symbols in finitely many places only, so that there is almost everywhere a precise match.

Definition 13. A *head context* is a context of the form $D[\Box N_1 \ldots N_m]$ where D is built from $\lambda x.\Box$, $\tau(\Box)$ and $\iota(\Box)$. A *head reduction step* $\to_h$ is a step in a head context (the position of the step is the position of the hole).

A *head normal form (hnf)* is a λ-term of the form $C[y]$ where C is a head context and $y \in \mathcal{X}$. A *weak head normal form (whnf)* is an hnf or an abstraction, that is, a whnf is a term of the form $xM_1 \ldots M_m$ or $\lambda x.M$. A term *has a (weak) hnf* if it reduces to one.

The following proposition states that clocks are accelerated under reduction:

Proposition 14. *If $M \twoheadrightarrow N$, then N improves M globally, i.e.,* $\mathrm{nf}_{\circledcirc}(M) \mathrel{\rightarrow\!\!\!\!\twoheadrightarrow}_{\mathrm{acc}} \mathrm{nf}_{\circledcirc}(N)$.

Proof. We reduce terms to their unique infinite normal form in a top-down fashion. A position in a term t is *(weakly) stable* if it is not (strictly) contained in a subterm $t' \equiv t'' N_1 \ldots N_m$ of t for which t'' is a redex. Observe that stable symbols (i.e., symbols at stable positions) cannot be touched by any reduction. A *top-redex* in a term t is a redex occurrence ρ whose position is weakly stable. Note that top-redexes stay top when other redexes are contracted. Fair contraction of top-redexes guarantees to reach the infinitary normal form in $\leq \omega$ steps.

By induction on the length of the reduction $M \twoheadrightarrow N$ it suffices to consider a single rewrite step $M \to N$. The step $\to$ can be modeled by a step $\to_{\circledcirc}$ with the only difference that the step $\to_{\text{⏲}}$ creates an additional symbol $\xi \in \{\tau, \iota\}$. Thus $M \to_{\text{⏲}} N'$ with $N' \to_{\text{acc}} N$ by dropping the symbol ξ. We trace the residuals of ξ over reductions with respect to $\to_{\text{⏲}}$. For this purpose we employ the standard notion of tracing [Ter03,BKdV00] except for the rules

$$\tau : \quad \tau(x)y \to \tau(xy) \qquad\qquad \iota : \quad \iota(x)y \to \iota(xy)$$

where we consider the τ and ι displayed in the right-hand sides to be residuals of the τ and ι displayed in the left-hand sides, respectively.

Consider a rewrite sequence $N' \equiv N'_1 \to_{\text{⏲}} N'_2 \to_{\text{⏲}} N'_3 \to_{\text{⏲}} \dots$ of length $\leq \omega$ to infinitary normal form $\mathrm{nf}(N')$ contracting only top-redexes. By uniqueness of normal forms (Lemma 11) we have $\mathrm{nf}(N') \equiv \mathrm{nf}(M)$. For $n = 1, 2, \dots$, we define N_i as the result of dropping all residuals of ξ from N'_i (that is, contracting all residuals of ξ with $\mathrel{\twoheadrightarrow\mkern-14mu\twoheadrightarrow}_{\text{acc}}$). The results N_i of the dropping are well-defined since the residuals of ξ are finitely nested (every step $\to_{\text{⏲}}$ can at most double the nesting depth). We then have a rewrite sequence $N \equiv N_1 \to_{\text{⏲}} N_2 \to_{\text{⏲}} \dots$ with limit $\mathrm{nf}(N)$. We have $\mathrm{nf}(N') \mathrel{\twoheadrightarrow\mkern-14mu\twoheadrightarrow}_{\text{acc}} \mathrm{nf}(N)$ as the limit of the reductions $N'_i \mathrel{\twoheadrightarrow\mkern-14mu\twoheadrightarrow}_{\text{acc}} N_i$ for $i \to \infty$.

This immediately yields the following discrimination method:

Theorem 15 (First Discrimination Criterion). *If N cannot be improved globally by any reduct of M, then $M \neq_{\beta\mu} N$.*

Proof. If $M =_{\beta\mu} N$ then by confluence these terms have a common reduct. By Proposition 14 this common reduct globally improves M.

We now define a class of 'simple' terms for which the clock is invariant under reduction (changes only in finitely many positions). The idea is that in reductions of simple terms there are no duplications of redexes. In fact, we only need to require this for the top-down reduction to Lévy–Longo Tree normal form. The following definition makes this precise:

Definition 16. [Simple terms] A redex $(\lambda x.M)N$ is called:

(i) *linear* if x has at most one occurrence in M;
(ii) *call-by-value* if N is a normal form; and
(iii) *simple* if it is linear or call-by-value.

A redex $\mu x.M$ is called *simple* if M is in normal form.

The set of *simple terms* is coinductively defined as follows (that is, the largest set such that the following conditions holds): A term M is *simple* if

(a) M is not in whnf, $M \to_h M'$ contracting a simple redex and M' is simple,
(b) $M \equiv \lambda x.M'$ with M' a simple term, or
(c) $M \equiv yM_1 \dots M_m$ with $M_1, \dots, M_m$ simple terms.

In contrast to previous work [EHKP13] this definition of 'simple' also considers reduction steps inside of unsolvables. While previously every unsolvable had the infinite normal form τ^ω, the clocked $\lambda\mu$-calculus allows to extract information from unsolvables as they are mapped to infinite towers consisting of τ's and ι's, see Example 8.

Example 17. Let us consider two unsolvables:

(i) The term $\Omega \equiv \omega\omega$ reduces in one step to Ω without duplicating a redex (the term ω does not contain a redex). Thus the terms $\Omega \equiv \omega\omega$ is simple.

(ii) The term $\mu x.\mathsf{I}x$ is not simple, but can be simplified, that is, reduced to a simple term. The term itself is not simple since the reduction step $\mu x.\mathsf{I}x \to \mathsf{I}\mu x.\mathsf{I}x$ duplicates the redex I. However, a reduction step $\mu x.\mathsf{I}x \to \mu x.x$ yields a simple term $\mu x.x$. Note that for discriminating terms M, N it is always sufficient to convertible terms $M' = M$ and $N' = N$.

Proposition 18. *Let N be a reduct of a simple term M. Then N eventually matches M (i.e., $\mathrm{nf}_{\circledast}(M) =_{\tau} \mathrm{nf}_{\circledast}(N)$).*

The following is a reformulation of [EHK10, Corollary 32] for Lévy–Longo Trees:

Corollary 19 (Second Discrimination Criterion). *If simple terms M, N do not eventually match ($\mathrm{nf}_{\circledast}(M) \neq_{\mathrm{acc}} \mathrm{nf}_{\circledast}(N)$), then they are not β-convertible: $M \neq_{\beta} N$.*

Proof. The proof proceeds the same as the proof of Theorem 15 with the additional observation that due to M, N being simple, the symbol ξ cannot be duplicated (stems from a redex).

Example 20. We discriminate the unsolvables in Example 8. The term Ω is simple, and $\mu x.\mathsf{I}x =_{\beta} \mu x.x$ with $\mu x.x$ simple; see Example 17. We have $\mathrm{nf}_{\circledast}(\Omega) = \tau^{\omega}$ and $\mathrm{nf}_{\circledast}(\mu x.x) = \iota^{\omega}$. Hence by the second discrimination criterion (Corollary 19), the terms Ω and $\mu x.x$ are not $=_{\beta\mu}$-convertible.

Note that every term without weak head normal form in the $\lambda\beta\mu$-calculus gives rise to a clocked normal form which in fact is an infinite stream of τ's and ι's. For simple terms without whnf the discrimination criterion thus amounts to eventually matching of their corresponding streams. The initial segments are not relevant, it is the behavior at infinity that counts.

In Section 7 we give more example applications of this discrimination method.

6 A Functional Programming Application

We discuss a potential application of clocks for the performance optimization of functional programs. The transformations we mention here are well-known in functional programming and form a part of compile-time optimizations. Our point is merely that the clocked $\lambda\beta\mu$-calculus gives rise to a measure for comparing the performance of programs, thereby illustrating why a certain variant is preferable.

We have the following distributivity law

$$\mu f^{A \to B}.\lambda x^{A}.t(x, fx) = \lambda x^{A}.\mu y^{B}.t(x, y) \tag{1}$$

First of all, notice that computation of the Böhm Tree of the term on the left side of (1) does indeed contract an additional redex every time a recursive node is reached (corresponding to occurrences of fx). It follows by the discrimination theorem that these two terms are not convertible via finitary reduction steps. At the same time, the term on the right is to be preferred in any practical implementation of a recursive function of type $A \to B$. It may thus be of some interest that such patterns are actually quite common in programming practice.

Example 21. Consider the standard map function (we use `Clean` notation):

```
map f [] = []
map f [a:as] = [f a : map f as]
```

In this code, the argument f must be passed on during each recursive call, yielding an additional β-reduction step at every turn. This additional time step can be recovered by reimplementing `map` as follows:

```
map f as = map' as where
    map' [] = []
    map' [a:as] = [f a : map' as]
```

There is a close correspondence with the fixed point combinators of Curry and Turing, see Example 9. The first implementation of `map` corresponds to Turing's fpc Y_1 which passes the argument f from recursive call to recursive call, while the second implementation of `map` corresponds to Curry's fpc Y_0 which abstracts over f outside of the recursion. As shown in Example 9, Y_0 has a faster clock than Y_1.

We note that, fixing all recursive definitions in a program by abstracting their constant arguments over the recursion (as above) might not in itself eliminate all threats to efficiency. Functional programs are often built up from various combinators. Yet when one such combinator is applied to another, new "hidden redexes" may appear.

For example, the above `map` function could be invoked in order to "whiteout" a list by some constant `c`:

```
fill lst = map (\x = c) lst
```

At every invocation, this turns into the equivalent code

```
fill [] = []
fill [a:as] = [(\x = c) a : fill as]
```

(In the notation of the λ-calculus, we could write `fill` $=$ `map'`$[f := \lambda x.c]$.)

We note that this code is suboptimal: it has a redex $(\lambda x.c)a$ which appears in every recursive call, but which gives the same value. In this case the program is convertible to its best version:

```
fill [] = []
fill [a:as] = [c : fill as]
```

But the example illustrates how the functional clock can become slow because there is an extraneous redex that is created at every iteration.

(The clock is also slower when the redex in question occurs inside the function supplied to `map`. However, this situation is well-studied: it is precisely the argument in favor of strict evaluation of those function arguments that come to the head position in the body of the function.)

All these optimizations are related to the concept of *inlining* from compiler theory. In a sense, inlining is an operation that lifts redexes out from run-time into compile-time, where they can be contracted before program execution begins. This optimization comes with obvious associated space costs. Yet when time is of priority, it is generally a good idea to inline as much as possible.

Our current framework provides a possibility to detect inlining opportunities based on static syntactic inspection of code. It does not appear that all modern compilers of functional languages take advantage of this possibility in full generality. We wonder whether the `Clean` compiler can make use of such information!

7 Sequences of Fixed Point Combinators

Fixed point combinators are very suitable to test discrimination methods, because there are so many of them, and because they all have the same Böhm Tree $\lambda f.f^\omega$. When they are constructed in different ways, they can be inconvertible, in the case of this paper with its main calculus $\lambda\beta\mu^{\rightarrow}$, using the β- and μ-reduction rules. In this section we consider the most 'canonical' sequence of fpc's, that we call the Böhm sequence, and next a less well-known sequence, that we call the Scott sequence, due to the history and background of its construction (see [EHK10,EHKP12]). As a third topic in this section we analyze the question whether in the calculus $\lambda\beta\mu^{\rightarrow}$ there are more singleton fpc-generators like $\Box\delta$, using Barendregt's inhabitation machines to help us enumerate certain simple types.

7.1 The Böhm Sequence

Just as in the case of untyped λ-calculus we can conjure up an infinite sequence of fpc's $\mathsf{Y}_0, \mathsf{Y}_1, \ldots$ where $\mathsf{Y}_0 \equiv \lambda f.\mu y.fy$ and $\mathsf{Y}_{n+1} \equiv \mathsf{Y}_n\delta$ with $\delta \equiv \lambda ab.b(ab)$. It is easily checked that all Y_n are fpc's. What is much harder to check is that they are mutually different with respect to $=_{\beta\mu}$:

$$\mathsf{Y}_n =_{\beta\mu} \mathsf{Y}_m \quad \Longleftrightarrow \quad n = m$$

We have $\mathsf{Y}_{n+1} = \mathsf{Y}_1\delta^{\sim n}$ and the infinite clocked normal form of $\mathsf{Y}_1\delta^{\sim n}$ can be computed as follows:

$$\begin{aligned}
\mathsf{Y}_1\delta^{\sim n} \equiv (\mu x.\lambda f.f(xf))\delta^{\sim n} &\rightarrow_\mu \cdot \rightarrow_\iota^* \iota(\ (\lambda f.f(\mathsf{Y}_1 f))\delta^{\sim n}\ ) \\
&\rightarrow_\beta \cdot \rightarrow_\tau^* \iota\tau(\ \delta(\mathsf{Y}_1\delta)\delta^{\sim(n-1)}\ ) \\
&\quad \rightarrow^* \iota\tau^{1+2(n-1)}(\ \delta(\mathsf{Y}_1\delta^{\sim n})\ )
\end{aligned}$$

$$\begin{aligned}
&\to^* \iota\tau^{2n}(\ \lambda f.f(\mathsf{Y}_1\delta^{\sim n}f)\ )\\
&\to^* \iota\tau^{2n}(\ \lambda f.f(f(\iota\tau^{2n+1}(\ \mathsf{Y}_1\delta^{\sim n}f\ )))\ )\\
&\twoheadrightarrow\!\!\!\!\twoheadrightarrow \iota\tau^{2n}(\lambda f.f(\iota\tau^{2n+1}(f(\iota\tau^{2n+1}(\ldots)))))\\
&\ \equiv \iota\tau^{2n}\ \lambda f.f(\iota\tau^{2n+1}f)^{\omega}
\end{aligned}$$

Thus $\mathrm{nf}_{\text{⏲}}(\mathsf{Y}_1\delta^{\sim n}) \equiv \iota\tau^{2n}\ \lambda f.f(\iota\tau^{2n+1}f)^{\omega}$. Note that in the computation of the clocked normal form of $\mathsf{Y}_1\delta^{\sim n}$ we really need the shift-rules for τ and ι in order to let these constants not impede the necessary reductions. Here we used the notation M^{ω} for $M(M(M(\ldots)))$, so e.g. $(\iota\tau f)^{\omega} \equiv \iota\tau f\iota\tau f\iota\tau f\ldots$ with all brackets associating to the right.

Note further that we have carried out the reduction in a top-down fashion, and none of the steps has duplicated a redex. Thus the terms $\mathsf{Y}_1\delta^{\sim n}$ are simple. By the second discrimination criterion (Corollary 19), we can discriminate these fixed point combinators pairwise with respect to $=_{\beta\mu}$ since their clocked normal forms do not eventually match.

7.2 The Scott Sequence

As shown in [EHK10,EHKP12], there is another way to generate new fpc's as follows: if Y is a reducing fpc, then $Y(\mathsf{SS})\mathsf{I}$ is an fpc. Indeed we calculate

$$Y(\mathsf{SS})\mathsf{I}x \twoheadrightarrow_{\beta} \mathsf{SS}(Y(\mathsf{SS}))\mathsf{I}x \twoheadrightarrow_{\beta} \mathsf{SI}(Y(\mathsf{SS})\mathsf{I})x \twoheadrightarrow_{\beta} x(Y(\mathsf{SS})Ix)$$

In fact we have for every $n \geq 0$: Y is a reducing fpc $\implies Y(\mathsf{SS})\mathsf{S}^{\sim n}\mathsf{I}$ is a reducing fpc. Here we use the notation $AB^{\sim n}$ defined by $AB^{\sim 0} = A$ and $AB^{\sim n+1} = ABB^{\sim n}$.

In this way we can generate many new fpc's. The question however is how to show that they are indeed new, i.e., that for different sequences of 'fpc-building blocks' $\pi_1, \ldots, \pi_k$ and $\pi'_1, \ldots, \pi'_k$ where each π_i and π'_j is $\square\delta$ or $\square(\mathsf{SS})\mathsf{S}^{\sim n}\mathsf{I}$ for some $n \geq 0$, we have $M_1 \equiv \mathsf{Y}_0\pi_1, \ldots, \pi_k \neq_{\beta\mu} \mathsf{Y}_0\pi'_1, \ldots, \pi'_k \equiv M_2$.

To perform the discrimination argument we can proceed in analogy with the treatment above for $\square\delta$ with two stipulations. First, we have to simplify the terms by reducing the subterms SS to their normal forms $\lambda abc.bc(abc)$. Second, we need the refined atomic clocks defined in Remark 10. Otherwise we could not distinguish the effect of swapping two blocks in the sequence $\pi_1, \ldots, \pi_k$.

7.3 Other Fpc-Generators

As we have seen above, the term $\delta \equiv \lambda ab.b(ab)$ has the peculiar property that it generates new fpc's when postfixed to an already available fpc: Y is an fpc $\implies$ $Y\delta$ is an fpc. We shall now consider the following problem:

Give the set of λ-terms G such that, for an fpc Y, YG is again an fpc.

For general G this problem becomes intractable due to the usual pathologies of the type-free λ-calculus. However, it is interesting to solve this problem for the simply-typed setting. That is, we shall work in $\lambda\beta\mu^{\to}$. In fact, with $\lambda\beta\eta\mu^{\to}$ as it is natural in this subsection to include the η-rule

$$\eta : \quad \lambda x.Mx \to M \qquad \text{if } x \text{ not free in } M$$

and long $\beta\eta$-normal forms (this is the only part of the paper where the η-rule is used).

In this context, we solve the above problem using the technique of Barendregt's Inhabitation Machines [BDS13].

Suppose G is such that YG is a fixed point combinator in $\lambda\beta\mu^{\to}$, for Y fpc. Since YG must have type $(\alpha \to \alpha) \to \alpha$ for any α, while Y has type $(A \to A) \to A$, for some A, we must have that $A = (\alpha \to \alpha) \to \alpha$.

For $\sigma \in \mathbb{T}$, write σ^o for the type $(\sigma \to \sigma) \to \sigma$. With this notation, the above becomes

$$YG : \alpha^o$$
$$Y : (\alpha^o \to \alpha^o) \to \alpha^o \quad (= \alpha^{oo})$$

whence we see that G must have type $\alpha^o \to \alpha^o$. Using the inhabitation machines, we enumerate all closed $\beta\eta$-normal forms of this type. (Here, as is usual in type theory, the normal forms for η are the "long" normal forms, where every subterm of type $A_1 \to \cdots \to A_n \to \alpha$ begins with n abstractions.)

Using [BDS13, 2.3] we get the diagram:

$$\begin{tikzcd}
& \boxed{\alpha^o \to (\alpha \to \alpha) \to \alpha} \arrow[d, "\lambda y^{(\alpha\to\alpha)\to\alpha}\lambda f^{\alpha\to\alpha}"'] \\
a & \boxed{\alpha} \arrow[l] \arrow[loop right, "f"] \arrow[d, bend right, "y"'] \\
& \boxed{\alpha \to \alpha} \arrow[u, bend right, "\lambda a"']
\end{tikzcd}$$

The paths through the above diagram terminating in a leaf node correspond to finite $\beta\eta$-normal forms of the given type, while infinite paths correspond to infinite $\beta\eta$-normal forms. From the diagram, we see that the general form of a term of type $\alpha^o \to \alpha^o$ is

$$\lambda y^{\alpha^o} \lambda f^{\alpha\to\alpha}.f^{n_0}(y(\lambda a_1.f^{n_1}(y(\lambda a_2.f^{n_2}(\ldots y(\lambda a_k.f^{n_k} a_i)\cdots)$$

Let $\mathcal{G}$ be the collection of such terms, and let $G \in \mathcal{G}$. We shall now investigate under what conditions G is an fpc generator.

Let us note immediately that n_0 must be positive, for otherwise the head variable of the term is its first abstracted variable, and an application of a fixed point combinator to such a term always results in an unsolvable.

Notice also that a_i occurs in G for exactly one i. So we can write

$$G = \lambda yf.f^{1+n_0'}(y(\mathsf{K}(f^{n_1}(\cdots y(\lambda a.f^{n_i}(y \cdots \mathsf{K}(f^{n_k} a)\cdots)$$

where $a = a_i$ and $n_0' = n_0 - 1$.

Proposition 22. *Let Y be a (w)fpc and $Y_G = YG$. Then*

$$Y_G f = f^m (Y_G f^n)$$

where $m = n_1 + \cdots + n_{i-1}$, and $n = n_i + \cdots + n_k$.

Proof. We are going to show that

$$\begin{aligned} Y_G f &= f^{\sum_{j=0}^{i-1} n_j}(Y_G(\lambda a.f^{\sum_{j=i}^{k} n_j} a)) \\ &= f^{n_1+\cdots+n_{i-1}}(Y_G(\lambda a.f^{n_i+\cdots+n_k} a)) \end{aligned}$$

by the following five steps.

1. For any number n_j, we have

$$\begin{aligned} Y_G(\mathsf{K}(f^{n_j} a)) &= G Y_G(\mathsf{K}(f^{n_j} a)) \\ &= (\mathsf{K}(f^{n_j} a))^{1+(n_0-1)}(Y_G(\mathsf{K}\cdots))) \\ &= \mathsf{K}(f^{n_j} a)\left((\mathsf{K}(f^{n_j} a))^{n_0-1}(Y_G\cdots)\right) \\ &= f^{n_j} a \end{aligned}$$

2. By induction, we have, for $j \leq j'$

$$\begin{aligned} &Y_G(\mathsf{K}(f^{n_j}(\cdots Y_G(\mathsf{K}(f^{n_{j'}} a))\cdots) \\ &\qquad = Y_G(\mathsf{K}(f^{n_j} a))[a := Y_G(\mathsf{K}(f^{n_{j+1}}(\cdots Y_G(\mathsf{K}(f^{n_{j'}} a))\cdots)] \\ &\qquad =_1 f^{n_j}(Y_G(\mathsf{K}(f^{n_{j+1}}(\cdots Y_G(\mathsf{K}(f^{n_{j'}} a))\cdots) \\ &\qquad =_{IH} f^{n_j}(f^{n_{j+1}+\cdots+n_{j'}} a) \\ &\qquad = f^{n_j+\cdots+n_{j'}} a \end{aligned}$$

3. In particular, we have

$$Y_G(\mathsf{K}(f^{n_i+1}(\cdots Y_G(\mathsf{K}(f^{n_k} a))\cdots) = f^{n_{i+1}+\cdots+n_k} a \tag{2}$$

$$\begin{aligned} Y_G(\mathsf{K}(f^{n_1}(\cdots Y_G(\mathsf{K}(f^{n_{i-1}} A)\cdots) &= Y_G(\mathsf{K}(f^{n_1}(\cdots Y_G(\mathsf{K}(f^{n_{i-1}} a)\cdots)[a := A] \\ &= f^{n_1+\cdots+n_{i-1}} A \end{aligned} \tag{3}$$

4. Using (2), we get

$$\lambda a.f^{n_i}(Y_G(\cdots \mathsf{K}(f^{n_k} a))) = \lambda a.f^{n_i+\cdots+n_k} a \tag{4}$$

5. Putting it all together gives

$$\begin{aligned} Y_G f &= f^{n_0}(Y_G(\mathsf{K}(f^{n_1}(\cdots Y_G(\lambda a.f^{n_i}(\cdots Y_G(\mathsf{K}(f^{n_k} a)\cdots) \\ &=_{(4)} f^{n_0}(Y_G(\mathsf{K}(f^{n_1}(\cdots Y_G(\lambda a.f^{n_i+\cdots+n_k} a)\cdots) \\ &=_{(3)} f^{n_0}(f^{n_1+\cdots+n_{i-1}}(Y_G(\lambda a.f^{n_i+\cdots+n_k} a)\cdots) \\ &= f^{n_0+\cdots+n_{i-1}}(Y_G f^{n_i+\cdots+n_k}) \end{aligned}$$

being what was required to show.

What conclusions can be drawn from this proposition? We know that a generic member of $\mathcal{G}$ is determined by the numbers $(k, i, n_1, \ldots, n_k)$, $1 \leq i \leq k$. While it seems likely that all these generators are "unique" – in the sense that when $G \neq G'$, there can be no finite conversion between Y_G and Y'_G – their behaviors nevertheless collapse to the 2-parameter family

$$g_{m,n} = \lambda yf.f^m(yf^n)$$

We can now consider two pre-generators $G, G' \in \mathcal{G}$ to be equivalent, if their "reduced pump" $\lambda yf.f^m(yf^n)$ is the same.

Let us observe the execution of such a reduced generator:

$$\begin{aligned}
Y_G f &= f^m(Y_G f^n) \\
&= f^m(f^{mn}(Y_G f^{n^2})) \\
&= f^m(f^{mn}(f^{mn^2}(Y_G f^{n^3}))) \\
&= \vdots \\
&= f^{m(1+n+\cdots+n^k)}(Y_G f^{n^{k+1}}) \\
&= \vdots \\
&= \begin{cases} \bot & m = 0 \\ f^m\bot & n = 0 \\ f^\omega & m, n > 0 \end{cases}
\end{aligned}$$

Thus, every such generator $g_{m,n}$ with $m, n > 0$ leads to a wfpc-generator. Conversely, the previous inhabitation argument shows that every finite simply typed wfpc-generator is of this form.

Simple intuition now tells us that, in order that $G = g_{m,n}$ be an fpc-generator, it must transpire that $m = n = 1$. Indeed, thinking of a given (w)fpc Y as a single "motor", we are poised to measure its clock velocity by counting how many fs it produces at each step, as well as how quickly it speeds itself up, by changing f to an n-fold composition of it. If $n > 1$, then the clock perpetually speeds itself up, so that Y_G cannot be an fpc. If $n = 1$ but $m > 1$, then the clocks are indeed the same, but they are "de-synced" between the terms $Y_G f$ and $f(Y_G f)$, because at k-th iteration the former will have km occurrences of f at the head, while the latter will have $km + 1$, which cannot be synchronized.

To complete the classification, it remains to ask which elements of $\mathcal{G}$ are equivalent to $g_{1,1}$. The answer brings us to the following result.

Theorem 23. *Let G be a finite simply-typed fpc-generator. There are integers $l, m, n \geq 0$ such that*

$$G \ = \ \lambda y\lambda f.f(y \circ \mathsf{K})^l(y(\lambda a.(y \circ \mathsf{K})^m(f((y \circ \mathsf{K})^n a))))$$

Remark 24. After the first iteration, it is seen that such a G exhibits the same behavior as the simplified term $g_{1,1} = \delta = \lambda yf.f(yf)$. It is precisely in this sense that this generator, discovered by Corrado Böhm and being the first term to be described as such, is unique. The uniqueness is with respect to the clocked behavioral equivalence of fpc 1-generators. It is the minimal representative of this equivalence class, being the only class of solutions that are candidate fpc-generators.

Interestingly, the inhabitation problem also generates infinite solutions:

$$G = \lambda y^{\alpha^o} \lambda f^{\alpha \to \alpha}.f^{n_0}(y(\lambda a_1.f^{n_1}(y(\lambda a_2.\cdots)\cdots)$$

with no occurrences of a_i.

Clearly, if $n_i = 0$ for all $i > 0$, then G is not an fpc-generator, because

$$Y_G f = G Y_G f = f^{n_0} Z$$

where $f \notin \mathsf{FV}(Z)$. Then $\mathsf{BT}(Y_G f) \neq f^\omega$.

Similarly, if $n_i = 0$ for $i > i_0$, then we can apply Step 2 of the above proposition to get

$$Y_G(\mathsf{K}(f^{n_0}(Y_G(\mathsf{K}(f^{n_1}\cdots(Y_G(\mathsf{K}(f^{n_{i_0}}Z)\cdots) = f^{\Sigma} Z$$

where $\Sigma = n_1 + \cdots + n_{i_0}$. But $f \notin \mathsf{FV}(Z)$. So

$$Y_G f = f^{n_0}(Y_G(\mathsf{K}(f^{n_1}\cdots)\cdots) = f^{\Sigma} Z \neq f^\omega$$

(In this example, as well as the previous one, the term Z can be given explicitly: $Z = Y_G(KZ) = (Y_G \circ K)^\omega$.)

Conversely, if $n_i \neq 0$ for infinitely many i, we have that

$$Y_G f = f^M(\cdots)$$

for M larger than any given number. Thus Y_G is indeed a weak fixed point combinator.

8 Further Questions

In this final section we discuss some important questions. In particular we point to a conjecture which could have several deep consequences (see Section 8.3). The conjecture connects the simply typed $\lambda\beta\mu^{\to}$-calculus with the untyped $\lambda\beta$-calculus. If it is true, it would be a striking example of how simple types can be used to obtain results in the pure untyped $\lambda\beta$-calculus.

8.1 Decidability of Fixed Point Combinators

The simultaneous restriction–extension of $\lambda\beta\mu^{\rightarrow}$ presents us with a more abstract, high level view on fixed point combinators. For, the simple types restriction disallows much of the possible complexity that fpc's may possess – spurious complexity one might say. In particular self-application in subterms is removed, at least between subterms of the same type. Thus the formerly simplest fpc's of Curry and Turing, respectively

$$\lambda f.(\lambda x.f(xx))(\lambda x.f(xx)) \qquad \text{and} \qquad (\lambda ab.b(aab))(\lambda ab.b(aab))$$

are ruled out, and in $\lambda\beta\mu^{\rightarrow}$ are replaced by the simpler

$$\lambda x.\mu y.xy \qquad \text{and} \qquad \mu x.\delta x\ ,$$

respectively.

At this point, it is interesting to speculate how complicated fpc's can be in $\lambda\beta\mu^{\rightarrow}$. Is the notion of fpc still undecidable? Does Scott's theorem used to show the undecidability of the notion of fpc's in pure lambda calculus still hold?

What number theoretic functions are definable in this calculus, when one restricts to working with the Church numerals (of type $\alpha \to (\alpha \to \alpha) \to \alpha$)?

8.2 Comparison with PCF

When we extend the calculus to its infinitary version (see Figure 1), similar questions can be asked. At present the meta-theory of the calculus $\lambda\beta\mu^{\rightarrow}$ is not fully clear to us, in particular its relation to PCF where the native type of natural numbers leads to Turing-completeness.

8.3 Completeness of μ-Reduction

The $\lambda\beta\mu^{\rightarrow}$-calculus can be interpreted in the untyped $\lambda\beta$-calculus by instantiating the μ-constructor with any fixed point combinator Y.

Definition 25. Let Y be an fpc. For a simply typed $\lambda\mu$-term t, its Y*-translation* $|t|_Y$ is defined by induction on t, as follows:

t	$\lvert t\rvert_Y$
x	x
st	$\lvert s\rvert_Y \lvert t\rvert_Y$
$\lambda y^A.t$	$\lambda y.\lvert t\rvert_Y$
$\mu z^A.t$	$Y(\lambda z.\lvert t\rvert_Y)$

The following is a deep conjecture about fixed point combinators.

Conjecture 26. For any fpc Y and simply typed s, t we have:

$$|s|_Y =_\beta |t|_Y \iff s =_{\beta\mu} t\,.$$

Remark 27. (i) The direction $\Leftarrow$ of the conjecture is trivial (soundness). The interesting part is completeness, $\Rightarrow$.

(ii) It is essential that the right-hand side involves types. Otherwise we have the following 'chiasm' counterexample:

$$s = \lambda z.z(\mu x.x)(Y(\lambda x.x)) \qquad\qquad t = \lambda z.z(Y(\lambda x.x))(\mu x.x)$$

Then $s \neq_{\beta\mu} t$ but $|s|_Y \equiv |t|_Y$.

(iii) It is also interesting to give the equivalent reformulation of the conjecture for the $\lambda\beta\mathsf{Y}^{\rightarrow}$-calculus.

To illustrate the fundamental importance of this conjecture, we show that its positive resolution would yield immediate answers to questions posed by Plotkin, Statman and Klop.

Question 28. (Plotkin) Does there exist an fpc Y such that $|\mu x.\mu y.fxy|_Y = |\mu x.fxx|_Y$?

The question of Plotkin has been answered in [EHKP12] using clocked Böhm Trees. A positive answer to Conjecture 26 would yield this result immediately.

Corollary 29. *If the conjecture holds, the answer to Plotkin's question is "no".*

Proof. For Plotkin's question, consider the terms $s \equiv \mu x.\mu y.fxy$ and $t \equiv \mu x.fxx$. As we noted before, these terms have the same infinitary normal form. However, they are not finitely convertible (every reduct of s has nesting of μ's whereas no reduct of t has). Hence for no fpc Y are their images under the $|\cdot|_Y$ map convertible, yielding a negative answer to Plotkins question.

Question 30. (Statman) Is it the case that for *no* fpc Y we have $Y\delta = Y$?

A proof of this conjecture given by Intrigila [Int97] turned out to contain a serious gap, see [EHKP13]. Thus the answer to this conjecture remains open. A positive answer to Conjecture 26 would immediately imply Statman's conjecture as follows.

Corollary 31. *If the conjecture holds, the answer to the conjecture of Statman is "yes".*

Proof. Suppose there exists an fpc Y convertible with $Y\delta$. Then $Y\delta =_\beta Y\delta\delta$. Let

$$s \equiv (\lambda f.\mu x.fx)\delta \qquad\qquad t \equiv (\lambda f.\mu x.fx)\delta\delta$$

It is not difficult to see that s and t are typable, and $(*)$ $s \neq_{\beta\mu} t$ since the terms have different clocks, see further the Böhm sequence in [EHKP12]. We have

$$|s|_Y = (\lambda f.Y(\lambda x.fx))\delta =_\beta Y(\lambda x.\delta x) =_\beta Y\delta \qquad\qquad |t|_Y =_\beta Y\delta\delta$$

If $Y\delta =_\beta Y\delta\delta$, then we have $|s|_Y =_\beta |t|_Y$. However, then by Conjecture 26 we get $s =_{\beta\mu} t$, contradicting $(*)$.

We briefly indicate that this method has much wider applicability, namely establishing a conjecture by Klop [EHK10,EHKP12], generalizing Statman's conjecture considerably. The conjecture refers to several fpc generating schemes, of which the following are examples:

(G_1) $\Box\delta$;
(G_2) $\Box(\mathsf{SS})\mathsf{S}^{\sim n}\mathsf{I}$ for $n \in \mathbb{N}$;
(G_3) $\Box(\mathsf{AAA})\mathsf{A}^{\sim n}\mathsf{II}$ for $n \in \mathbb{N}$.

Note that (G_2) and (G_3) are schemes of generating vectors. There are actually infinitely many of such fpc-generating schemes, but we will stick with the three as mentioned. They enable us to build fpcs in a modular way by repeatedly adding a vector as given by one of the schemes, starting with some arbitrary fpc Y.

Question 32. (Klop) Constructing fpcs in this way is a 'free construction' in that never non-trivial identifications will arise: Let Y, Y' be fpc's and let $B_1 \dots B_n$, $C_1 \dots C_k$ be picked from the fpc-generating vectors (i),(ii),(iii) above. Then we have:

(i) $YB_1 \dots B_n =_\beta Y'B_1 \dots B_n$ iff $Y = Y'$;
(ii) $YB_1 \dots B_n =_\beta YC_1 \dots C_k$ iff $B_1 \dots B_n \equiv C_1 \dots C_k$.

Already the restriction to (G_1), the generating vector δ, is a generalization of Statman's conjecture, stating that $Y\delta^n \neq Y\delta^m$ for any fpc Y and $n \neq m$ (instead of $Y \neq Y\delta$). This indicates how non-trivial this conjecture is for the general case. For the particular fpc Y_0 we have partial results:

(i) The sequence Y_0, $\mathsf{Y}_0\delta$, $\mathsf{Y}_0\delta\delta$,... is known as the Böhm sequence, and is known to not contain any duplicates.
(ii) For Y_0 in combination with the set of all generating vectors (G_2), the conjecture has been proven in [EHKP12].

Both results can easily be extended to the setting of $\lambda\beta\mu^{\rightarrow}$. We therefore think that there is hope to prove freeness of the construction for all generating vectors (G_1), (G_2), (G_3) for the $\lambda\beta\mu^{\rightarrow}$ calculus. Then a positive answer to Conjecture 26 would immediately yield a positive answer to Klop's conjecture.

In view of the strong consequences of the Conjecture 26 one must expect that the conjecture is indeed difficult to prove. Also a counterexample would be very interesting. Even if the conjecture fails, or as a partial result towards the conjecture, it would be interesting to determine a class of fixed point combinators for which the conjecture holds.

References

[AK95] Z.M. Ariola and J.W. Klop. Equational Term Graph Rewriting. Technical Report IR-391, Vrije Universiteit Amsterdam, 1995. `ftp://ftp.cs.vu.nl/pub/papers/theory/IR-391.ps.Z`.

[Bar84] H.P. Barendregt. *The Lambda Calculus. Its Syntax and Semantics*, volume 103 of *Studies in Logic and The Foundations of Mathematics*. North-Holland, revised edition, 1984.

[BDS13] H.P. Barendregt, W. Dekkers, and R. Statman. *Lambda Calculus with Types*. Perspectives in Logic. Cambridge University Press, 2013.

[Ber85] I. Bercovici. Unsolvable Terms in Typed Lambda Calculus with Fix-Point Operators. In Rohit Parikh, editor, *Logics of Programs*, volume 193 of *Lecture Notes in Computer Science*, pages 16–22. Springer, 1985.

[BKdV00] I. Bethke, J.W. Klop, and R.C. de Vrijer. Descendants and Origins in Term Rewriting. *Information and Computation*, 159(1–2):59–124, 2000.

[BN98] F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

[CC96] C. Coquand and T. Coquand. On the Definition of Reduction for Infinite Terms. *Comptes Rendus de l'Académie des Sciences. Série I*, 323(5):553–558, 1996.

[dMJB$^+$] P. de Mast, J.-M. Jansen, D. Bruin, J. Fokker, P. Koopman, S. Smetsers, M. van Eekelen, and R. Plasmeijer. *Functional Programming in Clean*.

[EGKvO11] J. Endrullis, C. Grabmayer, J.W. Klop, and V. van Oostrom. On Equal μ-Terms. *Theoretical Computer Science*, 412(28):3175–3202, 2011.

[EHH$^+$13] J. Endrullis, H.H. Hansen, D. Hendriks, A. Polonsky, and A. Silva. A Coinductive Treatment of Infinitary Term Rewriting, 2013. Submitted.

[EHK10] J. Endrullis, D. Hendriks, and J.W. Klop. Modular Construction of Fixed Point Combinators and Clocked Böhm Trees. In *Proc. Symp. on Logic in Computer Science (LICS 2010)*, pages 111–119, 2010.

[EHK12] J. Endrullis, D. Hendriks, and J.W. Klop. Highlights in Infinitary Rewriting and Lambda Calculus. *Theoretical Computer Science*, 464:48–71, 2012.

[EHKP12] J. Endrullis, D. Hendriks, J.W. Klop, and A. Polonsky. Discriminating Lambda-Terms using Clocked Böhm Trees. *Logical Methods in Computer Science*, 2012. In print.

[EHKP13] J. Endrullis, D. Hendriks, J.W. Klop, and A. Polonsky. Clocked Lambda Calculus. *Mathematical Structures in Computer Science*, 2013. Accepted for publication.

[EP13] J. Endrullis and A. Polonsky. Infinitary Rewriting Coinductively. In *Proc. Types for Proofs and Programs (TYPES 2012)*, volume 19 of *LIPIcs*, pages 16–27. Schloss Dagstuhl, 2013.

[HS08] J.R. Hindley and J.P. Seldin. *Lambda-Calculus and Combinators*. Cambridge University Press, 2008.

[Hut07] Graham Hutton. *Programming in Haskell*. Cambridge University Press, 2007.

[Int97] B. Intrigila. Non-Existent Statman's Double Fixed Point Combinator Does Not Exist, Indeed. *Information and Computation*, 137(1):35–40, 1997.

[KdV05] J.W. Klop and R.C. de Vrijer. Infinitary Normalization. In *We Will Show Them: Essays in Honour of Dov Gabbay*, volume 2, pages 169–192. College Publ., 2005. Techn. report: `http://www.cwi.nl/ftp/CWIreports/SEN/SEN-R0516.pdf`.

[KS09] J. Ketema and J.G. Simonsen. Infinitary Combinatory Reduction Systems: Confluence. *Logical Methods in Computer Science*, 5(4):1–29, 2009.

[NI89] T. Naoi and Y. Inagaki. Algebraic Semantics and Complexity of Term Rewriting Systems. In *Proc. Conf. on Rewriting Techniques and Applications (RTA 1989)*, volume 355 of *Lecture Notes in Computer Science*, pages 311–325. Springer, 1989.

[Pla84] R.A. Platek. *Foundations of Recursion Theory.* University Microfilms, 1984.

[Plo77] Gordon D. Plotkin. Lcf considered as a programming language. *Theor. Comput. Sci.*, 5(3):223–255, 1977.

[Smu85] R. Smullyan. *To Mock a Mockingbird, and Other Logic Puzzles: Including an Amazing Adventure in Combinatory Logic.* Alfred A. Knopf, New York, 1985.

[Sta02] R. Statman. On The Lambda Y Calculus. In *Proc. Symp. on Logic in Computer Science (LICS 2002)*, pages 159–166. IEEE, 2002.

[Ter03] Terese. *Term Rewriting Systems*, volume 55 of *Cambridge Tracts in Theoretical Computer Science.* Cambridge University Press, 2003.