The Recursive Polarized Dual Calculus

Aaron Stump
Computer Science
The University of Iowa
Iowa City, Iowa, USA
astump@acm.org

Abstract

This paper introduces the Recursive Polarized Dual Calculus (RP-DC), based on Wadler's Dual Calculus. RP-DC features a polarized form of reduction, which enables several simplifications over previous related systems. It also adds inductive types with recursion, from which coinductive types with corecursion can be defined. Typing and reduction relations are defined for RP-DC, and we consider several examples of practical programming. Logical consistency is proved, as well as a canonicity theorem showing that all closed values of a certain family of types are canonical. This shows how RP-DC can be used for practical programming, where canonical final results are required.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory; F.4.1 [*Mathematical Logic*]: Proof Theory

Keywords Dual Calculus; classical type theory; mixed induction/coinduction

1. Introduction

Starting with the seminal observation of Griffin [10], a number of authors have sought to develop type theories corresponding to classical logic, thus extending the well-known Curry-Howard isomorphism beyond intuitionistic logic. Among the most influential of such Computational Classical Type Theories (CCTTs) are those of Parigot, Curien and Herbelin, and Wadler [7, 16, 18]. This program has obvious theoretical appeal – to lift a connection that has been remarkably fruitful from intuitionistic to classical logic – as well as practical motivations:

1. Some natural forms of reasoning are simply not possible constructively. A beautiful example is given in a recent paper by Bezem, Nakata, and Uustalu [5], which shows, among other results, that various natural reasoning principles about infinite streams are equivalent to the Lesser Principle of Omniscience (an infinite stream of red/blue values is either all blue or contains a red value somewhere), which is not computationally realizable. For reasoning about general-recursive programs or computations on infinite streams, classical logic is needed in some cases. Therefore, type-theoretic support for classical reasoning is an important goal.

2. Most programming languages include some form of control operator, like exceptions or `call/cc` in Scheme, possibly in quite refined forms [8, 9]. Griffin's observation was that the types one naturally assigns to control operators like `call/cc` correspond to strictly classical validities. A type-theoretic view of classical logic promises to add support for control operators – a powerful idiom of practical programming – to type theory. We have the challenge of explaining how a language embracing the nonconstructivity of classical logic can still be suitable for programming (the paradigmatic informationally constructive activity).

3. The problem of combining induction and coinduction in a terminating type theory has received significant attention recently (e.g., [1, 4]). In some CCTTs, such as Wadler's Dual Calculus (DC) [18], the dualities of classical logic are clearly expressed in the type system. One can hope that the categorical duality between induction and coinduction could be expressed formally in such a system.

This paper proposes the Recursive Polarized Dual Calculus (RP-DC), based on DC. RP-DC differs from DC on several design points. First, the core of the type theory is based just on conjunction and negation, with disjunction and implication defined as usual in terms of these, along with their term constructs. The reductions one would expect for the term constructs are derivable. For this, I found it necessary to add a polarity $p \in \{+, -\}$ to the reduction judgment $p\ t_1 \bullet t_2 \rightsquigarrow p'\ t_1' \bullet t_2'$. RP-DC also uses a single polarized typing judgment $\Gamma \vdash t : p\ T$, rather than separate left and right judgments, as in DC. Another design goal is to pave the way for adding dependent types to the calculus, which is currently simply typed. This is done by adopting a different form $\iota\, x.t$, instead of $\iota_1 t$ and $\iota_2 t$, for terms which refute conjunctions.

RP-DC includes recursive types, and a term construct for recursion – additions already considered for DC by Kimura and Tatsuta [11]. But they take both μ-types (inductive) and ν-types (co-inductive) as primitive notions in their system $\mathrm{DC}_{\mu\nu}$. In contrast, RP-DC takes μ-types as primitive, and defines $\nu\, X.T$ as $\neg(\mu\, X.\neg[\neg X/X]\, T)$. This definition is standard in propositional μ-calculus [12]. The reduction rules for the recursor are much simpler in RP-DC than in $\mathrm{DC}_{\mu\nu}$, which rely on a complex meta-level function $\mathrm{mono}^{X.C}_{A,B,x.M} N$ [11]. In RP-DC, the recursor uses an accumulator, and so is similar to continuation-passing style (well-known to be as expressive as more flexible forms of recursion [3]).

This paper also resolves the tension between the constructive nature of programming with control operators, and the nonconstructive

PLPV '14, January 21, 2014, San Diego, CA, USA.

http://dx.doi.org/10.1145/2541568.2541575

character of classical logic. This is done first by identifying the locus of nonconstructivity in the failure, in general, of a property we can call *canonicity*. In the absence of control operators, typable closed normal forms are always values of certain canonical forms. Once control operators are added, however, canonicity is lost in general: some types are inhabited by closed normal forms which are not canonical in any obvious sense. The paper identifies a class of types containing the algebraic datatypes, for which a canonicity property does indeed hold. This is proved in Section 10.

To summarize the main contributions of RP-DC:

1. Typing and reduction rules for standard type constructs are derivable from a logically minimal set consisting of conjunction and negation, thanks to a polarized form of the reduction relation.
2. Recursion receives a novel formulation that is significantly simpler than in previous work [11], from which corecursion can be defined.
3. Two major metatheoretic results are established: logical consistency, which says that not every type is inhabited; and a canonicity theorem.

The ultimate goal is to design and implement a dependent type theory corresponding to classical logic, with direct support for control operators, a natural dual formulation supporting mixed induction/co-induction, and an identified subset of types where canonicity holds. Such a system would support verification by classical reasoning for pure functional programs with control, using both inductive and coinductive types. No existing type-theoretic verification system provides all these features.

2. Syntax of RP-DC

The syntax of types is:

$$\textit{types}\ T \quad ::= \quad X \mid \mu\, X.T \mid T \wedge T' \mid \neg T$$

The type constructs are type variables X, inductive types $\mu\, X.T$, conjunctions $T \wedge T'$, and negations $\neg T$. The scope of μ is as far to the right as possible, and negation binds more tightly than conjunction. The typing rules will restrict the bodies T of inductive types $\mu\, X.T$ to have only positive occurrences of X. In addition, we make the following definitions:

$$\begin{array}{lcl} \bot & := & \mu\, X.X \\ \top & := & \neg \bot \\ T \vee T' & := & \neg(\neg T \wedge \neg T') \\ T \to T' & := & \neg(T \wedge \neg T') \\ \nu\, X.T & := & \neg\mu\, X.\neg[\neg X/X]\, T \end{array}$$

The reader can easily confirm that the definitions of disjunction and implication are classically valid under the usual semantics of classical propositional logic. In the previous systems by Wadler [18] and Kimura and Tatsuta [11], some of these types were taken as primitive. We will see definitions of their term constructs below.

The syntax of terms is:

$$\begin{array}{rcl} \textit{terms}\ t & ::= & x \mid \mathbf{halt}\ T \mid (t, t') \mid \iota\, x.t \mid \mathbf{not}\ t \mid \\ & & \delta\, x.t \bullet t' \mid x[t] \mid \mathbf{rec}\, x[y = t].t' \end{array}$$

The term $\mathbf{halt}\ T$ represents a continuation for receiving the final result on computation. The term (t, t') is pairing, for proving conjunctions, while $\iota\, x.t$ is for refuting conjunctions. The term $\mathbf{not}\ t$ switches polarity from proving to refuting, and vice versa. The term $\delta\, x.t \bullet t'$ cuts a positive term t (that is, one which is typable at positive polarity) against a negative one t'. The remaining two constructs are for recursion. The term t in $\mathbf{rec}\, x[y = t].t'$ is called the *accumulator* of the **rec**-term. It will play a role similar to accumulator arguments in standard functional programming. It is initialized to t in $\mathbf{rec}\, x[y = t].t'$, and then updated to t'' in recursive calls of the form $x[t'']$, which may occur in t'. Note that δ and ι bind x and **rec** binds x and y. It is not standard to use δ as the binder in cut terms, but I prefer to reserve μ and ν for inductive and coinductive types, respectively. We give **not** higher parsing precedence than the other operators.

3. Typing

The form of typing judgment for RP-DC is $\Gamma \vdash t : p\ T$, where $p \in \{+, -\}$ is a polarity, and Γ is a context, specified by:

$$\textit{contexts}\ \Gamma \quad ::= \quad . \mid \Gamma, x : p\ T \mid \Gamma, x : p\ X \triangleright T$$

Unlike in DC, we work with a single context, to the left of the turnstile, and explicitly keep track of polarities p in assumptions $x : p\ T$. Positive polarity means the assumption is true, or computationally, producing data; while negative polarity means it is false, or computationally, consuming data. The second form of assumption, $x : p\ X \triangleright T$, is used for typing recursions (**rec**-terms). Both of these are said to *declare* x. We will see examples in Section 6 below.

In a judgment $\Gamma \vdash t : p\ T$, if p is $+$ the judgment is *affirming* T, and if the judgment holds we say T is *proved* (in context Γ); and if p is $-$, the judgment is *denying* T, and T is *refuted*. Also, we will assume that variables bound in terms are implicitly renamed so that no context declares the same variable twice, and so that in any context of the form $\Gamma, x : p\ X \triangleright T, \Gamma'$, context Γ does not contain X. We write $\bar{p}$ for the opposite polarity from p (i.e., the one which is not p).

The typing rules for RP-DC are given in Figure 1. The AX rule simply makes use of an assumption from the context, at the assumed polarity. ANDPOS is for proving a conjunction: one must prove both conjuncts separately. ANDNEG is for refuting a conjunction, by refuting the first conjunct, under an assumption x that the second conjunct holds. This single rule is sufficient, since if in fact we can refute the first conjunct, then we can cut that refutation against the assumption x, to derive a contradiction (and hence prove the second conjunct). The NOT rule says that to derive a negation $\neg T$ with polarity p, one must derive T with the opposite polarity ($\bar{p}$). The CUT rule says that to derive T with polarity p, it suffices to assume T with opposite polarity ($\bar{p}$), and then derive a contradiction by separately proving and refuting the same type T'. The construct $\mathbf{halt}\ T$ is included to signal the final result of a computation. Since *a priori* we allow computations to produce final results of any type T, we allow $\mathbf{halt}\ T$ to be typed negatively – which can be thought of as indicating that $\mathbf{halt}\ T$ is an output port – for any type T. Later (Section 10), we will see how to restrict T to ensure canonical final results.

The final three rules are for recursive types. The MU rule is a standard folding rule for recursive types. The MUBAR and RECCALL rules work together to type **rec**-terms. MUBAR and MU use a helper judgment $\mathbf{OccursOnly}\ +\ X\ T$ to express that the μ-bound variable X only occurs positively in the body T of $\mu\, X.T$. This judgment is defined in Figure 2. The idea expressed by MUBAR is that to type $\mathbf{rec}\, x[y = t_1].t_2$, we must first assign some type T', with some polarity p, to the accumulator t_1. Then we check that the body t_2 of the recursor has type T negatively, in an extended context. The type T will in general contain the μ-bound type variable X, so by saying that t_2 must have type T, we are

$$\frac{}{\Gamma_1, x : p\ T, \Gamma_2 \vdash x : p\ T}\ \textsc{Ax}$$

$$\frac{\Gamma \vdash t_1 : +\ T_1 \qquad \Gamma \vdash t_2 : +\ T_2}{\Gamma \vdash (t_1, t_2) : +\ T_1 \wedge T_2}\ \textsc{AndPos}$$

$$\frac{\Gamma, x : +\ T_1 \vdash t : -\ T_2}{\Gamma \vdash \iota\, x.t : -\ T_1 \wedge T_2}\ \textsc{AndNeg}$$

$$\frac{\Gamma \vdash t : \bar{p}\ T}{\Gamma \vdash \mathbf{not}\ t : p\ \neg T}\ \textsc{Not}$$

$$\frac{\Gamma, x : \bar{p}\ T \vdash t_1 : +\ T' \quad \Gamma, x : \bar{p}\ T \vdash t_2 : -\ T'}{\Gamma \vdash \delta\, x.t_1 \bullet t_2 : p\ T}\ \textsc{Cut}$$

$$\frac{}{\Gamma \vdash \mathbf{halt}\ T : -\ T}\ \textsc{Halt}$$

$$\frac{\mathbf{OccursOnly}\ +\ X\ T \quad \Gamma \vdash t : +\ [\mu\, X.T/X]T}{\Gamma \vdash t : +\ \mu\, X.T}\ \textsc{Mu}$$

$$\frac{\begin{array}{l}\mathbf{OccursOnly}\ +\ X\ T \\ \Gamma \vdash t_1 : p\ T' \\ \Gamma, x : p\ X \triangleright T', y : p\ T' \vdash t_2 : -\ T\end{array}}{\Gamma \vdash \mathbf{rec}\, x[y = t_1].t_2 : -\ \mu\, X.T}\ \textsc{MuBar}$$

$$\frac{x : p\ X \triangleright T' \in \Gamma \qquad \Gamma \vdash t : p\ T'}{\Gamma \vdash x[t] : -\ X}\ \textsc{RecCall}$$

Figure 1. Typing rules for RP-DC

$$\frac{}{\mathbf{OccursOnly}\ +\ X\ X}\ \textsc{OccVar}$$

$$\frac{\mathbf{OccursOnly}\ p\ X\ T_1 \quad \mathbf{OccursOnly}\ p\ X\ T_2}{\mathbf{OccursOnly}\ p\ X\ T_1 \wedge T_2}\ \textsc{OccAnd}$$

$$\frac{\mathbf{OccursOnly}\ \bar{p}\ X\ T}{\mathbf{OccursOnly}\ p\ X\ \neg T}\ \textsc{OccNot}$$

$$\frac{\mathbf{OccursOnly}\ p\ X\ T}{\mathbf{OccursOnly}\ p\ X\ \mu\, X'.T}\ \textsc{OccMu}$$

Figure 2. Definition of **OccursOnly**

saying that at certain positions in t_2, we must have terms of this unknown type X. The only way to form such terms is using the term construct $x[t]$ for recursive calls. This is where the extended context comes into play. MuBar extends the context with a variable $y : p\ T'$ referring to the current value of the accumulator, and an assumption $x : p\ X \triangleright T'$. Then in the RecCall rule, we use such an assumption to assign type X negatively to a recursive call $x[t]$, as long as t is again a suitable value for the iterator (i.e., it must have type T' with polarity p). The idea of using type abstraction (here, the unknown type X) to restrict the form of recursion is due to Mendler (see Chapter 3 of [14]).

4. Reduction

The reduction judgment for RP-DC is $c \rightsquigarrow c'$, where c and c' are *configurations* of the form $p\ t_1 \bullet t_2$. The intention is to reduce such a configuration c when t_1 has type T positively, and t_2 has type T negatively, for some type T. Thus, a configuration can be viewed as a (signed) top-level cut, and we call T the cut-type of the configuration. Note that reduction does not preserve the cut-type in general: when $c \rightsquigarrow c'$, the cut-type of c need not be the same c'.

The intention is that to evaluate a term t of type positive T (that is, a term which has type T with positive polarity), where that term does not contain any **halt** term, we will reduce the configuration $+\ t \bullet \mathbf{halt}\ T$. So $\mathbf{halt}\ T$ is the final continuation for receiving a result.

We elect to make reduction deterministic, since full non-deterministic reduction is nonconfluent, due to the well-known peak where t_1 and t_2 are both cuts. We use the polarity $p \in \{+, -\}$ to control the reduction order of the next reduction step starting from c. The side of the configuration which the polarity indicates should be reduced to a value is called the *active* side of the configuration; the other side is the *passive* side. If p is positive, we will not substitute a subterm of the left part of the configuration unless it is a value. If p is negative, we impose a similar restriction before substituting subterms of the right part of the configuration. Reduction will proceed left to right in the case of pairs (t, t'). We define the structural terms:

$$\textit{structural}\ s\ ::=\ \mathbf{halt}\ T \mid \delta\, x.t \bullet t'$$

We also use q as a meta-variable for terms which are not structural. The definition of values v and nonvalues n are given below (Section 4.1).

The reduction rules for RP-DC are given in Figure 3, in four groups. The first two rules in the figure (AnaAnd and AnaNot) are called *analysis* rules: they break down compound structures that are cut against each other at the top level. The next five rules (AndP1, AndP2, AndN, NotP, and NotN) we call *marshalling* rules. They shift parts of the term from the active side to the passive side, when the passive side is structural. Marshalling rules reveal cuts (i.e., δ-terms) which should be reduced first for the active side to reach a value. The **Nonvalue** premise in the AndN rule is defined in Figure 5, and discussed in detail below (Section 4.1). The next four rules are δ-rules, specifying how a configuration involving a cut should be reduced. A value restriction is imposed there: if the passive side of the configuration is a cut, then the active side must be a value. Standard capture-avoiding substitution $[t/x]t'$ is used here.

The final rule gives the reduction semantics for the recursor. The rule makes use of both the usual notion of substitution, and a special form $[t/x]_{\mathbf{rec}}\, t'$, defined in Figure 4. The crucial clause of Figure 4 is the last one, which updates the accumulator in a **rec**-term that is being substituted. We will see detailed examples of how this works below (Section 6).

We call a configuration $p\ t_1 \bullet t_2$ *irreducible* iff for all configurations c, it is not the case that $p\ t_1 \bullet t_2 \rightsquigarrow c$.

4.1 Values and nonvalues

The notion of value v which we require for purposes of canonicity is somewhat more complex than usual. We inductively define a relation $\mathbf{Nonvalue}[\theta]\ t$, where θ is either a variable x which restricts the application of the RaiseLa rule, or else !, meaning no restriction. The definition is in Figure 5. The rules in the figure which are recursive always refer to a strict subterm of the term in

$\boxed{[t'/x]_{\mathbf{rec}}\, t''}$

$$
\begin{array}{l}
[t'/x]_{\mathbf{rec}} x \equiv t' \\
[t'/x]_{\mathbf{rec}} y \equiv y \quad \textit{if } x \not\equiv y \\
[t'/x]_{\mathbf{rec}} (t_1, t_2) \equiv ([t'/x]_{\mathbf{rec}} t_1, [t'/x]_{\mathbf{rec}} t_2) \\
[t'/x]_{\mathbf{rec}} \iota\, y.t \equiv \iota\, y.[t'/x]_{\mathbf{rec}} t \\
[t'/x]_{\mathbf{rec}} \mathbf{not}\; t \equiv \mathbf{not}\; [t'/x]_{\mathbf{rec}} t \\
[t'/x]_{\mathbf{rec}} \delta\, y.t_1 \bullet t_2 \equiv \delta\, y.[t'/x]_{\mathbf{rec}} t_1 \bullet [t'/x]_{\mathbf{rec}} t_2 \\
[t'/x]_{\mathbf{rec}} \mathbf{rec}\, x'[y = t_1].t_2 \equiv \mathbf{rec}\, x'[y = [t'/x]_{\mathbf{rec}} t_1].[t'/x]_{\mathbf{rec}} t_2 \\
[t'/x]_{\mathbf{rec}} y[t] \equiv y[[t'/x]_{\mathbf{rec}} t] \quad \textit{if } x \not\equiv y \\
[\mathbf{rec}\, x[y = t].t'/x]_{\mathbf{rec}} x[t''] \equiv \mathbf{rec}\, x[y = t''].t'
\end{array}
$$

Figure 4. Special substitution for **rec**-terms

$$\frac{}{p\ (t_1, t_2) \bullet \iota\, x.t \rightsquigarrow p\ t_1 \bullet \delta\, x.t_2 \bullet t}\ \textsc{AnaAnd}$$

$$\frac{}{p\ \mathbf{not}\ t \bullet \mathbf{not}\ t' \rightsquigarrow \bar{p}\ t' \bullet t}\ \textsc{AnaNot}$$

$$\frac{}{+\ (n, t) \bullet s \rightsquigarrow +\ n \bullet \delta\, x.(x, t) \bullet s}\ \textsc{AndP1}$$

$$\frac{}{+\ (v, n) \bullet s \rightsquigarrow +\ n \bullet \delta\, x.(v, x) \bullet s}\ \textsc{AndP2}$$

$$\frac{\begin{array}{l} t = \delta\, z.s \bullet \iota\, x.\delta\, x'.x \bullet z \\ \mathbf{Nonvalue}[x]\ n \end{array}}{-\ s \bullet \iota\, x.n \rightsquigarrow -\ (\delta\, z.s \bullet \iota\, x.z) \bullet [t/x]n}\ \textsc{AndN}$$

$$\frac{}{+\ \mathbf{not}\ n \bullet s \rightsquigarrow -\ \delta\, y.\mathbf{not}\ y \bullet s \bullet n}\ \textsc{NotP}$$

$$\frac{}{-\ s \bullet \mathbf{not}\ n \rightsquigarrow +\ n \bullet \delta\, y.s \bullet \mathbf{not}\ y}\ \textsc{NotN}$$

$$\frac{}{+\ (\delta\, y.t_1 \bullet t_2) \bullet t \rightsquigarrow +\ [t/y]t_1 \bullet [t/y]t_2}\ \textsc{LP}$$

$$\frac{}{-\ (\delta\, y.t_1 \bullet t_2) \bullet v \rightsquigarrow -\ [v/y]t_1 \bullet [v/y]t_2}\ \textsc{LN}$$

$$\frac{}{+\ v \bullet (\delta\, y.t_1 \bullet t_2) \rightsquigarrow +\ [v/y]t_1 \bullet [v/y]t_2}\ \textsc{RP}$$

$$\frac{}{-\ t \bullet (\delta\, y.t_1 \bullet t_2) \rightsquigarrow -\ [t/y]t_1 \bullet [t/y]t_2}\ \textsc{RN}$$

$$\frac{}{\begin{array}{l} p\ q \bullet \mathbf{rec}\, x[y = t_1].t_2 \rightsquigarrow \\ p\ q \bullet [\mathbf{rec}\, x[y = t_1].t_2/x]_{\mathbf{rec}}[t_1/y]t_2 \end{array}}\ \textsc{Rec}$$

Figure 3. Reduction rules for RP-DC, where v denotes values, n nonvalues, s structural terms, and q non-structural terms.

the conclusion. So it is decidable whether or not $\mathbf{Nonvalue}[\theta]\ t$ holds, for any t. So we define the set of nonvalues n as those terms for which $\mathbf{Nonvalue}[!]\ n$ holds, and the set of values v as those for which $\mathbf{Nonvalue}[!]\ v$ does not hold. A positive characterization of values should be possible, but working it out is left for future work. Note that in the CCut rule, I am writing o as a meta-variable for constructor terms:

$$\textit{constructor terms } o\ ::=\ (t_1, t_2) \mid \iota\, x.t \mid \mathbf{not}\ t$$

Now let us further consider the motivation for the definition of $\mathbf{Nonvalue}$.

The marshalling rules are responsible for turning nonvalues in the active part of a configuration into values. On the one hand, we wish to expose redexes in those nonvalues, to reduce them to values. But because canonicity fails in RP-DC in general (see Section 10), we have to be careful to avoid nontermination when performing marshalling. For example, suppose we have a value v which does not contain y free. Then it is critical that we judge $\iota\, x.\delta\, y.x \bullet \mathbf{not}\ v$ to be a value, since otherwise we would get this reduction, where the first step is by AndN:

$$
\begin{array}{l}
-\ s \bullet \iota\, x.\delta\, y.x \bullet \mathbf{not}\ v \ \rightsquigarrow \\
-\ (\delta\, z.s \bullet \iota\, x.z) \bullet \delta\, y.(\delta\, z.s \bullet \iota\, x.\delta\, x'.x \bullet z) \bullet \mathbf{not}\ v \ \rightsquigarrow \\
-\ (\delta\, z.s \bullet \iota\, x.\delta\, x'.x \bullet z) \bullet \mathbf{not}\ v \ \rightsquigarrow \\
-\ s \bullet \iota\, x.\delta\, x'.x \bullet \mathbf{not}\ v
\end{array}
$$

The final configuration is α-equivalent to the first one, so we would diverge. But with the definition of Figure 5, we will not be able to derive $\mathbf{Nonvalue}[!]\ \iota\, x.\delta\, y.x \bullet \mathbf{not}\ v$, because of the restriction in the RaiseLa rule. So we will consider it a value, and hence not perform any marshalling reduction for it.

A few final notes on values. It may seem strange to define the recursor as a value. We will certainly use it to consume inductive data, where it would not matter if we viewed it as a value. But we can also view the recursor as a finitary representation of coinductive data, and there we need to view it as a value to avoid unfolding it infinitely. Also, unlike in DC, we only consider **not**-term of the form $\mathbf{not}\ v$ to be values, as opposed (in DC) to $\mathbf{not}\ t$. This allows us to give the expected reduction semantics for terms of defined types like disjunctions and implications (see Section 5 below).

5. Defined Propositional Types

Let us see how propositional types and their term constructs can be defined in RP-DC, and how the expected typings and reductions hold for them.

$$\frac{}{\mathbf{Nonvalue}[\theta]\ \delta\, x.o \bullet o'}\ \textsc{CCut}$$

$$\frac{}{\mathbf{Nonvalue}[\theta]\ \delta\, x.(\delta\, y.t_1 \bullet t_2) \bullet t}\ \textsc{NCutL}$$

$$\frac{}{\mathbf{Nonvalue}[\theta]\ \delta\, x.t \bullet (\delta\, y.t_1 \bullet t_2)}\ \textsc{NCutR}$$

$$\frac{x \notin \mathsf{FV}(t)}{\mathbf{Nonvalue}[\theta]\ \delta\, x.t \bullet y}\ \textsc{RaiseR}$$

$$\frac{\begin{array}{c}x \notin \mathsf{FV}(t)\\ y \neq y'\end{array}}{\mathbf{Nonvalue}[y]\ \delta\, x.y' \bullet t}\ \textsc{RaiseLa}$$

$$\frac{x \notin \mathsf{FV}(t)}{\mathbf{Nonvalue}[!]\ \delta\, x.y' \bullet t}\ \textsc{RaiseLb}$$

$$\frac{x \notin \mathsf{FV}(t)}{\mathbf{Nonvalue}[\theta]\ \delta\, x.t \bullet \mathbf{halt}\ T}\ \textsc{HaltR}$$

$$\frac{\begin{array}{c}\mathbf{Nonvalue}[!]\ t_d\\ d \in \{1,2\}\end{array}}{\mathbf{Nonvalue}[\theta]\ (t_1, t_2)}\ \textsc{NAndP}$$

$$\frac{\mathbf{Nonvalue}[y]\ t}{\mathbf{Nonvalue}[\theta]\ \iota\, y.t}\ \textsc{NAndN}$$

$$\frac{\mathbf{Nonvalue}[!]\ t}{\mathbf{Nonvalue}[\theta]\ \mathbf{not}\ t}\ \textsc{NNot}$$

$$\frac{\begin{array}{c}\mathbf{Nonvalue}[!]\ t_d\\ d \in \{1,2\}\end{array}}{\mathbf{Nonvalue}[\theta]\ \delta\, x.t_1 \bullet t_2}\ \textsc{NCut}$$

Figure 5. Specifying which terms are nonvalues. The metavariable o in CCUT ranges over terms of the form (t_1, t_2), $\iota\, x.t$, or $\mathbf{not}\ t$.

5.1 Disjunction

Recall that in Section 2, we defined the type $T_1 \vee T_2$ to be $\neg(\neg T_1 \wedge \neg T_2)$. Term constructs for disjunctions can be defined as follows:

$$\begin{array}{lcl}\mathbf{in}_1 t & := & \mathbf{not}\ \iota\, x.\delta\, y.x \bullet \mathbf{not}\ t\\ \mathbf{in}_2 t & := & \mathbf{not}\ \iota\, x.\mathbf{not}\ t\\ [t, t'] & := & \mathbf{not}\ (\mathbf{not}\ t_1, \mathbf{not}\ t_2)\end{array}$$

We assume that variables are renamed to avoid capture, so in particular, x and y are not free in t in the first two definitions. Those first two term constructs are called *injections*, and the last is for *co-pairs*. They can be typed with the derivations in Figure 6, where Γ' abbreviates $\Gamma, x : +\neg T_1, y : +\neg T_2$. The top-most inference in the first two derivations is by an admissible weakening rule.

We have the (derived) analytic reductions of Figure 7, to reduce top-level cuts involving these constructs for disjunction. There are also marshalling reductions, such as the example shown in Figure 8. There, we see a situation where we have a nonvalue n for the body of an injection $\mathbf{in}_1 n$, where that injection is the active term of a configuration. Reduction will raise n to be the active term of the configuration, thus exposing it to reduction before proceeding with a computation involving the injection. There are similar

$$\frac{\dfrac{}{\Gamma' \vdash x : +\neg T_1} \quad \dfrac{\dfrac{\Gamma \vdash t : +T_1}{\Gamma' \vdash t : +T_1}}{\Gamma' \vdash \mathbf{not}\ t : -\neg T_1}}{\dfrac{\dfrac{\Gamma, x : +\neg T_1 \vdash \delta\, y.x \bullet \mathbf{not}\ t : -\neg T_2}{\Gamma \vdash \iota\, x.\delta\, y.x \bullet \mathbf{not}\ t : -\neg T_1 \wedge \neg T_2}}{\Gamma \vdash \mathbf{in}_1 t : +T_1 \vee T_2}}$$

$$\frac{\dfrac{\dfrac{\Gamma \vdash t : +T_2}{\Gamma, x : +\neg T_1 \vdash t : +T_2}}{\dfrac{\Gamma, x : +\neg T_1 \vdash \mathbf{not}\ t : -\neg T_2}{\Gamma \vdash \iota\, x.\mathbf{not}\ t : -\neg T_1 \wedge \neg T_2}}}{\Gamma \vdash \mathbf{in}_2 t : +T_1 \vee T_2}$$

$$\frac{\dfrac{\Gamma \vdash t_1 : -T_1}{\Gamma \vdash \mathbf{not}\ t_1 : +\neg T_1} \quad \dfrac{\Gamma \vdash t_2 : -T_2}{\Gamma \vdash \mathbf{not}\ t_2 : +\neg T_2}}{\dfrac{\Gamma \vdash (\mathbf{not}\ t_1, \mathbf{not}\ t_2) : +\neg T_1 \wedge \neg T_2}{\Gamma \vdash [t_1, t_2] : -T_1 \vee T_2}}$$

Figure 6. Typing derivations for disjunction, where Γ' abbreviates $\Gamma, x : +\neg T_1, y : +\neg T_2$.

$$\begin{array}{l}+\ \mathbf{in}_1 t \bullet [t_1, t_2] \ \equiv\\ +\ \mathbf{not}\ \iota\, x.\delta\, y.x \bullet \mathbf{not}\ t \bullet \mathbf{not}\ (\mathbf{not}\ t_1, \mathbf{not}\ t_2) \ \rightsquigarrow\\ -\ (\mathbf{not}\ t_1, \mathbf{not}\ t_2) \bullet \iota\, x.\delta\, y.x \bullet \mathbf{not}\ t \ \rightsquigarrow\\ -\ \mathbf{not}\ t_1 \bullet \delta\, x.\mathbf{not}\ t_2 \bullet \delta\, y.x \bullet \mathbf{not}\ t \ \rightsquigarrow\\ -\ \mathbf{not}\ t_2 \bullet \delta\, y.\mathbf{not}\ t_1 \bullet \mathbf{not}\ t \ \rightsquigarrow\\ -\ \mathbf{not}\ t_1 \bullet \mathbf{not}\ t \ \rightsquigarrow\\ +\ t \bullet t_1\end{array}$$

$$\begin{array}{l}+\ \mathbf{in}_2 t \bullet [t_1, t_2] \ \equiv\\ +\ \mathbf{not}\ \iota\, x.\mathbf{not}\ t \bullet \mathbf{not}\ (\mathbf{not}\ t_1, \mathbf{not}\ t_2) \ \rightsquigarrow\\ -\ (\mathbf{not}\ t_1, \mathbf{not}\ t_2) \bullet \iota\, x.\mathbf{not}\ t \ \rightsquigarrow\\ -\ \mathbf{not}\ t_1 \bullet \delta\, x.\mathbf{not}\ t_2 \bullet \mathbf{not}\ t \ \rightsquigarrow\\ -\ \mathbf{not}\ t_2 \bullet \mathbf{not}\ t \ \rightsquigarrow\\ +\ t \bullet t_2\end{array}$$

Figure 7. Analytic reductions for derived term constructs for disjunction

$$\begin{array}{l}+\ \mathbf{in}_1 n \bullet s \ \equiv\\ +\ \mathbf{not}\ \iota\, x.\delta\, y.x \bullet \mathbf{not}\ n \bullet s \ \rightsquigarrow\\ -\ (\delta\, y'.\mathbf{not}\ y' \bullet s) \bullet \iota\, x.\delta\, y.x \bullet \mathbf{not}\ n \ \rightsquigarrow\\ -\ (\delta\, z.(\delta\, y'.\mathbf{not}\ y' \bullet s) \bullet \iota\, x.z) \bullet\\ \quad \delta\, y.(\delta\, z.(\delta\, y'.\mathbf{not}\ y' \bullet s) \bullet \iota\, x.\delta\, x'.x \bullet z) \bullet \mathbf{not}\ n \ \rightsquigarrow\\ -\ (\delta\, z.(\delta\, y.\mathbf{not}\ y \bullet s) \bullet \iota\, x.\delta\, x'.x \bullet z) \bullet \mathbf{not}\ n \ \rightsquigarrow\\ +\ n \bullet (\delta\, z'.(\delta\, z.(\delta\, y.\mathbf{not}\ y \bullet s) \bullet \iota\, x.\delta\, x'.x \bullet z) \bullet \mathbf{not}\ z')\end{array}$$

Figure 8. Sample marshalling reduction for injections

marshalling reductions for when the injection is $\mathbf{in}_2 n$, and when it is a co-pair (in a configuration with negative polarity).

Finally, we can observe that $\mathbf{in}_d v$ and $[v_1, v_2]$ are values. In the case of $\mathbf{in}_1 v$, for example, we have

$$\mathbf{not}\ \iota\, x.\delta\, y.x \bullet \mathbf{not}\ v$$

We can confirm that this term is syntactically a value (assuming $x \notin \mathbf{FV}(v)$), by reasoning contrapositively with the rules of Figure 5.

$$\dfrac{\dfrac{\dfrac{\Gamma, x :+ T_1 \vdash t :+ T_2}{\Gamma, x :+ T_1 \vdash \mathbf{not}\, t :- \neg T_2}}{\Gamma \vdash \iota\, x.\mathbf{not}\, t :- T_1 \wedge \neg T_2}}{\Gamma \vdash \lambda x.t :+ T_1 \to T_2}$$

$$\dfrac{\Gamma \vdash t_1 :+ T_1 \quad \dfrac{\Gamma \vdash t_2 :- T_2}{\Gamma \vdash \mathbf{not}\, t_2 :+ \neg T_2}}{\dfrac{\Gamma \vdash (t_1, \mathbf{not}\, t_2) :+ T_1 \wedge \neg T_2}{\Gamma \vdash \langle t_1, t_2 \rangle :- T_1 \to T_2}}$$

Figure 9. Typing derivations for implication

$$\begin{array}{l} +\ \lambda x.t \bullet \langle t_1, t_2 \rangle \ \equiv \\ +\ \mathbf{not}\, \iota\, x.\mathbf{not}\, t \bullet \mathbf{not}\, (t_1, \mathbf{not}\, t_2) \ \rightsquigarrow \\ -\ (t_1, \mathbf{not}\, t_2) \bullet \iota\, x.\mathbf{not}\, t \ \rightsquigarrow \\ -\ t_1 \bullet \delta\, x.\mathbf{not}\, t_2 \cdot \mathbf{not}\, t \ \rightsquigarrow \\ -\ \mathbf{not}\, t_2 \bullet \mathbf{not}\, [t_1/x]t \ \rightsquigarrow \\ +\ [t_1/x]t \bullet t_2 \end{array}$$

Figure 10. Analytic reduction for implication

5.2 Implication

In Section 2, we defined $T_1 \to T_2$ to be $\neg(T_1 \wedge \neg T_2)$. Term constructs for implication can be defined as follows:

$$\begin{array}{lcl} \lambda x.t & := & \mathbf{not}\, \iota\, x.\mathbf{not}\, t \\ \langle t_1, t_2 \rangle & := & \mathbf{not}\, (t_1, \mathbf{not}\, t_2) \end{array}$$

Figure 9 gives typing derivations for these. Figure 10 gives the analytic reduction sequence. Notice that we obtain call-by-name evaluation starting from a positive configuration. Cutting a positive term of type $T \to T'$ against a term $\langle t_1, t_2 \rangle$ which has that type negatively can be thought of as supplying both the positive input argument t_1 and the output continuation t_2 to the function.

Figure 11 gives an example of a marshalling reduction, where we pull a substitution instance of the nonvalue body of $\lambda x.n$ out of that λ-abstraction to the active position of the configuration. So as in normal-order evaluation for lambda calculus, reduction proceeds in this case into the body of the λ-abstraction. This is less strange than might first appear: because **rec**-terms are values, evaluation in the body of a recursive function will typically quickly stop, when it encounters the recursion.

Calling a function is, of course, a basic operation of functional programming, and it is convenient to define the standard notation for applications:

$$t\ t' \ := \ \delta\, x.t \cdot \langle t', x \rangle$$

To apply a function t of positive type $T \to T'$ to an argument t' of positive type T, we introduce a variable x of negative type T', to hold the output of the function, and then we cut the function against $\langle t', x \rangle$.

5.3 Classical principles

We can easily derive the strictly classical principle $\neg\neg T \to T$ for any type T:

$$\mathbf{dne} \ := \ \lambda x.\delta\, y.x \cdot \mathbf{not}\, \mathbf{not}\, y$$

Here, x has type $+\neg\neg T$, and y has type $-T$. So then $\mathbf{not}\, \mathbf{not}\, y$ has type $-\neg\neg T$, as required for the cut-term.

$$\begin{array}{l} +\ \lambda x.n \bullet s \ \equiv \\ +\ \mathbf{not}\, \iota\, x.\mathbf{not}\, n \bullet s \ \rightsquigarrow \\ -\ \delta\, y.\mathbf{not}\, y \cdot s \bullet \iota\, x.\mathbf{not}\, n \ \rightsquigarrow \\ -\ \delta\, y'.(\delta\, y.\mathbf{not}\, y \cdot s) \cdot \iota\, x.y' \bullet \\ \quad \mathbf{not}\, [\delta\, y'.(\delta\, y.\mathbf{not}\, y \cdot s) \cdot \iota\, x.\delta\, y''.x \cdot y'/x]n \ \rightsquigarrow \\ +\ [\delta\, y'.(\delta\, y.\mathbf{not}\, y \cdot s) \cdot \iota\, x.\delta\, y''.x \cdot y'/x]n \bullet \\ \quad \delta\, y''.(\delta\, y'.(\delta\, y.\mathbf{not}\, y \cdot s) \cdot \iota\, x.y') \cdot \mathbf{not}\, y'' \end{array}$$

Figure 11. Sample marshalling reduction for implication

The law of the excluded middle is also easily derivable in RP-DC:

$$\mathbf{lem} \ := \ \mathbf{not}\, \iota\, x.\mathbf{not}\, x$$

This term can be assigned type $T \vee \neg T$ for any type T, since unfolding the definition of disjunction we see that type is equal to $\neg(\neg T \wedge \neg\neg T)$. Typing **lem** will assign the type $\neg T$ positively to x, and so $\mathbf{not}\, x$ has the desired type $\neg\neg T$ negatively, as required to type a ι-term.

It is well known that if disjunction is defined in terms of negation (as here), the law of excluded middle is intuitionistically valid. But as pointed out by O'Connor, with that definition, the principle which is not intuitionistically valid is the usual principle of disjunction elimination [15]. So to further highlight the classical character of RP-DC, let us see how to derive that principle in RP-DC. The following defines a term construct for disjunction elimination:

$$\mathbf{case}\, t\, \mathbf{of}\, x.t_1, y.t_2 \ := \ \delta\, z.t \cdot [\delta\, x.t_1 \cdot z, \delta\, y.t_2 \cdot z]$$

Figure 12 gives the typing derivation for this construct. We are using our derived co-pairing construct to inhabit the disjunctive type $T_1 \vee T_2$ negatively.

5.4 False and True

Above we defined $\bot$ to be $\mu\, X.X$. This can be inhabited negatively by the term **false** defined by

$$\mathbf{false} := \mathbf{rec}\, x[y = \mathbf{dne}].x[\mathbf{dne}]$$

Then $\top$ can be easily inhabited by

$$\mathbf{true} := \mathbf{not}\, \mathbf{false}$$

5.5 Control

As is also done for other CCTTs, we can define a mechanism for exceptions this way:

$$\begin{array}{lcl} \mathbf{catch}\, x\ t_1\ t_1'\ t_2 & := & \delta\, y.(\delta\, x.t_1 \cdot t_1') \cdot t_2 \\ \mathbf{throw}\, t\, x & := & \delta\, y.t \cdot x \end{array}$$

This is very similar to what is done in the $\lambda\mu$-calculus; for example, in [6]. In $\mathbf{catch}\, x\ t_1\ t_1'\ t_2$, we declare exception x, which can be thrown inside the cut of t_1 against t_1'. The exception will be handled in t_2. The term $\underline{\mathbf{throw}\, t\, x}$ can be thought of as throwing t on named exception-channel x. For a small example of these in action, consider the term

$$\mathbf{catch}\, x\, (\lambda y.y)\, \langle \mathbf{throw}\, t\, x, t'' \rangle\, t_2$$

The idea here is that we want to call the identity function $\lambda y.y$. On normal execution we will send the result to t'', but if the exception on x is raised (as it will be), it should be handled by t_2. Unfolding some definitions, this term is equal to

$$\delta\, y.(\delta\, x.(\lambda y.y) \cdot \langle \delta\, z.t \cdot x, t'' \rangle) \cdot t_2$$

To run this term, let us suppose we are cutting it against $\mathbf{halt}\ T$. See Figure 13 for the reduction, where t_2' abbreviates $[\mathbf{halt}\ T/y]t_2$.

$$\dfrac{\Gamma' \vdash t : + T_1 \vee T_2 \qquad \dfrac{\dfrac{\dfrac{\Gamma, x : + T_1 \vdash t_1 : + T'}{\Gamma_1 \vdash t_1 : + T'} \quad \dfrac{}{\Gamma_1 \vdash z : - T'}}{\Gamma' \vdash \delta\, x.t_1 \bullet z : - T_1} \quad \dfrac{\dfrac{\dfrac{\Gamma, y : + T_2 \vdash t_2 : + T'}{\Gamma_2 \vdash t_2 : + T'} \quad \dfrac{}{\Gamma_2 \vdash z : - T'}}{\Gamma' \vdash \delta\, y.t_2 \bullet z : - T_2}}{\Gamma' \vdash [\delta\, x.t_1 \bullet z, \delta\, y.t_2 \bullet z] : - T_1 \vee T_2}}{\Gamma \vdash \mathbf{case}\, t\, \mathbf{of}\, x.t_1, y.t_2 : + T'}$$

Figure 12. Typing derivation for the derived disjunction elimination, where Γ' abbreviates $\Gamma, z : - T'$, Γ_1 abbreviates $\Gamma', x : + T_1$, and Γ_2 abbreviates $\Gamma', y : + T_2$.

$$\begin{array}{ll}
& + \delta\, y.(\delta\, x.(\lambda y.y) \bullet \langle \delta\, z.t \bullet x, t'' \rangle) \bullet t_2 \bullet \mathbf{halt}\ T \ \rightsquigarrow \\
& + (\delta\, x.(\lambda y.y) \bullet \langle \delta\, z.t \bullet x, t'' \rangle) \bullet t_2' \ \rightsquigarrow \\
& + (\lambda y.y) \bullet \langle \delta\, z.t \bullet t_2', t'' \rangle \ \rightsquigarrow^* \\
& + \delta\, z.t \bullet t_2' \bullet t'' \ \rightsquigarrow \\
& + t \bullet t_2'
\end{array}$$

Figure 13. Reduction illustrating the behavior of defined exceptions

Besides exceptions, we can easily mimic the control operator `call/cc`: it is a trivial matter to capture the passive part of a configuration by using a cut-term for the active part. It is not so clear, however, that delimited control can be implemented directly in RP-DC. This is because a cut-term like $\delta\, x.t_1 \bullet t_2$, if used positively, introduces a name x for the *entire* rest of the computation (the entire passive part of the configuration). Capturing the continutation of a term only up to some delimiting point, in contrast, is not obviously implementable.

6. Examples with Recursion

Let us now consider how to implement standard inductive types in RP-DC, specifically unary natural numbers and lists. A full-fledged type theory based on RP-DC would include some form of polymorphism, but for simplicity we are focused here just on a monomorphic language. So we will just consider monomorphic lists, and rely on parametrized meta-level definitions to describe lists and list operations for different types of data. We make these standard definitions:

$$\begin{array}{lcl}
\mathbb{L}\, A & := & \mu\, X.\top \vee (A \wedge X) \\
\mathbb{N} & := & \mathbb{L}\, \top
\end{array}$$

$\mathbb{L}\ A$ is the type for homogeneous lists of elements of type A. We are using the well-known fact that one can define unary numbers as lists containing just a single unit value, here $\mathbf{true}$. We can define constructors for lists:

$$\begin{array}{lcl}
\mathbf{nil} & := & \mathbf{in}_1 \mathbf{true} \\
\mathbf{cons} & := & \lambda x.\lambda y.\mathbf{in}_2(x, y)
\end{array}$$

These can be typed as $\mathbb{L}\ A$ and $A \to \mathbb{L}\ A \to \mathbb{L}\ A$, respectively. Let us also define this notation (familiar from Haskell, for example) for lists of length $n \geq 0$:

$$[t_1, \ldots, t_n] \ := \ \mathbf{in}_2\,(t_1, \ldots \mathbf{in}_2\,(t_n, \mathbf{in}_1 \mathbf{true}))$$

Note that $[v_1, \ldots, v_k]$ is a syntactic value (as it would not be if we defined it using $\mathbf{cons}$ and $\mathbf{nil}$). The numerals, which we will use with standard decimal notation below, can be constructed by:

$$\begin{array}{lcl}
\mathbf{Z} & := & \mathbf{nil} \\
\mathbf{Suc} & := & \lambda x.\mathbf{cons}\ \mathbf{true}\ x
\end{array}$$

$$\begin{array}{ll}
& + \mathbf{append}\ [1,2]\ [3] \bullet \mathbf{halt}\, \mathbb{L}\, \mathbb{N} \ \rightsquigarrow^* \\
& + \mathbf{in}_2(1, [2]) \bullet \\
& \quad \mathbf{rec}\, f[z = \mathbf{halt}\, \mathbb{L}\, \mathbb{N}]. \\
& \quad (\delta\, y'.[3] \bullet z, \iota\, a.f[\delta\, y'.\mathbf{cons}\ a\ y' \bullet z]) \ \rightsquigarrow \\
& + \mathbf{in}_2(1, [2]) \bullet \\
& \quad (\delta\, y'.[3] \bullet \mathbf{halt}\, \mathbb{L}\, \mathbb{N}, \\
& \quad \iota\, a.\mathbf{rec}\, f[z = \delta\, y'.\mathbf{cons}\ a\ y' \bullet \mathbf{halt}\, \mathbb{L}\, \mathbb{N}]. \\
& \quad (\delta\, y'.[3] \bullet z, \iota\, a.f[\delta\, y'.\mathbf{cons}\ a\ y' \bullet z])) \ \rightsquigarrow^* \\
& + (1, [2]) \bullet \\
& \quad \iota\, a.\mathbf{rec}\, f[z = \delta\, y'.\mathbf{cons}\ a\ y' \bullet \mathbf{halt}\, \mathbb{L}\, \mathbb{N}]. \\
& \quad (\delta\, y'.[3] \bullet z, \iota\, a.f[\delta\, y'.\mathbf{cons}\ a\ y' \bullet z]) \ \rightsquigarrow^* \\
& + \mathbf{in}_2(2, \mathbf{nil}) \bullet \\
& \quad \mathbf{rec}\, f[z = \delta\, y'.\mathbf{cons}\ 1\ y' \bullet \mathbf{halt}\, \mathbb{L}\, \mathbb{N}]. \\
& \quad (\delta\, y'.[3] \bullet z, \iota\, a.f[\delta\, y'.\mathbf{cons}\ a\ y' \bullet z]) \ \rightsquigarrow^* \\
& + \mathbf{in}_2(2, \mathbf{nil}) \bullet \\
& \quad (\delta\, y'.[3] \bullet \delta\, y'.\mathbf{cons}\ 1\ y' \bullet \mathbf{halt}\, \mathbb{L}\, \mathbb{N}, \\
& \quad \iota\, a.\mathbf{rec}\, f[z = \delta\, y'.\mathbf{cons}\ a\ y' \bullet \delta\, y'.\mathbf{cons}\ 1\ y' \bullet \\
& \qquad \mathbf{halt}\, \mathbb{L}\, \mathbb{N}]. \\
& \quad (\delta\, y'.[3] \bullet z, \iota\, a.f[\delta\, y'.\mathbf{cons}\ a\ y' \bullet z])) \ \rightsquigarrow^* \\
& + \mathbf{nil} \bullet \\
& \quad \mathbf{rec}\, f[z = \delta\, y'.\mathbf{cons}\ 2\ y' \bullet \delta\, y'.\mathbf{cons}\ 1\ y' \bullet \\
& \qquad \mathbf{halt}\, \mathbb{L}\, \mathbb{N}]. \\
& \quad (\delta\, y'.[3] \bullet z, \iota\, a.f[\delta\, y'.\mathbf{cons}\ a\ y' \bullet z]) \ \rightsquigarrow^* \\
& + \mathbf{in}_1 \mathbf{true} \bullet \\
& \quad (\delta\, y'.[3] \bullet \delta\, y'.\mathbf{cons}\ 2\ y' \bullet \delta\, y'.\mathbf{cons}\ 1\ y' \bullet \\
& \qquad \mathbf{halt}\, \mathbb{L}\, \mathbb{N}, \ldots) \ \rightsquigarrow^* \\
& + [3] \bullet \delta\, y'.\mathbf{cons}\ 2\ y' \bullet \delta\, y'.\mathbf{cons}\ 1\ y' \bullet \\
& \qquad \mathbf{halt}\, \mathbb{L}\, \mathbb{N} \ \rightsquigarrow^* \\
& + \mathbf{cons}\ 2\ [3] \bullet \delta\, y'.\mathbf{cons}\ 1\ y' \bullet \mathbf{halt}\, \mathbb{L}\, \mathbb{N} \ \rightsquigarrow^* \\
& + [2, 3] \bullet \delta\, y'.\mathbf{cons}\ 1\ y' \bullet \mathbf{halt}\, \mathbb{L}\, \mathbb{N} \ \rightsquigarrow^* \\
& + \mathbf{cons}\ 1\ [2, 3] \bullet \mathbf{halt}\, \mathbb{L}\, \mathbb{N} \ \rightsquigarrow^* \\
& + [1, 2, 3] \bullet \mathbf{halt}\, \mathbb{L}\, \mathbb{N}
\end{array}$$

Figure 14. Normalizing evaluation of $\mathbf{append}$ on input lists $[1, 2]$ and $[3]$

With natural numbers defined this way, addition is just list append, so we will focus on defining this function for an example:

$$\begin{array}{l}
\mathbf{append} := \\
\lambda x.\lambda y.\delta\, r.x \bullet \\
\quad \mathbf{rec}\, f[z = r]. \\
\quad (\delta\, y'.y \bullet z, \iota\, a.f[\delta\, y'.\mathbf{cons}\ a\ y' \bullet z])
\end{array}$$

To see how this works, consider the reduction sequence in Figure 14, where we unfold definitions implicitly for conciseness. At each recursive call, we are extending the negative list in the accumulator of the $\mathbf{rec}$-term so that when it is cut against a positive list L, it will add the elements from the first input list ($[1, 2]$) in the correct order to L. In the base case, we supply the second input list ($[3]$) for L, and so get the desired final result.

7. Corecursion

Now let us see how to operate on coinductively defined data in RP-DC. Recall the definition of coinductive types from Section 2:

$$\nu\, X.T \;:=\; \neg\mu\, X.\neg[\neg X/X]\,T$$

First, let us consider this definition from the perspective of positivity. We only wish to allow coinductive types $\nu\, X.T$ when X occurs positively in T. Since our definition of this coinductive type begins $\neg\mu X.\neg$, the type $\neg\mu\, X.\neg\, T$ would not satisfy, in general, the requirement on inductive types that the μ-bound variable X must occur only positively in the body (since we are assuming X occurs only positively in T, but T occurs negatively in $\neg\mu\, X.\neg T$). It is for this reason that we take $\neg\mu\, X.\neg[\neg X/X]\,T$ as the definition of $\nu\, X.T$, since substituting $\neg X$ for X in T ensures that the positivity constraint on the μ-type is satisfied. As already noted, this is standard in propositional μ-calculus [12].

Based on this definition of ν-types, we can see that the natural term construct for co-recursion is the following, where we write $[\neg f/f]t_2$ for the term which is exactly like t_2 except that wherever t_2 has $f[t]$, $[\neg f/f]t_2$ has $\mathbf{not}\, f[t]$:

$$\mathbf{corec}\, f[z = t_1].t_2 \;:=\; \mathbf{not}\,\mathbf{rec}\, f[z = t_1].\mathbf{not}\,[\neg f/f]t_2$$

The typing derivation for this term is shown in Figure 15. The effect of the substitution of $\neg f$ for f in the body t_2 is that whenever t_2 makes a corecursive call $f[t]$, this expands to $\mathbf{not}\, f[t]$. Since a recursive call $f[t]$ has type negative X (according to the RecCall rule of Figure 1), the corecursive call $\mathbf{not}\, f[t]$ will instead have type positive $\neg X$.

Let us see an example now of these definitions in action. Streams are a canonical example of a coinductive type:

$$\mathbb{S}\; A \;:=\; \nu\, X.A \wedge X$$

Expanding the definition of ν-types, we see we have:

$$\mathbb{S}\; A \;=\; \neg\mu\, X.\neg(A \wedge \neg X)$$

We can get the head and tail of a stream as follows:

$$\begin{array}{lcl} \mathbf{head} & := & \lambda x.\delta\, y.x \bullet \mathbf{not}\,(\delta\, x'.(\mathbf{not}\,\iota\, y'.\delta\, z.y' \bullet y) \bullet x') \\ \mathbf{tail} & := & \lambda x.\delta\, y.x \bullet \mathbf{not}\,(\delta\, x'.(\mathbf{not}\,\iota\, y'.y) \bullet x') \end{array}$$

The first has positive type $\mathbb{S}\; A \to A$, and the second $\mathbb{S}\; A \to \mathbb{S}\; A$. For example, $\mathbf{tail}$ takes in x of positive type $\mathbb{S}\; A$, and immediately does a cut introducing variable y negatively of type $\mathbb{S}\; A$. We want to unfold the type of x from $\mathbb{S}\; A$ to $\neg\neg(A \wedge \mathbb{S}\; A)$, but there are two problems: the Mu typing rule can only be applied for folding, and it must be applied to a term of positive type $\neg(A \wedge \mathbb{S}\; A)$. We can solve both problems by cutting x against the term which begins with $\mathbf{not}$, and then does a cut introducing x' of negative type $\mu\, X.\neg(A \wedge \neg X)$. We are cutting that x' against $\mathbf{not}\,\iota\, y'.y$. The latter term has type $\neg(A \wedge \mathbb{S}\; A)$ positively, and by the Mu typing rule, can also be assigned type $\mu\, X.\neg(A \wedge \neg X)$, as required.

Let us consider some standard basic examples of computing streams. The following term has positive type $A \to \mathbb{S}\; A$ for any type A, and given x of type A, returns the infinite stream of x's:

$$\mathbf{repeat} \;:=\; \lambda x.\mathbf{corec}\, f[z = \mathbf{true}].(x, f[\mathbf{true}])$$

Expanding the definition of $\mathbf{corec}$, $\mathbf{repeat}$ is

$$\lambda x.\mathbf{not}\,\mathbf{rec}\, f[z = \mathbf{true}].\mathbf{not}\,(x, \mathbf{not}\, f[\mathbf{true}])$$

The accumulator z is not used here, so we just supply a trivial value $\mathbf{true}$ for it. To see that this indeed returns an infinite stream of 3s, for example, when called with 3, we can select an element from the result by cutting that result against a term of type negative $\mathbb{S}\; \mathbb{N}$. This is shown in Figure 16, where we assume z is a term of negative type $\mathbb{N}$. It receives the second element from the stream of 3s, when that element has been selected.

$$\begin{array}{l} +\, \mathbf{head}\,(\mathbf{tail}\,(\mathbf{repeat}\; 3)) \bullet z \;\rightsquigarrow^* \\ +\, \mathbf{repeat}\; 3 \bullet \\ \quad \mathbf{not}\,\mathbf{not}\,\iota\, x.\mathbf{not}\,\mathbf{not}\,\iota\, y.\delta\, x'.y \bullet z \;\rightsquigarrow^* \\ -\, \mathbf{not}\,\iota\, x.\mathbf{not}\,\mathbf{not}\,\iota\, y.\delta\, x'.y \bullet z \bullet \\ \quad \mathbf{rec}\, f[z = \mathbf{true}].\mathbf{not}\,(3, \mathbf{not}\, f[\mathbf{true}]) \;\rightsquigarrow \\ -\, \mathbf{not}\,\iota\, x.\mathbf{not}\,\mathbf{not}\,\iota\, y.\delta\, x'.y \bullet z \bullet \\ \quad \mathbf{not}\,(3, \mathbf{not}\,\mathbf{rec}\, f[z = \mathbf{true}].\mathbf{not}\,(3, \mathbf{not}\, f[\mathbf{true}])) \;\rightsquigarrow \\ +\, (3, \mathbf{not}\,\mathbf{rec}\, f[z = \mathbf{true}].\mathbf{not}\,(3, \mathbf{not}\, f[\mathbf{true}])) \bullet \\ \quad \iota\, x.\mathbf{not}\,\mathbf{not}\,\iota\, y.\delta\, x'.y \bullet z \;\rightsquigarrow \\ +\, \mathbf{not}\,\mathbf{rec}\, f[z = \mathbf{true}].\mathbf{not}\,(3, \mathbf{not}\, f[\mathbf{true}]) \bullet \\ \quad \mathbf{not}\,\mathbf{not}\,\iota\, y.\delta\, x'.y \bullet z \;\rightsquigarrow \\ -\, \mathbf{not}\,\iota\, y.\delta\, x'.y \bullet z \bullet \\ \quad \mathbf{rec}\, f[z = \mathbf{true}].\mathbf{not}\,(3, \mathbf{not}\, f[\mathbf{true}]) \;\rightsquigarrow \\ -\, \mathbf{not}\,\iota\, y.\delta\, x'.y \bullet z \bullet \\ \quad \mathbf{not}\,(3, \mathbf{not}\,\mathbf{rec}\, f[z = \mathbf{true}]. \\ \qquad \mathbf{not}\,(3, \mathbf{not}\, f[\mathbf{true}])) \;\rightsquigarrow \\ +\, (3, \mathbf{not}\,\mathbf{rec}\, f[z = \mathbf{true}].\mathbf{not}\,(3, \mathbf{not}\, f[\mathbf{true}])) \bullet \\ \quad \iota\, y.\delta\, x'.y \bullet z \;\rightsquigarrow \\ +\, \mathbf{not}\,\mathbf{rec}\, f[z = \mathbf{true}].\mathbf{not}\,(3, \mathbf{not}\, f[\mathbf{true}]) \bullet \\ \quad \delta\, x'.3 \bullet z \;\rightsquigarrow \\ +\, 3 \bullet z \end{array}$$

Figure 16. Selecting the second element (i.e., 3) from $\mathbf{repeat}\; 3$, where we assume $z : -\mathbb{N}$.

Using an accumulator, it is easy to define a stream consisting of the natural numbers beginning from some point n, another standard basic example:

$$\mathbf{nats} \;:=\; \lambda n.\mathbf{corec}\, f[x = n].(n, f[\mathbf{Suc}\; n])$$

This term has positive type $\mathbb{N} \to \mathbb{S}\; \mathbb{N}$. Selecting an element from a stream like that resulting from $\mathbf{nats}\; 0$ works similarly to the example of Figure 16.

For a final example, let us consider mapping a function f over a stream x:

$$\mathbf{map} \;:=\; \lambda f.\lambda x.\mathbf{corec}\, h[y = x].(f\;(\mathbf{head}\; y), h[\mathbf{tail}\; y])$$

The type of this term is the expected $(A \to B) \to (\mathbb{S}\; A \to \mathbb{S}\; B)$. Here we apply the $\mathbf{head}$ and $\mathbf{tail}$ functions above to inspect and modify the accumulator y, of positive type $\mathbb{S}\; A$.

8. Mixed Recursion/Corecursion

One of the motivations for recent works on type theories with induction and coinduction is to support mixed inductive/coinductive types, along with mixed recursion/corecursion. The current version of Agda, for example, supports mixed inductive/coinductive types only in the limited form $\nu X.\mu Y.T$. The reverse prefix $\mu X.\nu Y$ cannot be defined [1, 2]. Here we give two simple examples showing how RP-DC can support mixed inductive/coinductive types with either prefix.

8.1 The prefix $\nu X.\mu Y$

The example is finitely branching trees where every branch is infinitely deep. The type of such trees is:

$$\mathbb{I} \;:=\; \nu\, X.\mu\; Y.(X \wedge Y) \vee \top$$

$$\frac{\mathbf{OccursOnly}\ +\ X\ \neg[\neg X/X]T \quad \Gamma \vdash t_1 : p\ T' \quad \dfrac{\Gamma, f : p\ X \triangleright T', z : p\ T' \vdash [\neg f/f]t_2 : +\,[\neg X/X]T}{\Gamma, f : p\ X \triangleright T', z : p\ T' \vdash \mathbf{not}\,[\neg f/f]t_2 : -\,\neg[\neg X/X]T}}{\dfrac{\Gamma \vdash \mathbf{rec}\,f[z = t_1].\mathbf{not}\,[\neg f/f]t_2 : -\,\mu\,X.\neg[\neg X/X]T}{\Gamma \vdash \mathbf{corec}\,f[z = t_1].t_2 : +\,\nu\,X.T}}$$

Figure 15. Typing derivation for corecursion

Expanding the definition of ν-types, we have

$$\mathbb{I} = \neg\mu\,X.\neg\mu\,Y.(\neg X \wedge Y) \vee \top$$

For an example, let us construct a function of type $\mathbb{N} \to \mathbb{I}$, which will return an element of type $\mathbb{I}$ where every node in the tree has branching degree n, when called with input n of positive type $\mathbb{N}$. The code for this is not long, but will require some explanation:

$$\begin{array}{l}\mathbf{tr}\ :=\\ \quad \lambda n.\mathbf{corec}\,f[z = \mathbf{true}].\\ \quad \delta\,r.n\bullet\\ \quad \mathbf{rec}\,g[y = \mathbf{in}_2\mathbf{true}].\\ \quad\quad [(\mathbf{true}, g[\mathbf{in}_1(\mathbf{cons}\,f[\mathbf{true}]\ y)]),\\ \quad\quad \delta\,z'.y \bullet r]\end{array}$$

The basic idea is that we have an outer corecursion with an inner recursion, where the inner recursion is analyzing the input n to generate the finite branching at the current node of the tree. Let us see that this term $\mathbf{tr}$ has positive type $\mathbb{N} \to \mathbb{I}$. First note that we are not using the accumulator for corecursive function f, and so we are just supplying $\mathbf{true}$ as a dummy value (where we have $z = \mathbf{true}$ and $f[\mathbf{true}]$). To type the term we use these typings for bound variables:

$$\begin{array}{lcl} n & : & +\mathbb{N}\\ f & : & +X \triangleright \top\\ r & : & -\mu\,Y.(\neg X \wedge Y) \vee \top\\ g & : & +Y \triangleright \mu\,Y.(\neg X \wedge Y) \vee \top\\ y & : & +\mu\,Y.(\neg X \wedge Y) \vee \top\\ z' & : & +\top \end{array}$$

The key idea is that we are using a positive accumulator of type $\mu\,Y.(\neg X \wedge Y) \vee \top$ in the inner recursion, to accumulate the list of branches at the current node of the tree. When we reach the base case of the recursion (for when n is 0), we return the accumulated branches, by cutting them against the negative variable r. This is the term $\delta\,z'.y \bullet r$. The inner recursion has type $\mu\,Y.(\top \wedge Y) \vee \top$ negatively, since we are cutting it against n of positive type $\mathbb{N}$ (α-equivalent to $\mu\,Y.(\top \wedge Y) \vee \top$). This means that the body of the inner recursion is required to have negative type $(\top \wedge Y) \vee \top$. So we take a co-pair for the body of the inner recursion, since co-pairs inhabit disjunctive types negatively.

Suppose we inspect the term $\mathbf{tr}\ 3$ by evaluating a configuration $+\ \mathbf{tr}\ 3 \bullet \mathbf{not}\,\mathbf{not}\ t$, where t has negative type $\mu\,Y.(\mathbb{I} \wedge Y) \vee \top$. The effect of this will be to substitute t for r in the code for $\mathbf{tr}$, then recursively compute (by the inner recursion) the list of three branches of the tree, and finally (in the base case) pass this list to t by cutting it the list against t. The reader carefully following this explanation may be puzzled by the described substitution of t for r, since the type for t is $-\mathbb{I}$, while the type I listed above for r is $-\mu\,Y.(\neg X \wedge Y) \vee \top$. The solution is that when RecN pushes in the outer corecursion, the type for r will change from $-\mu\,Y.(\neg X \wedge Y) \vee \top$ to the type $-\mu\,Y.(\mathbb{I} \wedge Y) \vee \top$ of t.

8.2 The prefix $\mu X.\nu Y$

The type of possibly infinitely branching trees where all branches are of finite depth has the same definition as the previous example, except with the prefix reversed:

$$\mathbb{J}\ :=\ \mu\,X.\nu\,Y.(X \wedge Y) \vee \top$$

We view descent down the side of the tree corresponding to X in the definition as entering a (finite-depth) branch, and descent down the side corresponding to Y to be proceeding to the next branch at the current node. Let us construct a function which given n of type $\mathbb{N}$, returns the element of $\mathbb{J}$ where all nodes are infinitely branching, and all branches have depth n:

$$\begin{array}{l}\mathbf{trj}\ :=\\ \quad \lambda n.\delta\,r.n\bullet\\ \quad \mathbf{rec}\,f[y = \mathbf{not}\,\mathbf{not}\,\mathbf{in}_2\mathbf{true}].\\ \quad\quad (\delta\,y'.y \bullet r,\\ \quad\quad f[\mathbf{corec}\,g[z = \mathbf{true}].\mathbf{in}_1(y, g[\mathbf{true}])])\end{array}$$

This can be assigned type $\mathbb{N} \to \mathbb{J}$ with these typings for bound variables:

$$\begin{array}{lcl} n & : & +\mathbb{N}\\ r & : & -J\\ f & : & +X \triangleright \mathbb{J}\\ g & : & +Y \triangleright \top \end{array}$$

The outer recursion is using accumulator y of type $\mathbb{J}$ to construct the desired tree. In the base case (where we have $\delta\,y'.y \bullet r$, with the initial value $\mathbf{not}\,\mathbf{not}\,\mathbf{in}_2\mathbf{true}$ for y), the accumulator is just the empty tree. In the step case, we make a recursive call to f, updating the accumulator with the infinite list of branches, where each branch goes to a tree given by the current value of the accumulator (this is accomplished with the pair $(y, g[\mathbf{true}])$). The Mu typing rule is used when typing the accumulator values.

9. Logical Consistency

A type theory is logically consistent iff there is some type which is not inhabited in the empty context. We can establish logical consistency by viewing the type theory as a logic (via the Curry-Howard isomorphism), and showing that there is some formula which is not provable in the logic. This is very easy to do in the case of RP-DC, since the corresponding logic is quite weak, just a fragment of propositional μ-calculus [12]. Because we intend $\mathbf{halt}\ T$ to be used only at the top-level in an initial configuration (to receive a final result), we consider only terms which do not contain $\mathbf{halt}$-subterms:

Theorem 1 (Logical Consistency of RP-DC)**.** *The type $T \wedge \neg T$ is not inhabited by any $\mathbf{halt}$-free term in the empty context, for any type T.*

Proof. A very easy way to prove this is first to define a simple boolean semantics of formulas. We define an interpretation $[\![T]\!]\rho$ of type T as a boolean value (we use *true* and *false* for these here, at the risk of some confusion with the term constructs we defined for types $\top$ and $\bot$). The interpretation is with respect to an assignment

ρ mapping all the type variables X in T to boolean values.

$$\begin{array}{lcl}
[\![X]\!]\rho & = & \rho(X) \\
[\![T_1 \wedge T_2]\!]\rho & = & [\![T_1]\!]\rho \wedge [\![T_2]\!]\rho \\
[\![\neg T]\!]\rho & = & \neg[\![T]\!]\rho \\
[\![\mu\, X\,.\, T]\!]\rho & = & \begin{cases} \textit{false, if } \neg[\![T]\!](\rho[X \mapsto \textit{false}]) \\ \textit{true, otherwise} \end{cases}
\end{array}$$

The idea in the clause for μ-types is to take $[\![\mu\, X\,.\, T]\!]\rho$ to be a least fixed point in the boolean domain, where *false* is considered smaller than *true*. So if $[\![T]\!](\rho[X \mapsto \textit{false}])$ is *false*, we have found a fixed point and return that as the value for $[\![\mu\, X\,.\, T]\!]\rho$. Otherwise, $[\![T]\!](\rho[X \mapsto \textit{false}])$ is *true*, and so monotonicity will ensure that *true* is the fixed point – although we do not need to prove monotonicity here, since it will be addressed when we prove soundness of the MU rule below.

Next, let us define a lifting operation $\ulcorner - \urcorner$ to map contexts and sequents to types (making use of the defined types for truth and implication). Let us write pT as notation for T if $p = +$ and $\neg T$ if $p = -$.

$$\begin{array}{lcl}
\ulcorner . \urcorner & = & \top \\
\ulcorner \Gamma, x : p\ T \urcorner & = & \ulcorner \Gamma \urcorner \wedge pT \\
\ulcorner \Gamma, x : p\ X \triangleright T \urcorner & = & \ulcorner \Gamma \urcorner \wedge \neg X \\
 & & \\
\ulcorner \Gamma \vdash t : p\ T \urcorner & = & \ulcorner \Gamma \urcorner \rightarrow pT
\end{array}$$

We will now prove soundness of RP-DC typing with respect to this semantics, for terms that do not contain $\mathbf{halt}\ T$: whenever $\Gamma \vdash t : p\ T$ is derivable and t does not contain any subterm of the form $\mathbf{halt}\ T'$, then $[\![\ulcorner \Gamma \vdash t : p\ T \urcorner]\!]\rho$ is true, for all assignments ρ to the free type variables of $\ulcorner \Gamma \vdash t : p\ T \urcorner$. This is sufficient for proving that $T \wedge \neg T$ is uninhabited by $\mathbf{halt}$-free terms, since we can easily note that $[\![T \wedge \neg T]\!]\rho$ is *false* for any assignment ρ, and so soundness implies we cannot possibly derive $. \vdash t : + T \wedge \neg T$ if the HALT typing rule is not used. We prove soundness by induction on the structure of the derivation of $\Gamma \vdash t : p\ T$. All cases are very easy, so let us just consider one:

Case:

$$\frac{\mathbf{OccursOnly}\ +\ X\ T \qquad \Gamma \vdash t : + [\mu\, X\,.\, T/X]\, T}{\Gamma \vdash t : + \mu\, X\,.\, T}\ \textsc{Mu}$$

By the induction hypothesis and basic boolean reasoning, it suffices to assume that $[\![[\mu\, X\,.\, T/X]\, T]\!]\rho$ is *true*, and prove that $[\![\mu\, X\,.\, T]\!]\rho$ is also *true*. For this, it suffices to prove that for any T', either the value of $[\![T']\!]\rho[X \mapsto \textit{true}]$ is the same as $[\![T']\!]\rho[X \mapsto \textit{false}]$; or else the value of $[\![T']\!](\rho[X \mapsto b])$ is b if X occurs only positively in T', and $\bar{b}$ (the negation of b) if it occurs only negatively. We omit this easy argument. Also, we can easily show the standard lemma that

$$[\![[T'/X]\, T]\!]\rho = [\![T]\!](\rho[X \mapsto [\![T']\!]\rho])$$

Putting these pieces together: we know that

$$[\![[\mu\, X\,.\, T/X]\, T]\!]\rho = [\![T]\!](\rho[X \mapsto [\![\mu\, X\,.\, T]\!]\rho])$$

It could be that the interpretation of T does not depend on the interpretation of X, and then we have that $[\![T]\!](\rho[X \mapsto \textit{false}])$ is *true*, and so by the semantics for μ-types, so is $[\![\mu\, X\,.\, T]\!]\rho$. Otherwise, since X occurs only positively in T by the first premise of the inference, if $[\![\mu\, X\,.\, T]\!]\rho$ is *false*, then so is

$$[\![[\mu\, X\,.\, T/X]\, T]\!]\rho$$

but we are assuming this is *true*.

□

$$\frac{\mathbf{Canon}\ t_1 : + T_1 \qquad \mathbf{Canon}\ t_2 : + T_2}{\mathbf{Canon}\ (t_1, t_2) : + T_1 \wedge T_2}\ \textsc{CanAndP}$$

$$\frac{\mathbf{Canon}\ t : - T_1}{\mathbf{Canon}\ \iota\, x.\delta\, y.x \bullet t : - T_1 \wedge T_2}\ \textsc{CanAndN1}$$

$$\frac{\mathbf{Canon}\ t : - T_2}{\mathbf{Canon}\ \iota\, x.t : - T_1 \wedge T_2}\ \textsc{CanAndN2}$$

$$\frac{\mathbf{OccursOnly}\ +\ X\ T \qquad \mathbf{Canon}\ t : + [\mu\, X\,.\, T/X]\, T}{\mathbf{Canon}\ t : + \mu\, X\,.\, T}\ \textsc{CanMu}$$

$$\frac{. \vdash \mathbf{rec}\, x[y = t].t' : - \bot}{\mathbf{Canon}\ \mathbf{rec}\, x[y = t].t' : - \bot}\ \textsc{CanFalse}$$

$$\frac{}{\mathbf{Canon}\ \mathbf{halt}\ T : - T}\ \textsc{CanHalt}$$

Figure 17. Canonical inhabitation

10. Canonicity

Classical reasoning is supposed to be nonconstructive, and hence noncomputational. Despite the fact that we have a reduction semantics for RP-DC, one might reasonably be concerned that it could fail to be suitable for programming. But languages with control operators are used heavily for practical programming. So what makes a computational classical type theory (CCTT) nonconstructive?

The answer is the failure, in general, of canonicity. For programming languages, we usually expect that closed normal forms of type T will be *canonical*: they will be built by constructors from canonical subdata. For example, if we have a closed normal form of type $\mathbb{N} \vee \mathbb{L}\ \mathbb{N}$, we expect that it is either the left injection of a canonical natural number, or the right injection of a canonical list of natural numbers. In CCTTs, canonicity fails for types like $T \vee \neg T$. In Section 5.3 we saw that a closed normal form of this type is $\mathbf{not}\, \iota\, x.\mathbf{not}\, x$. This is not the encoding of a left injection of a value of type T or the right injection of a value of type $\neg T$. Recall from Section 5.1 that the encodings of these are the following, where x and y are assumed not to be free in t:

$$\begin{array}{lcl}
\mathbf{in}_1\, t & := & \mathbf{not}\, \iota\, x.\delta\, y.x \bullet \mathbf{not}\, t \\
\mathbf{in}_2\, t & := & \mathbf{not}\, \iota\, x.\mathbf{not}\, t
\end{array}$$

The term $\mathbf{not}\, \iota\, x.\mathbf{not}\, x$ does not match the right hand side of the second defining equation, because such a match would take x for t – but t is required not to contain x free.

As we would expect from the practical application of programming languages with control operators, this failure of canonicity in general does not render a CCTT like RP-DC unusable for programming. For there are types where the closed normal forms are indeed of the expected canonical shape. Figure 17 defines one notion (others are surely possible) of what it means for a term to be canonical of particular positive and negative types. The CANFALSE rule gives us a base case for the inductive definition.

Let us define positive canonical types S and negative canonical types R as follows, where we additionally require that S is not just X in the case of $\mu\, X.S$:

$$\begin{array}{lcl}
\textit{positive canonical } S & ::= & X \mid S \wedge S' \mid \neg R \mid \\
 & & \mu\, X.S \\
\textit{negative canonical } R & ::= & R \wedge R' \mid \neg S \mid\ \bot
\end{array}$$

We do not intend to observe coinductive values directly, so we do not include ν-types in the definition of R. The following can then be proved by induction on the type in question (noting that $\bot$, which is defined to be $\mu\, X.X$, is not positive canonical):

Lemma 2. *No positive canonical type S is also negative canonical, and similarly no negative canonical type R is also positive canonical.*

We can also easily prove:

Lemma 3. *If* $\mathbf{Canon}\ t : p\ T$ *then* t *is closed.*

Using these lemmas, we obtain the following main result:

Theorem 4 (Canonicity)**.** *Suppose that t is a value; that is,* $\mathbf{Nonvalue}[!]\ t$ *is not derivable. Suppose also that the only* $\mathbf{halt}$*-subterms it contains are of the form* $\mathbf{halt}\ S'$*. Finally, suppose that every declaration in Γ is either of the form $x : -S_1$ or $x : +R_1$. Then if $\Gamma \vdash t : +S$, we have* $\mathbf{Canon}\ t : +S$*, and if $\Gamma \vdash t : -R$, we have* $\mathbf{Canon}\ t : -R$*.*

Proof. The proof is by induction on the structure of the assumed typing derivation.

Case:

$$\frac{}{\Gamma_1, x : +S, \Gamma_2 \vdash x : +S}$$

This case is impossible because of the assumption about the form of declarations in Γ, using Lemma 2 to conclude that none of our assumptions $x : +R$ (allowed in Γ) is also an assumption of the form $x : +S$. Similar reasoning shows the following case is also impossible:

$$\frac{}{\Gamma_1, x : -R, \Gamma_2 \vdash x : -R}$$

Case:

$$\frac{\Gamma, x : -S \vdash t_1 : +T' \quad \Gamma, x : -S \vdash t_2 : -T'}{\Gamma \vdash \delta\, x.t_1 \bullet t_2 : +S}$$

Because t is a value, it must be of one of the forms $\delta\, x.t' \bullet y$, $\delta\, x.y \bullet t'$, or $\delta\, x.t' \bullet \mathbf{halt}\ S'$, with $x \in \mathbf{FV}(t')$ and t' a value (since otherwise the NCUT rule applies). Let us break these cases out in turn.

Subcase:

$$\frac{\Gamma, x : -S \vdash t_1 : +T' \quad \Gamma, x : -S \vdash y : -T'}{\Gamma \vdash \delta\, x.t_1 \bullet y : +S}$$

Because of the assumption about the form of declarations in Γ, from the second premise, we can conclude that T' is of the form S', for some positive canonical type S'. So the inference we are considering is really of the form:

$$\frac{\Gamma, x : -S \vdash t_1 : +S' \quad \Gamma, x : -S \vdash y : -S'}{\Gamma \vdash \delta\, x.t_1 \bullet y : +S}$$

Now we may apply our induction hypothesis to conclude $\mathbf{Canon}\ t_1 : +S'$. By Lemma 3, this implies that t_1 is closed, contradicting $x \in \mathbf{FV}(t_1)$.

Subcase:

$$\frac{\Gamma, x : -S \vdash y : +T' \quad \Gamma, x : -S \vdash t_1 : -T'}{\Gamma \vdash \delta\, x.y \bullet t_1 : +S}$$

Similar reasoning as in the previous subcase applies, to show that our inference is really of the form:

$$\frac{\Gamma, x : -S \vdash y : +R' \quad \Gamma, x : -S \vdash t_1 : -R'}{\Gamma \vdash \delta\, x.y \bullet t_1 : +S}$$

So we may apply our IH to the second premise, to contradict $x \in \mathbf{FV}(t_1)$.

Subcase:

$$\frac{\Gamma, x : -S \vdash t_1 : +S' \quad \Gamma, x : -S \vdash \mathbf{halt}\ S' : -S'}{\Gamma \vdash \delta\, x.t_1 \bullet \mathbf{halt}\ S' : +S}$$

This is similar to the first subcase, and we again apply our IH to the first premise, to contradict $x \in \mathbf{FV}(t_1)$.

Case:

$$\frac{\Gamma, x : +R \vdash t_1 : +T' \quad \Gamma, x : +R \vdash t_2 : -T'}{\Gamma \vdash \delta\, x.t_1 \bullet t_2 : -R}$$

This case proceeds similarly to the previous one: because t is a value, it must be of one of the forms $\delta\, x.t' \bullet y$, $\delta\, x.y \bullet t'$, or $\delta\, x.t' \bullet \mathbf{halt}\ S'$, with $x \in \mathbf{FV}(t')$ and t' a value. All the subcases proceed just as above, with the only difference being that we have added an assumption $x : +R$ to the context, instead of one of the form $x : -S$. But our restriction on the form of the context is satisfied either way.

Case:

$$\frac{\Gamma, x : +R_1 \vdash t_1 : -R_2}{\Gamma \vdash \iota\, x.t_1 : -R_1 \wedge R_2}$$

There are several subcases we must consider, based on our assumption that t is a value. Reasoning contrapositively with the rules of Figure 5, it could be that t is of the form $\iota\, x.\delta\, y.y' \bullet t_1'$, where either $y = x$ or else $y \neq x$ and $y \in \mathbf{FV}(t_1')$, and t_1' is a value; or else it is of the form $\iota\, x.t$ where t is a value that is not of the form $\delta\, y.x \bullet t_1'$. Let us consider these subcases in turn:

Subcase:

$$\frac{\dfrac{\Gamma' \vdash x : +R_1 \quad \Gamma' \vdash t_1' : -R_1}{\Gamma, x : +R_1 \vdash \delta\, y.x \bullet t_1' : -R_2}}{\Gamma \vdash \iota\, x.\delta\, y.x \bullet t_1' : -R_1 \wedge R_2}$$

Here, I am abbreviating $\Gamma, x : +R_1, y : +R_2$ as Γ'. We may apply our IH to the second premise to conclude that $\mathbf{Canon}\ t_1 : -R_1$, which is now enough to apply the CANANDN1 rule to conclude $\mathbf{Canon}\ \iota\, x.\delta\, y.x \bullet t_1' : -R_1 \wedge R_2$.

Subcase:

$$\frac{\dfrac{\Gamma' \vdash y' : +R' \quad \Gamma' \vdash t_1' : -R'}{\Gamma, x : +R_1 \vdash \delta\, y.y' \bullet t_1 : -R_2}}{\Gamma \vdash \iota\, x.\delta\, y.y' \bullet t_1' : -R_1 \wedge R_2}$$

Again abbreviate $\Gamma, x : +R_1, y : +R_2$ as Γ'. We may apply our IH to the second premise to conclude $\mathbf{Canon}\ t_1' : -R'$, which contradicts the assumption that $x \in \mathbf{FV}(t_1')$.

Subcase:

$$\frac{\Gamma, x : +R_1 \vdash t_1 : -R_2}{\Gamma \vdash \iota\, x.t_1 : -R_1 \wedge R_2}$$

This is the subcase where t_1 is not of one of the forms of the previous two subcases. Because of the fact that all the rules of Figure 5 which could still apply have ! for θ in their premises (and allow any θ in their conclusion), we can easily prove that t_1

is a value. So we may apply our IH and then the ANDN2 rule to conclude $\mathbf{Canon}\, \iota\, x.t_1 \ : \ - R_1 \wedge R_2$. The case for positive conjunctions is easy, so we omit it.

Case:

$$\frac{\Gamma \vdash t : - R}{\Gamma \vdash \mathbf{not}\, t : + \neg R}$$

We may apply the IH and then CANNOT. The case for $\Gamma \vdash \mathbf{not}\, t : - \neg S$ is similar.

Case:

$$\frac{\Gamma \vdash t_1 : p\ T' \quad \Gamma, x : p\ X \triangleright T', y : p\ T' \vdash t_2 : - X}{\Gamma \vdash \mathbf{rec}\, x[y = t_1].t_2 : - \ \bot}$$

We may apply the CANFALSE rule to obtain the desired conclusion.

Case:

$$\frac{\mathbf{OccursOnly}\ +\ X\ S \quad \Gamma \vdash t : + [\mu\, X.S/X]S}{\Gamma \vdash t : + \mu\, X.S}$$

We need only observe that $[\mu\, X.S/X]S$ is still positive canonical, assuming X occurs only positively in S. So we may apply our IH and then the CANMU rule to obtain the desired result.

□

This theorem shows that final answers of positive canonical type S are indeed canonical values. So RP-DC can be used for practical programming, because for such types S, we can obtain an answer of the expected practical form. And note that this result includes types S corresponding to the general sums of products form for standard algebraic datatypes, because $S \vee S'$ is $\neg(\neg S \wedge \neg S')$, which is indeed positive canonical. So any RP-DC program which computes a final answer of type S is guaranteed to produce a canonical output.

11. Conclusion and Future Work

This paper has introduced the Recursive Polarized Dual Calculus (RP-DC), a development of Wadler's Dual Calculus with inductive types and recursion. Thanks to a polarized reduction relation, standard propositional types may be defined from the minimal set of conjunction and negation, with the expected reductions and typings. We saw several practical examples of programming with recursion, corecursion, and mixed recursion/corecursion. Finally, two significant metatheorems were established: logical consistency (Theorem 1), and canonicity (Theorem 4). Future work includes development of more metatheoretic results, in particular, the standard results of type preservation and normalization. Type preservation is likely straightforward, but I expect normalization to be much more involved. Fortunately, a wonderful tool for this has recently been developed by Krivine: *classical realizability* [13]. Other future work includes implementation, and extension to dependent types. For the latter, our term construct $\iota\, x.t$ should pave the way for dependent typing with a rule like the following, where $x \ : \ T \wedge T'$ binds x in T':

$$\frac{\Gamma, x : + T \vdash t : - T'}{\Gamma \vdash \iota\, x.t : - x : T \wedge T'}$$

It would also be interesting to explore new functional-programming idioms for computing with negative data.

Acknowledgments. Thanks to Harley Eades III for many discussions on computational classical type theories. Thanks also to Phil Wadler for discussing DC. Thanks to the anonymous PLPV '14 reviewers for their detailed and very helpful criticism. This work was supported by the National Science Foundation (NSF grant 0910500). The inference systems were typeset with the help of Ott [17].

References

[1] Andreas M. Abel and Brigitte Pientka. Wellfounded recursion with copatterns: a unified approach to termination and productivity. In Tarmo Uustalu, editor, *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 185–196. ACM, 2013.

[2] Thorsten Altenkirch and Nils Anders Danielsson. Termination Checking Nested Inductive and Coinductive Types. Short note supporting a talk given at PAR 2010, Workshop on Partiality and Recursion in Interactive Theorem Provers. Available from the authors' web pages.

[3] Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992.

[4] Robert Atkey and Conor McBride. Productive coprogramming with guarded recursion. In Tarmo Uustalu, editor, *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 197–208. ACM, 2013.

[5] Marc Bezem, Keiko Nakata, and Tarmo Uustalu. On streams that are finitely red. *Logical Methods in Computer Science*, 8(4), 2012.

[6] Tristan Crolard. A confluent lambda-calculus with a catch/throw mechanism. *J. Funct. Program.*, 9(6):625–647, 1999.

[7] Pierre Curien and Hugo Herbelin. The duality of computation. In *Proceedings of the fifth ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 233–243. ACM, 2000.

[8] Olivier Danvy and Andrzej Filinski. Abstracting control. In *LISP and Functional Programming*, pages 151–160, 1990.

[9] Matthias Felleisen. The theory and practice of first-class prompts. In Jeanne Ferrante and P. Mager, editors, *Fifteenth Annual ACM Symposium on Principles of Programming Languages (POPL), San Diego, California, USA, January 10-13, 1988*, pages 180–190, 1988.

[10] Timothy Griffin. A Formulae-as-Types Notion of Control. In Frances E. Allen, editor, *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages (POPL)*, pages 47–58. ACM Press, 1990.

[11] Daisuke Kimura and Makoto Tatsuta. Call-by-value and call-by-name dual calculi with inductive and coinductive types. *Logical Methods in Computer Science*, 9(1), 2013.

[12] Dexter Kozen. Results on the propositional μ-calculus . *Theoretical Computer Science*, 27(3):333 – 354, 1983.

[13] Jean-Louis Krivine. Realizability in classical logic. *Panoramas et synthèses*, 27:197–229, 2009. Interactive models of computation and program behaviour. Société Mathématique de France.

[14] Paul Francis Mendler. *Inductive Definition in Type Theory*. PhD thesis, Cornell University, 1988. Available at http://www.nuprl.org/documents/Mendler/InductiveDefinition.html.

[15] Russell O'Connor. Classical mathematics for a constructive world. *Mathematical Structures in Computer Science*, 21(4):861–882, 2011.

[16] Michel Parigot. Lambda-Mu-Calculus: An algorithmic interpretation of classical natural deduction. In Andrei Voronkov, editor, *Logic Programming and Automated Reasoning*, pages 190–201, 1992.

[17] Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar, and Rok Strnisa. Ott: Effective tool support for the working semanticist. *J. Funct. Program.*, 20(1):71–122, 2010.

[18] Philip Wadler. Call-by-Value Is Dual to Call-by-Name – Reloaded. In Jürgen Giesl, editor, *Rewriting Techniques and Applications (RTA)*, Lecture Notes in Computer Science, pages 185–203. Springer, 2005.