Command=$\langle$Value$|$Context$\rangle$

Hans-Dieter A. Hiep

July 10, 2018

1 Introduction

This document one of the results of the Literature Study course 2017–2018, at Vrije Universiteit Amsterdam. The other result was a presentation given on June 15, 2018, that complements this work. This work was supervised by Femke van Raamsdonk. Several people have read a draft version of this document: Benjamin Lion and Jana Wagemaker. Thank you: Femke, Benjamin and Jana!

The literature collected and studied here aims to provide insight into the following idea: constructive logics are used as foundations for certain interactive theorem provers that interprets its proofs as programs; can we also have an interactive theorem prover which has classical logic as its foundation, and how can we interpret its proofs as programs?

What follows in this document is structured in this way:

- First, we have collected paragraphs that intend to describe the original motivation, outline of the literature study, and its research agenda: what is a computational interpretation of classical proofs? We will see a high-level description of so-called *interaction*. Interaction is illustrated by an example, and we describe the hypothesis that classical logic can be interpreted as interactions between constructive processes.
- Next, we describe proof systems, natural deduction and sequent calculus. As the study progressed, the author has found the need to explain, in an intuitive and non-exact way, how proof systems can be viewed in general. These three

sections are *not* intended as a very precise, complete or definite description of proof systems.
Instead, we aim to convey a simple message: sequent calculus is a formalization of derivability in natural deduction. This message seems to fit with the original reasons sequent calculus was developed, as a calculus in which to study cut elimination for classical natural deduction.

- Then we consider focussed sequent calculus and the definition of generalized type connectives. A generalized type connective is a user-defined type constructor given either as a data or codata definition, from which rule schema are derived. Following these definitions, we show standard connectives and their derived rules in a focussed sequent calculus as an example.

- The sequent calculus with generalized type connectives has, as coto be unterpart to its proof system, a term language. We consider this term language, and describe how we add terms for example generalized connectives. The added terms are value terms (also called constructors), context terms (also called destructors), and (co)pattern matching.

- The following section animates a previous example by showing proof normalization, and that a term reduction relation exists. We discuss the notion of strategy to force the reduction relation to be confluent.

- We finally give an overview of the studied material, from a historical perspective.

§1 Motivation I am interested in expression languages for both programs and protocols. A program can be intuitively understood as a precise specification of the local behavior of a single processor. A protocol can be understood as a precise specification of the interaction between multiple processors, delineating global behavior. Together, programs and protocols realize overall behavior of systems. This study thus has as goal to find an elegant way of specifying both local and global behavior.

§2 Outline The approach in this study is twofold. One side is to investigate syntax. The other side is to investigate semantics. We will approach syntax by investigating the $\lambda\mu$-calculus [1, 2], the $\overline{\lambda}\mu\tilde{\mu}$-calculus [3], and subsequently the $\mu\tilde{\mu}$-calculus with user-defined (co-)data types [4, 5]. The other side is to approach semantics by investigating streams and stream differential equations [6], and timed data streams, constraint automata, and Reo components and compositions [7]. Although the latter has been investigated by the author, it is not reported in here.

§3 Proofs-as-programs Logic is closely related to computation, as intuitively understood by intuitionistic logic and by typed λ-calculi. The notion of proof normalization in proof theory corresponds to the notion of term reduction in typed λ-calculi, in a very precise manner. For example, formulas of minimal propositional logic and types of simply typed λ-calculus are in a 1-to-1 correspondence. Correspondingly, removing detours in natural deduction proofs and β-reduction of λ-terms are also related.

§4 Multi-machine models A multi-machine model consists of multiple machines that are interactive, parallel and distributed. Machines are also called processors, and these terms are synonyms.

- Interactivity comes from the assumption that processors cannot always individually make progress, but individual progress depends on communication with other machines.
- Parallelism captures the notion of independent progress, that is, a processor could make individual progress if it does not depend on the progress of other processors. Parallelism relates processors by an overlap in time. By 'embarrassing parallelism' we mean that two processors do not communicate. By 'concurrency' we mean that a processor alternates between making local progress and communication.
- Distributed processors are logically and spatially separated from each other, but may be physically connected by a network. The notions of 'network topology' and '(de)centralization' are related to distribution.

Evidently, the realization of many digital computers is done by modeling a computer as a multi-machine, where individual system components communicate through interconnects such as a bus.

Note that in this multi-machine model we may not assume that each machine is constructed or that a precise description of the workings of such machine is known. Some machines may be provided by nature and communication with them primarily happens through sensors or other peripherals. Some machines are proprietary, and their precise description is a trade secret.

§5 Interaction We can understand interaction as a communication between processors. Assume that individual behavior of a processor can be described by an algorithm: a procedure that operates by taking small steps. Interaction can now express the fact that a processor communicates, as illustrated in the following concrete example.

Take two machines where one reads a value while the other writes a value on a shared register. Whenever the reading machine tries to read the value, it is suspended if the value is not yet defined by a writer. The reading machine continues only after the writing machine actually provides a value. Similarly, whenever the writing machine tries to write a value, it is suspended if there is no place yet that can store the value. The writing machine continues only after the reading machine will read the value. The moment that both machines continue, an information exchange has taken place, and thus the machines have communicated. We call this notion synchronization.

Perhaps not so surprising, this behavior is also implemented by a real-world processor [8]:

> "Communication occurs when a sending node writes to a port address and a receiving node to which that port is connected reads the corresponding port address at the same time. When a node operates on a port the data transfer occurs [...] unless the other node connected to the port is not yet performing the complementary operation; in this case the operating node suspends, waking up when the other node connected to that port is performing the complementary operation."

§6 Classical Logic We will study classical logic. My motive behind this direction is the hypothesis that classical logic corresponds to multi-machine models. We cannot expect to prove such hypothesis based on this literature study alone.

A first corroborating piece of evidence is given in [9]. This paper is summarized here, but we will not see all technical details. In the paper, the principle of linearity (a classical axiom) is given a computational interpretation, namely the notion of synchronization. Consider the derivation:

$$\dfrac{\begin{array}{ccc} & [A \to B] & [B \to A] \\ & \mathcal{D}_1 & \mathcal{D}_2 \\ (A \to B) \vee (B \to A) & X & X \end{array}}{X} \vee_e$$

We call the application of the last rule communication. By proofs-as-programs, we obtain two corresponding programs for subderivations $\mathcal{D}_1$ and $\mathcal{D}_2$. We can execute the programs interactively: if at some point the first program wants to obtain any B it has to provide some A, cf. write a value. If the second program wants to obtain any A, cf. read a value, it needs to provide some B.

The actual argument is more subtle. Consider that programs themselves also could communicate, and that proofs-as-programs traditionally holds only for intuitionistic logics. It is suggested to permute communication down the root of the derivation tree such that the resulting tree is in parallel form, that is, communication only occurs at the lowest levels, with intuitionistic subderivations on top. The story does not end there—in fact, a complicated procedure called cross reductions is necessary to remove detours. This involves migration of derivations from between two branches and the notion of communication complexity to show termination.

Syntax

The syntax of the languages we will subsequently consider is divided in three parts: we have proof systems, term languages, and reduction relations. First we will provide some background regarding proof systems: natural deduction and in more detail sequent calculus. We will then see the expression of user-defined types. Then shall we consider the term language of $\mu\tilde{\mu}$-calculus, and we show its type system. We will see term reductions to animate some examples. Finally, we summarize some historical context.

We assume to possess basic preliminary knowledge concerning the topics: logic in computer science, formal languages, abstract algebra, proof theory, lambda calculus, term rewriting, and functional and imperative programming languages.

2 Proof systems

A proof system consists of a set of objects and a set of inference rules. Often, the set of objects is assumed to have a certain syntactical structure—say formulas or sequences. An inference rule relates certain subsets of objects. Rules are schematically defined, that is, the rule schema contains meta-variables that can be instantiated by concrete objects to obtain a concrete inference rule.

Rule schemas are depicted as follows

$$\frac{a_1 \quad \dots \quad a_n}{c_1 \quad \dots \quad c_m}$$

where n and m are non-negative integers. Above the line, each a_1 up to a_n is called an assumption. Below the line, each c_1 up to c_m is called a conclusion. Assumptions and conclusions are objects, possibly containing meta-variables. An inference rule is called an axiom if its set of assumptions is empty.

We consider proof systems that precisely have one conclusion for each rule. A derivation of such a proof system is a labelled tree. Each vertex in the tree is labeled by a concrete inference rule. The number of assumptions of the rule dictate the number of outgoing branches of the vertex. In addition, there is the restriction that the assumption of one rule and the conclusion of another rule are the same when vertices labeled by such rules are linked in the tree.

We remember that the formulation of a proof system itself says nothing about the validity of its derivations. Proof systems are just systems of conventional notation and form. Only meta-theoretical argumentation can convince us that derivations of proof systems are sound with respect to some mathematical conception.

2.1 Natural deduction

Natural deduction is understood as a proof system in which the objects are formulas, and the rules are logically sound inference rules. We first consider the implicational fragment of natural deduction, called minimal logic. Later we extend it to propositional classical logic.

Formulas contain atomic propositions, denoted by P, Q. Formulas are closed under implication, that is, given formulas ϕ and ψ we can construct the formula $(\phi \to \psi)$. The use of parenthesis is conventional.

The proof system of minimal natural deduction has three rules. The first is the axiom of premise: a rule without assumptions and as conclusion some formula ϕ. The second is the modus ponens rule. It has as assumptions formulas $(\phi \to \psi)$ and ϕ, and as conclusion some formula ψ. The third is the implication introduction rule. It has as assumption some formula ψ and as conclusion $(\phi \to \psi)$.

Derivations making use of these rule schemas are considered logically valid. To make this more precise, we need to consider the notion of open and closed premises. The set of open premises is defined inductively over the tree structure of derivations.

A derivation tree with one vertex labeled by the axiom of premise has as set of open premises the singleton set containing the conclusion of the rule. Given a derivation tree with the modus ponens rule at the root, and the two sets of open premises of the subderivations, then the set of open premises of the whole tree is the union. A derivation tree with implication introduction at the root has as set of open premises, the set of open premises of the subderivation

without formula ϕ given that the conclusion is $(\phi \to \psi)$.

We define the derivability relation $\vdash$ that relates a set of formulas to a single formula. Let Γ denote a set of formulas. We define that $\Gamma \vdash \phi$ holds iff there exists a derivation tree such that its set of open premises is Γ and the conclusion of the rule at the root is ϕ. Examples are: $P \vdash P$ and $\vdash P \to P$, since we often forget to write $\{\}$.

Minimal logic plus the axiom schema of double negation elimination $(\neg\neg\phi) \to \phi$ results in a logic equivalent to classical logic[10]. We let $\neg\phi$ be an abbreviation of $\phi \to \bot$ for any formula ϕ. This extension of minimal logic hence requires some fixed constant proposition $\bot$ to represent falsehood.

Thus, the formulas of classical logic are: P, Q for atomic propositions, $\bot$ for falsehood, and closure under implication. The proof system of classical natural deduction has all the rules from minimal logic, and in addition the axiom schema with as conclusion $((\phi \to \bot) \to \bot) \to \phi$. Open and closed premises are defined as before, with the addition that, the set of open premises for the double negation elimination axiom is empty. We adapt the definition of the derivability relation $\vdash$ accordingly. For example, it holds that $\vdash ((P \to Q) \to P) \to P$.

We could abbreviate $((\phi \to \psi) \to \psi)$ as $\phi \vee \psi$ and abbreviate $\neg(\neg\phi \vee \neg\psi)$ as $\phi \wedge \psi$, and thereby we obtain the conventional connectives of propositional classical logic. In addition to the given rule schemas, we also have that the conventional rule schemas of classical logic are admissible. A rule is admissible if one can show that it can be mimicked[10].

Now, the derivability relation $\vdash$ is an interesting object of study itself. For example, we could prove that $\vdash \phi \to \psi$ holds iff $\phi \vdash \psi$ holds.

2.2 Sequent calculus

Sequent calculus can be intuitively understood as a formalization of the derivability relation $\vdash$. A main drawback, in my opinion, of the proof system for natural deduction is the implicit treatment of open premises. Sequent calculus formalizes this notion explicitly. Another difference is that the derivability relation relates sets of formula to a single formula—an asymmetry that can be removed. Instead, we tend towards derivability $\vdash$ that relates two sets of formulas.

In this consideration we shall leave formulas uninterpreted. The formulas of classical logic shall be recovered in a later section, after we have introduced generalized type connectives. The proof system of sequent calculus has as objects sequents. A sequent is, formally, two sequences of formulas separated by commas conjoined by the symbol $\vdash$. Let Γ and Δ be sequences of formulas separated by commas. Typically, rule schemas of sequent calculus include the following structural rule schemas—we formalize that the two sides of $\vdash$ are actually sets:

$$\frac{\Gamma, \phi, \phi \vdash \Delta}{\Gamma, \phi \vdash \Delta} \qquad \frac{\Gamma_1, \phi, \psi, \Gamma_2 \vdash \Delta}{\Gamma_1, \psi, \phi, \Gamma_2 \vdash \Delta} \qquad \frac{\Gamma \vdash \Delta, \phi, \phi}{\Gamma \vdash \Delta, \phi} \qquad \frac{\Gamma \vdash \Delta_1, \phi, \psi, \Delta_2}{\Gamma \vdash \Delta_1, \psi, \phi, \Delta_2}$$

We also call Γ (a set of) assumptions and Δ (a set of) conclusions. The logical interpretation of a sequent $\Gamma \vdash \Delta$ is that its derivability in sequent calculus means that a derivation in natural deduction exists: if $\Gamma \vdash \Delta$ holds then there exists a derivation in natural deduction with as set open premises Γ and a conclusion in Δ.

For example, consider the weaking rule schemas:

$$\frac{\Gamma \vdash \Delta}{\Gamma, \phi \vdash \Delta} \qquad \frac{\Gamma \vdash \Delta}{\Gamma \vdash \phi, \Delta}$$

We can read the left weakening rule as stating: if there exists a derivation tree in natural deduction with open premises Γ and conclusion in Δ, then we can add an unrelated open premise (e.g. using conjunction introduction and elimination) and construct a derivation tree such that it witnesses $\Gamma, \phi \vdash \Delta$. The right weakening rule states: since $\Gamma \vdash \Delta$ holds there exists a derivation tree of some conclusion in Δ, and this derivation tree can also be directly used as a witness for $\Gamma \vdash \phi, \Delta$.

In summary, the sequent calculus we have outlined can be thought of as a meta-proof system that formalizes the derivability relation of natural deduction.

We will formalize a slightly different sequent calculus here that introduces the notion of focus. What we will see is a simplification of the type system in [3].

We have three syntactic categories: values, environments, and commands. Environments are also called contexts.

$$\Gamma \vdash \phi \mid \Delta \qquad \text{(value)}$$
$$\Gamma \mid \phi \vdash \Delta \qquad \text{(context)}$$
$$\Gamma \vdash \Delta \qquad \text{(command)}$$

The symbols $\vdash$ and $\mid$ are special separators, part of the sequent. The reason for using the special symbol $\mid$ is because it is different than commas that separate formulas in Γ and Δ. Γ and Δ stand for

sequences of formulas seperated by commas, and we assume similar structural rules as before allowing us to handle such sequences as sets. The formula ϕ above is called the focussed formula: that focussed formula is on the right of $\vdash$ in $\Gamma \vdash \phi \mid \Delta$ and it is on the left of $\vdash$ in $\Gamma \mid \phi \vdash \Delta$. A sequent that has a focussed formula is called a focussed sequent. Hence, the last sequent is unfocussed.

All formulas on the left of $\vdash$ are still called assumptions, including the focussed formula. Similarly, all formulas on the right of $\vdash$ are still called conclusions. The logical intuition behind sequents is the same: there exists a derivation in natural deduction with as open premises the set of assumptions and a conclusion on the right.

We define the following rule schemas on focussed sequents:

$$\frac{\Gamma \vdash \phi \mid \Delta \quad \Gamma \mid \phi \vdash \Delta}{\Gamma \vdash \Delta}\ \mathit{cut} \qquad \frac{}{\Gamma \mid \phi \vdash \phi, \Delta} \qquad \frac{}{\Gamma, \phi \vdash \phi \mid \Delta}$$

(Compare to section 4 of [3].)

$$\frac{\Gamma, \phi \vdash \Delta}{\Gamma \mid \phi \vdash \Delta} \qquad \frac{\Gamma \vdash \phi, \Delta}{\Gamma \vdash \phi \mid \Delta}$$

The cut rule is the only rule that allows one to eliminate a focussed formula, while the other rules introduce a focussed formula: the two rules on the top right close the focussed formula by assumption, the two bottom right rules introduce focus by eliminating a command.

The reason for having a focussed formula is still elusive at this point. We shall return to this issue later on, where we deal with computational interpretation.

2.3 (Co-)data types

In the previous section we have only considered uninterpreted formulas. The proof system can be adapted to allow user-defined types, to allow the specification of type constructors.

In programming languages and interactive theorem proving, providing a facility for user-defined types is important. Types allow one, in the case of programming, to organize information and processes. In the case of interactive theorem proving, formulas-as-types allows one to formalize theorems, and by proofs-as-programs to prove them valid if the type system is sound.

We shall consider a systematic approach for defining type connectives following [4, 5]. A definition of a type connective consists of: the connective which is defined, free type variables and a set of sequents. The set of sequents may only consist of focussed sequents, and focussed formulas must have the defined type connective at the root of the focussed formula. Such sequents are called the defining clauses of the type definition. One should be careful to distinguish the use of sequents in two different places: sequents as the defining clauses of a type definition, sequents as the objects of our proof system.

The computational intuition behind such clauses can be found in [4]: "assumptions act as inputs and conclusions act as outputs." This intuition will be further developed in the next few sections.

There are two kinds of type definitions: **data** and **codata**. For the first kind, all defining clauses have focus on the right, that is, all defining clauses are value sequents. For the second kind, all defining clauses have focus on the left, i.e. are context sequents.

Example 1. Well-known constructions for sets are shown below. Consider the first data type definition. Given two types A and B, then the disjoint union $A \oplus B$ is formed by either an element of A or an element of B. Following our previous computation intuition: given as input some A then there exists (by definition) some output $A \oplus B$, and similar for the second clause. The other data type definitions have a similar intuition: $A \otimes B$ pairs two elements and 1 is a unit type.

data $A \oplus B$	**data** $A \otimes B$	**data** 1	**data** 0
$A \vdash A \oplus B \mid$	$A, B \vdash A \otimes B \mid$	$\vdash 1 \mid$	
$B \vdash A \oplus B \mid$			

Remark that the definition of 0 has no defining clauses. This is valid since a type definition consists of a set of clauses that could possibly be empty. Remember that for data types, all clauses have focus on the right.

Example 2. Lesser-known constructions for logical connectives:

codata $A \& B$	**codata** $A \parr B$	**codata** $\bot$	**codata** $\top$
$\mid A \& B \vdash A$	$\mid A \parr B \vdash A, B$	$\mid \bot \vdash$	
$\mid A \& B \vdash B$			

Remember that for codata types, all clauses have focus on the left.

It might seem at first counter-intuitive, that both 0 and $\top$ have no defining clauses. The intuition is that a value of 0 is never constructable, whereas $\top$ is an abstract object with no possible observations [4]. The defining clause $\mid \bot \vdash$ signifies that there exists a derivation in natural deduction with its conclusion in the empty set—that is obviously absurd, and we might believe that $\bot$ is false.

Example 3. For implication we give the following definition. In addition we define the dual of implication, called difference.

codata $A \to B$

$$A \mid A \to B \vdash B$$

data $A - B$

$$A \vdash A - B \mid B$$

User-defined types extend the inference rules of the proof system we consider. We will later employ such extended proof system as a type system for checking whether terms have a valid type. Given a type definition, we have that certain witnesses correspond to the defining clauses. In our proof system each type definition has two associated kinds of inference rules: a left and a right rule. Note that these typing rules were found in [5], but were absent from [4].

Instead of presenting the generalized rules, which in my opinion are confusing, we will show the typing rules by example. In general we have the following pattern:
For data types, there is precisely one left rule which has, for each clause an assumption, and for each clause there is one right rule.
For codata types, there is precisely one right rule which has, for each clause an assumption, and for each clause there is one left rule.

The assumptions of the left rule for data and the right rule for codata are unfocussed, and each assumption sequent per clause corresponds to the additional input and output types of that clause.

Each right rule for data and left rule for codata, per clause has as many assumptions as there are other input and output types.

For the right rules for data and the left rules for codata, the assumptions corresponding to input are focussed on the right, and assumptions corresponding to output are focussed on the left.

Example 4. The typing rules for all the data definitions of the previous examples are given here. We first see the data type rules of $A \oplus B$, $A \otimes B$, 1, 0, and $A - B$. The codata type rules of $A \& B$, $A \parr B$, $\bot$, $\top$, and $A \to B$ are quite similar to those of the data types.

$$\frac{\Gamma, A \vdash \Delta \quad \Gamma, B \vdash \Delta}{\Gamma \mid A \oplus B \vdash \Delta}\, L_\oplus \qquad \frac{\Gamma \vdash A \mid \Delta}{\Gamma \vdash A \oplus B \mid \Delta}\, R_{\oplus,1} \qquad \frac{\Gamma \vdash B \mid \Delta}{\Gamma \vdash A \oplus B \mid \Delta}\, R_{\oplus,2}$$

$$\frac{\Gamma, A, B \vdash \Delta}{\Gamma \mid A \otimes B \vdash \Delta}\, L_\otimes \qquad \frac{\Gamma \vdash A \mid \Delta \quad \Gamma \vdash B \mid \Delta}{\Gamma \vdash A \otimes B \mid \Delta}\, R_\otimes$$

$$\frac{\Gamma \vdash \Delta}{\Gamma \mid 1 \vdash \Delta}\, L_1 \qquad \frac{}{\Gamma \mid 0 \vdash \Delta}\, L_0 \qquad \frac{}{\Gamma \vdash 1 \mid \Delta}\, R_1$$

$$\frac{\Gamma, B \vdash A, \Delta}{\Gamma \mid A - B \vdash \Delta}\, L_- \qquad \frac{\Gamma \mid A \vdash \Delta \quad \Gamma \vdash B \mid \Delta}{\Gamma \vdash A - B \mid \Delta}\, R_-$$

$$\frac{\Gamma \mid A \vdash \Delta}{\Gamma \mid A \& B \vdash \Delta}\, L_{\&,1} \qquad \frac{\Gamma \mid B \vdash \Delta}{\Gamma \mid A \& B \vdash \Delta}\, L_{\&,2} \qquad \frac{\Gamma \vdash A, \Delta \quad \Gamma \vdash B, \Delta}{\Gamma \vdash A \& B \mid \Delta}\, R_\&$$

$$\frac{\Gamma \mid A \vdash \Delta \quad \Gamma \mid B \vdash \Delta}{\Gamma \mid A \parr B \vdash \Delta}\, L_\parr \qquad \frac{\Gamma \vdash A, B, \Delta}{\Gamma \vdash A \parr B \mid \Delta}\, R_\parr$$

$$\frac{}{\Gamma \mid \bot \vdash \Delta}\, L_\bot \qquad \frac{\Gamma \vdash \Delta}{\Gamma \vdash \bot \mid \Delta}\, R_\bot \qquad \frac{}{\Gamma \vdash \top \mid \Delta}\, R_\top$$

$$\frac{\Gamma \vdash A \mid \Delta \quad \Gamma \mid B \vdash \Delta}{\Gamma \mid A \to B \vdash \Delta}\, L_\to \qquad \frac{\Gamma, A \vdash B, \Delta}{\Gamma \vdash A \to B \mid \Delta}\, R_\to$$

In my opinion, we can appreciate two kinds of dualities here. A dual is a very simple syntactic operation that swaps two symbols and is an involution.

The first duality is swapping conjunction and disjunction. In the typing rules, we see that $L_{\oplus}$ and $L_{\otimes}$ are very similar: one has meta-level conjunction with multiple assumptions in the rule, the other has object-level conjunction with multiple assumptions in the sequent. We also see a similarity between $R_{\oplus}$ and $R_{\otimes}$.

The second duality is swapping data and codata. In the typing rules, we see that L and R rules are swapped. For focussed sequents, swap the symbols $\mid$ and $\vdash$ in assumptions and conclusion of the inference rules. Furthermore, we swap the additional variables on the left and right of $\vdash$ for unfocussed sequents. We swap [$\oplus$ and $\&$], [$\otimes$ and $\parr$] and all other type connectives that were shown side by side in examples 1 and 2.

We will finally remark that the type system is kinded. This means that type definitions themselves are also stratified in a kind system. Kinds are super-types of types. Our type definitions are implicitly kinded. The language of types and kinds is the simply typed lambda calculus, with $*$ as base kind. Type variables such as A and B are of kind $*$. Type connectives have a fixed arity, and applying the right number of variables also gives the kind $*$.

In [5] it is suggested that this approach is also suitable for defining (co-)inductive data types. This allows users to formalize natural numbers and binary trees on the one hand, and streams on the other hand. Certain type connectives such as $\exists :: (* \to *) \to *$ and $\forall :: (* \to *) \to *$ can be used to express quantification. We will not consider this aspect in more detail here.

3 Term languages

In this section we will treat the $\mu\tilde{\mu}$-calculus, and relate it to user-defined types of the previous section.

The type definitions we have seen previously will be used to construct inhabitant terms of types. Existence of an inhabitant of a user-defined type is justified by one of its defining clauses. The core calculus also consists of primitive terms, which we will see now.

3.1 The core $\mu\tilde{\mu}$-calculus

The core $\mu\tilde{\mu}$-calculus has three syntactic categories of terms: values, contexts and commands. These are defined by the following grammar:

$$v ::= x \mid \mu\alpha.c \qquad e ::= \alpha \mid \tilde{\mu}x.c \qquad c ::= \langle v \mid e \rangle$$

By $x, y, z, \ldots$ we denote value variables of which there are countably many. By $\alpha, \beta, \gamma, \ldots$ we denote context variables, disjoint from value variables, of which there are also countably many. There are two binding constructs: $\mu\alpha.c$ is a value which itself binds an environment variable α in the nested command c, and $\tilde{\mu}x.c$ is an environment which binds a value variable. We consider our language up to renaming of bound variables.

Example terms are: x, α, $\mu\alpha.\langle x \mid \alpha \rangle$, $\tilde{\mu}x.\langle x \mid \alpha \rangle$ and $\langle \mu\alpha.\langle x \mid \alpha \rangle \mid \tilde{\mu}x.\langle x \mid \alpha \rangle \rangle$. A term is closed if all its variable occurrences are bound by a surrounding μ or $\tilde{\mu}$ term. A nice exercise is to think of all possible closed terms of this calculus.

3.2 The $\mu\tilde{\mu}$-calculus with typesto be

Once we have the definition of user-defined types, we also need to extend the syntax of terms. We shall do this by example, but this can be generalized in full. We show how it works by example, because the general syntax of [5] is involved and not quickly understandable.

Example 5. Consider the type definition of functions. We now shall show the full syntax of defining clauses: not only formulas, but typing judgements of the form $t : F$. Every unfocussed judgement on the left-hand side of $\vdash$ types a value variable. Every unfocussed judgement on the right types a context variable. For data, each clause introduces a value term, and for codata, each clause introduces a context term. The term typed by the focussed formula that contains variables occurring in the rest of the clause.

codata $A \to B$ **where**

$x : A \mid A[x,\alpha] : A \to B \vdash \alpha : B$

We abbreviate $A[x,a]$ as $x \cdot \alpha$. We can reconsider the syntactic categories of values and contexts by adding:

$$v ::= \ldots \mid \mu(x \cdot \alpha.c) \qquad e ::= \ldots \mid v \cdot e$$

Introducing a data type involves the following changes:

data $A \otimes B$ **where**

$x : A, y : B \vdash P(x,y) : A \otimes B \mid$ we abbreviate $P(x,y)$ as just (x,y)

$$v ::= \ldots \mid (v,v) \qquad e ::= \ldots \mid \tilde{\mu}[(x,y).c]$$

Having multiple clauses reveals more interesting patterns:

data $A \oplus B$ **where**	**codata** $A \& B$
$x : A \vdash L(x) : A \oplus B \mid$	$\mid I[\alpha] : A \& B \vdash \alpha : A$
$x : B \vdash R(x) : A \oplus B \mid$	$\mid J[\alpha] : A \& B \vdash \alpha : B$

$$v ::= \ldots \mid L(v) \mid R(v) \qquad e ::= \ldots \mid \tilde{\mu}[L(x).c \mid R(x).c]$$

$$v ::= \ldots \mid \mu(I[\alpha].c \mid J[\alpha].c) \qquad e ::= \ldots \mid I[e] \mid J[e]$$

The binding construct becomes a (co)pattern matching construct: it now contains more than one nested commands. We will see that, for example, $\langle \mu(I[\alpha].c_1 \mid J[\alpha].c_2) \mid I[e] \rangle$ reduces to c_1 with α substituted by e: the copattern indicates that we have a choice between c_1 and c_2 that depends on the observation of the environment. Dually, $\langle R(v) \mid \tilde{\mu}(L(x).c_1 \mid R(x).c_2) \rangle$ reduces to c_2 with x substituted by v: again we have a choice that depends on the construction of the value.

3.3 Typing Rules

We observe that the proof rules corresponding to a user-defined connectives also are related to the typing rules of its introduced terms. The typing rules for the core $\mu\tilde{\mu}$-calculus are first shown. Just as the proof rules are generally applicable, we can always type the core terms using these rules. The typing rules for the parametric $\mu\tilde{\mu}$-calculus is not given in full generality, but we will only consider an example.

We have found these rules in section 4 of [3] and in figure 4 of [5].

$$\frac{\Gamma \vdash v : \phi \mid \Delta \quad \Gamma \mid e : \phi \vdash \Delta}{\langle v \mid e \rangle : (\Gamma \vdash \Delta)}\ \mathit{Cut}$$

$$\frac{}{\Gamma \mid \alpha : \phi \vdash \alpha : \phi, \Delta}\ \mathit{CoVar} \qquad \frac{}{\Gamma, x : \phi \vdash x : \phi \mid \Delta}\ \mathit{Var}$$

$$\frac{c : (\Gamma, x : \phi \vdash \Delta)}{\Gamma \mid \tilde{\mu} x.c : \phi \vdash \Delta}\ \mathit{CoAct} \qquad \frac{c : (\Gamma \vdash \alpha : \phi, \Delta)}{\Gamma \vdash \mu\alpha.c : \phi \mid \Delta}\ \mathit{Act}$$

Example 6. The typing rules for terms of $A \oplus B$ are below. This is very closely related to the proof rules we have seen before for the user-defined type $A \oplus B$.

$$\frac{c_1 : (\Gamma, x : A \vdash \Delta) \quad c_2 : (\Gamma, x : B \vdash \Delta)}{\Gamma \mid \tilde{\mu}(L(x).c_1 \mid R(x).c_2) : A \oplus B \vdash \Delta}\ L_{\oplus}$$

$$\frac{\Gamma \vdash v : A \mid \Delta}{\Gamma \vdash L(v) : A \oplus B \mid \Delta}\ R_{\oplus,1} \qquad \frac{\Gamma \vdash v : B \mid \Delta}{\Gamma \vdash R(v) : A \oplus B \mid \Delta}\ R_{\oplus,2}$$

§7 Functions The application operator $\cdot$ deserves more attention. We repeat the definition of $A \to B$ below, with $x \cdot \alpha$ for $A[x, \alpha]$.

codata $A \to B$ **where**

$x : A \mid x \cdot \alpha : A \to B \vdash \alpha : B$

The function type is a co-data type, since the focussed judgement is on the left. The type constructor $\to$ is of kind $* \to * \to *$. We now have that λ-abstraction is a derived concept, as corroborated by the following definition:

$$\lambda x.v = \mu(x \cdot \alpha.\langle v \mid \alpha \rangle)$$

From the definition of $A \to B$ above it is apparent that only application between a proof and an context is defined. But application of two proof terms is also a derived concept:

$$v \cdot w = \mu\beta.\langle v \mid w \cdot \beta \rangle$$

Given that $\lambda x.v = \mu(x \cdot \alpha.\langle v \mid \alpha \rangle)$, the typical typing rules for λ-abstraction and λ-application are admissible:

$$\dfrac{\dfrac{\dfrac{\Gamma, x : A \vdash v : B \mid \Delta}{\Gamma, x : A \vdash v : B \mid \alpha : B, \Delta}\, W \qquad \dfrac{}{\Gamma, x : A \mid \alpha : A \vdash \alpha : A, \Delta}\, \begin{matrix} CoVar \\ Cut \end{matrix}}{\langle v \mid \alpha \rangle : (\Gamma, x : A \vdash \alpha : B, \Delta)}}{\Gamma \vdash \mu(x \cdot \alpha.\langle v \mid \alpha \rangle) : A \to B \mid \Delta}\, R_{\to} \qquad tobe$$

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash v : A \to B \mid \Delta}{\Gamma \vdash v : A \to B \mid \beta : B, \Delta}\, W \qquad \dfrac{\dfrac{\Gamma \vdash w : A \mid \Delta}{\Gamma \vdash w : A \mid \beta : B, \Delta}\, W \qquad \dfrac{}{\Gamma \mid \beta : B \vdash \beta : B, \Delta}\, \begin{matrix} CoVar \\ L_{\to} \end{matrix}}{\Gamma \mid w \cdot \beta : A \to B \vdash \beta : B, \Delta}\, Cut}{\langle v \mid w \cdot \beta \rangle : (\Gamma \vdash \beta : B, \Delta)}}{\Gamma \vdash \mu\beta.\langle v \mid w \cdot \beta \rangle : B \mid \Delta}\, Act$$

Where W is the admissible weakening rule, since the *CoVar* and *Var* rules have arbitrary contexts Γ and Δ we can modify the derivation on top by adding aribtrary judgements.

We will animate these expressions in the next section.

4 Reduction relations

There are two core rewrite rules:

$$\langle \mu\alpha.c \mid e \rangle \to_{\mu} c\{\alpha := e\} \qquad \langle v \mid \tilde{\mu}x.c \rangle \to_{\tilde{\mu}} c\{x := v\}$$

where we substitute, avoiding capture by suitable renaming, the free occurrences of variables in the nested command by the variable bound in the outer construct. This system is not confluent since the critical pair

$$c[\alpha := \tilde{\mu}x.c'] \leftarrow \langle \mu\alpha.c \mid \tilde{\mu}x.c' \rangle \to c'[x := \mu\alpha.c]$$

is not necessarily joinable. Confluence can be restored by prioritizing one rule over the other. A particular priority scheme is called a strategy. The calculus is parametric over its evaluation strategy. Two strategies are well-known: the call-by-value strategy always prefers a μ-step, and the call-by-name strategy always prefers a $\tilde{\mu}$-step. Other strategies exists.

Example 7. Given the term $\tilde{\mu}x.\langle \mu\alpha.\langle x \mid \alpha \rangle \mid \tilde{\mu}y.\langle y \mid \alpha \rangle \rangle$. This environment term is not closed, since α occurs free. However it also contains a bound α, which we can freely rename to β: $\tilde{\mu}x.\langle \mu\beta.\langle x \mid \beta \rangle \mid \tilde{\mu}y.\langle y \mid \alpha \rangle \rangle$. There are two reduction paths to the normal form $\tilde{\mu}x.\langle x \mid \alpha \rangle$, namely one by reducing the inner command $\langle \mu\beta.\langle x \mid \beta \rangle \mid \ldots \rangle$ and the other reducing $\langle \ldots \mid \tilde{\mu}y.\langle y \mid \alpha \rangle \rangle$.

The rewrite rules are also extended for user-defined types. Again, instead of giving the generalized version, we will instead show it by example.

For the two types $A \otimes B$ and $A \to B$ we have the reductions:

$$\langle (v,w) \mid \tilde{\mu}[(x,y).c] \rangle \to_{\otimes} c\{x := v, y := w\}$$

and

$$\langle \mu(x \cdot \alpha.c) \mid v \cdot e \rangle \to_{\to} c\{x := v, \alpha := e\}$$

Example 8. Consider $A \otimes B \to A$. We show a term that inhabits this type, and its typing derivation: $\mu(x \cdot \alpha.\langle x \mid \tilde{\mu}[(l,r).\langle l \mid \alpha \rangle]\rangle)$.
Let $\Gamma = x : A \otimes B, l : A, r : B$ in

$$\dfrac{\dfrac{\dfrac{}{x : A \otimes B \vdash x : A \otimes B \mid \alpha : A}\ Var \quad \dfrac{\dfrac{\dfrac{}{\Gamma \vdash l : A \mid \alpha : A}\ Var \quad \dfrac{}{\Gamma \mid \alpha : A \vdash \alpha : A}\ CoVar}{\langle l \mid \alpha \rangle : (x : A \otimes B, l : A, r : B \vdash \alpha : A)}\ Cut}{x : A \otimes B \mid \tilde{\mu}[(l,r).\langle l \mid \alpha \rangle] : A \otimes B \vdash \alpha : A}\ L_{\otimes}}{\langle x \mid \tilde{\mu}[(l,r).\langle l \mid \alpha \rangle]\rangle : (x : A \otimes B \vdash \alpha : A)}\ Cut}{\vdash \mu(x \cdot \alpha.\langle x \mid \tilde{\mu}[(l,r).\langle l \mid \alpha \rangle]\rangle) : A \otimes B \to A \mid}\ R_{\to}$$

Let us call above expression $\pi_1 : A \otimes B \to A$. We can now apply this function to an argument as follows: $\pi_1 \cdot (y,z)$ for variables $y : A, z : B$.

$$\begin{aligned}
\pi_1 \cdot (y,z) &= \mu\beta.\langle \pi_1 \mid (y,z) \cdot \beta \rangle && \text{(def)} \\
&= \mu\beta.\langle \mu(x \cdot \alpha.\langle x \mid \tilde{\mu}[(l,r).\langle l \mid \alpha \rangle]\rangle) \mid (y,z) \cdot \beta \rangle && \text{(def)} \\
&\to \mu\beta.\langle (y,z) \mid \tilde{\mu}[(l,r).\langle l \mid \beta \rangle]\rangle && \to_{\to} \\
&\to \mu\beta.\langle y \mid \beta \rangle && \to_{\otimes}
\end{aligned}$$

The reader might find it an interesting exercise to prove that the term is well-typed by deriving $y : A, z : B \vdash \pi_1 \cdot (y,z) : A \mid$.

5 Historical Context

As mentioned in the introduction of [2], the type of double negation elimination is given to the control operator $\mathcal{C}$ due to Felleisen and Griffin. This extends the syntax of simply typed λ-calculus in two ways, namely by adding the type $\bot$, and by adding the constant $\mathcal{C}$ such that there are two kinds of applications: (MN) and $(\mathcal{C}N)$ for terms M and N.

Parigot found that a natural deduction system with multiple conclusions was more convenient to work with. In my mind, this already brings the resulting natural deduction system closer to sequent calculus. The result is called the $\lambda\mu$-calculus, which was developed in the quest of finding a suitable computational interpretation that corresponds to classical logic [1].

In $\lambda\mu$-calculus an additional binding term is introduced to λ-calculus, denoted $\mu\alpha.[\beta]M$ for M any term. Named terms are $[\beta]M$ for any term M, and are themselves also considered terms. The calculus can be untyped or typed. Intuitively, we can understand the operator μ as being a λ which potentially accepts an infinite number of arguments [2]: "The effect of the reduction of $(\mu\beta.u)v_1 \dots v_n$ is to give the arguments $v_1, \dots, v_n$ to the subterms of u named β, and this independently of the number n of arguments $\mu\beta.u$ is applied to:" the reduction rule mentioned here is defined as $(\mu\beta.u)v \to \mu\beta.u[v/^*\beta]$ where $u[v/^*\beta]$ replaces in u each subterm $[\beta]w$ by $[\beta](w)v$.

In proof theory we can embed classical logic in intuitionistic logic by applying the double negation translation. In translating $\lambda\mu$-terms to λ-terms, this is called continuation-passing-style translation. According to [2], the difference between λ-calculus and

$\lambda\mu$-calculus, is that only the latter allows infinite arguments whereas the same term translated back to λ-calculus does not. It is then claimed that this difference allows classical logics to reproduce imperative features of programming—which according to my hypothesis could be interpreted as executions of multi-machine models: this connection is however not mentioned in the literature.

We then move to sequent calculus. A goal behind earlier work was to find a "sequent calculus version" of Parigot's $\lambda\mu$-calculus, which in Herbelin's Ph.D. dissertation resulted in the $\bar{\lambda}\mu$-calculus. Sequent calculus was originally developed by Gentzen to study cut-elimination procedures for first-order classical logic, and is considered by [3] to be more well-behaved than natural deduction.

The $\bar{\lambda}\mu$-calculus is presented as a sequent calculus variant of $\lambda\mu$-calculus. $\bar{\lambda}\mu$-calculus is essentially the same as $\lambda\mu$-calculus, in the sense that it preserves normal forms, and that a homomorphism with respect to call-by-name evaluation exists. But it differs in the sense that cut-elimination and reduction steps are no longer directly corresponding.

The λ-calculus and $\lambda\mu$-calculus have the property of confluence: the outcome of reduction does not depend on the chosen evaluation strategy. However, it might be useful to have direct control over evaluation strategy. This became apparent in $\bar{\lambda}\mu$-calculus, which only preserves call-by-name reduction of translated $\lambda\mu$-terms. The $\bar{\lambda}\mu$-calculus is extended with the dual of the μ binder, denoted as $\tilde{\mu}$. The result is called $\overline{\lambda}\mu\tilde{\mu}$-calculus, which no longer is confluent [3].

It turns out that precise control over evaluation strategy now is necessary to regain confluence. This calculus has two confluent subsyntaxes, one preserves call-by-name reduction of translated

$\lambda\mu$-terms and the other preserves call-by-value reduction. The two sub syntaxes are dual, in the sense that a syntactic transformation exists between the two: this explains why some consider the evaluation strategies call-by-name and call-by-value to be duals.

The calculus that is considered here is the parametric $\mu\tilde{\mu}$ core with user-defined (co-)data types as given in [4]. We have derived λ-terms as instances of a user-defined type, and the usual typing rules can be mimicked by the typing rules that are instances of the given generalized system.

It seems that there is no clear reference to the origin of this generalized type system; it does not appear in [4], but in [5] it cites [4] as the origin. Also, in our version of [5], the type system contained some typos, but which turned out not to be of big importance for our interpretation. The reason for absence of reference turned out that [11] was not yet finished at the time of writing [4, 5]. It seems that [11] is the first complete description of the generalized type system.

Bibliography

[1] Michel Parigot. $\lambda\mu$-calculus: an algorithmic interpretation of classical natural deduction. In *3rd International Conference on Logic for Programming Artificial Intelligence and Reasoning*, pages 190–201. Springer, 1992.

[2] Michel Parigot. Classical proofs as programs. In *Kurt Gödel Colloquium on Computational Logic and Proof Theory*, pages 263–276. Springer, 1993.

[3] Pierre-Louis Curien and Hugo Herbelin. The duality of computation. In *5th ACM SIGPLAN International Conference on Functional Programming*, volume 35, pages 233–243. ACM, 2000.

[4] Paul Downen and Zena Ariola. The duality of construction. In *17th European Symposium on Programming Languages and Systems*, pages 249–269. Springer, 2014.

[5] Paul Downen, Philip Johnson-Freyd, and Zena M Ariola. Structures for structural recursion. In *20th ACM SIGPLAN International Conference on Functional Programming*, volume 50, pages 127–139. ACM, 2015.

[6] Henning Basold, Helle Hvid Hansen, Jean-Éric Pin, and Jan Rutten. Newton series, coinductively: a comparative study of composition. In *Mathematical Structures in Computer Science*, pages 1–29. Springer, 2017.

[7] Christel Baier, Marjan Sirjani, Farhad Arbab, and Jan Rutten. Modeling component connectors in reo by constraint automata. *Science of computer programming*, 61(2):75–113, 2006.

[8] *GreenArrays Product Data Book DB001: F18A Technology Reference*. GreenArrays Incorporated, 2017.

[9] Federico Aschieri, Agata Ciabattoni, and Francesco A Genco. Gödel logic: From natural deduction to parallel computation. In *32nd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS), 2017*, pages 1–12. IEEE, 2017.

[10] Hans-Dieter Hiep. Alternative connectives for classical propositional logic.

[11] Paul Downen. *Sequent Calculus: A Logic and a Language for Computation and Duality*. PhD thesis, University of Oregon, 2017.