Universiteit Utrecht

Faculty of Science

Dept. of Information and Computing Sciences

MMH: High-level programming with the Mu-Mu-Tilde-calculus

Author

Rik van Toor
ICA-4239776

Supervisors

Dr. Wouter Swierstra
Dr. Alejandro Serrano Mena

March 17, 2020

CONTENTS

Acknowledgements

I would like to thank both my supervisors, Dr. Wouter Swierstra and Dr. Alejandro Serrano Mena, for their support and invaluable advice they have provided throughout this project. I especially want to acknowledge Dr. Serrano Mena's contributions to this work, given that he has done all of it voluntarily in his spare time, purely out of interest in the subject, for which I am very grateful.

Abstract

The $\mu\tilde{\mu}$-calculus is a small core programming language, for which the separation between data and co-data is essential. To make the power of this separation more accessible, we introduce *MMH*, a high-level functional programming language that compiles to the $\mu\tilde{\mu}$-calculus. We show how λ-calculus programs can be converted to $\mu\tilde{\mu}$-calculus programs, and extend the calculus with programmer-friendly features, such as nested (co-)pattern matching. We introduce a polymorphic typing system to the $\mu\tilde{\mu}$-calculus for which type inferencing is decidable, and allow *MMH* to reap the benefits.

1 Introduction

Since its inception, the $\mu\tilde{\mu}$-calculus, pronounced "mu-mu-tilde-calculus", has long been an interest of study. The ideas and concepts that over time have been introduced to the calculus, have inspired practical implementations, such as *Sequent Core*, an alternative core language to GHC by Downen et al. [Dow+16]. Unfortunately, these implementations often fail at doing either of two things: fully exposing the power of the $\mu\tilde{\mu}$-calculus, or providing a comfortable programming surface. The Sequent Core compiler plugin allows Haskell programs to be compiled to the $\mu\tilde{\mu}$-calculus. However, as its programs are written in Haskell, not all features of the calculus are exposed to the user. Co-datatypes, for example, are not supported. Similarly, experimental evaluators and compilers for the $\mu\tilde{\mu}$-calculus expose all features of the calculus, but writing programs for them is not done in a modern programming language, but in raw $\mu\tilde{\mu}$-calculus (co-)terms.

1.1 Contributions

This thesis contains detailed descriptions of solutions to numerous shortcomings of the $\mu\tilde{\mu}$-calculus. The following solutions are offered and explained:

- A formal way to convert λ-calculus terms to $\mu\tilde{\mu}$-calculus terms
- Adding support for nested (co-)patterns in the $\mu\tilde{\mu}$-calculus by adapting Augustsson's algorithm to include co-data and commands
- Defining syntactical structures to allow μ-terms, co-terms and commands to be written. These structures, in addition to a large part of Haskell, form a new programming language based on the $\mu\tilde{\mu}$-calculus, called *MMH*.
- Adding type polymorphism to the $\mu\tilde{\mu}$-calculus by implementing a Hindley-Milner typing system.

1.2 Structure

First, an extensive background section will give an introduction to the $\mu\tilde{\mu}$-calculus by explaining the natural deduction and the sequent calculus, and their relations to the λ-calculus and the $\mu\tilde{\mu}$-calculus.

This thesis then introduces a high-level programming language that uses the $\mu\tilde{\mu}$-calculus at its core, and supports advanced datatypes and co-datatypes to be defined and processed in a straightforward way. To do so, first, a method to convert programs written for the λ-calculus to the $\mu\tilde{\mu}$-calculus is proposed, therefore allowing Haskell programs to be transformed to $\mu\tilde{\mu}$-calculus programs. Next, Augustsson's pattern expansion is adapted for use in the $\mu\tilde{\mu}$-calculus to allow nested pattern matching. Finally, new syntax is introduced to the language, which is used to define co-terms and commands.

After defining this programming language, this thesis proposes a polymorphic type system for the $\mu\tilde{\mu}$-calculus. An existing system for polymorphism is explored, and its advantages and disadvantages are laid out. Then, a new polymorphic type system that is heavily based on the Hindley-Milner typing system is formulated.

This thesis ends by listing the conclusions that can be taken from the earlier sections, and suggesting multiple related questions and topics that can be researched in the future.

2

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

The $\mu\tilde{\mu}$-calculus, as formalised by Herbelin, is a logical system within which computation can be modeled [Her05]. The calculus can be seen as a minimalistic theoretical programming language, based on the sequent calculus. In this chapter, the ins and outs of the $\mu\tilde{\mu}$-calculus are explored. First, the sequent calculus and its origins are explained. Next, the $\mu\tilde{\mu}$-calculus' relation to the sequent calculus is described. Finally, the differences between the sequent calculus and natural deduction, as well as the differences between the $\mu\tilde{\mu}$-calculus and the λ-calculus are explained.

2.1 Introduction to Gentzen's sequent calculus

In 1935, Gerhard Gentzen invented two separate formal systems of logical reasoning [Gen35]: *natural deduction* and the *sequent calculus*. Both are very similar in the sense that both are systems for second-order propositional logic. The two systems work with logical propositions that either consist of an atomic variable (X, Y, Z), or a proposition constructor. The supported constructors are truth ($\top$), falsehood ($\bot$), disjunction ($\vee$), conjunction ($\wedge$), implication ($\supset$), universal quantification ($\forall$), and existential quantification ($\exists$). The syntax of the propositions that will be used is defined in Figure 2.1.

Both systems support proofs of these propositions using *judgements*. However, the two systems differ in the way these judgements are formed. In natural deduction, a judgement simply consists of a proposition. In the sequent calculus, however, a judgement consists of two sets of propositions. One set represents the judgement's hypotheses, while the other set represents its consequences. Figure 2.2 shows the syntactical definitions of judgements in both natural deduction and the sequent calculus.

$$\begin{aligned} X, Y, Z \in PropVariable &::= ... \\ A, B, C \in Proposition &::= X \mid \top \mid \bot \mid A \wedge B \mid A \vee B \mid A \subset A \mid \forall X.A \mid \exists X.A \end{aligned}$$

FIGURE 2.1: *Propositions in the sequent calculus and natural deduction*

$$H, J \in Judgement ::= \vdash A$$

Natural deduction

$$\begin{aligned} \Gamma \in Hypothesis &::= A, B, C \\ \Delta \in Consequence &::= A, B, C \\ H, J \in Judgement &::= \Gamma \vdash \Delta \end{aligned}$$

Sequent calculus

FIGURE 2.2: *Judgements in natural deduction and the sequent calculus*

The semantics of both systems are defined by inference rules, which state that zero or more specific premises lead to a certain conclusion. Premises, as well as conclusions, usually consist of judgements, and inference rules are denoted as

$$\frac{\text{Premise } 1 \qquad \text{...} \qquad \text{Premise } n}{\text{Conclusion}}$$

In these inference rules, we differentiate between several types of rules over two dimensions. If a rule introduces a constructor in the conclusion that was not present in the premises, we categorise this rule as an *introduction rule*. When a rule does the exact oposite - leaving a constructor that is a part of the premises out of the conclusion - we call this rule an *elimination rule*. Inference rules often specify introduction or elimination on either the right side, or the left side of the turnstile ($\vdash$) in judgements. Inference rules like these are called *left-hand rules* and *right-hand rules* accordingly. As Figure 2.2 shows, judgements in natural deduction only contain propositions on the right-hand side of the turnstile.

Natural deduction therefore consists of only right-hand rules. Judgements in the sequent calculus do contain propositions on both sides. For this reason, the sequent calculus does contain both left-hand and right-hand rules. However, unlike natural deduction, the sequent calculus has no need for elimination rules, but instead solely consists of introduction rules.

In the sequent calculus, judgements consist of two sets of propositions: hypotheses and consequences, denoted as $\Gamma \vdash \Delta$. Semantically, $\Gamma \vdash \Delta$ means that when every proposition in Γ is true, then at least one proposition in Δ must be true as well. The absence of hypotheses in natural deduction causes the key difference between the sequent calculus and natural deduction: the sequent calculus is conditional, while natural deduction is not.

The distinction between hypotheses and consequences can be used to model truth and falsehood into the sequent calculus. "A is true" can be encoded as $\vdash A$. There are no hypotheses, which means that, by default, all hypotheses are true. Since this is the case, at least one proposition in the set of consequences must be true. Since the set of consequences is a singleton of A, A must always be true. Similarly, "A is false" can be encoded as $A \vdash$. In this case, the judgement is conditional; the set of hypotheses is a singleton of A. Whenever A is true, the judgement claims that at least one proposition in the empty set of consequences must be true. This is, of course, not possible. Therefore, whenever A is true, we end up in a contradiction. Ergo, A can never be true, and it must be false.

The sequent calculus consists of two core rules that form the basis of any logical proof in the calculus. Both are defined in Figure 2.3. The first rule, the *axiom* rule, or Ax, allows any proposition to be introduced on both sides of the turnstile. The second and more interesting one, the *cut* rule, allows intermediate results to be cut out of a proof. For example, if we know a proposition A is both true, and false, we know a contradiction has taken place. We can use the cut rule to prove this:

$$\frac{A \vdash \qquad \vdash A}{\vdash} \; Cut$$

In addition to the core rules, the calculus specifies six *structural rules*. The structural rules - defined in Figure 2.4 are split into three different kinds that have been implemented for both the left-hand side, as well as the right-hand side.

$$\frac{}{A \vdash A}\,Ax \qquad \frac{\Gamma \vdash A, \Delta \qquad \Gamma', A \vdash \Delta'}{\Gamma', \Gamma \vdash \Delta', \Delta}\,Cut$$

FIGURE 2.3: *The core rules of the sequent calculus*

$$\frac{\Gamma \vdash \Delta}{\Gamma \vdash A, \Delta}\,WR \qquad \frac{\Gamma \vdash \Delta}{\Gamma, A \vdash \Delta}\,WL$$

$$\frac{\Gamma \vdash A, A, \Delta}{\Gamma \vdash A, \Delta}\,CR \qquad \frac{\Gamma, A, A \vdash \Delta}{\Gamma, A \vdash \Delta}\,CL$$

$$\frac{\Gamma \vdash \Delta, A, B, \Delta'}{\Gamma \vdash \Delta, B, A, \Delta'}\,XR \qquad \frac{\Gamma', B, A, \Gamma \vdash \Delta}{\Gamma', A, B, \Gamma \vdash \Delta}\,XL$$

FIGURE 2.4: *The structural rules of the sequent calculus*

First, there are the *weakening* rules: WR and WL. The weakening rules allow extra propositions to be introduced into a judgement. This is allowed, since adding an extra proposition to either side of a judgement does not compromise its validity. Assume valid judgement $A, B \vdash C, D$. This judgement means that at least one proposition out of C and D must be true whenever A and B are both true. If we add proposition E to the left-hand side, we get $A, B, E \vdash C, D$. This is still valid, since we now claim that C or D must be true when A, B, and E are true. When this is the case, A and B are still true. If we add E to the right-hand side instead, we get the judgement $A, B \vdash C, D, E$. This, too, is a valid transformation. When A and B are both true, at least one out of C and D must be true. Adding E does not change the validity of the judgement.

The *contraction* rules (CR, CL) and *exchange* rules (XR, XL) are more trivial. The contraction rules specify that duplicated propositions in both the hypotheses and the consequences can be safely ignored. Γ, A, A is functionally the exact same set of hypotheses as Γ, A. The exchange rules formalise how the order of propositions are irrelevant on both the left-hand and the right-hand sides.

2.1.1 LOGICAL RULES

The rules that dictate the behaviour of the different kinds of proposition constructors are called *logical* rules. In this section the logical rules for the sequent calculus will be explained, by comparing them with their natural deduction counterparts. The full definition of all logical rules in this version of the sequent calculus is found in Figure 2.5. Figure 2.6 shows the logical rules for the same proposition constructors for natural deduction.

Let us begin with the very simplest case: truth and falsehood. In the sequent calculus, truth can always be introduced on the right-hand side of a judgement, while falsehood can always be introduced on the left-hand side. Introducing truth or falsehood on the opposite ends is not possible, for this would mean ”true is false”, or ”false is true”. In natural deduction, the introduction rule for truth is effectively the same: $\frac{}{\vdash \top}\top I$. There is no left-hand introduction rule for truth in the sequent calculus. Similarly, there is no elimination rule for truth in natural deduction. If there had been one, it would imply any proposition could always be reduced to truth, even false propositions.

Falsehood is the dual of truth, and its rules reflect this relation. In natural deduction, falsehood can be eliminated using the follwing rule: $\frac{\vdash \bot}{\vdash C}\bot E$. This falsehood elimination rule is effectively a crash-and-burn tactic. Any proposition can be reduced to falsehood, but doing so halts the proof. Since there is no introduction rule for $\bot$, the $\bot$ can never be removed from the judgement.

Disjunction is another example where the implementations of the sequent calculus and natural deduction are quite similar. The sequent calculus' $\wedge R$ rule is virtually identical to natural deduction's $\wedge I$

$$\frac{}{\Gamma \vdash \top, \Delta}\,\top R \qquad \text{No } \top L \text{ rule} \qquad \text{No } \bot R \text{ rule} \qquad \frac{}{\Gamma, \bot \vdash \Delta}\,\bot L$$

$$\frac{\Gamma \vdash A, \Delta \qquad \Gamma \vdash B, \Delta}{\Gamma \vdash A \wedge B, \Delta}\,\wedge R \qquad \frac{\Gamma, A \vdash \Delta}{\Gamma, A \wedge B \vdash \Delta}\,\wedge L_1 \qquad \frac{\Gamma, B \vdash \Delta}{\Gamma, A \wedge B \vdash \Delta}\,\wedge L_2$$

$$\frac{\Gamma \vdash A, \Delta}{\Gamma \vdash A \vee B, \Delta}\,\vee R_1 \qquad \frac{\Gamma \vdash B, \Delta}{\Gamma \vdash A \vee B, \Delta}\,\vee R_2 \qquad \frac{\Gamma, A \vdash \Delta \qquad \Gamma, B \vdash \Delta}{\Gamma, A \vee B \vdash \Delta}\,\vee L$$

$$\frac{\Gamma, A \vdash \Delta}{\Gamma \vdash \neg A, \Delta}\,\neg R \qquad \frac{\Gamma \vdash A, \Delta}{\Gamma, \neg A \vdash \Delta}\,\neg L$$

$$\frac{\Gamma, A \vdash B, \Delta}{\Gamma \vdash A \supset B, \Delta}\,\supset R \qquad \frac{\Gamma \vdash A, \Delta \qquad \Gamma', B \vdash \Delta'}{\Gamma', \Gamma, A \supset B \vdash \Delta', \Delta}\,\supset L$$

$$\frac{\Gamma \vdash A, \Delta \qquad X \notin FV(\Gamma \vdash \Delta)}{\Gamma \vdash \forall X.A, \Delta}\,\forall R \qquad \frac{\Gamma, A\{B/X\} \vdash \Delta}{\Gamma, \forall X.A \vdash \Delta}\,\forall L$$

$$\frac{\Gamma \vdash A\{B/X\}, \Delta}{\Gamma \vdash \exists X.A, \Delta}\,\exists R \qquad \frac{\Gamma, A \vdash \Delta \qquad X \notin FV(\Gamma \vdash \Delta)}{\Gamma, \exists X.A \vdash \Delta}\,\exists L$$

FIGURE 2.5: *Logical rules for truth ($\top$), falsehood ($\bot$), conjunction ($\wedge$), disjunction ($\vee$), negation ($\neg$), implication ($\supset$), universal quantification ($\forall$) and existential quantification ($\exists$) in the sequent calculus.*

$$\frac{}{\vdash \top}\,\top I \qquad \text{No } \top E \text{ rule} \qquad \text{No } \bot I \text{ rule} \qquad \frac{\vdash \bot}{\vdash C}\,\bot E$$

$$\frac{\vdash A \qquad \vdash B}{\vdash A \wedge B}\,\wedge I \qquad \frac{\vdash A \wedge B}{\vdash A}\,\wedge E_1 \qquad \frac{\vdash A \wedge B}{\vdash B}\,\wedge E_2$$

$$\frac{\vdash A}{\vdash A \vee B}\,\vee I_1 \qquad \frac{\vdash B}{\vdash A \vee B}\,\vee I_2 \qquad \frac{\vdash A \vee B \qquad \begin{array}{c}\overline{\vdash A}^{\,x}\\ \vdots\\ \vdash C\end{array} \qquad \begin{array}{c}\overline{\vdash B}^{\,y}\\ \vdots\\ \vdash C\end{array}}{\vdash C}\,\vee E_{x,y}$$

$$\frac{\begin{array}{c}\overline{\vdash A}^{\,x}\\ \vdots\\ \vdash B\end{array}}{\vdash A \supset B}\,\supset I_x \qquad \frac{\vdash A \supset B \qquad \vdash A}{\vdash B}\,\supset E$$

$$\frac{\begin{array}{c}\vdots \quad (X \notin FV(*))\\ \vdash A\end{array}}{\vdash \forall X.A}\,\forall I_X \qquad \frac{\vdash \forall X.A}{\vdash A\{B/X\}}\,\forall E$$

$$\frac{\vdash A\{B/X\}}{\vdash \exists X.A}\,\exists I \qquad \frac{\vdash \exists X.A \qquad \begin{array}{c}\overline{\vdash A}^{\,x}\\ \vdots \quad (X \notin FV(*))\\ \vdash C\end{array} \qquad (X \notin FV(C))}{C}\,\exists E_{X,x}$$

FIGURE 2.6: *Logical rules for truth ($\top$), falsehood ($\bot$), conjunction ($\wedge$), disjunction ($\vee$), implication ($\supset$), universal quantification ($\forall$) and existential quantification ($\exists$) in natural deduction.*

$$\dfrac{\begin{array}{c}\dfrac{}{\vdash A}\,x\\ \vdots\\ \vdash B\end{array}}{\vdash A \supset B}\supset I_x$$

Natural deduction

$$\dfrac{\Gamma, A \vdash B, \Delta}{\Gamma \vdash A \supset B, \Delta}\supset R$$

Sequent calculus

FIGURE 2.7: *Implication in natural deduction and the sequent calculus*

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash (A \wedge B) \wedge C}\,x}{\vdash A \wedge B}\wedge E_1}{\vdash A}\wedge E_1 \qquad \dfrac{\dfrac{}{\vdash (A \wedge B) \wedge C}\,x}{\vdash C}\wedge E_2}{\vdash A \wedge C}\wedge I}{\vdash (A \wedge B) \wedge C \supset A \wedge C}\supset I_x$$

Natural deduction

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{A \vdash A}\,Ax}{A \wedge B \vdash A}\wedge L_1}{(A \wedge B) \wedge C \vdash A}\wedge L_1 \qquad \dfrac{\dfrac{}{C \vdash C}\,Ax}{(A \wedge B) \wedge C \vdash C}\wedge L_2}{(A \wedge B) \wedge C \vdash A \wedge C}\wedge R}{\vdash (A \wedge B) \wedge C \supset A \wedge C}\supset R$$

Sequent calculus

FIGURE 2.8: *An example proof of the proposition $(A \wedge B) \wedge C \supset A \wedge C$ in the two logical systems.*

rule: $A \wedge B$ can be introduced when we know that both A and B are true. The elimination rules are straight forward too. If we know that $A \wedge B$ holds, we can safely eliminate either proposition to end up with A or B. The sequent calculus does not have elimination rules, of course, but instead defines behaviour for the left-hand side. Take the special case of $\wedge L_1$ where the starting judgement is $A \vdash C$. This judgement claims that C must be true if A is true. If we were to change the hypothesis A to $A \wedge B$, C would still be true, as $A \wedge B$ dictates that A must be true. We thus end up with $\dfrac{A \vdash C}{A \wedge B \vdash C}$. This step can be generalised to rule $\wedge L_1$. Since $\wedge$ is symmetric, we can use the same steps to gain rule $\wedge L_2$.

The rules for implication display the true power of the sequent calculus. Natural deduction's $\supset I_x$ rule handles implication by creating a new verification rule x and allowing this rule to be used somewhere on above the right-hand side of the implication. Since the sequent calculus is conditional, it is not necessary to create new rules. The proposition on the left-hand side of the implication simlpy gets moved to the hypotheses, while the right-hand side remains in the consequences. This allows for a much simpler and more straight forward rule, and by proxy, proofs. Figure 2.7 contains the right introduction rules for implication in both systems.

Figure 2.8 shows example proofs for the proposition $(A \wedge B) \wedge C \supset A \wedge C$ in both natural deduction and the sequent calculus. The two proof trees display the implications of the different approaches taken by the two systems. In both systems, the implication is first reduced to the right-hand side $A \wedge C$, after which the conjunction is split into A and C. In natural deduction, the two propositions need to be expanded to $(A \wedge B) \wedge C$ using elimination rules, before the generated verification rule x can be utilised to finish the proof. In the sequent calculus, the system 'remembers' $(A \wedge B) \wedge C$ as a hypothesis, and this can be reduced to either A or C.

Another advantage the sequent calculus has over natural deduction is its neutrality. The sequent calculus can be used to prove both true and false judgements, whereas natural deduction favours truth over falsehood. This neutrality is used to implement negation ($\neg$) into the sequent calculus. In natural deduction, this is not possible. Instead, a negation of A can be implied by proving the judgement $A \supset \bot$ as follows:

$$\dfrac{\begin{array}{c}\dfrac{}{\vdash A}\,x \\ \vdots \\ \vdash \bot\end{array}}{\vdash A \supset \bot} \supset I_x$$

Using this encoding, it is possible to prove $\neg\bot$, or "false is false".

$$\dfrac{\dfrac{}{\vdash \bot}\,x}{\vdash \bot \supset \bot} \supset I_x$$

A final remark on the logical rules for both the sequent calculus and natural deduction is about the quantification rules. The rules for both existential and universal quantification in both systems contain *side conditions*. Let us look at the right rule and the introduction rule for universal quantification.

$$\dfrac{\Gamma \vdash A, \Delta \qquad (X \notin FV(\Gamma \vdash \Delta))}{\Gamma \vdash \forall X.A, \Delta} \supset R \qquad\qquad \dfrac{\begin{array}{c}\vdots\ (X \notin FV(*)) \\ \vdash A\end{array}}{\vdash \forall X.A} \forall I_X$$

Both rules contain a side condition: $(X \notin FV(\Gamma \vdash \Delta))$ and $(X \notin FV(*))$ respectively. Here, FV refers to a function that returns all the *free variables* in its argument. A free variable is a variable that is not bound by any quantifiers. For example, the result of $FV(\forall X.(X \vee Y \vee Z))$ would be the set of Z and Y.

Another reoccurring concept in the logical rules is *substitution*. The substitution of B for X in proposition A is denoted as $A\{B/X\}$. This means that every *free* occurrence of X within A is replaced by B. Substitution is used in - among others - the $\exists R$ rule.

$$\dfrac{\Gamma \vdash A\{B/X\}, \Delta}{\Gamma \vdash \exists X.A, \Delta} \exists R$$

This rule permits the introduction of existential quantifieres over any free variable. For example, this rule can be used to transform the judgement $\Gamma \vdash A \wedge B, \Delta$ to $\Gamma \vdash \exists X.X \wedge B, \Delta$, as $A \wedge B$ is equivalent to $X \wedge B\{A/X\}$.

2.2 THE $\mu\tilde{\mu}$-CALCULUS

In this section the $\mu\tilde{\mu}$-calculus is introduced and fully specified. First, we recall some notions about the λ-calculus. Then, the simplest version of the $\mu\tilde{\mu}$-calculus is defined, and compared to the λ-calculus. Next, the $\mu\tilde{\mu}$-calculus is extended with advanced types and type constructors, and the effects of this on typing and evaluation rules are explained. Finally, function types are introduced using these concepts, and examples of programming with the $\mu\tilde{\mu}$-calculus are given.

2.2.1 THE λ-CALCULUS

Alonzo Church invented the λ-calculus [Chu36], a minimalistic core programming language based on natural deduction, in 1936. The λ-calculus in itself consists of just three components: variables, functions and term application.

$$\begin{aligned} x, y, z \in Variable &::= ... \\ M, N \in Term &::= x \mid \lambda x.M \mid M\ N \end{aligned}$$

The dynamic behaviour of these terms is defined by three simple rules:

$$\begin{aligned} \alpha\text{-equivalence: } & \lambda x.M =_\alpha \lambda y.M \\ \beta\text{-reduction: } & (\lambda x.M)\ N \succ_\beta M\{N/x\} \\ \eta\text{-conversion: } & (\lambda x.M\ x) \succ \eta M \qquad\qquad (x \notin FV(M)) \end{aligned}$$

Once we start adding types according to Church's *simply typed λ-calculus* [Chu40], the similarities between the calculus and natural deduction become striking. First, we add a function type ($\rightarrow$) and implement typing rules for functions and function applications. Functions can be compared to implication in natural deduction. Where $A \supset B$ means "if A is true, B is true," function $\lambda x.M$ with type $A \rightarrow B$ can be seen as "if x has type A, M has type B." This way, a λ-function reflects the introduction rule for implications, as defined in Figure 2.6. Implication's elimination rule is useful too. Where $A \supset B$ and A are true, B must be true as well. Similarly, where M has type $A \rightarrow B$, and N has type A, $M\ N$ must have type B. Using the connections between functions and implication, we are able to deduce the introduction and elimination rules for functions: the $\rightarrow I_x$ and the $\rightarrow E$ rules in Figure 2.9.

Next, we add product types to the calculus. To do so, we extend our definition of terms by pairs and projections, and add a product type:

$$\begin{aligned} M, N \in Term &::= \cdots \mid (M, N) \mid \pi_1(M) \mid \pi_2(M) \\ A, B, C \in Type &::= X \mid A \rightarrow B \mid A \times B \end{aligned}$$

Dynamically, $\pi_1(M, N)$ evaluates to M, while $\pi_2(M, N)$ evaluates to N. In other words, π_1 and π_2 eliminate terms of product types. Like functions, product types bear a strong resemblance to a propositional constructor in natural deduction: conjunction. Where M has type A, and N has type B, (M, N) must have type $A \times B$. This reflects the behaviour of conjunction: if A is true and B is true, $A \wedge B$ must be true. Once again, we can infer the typing rules from natural deduction. The $\times I$, $\times E_1$ and $\times E_2$ rules are added to the calculus in Figure 2.9.

Finally, we add sum types ($+$), and extend the definition of terms once more:

$$\begin{aligned} M, N \in Term &::= \cdots \mid \iota_1(M) \mid \iota_2(M) \\ &\qquad \mid (case\ M\ of\ \iota_1(x) \Rightarrow N_1 \mid \iota_2(y) \Rightarrow N_2) \\ A, B, C \in Type &::= X \mid A \rightarrow B \mid A \times B \mid A + B \end{aligned}$$

With these new terms come more evaluation rules. $case\ \iota_1(M)\ of\ \iota_1(x) \Rightarrow N_1 \mid \iota_2(y) \Rightarrow N_2$ evaluates to $N_1\{M/x\}$, while $case\ \iota_2(M)\ of\ \iota_1(x) \Rightarrow N_1 \mid \iota_2(y) \Rightarrow N_2$ evaluates to $N_2\{M/x\}$. In a way, sum types are the dual of product types. While products consist of one constructor that takes two terms, and two destructors, sums consist of two constructors that take one term, and one destructor. It may come as no surprise that product types heavily resemble disjunction. As seen before, we derive the product typing rules from natural deduction's disjunction rules, and define the $+I_1$, $+I_2$ and $+E_{x,y}$ in Figure 2.9.

The complete definition of this version of the λ-calculus is found in Figure 2.9.

$$
\begin{aligned}
X, Y, Z \in TypeVariable &::= ... \\
A, B, C \in Type &::= X \mid A \to B \mid A \times B \mid A + B \\
x, y, z \in Variable &::= ... \\
M, N \in Term &::= x \mid \lambda x.M \mid M\,N \mid (M, N) \\
&\mid \pi_1(M) \mid \pi_2(M) \mid \iota_1(M) \mid \iota_2(M) \\
&\mid (case\, M\, of\, \iota_1(x) \Rightarrow N_1 \mid \iota_2(y) \Rightarrow N_2) \\
H, J \in Judgement &::= M : A
\end{aligned}
$$

$$
\frac{M : A \qquad N : B}{(M, N) : A \times B}\times I \qquad \frac{M : A \times B}{\pi_1(M) : A}\times E_1 \qquad \frac{M : A \times B}{\pi_2(M) : B}\times E_2
$$

$$
\frac{M : A}{\iota_1(M) : A + B} +I_1 \qquad \frac{M : B}{\iota_2(M) : A + B} +I_2
$$

$$
\frac{M : A + B \qquad \begin{matrix}\overline{x : A}^{\,x} \\ \vdots \\ N_1 : C\end{matrix} \qquad \begin{matrix}\overline{y : B}^{\,y} \\ \vdots \\ N_2 : C\end{matrix}}{\textbf{case}\ M\ \textbf{of}\ \iota_1(x) \Rightarrow N_1 \mid \iota_2(y) \Rightarrow N_2 : C} +E_{x,y}
$$

$$
\frac{\begin{matrix}\overline{x : A}^{\,x} \\ \vdots \\ M : B\end{matrix}}{\lambda x.M : A \to B} \to I_x \qquad \frac{M : A \to B \qquad N : A}{M\,N : B} \to E
$$

FIGURE 2.9: *The simply typed λ-calculus with function ($\to$), sum ($+$) and product ($\times$) types.*

2.2.2 THE $\mu\tilde{\mu}$-CALCULUS

We have seen how the λ-calculus relates to natural deduction. In this section we will explore the results of creating a programming language based on the sequent calculus: the $\mu\tilde{\mu}$-calculus. The $\mu\tilde{\mu}$-calculus originally stems from Herbelin [Her05]. Since the sequent calculus is ambidextrous, meaning that it has both right and left-hand rules, so is the $\mu\tilde{\mu}$-calculus. In the context of programming, this translates to the calculus differentiating between *terms* and *co-terms*: Terms on the left side of the turnstile, co-terms on the right side.

Terms are comparable to the terms in the λ-calculus. They consist of variables and various constructors, which together represent pieces of data. Co-terms are the dual to terms. They are not formed by constructors, but rather destructors, observers, or consumers. Co-terms represent co-data, or co-inductively defined data. Co-data is often used to represent infinite structures, such as infinite streams. Terms and co-terms do not have separate type systems; a term can have the same type as a co-term.

In the $\mu\tilde{\mu}$-calculus, computation happens in ***commands***: structures where a term and a co-term of the same type come together.

$$c \in Command ::= \langle Term \parallel CoTerm \rangle$$

This may seem complicated, especially when compared to the λ-calculus, where computation simply happens in function applications. However, recall that terms are producers of a type where co-terms are consumers. Commands allow producers and consumers to interact with each other, which is the essence of computation. After all, producing data which cannot be consumed is not very useful, nor is a consumer of data that cannot be produced.

Terms and co-terms can interact using μ and $\tilde{\mu}$-constructions.

$$\begin{aligned} x, y, z \in Variable &::= \ldots & \alpha, \beta, \gamma \in CoVariable &::= \ldots \\ v \in Term &::= x \mid \mu\alpha.c & e \in CoTerm &::= \alpha \mid \tilde{\mu}x.c \end{aligned}$$
$$c \in Command ::= \ \langle v \parallel e \rangle$$

At first glance, μ and $\tilde{\mu}$ look to be the $\mu\tilde{\mu}$-calculus equivalent of the λ-calculus' λ. This is, however, not the case. λ-constructions are actual functions. They take an argument and produce a term that may use this argument. μ and $\tilde{\mu}$-constructions take an argument, a co-term and a term respectively, and execute a command that may use this argument. They do not return any value by themselves, and are therefore not functions. They are perhaps more accurately compared to void methods, as found in many imperative programming languages, such as C.

2.2.2.1 TYPING

Judgements in the sequent calculus consist of a set of hypotheses and a set of consequences. Judgements in the $\mu\tilde{\mu}$-calculus are similar. Hypotheses are replaced by an input environment, which stores variables and their types. Consequences are replace by an output environment, which does exactly the same for co-variables. The judgements come in different shapes: passive and active. Passive judgements, which are denoted by $c : (\Gamma \vdash \Delta)$, simply bind an input environment and an output environment to a command. Then there are active judgements: $(\Gamma \vdash v : A \mid \Delta)$ and $(\Gamma \mid e : A \vdash \Delta)$. Active judgements highlight one proposition, which is called the active proposition. The active proposition is the proposition that we currently want to prove. It can therefore not be removed by using weakening rules in a proof tree. The full definition of the core $\mu\tilde{\mu}$-calculus is given in Figure 2.10.

Since judgements have been extended from the sequent calculus, the Ax and Cut rule are not sufficient anymore for the $\mu\tilde{\mu}$-calculus. Instead, we now split the Ax rule into two verification rules: VR and VL. These rules complete a proof of a right active judgement and a left active judgement respectively.

$$\frac{}{x : A \vdash x : A \mid}\ VR \qquad \frac{}{\mid \alpha : A \vdash \alpha : A}\ VL$$

$$
\begin{array}{rlrl}
A,B,C \in Type & ::= X & X,Y,Z \in TypeVariable & ::= \ldots \\
x,y,z \in Variable & ::= \ldots & \alpha,\beta,\gamma \in CoVariable & ::= \ldots \\
v \in Term & ::= x \mid \mu\alpha.c & e \in CoTerm & ::= \alpha \mid \tilde{\mu}x.c \\
c \in Command & ::= \langle v \parallel e \rangle & & \\
\Gamma \in InputEnv & ::= x_1 : A_1, \ldots, x_n : A_n & & \\
\Delta \in OutputEnv & ::= \alpha_1 : A_1, \ldots, \alpha_n : A_n & & \\
Judgement & ::= c : (\Gamma \vdash \Delta) \mid (\Gamma \vdash v : A \mid \Delta) \mid (\Gamma \mid e : A \vdash \Delta) & &
\end{array}
$$

Core rules:

$$\frac{}{x : A \vdash x : A \mid}\ VR \qquad \frac{}{\mid \alpha : A \vdash \alpha : A}\ VL$$

$$\frac{c : (\Gamma \vdash \alpha : A, \Delta)}{\Gamma \vdash \mu\alpha.c : A \mid \Delta}\ AR \qquad \frac{c : (\Gamma, x : A \vdash \Delta)}{\Gamma \mid \tilde{\mu}x.c : A \vdash \Delta}\ AL$$

$$\frac{\Gamma \vdash v : A \mid \Delta \qquad \Gamma' \mid e : A \vdash \Delta'}{\langle v \parallel e \rangle : (\Gamma', \Gamma \vdash \Delta', \Delta)}\ Cut$$

FIGURE 2.10: *The core $\mu\tilde{\mu}$-calculus*

Next, we alter the cut rule to adopt the new judgement definitions. Whereas before it cut a proposition out of the hypotheses and the consequences, it now cuts out an active left judgement and an active right judgement, and combines them into a command. In practice, this rule forces the term and the co-term in a command to be of the same type.

$$\frac{\Gamma \vdash v : A \mid \Delta \qquad \Gamma' \mid e : A \vdash \Delta'}{\langle v \parallel e \rangle : (\Gamma', \Gamma \vdash \Delta', \Delta)}\ Cut$$

Finally, we add activation rules, which allow a passive judgement to be turned into an active one by creating either a μ or a $\tilde{\mu}$-construction.

$$\frac{c : (\Gamma \vdash \alpha : A, \Delta)}{\Gamma \vdash \mu\alpha.c : A \mid \Delta}\ AR \qquad \frac{c : (\Gamma, x : A \vdash \Delta)}{\Gamma \mid \tilde{\mu}x.c : A \vdash \Delta}\ AL$$

2.2.2.2 EVALUATION

Like the λ-calculus, the core $\mu\tilde{\mu}$-calculus evaluates according to serveral simple rules. First, the η_μ and $\eta_{\tilde{\mu}}$ reduction rules. These allow μ and $\tilde{\mu}$-constructions to be removed when their argument is directly applied in its command. As the name suggests, the η_μ and $\eta_{\tilde{\mu}}$ rules are similar to the λ-calculus' η rule.

$$\mu\alpha.\langle v \parallel \alpha \rangle \succ_{\eta_\mu} v \qquad \tilde{\mu}x.\langle x \parallel e \rangle \succ_{\eta_{\tilde{\mu}}} e$$

Next, we have the μ and $\tilde{\mu}$-rewriting rules. These specify how commands can be evaluated, similar to β-reduction in the λ-calculus.

$$\langle \mu\alpha.c \parallel e \rangle \succ_\mu c\{e/\alpha\} \qquad \langle v \parallel \tilde{\mu}x.c \rangle \succ_{\tilde{\mu}} c\{v/x\}$$

The attentive reader may have noticed that the μ and $\tilde{\mu}$-rewriting rules get into conflict in specific situations. What happens with a command that has a μ-construction on the left side, and a $\tilde{\mu}$-construction on the right side? With the current rules, we could rewrite $\langle \mu\alpha.c_1 \parallel \tilde{\mu}x.c_2 \rangle$ to both $c_1\{\tilde{\mu}x.c_2/\alpha\}$ and $c_2\{\mu\alpha.c_1/x\}$. Which result is the correct one?

Actually, both results are correct, it is simply a matter of preference. As Curien and Herbelin formalised, the two evaluation options represent different evaluation strategies: call-by-name and call-by-value [CH00]. Call-by-value prioritises μ-rewriting over $\tilde{\mu}$-rewriting, while call-by-name prioritises $\tilde{\mu}$-rewriting over μ-rewriting.

2.2.3 ADDING CONSTRUCTORS AND TYPES

So far, we have seen the $\mu\tilde{\mu}$-calculus where all types are atomic type variables. The next step is to add basic types connectors and associated constructors, as we have done for the λ-calculus in Figure 2.9.

First, we will add product types ($\times$). Once again, we define tuples as a constructor for product types. However, where the λ-calculus has π_1 and π_2 as terms, they are co-terms in the $\mu\tilde{\mu}$-calculus.

$$v \in Term ::= \cdots \mid (v, v) \qquad e \in CoTerm ::= \cdots \mid \pi_1[e] \mid \pi_2[e]$$

Just like for the λ-calculus, we can use the logical foundations of the $\mu\tilde{\mu}$-calculus in the definitions of the typing rules for products. We have previously established the relationship between product types and conjunction. We use the logical rules for conjunction to derive typing rules.

$$\frac{\Gamma \vdash A, \Delta \qquad \Gamma \vdash B, \Delta}{\Gamma \vdash A \wedge B, \Delta}\ \wedge R \qquad \frac{\Gamma \vdash v : A \mid \Delta \qquad \Gamma \vdash v' : B \mid \Delta}{\Gamma \vdash (v, v') : A \times B \mid \Delta}\ \times R$$

$$\frac{\Gamma, A \vdash \Delta}{\Gamma, A \wedge B \vdash \Delta}\ \wedge L_1 \qquad \frac{\Gamma \mid e : A \vdash \Delta}{\Gamma \mid \pi_1[e] : A \times B \vdash \Delta}\ \times L_1$$

$$\frac{\Gamma, B \vdash \Delta}{\Gamma, A \wedge B \vdash \Delta}\ \wedge L_2 \qquad \frac{\Gamma \mid e : B \vdash \Delta}{\Gamma \mid \pi_2[e] : A \times B \vdash \Delta}\ \times L_2$$

Note that π_1 and π_2 take co-terms as arguments, instead of terms. Unlike in the λ-calculus, it is not possible to insert a term (v, v') into π_1 and get v as a result. Instead, a consumer of v is passed to π_1 This allows for the following reduction rules:

$$\langle (v, v') \parallel \pi_1[e] \rangle \succ_{\pi_1} \langle v \parallel e \rangle$$
$$\langle (v, v') \parallel \pi_2[e] \rangle \succ_{\pi_2} \langle v' \parallel e \rangle$$

It may come as no surprise that we add sum types to the calculus next. Sum types consist of two constructors, ι_1 and ι_2, and one consumer. In the λ-calculus, this consumer was modeled as a $case\ of$ expression. In the $\mu\tilde{\mu}$-calculus, the syntax $[e, e']$ is used, meaning a pair of two co-terms. For intuition's sake, this can be read as

$$case\ v\ of\ \iota_1(x) \Rightarrow \langle x \parallel e \rangle \parallel \iota_2(y) \Rightarrow \langle y \parallel e' \rangle$$

In words, whenever $[e, e']$ encounters a term v within a command, this resolves to a computation of the argument to ι_1 and e, or the argument to ι_2 and e', according to the shape of v. Formally:

$$\langle \iota_1(x) \parallel [e, e'] \rangle \succ_{\iota_1} \langle x \parallel e \rangle$$
$$\langle \iota_2(y) \parallel [e, e'] \rangle \succ_{\iota_2} \langle y \parallel e' \rangle$$

The typechecking rules can be derived from the logical rules for disjunction in the sequent calculus once more.

$$\frac{\Gamma \vdash A, \Delta}{\Gamma \vdash A \vee B, \Delta} \vee R_1 \qquad \frac{\Gamma \vdash v : A \mid \Delta}{\Gamma \vdash \iota_1(v) : A + B \mid \Delta} +R_1$$

$$\frac{\Gamma \vdash B, \Delta}{\Gamma \vdash A \vee B, \Delta} \vee R_2 \qquad \frac{\Gamma \vdash v : B \mid \Delta}{\Gamma \vdash \iota_2(v) : A + B \mid \Delta} +R_2$$

$$\frac{\Gamma, A \vdash \Delta \qquad \Gamma, B \vdash \Delta}{\Gamma, A \vee B \vdash \Delta} \vee L \qquad \frac{\Gamma \mid e : A \vdash \Delta \qquad \Gamma \mid e' : B \vdash \Delta}{\Gamma \mid [e, e'] : A + B \vdash \Delta} +L$$

2.2.4 User-defined (co-)datatypes

So far we have extended the calculus with sum and product types. Using these together allow complex data structures to be modeled. However, this is not intuitive, since the structures cannot be named. As an alternative, Downen added user-defined (co-)datatypes to the $\mu\tilde{\mu}$-calculus in his PhD thesis [Dow17].

The calculus is extended with declarations of both datatypes and co-datatypes. Datatypes consist of any number of constructors. Co-datatypes are defined by any number of observers instead. Both constructors and observers can take any number of arguments, which can be both terms and co-terms. The (co-)datatype declarations specify of which types the arguments to constructors and observers should be. The constructors and observers can be used as terms and co-terms respectively.

$$\begin{aligned}
A, B, C \in Type &::= X \mid F(\vec{A}) \\
F, G \in Connector &::= ... \\
K \in Constructor &::= ... \\
O \in Observer &::= ... \\
decl \in Declaration &::= data\, F(\vec{X})\, where\, \overrightarrow{K : \vec{A} \vdash F(\vec{X}) \mid \vec{B}} \\
&\quad\ \mid codata\, G(\vec{X})\, where\, \overrightarrow{O : \vec{A} \mid F(\vec{X}) \vdash \vec{B}}
\end{aligned}$$

In this notation of declarations, each data declaration can define any number of constructors. Each constructor has its own type system $\vec{A} \vdash F(\vec{X}) \mid \vec{B}$. Here, the vector $\vec{A}$ contains all the types of the term arguments that the constructor takes. Similarly, $\vec{B}$ represents the collection of the types of the co-term arguments that the constructor takes. $F(X)$ is the type that the constructor will have with all of its arguments. The type constructor declaration $K : \vec{A} \vdash F(\vec{X}) \mid \vec{B}$ essentially means that constructor K has type $F(\vec{X})$, given that it is supplied with term arguments with the types $\vec{A}$, and co-term arguments with the types $\vec{B}$. This syntax works the exact same way for co-datatype declarations and observers, of course. As an example, product and sum types can be implemented this way:

$$\begin{aligned}
&data\, A \times B\, where\ \ (\_, \_) : A, B \vdash A \times B \mid \\
&data\, A + B\, where\ \ \iota_1(\_) : A \vdash A + B \mid \\
&\qquad\qquad\qquad\quad \mid \iota_2(\_) : B \vdash A + B \mid
\end{aligned}$$

Underscores (_) are used as placeholders for (co-)terms in the constructors of both declarations, meaning that for the constructor (x, y) with type $A \times B$, x must have type A, and y must have type B.

Using these user-defined declarations of sum and product types has one drawback in comparison to the hard-coded variant that we have seen in Section 2.2.3: the declarations do not offer ways to consume the created data. To be able to do so anyway, the μ and $\tilde{\mu}$ constructions are altered. Whereas before, $\tilde{\mu}$ took a variable as its argument, it will now take a *pattern*. Likewise, μ will no longer take a co-variable, but a co-pattern instead. Patterns consist of either a single variable, or a constructor with any number of sub(co-)variables. Of course, co-patterns are the exact same thing, but using co-variables and observers instead of variables and constructors. In addition to the introduction of patterns, μ and $\tilde{\mu}$ can now take multiple pairs of patterns and commands.

$$\begin{aligned}
p \in Pattern &::= x \mid K(\vec{\alpha}, \vec{x}) \\
\tilde{p} \in CoPattern &::= \alpha \mid O[\vec{x}, \vec{\alpha}] \\
v \in Term &::= x \mid \mu(\overrightarrow{\tilde{p}.c}) \mid K(\vec{e}, \vec{v}) \\
e \in CoTerm &::= \alpha \mid \tilde{\mu}[\overrightarrow{p.c}] \mid O[\vec{v}, \vec{e}]
\end{aligned}$$

Formal typing rules for these generalised (co-)data declarations are needed. Because of the nested structure of the constructors and observers, these rules are very complicated. for simplicity's sake, let us first look at sum types, and generalise from there. As we have just seen, sum types consist of two constructors: ι_1 and ι_2. Both take one term argument. When we pass a term v of type A to ι_1, the new term has type $A + B$. If we instead pass v to ι_2, the result's type is $B + A$. So, given that we have

$$\begin{aligned}
data\, A + B\, where\ & \iota_1(\_) : A \vdash A + B \mid \\
& \mid \iota_2(\_) : B \vdash A + B \mid
\end{aligned}$$

we know both constructors turn an argument into a sum type as follows:

$$\frac{\Gamma \vdash v : A \mid \Delta}{\Gamma \vdash \iota_1(v) : A + B \mid \Delta}\ \iota_1 \qquad \frac{\Gamma \vdash v : B \mid \Delta}{\Gamma \vdash \iota_2(v) : A + B \mid \Delta}\ \iota_2$$

For a generic data declaration $data\ F(\vec{X})\ where\ \overrightarrow{K_i : \overrightarrow{A_{ij}}^j \vdash F(\vec{X}) \mid \overrightarrow{B_{ij}}^j}^i$, this strategy gives us the following typing rule:

$$\frac{\overrightarrow{\Gamma \mid e : B_{ij}\{\overrightarrow{C/X}\} \vdash \Delta}^j \qquad \overrightarrow{\Gamma \vdash v : A_{ij}\{\overrightarrow{C/X}\} \mid \Delta}^j}{\Gamma \vdash K_i(\vec{e}, \vec{v}) : F(\vec{C}) \mid \Delta}\ FR_{K_i}$$

In words, this rule tells us that for each constructor in the declaration, we are able to construct type F, substituting X for whatever types we decide to use, as long as they match the declaration. In the context of sum types, F would be $+$, X would be A and B, K_1 would be ι_1, K_2 would be ι_2, A_1 would be a single A, A_2 a single B, and both B_1 and B_2 would be empty. As an example, assuming a variable x with type C, this rule can be used to prove that $\iota_1(x)$ has type $C + B$.

$$\frac{\overline{\Gamma \vdash x : C \mid \Delta}\ VR}{\Gamma \vdash \iota_1(x) : C + B \mid \Delta}\ FR_{\iota_1}$$

The same logic can be used to define typing rules for co-datatypes and observers. For a given declaration $codata\ G(\vec{X})\ where\ \overrightarrow{O_i : \overrightarrow{A_{ij}}^j \mid G(\vec{X}) \vdash \overrightarrow{B_{ij}}^j}^i$, we have the rule

$$\frac{\overrightarrow{\Gamma \vdash v : A_{ij}\{\overrightarrow{C/X}\} \mid \Delta}^j \qquad \overrightarrow{\Gamma \mid e : B_{ij}\{\overrightarrow{C/X}\} \vdash \Delta}^j}{\Gamma \mid O_i[\vec{v}, \vec{e}] : G(\vec{C}) \vdash \Delta}\ GL_{O_i}$$

The GL rule is the exact same as the FR rule, except it introduces an observer instead of a constructor.

Observers and constructors are not the only extensions to the calculus we have made. (Co-)patterns have been added as well, which have caused μ and $\tilde{\mu}$ definitions to be altered. Before, activation rules AR and AL could be used to typecheck μ and $\tilde{\mu}$ (co-)terms. With the renewed definitions, those rules are not going to be sufficient anymore. Let us analyse the AR rule to find its essence, which can be used for a new rule that includes patterns.

$$\frac{c : (\Gamma \vdash \alpha : A, \Delta)}{\Gamma \vdash \mu\alpha.c : A \mid \Delta}\ AR$$

The AR rule allows us to turn a command into a μ-term, using a co-variable that exists in the output environment as the parameter. The new rule should do the same thing, but for a co-pattern. In practice, this means that every (co-)variable in the co-pattern must be known, and that the observer in the co-pattern must be well-typed. We introduce two functions. One that takes a pattern or co-pattern and a type, and matches every variable that occurs in the (co-)pattern with the correct type, to finally return an input environment containing all the variables in the original (co-)pattern. The other function we introduce is one that does the exact same, but with co-variables, which will therefore return an output environment.

$$VarsTypes(x, C) = x : C$$
$$VarsTypes(\alpha, C) = \emptyset$$
$$\text{Given } data\ F(\vec{X})\ where\ \overrightarrow{K_i : \overrightarrow{A_{ij}}^j \vdash F(\vec{X}) \mid \overrightarrow{B_{ij}}^j}^i$$
$$VarsTypes(K_i(\vec{\alpha_i}, \vec{x_i}), F(\vec{C})) = \overrightarrow{x_{ij} : A_{ij}\{C/X\}}^j$$
$$\text{Given } codata\ G(\vec{X})\ where\ \overrightarrow{O_i : \overrightarrow{A_{ij}}^j \mid G(\vec{X}) \vdash \overrightarrow{B_{ij}}^j}^i$$
$$VarsTypes(O_i[\vec{x_i}, \vec{\alpha_i}], G(\vec{C})) = \overrightarrow{x_{ij} : A_{ij}\{C/X\}}^j$$

$$CoVarsTypes(\alpha, C) = \alpha : C$$
$$CoVarsTypes(x, C) = \emptyset$$
$$\text{Given } codata\ G(\vec{X})\ where\ \overrightarrow{O_i : \overrightarrow{A_{ij}}^j \mid G(\vec{X}) \vdash \overrightarrow{B_{ij}}^j}^i$$
$$CoVarsTypes(O_i[\vec{x_i}, \vec{\alpha_i}], G(\vec{C})) = \overrightarrow{\alpha_{ij} : B_{ij}\{C/X\}}^j$$
$$\text{Given } data\ F(\vec{X})\ where\ \overrightarrow{K_i : \overrightarrow{A_{ij}}^j \vdash F(\vec{X}) \mid \overrightarrow{B_{ij}}^j}^i$$
$$CoVarsTypes(K_i(\vec{\alpha_i}, \vec{x_i}), F(\vec{C})) = \overrightarrow{\alpha_{ij} : B_{ij}\{C/X\}}^j$$

As an example, the result of $VarsTypes((x, y),\ A \times (B \times C))$ would be the environment $x : A, y : B \times C$. Now that we are able to process entire patterns and co-patterns, we can define new rules for μ and $\tilde{\mu}$ constructions.

$$\frac{\overrightarrow{c_i : (\Gamma, VarsTypes(p_i, A) \vdash CoVarsTypes(p_i, A), \Delta)}^i}{\Gamma \mid \tilde{\mu}[\overrightarrow{p_i.c_i}^i] : A \vdash \Delta}\ FL$$

$$\frac{\overrightarrow{c_i : (\Gamma, VarsTypes(p_i, A) \vdash CoVarsTypes(p_i, A), \Delta)}^i}{\Gamma \vdash \mu(\overrightarrow{\tilde{p}_i.c_i}^i) : A \mid \Delta}\ GR$$

These rules can be used to typecheck μ and $\tilde{\mu}$ constructions, such as the judgement $\Gamma \mid \tilde{\mu}[\iota_1(x).c_1 \mid \iota_2(y).c_2] : X + Y \vdash \Delta$:

$$\frac{c_1 : (\Gamma, x : X \vdash \Delta) \qquad c_2 : (\Gamma, y : Y \vdash \Delta)}{\Gamma \mid \tilde{\mu}[\iota_1(x).c_1 \mid \iota_2(y).c_2] : X + Y \vdash \Delta}\ FL$$

Figure 2.11 shows the full definition of the $\mu\tilde{\mu}$-calculus with user-defined (co-)datatypes and its typing rules.

$$\begin{array}{rlrl}
A, B, C \in \mathit{Type} & ::= X \mid F(\overrightarrow{A}) & X, Y, Z \in \mathit{TypeVariable} & ::= \ldots \\
F, G \in \mathit{Connector} & ::= \ldots & & \\
K \in \mathit{Constructor} & ::= \ldots & O \in \mathit{Observer} & ::= \ldots \\
x, y, z \in \mathit{Variable} & ::= \ldots & \alpha, \beta, \gamma \in \mathit{CoVariable} & ::= \ldots \\
p \in \mathit{Pattern} & ::= x \mid K(\overrightarrow{\tilde{p}}, \overrightarrow{p}) & \tilde{p} \in \mathit{CoPattern} & ::= \alpha \mid O[\overrightarrow{p}, \overrightarrow{\tilde{p}}] \\
v \in \mathit{Term} & ::= x \mid \mu(\overrightarrow{\tilde{p}.c}) \mid K(\overrightarrow{e}, \overrightarrow{v}) & e \in \mathit{CoTerm} & ::= \alpha \mid \tilde{\mu}[\overrightarrow{p.c}] \mid O[\overrightarrow{v}, \overrightarrow{e}]
\end{array}$$

$$\begin{array}{rl}
\mathit{decl} \in \mathit{Declaration} ::= & \mathit{data}\ F(\overrightarrow{X})\ \mathit{where}\ \overrightarrow{K : \overrightarrow{A} \vdash F(\overrightarrow{X}) \mid \overrightarrow{B}} \\
& \mid \mathit{codata}\ G(\overrightarrow{X})\ \mathit{where}\ \overrightarrow{O : \overrightarrow{A} \mid F(\overrightarrow{X}) \vdash \overrightarrow{B}}
\end{array}$$

$$\begin{array}{rl}
c \in \mathit{Command} & ::= \langle v \parallel e \rangle \\
\Gamma \in \mathit{InputEnv} & ::= x_1 : A_1, \ldots, x_n : A_n \\
\Delta \in \mathit{OutputEnv} & ::= \alpha_1 : A_1, \ldots, \alpha_n : A_n \\
\mathit{Judgement} & ::= c : (\Gamma \vdash \Delta) \mid (\Gamma \vdash v : A \mid \Delta) \mid (\Gamma \mid e : A \vdash \Delta)
\end{array}$$

Core rules:

$$\frac{}{x : A \vdash x : A \mid}\ VR \qquad \frac{}{\mid \alpha : A \vdash \alpha : A}\ VL$$

$$\frac{\Gamma \vdash v : A \mid \Delta \qquad \Gamma' \mid e : A \vdash \Delta'}{\langle v \parallel e \rangle : (\Gamma', \Gamma \vdash \Delta', \Delta)}\ Cut$$

Logical rules:

Given $\mathit{data}\ F(\overrightarrow{X})\ \mathit{where}\ \overrightarrow{K_i : \overrightarrow{A_{ij}}^{j} \vdash F(\overrightarrow{X}) \mid \overrightarrow{B_{ij}}^{j}}^{i}$:

$$\frac{\overrightarrow{\Gamma \mid e : B_{ij}\overrightarrow{\{C/X\}} \vdash \Delta}^{j} \qquad \overrightarrow{\Gamma \vdash v : A_{ij}\overrightarrow{\{C/X\}} \mid \Delta}^{j}}{\Gamma \vdash K_i(\overrightarrow{e}, \overrightarrow{v}) : F(\overrightarrow{C}) \mid \Delta}\ FR_{K_i}$$

$$\frac{\overrightarrow{c_i : (\Gamma, \mathit{VarsTypes}(p_i, A) \vdash \mathit{CoVarsTypes}(p_i, A), \Delta)}^{i}}{\Gamma \mid \tilde{\mu}[\overrightarrow{p_i.c_i}^{i}] : A \vdash \Delta}\ FL$$

Given $\mathit{codata}\ G(\overrightarrow{X})\ \mathit{where}\ \overrightarrow{O_i : \overrightarrow{A_{ij}}^{j} \mid G(\overrightarrow{X}) \vdash \overrightarrow{B_{ij}}^{j}}^{i}$:

$$\frac{\overrightarrow{c_i : (\Gamma, \mathit{VarsTypes}(p_i, A) \vdash \mathit{CoVarsTypes}(p_i, A), \Delta)}^{i}}{\Gamma \vdash \mu(\overrightarrow{\tilde{p}_i.c_i}^{i}) : A \mid \Delta}\ GR$$

$$\frac{\overrightarrow{\Gamma \vdash v : A_{ij}\overrightarrow{\{C/X\}} \mid \Delta}^{j} \qquad \overrightarrow{\Gamma \mid e : B_{ij}\overrightarrow{\{C/X\}} \vdash \Delta}^{j}}{\Gamma \mid O_i[\overrightarrow{v}, \overrightarrow{e}] : G(\overrightarrow{C}) \vdash \Delta}\ GL_{O_i}$$

FIGURE 2.11: *The $\mu\tilde{\mu}$-calculus with user-defined (co-)datatypes.*

2.2.4.1 Evaluation

In addition to new typing rules, we also need new reduction rules to define evaluation for the new user-defined (co-)datatypes. We already have the μ and $\tilde{\mu}$ rewriting rules. Those only work on μ and $\tilde{\mu}$ (co-)terms where there is only a single (co-)pattern, and it consists of a one simple (co-)variable. Fortunately, we are able to rewrite more complicated (co-)patterns to a series of simple (co-)patterns. We could for example rewrite the command $\langle (v, v') \parallel \tilde{\mu}[(x, y).c]\rangle$ to $\langle v \parallel \tilde{\mu}x.\langle v' \parallel \tilde{\mu}y.c\rangle\rangle$. This way both v and v' get matched to x and y respectively, and every $\tilde{\mu}$ co-term consist of single variables as patterns. After rewriting the μ and $\tilde{\mu}$ rewriting rules can be used to further evaluate the command.

$$\langle v \parallel \tilde{\mu}x.\langle v' \parallel \tilde{\mu}y.c\rangle\rangle \succ_{\tilde{\mu}} \langle v' \parallel \tilde{\mu}y.c\rangle\{v/x\} \succ_{\tilde{\mu}} c\{v/x, v'/y\}$$

What actually happens during the rewriting of patterns, is that every sub-pattern and sub-co-pattern is taken from the input (co-)pattern and moved to a separate command. Of course, this only happens if the constructor or observer in the (co-)pattern matches the constructor or observer that is supplied to the pattern in the original command. We can formalise these rewriting rules, called β_F and β_G, as follows:

$$\langle K_i(\vec{e}, \vec{v}) \parallel \tilde{\mu}[\cdots \mid K_i(\vec{\tilde{p}}, \vec{p}).c_i \mid \ldots]\rangle \succ_{\beta_F} \langle \mu\vec{\tilde{p}}.\langle \vec{v} \parallel \tilde{\mu}\vec{p}.c_i\rangle \parallel \vec{e}\rangle$$
$$\langle \mu(\cdots \mid O_i[\vec{p}, \vec{\tilde{p}}].c_i \mid \ldots) \parallel O_i[\vec{v}, \vec{e}]\rangle \succ_{\beta_G} \langle \vec{v} \parallel \tilde{\mu}\vec{p}.\langle \mu\vec{\tilde{p}}.c_i \parallel \vec{e}\rangle\rangle$$

The β_F and β_G rules also work for (co-)patterns that have sub-(co-)patterns that do not consist of a single (co-)variable. The new rules can, for instance, be used to evaluate the command $\langle ((v_1, v_2), v_3) \parallel \tilde{\mu}[((x, y), z).c]\rangle$ as follows:

$$\begin{gathered}
\langle ((v_1, v_2), v_3) \parallel \tilde{\mu}[((x, y), z).c]\rangle \\
\succ_{\beta_F} \\
\langle (v_1, v_2) \parallel \tilde{\mu}[(x, y).\langle v_3 \parallel \tilde{\mu}z.c\rangle]\rangle \\
\succ_{\beta_F} \\
\langle v_1 \parallel \tilde{\mu}x.\langle v_2 \parallel \tilde{\mu}y.\langle v_3 \parallel \tilde{\mu}z.c\rangle\rangle\rangle \\
\succ_{\tilde{\mu}} \\
\langle v_2 \parallel \tilde{\mu}y.\langle v_3 \parallel \tilde{\mu}z.c\rangle\rangle\{v_1/x\} \\
\succ_{\tilde{\mu}} \\
\langle v_3 \parallel \tilde{\mu}z.c\rangle\{v_1/x, v_2/y\} \\
\succ_{\tilde{\mu}} \\
c\{v_1/x, v_2/y, v_3/z\}
\end{gathered}$$

2.2.4.2 Nested evaluation

Despite the new rules, there are still situations where evaluation gets stuck, even though it should not be. This occurs whenever a μ-term or $\tilde{\mu}$-co-term exists within a constructor or observer. For example, we are currently not able to further evaluate the command $\langle (x, \mu\beta.\langle y \parallel \beta\rangle) \parallel \alpha\rangle$, even though we know that $\mu\beta.\langle y \parallel \beta\rangle$ could be simplified to just y by the η_μ reduction rule. To allow those inner (co-)terms to be lifted, we first need to define *values* and *co-values*. A (co-)value is a (co-)term that cannot be evaluated any further. Whether a (co-)term is or is not a (co-)value, depends on the evaluation context. Some (co-)terms can be evaluated further under call-by-value($\mathcal{V}$), while they have already reached their final form under call-by-name($\mathcal{N}$) and vice versa. Under call-by-value, every co-term is a co-value, but only some terms are co-terms: variables, constructors where every sub-term is a value, and μ-terms for which at least one of the co-patterns contains an observer. Under call-by-name, this is almost exactly reversed: every term is a value, but only co-variables, observers where every sub-co-term is a co-value, and $\tilde{\mu}$-co-terms with at least one constructor in any of its patterns are co-values.

$$\begin{aligned}
V \in Value_{\mathcal{V}} &::= x \mid K(\vec{e}, \vec{V}) \mid \mu(\overrightarrow{O[\vec{p}, \vec{\tilde{p}}].c}) \\
E \in CoValue_{\mathcal{V}} &::= e \\
V \in Value_{\mathcal{N}} &::= v \\
E \in CoValue_{\mathcal{N}} &::= \alpha \mid O[\vec{v}, \vec{E}] \mid \tilde{\mu}[\overrightarrow{K(\vec{\tilde{p}}, \vec{p}).c}]
\end{aligned}$$

This means that under call-by-value, constructors that have sub-terms that are not values and μ-terms that have just a single co-pattern, which is a single co-variable, are non-values. Similarly, under call-by-name, observers that have sub-co-terms and $\tilde{\mu}$-co-terms that have a single pattern, consisting of a variable, are non-co-values. Now that we know which (co-)terms should be liftable, we can define lifting rules ς:

$$\begin{aligned}
K(\vec{E}, e', \vec{e}, \vec{v}) &\succ_{\varsigma} \mu\alpha.\langle \mu\beta.\langle K(\vec{E}, \beta, \vec{e}, \vec{v}) \parallel \alpha\rangle \parallel e'\rangle \\
K(\vec{E}, \vec{V}, v', \vec{v}) &\succ_{\varsigma} \mu\alpha.\langle v' \parallel \tilde{\mu}y.\langle K(\vec{e}, \vec{V}, y, \vec{v}) \parallel \alpha\rangle\rangle \\
O[\vec{V}, v', \vec{v}, \vec{e}] &\succ_{\varsigma} \tilde{\mu}x.\langle v' \parallel \tilde{\mu}y.\langle x \parallel O[\vec{V}, y, \vec{v}, \vec{e}]\rangle\rangle \\
O[\vec{V}, \vec{E}, e', \vec{e}] &\succ_{\varsigma} \tilde{\mu}x.\langle \mu\beta.\langle x \parallel O[\vec{v}, \vec{E}, \beta, \vec{e}]\rangle \parallel e'\rangle
\end{aligned}$$

where x, y, α and β are free (co-)variables, and v' and e' are non-values and non-co-values, respectively.

Now, we can use the new lifting rules to further evaluate our example, but only under call-by-value. Under call-by-name, $\langle (x, \mu\beta.\langle y \parallel \beta\rangle) \parallel \alpha\rangle$ cannot be evaluated anyfurther, because $\mu\beta.\langle y \parallel \beta\rangle$ is seen as a value in this strategy.

$$\begin{gathered}
\langle (x, \mu\beta.\langle y \parallel \beta\rangle) \parallel \alpha\rangle \\
\succ_{\varsigma} \\
\langle \mu\gamma.\langle \mu\beta.\langle y \parallel \beta\rangle \parallel \tilde{\mu}z.\langle (x, z) \parallel \gamma\rangle\rangle \parallel \alpha\rangle \\
\succ_{\mu} \\
\langle \mu\beta.\langle y \parallel \beta\rangle \parallel \tilde{\mu}z.\langle (x, z) \parallel \alpha\rangle\rangle \\
\succ_{\mu} \\
\langle y \parallel \tilde{\mu}z.\langle (x, z) \parallel \alpha\rangle\rangle \\
\succ_{\tilde{\mu}} \\
\langle (x, y) \parallel \alpha\rangle
\end{gathered}$$

The complete definition of all evaluation rules can be found in Figure 2.12.

What we have not seen implemented in the $\mu\tilde{\mu}$-calculus with user-defined (co-)datatypes are the previously built-in consumers to sum and product types. Before, we could consume them using the co-term $[e, e']$ for sum types, and π_1 and π_2 for product types. However, since datatypes cannot have observers, and co-datatypes cannot have constructors, these have disappeared from the new calculus. To re-integrate them, we need to define them as *functions*.

2.2.5 FUNCTIONS IN THE $\mu\tilde{\mu}$-CALCULUS

While functions are an integral built-in piece of the λ-calculus, they fulfill no special role in the $\mu\tilde{\mu}$-calculus, and are therefore not built into the calculus. However, that does not mean that it is not possible to define functions in the $\mu\tilde{\mu}$-calculus; we just need to declare the function co-datatype and its observer first.

$$\begin{aligned} \text{Evaluation strategies}: \;& \text{call-by-value } \mathcal{V} \\ \mid \;& \text{call-by-name } \mathcal{N} \end{aligned}$$

$$\begin{aligned}
V \in Value_{\mathcal{V}} &::= x \mid K(\vec{e}, \vec{V}) \mid \mu(\overrightarrow{O[\vec{p}, \vec{\tilde{p}}].c}) \\
E \in CoValue_{\mathcal{V}} &::= e \\
V \in Value_{\mathcal{N}} &::= v \\
E \in CoValue_{\mathcal{N}} &::= \alpha \mid O[\vec{v}, \vec{E}] \mid \tilde{\mu}[\overrightarrow{K(\vec{\tilde{p}}, \vec{p}).c}]
\end{aligned}$$

Evaluation rules

$$\begin{aligned}
\langle \mu\alpha.c \parallel e \rangle &\succ_{\mu} c\{e/\alpha\} \\
\langle v \parallel \tilde{\mu}x.c \rangle &\succ_{\tilde{\mu}} c\{v/x\} \\
\mu\alpha.\langle v \parallel \alpha \rangle &\succ_{\eta_\mu} v \\
\tilde{\mu}x.\langle x \parallel e \rangle &\succ_{\eta_{\tilde{\mu}}} e \\
\langle K_i(\vec{e}, \vec{v}) \parallel \tilde{\mu}[\cdots \mid K_i(\vec{\tilde{p}}, \vec{p}).c_i \mid \dots] \rangle &\succ_{\beta_F} \langle \mu\vec{\tilde{p}}.\langle \vec{v} \parallel \tilde{\mu}\vec{p}.c_i \rangle \parallel \vec{e} \rangle \\
\langle \mu(\cdots \mid O_i[\vec{p}, \vec{\tilde{p}}].c_i \mid \dots) \parallel O_i[\vec{v}, \vec{e}] \rangle &\succ_{\beta_G} \langle \vec{v} \parallel \tilde{\mu}\vec{p}.\langle \mu\vec{\tilde{p}}.c_i \parallel \vec{e} \rangle \rangle \\
K(\vec{E}, e', \vec{e}, \vec{v}) &\succ_{\varsigma} \mu\alpha.\langle \mu\beta.\langle K(\vec{E}, \beta, \vec{e}, \vec{v}) \parallel \alpha \rangle \parallel e' \rangle \\
K(\vec{E}, \vec{V}, v', \vec{v}) &\succ_{\varsigma} \mu\alpha.\langle v' \parallel \tilde{\mu}y.\langle K(\vec{e}, \vec{V}, y, \vec{v}) \parallel \alpha \rangle \rangle \\
O[\vec{V}, v', \vec{v}, \vec{e}] &\succ_{\varsigma} \tilde{\mu}x.\langle v' \parallel \tilde{\mu}y.\langle x \parallel O[\vec{V}, y, \vec{v}, \vec{e}] \rangle \rangle \\
O[\vec{V}, \vec{E}, e', \vec{e}] &\succ_{\varsigma} \tilde{\mu}x.\langle \mu\beta.\langle x \parallel O[\vec{v}, \vec{E}, \beta, \vec{e}] \rangle \parallel e' \rangle
\end{aligned}$$

FIGURE 2.12: *Evaluation rules and strategies in the $\mu\tilde{\mu}$-calculus with user-defined (co-)datatypes.*

A function is something that takes a term of one type, and produces a new term of a second type. Of course, in the $\mu\tilde{\mu}$-calculus, producing a new term and returning it is not as trivial as it is in the λ-calculus. Producing and returning a term requires it to be part of a command, which in turn requires a co-term of the same type. The observer for the function type will therefore take a term of the input type, and a co-term of the output type.

$$codata\ A \to B\ where\ \_\cdot\_\ : A \mid A \to B \vdash B$$

The observer $v \cdot e$ represents a *call-stack*.

Using this new definition, we can re-implement π_1 and π_2, the product type's consumers.

$$\pi_1 = \mu((x,y)\cdot\alpha.\langle x \parallel \alpha\rangle) : A \times B \to A$$
$$\pi_2 = \mu((x,y)\cdot\alpha.\langle y \parallel \alpha\rangle) : A \times B \to B$$

While π_1 and π_2 were co-terms before, they are terms now. As such, π_1 and π_2 are strictly no longer consumers, but rather producers of functions. This changes the way they are used. Before, we could extract v out of (v, v') using the command $\langle (v,v') \parallel \pi_1[e]\rangle$, which would evaluate to $\langle v \parallel e\rangle$. With the new version π_1, we build a call-stack containing (v, v') and the output co-term e instead: $\langle \pi_1 \parallel (v,v')\cdot e\rangle$, or the fully expanded version $\langle \mu((x,y)\cdot\alpha.\langle x \parallel \alpha\rangle) \parallel (v,v')\cdot e\rangle$. This command eventually evaluates to the same result $\langle v \parallel e\rangle$, according to the new evaluation rules:

$$\begin{gathered}
\langle \mu((x,y)\cdot\alpha.\langle x \parallel \alpha\rangle) \parallel (v,v')\cdot e\rangle \\
\succ_{\beta_G} \\
\langle (v,v') \parallel \tilde{\mu}((x,y).\langle \mu\alpha.\langle x \parallel \alpha\rangle \parallel e\rangle)\rangle \\
\succ_{\beta_F} \\
\langle v \parallel \tilde{\mu}x.\langle v' \parallel \tilde{\mu}y.\langle \mu\alpha.\langle x \parallel \alpha\rangle \parallel e\rangle\rangle\rangle \\
\succ_{\tilde{\mu}} \\
\langle v' \parallel \tilde{\mu}y.\langle \mu\alpha.\langle v \parallel \alpha\rangle \parallel e\rangle\rangle \\
\succ_{\tilde{\mu}} \\
\langle \mu\alpha.\langle v \parallel \alpha\rangle \parallel e\rangle \\
\succ_{\mu} \\
\langle v \parallel e\rangle
\end{gathered}$$

With our new definitions, π_1 and π_2 are the $\mu\tilde{\mu}$-calculus equivalents of the Haskell functions fst and snd.

The consumer for sum types $[e, e']$ is different. Since this co-term requires two separate sub-co-terms, and call-stacks only contain one sub-co-term, we cannot implement $[e, e']$ without first declaring a wrapper co-datatype $A ⅋ B$:

$$codata\ A ⅋ B\ where\ [\_,\_]\ :\mid A ⅋ B \vdash A, B$$

With this new co-datatype, we are able to define a function which takes a term of a sum-type, and consumes it, as seen below.

$$\begin{aligned}
&handleSum : A + B \to A ⅋ B \\
&handleSum = \mu(\iota_1(x)\cdot[\alpha_1,\alpha_2].\langle x \parallel \alpha_1\rangle \\
&\qquad\qquad\qquad \mid \quad \iota_2(y)\cdot[\alpha_1,\alpha_2].\langle y \parallel \alpha_2\rangle)
\end{aligned}$$

Like with product types and π_1 and π_2, $handleSum$ is a term, instead of a co-term. Once again, this changes the way $handleSum$ is used, compared to $[e, e']$ in the $\mu\tilde{\mu}$-calculus without user-defined (co-)datatypes. We are still able to evaluate the commands we write using this new system to the same command that we expect to get in the old system.

$$\begin{gathered}
\langle handleSum \parallel \iota_1(v) \cdot [e, e'] \rangle \\
\succ_{\beta_G} \\
\langle \iota_1(v) \parallel \tilde{\mu}(\iota_1(x).\langle \mu[[\alpha_1, \alpha_2].\langle x \parallel \alpha_1 \rangle] \parallel [e, e'] \rangle) \rangle \\
\succ_{\beta_F} \\
\langle v \parallel \tilde{\mu}x.\langle \mu[[\alpha_1, \alpha_2].\langle x \parallel \alpha_1 \rangle] \parallel [e, e'] \rangle \rangle \\
\succ_{\tilde{\mu}} \\
\langle \mu[[\alpha_1, \alpha_2].\langle v \parallel \alpha_1 \rangle] \parallel [e, e'] \rangle \\
\succ_{\beta_G} \\
\langle \mu\alpha_1.\langle \mu\alpha_2.\langle v \parallel \alpha_1 \rangle \parallel e' \rangle \parallel e \rangle \\
\succ_{\mu} \\
\langle \mu\alpha_2.\langle v \parallel e \rangle \parallel e' \rangle \\
\succ_{\mu} \\
\langle v \parallel e \rangle
\end{gathered}$$

$$\begin{gathered}
\langle handleSum \parallel \iota_2(v') \cdot [e, e'] \rangle \\
\succ_{\beta_G} \\
\langle \iota_2(v') \parallel \tilde{\mu}(\iota_2(y).\langle \mu[[\alpha_1, \alpha_2].\langle y \parallel \alpha_2 \rangle] \parallel [e, e'] \rangle) \rangle \\
\succ_{\beta_F} \\
\langle v' \parallel \tilde{\mu}y.\langle \mu[[\alpha_1, \alpha_2].\langle y \parallel \alpha_2 \rangle] \parallel [e, e'] \rangle \rangle \\
\succ_{\tilde{\mu}} \\
\langle \mu[[\alpha_1, \alpha_2].\langle v' \parallel \alpha_2 \rangle] \parallel [e, e'] \rangle \\
\succ_{\beta_G} \\
\langle \mu\alpha_1.\langle \mu\alpha_2.\langle v' \parallel \alpha_2 \rangle \parallel e' \rangle \parallel e \rangle \\
\succ_{\mu} \\
\langle \mu\alpha_2.\langle v' \parallel \alpha_2 \rangle \parallel e' \rangle \\
\succ_{\mu} \\
\langle v' \parallel e' \rangle
\end{gathered}$$

In addition to re-implementing lost consumers, functions and user-defined (co-)datatypes allow us to define much more powerful (co-)terms than we could before. We are now able to declare recursive (co-)datatypes, such as lists and streams.

$$\begin{array}{ll}
\textit{data List A where} & \textit{codata Stream A} \\
\quad Nil : \vdash List\ A\ | & \quad Head :|\ Stream\ A \vdash A \\
\quad Cons : A, List\ A \vdash List\ A\ | & \quad Tail :|\ Stream\ A \vdash Stream\ A
\end{array}$$

Next, we are able to define functions that alter the structure of these (co-)datatypes. For example, we are able to implement **map** and **foldr** for lists:

$$\begin{aligned}
\textbf{map} &: (A \to B) \to \mathit{List}\ A \to \mathit{List}\ B \\
\textbf{map} &= \mu\Big(f \cdot \alpha.\langle \mu(y \cdot \beta.\langle y \parallel \tilde{\mu}[\mathit{Cons}\ x\ xs.\langle f \parallel x \cdot \tilde{\mu}z.\langle \mathit{map} \parallel f \cdot xs \cdot \tilde{\mu}zs. \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \langle \mathit{Cons}\ z\ zs \parallel \beta\rangle\rangle\rangle \\
&\qquad\qquad\qquad\qquad\qquad \mid \mathit{Nil}.\langle \mathit{Nil} \parallel \beta\rangle]\rangle) \parallel \alpha\rangle\Big)
\end{aligned}$$

$$\begin{aligned}
\textbf{foldr} &: (A \to B \to B) \to B \to \mathit{List}\ A \to B \\
\textbf{foldr} &= \mu\Big(f \cdot b \cdot \mathit{Nil} \cdot \alpha.\langle \mathit{Nil} \parallel \alpha\rangle \\
&\quad \Big| \quad f \cdot b \cdot \mathit{Cons}\ x\ xs \cdot \alpha.\langle \mathit{foldr} \parallel f \cdot b \cdot xs \cdot \tilde{\mu}b'.\langle f \parallel x \cdot b' \cdot \alpha\rangle\rangle\Big)
\end{aligned}$$

These definitions display the power of the call-stack: We can see exactly which computations happen, and in what order.

3 Programming with the $\mu\tilde{\mu}$-calculus

In this chapter we propose a high-level programming language *MMH*(Mu-Mu-tilde-Haskell), which uses the $\mu\tilde{\mu}$-calculus as an intermediate language. This means that programs written in *MMH* are compiled to $\mu\tilde{\mu}$-calculus programs. In this form, the programs can be typechecked and evaluated using the $\mu\tilde{\mu}$-calculus' rules. The $\mu\tilde{\mu}$-calculus will fulfill the same role for *MMH* as the λ-calculus does for several languages, such as Haskell and ML.

A good starting point would be transpiling λ-calculus programs to $\mu\tilde{\mu}$-calculus programs. With such a transpiler, we are able to evaluate Haskell programs in the $\mu\tilde{\mu}$-calculus. Next, we can extend Haskell's syntax to allow the power of the $\mu\tilde{\mu}$-calculus - the separation of data and co-data - to be exploited.

3.1 Converting λ-terms to $\mu\tilde{\mu}$-terms

Before we can start translating λ-programs to $\mu\tilde{\mu}$-programs, we first need a renewed definition of the λ-calculus; one that supports user-defined datatypes and constructors. This definition is given in Figure 3.1.

We define an algorithm **Conv**, which converts λ-structures to $\mu\tilde{\mu}$-structures. There are many similarities between the λ-calculus and the $\mu\tilde{\mu}$-calculus. For example, the type system is nearly the same. In both calculi, types can consist of either a type variable, or a connector with a collection of subtypes. These types are therefore valid in both calculi, meaning that conversion is not necessary.

Additionally, the λ-calculus also has built-in support for function types, while the $\mu\tilde{\mu}$-calculus does not. For this reason, we require an implementation of function types in the $\mu\tilde{\mu}$-system. We will henceforth assume the definition that we have seen before:

$$codata\ A \to B\ where\ \_ \cdot \_\ : A \mid A \to B \vdash B$$

Using this definition, we can copy function types from the λ-calculus as well, meaning that all λ-calculus types can be safely copied to the $\mu\tilde{\mu}$-calculus, without any modifications.

Datatypes and constructors are similar between the two calculi as well. The only difference is that constructors in the $\mu\tilde{\mu}$-calculus may contain co-terms in addition to terms, while λ-calculus constructors solely contain terms. This effectively means that the collection of $\mu\tilde{\mu}$-constructors is a superset of the collection of λ-constructors; every λ-constructor can be a $\mu\tilde{\mu}$-constructor, but not every $\mu\tilde{\mu}$-constructor

$$
\begin{aligned}
A, B, C \in \mathit{Type} &::= X \mid F(\overrightarrow{A}) \mid A \to B \\
X, Y, Z \in \mathit{TypeVariable} &::= \ldots \\
F \in \mathit{Connector} &::= \ldots \qquad\qquad K \in \mathit{Constructor} ::= \ldots \\
x, y, z \in \mathit{Variable} &::= \ldots \qquad\qquad p \in \mathit{Pattern} ::= x \mid K(\overrightarrow{x}) \\
M, N \in \mathit{Term} &::= x \mid \lambda[\overrightarrow{p.M}] \mid M\ N \mid K(\overrightarrow{M}) \\
\mathit{decl} \in \mathit{Declaration} &::= \mathit{data}\ F(\overrightarrow{X})\ \mathit{where}\ \overrightarrow{K\ \overrightarrow{A} : F(\overrightarrow{X})} \\
H, J \in \mathit{Judgement} &::= M : A
\end{aligned}
$$

Typing rules:

$$
\dfrac{\dfrac{}{x : A}\,x \\ \vdots \\ M : B}{\lambda x.M : A \to B} \to I_x
\qquad\qquad
\dfrac{M : A \to B \qquad N : A}{M\ N : B} \to E
$$

Given $\mathit{data}\ F(\overrightarrow{X})\ \mathit{where}\ \overrightarrow{K_i\ \overrightarrow{A_{ij}}^{j} : F(\overrightarrow{X})}^{i}$:

$$
\dfrac{\overrightarrow{M_j : A_{ij}\{\overrightarrow{B/X}\}}^{j}}{K_i(\overrightarrow{M}) : F(\overrightarrow{B})}\,FI
\qquad\qquad
\dfrac{\overrightarrow{\overrightarrow{\dfrac{}{x_{ij} : A_{ij}\{\overrightarrow{C/X}\}}\,x_{ij}}^{j}}^{i} \\ \vdots \\ M_i : B}{\lambda[\overrightarrow{K_i(\overrightarrow{x_i}).M_i}^{i}] : F(\overrightarrow{C}) \to B} \to I_{K_i}
$$

Evaluation:

$$
\begin{aligned}
\lambda x.M\ N &\succ_{\beta_x} M\{N/x\} \\
\lambda[\cdots \mid K_i(\overrightarrow{x}).M_i \mid \ldots]\ K_i(\overrightarrow{N}) &\succ_{\beta_F} M_i\{\overrightarrow{N/x}\}
\end{aligned}
$$

FIGURE 3.1: *The λ-calculus with datatypes and constructors.*

can be a λ-constructor. The case for declarations in **Conv** is therefore a simple one:

$$\textbf{Conv}\Big(data\ F(\overrightarrow{X})\ where\ \overrightarrow{K_i\ \overrightarrow{A_{ij}}^{j}\ :\ F(\overrightarrow{X})}^{i}\Big)$$
$$=$$
$$data\ \textbf{Conv}(F(\overrightarrow{X}))\ where\ \overrightarrow{K_i : \overrightarrow{\textbf{Conv}(A_{ij})}^{j} \vdash \textbf{Conv}(F(\overrightarrow{X}))\ |}^{i}$$

With types and constructors done, we are able to define **Conv**'s behaviour for variables and patterns, before we can finally start converting terms.

$$\textbf{Conv}(x) = x \qquad \textbf{Conv}(K(\overrightarrow{x_i}^{i})) = K(\emptyset, \overrightarrow{\textbf{Conv}(x_i)}^{i})$$

In the λ-calculus, terms come in four different shapes: variables, λ-functions, function applications, and constructors. We have just shown how to convert variables and constructors, which leaves us with λ-functions and function applications. In the $\mu\tilde{\mu}$-calculus, functions need to have both an input term and an output co-term. For the simplest kind of functions, the ones only have variables as patterns, and no constructors, conversion is as simple as adding a fresh co-variable in the function's pattern and returning the function's term to this co-variable.

$$\textbf{Conv}(\lambda x.M) = \mu(\textbf{Conv}(x) \cdot \alpha.\langle \textbf{Conv}(M) \parallel \alpha\rangle)$$

λ-functions that have a constructor as at least one of its patterns require more work. In the $\mu\tilde{\mu}$-calculus, constructors and observers in (co-)patterns can only contain variables. This holds for the call-stack observer $(\cdot)$ as well. For this reason, we cannot convert a λ-function $\lambda[K(\overrightarrow{x}).M]$ to a μ-term $\mu(K(\overrightarrow{x})\cdot\alpha.\langle\textbf{Conv}(M) \parallel \alpha\rangle)$, as $K(\overrightarrow{x})\cdot\alpha$ is a nested pattern. This problem can be easily circumvented by splitting the pattern into two separate parts. We first generate a μ-term that expects a call-stack. This call-stack contains a fresh variable y and a fresh co-variable α. We then generate a $\tilde{\mu}$-co-term that contains the converted patterns and the converted terms of each branch in the λ-function. The term from the call-stack, y, is directly applied to the generated $\tilde{\mu}$-co-term. The command in the $\tilde{\mu}$-co-term connects the converted λ-term to the output of the call-stack, α.

$$\textbf{Conv}(\lambda[\overrightarrow{K_i(\overrightarrow{x_i}).M_i}^{i}]) = \mu\Big(y\cdot\alpha.\langle y \parallel \tilde{\mu}\Big[\overrightarrow{\textbf{Conv}(K_i(\overrightarrow{x_i})).\langle\textbf{Conv}(M_i) \parallel \alpha\rangle}^{i}\Big]\rangle\Big)$$

Finally, we define the behaviour of **Conv** for function applications. In the λ-calculus, computation happens in function applications. In the $\mu\tilde{\mu}$-calculus, this happens in commands. This complicates the transformation algorithm, since function applications are terms, but commands are not. The generated term will therefore be a μ-term which takes a single fresh co-variable as its pattern. Its command uses the left-hand side of the function application as the term, while it uses the right-hand side in a call-stack as the co-term.

$$\textbf{Conv}(M\ N) = \mu\alpha.\langle\textbf{Conv}(M) \parallel \textbf{Conv}(N)\cdot\alpha\rangle$$

This approach of converting function applications to the $\mu\tilde{\mu}$-calculus can be problematic if there exists a function application within the right-hand side of a function application. Let us for example define the datatype $List$, and the function $tail$, which takes a list and returns the same list, but without its first element:

$$data\ List\ A\ where\ Nil : List\ A \mid Cons\ A\ (List\ A) : List\ A$$
$$tail = \lambda[(Cons\ x\ xs).xs]$$

If we now apply $tail$ twice in a row to a list $Cons\ x\ (Cons\ y\ Nil)$, we expect to see Nil as the result. In the λ-calculus, this result is achieved as follows:

$$tail\ (tail\ (Cons\ x\ (Cons\ y\ Nil))) \succ_{\beta_F} tail\ (Cons\ y\ Nil) \succ_{\beta_F} Nil$$

$$\begin{aligned}
&data\ List\ A\ where\ Nil : (\vdash List\ A\ |) \mid Cons : (A, List\ A \vdash List\ A\ |)\\
&tail = \mu(y \cdot \alpha.\langle y \parallel \tilde{\mu}[Cons\ x\ xs.\langle xs \parallel \alpha\rangle]\rangle)\\
&call = \mathbf{Conv}(tail\ (tail\ (Cons\ x\ (Cons\ y\ Nil))))\\
&\quad = \mu\alpha.\langle tail \parallel (\mu\beta.\langle tail \parallel (Cons\ x\ (Cons\ y\ Nil)) \cdot \beta\rangle) \cdot \alpha\rangle
\end{aligned}$$

$$\begin{gathered}
\langle call \parallel \alpha\rangle\\
\succ_\mu\\
\langle tail \parallel (\mu\beta.\langle tail \parallel (Cons\ x\ (Cons\ y\ Nil)) \cdot \beta\rangle) \cdot \alpha\rangle\\
\succ_\mu\\
\langle \mu\beta.\langle tail \parallel (Cons\ x\ (Cons\ y\ Nil)) \cdot \beta\rangle \parallel \tilde{\mu}[Cons\ x\ xs.\langle xs \parallel \alpha\rangle]\rangle\\
\succ_\mu\\
\langle tail \parallel (Cons\ x\ (Cons\ y\ Nil)) \cdot \tilde{\mu}[Cons\ x\ xs.\langle xs \parallel \alpha\rangle]\rangle\\
\succ_\mu\\
\langle Cons\ x\ (Cons\ y\ Nil) \parallel \tilde{\mu}[Cons\ x\ xs.\langle xs \parallel \tilde{\mu}[Cons\ x\ xs.\langle xs \parallel \alpha\rangle]\rangle]\rangle\\
\succ_{\tilde{\mu}}\\
\langle Cons\ y\ Nil \parallel \tilde{\mu}[Cons\ x\ xs.\langle xs \parallel \alpha\rangle]\rangle\\
\succ_{\tilde{\mu}}\\
\langle Nil \parallel \alpha\rangle
\end{gathered}$$

FIGURE 3.2: *Evaluating $tail\ (tail\ (Cons\ x\ (Cons\ y\ Nil)))$ using a call-by-value strategy in the $\mu\tilde{\mu}$-calculus.*

In the $\mu\tilde{\mu}$-calculus, however, this is a much more complicated process. To see the evaluation steps, we first need to convert the definitions to the $\mu\tilde{\mu}$-calculus using **Conv**.

$$\begin{aligned}
&data\ List\ A\ where\ Nil : (\vdash List\ A\ |) \mid Cons : (A, List\ A \vdash List\ A\ |)\\
&tail = \mu(y \cdot \alpha.\langle y \parallel \tilde{\mu}[Cons\ x\ xs.\langle xs \parallel \alpha\rangle]\rangle)\\
&call = \mathbf{Conv}(tail\ (tail\ (Cons\ x\ (Cons\ y\ Nil))))\\
&\quad = \mu\alpha.\langle tail \parallel (\mu\beta.\langle tail \parallel (Cons\ x\ (Cons\ y\ Nil)) \cdot \beta\rangle) \cdot \alpha\rangle
\end{aligned}$$

Of course, we cannot directly evaluate $call$, since it is a term, and not a command. Instead, we will evaluate the command $\langle call \parallel \alpha\rangle$, for some arbitrary α. We expect the result to be $\langle Nil \parallel \alpha\rangle$. There are two evaluation strategies, meaning there are two different reduction paths from $\langle call \parallel \alpha\rangle$. Figure 3.2 shows the successful evaluation from $\langle call \parallel \alpha\rangle$ to $\langle Nil \parallel \alpha\rangle$ using the call-by-value strategy. Evaluating the command using the call-by-name strategy, however, never reaches its end goal $\langle Nil \parallel \alpha\rangle$. The first two reduction steps using call-by-name are the same as the first two steps when using call-by-value.

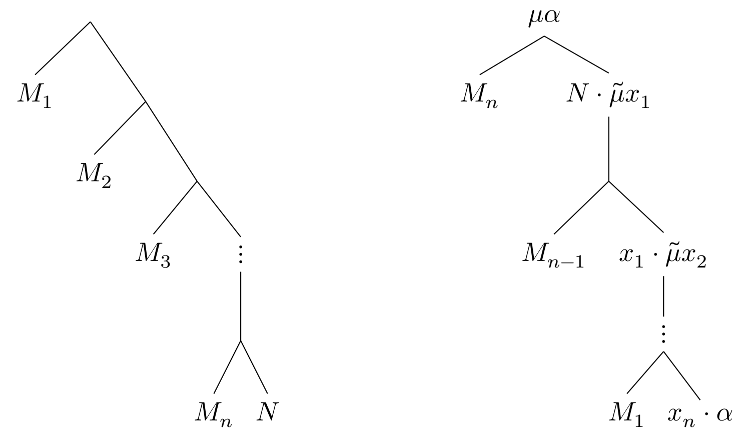
$$\begin{gathered}\langle call \parallel \alpha\rangle \\ \succ_\mu \\ \langle tail \parallel (\mu\beta.\langle tail \parallel (Cons\, x\ (Cons\, y\, Nil)) \cdot \beta\rangle) \cdot \alpha\rangle \\ \succ_\mu \\ \langle \mu\beta.\langle tail \parallel (Cons\, x\ (Cons\, y\, Nil)) \cdot \beta\rangle \parallel \tilde{\mu}[Cons\, x\, xs.\langle xs \parallel \alpha\rangle]\rangle\end{gathered}$$

At this point, evaluation gets stuck. We are using call-by-name, meaning that we prioritise $\tilde{\mu}$-reductions over μ-reductions. The $\tilde{\mu}$-co-term in question expects a $Cons$ constructor as its argument, but the term that is supplied to it is $\mu\beta.\langle tail \parallel (Cons\, x\ (Cons\, y\, Nil)) \cdot \beta\rangle$, a μ-term.

To get around this problem, we can alter the definition of **Conv** for function applications. Instead of generating terms that contain nested computations, we make sure function applications never contain another function application on the right-hand side. To do this, we need to reverse the order in which computations happen. The innermost computation needs to happen first, after which the result is passed to the second innermost computation, et cetera. To do this, we extract all the function applications in the application that we are trying to convert to the $\mu\tilde{\mu}$-calculus, and create a call-stack of all the left-hand sides in the extracted function applications. For a nested function application term $M_1\ (M_2\ (...\ (M_n\ N)))$, where N is not a function application, this is done as follows:

$$\begin{aligned}&\textbf{Conv}(M_1\ (M_2\ (...\ (M_n\ N)))) = \\ &\quad \mu\alpha.\langle \textbf{Conv}(M_n) \parallel \textbf{Conv}(N) \cdot \tilde{\mu}x_1.\langle \textbf{Conv}(M_{n-1}) \parallel x_1 \cdot ...\ \tilde{\mu}x_n.\langle M_1 \parallel x_n \cdot \alpha\rangle\rangle\rangle\end{aligned}$$

Graphically, we can see that a tree of function applications is essentially turned upside down into a tree of terms and co-terms, connected by commands and call-stacks.



We can use this new rule for our earlier example, $tail\ (tail\ (Cons\, x\ (Cons\, y\, Nil)))$:

$$\begin{aligned}&\textbf{Conv}(tail\ (tail\ (Cons\, x\ (Cons\, y\, Nil)))) \\ &\quad = \mu\alpha.\langle \textbf{Conv}(tail) \parallel \textbf{Conv}(Cons\, x\ (Cons\, y\, Nil)) \cdot \tilde{\mu}x.\langle \textbf{Conv}(tail) \parallel x \cdot \alpha\rangle\end{aligned}$$

Figure 3.3 shows the reduction steps for this new definition for both call-by-value and call-by-name evaluation strategies.

3.1.1 TYPECHECKING

We now have a complete definition for **Conv**. In this section we will prove that the conversion from λ-calculus terms to $\mu\tilde{\mu}$-calculus terms by **Conv** is type-safe. In essence, this means that for any λ-calculus

$$\begin{aligned}
tail &= \mu(y \cdot \alpha.\langle y \parallel \tilde{\mu}[Cons\,x\,xs.\langle xs \parallel \alpha\rangle]\rangle) \\
call &= \textbf{Conv}(tail\ (tail\ (Cons\,x\ (Cons\,y\,Nil)))) \\
&= \mu\alpha.\langle tail \parallel Cons\,x\ (Cons\,y\,Nil) \cdot \tilde{\mu}x.\langle tail \parallel x \cdot \alpha\rangle
\end{aligned}$$

$$\begin{gathered}
\langle call \parallel \alpha\rangle \\
\succ_{\mu} \\
\langle tail \parallel Cons\,x\ (Cons\,y\,Nil) \cdot \tilde{\mu}x.\langle tail \parallel x \cdot \alpha\rangle \\
\succ_{\mu} \\
\langle Cons\,x\ (Cons\,y\,Nil) \parallel \tilde{\mu}[Cons\,x\,xs.\langle xs \parallel \tilde{\mu}x.\langle tail \parallel x \cdot \alpha\rangle\rangle]\rangle \\
\succ_{\tilde{\mu}} \\
\langle Cons\,y\,Nil \parallel \tilde{\mu}x.\langle tail \parallel x \cdot \alpha\rangle\rangle \\
\succ_{\tilde{\mu}} \\
\langle tail \parallel (Cons\,y\,Nil) \cdot \alpha\rangle \\
\succ_{\mu} \\
\langle Cons\,y\,Nil \parallel \tilde{\mu}[Cons\,x\,xs.\langle xs \parallel \alpha\rangle]\rangle \\
\succ_{\tilde{\mu}} \\
\langle Nil \parallel \alpha\rangle
\end{gathered}$$

FIGURE 3.3: *Evaluating $tail\ (tail\ (Cons\,x\ (Cons\,y\,Nil)))$ using the improved version of **Conv** that generates call-stacks for nested function applications. These reduction steps are the same for both call-by-name and call-by-value evaluation.*

term M with type A, the result of **Conv**(M) must have type A too. Terms come in four different shapes in the λ-calculus: variables, λ-functions, function applications and constructors. Terms of all shapes will be proven to be type-safe.

First, we convert variables. Variables are a special case, because their types cannot be proven without any context. In both the λ-calculs and the $\mu\tilde{\mu}$-calculus, it is not possible to prove that a variable x has the type A without some form of context. In the λ-calculus, this context is provided via the $\to I$ rules, while in the $\mu\tilde{\mu}$-calculus, the context is the input environment Γ. Because of this, we can make no claims about the type of an arbitrary variable x in the λ-calculus. Likewise, we cannot make any claims about the type of the result of **Conv**(x) in the $\mu\tilde{\mu}$-calculus either. However, not knowing the type of x in either calculus means that all type information is retained during the conversion.

Secondly, we convert constructors. We will convert an arbitrary constructor $K_i(\overrightarrow{M})$ with datatype declaration $data\ F(\overrightarrow{X})\ where\ \overrightarrow{K\ \overrightarrow{A} : F(\overrightarrow{X})}$ and type $F(\overrightarrow{B})$. First, we will transform the datatype declaration to the format of the $\mu\tilde{\mu}$-calculus: $data\ F(\overrightarrow{X})\ where\ \overrightarrow{K_i : \overrightarrow{A_{ij}}^j \vdash F(\overrightarrow{X}) \mid}^i$. We can now convert $K_i(\overrightarrow{M})$ to the $\mu\tilde{\mu}$-calculus by using **Conv**. **Conv**$(K_i(\overrightarrow{M})) = K_i(\emptyset, \overrightarrow{\mathbf{Conv}(M_j)}^j)$. Finally, we need to prove that the constructor's type remains the same after conversion to the $\mu\tilde{\mu}$-calculus. The FI rule tells us that we can only reach the conclusion that $K_i(\overrightarrow{M})$ has type $F(\overrightarrow{B})$ when each term M_j in $\overrightarrow{M}$ has type $A_{ij}\overrightarrow{\{B/X\}}$. Similarly, according to the FR_{K_i} rule, each term **Conv**(M_j) must have type $A_{ij}\overrightarrow{\{B/X\}}$ before we can claim $K_i(\emptyset, \overrightarrow{\mathbf{Conv}(M_j)}^j)$ has type $F(\overrightarrow{B})$.

$$\frac{\overrightarrow{M_j : A_{ij}\overrightarrow{\{B/X\}}}^j}{K_i(\overrightarrow{M}) : F(\overrightarrow{B})}\,FI \qquad \frac{\overrightarrow{\vdash \mathbf{Conv}(M_j) : A_{ij}\overrightarrow{\{B/X\}} \mid}^j}{\vdash K_i(\emptyset, \overrightarrow{\mathbf{Conv}(M_j)}^j) : F(\overrightarrow{B}) \mid}\,FR_{K_i}$$

We now have reduced the problem to proving for each of M_j that their types do not change when transforming them to the $\mu\tilde{\mu}$-calculus using **Conv**. We have therefore inductively proven that the transformation of constructors from the λ-calculus to the $\mu\tilde{\mu}$-calculus is type-safe.

Next, we will do the same for λ-functions. The result of using **Conv** on a λ-function depends on the patterns in the function. For a single variable pattern, the result is simpler that for one or more patterns that contain a constructor. Let us first look at the former case: a lambda function $\lambda x.M$ with type $A \to B$. the $\to I_x$ rule tells us that to prove $\lambda x.M : A \to B$, we need to prove $M : B$, and are allowed to use a verification rule that confirms $x : A$. The $\mu\tilde{\mu}$-calculus version of $\lambda x.M : A$ is $\mu(x \cdot \alpha.\langle \mathbf{Conv}(M) \parallel \alpha\rangle)$, with the usual co-datatype declaration of functions. The GR rule can be used to reduce the judgement to $\langle \mathbf{Conv}(M) \parallel \alpha\rangle : (x : A \vdash \alpha : B)$. The Cut rule splits this command into its term and its co-term. Both need to have the same type, B. The VL rule is used to verify the co-term branch. What is left is the judgement $x : A \vdash \mathbf{Conv}(M) : B \mid$. This means we need to prove that the $\mu\tilde{\mu}$-calculus version of M has type B, and we know that x has type A. This premise is equivalent to the one we still needed to prove for the λ-calculus, meaning that λ-functions with a single variable as its pattern can be safely converted to the $\mu\tilde{\mu}$-calculus.

$$\begin{array}{c}\dfrac{}{x : A}\,x \\ \vdots \\ \dfrac{M : B}{\lambda x.M : A \to B}\to I_x\end{array} \qquad \dfrac{\dfrac{x : A \vdash \mathbf{Conv}(M) : B \mid \qquad \dfrac{}{\mid \alpha : B \vdash \alpha : B}\,VL}{\langle \mathbf{Conv}(M) \parallel \alpha\rangle : (x : A \vdash \alpha : B)}\,Cut}{\vdash \mu(x \cdot \alpha.\langle \mathbf{Conv}(M) \parallel \alpha\rangle) : A \to B \mid}\,GR$$

The strategy for proving the type-safety of the conversion of λ-functions one or more constructors in its patterns is similar. For a given declaration $data\ F(\overrightarrow{X})\ where\ \overrightarrow{K\ \overrightarrow{A} : F(\overrightarrow{X})}$ and a λ-function $\lambda[\overrightarrow{K_i(\overrightarrow{x_i}).M_i}^i]$ with type $F(\overrightarrow{C}) \to B$, we can use the $\to I_{K_i}$ rule. After applying this rule, we need to prove that every branch's output term M_i is of type B. For each of those branches, we get access to verification functions for all variables x_{ij} in the pattern of the branch.

$$\dfrac{\begin{array}{c}\overrightarrow{\overrightarrow{\overline{x_{ij} : A_{ij}\{C/X\}}^{\,x_{ij}}}^{j}}^{i}\\ \vdots \\ M_i : B\end{array}}{\lambda[\overrightarrow{K_i(\overrightarrow{x_i}).M_i}^{i}] : F(\overrightarrow{C}) \to B}\to I_{K_i}$$

Conv transforms this λ-function into $\mu(y \cdot \alpha.\langle y \parallel \tilde{\mu}[\overrightarrow{K_i(\overrightarrow{x}).\langle \mathbf{Conv}(M_i) \parallel \alpha\rangle}^{i}]\rangle)$. Just like for λ-functions with a single variable as its pattern, we can consecutively apply the GR and the Cut rule. The VR rule is used to verify the term side of the command. This leaves us with the co-term $\tilde{\mu}[\overrightarrow{K_i(\overrightarrow{x}).\langle \mathbf{Conv}(M_i) \parallel \alpha\rangle}^{i}]$, which should have the type $F(\overrightarrow{C})$. To further reduce this judgement, we apply the FL rule. As a result of this, we need to prove that for each branch i in the $\tilde{\mu}$-co-term, $\langle \mathbf{Conv}(M_i) \parallel \alpha\rangle$ is type-correct. As we already know that α has type B, type-correct in this instance means that $\mathbf{Conv}(M_i)$ must have type B.

$$\dfrac{\dfrac{}{y : F(\overrightarrow{C}) \vdash y : F(\overrightarrow{C}) \mid}VR \qquad \dfrac{\overrightarrow{\dfrac{\overrightarrow{x_{ij} : A_{ij}\{C/x\}}^{j} \vdash \mathbf{Conv}(M_i) : B \mid \qquad \dfrac{}{\mid \alpha : B \vdash \alpha : B}VL}{\langle \mathbf{Conv}(M_i) \parallel \alpha\rangle : (\overrightarrow{x_{ij} : A_{ij}\{C/x\}}^{j} \vdash \alpha : B)}Cut}^{i}}{\mid \tilde{\mu}[\overrightarrow{K_i(\overrightarrow{x}).\langle \mathbf{Conv}(M_i) \parallel \alpha\rangle}^{i}] : F(\overrightarrow{C}) \vdash \alpha : B}FL}{\dfrac{\langle y \parallel \tilde{\mu}[\overrightarrow{K_i(\overrightarrow{x}).\langle \mathbf{Conv}(M_i) \parallel \alpha\rangle}^{i}]\rangle : (y : F(\overrightarrow{C}) \vdash \alpha : B)}{\vdash \mu(y \cdot \alpha.\langle y \parallel \tilde{\mu}[\overrightarrow{K_i(\overrightarrow{x}).\langle \mathbf{Conv}(M_i) \parallel \alpha\rangle}^{i}]\rangle) : F(\overrightarrow{C}) \to B \mid}GR}Cut$$

Once again, the remaining premises for the $\mu\tilde{\mu}$-calculus are equivalent to those remaining in the proof tree for the λ-calculus.

Finally, we will convert function applications to the $\mu\tilde{\mu}$-calculus and prove type-safety. Recall how nested function applications alter the results of **Conv**. For this reason, we will use the term $M_n\ (M_{n-1}\ (...\ (M_1\ N)))$ for this proof. Assuming we want to prove this term has type A_n, we can repeatedly use the $\to E$ rule. This rule tells us that for our term, M_n must be of type $A_{n-1} \to A_n$, while $(M_{n-1}\ (...\ (M_1\ N)))$ must have type A_{n-1}. Applying this rule once more tells us that M_{n-1} must be of type $A_{n-2} \to A_{n-1}$, and $(...\ (M_1\ N))$ of type A_{n-2}. Repeating this process until we can no longer continue will eventually tell us that every M_i must have type $A_{i-1} \to A_i$. The innermost input term, N, is supplied to M_1, which must have type $A_0 \to A_1$. N must therefore be of type A_0.

$$\dfrac{M_n : A_{n-1} \to A_n \qquad \dfrac{M_{n-1} : A_{n-2} \to A_{n-1} \qquad \begin{array}{c}\dfrac{M_1 : A_0 \to A_1 \qquad N : A_0}{M_1\ N : A_1}\to E\\ \vdots\end{array}}{M_{n-1}\ (...\ (M_1\ N)) : A_{n-1}}\to E}{M_n\ (M_{n-1}\ (...\ (M_1\ N))) : A_n}\to E$$

Transforming $M_n\ (M_{n-1}\ (...\ (M_1\ N)))$ to the $\mu\tilde{\mu}$-calculus using **Conv** yields $\mu\alpha.\langle \mathbf{Conv}(M_1) \parallel \mathbf{Conv}(N) \cdot \tilde{\mu}x_1.\langle \mathbf{Conv}(M_2) \parallel x_1 \cdot \tilde{\mu}x_2.\langle ...\ \tilde{\mu}x_{n-1}.\langle \mathbf{Conv}(M_n) \parallel x_{n-1} \cdot \alpha\rangle\rangle\rangle\rangle$. For the sake of simplicity and readability, we will refer to this term as $\mu\alpha.\langle M_1 \parallel N \cdot \tilde{\mu}x_1.\langle M_2 \parallel x_1 \cdot \tilde{\mu}x_2.\langle ...\ \tilde{\mu}x_{n-1}.\langle M_n \parallel x_{n-1} \cdot \alpha\rangle\rangle\rangle\rangle$, meaning we remove the mention of **Conv**. Typechecking this can be done by first using the the AR and cut rules. This tells us that M_1 needs to be of type $A_0 \to A_1$, and that the co-term supplied to M_1, $N \cdot \tilde{\mu}x_1.\langle M_2 \parallel x_1 \cdot \tilde{\mu}x_2.\langle ...\ \tilde{\mu}x_{n-1}.\langle M_n \parallel x_{n-1} \cdot \alpha\rangle\rangle\rangle$, must be of the same type.

$$\dfrac{\dfrac{\vdash M_1 : A_0 \to A_1 \mid \qquad \begin{array}{c}\mid N \cdot \tilde{\mu}x_1.\langle M_2 \parallel x_1 \cdot \tilde{\mu}x_2.\langle ...\ \tilde{\mu}x_{n-1}.\langle M_n \parallel x_{n-1} \cdot \alpha\rangle\rangle\rangle\\ : A_0 \to A_1 \vdash \alpha : A_n\end{array}}{\langle M_1 \parallel N \cdot \tilde{\mu}x_1.\langle M_2 \parallel x_1 \cdot \tilde{\mu}x_2.\langle ...\ \tilde{\mu}x_{n-1}.\langle M_n \parallel x_{n-1} \cdot \alpha\rangle\rangle\rangle\rangle : (\vdash \alpha : A_n)}Cut}{\vdash \mu\alpha.\langle M_1 \parallel N \cdot \tilde{\mu}x_1.\langle M_2 \parallel x_1 \cdot \tilde{\mu}x_2.\langle ...\ \tilde{\mu}x_{n-1}.\langle M_n \parallel x_{n-1} \cdot \alpha\rangle\rangle\rangle\rangle : A_n \mid}AR$$

From this point on, we can start recursively applying the GL_{O_i} rule, then applying the AL rule on the co-term we get out of the GL_{O_i} rule, and finally applying the Cut rule. This leaves us with three new premises to prove. Knowing that the last term we saw was M_i, M_1 in this case, we now need to prove that the input to this function has type A_{i-1}. We also add the next function term, M_{i+1}, which has type $A_{i+1} \to A_{i+2}$. Finally, we also need to prove the co-term supplied to M_{i+1} has the same type.

$$\dfrac{\vdash N : A_0 \mid \qquad \dfrac{\dfrac{\vdash M_2 : A_1 \to A_2 \mid \qquad \begin{array}{c} x_1 : A_1 \mid x_1 \cdot \tilde{\mu}x_2.\langle ... \tilde{\mu}x_{n-1}.\langle M_n \\ \parallel x_{n-1} \cdot \alpha\rangle\rangle : A_1 \to A_2 \vdash \alpha : A_n \end{array}}{\langle M_2 \parallel x_1 \cdot \tilde{\mu}x_2.\langle ... \tilde{\mu}x_{n-1}.\langle M_n \parallel x_{n-1} \cdot \alpha\rangle\rangle : (x_1 : A_1 \vdash \alpha : A_n)}\,Cut}{\mid \tilde{\mu}x_1.\langle M_2 \parallel x_1 \cdot \tilde{\mu}x_2.\langle ... \tilde{\mu}x_{n-1}.\langle M_n \parallel x_{n-1} \cdot \alpha\rangle\rangle : A_1 \vdash \alpha : A_n}\,AL}{\mid N \cdot \tilde{\mu}x_1.\langle M_2 \parallel x_1 \cdot \tilde{\mu}x_2.\langle ... \tilde{\mu}x_{n-1}.\langle M_n \parallel x_{n-1} \cdot \alpha\rangle\rangle : A_0 \to A_1 \vdash \alpha : A_n}\,GL_{O_i}$$

We can apply this same strategy once again, and keep doing this until it is no longer possible to do so. Each step adds a new premise that needs to be proven: M_{i+1}.

$$\dfrac{\dfrac{}{x_1 : A_1 \vdash x_1 : A_1}\,VR \qquad \dfrac{\dfrac{\vdash M_3 : A_2 \to A_3 \mid \qquad \begin{array}{c} x_2 : A_2 \mid x_2 \cdot \tilde{\mu}x_3.\langle ... \rangle \\ : A_2 \to A_3 \vdash \alpha : A_n \end{array}}{\langle ... \tilde{\mu}x_{n-1}.\langle M_n \parallel x_{n-1} \cdot \alpha\rangle\rangle : (x_2 : A_2 \vdash \alpha : A_n)}\,Cut}{\mid \tilde{\mu}x_2.\langle ... \tilde{\mu}x_{n-1}.\langle M_n \parallel x_{n-1} \cdot \alpha\rangle\rangle : A_2 \vdash \alpha : A_n}\,AL}{x_1 : A_1 \mid x_1 \cdot \tilde{\mu}x_2.\langle ... \tilde{\mu}x_{n-1}.\langle M_n \parallel x_{n-1} \cdot \alpha\rangle\rangle : A_1 \to A_2 \vdash \alpha : A_n}\,GL_{O_i}$$

At some point, it is no longer possible to apply these steps, because the judgement that needs to be proven will be $x_{n-1} : A_{n-1} \mid x_{n-1} \cdot \alpha : A_{n-1} \to A_n \vdash \alpha : A_n$. This final step can be completed using the GL_{O_i} rule, and using the two verification rules on the results.

$$\dfrac{\dfrac{}{x_{n-1} : A_{n-1} \vdash x_{n-1} : A_{n-1} \mid}\,VR \qquad \dfrac{}{\mid \alpha : A_n \vdash \alpha : A_n}\,VL}{x_{n-1} : A_{n-1} \mid x_{n-1} \cdot \alpha : A_{n-1} \to A_n \vdash \alpha : A_n}\,GL_{O_i}$$

After this entire process, there are a number of judgements that have not been proven. These are the judgements deciding the type of the innermost input term N, and all the functions $\overrightarrow{M}$. These coincide exactly with the premises left open-ended in the λ-calculus. We can therefore safely conclude that function applications can be transformed to the $\mu\tilde{\mu}$-calculus in a type-safe manner.

3.1.2 Evaluation

In addition to typechecking, it is important that λ-calculus terms do not change their meaning when they are converted to the $\mu\tilde{\mu}$-calculus. The term that they evaluate to must be the same in both calculi. However, as terms by themselves can often not be fully evaluated in the $\mu\tilde{\mu}$-calculus, we want the evaluation of a λ-calculus term M to be the same in a $\mu\tilde{\mu}$-calculus command with a fresh co-variable α: $\langle \textbf{Conv}(M) \parallel \alpha\rangle$. Formally, we want to prove that if M evaluates to N, then $\langle \textbf{Conv}(M) \parallel \alpha\rangle$ evaluates to $\langle \textbf{Conv}(N) \parallel \alpha\rangle$ under both evaluation strategies.

The evaluation of variables, constructors and λ-functions are trivial in this case. Neither can be evaluated further in the λ-calculus, which means M is equivalent to N. By proxy, $\textbf{Conv}(M)$ is equivalent to $\textbf{Conv}(N)$, and therefore $\langle \textbf{Conv}(M) \parallel \alpha\rangle$ is already evaluated to $\langle \textbf{Conv}(N) \parallel \alpha\rangle$.

Function applications are the only terms that can be β-reducable in the λ-calculus. These applications come in two different shapes, according to the two β-reduction rules:

$$\lambda x.M\ N \succ_{\beta_x} M\{N/x\}$$
$$\lambda[\cdots \mid K_i(\vec{x}).M_i \mid ...]\ K_i(\overrightarrow{N}) \succ_{\beta_F} M_i\{\overrightarrow{N/x}\}$$

First, we will check the former. The result of $\textbf{Conv}(\lambda x.M\ N)$ is $\mu\alpha.\langle \mu(x \cdot \beta.\langle \textbf{Conv}(M) \parallel \beta\rangle) \parallel \textbf{Conv}(N) \cdot \alpha\rangle$. In a command, we can evaluate this term as follows:

$$\begin{gathered}
\langle \mu\alpha.\langle \mu(x \cdot \beta.\langle \textbf{Conv}(M) \parallel \beta\rangle) \parallel \textbf{Conv}(N) \cdot \alpha\rangle \parallel \alpha\rangle \\
\succ_{\mu} \\
\langle \mu(x \cdot \beta.\langle \textbf{Conv}(M) \parallel \beta\rangle) \parallel \textbf{Conv}(N) \cdot \alpha\rangle \\
\succ_{\beta_G} \\
\langle \textbf{Conv}(N) \parallel \tilde{\mu}x.\langle \mu\beta.\langle \textbf{Conv}(M) \parallel \beta\rangle \parallel \alpha\rangle\rangle \\
\succ_{\tilde{\mu}} \\
\langle \mu\beta.\langle \textbf{Conv}(M) \parallel \beta\rangle \parallel \alpha\rangle\{\textbf{Conv}(N)/x\} \\
\succ_{\mu} \\
\langle \textbf{Conv}(M) \parallel \alpha\rangle\{\textbf{Conv}(N)/x\}
\end{gathered}$$

This final result corresponds to the evalation in the λ-calculus.

Next, we do the same for type application on a λ-function with a constructor as its argument. We use **Conv** to convert $\lambda[\cdots \mid K_i(\vec{x}).M_i \mid \ldots]\ K_i(\vec{N})$ to $\mu\alpha.\langle \mu y \cdot \beta.\langle y \parallel \tilde{\mu}[\cdots \mid K_i(\vec{x}).\langle \textbf{Conv}(M_i) \parallel \beta\rangle \mid \ldots]\rangle \parallel K_i(\overrightarrow{\textbf{Conv}(N_j)}^j) \cdot \alpha\rangle$. This term can be evaluated in a command as follows:

$$\begin{gathered}
\langle \mu\alpha.\langle \mu y \cdot \beta.\langle y \parallel \tilde{\mu}[\cdots \mid K_i(\vec{x}).\langle \textbf{Conv}(M_i) \parallel \beta\rangle \mid \ldots]\rangle \parallel K_i(\overrightarrow{\textbf{Conv}(N_j)}^j) \cdot \alpha\rangle \parallel \alpha\rangle \\
\succ_{\mu} \\
\langle \mu y \cdot \beta.\langle y \parallel \tilde{\mu}[\cdots \mid K_i(\vec{x}).\langle \textbf{Conv}(M_i) \parallel \beta\rangle \mid \ldots]\rangle \parallel K_i(\overrightarrow{\textbf{Conv}(N_j)}^j) \cdot \alpha\rangle \\
\succ_{\beta_G} \\
\langle K_i(\overrightarrow{\textbf{Conv}(N_j)}^j) \parallel \tilde{\mu}y.\langle \mu\beta.\langle y \parallel \tilde{\mu}[\cdots \mid K_i(\vec{x}).\langle \textbf{Conv}(M_i) \parallel \beta\rangle \mid \ldots]\rangle \parallel \alpha\rangle\rangle \\
\succ_{\tilde{\mu}} \\
\langle \mu\beta.\langle K_i(\overrightarrow{\textbf{Conv}(N_j)}^j) \parallel \tilde{\mu}[\cdots \mid K_i(\vec{x}).\langle \textbf{Conv}(M_i) \parallel \beta\rangle \mid \ldots]\rangle \parallel \alpha\rangle \\
\succ_{\mu} \\
\langle K_i(\overrightarrow{\textbf{Conv}(N_j)}^j) \parallel \tilde{\mu}[\cdots \mid K_i(\vec{x}).\langle \textbf{Conv}(M_i) \parallel \alpha\rangle \mid \ldots]\rangle \\
\succ_{\beta_F} \\
\langle \overrightarrow{\textbf{Conv}(N_j)}^j \parallel \tilde{\mu}\vec{x}.\langle \textbf{Conv}(M_i) \parallel \alpha\rangle\rangle \\
\overrightarrow{\succ_{\tilde{\mu}}} \\
\langle \textbf{Conv}(M_i) \parallel \alpha\rangle\{\overrightarrow{\textbf{Conv}(N)/x}\}
\end{gathered}$$

Like the earlier case, function application on λ-functions that take constructors as arguments evaluate to the same term in both calculi. We can therefore conclude that the generated $\mu\tilde{\mu}$-calculus terms from **Conv** can be safely evaluated, as they reduce to the same term as they would in the λ-calculus.

3.2 Nested (co-)patterns

So far, the $\mu\tilde{\mu}$-calculus has only supported *simple (co-)patterns*. A simple (co-)pattern is a (co-)pattern which exists of either a (co-)variable, or a constructor or observer to which all arguments are (co-)variables. While μ-terms and $\tilde{\mu}$-terms solely using these simple (co-)patterns are relatively easy to evaluate, they can also become verbose. To avoid this, we add support nested (co-)patterns. This means that constructors and observers in (co-)patterns can now contain additional constructors and observers, instead of just (co-)variables.

$$p \in Pattern ::= \ x \mid K(\vec{\tilde{p}}, \vec{p}) \qquad \tilde{p} \in CoPattern ::= \ \alpha \mid O[\vec{p}, \vec{\tilde{p}}]$$

These nested (co-)patterns allow more readable (co-)terms to be written. For example, let us compare our previous definition of the function map to a new definition that nested (co-)patterns allow us to write.

$$\begin{aligned}
&\mathbf{map} \ : (A \to B) \to List\ A \to List\ B \\
&\mathbf{map} = \mu\Big(f \cdot \alpha.\langle \mu(y \cdot \beta.\langle y \parallel \tilde{\mu}[Cons\ x\ xs.\langle f \parallel x \cdot \tilde{\mu}z.\langle map \parallel f \cdot xs \cdot \tilde{\mu}zs. \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \langle Cons\ z\ zs \parallel \beta\rangle\rangle\rangle \\
&\qquad\qquad\qquad\qquad\qquad \mid Nil.\langle Nil \parallel \beta\rangle]\rangle) \parallel \alpha\rangle\Big) \\
&\mathbf{map} = \mu\Big(f \cdot Cons\ x\ xs \cdot \alpha.\langle f \parallel x \cdot \tilde{\mu}y.\langle map \parallel f \cdot xs \cdot \tilde{\mu}z.\langle Cons\ y\ z \parallel \alpha\rangle\rangle\rangle \\
&\qquad \Big| \quad f \cdot Nil \cdot \alpha.\langle Nil \parallel \alpha\rangle\Big)
\end{aligned}$$

We are now able to write $f \cdot Cons\ x\ xs \cdot \alpha$ as a co-pattern, whereas before, it was necessary to split this co-pattern into pieces: $f \cdot \alpha$, $y \cdot \beta$ and $Cons\ x\ xs$. Reducing this to a single co-pattern also reduces the number of commands in the term, resulting in a much more concise and readable definition.

Similarly, nested patterns can be added to the λ-calculus lc. Once both the λ-calculus and the $\mu\tilde{\mu}$-calculus support nested patterns, we can simplify one of the cases in **Conv**. Previously, λ-functions that have a constructor in their patterns were converted to the $\mu\tilde{\mu}$-calculus using the following rule:

$$\mathbf{Conv}(\lambda[\overrightarrow{K_i(\vec{x_i}).M_i}^{i}]) = \mu\Big(y \cdot \alpha.\langle y \parallel \tilde{\mu}\Big[\overrightarrow{\mathbf{Conv}(K_i(\vec{x_i})).\langle \mathbf{Conv}(M_i) \parallel \alpha\rangle}^{i}\Big]\rangle\Big)$$

Now that nested patterns are supported, this can be simplified by removing the inner $\tilde{\mu}$-co-term, and instead placing the converted patterns inside a call-stack pattern:

$$\mathbf{Conv}(\lambda[\overrightarrow{K_i(\vec{p}).M_i}^{i}]) = \mu\Big(\overrightarrow{\mathbf{Conv}(K_i(\vec{p})) \cdot \alpha.\langle \mathbf{Conv}(M_i) \parallel \alpha\rangle}^{i}\Big)$$

We could use the existing β-reduction rules in the $\mu\tilde{\mu}$-calculus to evaluate nested (co-)patterns, but doing so can lead to problematic situations. As an example, we will apply an arbitrary function g to each element in an empty list Nil, using the definition of map with nested co-patterns. We will use a fresh variable y as the final co-term in the command.

$$\langle \mu(f \cdot Cons\ x\ xs \cdot \alpha.\langle \dots \rangle \mid f \cdot Nil \cdot \alpha.\langle \dots \rangle) \parallel g \cdot Nil \cdot \gamma\rangle$$

We can rewrite this command using the β_G rule. However, this rule does not match patterns recursively, but simply rewrites a command whenever a μ-term on the left side contains a co-pattern with the same observer in it as the co-term on the right side of the command.

$$\langle \mu(\cdots \mid O_i[\vec{p}, \vec{\tilde{p}}].c_i \mid \dots) \parallel O_i[\vec{v}, \vec{e}]\rangle \succ_{\beta_G} \langle \vec{v} \parallel \tilde{\mu}\vec{p}.\langle \mu\vec{\tilde{p}}.c_i \parallel \vec{e}\rangle\rangle$$

This causes the previous command to be rewritten to the following command:

$$\langle g \parallel \tilde{\mu}f.\langle \mu(Cons\ x\ xs \cdot \alpha.\langle \dots \rangle) \parallel Nil \cdot \gamma\rangle\rangle$$

This happens because both cases in the original μ-term contain a call-stack as its co-pattern. The β_G rule therefore matches the co-term $g \cdot Nil \cdot \gamma$ on the first case that it matches on. If we continue evaluating the command, we get the following steps:

$$\begin{gathered}\langle g \parallel \tilde{\mu} f.\langle \mu(Cons\,x\,xs \cdot \alpha.\langle \dots \rangle) \parallel Nil \cdot \gamma \rangle\rangle \\ \succ_{\tilde{\mu}} \\ \langle \mu(Cons\,x\,xs \cdot \alpha.\langle \dots \rangle) \parallel Nil \cdot \gamma \rangle \\ \succ_{\beta_G} \\ \langle Nil \parallel \tilde{\mu}[Cons\,x\,xs.\langle \mu\alpha.\langle \dots \rangle \parallel \gamma \rangle] \rangle\end{gathered}$$

From this point, evaluation cannot continue, because we have Nil as a term, while the $\tilde{\mu}$-co-term on the right side of the command only accepts $Cons\ x\ xs$.

This is not the behaviour that is expected. To combat this, we could alter the β-reduction rules to require nested (co-)patterns to recursively match any incoming (co-)terms. This would work fine for our earlier example. $\langle map \parallel g \cdot Nil \cdot \gamma \rangle$ would simply evaluate to $\langle Nil \parallel \gamma \rangle$ in one single reduction step.

Unfortunately, this approach can lead to problems as well. If a sub-(co-)term is not fully evaluated yet, this can cause mistakes in the (co-)pattern matching. As an example, we will write a function not, which takes a boolean parameter and returns its inverse. To do so, we first need to declare the boolean datatype.

$$data\ Bool\ where\ True : (\vdash Bool \mid) \mid False : (\vdash Bool \mid)$$

With this definition, we can define not.

$$\begin{aligned} not\ &: Bool \to Bool \\ not = &\ \mu(True \cdot \alpha.\langle False \parallel \alpha \rangle \mid False \cdot \alpha.\langle True \parallel \alpha \rangle) \end{aligned}$$

Now, applying an unevaluated term that should evaluate to either $True$ or $False$ to this function can lock the evaluation context. As an example, let us apply the term $\mu\beta.\langle True \mid \beta \rangle$ to this function.

$$\langle \mu(True \cdot \alpha.\langle False \parallel \alpha \rangle \mid False \cdot \alpha.\langle True \parallel \alpha \rangle) \parallel \mu\beta.\langle True \mid \beta \rangle \cdot \gamma \rangle$$

We cannot evaluate this, since $\mu\beta.\langle True \mid \beta \rangle$ cannot be matched by either $True$ or $False$.

3.2.1 Expanding (co-)patterns

Instead of recursively (co-)pattern matching during evaluation, we allow (co-)terms to be written with nested (co-)patterns, but rewrite all of the (co-)terms to ones that only contain simple patterns. This process is called *pattern expansion*. Pattern expansion is a technique of simplifying terms often used in languages based on the λ-calculus that can be implemented for the $\mu\tilde{\mu}$-calculus too.

The standard algorithm to compute such transformations has been Augustsson's algorithm **C** [Aug85] for many years. Augustsson originally developed this algorithm for his own *Lazy ML* compiler [Aug84]. ML is based on the λ-calculus, of course, but with some slight alterations, algorithm **C** can be used for the $\mu\tilde{\mu}$-calculus as well. From this point on, Augustsson's algorithm will be referred to as **C**, while the transformation algorithm for the $\mu\tilde{\mu}$-calculus will be referred to as $\mathbf{C}_\mu$.

In the λ-calculus, every object is a term, including computations. **C** therefore produces terms. In the $\mu\tilde{\mu}$-calculus, however, we can also write co-terms and commands. Since terms and co-terms cannot interact with each other by themselves, algorithm $\mathbf{C}_\mu$ produces commands.

Both **C** and $\mathbf{C}_\mu$ are functions that take three input parameters. For **C**, these three arguments are a list of terms, a list of tuples of a list of patterns and a term, and a default term. For $\mathbf{C}_\mu$, the three arguments are a list of *expressions*, which can be both terms and co-terms, a list of tuples of a list of (co-)patterns and a command, and a default command. Both functions are expressed below.

$$\begin{array}{l} M \in Term \\ p \in Pattern \\ \mathbf{C}(\overrightarrow{M}, \overrightarrow{(\overrightarrow{p}, M)}, M) : M \end{array} \qquad \begin{array}{ll} v \in Term & e \in CoTerm \\ p \in Pattern & \tilde{p} \in CoPattern \\ exp ::= v \mid e & q ::= p \mid \tilde{p} \\ c \in Command & \\ \mathbf{C}_\mu(\overrightarrow{exp}, \overrightarrow{(\overrightarrow{q}, c)}, c) : c & \end{array}$$

Several constraints must be applied to the input arguments for the algorithm to function correctly. First, each pattern sequence needs to be of exactly the same length as the list of expressions. Next, the expressions and (co-)patterns should match their kinds, meaning that a term should never be matched to a co-pattern, and a co-term should never be matched to a pattern.

Theorem 3.1. *Given $\mathbf{C}_\mu(\overrightarrow{exp}, \overrightarrow{(\overrightarrow{q}_i, c_i)}^i, d)$, for each i, $\overrightarrow{exp}$ and $\overrightarrow{q}_i$ must have the same cardinality. Additionally, for each i and for each j, either exp_j is a term, and q_{ij} is a pattern, or exp_j is a co-term, and q_{ij} is a co-pattern.*

The initial input of $\mathbf{C}_\mu$ requires a default input expression. To do this, we generate a fresh variable when we want to expand a co-term's (co-)patterns, or a fresh co-variable when we want to expand a term's (co-)patterns. Using this method, the input to $\mathbf{C}_\mu$ for *not* would be

$$\mathbf{C}_\mu([e], \left\{ \begin{array}{l} ([True \cdot \alpha],\ \langle False \parallel \alpha \rangle) \\ ([False \cdot \alpha],\ \langle True \parallel \alpha \rangle) \end{array} \right\}, \textbf{error})$$

where e is a fresh co-variable. The default case is used when matching fails on all (co-)patterns. Since there is no default case when matching fails for the entire term, **error** is used. Similarly, the input to $\mathbf{C}_\mu$ for *map* would be:

$$\mathbf{C}_\mu([e], \left\{ \begin{array}{c} ([f \cdot Cons\ x\ xs \cdot \alpha],\ \langle f \parallel x \cdot \tilde{\mu}y.\langle map \parallel f \cdot xs \cdot \tilde{\mu}z.\langle Cons\ y\ z \parallel \alpha \rangle\rangle\rangle) \\ ([f \cdot Nil \cdot \beta],\ \langle Nil \parallel \beta \rangle) \end{array} \right\}, \textbf{error})$$

Like in **C**, there are four different cases in $\mathbf{C}_\mu$ that determine the result of the algorithm.

CASE 1

$$\mathbf{C}_\mu([],\ \{\ ([],\ c)\ \},\ d)$$

This is the simplest case. There are no expressions left to match, and no (co-)patterns that need matching either. All that is left of the input is a command c and the default command d. Since there are no (co-)patterns left, it would not be possible to fail to match at this point. This means command c is returned.

CASE 2

$$\mathbf{C}_\mu([v, q_2 \dots, q_n], \begin{bmatrix} ([x_1, p_{12}, \dots, p_{1n}],\ c_1) \\ ([x_2, p_{22}, \dots, p_{2n}],\ c_2) \\ \dots \\ ([x_m, p_{m2}, \dots, p_{mn}],\ c_m) \end{bmatrix}, d)$$

or

$$\mathbf{C}_\mu([e, q_2 \cdots, q_n], \begin{bmatrix} ([\alpha_1, p_{12}, \cdots, p_{1n}],\ c_1) \\ ([\alpha_2, p_{22}, \cdots, p_{2n}],\ c_2) \\ \cdots \\ ([\alpha_m, p_{m2}, \cdots, p_{mn}],\ c_m) \end{bmatrix}, d)$$

In this case, every first (co-)pattern in the lists is a variable or a co-variable. Since matching on a variable is always possible, all that needs to happen at this stage is substituting the first input expression for all the first (co-)variables. This is achieved by returning a μ or $\tilde{\mu}$ construction. For terms, the result is:

$$\left\langle v \middle\| \tilde{\mu}[x].\mathbf{C}_\mu([q_2 \cdots, q_n], \begin{bmatrix} ([p_{12}, \cdots, p_{1n}],\ c_1\{x/x_1\}) \\ ([p_{22}, \cdots, p_{2n}],\ c_2\{x/x_2\}) \\ \cdots \\ ([p_{m2}, \cdots, p_{mn}],\ c_m\{x/x_m\}) \end{bmatrix}, d) \right\rangle$$

and for co-terms:

$$\left\langle \mu[\alpha].\mathbf{C}_\mu([q_2 \cdots, q_n], \begin{bmatrix} ([p_{12}, \cdots, p_{1n}],\ c_1\{\alpha/\alpha_1\}) \\ ([p_{22}, \cdots, p_{2n}],\ c_2\{\alpha/\alpha_2\}) \\ \cdots \\ ([p_{m2}, \cdots, p_{mn}],\ c_m\{\alpha/\alpha_m\}) \end{bmatrix}, d) \middle\| e \right\rangle$$

CASE 3

$$\mathbf{C}_\mu([v, q_2 \cdots, q_n], \begin{bmatrix} ([K_1\ cps_1\ ps_1, p_{12}, \cdots, p_{1n}],\ c_1) \\ \cdots \\ ([K_k\ cps_k\ ps_k, p_{k2}, \cdots, p_{kn}],\ c_k) \\ ([x_{k+1}, p_{k+1,2}, \cdots, p_{k+1,n}],\ c_{k+1}) \\ \cdots \\ ([x_{k+r}, p_{k+r,2}, \cdots, p_{k+r,n}],\ c_{k+r}) \end{bmatrix}, d)$$

or

$$\mathbf{C}_\mu([e, q_2 \cdots, q_n], \begin{bmatrix} ([O_1\ (cp_{11}, \cdots, cp_{1m})\ (p'_{11}, \cdots, p'_{1m}), p_{12}, \cdots, p_{1n}],\ c_1) \\ \cdots \\ ([O_k\ (cp_{k1}, \cdots, cp_{km})\ (p'_{k1}, \cdots, p'_{km}), p_{k2}, \cdots, p_{kn}],\ c_k) \\ ([\alpha_{k+1}, p_{k+1,2}, \cdots, p_{k+1,n}],\ c_{k+1}) \\ \cdots \\ ([\alpha_{k+r}, p_{k+r,2}, \cdots, p_{k+r,n}],\ c_{k+r}) \end{bmatrix}, d)$$

In this case the first pattern in each pattern sequence can be either a (co-)constructor or a (co-)variable. However, the order of the pattern sequences is important. Every pattern sequence starting with a (co-)variable comes after every pattern sequence that starts with a (co-)constructor. In this case the pattern sequences are grouped. All pattern sequences starting with the same (co-)constructor are grouped together, and all pattern sequences starting with a variable are part of the same group. For example, take the following list of pattern sequences:

$$\begin{bmatrix} ([Cons\; x\; xs, \dots],\; c_1) \\ ([Nil, \dots],\; c_2) \\ ([Cons\; y\; ys, \dots],\; c_3) \\ ([z, \dots]\; c_4) \end{bmatrix}$$

The formed groups will be $\begin{bmatrix} ([Cons\; x\; xs, \dots],\; c_1) \\ ([Cons\; y\; ys, \dots],\; c_3) \end{bmatrix}$, $\begin{bmatrix} ([Nil, \dots],\; c_2) \end{bmatrix}$ and $\begin{bmatrix} ([z, \dots]\; c_4) \end{bmatrix}$. The groups will form the different cases in the command that is returned. The result for terms is:

$$\Big\langle v \;\|\tilde{\mu}[K^1\; (x_{11}, \dots, x_{1m})\; (\alpha_{11}, \dots, \alpha_{1m})].\mathbf{C}_\mu([q_2, \dots, q_n],\; \begin{bmatrix} ([cps_{11}, ps_{11}, p_{12}, \dots, p_{1n}], \dots) \\ \dots \end{bmatrix},\; \textbf{default})\Big\rangle$$

$$\mid \dots$$

$$\mid [K^N\; (x_{j1}, \dots, x_{jm})\; (\alpha_{j1}, \dots, \alpha_{jm})].\mathbf{C}_\mu([q_2, \dots, q_n],\; \begin{bmatrix} ([cps_{j1}, ps_{j1}, p_{12}, \dots, p_{1n}], \dots) \\ \dots \end{bmatrix},\; \textbf{default})$$

$$\mid [x].\mathbf{C}_\mu(q_2, \dots, q_n,\; \{([p_{k+1,2}, \dots, p_{k+1,n}], c_{k+1}\{x/x_{k+1}\})\},\; d)$$

Here, **default** can refer to two different commands, depending on the context. If the input set of (co-)pattern sequences contains a sequence that starts with a (co-)variable, **default** refers to the result of $\mathbf{C}_\mu$ that is called for the (co-)variable case, as seen in the results above. If there is no (co-)pattern sequence starting with a (co-)variable, **default** refers to d. In that case, an extra case is added to the μ or $\tilde{\mu}$ construction: $[v].d$ or $[\alpha].d$.

Case 4

The final case is similar to case 3: The (co-)pattern sequences can start with both (co-)variables and (co-)constructors. However, in this situation, the sequences are not ordered. The pattern sequences starting with a (co-)variable are not necessarily at the end of the sequence set. To solve this issue, the (co-)pattern-sequences are once again grouped, but this time in order. The sequences are divided by repeatedly taking the longest possible prefix that is sorted. If P_k is the set of those groups, the result for both terms and co-terms is:

$$\begin{aligned} &\mathbf{C}_\mu([q_1, \dots, q_n],\; P_1,\; d_1) \\ where\; d_1 = \;&\mathbf{C}_\mu([q_1, \dots, q_n],\; P_2,\; d_2) \\ where\; d_2 = \;&\mathbf{C}_\mu([q_1, \dots, q_n],\; P_3,\; d_3) \\ &\dots \\ where\; d_k = \;&\mathbf{C}_\mu([q_1, \dots, q_n],\; P_k,\; d) \end{aligned}$$

3.2.2 Adding join points

The (co-)pattern expansion introduced by algorithm $\mathbf{C}_\mu$ can lead to code duplication. In case 3, the result of the function may contain several branches that all use the same **default** value. To demonstrate this, we define a function $secondIsTrue$ that takes a list of boolean value, and returns $True$ if the second value is $True$, but $False$ otherwise.

$$\begin{aligned} &secondIsTrue : List\; Bool \to Bool \\ &secondIsTrue = \mu((Cons\; x\; (Cons\; True\; xs)) \cdot \alpha.\langle True \parallel \alpha\rangle \\ &\qquad\qquad\qquad \mid y \cdot \beta.\langle False \parallel \beta\rangle) \end{aligned}$$

Expanding the (co-)patterns in this function using $\mathbf{C}_\mu$ yields

$$\begin{aligned}
\mu(l \cdot \alpha.\langle l \parallel \tilde{\mu}[&Cons\, x\, xs.\langle xs \parallel \tilde{\mu}[Cons\, y\, ys.\langle y \parallel \tilde{\mu}(True.\langle True \parallel \alpha\rangle)\rangle]\rangle]\rangle \\
&\qquad\qquad\qquad\qquad\qquad\qquad \mid z.\langle False \parallel \alpha\rangle \\
&\qquad\qquad\qquad \mid z.\langle False \parallel \alpha\rangle \\
&\mid z.\langle False \parallel \alpha\rangle)
\end{aligned}$$

Even for such a short and simple term, the (co-)pattern expansion algorithm has duplicated the **default** command twice. The command $\langle False \parallel \alpha\rangle$ occurs three times in the expanded version of $secondIsTrue$.

To combat this duplication of commands, we can use *join points* [Mau+17]. Instead of duplicating the **default** command, join points let us define the command once, and storing it under some variable. Whenever the command should occur, we can instead refer to this variable.

To implement this in the $\mu\tilde{\mu}$-calculus, we first introduce command-variables, and alter the definitions of commands.

$$\begin{aligned}
d \in CommandVar &::= ... \\
c \in Command &::= \langle v \parallel e\rangle \mid d
\end{aligned}$$

Using this new definition, μ-terms and $\tilde{\mu}$-co-terms are able to refer to a command using a command-variable d. To add meaning to this ability, we add let-bindings for commands to the calculus to both terms and co-terms next.

$$\begin{aligned}
v \in Term &::= \cdots \mid let\, d = c\, in\, v \\
e \in CoTerm &::= \cdots \mid let\, d = c\, in\, e
\end{aligned}$$

As command-variables are only supposed to be introduced using these let-bindings, we do not need to introduce typechecking rules that introduce a passive judgement with a command-variable. Instead, we only add rules that introduce let-bindings, and make sure that the command-variables are not present in the premises of the rules, but rather replaced by their definitions.

$$\frac{\Gamma \vdash v\{c/d\} : A \mid \Delta}{\Gamma \vdash let\, d = c\, in\, v : A \mid \Delta}\, letR \qquad \frac{\Gamma \mid e\{c/d\} : A \vdash \Delta}{\Gamma \mid let\, d = c\, in\, e : A \vdash \Delta}\, letL$$

Using these rules, we know that we can never get a valid passive judgement of a command-variable in a typechecking tree. If we do encounter such a judgement, it means that a command-variable has been used that has not been defined.

Evaluating join points can be done in one of two separate approaches. First, we could define an environment Φ, which stores command-variables and the commands they are bound to. Using this approach, we could reduce a term $let\, d = c\, in\, v$ with command environment Φ to v with the updated environment $\Phi, d = c$. Then, whenever we encounter the command-variable d, we can look up its meaning in the command environment. Alternatively, we could leave out the environment, and instead replace every occurence of d with c as soon as we encounter a let-binding. This way, a term $let\ d = c\ in\ v$ evaluates to $v\{c/d\}$. Similarly to the typechecking rules, this approach causes encountering a command-variable leading to an error, as each command-variable should have been replaced by the let rules.

Whichever approach we choose, it is important that the evaluation of let-bindings has precedence over the β-reduction rules. Not having this precedence can lead to pattern match failure. As an example, the command

$$\langle let\, d = c\, in\, Cons\ (\mu\alpha.d)\ xs \parallel \tilde{\mu}[Cons\, y\, ys.\langle ... \rangle]\rangle$$

should see the let-binding be reduced before applying β-reduction as follows:

$$\begin{gathered}\langle let\, d = c\, in\, Cons\ (\mu\alpha.d)\ xs \parallel \tilde{\mu}[Cons\, y\, ys.\langle ...\rangle]\rangle \\ \succ \\ \langle Cons\ (\mu\alpha.c)\ xs \parallel \tilde{\mu}[Cons\, y\, ys.\langle ...\rangle]\rangle \\ \succ_{\beta_F} \\ \dots\end{gathered}$$

Or, using the first approach:

$$\begin{gathered}\langle let\, d = c\, in\, Cons\ (\mu\alpha.d)\ xs \parallel \tilde{\mu}[Cons\, y\, ys.\langle ...\rangle]\rangle\ (\Phi) \\ \succ \\ \langle Cons\ (\mu\alpha.d)\ xs \parallel \tilde{\mu}[Cons\, y\, ys.\langle ...\rangle]\rangle\ (\Phi, d = c) \\ \succ_{\beta_F} \\ \dots\end{gathered}$$

If we instead give precedence to the β-reduction rules, evaluation will get stuck, since a let-binding cannot be matched with a (co-)pattern containing a constructor or an observer.

3.2.2.1 Adding join points to $\mathbf{C}_\mu$

Now that join points have been implemented in the $\mu\tilde{\mu}$-calculus, we need algorithm $\mathbf{C}_\mu$ to produce join points, instead of duplicating commands. We have already established that the duplication of commands happens by passing the same **default** command to multiple branches in a μ-term or $\tilde{\mu}$-co-term. The only point at which this happens in algorithm $\mathbf{C}_\mu$ is in case 3. Note that commands that were already duplicated in the input of $\mathbf{C}_\mu$ will still occur multiple times. This technique only stops the system from duplicating commands itself. These user-duplicated commands could be removed using join points as well, by analysing the result of $\mathbf{C}_\mu$. This thesis will, however, not go into detail about this process.

As described in Section 3.2.1, $\mathbf{C}_\mu$ takes a term or co-term, and a list of patterns and co-patterns. If the input is a term, and all the first elements in the list of (co-)patterns are patterns, the algorithm produces a command with the input term on the left side, and a $\tilde{\mu}$-co-term using each of these patterns as its cases on the right side. Likewise, if the input is a co-term, and all the first (co-)patterns are co-patterns, the produced command consists of a μ-term on the left side, and the input co-term on the right side. This generated μ-term or $\tilde{\mu}$-co-term may contain the **default** command several times. To prevent this, we enclose the generated μ-term or $\tilde{\mu}$-co-term in a let-binding where we bind a fresh command-variable d' to **default**. We then pass d' to the next recursive usages of $\mathbf{C}_\mu$, instead of **default**.

Given the input

$$\mathbf{C}_\mu([v, q_2 \dots, q_n], \begin{bmatrix} ([K_1\ cps_1\ ps_1, p_{12}, \dots, p_{1n}],\ c_1) \\ \dots \\ ([K_k\ cps_k\ ps_k, p_{k2}, \dots, p_{kn}],\ c_k) \\ ([x_{k+1}, p_{k+1,2}, \dots, p_{k+1,n}],\ c_{k+1}) \\ \dots \\ ([x_{k+r}, p_{k+r,2}, \dots, p_{k+r,n}],\ c_{k+r}) \end{bmatrix}, d)$$

$\mathbf{C}_\mu$ will produce

$$\Big\langle v \parallel let\ d' = \textbf{default}\ in$$
$$\tilde{\mu}[K^1\ (x_{11},\dots,x_{1m})\ (\alpha_{11},\dots,\alpha_{1m})].\mathbf{C}_\mu([q_2,\dots,q_n], \begin{bmatrix}([cps_{11}, ps_{11}, p_{12},\dots,p_{1n}],\dots)\\ \dots\end{bmatrix}, d')\Big\rangle$$
$$\mid \dots$$
$$\mid [K^N\ (x_{j1},\dots,x_{jm})\ (\alpha_{j1},\dots,\alpha_{jm})].\mathbf{C}_\mu([q_2,\dots,q_n], \begin{bmatrix}([cps_{j1}, ps_{j1}, p_{12},\dots,p_{1n}],\dots)\\ \dots\end{bmatrix}, d')$$
$$\mid [x].\mathbf{C}_\mu(q_2,\dots,q_n,\ \{([p_{k+1,2},\dots,p_{k+1,n}], c_{k+1}\{x/x_{k+1}\})\},\ d)$$

Likewise, with input

$$\mathbf{C}_\mu([e, q_2\dots,q_n], \begin{bmatrix}([O_1\ (cp_{11},\dots,cp_{1m})\ (p'_{11},\dots,p'_{1m}), p_{12},\dots,p_{1n}],\ c_1)\\ \dots\\ ([O_k\ (cp_{k1},\dots,cp_{km})\ (p'_{k1},\dots,p'_{km}), p_{k2},\dots,p_{kn}],\ c_k)\\ ([\alpha_{k+1}, p_{k+1,2},\dots,p_{k+1,n}],\ c_{k+1})\\ \dots\\ ([\alpha_{k+r}, p_{k+r,2},\dots,p_{k+r,n}],\ c_{k+r})\end{bmatrix}, d)$$

$\mathbf{C}_\mu$ will produce

$$\Big\langle let\ d' = \textbf{default}\ in \qquad\qquad \parallel\ e\Big\rangle$$
$$\mu[O^1\ (x_{11},\dots,x_{1m})\ (\alpha_{11},\dots,\alpha_{1m})].\mathbf{C}_\mu([q_2,\dots,q_n], \begin{bmatrix}([cps_{11}, ps_{11}, p_{12},\dots,p_{1n}],\dots)\\ \dots\end{bmatrix}, d')$$
$$\mid \dots$$
$$\mid [O^N\ (x_{j1},\dots,x_{jm})\ (\alpha_{j1},\dots,\alpha_{jm})].\mathbf{C}_\mu([q_2,\dots,q_n], \begin{bmatrix}([cps_{j1}, ps_{j1}, p_{12},\dots,p_{1n}],\dots)\\ \dots\end{bmatrix}, d')$$
$$\mid [x].\mathbf{C}_\mu(q_2,\dots,q_n,\ \{([p_{k+1,2},\dots,p_{k+1,n}], c_{k+1}\{x/x_{k+1}\})\},\ d)$$

When we use this new version of $\mathbf{C}_\mu$ on our definition of $secondIsTrue$, we now get a term without any command duplication:

$$\begin{aligned}&\mu(l\cdot\alpha.\langle l \parallel let\ d = \langle False \parallel \alpha\rangle\ in\\ &\qquad\qquad \tilde{\mu}[Cons\ x\ xs.\langle xs \parallel \tilde{\mu}[Cons\ y\ ys.\langle y \parallel \tilde{\mu}(True.\langle True \parallel \alpha\rangle)\rangle]\rangle]\rangle\\ &\qquad\qquad\qquad\qquad\qquad\qquad \mid z.d\\ &\qquad\qquad\qquad\qquad \mid z.d\\ &\qquad\qquad \mid z.d)\end{aligned}$$

3.3 MU-MU-TILDE-HASKELL

We have shown how λ-calculus programs can be converted to $\mu\tilde{\mu}$-calculus programs, and how these converted programs can be typechecked and evaluated. Using this process, we can use high-level programming languages that compile to the λ-calculus as a front-end to the $\mu\tilde{\mu}$-calculus. A widely used

$$
\begin{aligned}
X, Y, Z \in TypeVariable &::= ...\\
A, B, C \in Type &::= X \mid F\,\vec{A}\\
x \in Variable &::= ...\\
\alpha \in CoVariable &::= ...\\
c \in Command &::= v\ <||>\ e\\
v \in Term &::= x \mid K\,\overrightarrow{(v \mid e)} \mid v\,v \mid (\backslash\tilde{p} \rightarrow c) \mid (\backslash case\,\overrightarrow{\tilde{p} \rightarrow c})\\
&\mid let\,\overrightarrow{decl}\,in\,v \mid v\,where\,\overrightarrow{decl}\\
e \in CoTerm &::=\ \sim\!\alpha \mid O\,\overrightarrow{(v \mid e)} \mid (\backslash p \rightarrow c) \mid (\backslash case\,\overrightarrow{p \rightarrow c})\\
&\mid let\,\overrightarrow{decl}\,in\,e \mid e\,where\,\overrightarrow{decl}\\
p \in Pattern &::= x \mid K\,\overrightarrow{(p \mid \tilde{p})}\\
\tilde{p} \in CoPattern &::=\ \sim\!\alpha \mid O\,\overrightarrow{(p \mid \tilde{p})}\\
datadecl \in DataDeclaration &::=\ \ data\,F\,\vec{X}\ =\ \overrightarrow{K\,\overrightarrow{X \mid \sim\!X}}\\
&\mid codata\,G\,\vec{X}\ =\ \overrightarrow{K\,\overrightarrow{X \mid \sim\!X}}\\
decl \in Declaration &::=\ \ x\ ::\ A\\
&\mid \sim\!\alpha\ ::\ A\\
&\mid x\,\vec{p}\ =\ v\\
&\mid \sim\!\alpha\ =\ e\\
program \in Program &::= \overrightarrow{(datadecl \mid decl)}
\end{aligned}
$$

FIGURE 3.4: *The syntax of MMH.*

example of such a language is Haskell. We can use Haskell to write $\mu\tilde{\mu}$-calculus programs in a much more comfortable way. However, as Haskell has no notion of co-data, writing programs in plain Haskell does not allow us to fully control the $\mu\tilde{\mu}$-calculus. Instead, we propose a new programming language that is heavily based on Haskell, but has some syntactical additions and alterations to allow the unique features of the $\mu\tilde{\mu}$-calculus to be controlled: *MMH*. Figure 3.4 shows the syntactical definition of *MMH*.

At the top-level, a *MMH* program exists of declarations. A declaration can be a datatype declaration, a codatatype declaration, a (co-)term type declaration, or a (co-)term definition declaration. Datatype declarations are nearly the same as they are in Haskell. There is one difference: in the $\mu\tilde{\mu}$-calculus, and therefore in *MMH*, constructors can contain both terms and co-terms as arguments. To accomodate for this feature, we place a tilde ($\sim$) before types that represent a co-term, and do not place this tilde for types that represent a term.

$$
\begin{aligned}
data\,F\,\vec{X} = &K_1\,\overrightarrow{X \mid \sim\!X}\\
&\vdots\\
&K_n\,\overrightarrow{X \mid \sim\!X}
\end{aligned}
$$

As an example, we can define the datatype $Example$: $data\ Example\ X\ =\ T\ X\ \mid\ C\ \ \sim\!X$. To construct a value of type $Example\ X$, we either use T with a term of type X, or C with a co-term of type X. We implement co-datatype declarations the same way.

$$\begin{aligned} codata\ G\ \vec{X} = &\ O_1\ \overrightarrow{X \mid \sim X} \\ &\ \vdots \\ &\ O_n\ \overrightarrow{X \mid \sim X} \end{aligned}$$

The function co-datatype is declared in *MMH* as $codata\ a \to b\ =\ a\ .\ \sim b$. We use a similar technique to differentiate between term and co-term declarations. $name\ =\ \ldots$ will define a term called $name$, while $\sim name = \ldots$ will define a co-term $name$.

To represent commands in *MMH*, a reserved operator ($<||>$) is introduced. Using this operator, we can write $v\ \ <||> \sim a$, which is compiled to $\langle v \parallel a\rangle$ in the $\mu\tilde{\mu}$-calculus.

In Haskell, term declarations can contain patterns on the left side of the equals sign, e.g. $map\ f\ (x : xs)\ =\ \ldots$. Internally, this is read as a string of λ-functions, $\lambda f.\lambda(x : xs)\ldots$. Since this is only relevant for function terms, we allow these patterns for term declarations in *MMH*, but not in co-term declarations.

$$\begin{aligned} decl \in Declaration ::= &\ \ x\ ::\ A \\ &|\!\sim \alpha\ ::\ A \\ &|\ x\ \vec{p}\ =\ v \\ &|\!\sim \alpha\ =\ e \end{aligned}$$

Terms in Haskell consist of a variable, a constructor, a function application, a λ-function, a let-binding, or a $where$-construction. These are integrated in *MMH* accordingly. Unfortunately, this does not provide full coverage of the terms in the $\mu\tilde{\mu}$-calculus. We are able to write variables and constructors, but μ-terms cannot be written. To implement this feature, we remove the syntax for anonymous λ-functions in Haskell, and instead use a similar syntactical structure to support μ-terms. Whereas in Haskell, we could write $(\backslash x\ \ \to\ \ M)$, which would be translated to $\lambda x.M$ in the λ-calculus, we can now write $(\backslash\ \sim\ a\ \ \to\ \ v\ \ <||>\ \ e)$, or $\mu a.\langle v \parallel e\rangle$ in the $\mu\tilde{\mu}$-calculus. Of course, this same syntax also supports observers as patterns. We use the same syntax to write $\tilde{\mu}$-co-terms. The difference between the two is the same as it is in the $\mu\tilde{\mu}$-calculus: the co-term takes a pattern, while the term takes a co-pattern.

$$\begin{aligned} v \in Term ::= &\ \cdots \mid (\backslash\tilde{p} \to c) \\ e \in CoTerm ::= &\ \cdots \mid (\backslash p \to c) \end{aligned}$$

This addition does mean that the Haskell syntax for anonymous functions can no longer be used. However, non-anonymous functions can still be introduced using let-bindings, or $where$-structures. Alternatively, the new syntax for μ-terms can be used to introduce anonymous functions, although this syntax is closer to the $\mu\tilde{\mu}$-calculus than it is to the λ-calculus. Where we would write $(\backslash x\ \ \to\ \ M)$ in Haskell, we write $(\backslash x.\sim a\ \ \to\ \ M <||> \sim a)$ in *MMH*.

There is one more thing missing. As it stands, μ-terms and $\tilde{\mu}$-co-terms can only contain one single (co-)pattern in *MMH*. This means we cannot write (co-)terms that pattern match on multiple cases. To allow this, we draw inspiration from GHC's `LambdaCase` extension [Men19]. We allow the keyword $case$ to be added to $(\backslash)$-construction. When this is done, we allow multiple cases to be defined within the (co-)term as follows:

$$\begin{aligned} v \in Term ::= &\ \cdots \mid (\backslash case\ \overrightarrow{\tilde{p} \to c}) \\ e \in CoTerm ::= &\ \cdots \mid (\backslash case\ \overrightarrow{p \to c}) \end{aligned}$$

These alterations and additions to Haskell together form a solid high-level programming language that uses the $\mu\tilde{\mu}$-calculus as its foundation. Figure 3.5 shows an example *MMH* program.

```
data List a = Nil
   | Cons a (List a)
map :: (a → b) → List a → List b
map f (Cons x xs) = Cons (f x) (map f xs)
map f Nil = Nil
data Either a b = Left a
   | Right b
codata a & b = ∼a & ∼b
handleEither :: Either a b → a & b
handleEither = λcase (Left a) ∘ (∼a & ∼b) → a < || > ∼a
  (Right b) ∘ (∼a & ∼b) → b < || > ∼b)
foldr1 :: (a → b → b) → b → List a → b
foldr1 f b Nil = b
foldr1 f b (Cons x xs) = f x (foldr1 f b xs)
foldr2 :: (a → b → b) → b → List a → b
foldr2 = λcase f ∘ b ∘ Nil ∘ ∼a → b < || > ∼a
  f ∘ b ∘ (Cons x xs) ∘ ∼a →
     foldr2 < || > f ∘ b ∘ xs
        ∘ (λy → f < || > x ∘ y ∘ ∼a)
data Bool = True | False
codata Negation a = Not a
∼coval :: Negation Bool
∼coval = Not True
```

FIGURE 3.5: *An example MMH program*

3.3.1 Conclusion

The $\mu\tilde{\mu}$-calculus provides a solid basis for a modern high-level programming language. In this chapter, we have shown how λ-calculus programs can be converted to $\mu\tilde{\mu}$-calculus programs. Next, we added support for nested (co-)patterns to the $\mu\tilde{\mu}$-calculus, using an extended version of Augustsson's algorithm **C**. With these two definitions, we are able to directly convert most of the programming language Haskell to the $\mu\tilde{\mu}$-calculus. Finally, we have formalised a new language based on Haskell, called *MMH*. *MMH* features some additional syntax to add support for the unique features of the $\mu\tilde{\mu}$-calculus.

Adding polymorphism

At this point, the $\mu\tilde{\mu}$-calculus and *MMH* do not support type polymorphism. In this chapter, we will propose a way to add polymorphism to the $\mu\tilde{\mu}$-calculus in a way that is useful in the context of high-level programming languages. To do so, we will first look at a system for polymorphism in the $\mu\tilde{\mu}$-calculus by Paul Downen and Zena Ariola. We will discuss how this approach is not suited when the $\mu\tilde{\mu}$-calculus is used as a core language for a high-level programming language. Finally, we will propose our own system, based on Hindley-Milner polymorphism in the λ-calculus.

4.1 Downen and Ariola's polymorphism

One option to introduce a powerful polymorphic typing system to the $\mu\tilde{\mu}$-calculus is Downen & Ariola's *System* $\mathcal{CD}$ [DA19]. System $\mathcal{CD}$ is a polymorphic variant of the $\bar{\lambda}\mu\tilde{\mu}$-calculus [CH00]. In this system, polymorphism is possible through type functions.

Before, types consisted of type variables, or connectors. In addition to this, type functions and type applications are included. Type functions ($\lambda X.A$) take a type variable X, and produce a new type A. Type application works by applying one type to another one ($A\ B$).

$$F \in \mathit{Connector} ::= \ldots \qquad A, B, C \in \mathit{Type} ::= X \mid F(\vec{A}) \mid \lambda X.A \mid A\ B$$

In essence, the untyped λ-calculus is now embedded within the $\mu\tilde{\mu}$-calculus' type system. As a result, the type system needs its own β and η-reduction rules: $(\lambda X.A)\ B \succ_\beta A\{B/X\}$ and $\lambda X.A\ X \succ_\eta A$.

Next, we extend constructors, observers and (co-)patterns by allowing them to contain type information. In addition to terms and co-terms, constructors and observers can now have types as arguments. Patterns and co-patterns are allowed to contain type variables accordingly. An important distinction is the fact that (co-)patterns can only contain type *variables*, and not types themselves. This makes it possible to define polymorphic (co-)terms, but the system cannot pattern match on types. Since the definition of constructors and observers has been altered, we need to renew (co-)datatype declarations as well. In declarations, constructors and observers can contain type variables that represent the type arguments. These type variables are then bound to the rest of the declaration, meaning that it is possible to declare a constructor or observer with an argument that has one of the type arguments as its type.

$$\begin{aligned}
v \in Term &::= \cdots \mid K(\vec{A}, \vec{e}, \vec{v}) \mid ...\\
e \in CoTerm &::= \cdots \mid O[\vec{A}, \vec{v}, \vec{e}] \mid ...\\
p \in Pattern &::= x \mid K(\vec{X}, \vec{\tilde{p}}, \vec{p})\\
\tilde{p} \in CoPattern &::= \alpha \mid O[\vec{X}, \vec{p}, \vec{\tilde{p}}]\\
decl \in Declaration &::= \ data\ F(\vec{X})\ where\ \overrightarrow{K\,\vec{Y} : \vec{A} \vdash F(\vec{X}) \mid \vec{B}}\\
&\qquad \mid codata\ G(\vec{X})\ where\ \overrightarrow{O\,\vec{Y} : \vec{A} \mid F(\vec{X}) \vdash \vec{B}}
\end{aligned}$$

With these additions to the calculus come new typing rules. We redefine the FR and GL rules to make sure that type arguments are properly substituted in the other types in both constructors and observers.

$$\dfrac{\text{Given } data\ F(\vec{X})\ where\ \overrightarrow{K\,\vec{Y} : \vec{A} \vdash F(\vec{X}) \mid \vec{B}}: \qquad \overrightarrow{\Gamma \mid e : B_{ij}\{C/X, \overrightarrow{C'/Y}\} \vdash \Delta}^{j} \qquad \overrightarrow{\Gamma \vdash v : A_{ij}\{C/X, \overrightarrow{C'/Y}\} \mid \Delta}^{j}}{\Gamma \vdash K_i(\vec{C'}, \vec{e}, \vec{v}) : F(\vec{C}) \mid \Delta}\ FR_{K_i}$$

$$\dfrac{\text{Given } codata\ G(\vec{X})\ where\ \overrightarrow{O\,\vec{Y} : \vec{A} \mid F(\vec{X}) \vdash \vec{B}}: \qquad \overrightarrow{\Gamma \vdash v : A_{ij}\{C/X, \overrightarrow{C'/Y}\} \mid \Delta}^{j} \qquad \overrightarrow{\Gamma \mid e : B_{ij}\{C/X, \overrightarrow{C'/Y}\} \vdash \Delta}^{j}}{\Gamma \mid O_i(\vec{C'}, \vec{v}, \vec{e}) : O(\vec{C}) \vdash \Delta}\ GL_{O_i}$$

We also add new type conversion rules TCR and TCL. These conversion rules allow types to be equal under β and η reduction. They contain premises in the shape of $A =_{\beta\eta} B$. Formally, $A =_{\beta\eta} B$ when both A and B can be evaluated to the same type C using β-reduction and/or η-reduction.

$$\dfrac{\Gamma \vdash v : A \mid \Delta \qquad A =_{\beta\eta} B}{\Gamma \vdash v : B \mid \Delta}\ TCR \qquad \dfrac{\Gamma \mid e : A \vdash \Delta \qquad A =_{\beta\eta} B}{\Gamma \mid e : B \vdash \Delta}\ TCL$$

Using type parameters in constructors and observers, we are able to implement both existential and universal quantification as user-defined (co-)datatypes. We denote type variables by prefixing them with '@'.

$$codata\ Forall\ x\ = \qquad\qquad data\ Exists\ x\ =$$
$$Forall\ @y\ {\sim}(x\ y) \qquad\qquad Exists\ @y\ (x\ y)$$

Using the co-datatype for universal quantification, we are able to write polymorphic functions, such as a polymorphic identity function.

$$\begin{aligned}
&id\ ::\ Forall\ (\backslash x.x \to x)\\
&id\ =\ \backslash(Forall\ @y\ (x \cdot\ {\sim}a))\ \to\ x <||> {\sim}a\\
\\
&idTrue\ ::\ Bool\\
&idTrue\ =\ \backslash\ {\sim}a\ \to\ id <||> Forall\ @Bool\ (True \cdot {\sim}a)
\end{aligned}$$

We can use the new typing rules to typecheck these terms. First, we will typecheck the identity function. To do so, we start with the judgement $\vdash \mu(Forall\ Y\ (x \cdot \alpha).\langle x \parallel \alpha\rangle) : Forall\ (\lambda X.X \to X) \mid$. The GR rule lets us remove the co-pattern from this term, and typecheck its command instead. To do so, we first check the outer co-pattern: $Forall\,Y\;(x \cdot \alpha)$. The $Forall$ co-datatype definition tells us that for the type $Forall\;(\lambda X.X \to X)$, the pattern $(x \cdot \alpha)$ must be of type $(\lambda X.X \to X)\;Y$. We can use the β-reduction rule to reduce this type to $Y \to Y$. We can now split the pattern $x \cdot \alpha$ into separate parts again, and find out that both x and α must be of type Y. We insert them into the input and output environments, and can continue typechecking the judgement $\langle x \parallel \alpha\rangle : (x : Y \vdash \alpha : Y)$. To do so, we apply the Cut rule to split off the term and co-term. Both can be inmediately verified using the VR and VL rules respectively.

$$\dfrac{\dfrac{\dfrac{}{x : Y \vdash x : Y \mid}\,VR \qquad \dfrac{}{\mid \alpha : Y \vdash \alpha : Y}\,VL}{\langle x \parallel \alpha \rangle : (x : Y \vdash \alpha : Y)}\,Cut}{\vdash \mu(Forall\,Y\;(x \cdot \alpha).\langle x \parallel \alpha\rangle) : Forall\;(\lambda X.X \to X) \mid}\,GR$$

Next, we will typecheck the usage of the identity function in $idTrue$. We start with the judgement $id : Forall\;(\lambda X.X \to X) \vdash \mu\alpha.\langle id \parallel Forall\;Bool\;(True \cdot \alpha)\rangle : Bool \mid$, as the type of id has already been proven. We use the AR rule to remove α, and instead continue checking the command. We use the Cut rule to split the command, and need to prove that both sides of the command have the same type. We already know id's type, meaning both sides need to be of type $Forall\;(\lambda X.X \to X)$. The term-premise is swiftly closed by the VR rule, but the co-term-premise needs more work. We need to prove the judgement $\mid Forall\;Bool\;(True \cdot \alpha) : Forall\;(\lambda X.X \to X) \vdash \alpha : Bool$. We can use the GL rule to do this. The definition of $Forall$ tells us that the co-term $True \cdot \alpha$ needs to be of type $(\lambda X.X \to X)\;Bool$, as $Bool$ is the type argument to $Forall$ in this example. We can further evaluate this type to $Bool \to Bool$ using β-reduction. Knowing this, we can use the GL once again to split $True \cdot \alpha$ into a separate term and co-term. These new judgements can be proven using the FR and VL rules.

$$\dfrac{\dfrac{\dfrac{}{id : \cdots \vdash id : \cdots \mid}\,VR \qquad \dfrac{\dfrac{\dfrac{}{\vdash True : Bool \mid}\,FR \qquad \dfrac{}{\mid \alpha : Bool \vdash \alpha : Bool}\,VL}{\mid True \cdot \alpha : (\lambda X.X \to X)\;Bool \vdash \alpha : Bool}\,GL}{\mid Forall\;Bool\;(True \cdot \alpha) : Forall\;(\lambda X.X \to X) \vdash \alpha : Bool}\,GL}{\langle id \parallel Forall\;Bool\;(True \cdot \alpha)\rangle : (id : Forall\;(\lambda X.X \to X) \vdash \alpha : Bool)}\,Cut}{id : Forall\;(\lambda X.X \to X) \vdash \mu\alpha.\langle id \parallel Forall\;Bool\;(True \cdot \alpha)\rangle : Bool \mid}\,AR$$

4.1.1 Type inference

Although very powerful, there is a limitation to this type of polymorphism: type inferencing is undecidable. We can find the types of non-polymorphic (co-)terms by attempting to typecheck the (co-term) using a fresh type variable as the (co-)terms supposed type. The typing rules tell us useful information about this type in the shape of equality. For example, if we have the judgement $\vdash True : \tau \mid$, we can use the FR rule to find that type variable τ must be equivalent to the type $Bool$. This is denoted as $\tau \sim Bool$. These equivalencies, often called *constraints* [MH88], can be accumulated throughout a typechecking tree. After this is done, we can use them to evaluate the type of the original judgement by substituting types according to the found constraints. Using this technique, we can, for instance, infer the type of not. We start a proof tree with the judgement $\vdash \mu(True \cdot \alpha.\langle False \parallel \alpha\rangle \mid False \cdot \alpha.\langle True \parallel \alpha\rangle) : \tau_1 \mid$ where τ_1 is a fresh type variable. We can apply the GR rule to remove the μ, and continue proving the types of the commands in all branches. Doing so, we first use co-pattern matching to fill the input and output environments, and report any constraints to τ_1. There are two co-patterns in the μ-term: $True \cdot \alpha$ and $False \cdot \alpha$. As the co-patterns are call-stacks, we know that τ_1 must be equivalent to a function type. We also know that $True$ and $False$ have type $Bool$. We do not know the type of α yet, however. We therefore assign a fresh type variable τ_2 to α in both branches, and store the constraint $\tau_1 \sim Bool \to \tau_2$.

$$\dfrac{\begin{array}{c}\vdots\\ \langle False \parallel \alpha\rangle : (\vdash \alpha : \tau_2)\end{array} \qquad \begin{array}{c}\vdots\\ \langle True \parallel \alpha\rangle : (\vdash \alpha : \tau_2)\end{array} \qquad \tau_1 \sim Bool \to \tau_2}{\vdash \mu(True \cdot \alpha.\langle False \parallel \alpha\rangle \mid False \cdot \alpha.\langle True \parallel \alpha\rangle) : \tau_1 \mid}\,GR$$

Next, we continue with the two commands. Both can be proven in the same way. We apply the Cut rule to split the command into a term and co-term. The Cut rule tells us that the term and co-term need to be of the same type. We already know that α is of type τ_2 in both comands, while $True$ and $False$ are constructors of the datatype $Bool$. As such, we add the constraint $\tau_2 \sim Bool$. The remaining two judgements can be proven with the FR and the VL rules in both branches.

$$\dfrac{\dfrac{}{\vdash False : Bool \mid}\,FR \qquad \dfrac{}{\mid \alpha : \tau_2 \vdash \alpha : \tau_2}\,VL \qquad \tau_2 \sim Bool}{\langle False \parallel \alpha \rangle : (\vdash \alpha : \tau_2)}\,Cut$$

$$\dfrac{\dfrac{}{\vdash True : Bool \mid}\,FR \qquad \dfrac{}{\mid \alpha : \tau_2 \vdash \alpha : \tau_2}\,VL \qquad \tau_2 \sim Bool}{\langle True \parallel \alpha \rangle : (\vdash \alpha : \tau_2)}\,Cut$$

After finishing the entire proof tree, we are left with two constraints: $\tau_1 \sim Bool \to \tau_2$ and $\tau_2 \sim Bool$. We can combine these constraints to find that the type of *not* must be $Bool \to Bool$.

However, this technique does not work for polymorphic (co-)terms. We can try to execute the same steps for the polymorphic identity function we defined earlier. We start with the judgement $\vdash \mu(Forall\,Y\,(x \cdot \alpha).\langle x \parallel \alpha \rangle) : \tau_1 \mid$. We first apply the GR rule to remove μ-term and its co-pattern, and continue with the command $\langle x \parallel \alpha \rangle$. In doing so, we find two constraints. The first is $\tau_1 \sim Forall\,\tau_2$, because the μ-term contains a $Forall$ constructor, but we are not sure of the $Forall$ connector's argument. We do know that applying Y to this argument would give us a function type $\tau_3 \to \tau_4$, since $x \cdot \alpha$ is a call-stack. We therefore get the constraint $\tau_2\,Y \sim \tau_3 \to \tau_4$, and the input and output environment are extended with $x : \tau_3$ and $\alpha : \tau_4$. Next, we can split the command $\langle x \parallel \alpha \rangle$ by using the Cut rule. This tells us that x and α are of the same type, leading to the constraint $\tau_4 \sim \tau_3$. We can then use the verification rules to finish the proof.

$$\dfrac{\dfrac{\dfrac{}{x : \tau_3 \vdash x : \tau_3 \mid}\,VR \qquad \dfrac{}{\mid \alpha : \tau_4 \vdash \alpha : \tau_4}\,VL \qquad \tau_4 \sim \tau_3}{\langle x \parallel \alpha \rangle : (x : \tau_3 \vdash \alpha : \tau_4)}\,Cut \qquad \tau_1 \sim Forall\,\tau_2, \tau_2\,Y \sim \tau_3 \to \tau_4}{\vdash \mu(Forall\,Y\,(x \cdot \alpha).\langle x \parallel \alpha \rangle) : \tau_1 \mid}\,GR$$

As a result, we have three constraints: $\tau_1 \sim Forall\,\tau_2$, $\tau_2\,Y \sim \tau_3 \to \tau_4$, and $\tau_3 \sim \tau_4$. We can simplify this by applying the last constraint to the others. We are then left with two constraints: $\tau_1 \sim Forall\,\tau_2$ and $\tau_2\,Y \sim \tau_3 \to \tau_3$. We are not able to solve this set of constraints any further. We cannot possibly determine what τ_2 must be, which means that all we know of the type of the identity function is that it must be $Forall\,\tau_2$, where τ_2 could be anything. As this is not useful information, this example proves that this typing system does not suffice, and another approach is needed.

4.2 Hindley-Milner polymorphism

Type inferencing being undecidable for polymorphic terms is a problem that is not unique to the $\mu\tilde{\mu}$-calculus. In fact, it holds true for the polymorphic λ-calculus *System F* as well [Wel99]. To support both polymorphism and type inference, many functional programming languages use the Hindley-Milner type system [Mil78].

The Hindley-Milner type system was invented for the λ-calculus, but its core ideas can be used in the $\mu\tilde{\mu}$-calculus. In the λ-calculus this typing system differentiates between types and type *schemes*. Types can be type variables or connectors, while type schemes are either a type, or a universally quantified type. Most terms have types, but top-level declared terms and let-bound terms are allowed to have a type scheme. Using this system, the λ-calculus can support polymorphism, but only on the top-level.

$$F \in Connector ::= ... \qquad A, B, C \in Type ::= X \mid F(\overrightarrow{A}) \qquad \sigma \in TypeScheme ::= A \mid \forall X.A$$

It is not possible to use a constructor to create a quantified type. Instead, we *generalise* the inferred types of all top-level definitions. For example, if we have a top level definition $id = \lambda x.x$, we can infer its type as follows:

$$\dfrac{\dfrac{\tau_2 \sim \tau_3}{x : \tau_3}\, x \qquad \tau_1 \sim \tau_2 \to \tau_3}{\lambda x.x : \tau_1} \to I_x$$

The constraints we find in this typechecking tree, $\tau_1 \sim \tau_2 \to \tau_3$ and $\tau_2 \sim \tau_3$, tell us that τ_1 can be rewritten to $\tau_3 \to \tau_3$. We generalise types by finding all free type variables in the type, and binding them with a universal quantifier. In the case of id, this means the final type will be $\forall \tau_3.\tau_3 \to \tau_3$.

Whenever id is used in a term, we *instantiate* its type. If we infer the type of the term $id\, True$, we start the process by claiming its type is a fresh type variable τ_1. We then use the $\to E$ rule to split the term into two parts: id and $True$. We know that id must have some function type that produces a term of type τ_1. We also know that id takes $True$ as an argument. Since $True$'s type is $Bool$, we know that in this step, id must be of type $Bool \to \tau_1$. As we have determined previously, id's fully quantified type is actually $\forall \tau_3.\tau_3 \to \tau_3$. This means that $Bool \to \tau_1$ must be an *instance* of $\forall \tau_3.\tau_3 \to \tau_3$. This is a new type of constraint, denoted as $Bool \to \tau_1 < \forall \tau_3.\tau_3 \to \tau_3$.

$$\dfrac{\dfrac{Bool \to \tau_1 < \forall \tau_3.\tau_3 \to \tau_3}{id : Bool \to \tau_1} \qquad \dfrac{}{True : Bool}}{id\, True : \tau_1} \to E$$

We can instantiate the right-hand side of this constraint by passing it a fresh type-variable τ_4. This gives us the constraint $Bool \to \tau_1 \sim \tau_4 \to \tau_4$. This constraint can then be split into $\tau_1 \sim \tau_4$ and $\tau_4 \sim Bool$. Reducing these constraints once more gives us $\tau_1 \sim Bool$, which means that the type of the term $id\, True$ must be $Bool$.

4.2.1 HINDLEY-MILNER IN THE $\mu\tilde{\mu}$-CALCULUS

We can implement a similar typing system in the $\mu\tilde{\mu}$-calculus, and therefore in *MMH*. Just like in the λ-calculus, we differentiate between types and type schemes. (Co-)terms that are defined at the top-level, in let-bindings or in where-clauses have type schemes, other (co-)terms have types. To do this, the inferred types of all the (co-)terms that can have a type scheme are generalised. Because of these changes, our type inferencing process can now result in two kinds of constraints: equivalencies ($A \sim B$), and instances ($A < B$). The difference between the two is subtle. $A < B$ means that A and B must be equivalent when all quantification from B is removed, and the quantified variables are replaced with fresh variables. $A \sim B$ means that A and B are simply equivalent. We define three constraint rules that allow instance constraints to be simplified.

The first rule we define is the $Con - Mono$ rule. This rules specifies that if two *types* - and not type schemes - are equal, they are automatically an instance of each other.

$$\dfrac{A \sim B}{A < B}\; Con - Mono$$

Next, we add the $Con - Inst$ rule. This rule tells us that if a type scheme σ_1 is an instance of another type scheme σ_2, σ_1 will also be an instance of a quantified version of σ_2.

$$\dfrac{\sigma_1 < \sigma_2\{Y/X\}}{\sigma_1 < \forall X.\sigma_2}\; Con - Inst$$

Then, we define the $Con - Skol$ rule, which dictates that for a type scheme σ_1 that is an instance of another type scheme σ_2, a quantified version of σ_1 is also an instance of σ_2, as long as the type parameter of this quantified version does not occur in σ_2.

$$\begin{array}{rr} A, B, C \in Type ::= X \mid F(\vec{A}) & X, Y, Z \in TypeVariable ::= \ldots \\ \sigma \in TypeScheme ::= A \mid \forall X.\sigma & F, G \in Connector ::= \ldots \end{array}$$

Core rules:

$$\frac{B < A}{x : A \vdash x : B \mid}\, VR \qquad \frac{B < A}{\mid \alpha : B \vdash \alpha : A}\, VL$$

Constraint rules:

$$\frac{A \sim B}{A < B}\, CM \qquad \frac{}{A \sim A}\, CV$$

$$\frac{\sigma_1 < \sigma_2\{Y/X\}}{\sigma_1 < \forall X.\sigma_2}\, CI \qquad \frac{\sigma_1 < \sigma_2 \qquad X \notin FV(\sigma_2)}{\forall X.\sigma_1 < \sigma_2}\, CS$$

FIGURE 4.1: *Changes to the $\mu\tilde{\mu}$-calculus to support the Hindley-Milner type system.*

$$\frac{\sigma_1 < \sigma_2 \qquad X \notin FV(\sigma_2)}{\forall X.\sigma_1 < \sigma_2}\, Con - Skol$$

Finally, we add one more rule: the $Con - Verify$ rule. This rule does not handle instante constraints, but it allows equivalency constraints to be removed if the constraint represent a type being equivalent to itself. These kinds of constraints are trivially true, and therefore not useful.

$$\frac{}{A \sim A}\, Con - Verify$$

Figure 4.1 shows all the changes that are made to the $\mu\tilde{\mu}$-calculus to support the Hindley-Milner type system.

With these new rules in place, we are able to write polymorphic (co-)terms. Just like we did for the λ-calculus, we will write an identity function id, and show its type inferencing process. As we have seen before, an identity function in the $\mu\tilde{\mu}$-calculus can be written as $\mu(x \cdot \alpha.\langle x \parallel \alpha\rangle)$. We will start the type inference process by claiming this term has type τ_1. We can use the GR rule to continue. This tells us that τ_1 must be of some function type. We therefore get the constraint $\tau_1 < \tau_2 \to \tau_3$, and insert $x : \tau_2$ and $\alpha : \tau_3$ into the input and output environments. As the constraint has unquantified types on both sides of the instance operator, it can be reduced to the equivalency constraint $\tau_1 \sim \tau_2 \to \tau_3$. Next, we typecheck the command $\langle x \parallel \alpha\rangle$, with environments $x : \tau_2 \vdash \alpha : \tau_3$. We use the Cut rule to split the command. As both sides of the command must have the same type, we obtain the constraint $\tau_2 \sim \tau_3$. We also obtain two more judgements to prove: $x : \tau_2$ and $\alpha : \tau_3$. These can be proven using the VR and VL rules respectively. This gives us the constraints $\tau_2 < \tau_2$ and $\tau_3 < \tau_3$. These can be reduced to equivalencies using the CM rule, after which they can be removed with the CV rule.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\tau_2 \sim \tau_2}\, CV}{\tau_2 < \tau_2}}{x : \tau_2 \vdash x : \tau_2 \mid}\, VR \qquad \dfrac{\dfrac{\dfrac{}{\tau_3 \sim \tau_3}\, CV}{\tau_3 < \tau_3}}{\mid \alpha : \tau_3 \vdash \alpha : \tau_3}\, VL \qquad \tau_2 \sim \tau_3}{\langle x \parallel \alpha\rangle : (x : \tau_2 \vdash \alpha : \tau_3)}\, Cut \qquad \dfrac{\tau_1 \sim \tau_2 \to \tau_3}{\tau_1 < \tau_2 \to \tau_3}\, CM}{\vdash \mu(x \cdot \alpha.\langle x \parallel \alpha\rangle) : \tau_1 \mid}\, GR$$

The constraints we get from the typechecking tree are $\tau_2 \sim \tau_3$ and $\tau_1 \sim \tau_2 \to \tau_3$. We can reduce this to $\tau_1 \sim \tau_3 \to \tau_3$. Finally, we generalise this type and find that id's type is $\forall \tau_3.\tau_3 \to \tau_3$.

Next, we will infer the type of a term that uses id: $\mu\alpha.\langle id \parallel True \cdot \alpha\rangle$. As usual, we start with a judgement that binds this term to a fresh type variable τ_1. We already know the type of id, which is stored

$$\frac{\Gamma \vdash v_1 : B \mid \Delta \qquad \Gamma, x : gen(B) \vdash v_2 : A \mid \Delta}{\Gamma \vdash let\; x = v_1\; in\; v_2 : A \mid \Delta}\; LetR_1$$

$$\frac{\Gamma \mid e : B \vdash \Delta \qquad \Gamma \vdash v : A \mid \alpha : gen(B), \Delta}{\Gamma \vdash let\; \alpha = e\; in\; v : A \mid \Delta}\; LetR_2$$

$$\frac{\Gamma \vdash v : B \mid \Delta \qquad \Gamma, x : gen(B) \mid e : A \vdash \Delta}{\Gamma \mid let\; x = v\; in\; e : A \vdash \Delta}\; LetL_1$$

$$\frac{\Gamma \mid e_1 : B \vdash \Delta \qquad \Gamma \mid e_2 : A \vdash \alpha : gen(b), \Delta}{\Gamma \mid let\; \alpha = e_1\; in\; e_2 : A \vdash \Delta}\; LetL_2$$

FIGURE 4.2: *Typing rules for let-bindings in MMH. These rules can be used for top-level definitions and where-clauses too.*

in the input environment. The starting judgement is $id : \forall\tau_3.\tau_3 \to \tau_3 \vdash \mu\alpha.\langle id \parallel True\cdot\alpha\rangle : \tau_1 \mid$. The first step is to apply the AR rule. This removes the μ-term, and places $\alpha : \tau_1$ in the output environment. The remaining judgement is a passive one, and it can be split using the Cut rule. This rule forces the term id and the co-term $True \cdot \alpha$ to be of the same type. Since we know that $True$ has type $Bool$ and α has type τ_1, this type will be $Bool \to \tau_1$. The co-term side is relatively easily proven. We apply the GL rule to split it into $True$ and α. We can use FR to finish the judgement $\vdash True : Bool \mid$. Applying VL to $\mid \alpha : \tau_1 \vdash \alpha : \tau_1$ gives us the constraint $\tau_1 < \tau_1$. Since both of these types are unquantified, and equivalent, we can use the CM and CV rules to finish this branch of the proof. Next, we go back to proving the judgement $id : \forall\tau_3.\tau_3 \to \tau_3 \vdash id : Bool \to \tau_1 \mid$. We can use the VR rule to get the constraint $Bool \to \tau_1 < \forall\tau_3.\tau_3 \to \tau_3$. The right-hand side of this constraint is then instantiated by the CI rule. This results in $Bool \to \tau_1 < \tau_4 \to \tau_4$. As both types are now unquantified, we can use CM to turn the instance constraint into an equivalency constraint $Bool \to \tau_1 \sim \tau_4 \to \tau_4$. This is the only constraint that remains from the type inference tree. It can be split into two new constraints: $\tau_4 \sim Bool$ and $\tau_1 \sim \tau_4$. Combining these two results into $\tau_1 \sim Bool$, which means the original term has type $Bool$.

$$\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{Bool \to \tau_1 \sim \tau_4 \to \tau_4}{Bool \to \tau_1 < \tau_4 \to \tau_4}\; CM}{Bool \to \tau_1 < \forall\tau_3.\tau_3 \to \tau_3}\; CI}{id : \forall\tau_3.\tau_3 \to \tau_3 \vdash id : Bool \to \tau_1 \mid}\; VR \qquad \cfrac{\cfrac{}{\vdash True : Bool \mid}\; FR \quad \cfrac{\cfrac{\cfrac{}{\tau_1 \sim \tau_1}\; CV}{\tau_1 < \tau_1}\; CM}{\mid \alpha : \tau_1 \vdash \alpha : \tau_1}\; VL}{\mid True \cdot \alpha : Bool \to \tau_1 \vdash \alpha : \tau_1}\; GL}{\langle id \parallel True \cdot \alpha\rangle : (id : \forall\tau_3.\tau_3 \to \tau_3 \vdash \alpha : \tau_1)}\; Cut}{id : \forall\tau_3.\tau_3 \to \tau_3 \vdash \mu\alpha.\langle id \parallel True \cdot \alpha\rangle : \tau_1 \mid}\; AR$$

We have seen how polymorphic (co-)terms can be defined and used in the $\mu\tilde{\mu}$-calculus. This is made possible by generalising the types of all top-level definitions. In *MMH*, we need several extra typing rules, because it should also be possible to define polymorphic (co-)terms using let-bindings and where-clauses. To support this, the types of (co-)terms defined this way must be generalised. Figure 4.2 formalises the typing rules that facilitate this.

4.2.2 CONCLUSION

We have seen two separate approaches to type polymorphism in the $\mu\tilde{\mu}$-calculus. First, we have explored Downen and Ariola's system. While this system is very powerful, it renders type inference undecidable. For this reason, we have implemented Hindley-Milner polymorphism into the $\mu\tilde{\mu}$-calculus. Using this system, we have shown how it is possible to write polymorphic terms and co-terms, and how the types of these (co-)terms can be inferred.

5

Closing remarks

5.1 Conclusions

This thesis has proposed a modern high-level programming language called *MMH*, that uses the $\mu\tilde{\mu}$-calculus at its core. Syntactically, *MMH* is an extension to Haskell. *MMH* has been formalised by first defining conversion steps for λ-calculus terms to $\mu\tilde{\mu}$-calculus terms. Since Haskell can be compiled to the λ-calculus, this conversion method lets Haskell programs be converted to a collection of $\mu\tilde{\mu}$-calculus terms. Next, support for nested (co-)patterns in the $\mu\tilde{\mu}$-calculus has been implemented. This is done by formalising a pattern expansion algorithm $\mathbf{C}_\mu$, based on Augustsson's algorithm **C**. Finally, custom syntactical extensions have been added to Haskell to allow co-terms and commands to be written. This way, users of *MMH* are able to take full control of the power of the $\mu\tilde{\mu}$-calculus.

In addition, the $\mu\tilde{\mu}$-calculus and *MMH* have been given support for polymorphic types. This has been done by adapting the Hindley-Milner typing system for the $\mu\tilde{\mu}$-calculus.

5.2 Future work

5.2.1 Formalisation

Although *MMH* has strong fundamental roots in both the λ-calculus and the $\mu\tilde{\mu}$-calculus, its soundness has not been proven in this thesis. For future stability, proving this, as well as Turing completeness, would be a logical next step.

5.2.2 Adding modern features

MMH can be used as a general purpose programming language. However, it lacks certain features that the modern programmer may expect. For instance, there is as of yet no way of creating interfaces. This could perhaps be done by implementing type classes in a similar way as Haskell has it. Likewise, the specifications of *MMH* in this thesis do not discuss splitting programs over multiple files. This means that *MMH* programs need to be written in one single file, and there is no way to organise code into chunks. From a user's point of view, this is unacceptable for a modern programming language. Another feature that is lacking in *MMH*, is a way to interact with the user. As it stands, *MMH* programs cannot read or write files, and they have no way of receiving user input. Implementing such features could be interesting.

5.2.3 Code generation

This thesis has focused on compiling programs to the $\mu\tilde{\mu}$-calculus. However, in a compiler, $\mu\tilde{\mu}$-calculus would not be the end station. The $\mu\tilde{\mu}$-calculus would be used as an intermediate language. More research is needed on converting this intermediate language to actual machine code. An especially interesting point here, are the call-stacks that are generated by functions in *MMH*. Since this neatly organises all computations in a program in a strict order, they can perhaps be taken advantage of.

5.2.4 Adding dependent types

In dependently typed languages, types traditionally are seen as terms. How could a dependently typed programming language be built on top of the $\mu\tilde{\mu}$-calculus? The $\mu\tilde{\mu}$-calculus differentiates between terms and co-terms. What would the effect of this be in a dependently typed setting?

5.2.5 Converting $\mu\tilde{\mu}$-calculus programs to the λ-calculus

Section 3.1 describes a process of transforming λ-calculus programs to $\mu\tilde{\mu}$-calculus programs. Doing this in the reversed direction could be an interesting alternative, as the λ-calculus does not contain a direct replacement for co-terms in the $\mu\tilde{\mu}$-calculus.

Listing of figures

Bibliography

[Aug84] Lennart Augustsson. “A compiler for lazy ML”. In: *Proceedings of the 1984 ACM Symposium on LISP and functional programming*. ACM. 1984, pp. 218–227.

[Aug85] Lennart Augustsson. “Compiling pattern matching”. In: *Conference on Functional Programming Languages and Computer Architecture*. Springer. 1985, pp. 368–381.

[CH00] Pierre-Louis Curien and Hugo Herbelin. “The duality of computation”. In: *ACM sigplan notices* 35.9 (2000), pp. 233–243.

[Chu36] Alonzo Church. “An unsolvable problem of elementary number theory”. In: *American journal of mathematics* 58.2 (1936), pp. 345–363.

[Chu40] Alonzo Church. “A formulation of the simple theory of types”. In: *The journal of symbolic logic* 5.2 (1940), pp. 56–68.

[DA19] Paul Downen and Zena M Ariola. “Compiling With Classical Connectives”. In: *arXiv preprint arXiv:1907.13227* (2019).

[Dow+16] Paul Downen et al. “Sequent calculus as a compiler intermediate language”. In: *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*. 2016, pp. 74–88.

[Dow17] Paul Downen. “Sequent Calculus: A Logic and a Language for Computation and Duality”. In: (2017).

[Gen35] Gerhard Gentzen. “Untersuchungen über das logische Schließen. I”. In: *Mathematische zeitschrift* 39.1 (1935), pp. 176–210.

[Her05] Hugo Herbelin. “C’est maintenant qu’on calcule: au cœur de la dualité”. In: *Mémoire d’habilitation, available from cited url* (2005).

[Mau+17] Luke Maurer et al. “Compiling without continuations”. In: *ACM SIGPLAN Notices*. Vol. 52. 6. ACM. 2017, pp. 482–494.

[Men19] Alejandro Serrano Mena. “Increasing Code Reuse”. In: *Practical Haskell*. Springer, 2019, p. 71.

[MH88] John C Mitchell and Robert Harper. “The essence of ML”. In: *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 1988, pp. 28–46.

[Mil78] Robin Milner. “A theory of type polymorphism in programming”. In: *Journal of computer and system sciences* 17.3 (1978), pp. 348–375.

[Wel99] Joe B Wells. “Typability and type checking in System F are equivalent and undecidable”. In: *Annals of Pure and Applied Logic* 98.1-3 (1999), pp. 111–156.