Eberhard Karls Universität Tübingen
Mathematisch-Naturwissenschaftliche Fakultät
Wilhelm-Schickard-Institut für Informatik

Bachelor Thesis Computer Science

Higher-Order Unification for Data and Codata Types

Julia Wegendt

Date

Reviewer

Prof. Dr. Klaus Ostermann
Department of Computer Science
University of Tübingen

Wegendt, Julia
Higher-Order Unification for Data and Codata Types
Bachelor Thesis Computer Science
Eberhard Karls Universität Tübingen
Period: from-till

Abstract

Algorithms for solving different subproblems of the general higher-order unification problem have been discussed in the literature. I present an untyped calculus with a focus on the data-codata duality, and discuss typing for this calculus.

Furthermore, I go over the decidable subproblem of first-order unification and an accompanying algorithm adapted to our syntax. I explain higher-order unification and the specific problem of pattern unification formally, and discuss how they differ, especially what parts are decidable.

Finally, I present a framework for our algorithm, as well as our unification algorithm for the aforementioned calculus. This algorithm covers pattern unification problems, as well as problems which can be reduced to pattern unification problems.

Acknowledgements

Write here your acknowledgements.

Contents

1 Introduction

In this thesis, I want to present a higher-order unification algorithm for a calculus with codata and data.. For this, you need to know what data and codata conceptually are, how they are presented in our syntax and how they function. You also need to know what higher-order unification is and how it differs from the first-order unification you probably already know. These are the goals for the next few paragraphs, after which I then give a quick overview of the different parts of this thesis.

1.1 Data and codata

While algebraic data are a well-known concept, their brother codata is less talked about. Codata is dual to data, which (roughly) means that you can express any concept with codata that you would with data. On a high level: While data is concerned with how values are constructed or put in, codata is concerned with how values are destructed or taken out. For instance, look at these different ways to define the same tuple:

Example 1.1 (Two ways to define the same tuple)**.**

$$\begin{aligned} &\texttt{Tup}(1, 5) && \text{Data} \\ &\textbf{cocase}\ \{\texttt{fst} \Rightarrow 1, \texttt{snd} \Rightarrow 5\} && \text{Codata} \end{aligned}$$

While the first variant focuses on what we put into the tuple, the second focuses on how we can take its components out of it.

One form of codata often talked about is streams:

Example 1.2 (Streams)**.**

$$\mathit{Trues} = \textbf{cocase}\ \{\texttt{hd} \Rightarrow \texttt{True}, \texttt{tl} \Rightarrow \mathit{Trues}\}$$

This is the stream that contains an infinite amount of Trues. Also notice again that there is an emphasis on how we can take things out of our stream, rather than how it is constructed. As you can see, we can also instantiate infinite codata.

In our syntax, codata is also used for function application, which is the part of codata you are likely most familiar with:

Example 1.3 (Function application through codata)**.**

$$\mathit{id} = \textbf{cocase}\ \{\texttt{ap}(x) \Rightarrow x\}$$

This is the identity function, which we get from mapping the function applicator `ap`() to the same variable we apply it to.

1.2 Higher-Order Unification

Most people probably first encounter unification problems in school, when solving simple math problems. Unification is also necessary in a few important applications in computer science: Type checking and type inference, in proof assistants, as well as more particular fields like logic programming and computational linguistics.

While unification means solving equations with unknown variables, there are two major types of unification: First-order unification and higher-order unification. First-order unification is enough for the type systems in most typed programming languages. It is only when more complex types are introduced, like in the interactive theorem prover Coq or in the programming language Agda, that higher-order unification is required.

Example 1.4 (First-order unification problems)**.**

$$\mathtt{5} \equiv \alpha^? \tag{1}$$
$$\mathtt{True} \equiv \mathtt{False} \tag{2}$$
$$\mathsf{List}(\alpha^?) \equiv \mathsf{List}(\beta^?) \tag{3}$$
$$\mathtt{Tup}(\mathtt{1}, \alpha^?) \equiv \beta^? \tag{4}$$

Each of the given equations can be seen as their own unification problem. The first has an obvious solution $[\mathtt{5}/\alpha^?]$, so substituting $\mathtt{5}$ for $\alpha^?$ (even though there are multiple solutions even here). The second is a unification problem without unification variables, and has no solution. The third and forth are both unification problems with two unification variables, and have many solutions, but one most general solution.

Note that unification problems may also have multiple equations.

Example 1.5 (Higher-order unification problems)**.** In this first example, we are looking for a function that when applied to $\mathtt{5}$, evaluates to $\mathtt{5}$:

$$\alpha^?.\mathtt{ap}(\mathtt{5}) \equiv \mathtt{5} \tag{1}$$

We can think of two obvious solutions for this: The identity function: $[id/\alpha^?]$, and the constant function which always returns $\mathtt{5}$: $[\mathbf{cocase}\ \{\mathtt{ap}(x) \Rightarrow \mathtt{5}\}/\alpha^?]$. As already implied before, sometimes our unification problems have multiple solutions, but we are interested in one particular solution. This is the solution that is more general, the one from which the others can be obtained by instantiation. In this case however, there is no one such solution. This can only happen in higher-order unification, but this will be explored more formally below. Another example of higher-order unification:

$$\alpha^?.\mathbf{case}\ \{\mathtt{True} \Rightarrow \mathtt{1}, \mathtt{False} \Rightarrow \mathtt{2}\} \equiv \mathtt{1} \tag{2}$$

Here we have an example of pattern matching implementing an if-clause. If we substituted $[\texttt{True}/\alpha^?]$, the left side would reduce to `1`, since the first case matches — solving the equation.

1.3 Overview

This Bachelors thesis is structured as follows:

First, I will explore a calculus with a heavy focus on the dualities between data and codata in section 2. I will give examples and show how we can use this syntax to define everything we want to define, and point out how the duality plays out in our syntax. In addition, I will introduce some concepts we need so that we can start to play around with terms.

In section 3, I give examples of how we can type terms we have seen in our examples, and present a typing system to categorize many terms with simple types.

To make our way to higher-order unification, I will first discuss first-order unification in section 4. I explain the problem formally and give an algorithm to solve it.

Then I explain higher-order unification (section 5) formally, as well as the pattern unification problem which is easier to solve.

In section 6, I present the framework for our equations, and then present our algorithm for higher-order unification problems written in our calculus. The algorithm is made up of two parts, which I go though in more detail, and describe how they fit together.

In the end, there is a section about related works (section 7), as well as a small conclusion in section 8.

2 The Untyped Calculus ND

I will introduce the Untyped Calculus ND, based on [2]. This calculus may be seen as an extension to the lambda calculus, as function definition is only one application of codata in this calculus, meaning we can do more than just function application!

2.1 Syntax of the Untyped Calculus ND

I write $\overline{e}$ to mean a (possibly empty) list of expressions: $\overline{e} = e_1, ..., e_n$. Constructors $K(\overline{e})$ and destructors $e.d(\overline{e})$ both get such a list of expressions and construct or destruct terms. A pattern match $e.\textbf{case}\ \{\overline{K(\overline{x}) \Rightarrow t}\}$ matches an expression e against a sequence of clauses, where each clause consists of a

constructor and an term t. For a copattern match **cocase** $\{\overline{d(\overline{x}) \Rightarrow t}\}$ similar rules apply, but the clauses contain destructors instead of constructors.

Definition 2.1 (Terms of the Calculus ND)**.**

$$
\begin{aligned}
e, r, s, t ::= {} & x, y && \text{Variable} \\
\mid {} & K(\overline{e}) && \text{Constructor} \\
\mid {} & e.\mathbf{case}\ \{\overline{K(\overline{x}) \Rightarrow t}\} && \text{Pattern match} \\
\mid {} & \mathbf{cocase}\ \{\overline{d(\overline{x}) \Rightarrow t}\} && \text{Copattern match} \\
\mid {} & e.d(\overline{e}) && \text{Destructor}
\end{aligned}
$$

In pattern and copattern matching, every constructor or destructor may occur no more than once. This is so we don't match multiple clauses: In $e.\mathbf{case}\ \{\mathtt{Tup}(x_1, x_2) \Rightarrow t_1, \mathtt{Tup}(x, y) \Rightarrow t_2\}$, the same tuple could be reduced to two different terms, t_1, t_2.

I use different fonts for different uses of our syntax: Variables, and names of terms are in $VarFont$. When constructors and destructors are used for data and codata, they appear in `CtorDtorFont`, but when constructors are used for types, they appear in TypeFont.

2.2 How is this used?

To get familiar with the syntax and to understand the underlying semantics, I will go through some examples and explain what they implement.

We use constructors for types:

Example 2.1 (Constructing types with constructors)**.** Simple Types like Int, Bool, ... are constructors applied to an empty sequence, while composite types of course require arguments.

$$
\begin{aligned}
&\mathsf{Bool} && (1)\\
&\mathsf{List}(\mathsf{Int}) && (2)\\
&\mathsf{Pair}(\mathsf{Int}, \mathsf{Bool}) && (3)
\end{aligned}
$$

Types are explored later on in section 3.

We of course also want to build terms for those data types. Some terms we build with constructors:

Example 2.2 (Instantiating data with constructors)**.**

$$
\begin{aligned}
&\mathtt{True},\ \mathtt{False} && (1)\\
&\mathtt{Tup}(1, 2) && (2)\\
&\mathtt{Date}(27, 08, 2005) && (3)\\
&\mathtt{Cons}(\mathtt{True}, \mathtt{Cons}(\mathtt{False}, \mathtt{Nil})) && (4)
\end{aligned}
$$

As you can see from the last example, we can use constructors to define recursive data types. Numbers for instance, are also implemented recursively as Peano numbers. I use numerals ($\mathtt{1}, \mathtt{2}, \mathtt{5}, ...$) as a shorthand for $\mathtt{suc}(\mathtt{zero}), \mathtt{suc}(\mathtt{suc}(\mathtt{zero})), ...$ throughout this thesis, though.

If we construct terms, we might also want a way to take them apart. For the terms we just saw, this can be done using pattern matching:

Example 2.3 (Taking apart data using pattern matching)**.**

$$\mathtt{True}.\mathbf{case}\ \{\mathtt{True} \Rightarrow \mathtt{False}, \mathtt{False} \Rightarrow \mathtt{True}\} \tag{1}$$

$$\mathtt{Tup}(\mathtt{1}, \mathtt{2}).\mathbf{case}\ \{\mathtt{Tup}(x, y) \Rightarrow x\} \tag{2}$$

$$\mathtt{Cons}(\mathtt{True}, \mathtt{Cons}(\mathtt{False}, \mathtt{Nil})).\mathbf{case}\ \{\mathtt{Cons}(x, y) \Rightarrow x\} \tag{3}$$

This is the pattern matching you probably already know. We will see in definition 2.9 how these get evaluated to `False`, `1` and `True`, respectively.

Since we use pattern matching to deconstruct data like Booleans, naturally we can also use it to implement conditionals:

Example 2.4 (Representing conditionals with pattern matching)**.**

$$t.\mathbf{case}\ \{\mathtt{True} \Rightarrow e_1, \mathtt{False} \Rightarrow e_2\} \tag{1}$$

$$e.\mathbf{case}\ \{\mathtt{Cons}(\mathtt{True}, x) \Rightarrow \mathtt{True}, \mathtt{Nil} \Rightarrow \mathtt{False}, \mathtt{Cons}(x, y) \Rightarrow \mathtt{False}\} \tag{2}$$

The first example is equivalent to if t then e_1 else e_2. The second expression tests whether a given list starts with `True`.

Until now, we have seen how to instantiate data, and how to take it apart. Now this is where codata comes into play —this is how to instantiate it:

Example 2.5 (Instantiating codata with copattern matching)**.**

$$\mathbf{cocase}\ \{\mathtt{fst} \Rightarrow \mathtt{1}, \mathtt{snd} \Rightarrow \mathtt{2}\} \tag{1}$$

$$Trues = \mathbf{cocase}\ \{\mathtt{hd} \Rightarrow \mathtt{True}, \mathtt{tl} \Rightarrow Trues\} \tag{2}$$

We have seen examples like this in the introduction; the first one is a tuple with the values `1` and `2`. The second example is the infinite stream of $Trues$.

Now for how to take instantiations of codata apart:

Example 2.6 (Taking apart codata using destructors)**.**

$$\mathbf{cocase}\ \{\mathtt{fst} \Rightarrow \mathtt{1}, \mathtt{snd} \Rightarrow \mathtt{2}\}.\mathtt{fst} \tag{1}$$

$$Trues.\mathtt{hd} \tag{2}$$

$$id.\mathtt{ap}(\mathtt{5}) \tag{3}$$

These examples do exactly what you think they do: We select the first value of the tuple, take out the first element of our stream, and apply the identity function on `5`, respectively. We will see how these examples resolve to `1`, `True` and `5` in definition 2.9.

You might have already noticed that there is no option for function definition in the syntax of our calculus. This is because we can use copattern matching to implement function definitions:

Example 2.7 (Defining functions using copattern matching)**.**

$$id = \textbf{cocase}\ \{\mathtt{ap}(x) \Rightarrow x\} \tag{1}$$
$$\textbf{cocase}\ \{\mathtt{ap}(x) \Rightarrow \textbf{cocase}\ \{\mathtt{ap}(y) \Rightarrow x\}\} \tag{2}$$
$$\textbf{cocase}\ \{\mathtt{ap}(x) \Rightarrow \textbf{cocase}\ \{\mathtt{ap}(y) \Rightarrow y\}\} \tag{3}$$

The first example implements the identity function, which you have already seen. The second and third examples implement functions that take two arguments, ignore one and give back the other.

2.3 Dualities in data and codata

You might have wondered why in the syntax definition, copatterns are defined first, even though they need destructors to function, which are defined only afterwards. The reason becomes clear when you reread the titles of the examples: Constructors are used to instantiate data, while pattern matching is used to take apart data. Copattern matching is used to instantiate codata, while destructors are used to take apart codata. The order of the syntax definition emphasizes these conceptual dualitites: Data is dual to codata, but this is only because constructors are dual to copattern matching, and pattern matching is dual to destructors. Here is one more clean example to drive this point home:

Example 2.8 (Dualities in a tuple defined in two ways)**.**

	data	codata
instantiate	$\mathtt{Tup}(1,2)$	$\textbf{cocase}\ \{\mathtt{fst} \Rightarrow 1, \mathtt{snd} \Rightarrow 2\}$
take apart	$.\textbf{case}\ \{\mathtt{Tup}(x,y) \Rightarrow x\}$	$.\mathtt{fst}$
example	$\mathtt{Tup}(1,2).\textbf{case}\ \{\mathtt{Tup}(x,y) \Rightarrow x\}$	$\textbf{cocase}\ \{\mathtt{fst} \Rightarrow 1, \mathtt{snd} \Rightarrow 2\}.\mathtt{fst}$

2.4 Free Variables, Substitutions, Contexts

Definition 2.2 (Free Variables)**.** The set of free variables of a term e is $\mathrm{FV}(e)$. A term is closed if this set is empty. Free Variables are defined recursively over the structure of terms as follows:

$$\begin{aligned}
\mathrm{FV}(x) &:= \{x\} \\
\mathrm{FV}(K(e_1, ...e_n)) &:= \mathrm{FV}(e_1) \cup \cdots \cup \mathrm{FV}(e_n) \\
\mathrm{FV}(e.\textbf{case}\ \{\overline{K(\overline{x}) \Rightarrow t}\}) &:= \mathrm{FV}(e) \cup (\mathrm{FV}(t_1) \backslash (\overline{x})_1) \cup \cdots \cup (\mathrm{FV}(t_n) \backslash (\overline{x})_n) \\
\mathrm{FV}(\textbf{cocase}\ \{\overline{d(\overline{x}) \Rightarrow t}\}) &:= (\mathrm{FV}(t_1) \backslash (\overline{x})_1) \cup \cdots \cup (\mathrm{FV}(t_n) \backslash (\overline{x})_n) \\
\mathrm{FV}(e.d(e_1, ...e_n)) &:= \mathrm{FV}(e) \cup FV(e_1) \cup \cdots \cup \mathrm{FV}(e_n)
\end{aligned}$$

Definition 2.3 (Substitution)**.** A simultaneous substitution σ of the terms $e_1, ...e_n$ for the distinct variables $x_1, ...x_n$ is defined as follows:

$$\sigma ::= [e_1, ...e_n \backslash x_1, ...x_n]$$

Definition 2.4 (Domain and Range of a Substitution)**.** The definitions of Domain and Range of a Substitution are as follows:

$$\begin{aligned}\texttt{dom}([e_1, ...e_n/x_1, ...x_n]) &:= \{x_1, ...x_n\}\\ \texttt{rng}([e_1, ...e_n/x_1, ...x_n]) &:= \mathrm{FV}(e_1) \cup \cdots \cup \mathrm{FV}(e_n)\end{aligned}$$

The domain is the set of variables for which the substitution is defined, and the range is the set of free variables which appear in the substitution.

The interesting thing however, is what happens when we apply a substitution to an expression:

Definition 2.5 (Action of a Substitution)**.** The action of a substitution σ on a term e, written as $e\sigma$ and is defined as follows:

$$\begin{aligned}x[e_1, ...e_n/x_1, ...x_n] :=& e_i \quad (\text{if } x = x_i)\\ y\sigma :=& y \quad (\text{if } y \notin \texttt{dom}(\sigma))\\ (K(e_1, ...e_n))\sigma :=& K(e_1\sigma, \ldots, e_n\sigma)\\ (\textbf{case } \{\overline{K(\overline{x}) \Rightarrow e}\})\sigma :=& (e\sigma).\textbf{case}\{\overline{K(\overline{y}) \Rightarrow (e\sigma')\sigma}\}\\ (\textbf{cocase } \{\overline{d(\overline{x}) \Rightarrow e}\})\sigma :=& \textbf{cocase } \{\overline{d(\overline{y}) \Rightarrow (e\sigma')\sigma}\}\\ (e.d(e_1, ...e_n))\sigma :=& (e\sigma).d(e_1\sigma, \ldots, e_n\sigma)\end{aligned}$$

Where σ' is a substitution that ensures that we don't bind new variables: σ' has the form $[y_1, \ldots, y_n/x_1, ...x_n]$ and all y_i are fresh for both the domain and the range of σ.

Sometimes we want to attach multiple substitutions together. This process is called composition of substitutions and looks like this: $\sigma = \sigma_2 \circ \sigma_1$, which is equivalent to first applying the substitution σ_1, then the substitution σ_2.

Definition 2.6 (Composition of Subsitutions)**.** Given two substitutions

$$\sigma_1 := [e_1, ...e_n/x_1, ...x_n], \qquad \sigma_2 := [t_1, \ldots, t_m/y_1, \ldots, y_m],$$

composition is defined as:

$$\sigma_2 \circ \sigma_1 := [e_1\sigma_2, \ldots, e_n\sigma_2, t_j, \ldots, t_k/x_1, ...x_n, y_j, \ldots, y_k]$$

Where $j, \ldots, k$ ist the greatest sub-range of indices $1, \ldots, m$ such that none of the variables y_j to y_k is in the domain of σ_1.

Definition 2.7 (Idempotency)**.** A substitution σ is idempotent, iff. $\sigma \circ \sigma = \sigma$. Concretely, this means that it doesn't matter how often we apply a substitution to a given expression.

Consider these two examples. One of an idempotent substitution, and one of a substitution that is not idempotent:

Example 2.9 (Idempotency)**.** $[\mathbf{cocase}\ \{\mathtt{ap}(x) \Rightarrow x\}/y]$ is idempotent, since:

$$\begin{aligned} & [\mathbf{cocase}\ \{\mathtt{ap}(x) \Rightarrow x\}/y] \circ [\mathbf{cocase}\ \{\mathtt{ap}(x) \Rightarrow x\}/y] \\ =& [\mathbf{cocase}\ \{\mathtt{ap}(x) \Rightarrow x\}[\mathbf{cocase}\ \{\mathtt{ap}(x) \Rightarrow x\}/y]/y] \\ =& [\mathbf{cocase}\ \{\mathtt{ap}(x) \Rightarrow x\}/y] \end{aligned} \tag{1}$$

On the other hand, the substitution $[\mathbf{cocase}\ \{\mathtt{ap}(y) \Rightarrow x\}/x]$ is not idempotent, since:

$$\begin{aligned} & [\mathbf{cocase}\ \{\mathtt{ap}(y) \Rightarrow x\}/x] \circ [\mathbf{cocase}\ \{\mathtt{ap}(y) \Rightarrow x\}/x] \\ =& [\mathbf{cocase}\ \{\mathtt{ap}(y) \Rightarrow x\}[\mathbf{cocase}\ \{\mathtt{ap}(y) \Rightarrow x\}/x]/x] \\ =& \mathbf{cocase}\ \{\mathtt{ap}(y) \Rightarrow (\mathtt{ap}(y) \Rightarrow x)\} \neq [\mathbf{cocase}\ \{\mathtt{ap}(y) \Rightarrow x\}/x] \end{aligned} \tag{2}$$

Definition 2.8 (More general)**.** A substitution σ is more general than a substitution θ, iff. there exists a mapping τ, such that: $\theta = \tau \circ \sigma$.

For example, in the following unification problem, we are trying to substitute types for two unification variables:

Example 2.10 (More general)**.**

$$\begin{aligned} \mathsf{List}(\alpha^?) &\equiv \mathsf{List}(\beta^?) \\ \alpha^? &\equiv \mathsf{Int} \end{aligned}$$

One solution might be: $\theta = [\mathsf{Int}, \mathsf{Int}/\alpha^?, \beta^?]$, so substituting Int for both unification variables. The more general solution is $\sigma = [\mathsf{Int}, \alpha^?/\alpha^?, \beta^?]$, however. This is because there exists a mapping $\tau = [\mathsf{Int}/\alpha^?]$, such that:

$$\begin{aligned} \tau \circ \sigma &= [\mathsf{Int}/\alpha^?] \circ [\mathsf{Int}, \alpha^?/\alpha^?, \beta^?] \\ &= [\mathsf{Int}[\mathsf{Int}/\alpha^?], \alpha^?[\mathsf{Int}/\alpha^?]/\alpha^?, \beta^?] = [\mathsf{Int}, \mathsf{Int}/\alpha^?, \beta^?] = \theta \end{aligned}$$

2.5 Conversion

Definition 2.9 (Beta-conversion)**.** A single step of beta-conversion $e_1 \equiv^1_\beta e_2$ is defined as follows:

$$\mathbf{cocase}\ \{\ldots, d(\overline{x}) \Rightarrow e, \ldots\}.d(\overline{e}) \equiv^1_\beta e[\overline{e}/\overline{x}] \qquad (\beta\text{-codata})$$

$$K(\overline{e}).\mathbf{case}\{\ldots, K(\overline{x}) \Rightarrow e, \ldots\} \equiv^1_\beta e[\overline{e}/\overline{x}] \qquad (\beta\text{-data})$$

We require that the constructor $K(\overline{e})$ and the constructor $K(\overline{x})$ have the same number of arguments. This, in short ensures that we don't generate stuck terms (terms that can't be evaluated).

Consider these examples of beta-conversion:

Example 2.11 (Beta-conversion)**.**

$$\textbf{cocase}\ \{\texttt{ap}(x) \Rightarrow \texttt{True}\}.\texttt{ap}(x) \equiv^1_\beta \texttt{True}[x/x] = \texttt{True} \quad (1)$$

$$\textbf{cocase}\ \{\texttt{ap}(y) \Rightarrow y\}.\texttt{ap}(x) \equiv^1_\beta y[x/y] = x \quad (2)$$

$$\texttt{True}.\textbf{case}\ \{\texttt{False} \Rightarrow \texttt{True}, \texttt{True} \Rightarrow \texttt{False}\} \equiv^1_\beta \texttt{False} \quad (3)$$

$$\textbf{cocase}\ \{\texttt{fst} \Rightarrow \texttt{1}, \texttt{snd} \Rightarrow \texttt{2}\}.\texttt{fst} \equiv^1_\beta \texttt{1} \quad (4)$$

Intuitively, beta-conversion means not only function application but also the reduction under pattern or copattern matching.

Definition 2.10 (Eta-Conversion for codata)**.** A single step of eta-conversion $e_1 \equiv^1_\eta e_2$ is defined as follows:

$$\textbf{cocase}\ \{\overline{d(\overline{x}) \Rightarrow e.d(\overline{x})}\} \equiv^1_\eta e \quad (\text{if } \overline{x} \notin \mathrm{FV}(e)) \qquad (\eta\text{-codata})$$

The expression e needs to be the same in all the different clauses!

Now consider these examples of eta-conversion:

Example 2.12 (Eta-conversion for codata)**.**

$$\textbf{cocase}\ \{\texttt{ap}(y) \Rightarrow id.\texttt{ap}(y)\} \equiv^1_\eta id \quad (1)$$

$$\textbf{cocase}\ \{\texttt{fst} \Rightarrow specificPair.\texttt{fst}, \texttt{snd} \Rightarrow specificPair.\texttt{snd}\} \equiv^1_\eta specificPair \quad (2)$$

where $specificPair = \textbf{cocase}\ \{\texttt{fst} \Rightarrow \texttt{1}, \texttt{snd} \Rightarrow \texttt{2}\}$

The first example says: The function that takes a term and applies the identity function to it, does the same thing as the identity function. In the second example, we have a pair that contains the first value of a specific pair, and the second value of a specific pair, which is obviously just that specific pair.

3 Simple Types

We want the unification algorithm to work on typed terms. To start, let's look at some example terms and their types:

Example 3.1 (Basic types)**.**

$$\begin{aligned}
\texttt{False} &: \mathsf{Bool} \\
\mathtt{Tup}(\mathtt{1}, \mathtt{True}) &: \mathsf{Pair}(\mathsf{Int}, \mathsf{Bool}) \\
\textbf{cocase}\ \{\mathtt{fst} \Rightarrow \mathtt{1}, \mathtt{snd} \Rightarrow \mathtt{True}\} &: \mathsf{LPair}(\mathsf{Int}, \mathsf{Bool}) \\
\mathtt{Cons}(\mathtt{1}, \mathtt{Cons}(\mathtt{2}, \mathtt{Nil})) &: \mathsf{List}(\mathsf{Int}) \\
Trues = \textbf{cocase}\ \{\mathtt{hd} \Rightarrow \mathtt{True}, \mathtt{tl} \Rightarrow Trues\} &: \mathsf{Stream}(\mathsf{Bool}) \\
\textbf{cocase}\ \{\mathtt{ap}(x) \Rightarrow x.\textbf{case}\ \{\mathtt{True} \Rightarrow \mathtt{False}, \mathtt{False} \Rightarrow \mathtt{True}\}\} &: \mathsf{Bool} \to \mathsf{Bool}
\end{aligned}$$

Here are the the basic types we consider in our examples:

Definition 3.1 (Basic types)**.**

$$\begin{aligned}
\tau, \tau_1, \tau_2, ... ::= & \ \mathsf{Bool} \\
& \mid \mathsf{Pair}(\tau_1, \tau_2) \\
& \mid \mathsf{Lpair}(\tau_1, \tau_2) \\
& \mid \mathsf{List}(\tau) \\
& \mid \mathsf{Stream}(\tau) \\
& \mid \tau_1 \to \tau_2 \\
\Gamma ::= & \ \cdot \mid x_1 : \tau, \Gamma
\end{aligned}$$

Γ is the typing context, and contains distinct variables and their types. When we write $\Gamma = \cdot$, we mean that the typing context is empty.

These of course don't cover all the types one could construct with our calculus, but they are enough to construct helpful examples. I omitted the formal definition for natural numbers, since I use a shorthand anyways.

3.1 Typing Rules

To have a formal basis for how to assign the basic types to terms, we will introduce typing rules. When we write, $\Gamma \vdash t : \tau$, we mean that from the variables and their types in Γ, we can deduce that t has the type τ.

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}\ \textsc{Var} \qquad \frac{}{\Gamma \vdash \mathtt{True} : \mathsf{Bool}}\ \textsc{True} \qquad \frac{}{\Gamma \vdash \mathtt{False} : \mathsf{Bool}}\ \textsc{False}$$

$$\frac{\Gamma \vdash t_1 : \tau_1 \qquad \Gamma \vdash t_2 : \tau_2}{\Gamma \vdash \mathsf{Tup}(t_1, t_2) : \mathsf{Pair}(\tau_1, \tau_2)}\ \textsc{Tup}$$

$$\frac{\Gamma \vdash t : \mathsf{Pair}(\tau_1, \tau_2) \qquad \Gamma, x \vdash : \tau_1, y : \tau_2, t' : \tau'}{\Gamma \vdash t.\textbf{case}\ \{\mathsf{Tup}(t_1, t_2) \Rightarrow t'\} : \tau'}\ \textsc{Case-Pair}$$

$$\frac{}{\Gamma \vdash \texttt{Nil} : \mathsf{List}(\tau)}\ \textsc{Nil} \qquad \frac{\Gamma \vdash t_1 : \tau \qquad \Gamma \vdash t_2 : \mathsf{List}(\tau)}{\Gamma \vdash \texttt{Cons}(t_1, t_2) : \mathsf{List}(\tau)}\ \textsc{Cons}$$

$$\frac{\Gamma \vdash t : \mathsf{List}(\tau') \qquad \Gamma \vdash t_1 : \tau \qquad \Gamma, y : \tau', z : \mathsf{List}(\tau') \vdash t_2 : \tau}{\Gamma \vdash t.\mathbf{case}\ \{\texttt{Nil} \Rightarrow t_1, \texttt{Cons}(y, z) \Rightarrow t_2\} : \tau}\ \textsc{Case-List}$$

$$\frac{\Gamma \vdash t_1 : \tau \qquad \Gamma \vdash t_2 : \mathsf{Stream}(\tau)}{\Gamma \vdash \mathbf{cocase}\ \{\texttt{hd} \Rightarrow t_1, \texttt{tl} \Rightarrow t_2\} : \mathsf{Stream}(\tau)}\ \textsc{Stream}$$

$$\frac{\Gamma \vdash t : \mathsf{Stream}(\tau)}{\Gamma \vdash t.\texttt{hd} : \tau}\ \textsc{Hd} \qquad \frac{\Gamma \vdash t : \mathsf{Stream}(\tau)}{\Gamma \vdash t.\texttt{tl} : \mathsf{Stream}(\tau)}\ \textsc{Tl}$$

$$\frac{\Gamma \vdash t_1 : \tau_1 \qquad \Gamma \vdash t_2 : \tau_2}{\Gamma \vdash \mathbf{cocase}\ \{\texttt{fst} \Rightarrow t_1, \texttt{snd} \Rightarrow t_2\} : \mathsf{LPair}(\tau_1, \tau_2)}\ \textsc{Lpair}$$

$$\frac{\Gamma \vdash t : \mathsf{Lpair}(\tau_1, \tau_2)}{\Gamma \vdash t.\texttt{fst} : \tau_1}\ \textsc{Fst} \qquad \frac{\Gamma \vdash t : \mathsf{Lpair}(\tau_1, \tau_2)}{\Gamma \vdash t.\texttt{snd} : \tau_2}\ \textsc{Snd}$$

$$\frac{\Gamma \vdash t_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash t_2 : \tau_1}{\Gamma \vdash t_1.\mathsf{ap}(t_2) : \tau_2}\ \textsc{App}$$

$$\frac{\Gamma, x : \tau_1 \vdash t : \tau_2}{\Gamma \vdash \mathbf{cocase}\ \{\mathsf{ap}(x) \Rightarrow t\} : \tau_1 \to \tau_2}\ \textsc{Fun}$$

4 First-Order Unification

We want to slowly make our way to higher-order unification, and thus touch on simpler problems first. Furthermore, we need a couple more concepts to talk about unification problems and describe our algorithm.

A unification problem is described by a set of equations with expressions on each side $\overline{e \equiv e}$ containing unknown unification variables $\alpha_1^?, \alpha_2^?, \beta^?$..., where our goal is to find a simultaneous substitution $[e_1, ...e_n/\alpha_1^?, \ldots, \alpha_n^?]$ which substitutes expressions for unification variables, such that the sides of the given equations are the same.

Definition 4.1 (Solution)**.** A solution to a given unification problem is described by a simultaneous substitution $[e_1, ...e_n/\alpha_1^?, \ldots, \alpha_n^?]$ which when applied to the problem solves it, i.e. makes the sides of the equations equal.

Definition 4.2 (Most General)**.** A solution is the most general unifier (mgu), iff. it is more general (see definition 2.8) than all other solutions.

Now let's take a look at first-order unification. Even though first-order unification is a pretty limited subproblem of higher-order unification, in many applications it is all that is needed — the types of most (non-dependent) programming languages can be represented with first-order terms.

Consider some examples of first-order unification. On the left is the problem, on the right its solution or $\bot$ if there is no solution.

Example 4.1 (First-order unification)**.**

$$\begin{aligned} \alpha^? &\equiv \texttt{True} & \sigma_1 &= [\texttt{True}/\alpha^?] & (1)\\ \texttt{True} &\equiv \texttt{False} & \sigma_2 &= \bot & (2)\\ \mathsf{Int} \to \alpha^? &\equiv \mathsf{Int} \to \mathsf{Bool} & \sigma_3 &= [\mathsf{Bool}/\alpha^?] & (3)\\ \alpha^? &\equiv \mathsf{List}(\alpha^?) & \sigma_4 &= \bot & (4) \end{aligned}$$

The forth example is an instance of a failing occurs check.

Definition 4.3 (First-order unification)**.** A first-order unification problem consists of expressions from this restricted grammar:

$$\begin{aligned} e, r, s, t ::= {} & \alpha^?, \beta^? && \text{Unification variable}\\ \mid {} & x && \text{Variable}\\ \mid {} & K(\overline{e}) && \text{Constructor} \end{aligned}$$

Theorem 4.1 (Decidability of First-Order Unification)**.** For first-order unification, there exists an algorithm on equations $\overline{e \equiv e}$, which always terminates, and returns the solution if there exists one. In particular, this solution is always a mgu (i.e. if there is a solution, then there always exists a most general one).

Definition 4.4 (Unification algorithm for First-Order Unification)**.** $\bot$ is the symbol for fail. The algorithm is defined by non-deterministically applying the rules below:

$$\begin{aligned} & E \cup \{e \equiv e\} \Rightarrow E && \text{(delete)}\\ & E \cup \{K(e_1, ..., e_n) \equiv K(t_1, ..., t_n)\} \Rightarrow E \cup \{e_1 \equiv t_1, ..., e_n \equiv t_n\} && \text{(decompose)}\\ & E \cup \{K_1(e_1, ..., e_n) \equiv K_2(t_1, ..., t_m)\} \Rightarrow \bot \quad \text{if } K_1 \neq K_2 \text{ or if } n \neq m && \text{(conflict)}\\ & E \cup \{e \equiv \alpha^?\} \Rightarrow E \cup \{\alpha^? \equiv e\} && \text{(swap)}\\ & E \cup \{\alpha^? \equiv e\} \Rightarrow E[e/\alpha^?] \cup \{\alpha^? \equiv e\} \quad \text{if } \alpha^? \in E \text{ and } \alpha^? \notin e && \text{(eliminate)}\\ & E \cup \{\alpha^? \equiv e\} \Rightarrow \bot \quad \text{if } \alpha^? \in e && \text{(occurs check)} \end{aligned}$$

This algorithm is based on the version presented by Martelli and Montanari in [5], adapted to our syntax.

5 Higher-Order Unification

In this section, I introduce higher-order unification.

Definition 5.1 (Higher-Order Unification)**.** A higher-order unification problem consists of expressions from the following grammar:

$$
\begin{array}{rll}
e, r, s, t ::= \alpha^{?}\sigma & & \text{Unification variable with substitution} \\
\mid x & & \text{Variable} \\
\mid K(\overline{e}) & & \text{Constructor} \\
\mid e.\mathbf{case}\ \{\overline{K(\overline{x}) \Rightarrow t}\} & & \text{Pattern match} \\
\mid \mathbf{cocase}\ \{\overline{d(\overline{x}) \Rightarrow t}\} & & \text{Copattern match} \\
\mid e.d(\overline{e}) & & \text{Destructor}
\end{array}
$$

Note that this is encompassed by our syntax described in section 2.1, but with the addition of unification variables with substitutions. To illustrate the need for this substitution, look at what problem arises when we omit the substitution:

Example 5.1 (Substitutions on unification variables)**.** Consider this example:

$$\mathbf{cocase}\ \{\mathtt{ap}(x) \Rightarrow \alpha^{?}\}.\mathtt{ap}(y) \equiv \beta^{?}$$

If you focus on the left side, you might notice that there is a redex. What happens if we reduce it?

$$\mathbf{cocase}\ \{\mathtt{ap}(x) \Rightarrow \alpha^{?}\}.\mathtt{ap}(y) \equiv^{1}_{\beta} \alpha^{?}[y/x]$$

If we now found the solution $\sigma = [x/\alpha^{?}]$ through another equation, we would actually need to substitute y for x in $\alpha^{?} = x$!

This motivates our need for substitutions on unification variables: Since we don't know what the solution for a unification variable will be, we might need to perform a substitution on it later. Note that this is *not* possible in first-order unification, since we don't create substitutions through redexes! When the substitution is trivial, we may write $\alpha^{?}, \beta^{?}$ instead.

In higher-order unification in contrast to first-order unification, we are not interested in syntactic equality, but want a broader set of terms to be equal to one another. Depending on the type of unification problem one wants to solve, one may want to only include beta-equality or both beta- and eta-equality. This further motivates the use of the symbol $\equiv$ so far. Whereas in first-order unification it just stands for syntactic equality, in higher-order unification, I use it to mean syntactic equality, beta-equality or eta-equality. This essentially means that two terms are equivalent if they are equivalent after

function application, (co-)pattern matching evaluation, and/or are equivalent extensionally.

In the introduction in example 1.5, I alluded to the fact that many problems have multiple solutions, but we prefer a certain, most general solution. Let's next consider a familiar example again:

Example 5.2 (No most general solution)**.**

$$\alpha^?.\mathtt{ap}(5) \equiv \mathtt{5}$$

This problem has multiple solutions:

$$\sigma_1 = [\textbf{cocase}\ \{\mathtt{ap}(x) \Rightarrow x\}/\alpha^?] \qquad \text{identity function}$$
$$\sigma_2 = [\textbf{cocase}\ \{\mathtt{ap}(x) \Rightarrow 5\}/\alpha^?] \qquad \text{constant function, always returns 5}$$

In this case, neither solution is more general. The reason for this becomes apparent when you take a look at the action of a substitution again. Let's say we wanted to find a substitution to apply after the identity function to get the constant function. When we apply a substitution to a copattern match, we substitute new variables for our bound variables to not accidentally change the meaning of the term. This prevents us from changing our function. Since neither solution is more general (and there also is no other solution that is more general), there is no mgu for this problem!

Unification problems having no mgu is specific to higher-order unification, since in first-order unification we don't look at higher-order terms and thus don't have that problem of not changing variables.

Theorem 5.1 (Decidability of higher-order unification)**.** Higher-order unification is not decidable. This can be proven through reducing Hilbert's tenth problem to the unification problem.

5.1 Pattern Unification

Pattern unification, also sometimes called the pattern fragment is a subset of higher-order unification problems, and finding a solution is as simple as in to first-order unification. It was described first by Miller in [6]. Since our calculus ND amounts to an extension of the lambda calculus, we need to extend our definition to more than just function applications. This means that the pattern fragment described by Miller is a subset of our definition.

Definition 5.2 (Pattern)**.** A pattern is any term p

$$p ::= \alpha^? \mid p.d(x_1, ...x_n)$$

where it holds that all $x_1, ...x_n$ are distinct variables.

Consider some examples for patterns. On the left is the pattern unification problem and on the right its solution:

Example 5.3 (Pattern)**.**

$$\alpha^?.\texttt{ap}(x).\texttt{ap}(y) \equiv x \quad \sigma_1 = [\textbf{cocase}\ \{\texttt{ap}(x) \Rightarrow \textbf{cocase}\ \{\texttt{ap}(y) \Rightarrow x\}\}/\alpha^?] \quad (1)$$
$$\alpha^?.\texttt{fst} \equiv 2 \quad \sigma_2 = [\textbf{cocase}\ \{\texttt{fst} \Rightarrow 2, \texttt{snd} \Rightarrow \beta^?\}/\alpha^?] \quad (2)$$

Theorem 5.2 (Decidability of Pattern Unification)**.** Pattern Unification is decidable. If there exists a solution, there also exists a mgu.

Note that first-order unification is not a subset of pattern unification. Equations that contain first-order terms, but no patterns still remain solvable.

The reason there always exists an mgu lies in the constraint we put on our definition: All the variables must be distinct. To illustrate this, take the following problem where the variables are **not** distinct:

Example 5.4 (Why distinct variables in patterns?)**.**

$$\alpha^?.\texttt{ap}(x).\texttt{ap}(x) \equiv x$$

We can name two solutions:

$$\sigma_1 = [\textbf{cocase}\ \{\texttt{ap}(x) \Rightarrow \textbf{cocase}\ \{\texttt{ap}(y) \Rightarrow x\}\}/\alpha^?]$$
$$\sigma_2 = [\textbf{cocase}\ \{\texttt{ap}(x) \Rightarrow \textbf{cocase}\ \{\texttt{ap}(y) \Rightarrow y\}\}/\alpha^?]$$

(Intuitively, the solutions say to select the first or second argument of the function applications, respectively.) These solutions are equivalent, in that no solution is more general than the other. Thus, there exists no mgu for this problem. The solutions aren't unique because the variables aren't unique.

To talk about the algorithms for solving unification problems, we need another definition:

Definition 5.3 (Normal Form)**.** The normal form **NF** is defined as follows:

$$n ::= x \mid \alpha^? \mid n.d(\overline{v}) \mid n.\textbf{case}\{\overline{K(\overline{x}) \Rightarrow v}\}$$
$$v ::= n \mid K(\overline{v}) \mid \textbf{cocase}\ \{\overline{d(\overline{x}) \Rightarrow v}\}$$

Terms that satisfy the n-definition are called neutral terms, v-terms are called values.

Note that these are the terms that do not contain beta-redexes (a term that can be reduced through beta-conversion). This is apparent when one tries to construct terms that do beta-redexes in normal form: To construct the first kind of beta-redex: $K(\overline{e}).\textbf{case}\{\ldots, K(\overline{x}) \Rightarrow e\}$, we start with the

term $n.\mathbf{case}\{\overline{K(\overline{x}) \Rightarrow v}\}$, and now want to substitute $K(\overline{e})$ for n. This is not possible because we are limited to neutral terms, which constructors are not a part of. Similarly for the second kind of beta-redex: $\mathbf{cocase}\ \{\ldots, d(\overline{x}) \Rightarrow e, \ldots\}.d(\overline{e})$, here we can not substitute the cocase in $n.d(\overline{e})$ because we are limited to neutral terms.

This underlines the intuition that neutral terms are terms that cannot be reduced because they contain a term which blocks the evaluation (either a variable or a unification variable) in the front. Even though we don't have this assurance in values, that is not a problem since values don't contain the building blocks for beta-reductions. Some examples of terms in normal form:

Example 5.5 (Normal form)**.**

$$\alpha^?.\mathtt{ap}(\mathbf{cocase}\ \{\mathtt{ap}(x) \Rightarrow x\}) \tag{1}$$

$$\mathbf{cocase}\ \{\mathtt{ap}(x) \Rightarrow \mathtt{Cons}(x, \mathtt{Cons}(y, \mathtt{Nil}))\} \tag{2}$$

$$\alpha^?.\mathtt{ap}(x_1, x, y, z) \tag{3}$$

The first and second example are in normal form and the third is a pattern in normal form.

Theorem 5.3. A unification problem has a solution if and only if that solution is the same for the normal form of that problem.

This is helpful for us since we only have to consider normal forms in our algorithm. Concretely: We first reduce a given term it to its normal form, and then apply the steps to find our solution. Thus, from this point on, we will only be looking at terms that have a normal form. This is enough in most applications.

For a practical example for why normal forms are helpful, consider the following problem:

Example 5.6 (Beta-reductions before solving)**.**

$$\mathtt{True}.\mathbf{case}\ \{\mathtt{True} \Rightarrow \alpha^?, \mathtt{False} \Rightarrow 2\} \equiv 3 \tag{1}$$

has the solution $[3/\alpha^?]$, but this is more obvious after performing a beta-reduction on the left side to bring it into normal form:

$$\begin{aligned} &\mathtt{True}.\mathbf{case}\ \{\mathtt{True} \Rightarrow \alpha^?, \mathtt{False} \Rightarrow 2\} \equiv^1_\beta \alpha^? \\ &\implies \alpha^? \equiv 3 \end{aligned}$$

Let's take a look at another unification problem where beta-conversion is helpful:

$$\begin{aligned} &\alpha^? \equiv \mathtt{True} \\ &\alpha^?.\mathbf{case}\ \{\mathtt{True} \Rightarrow 2, \mathtt{False} \Rightarrow 3\} \equiv \beta^? \end{aligned} \tag{2}$$

To find out that $[2/\beta^?]$ is the solution to the second equation, we first need to find out that $[\texttt{True}/\alpha^?]$ is the solution to the first equation and substitute it in the second. This is what is called **dynamic pattern unification**, where we hold off on solving some equations until we have all the necessary information.

6 The Algorithm For Higher-Order Unification

In this section, I present the algorithm for higher-order unification. First, we need a framework to solve our algorithm in.

6.1 Constraints

Even though higher-order unification is undecidable, in many cases we can still find solutions, or conclude that there is no solution. So far, we have only described the problem of higher-order unification. Now we want to detail the steps necessary to solve higher-order unification problems. Thus, we need a framework for our equations — one where we can simplify our equations, and store what we have found out about our unknown variables. This is where constraints come into play:

Definition 6.1 (Constraint)**.**

$$C ::= \top \mid \bot \mid \Psi \vdash e \equiv t \qquad \text{where } \Psi = x_1, ..., x_n$$
$$\mathcal{C} ::= C \wedge \mathcal{C}$$

Where $\top$ means that an equation is trivially true, whereas $\bot$ is for contradictory equations. The variables in the context Ψ need to be distinct. Ψ may also be empty.

$\Psi \vdash e \equiv t$ means that given the variables Ψ, we deduce that $e \equiv t$. Ψ contains those variables we have seen before and want to remember. More precisely, $\Psi \vdash e \equiv t \Rightarrow FV(e) \subseteq \Psi \wedge FV(t) \subseteq \Psi$. This motivates the name context for Ψ, as well.

We formulate our constraints from a given unification problem as follows: We take each given equation and formulate a constraint K with empty context Ψ. We join them using ands: The equations $\overline{e \equiv t}$ become $\mathcal{C} = C_1 \wedge ... \wedge C_n = \vdash e_1 \equiv t_1 \wedge ... \wedge \vdash e_n \equiv t_n$

For clarification, $C \wedge \mathcal{C}$ is the regular logical "and", so therefore $\top \wedge \mathcal{C}$ is equivalent to $\mathcal{C}$ as well as $\bot \wedge \mathcal{C}$ to $\bot$. We can also apply substitutions to a set of constraints, which just means applying the substitution to the equation part of all the constraints in the set: $\mathcal{C}\sigma = (C \wedge \mathcal{C}')\sigma = C\sigma \wedge \mathcal{C}'\sigma = \Psi \vdash e\sigma \equiv t\sigma \wedge \mathcal{C}'\sigma$. Applying a substitution to $\top$ or $\bot$ changes nothing.

6.2 Decomposing Constraints

It is helpful to simplify our equations a bit before starting the unification process. We do this by taking out redundant equations, spotting clearly contradictory equations or splitting our equations into smaller parts which we can further simplify and unify.

Let's look at some examples. We would like the following to be true:

Example 6.1 (There are two kinds of variables!)**.**

$$x \vdash x.\texttt{fst} \equiv x.\texttt{fst} \qquad \mapsto \top \tag{1}$$
$$x \vdash x.\texttt{fst} \equiv x.\texttt{snd} \qquad \mapsto \bot \tag{2}$$
$$x, y \vdash x.\texttt{fst} \equiv y.\texttt{fst} \qquad \mapsto \bot \tag{3}$$

These examples might be confusing at first. Looking at the second example, one might argue that there exists a pair, say $x = \textbf{cocase}\ \{\texttt{fst} \Rightarrow 2, \texttt{snd} \Rightarrow 2\}$, such that $x.\texttt{fst} = x.\texttt{snd}$. This is where it is important to remember that we are not looking for solutions to variables like x. Whereas to our unification variables $\alpha^?$ we want to assign any term such that our equations hold, variables x add **constraints** to our equations. The equation $x \vdash x.\texttt{fst} \equiv x.\texttt{snd}$ must hold for **any** variable x, since we cannot choose x! Since for $x = \textbf{cocase}\ \{\texttt{fst} \Rightarrow 2, \texttt{snd} \Rightarrow 3\}$, the equation does not hold, we can simplify it to $\bot$.

Example 6.2 (Constructors and Destructors)**.**

$$\vdash \texttt{Cons}(x, \texttt{Nil}) \equiv \texttt{Cons}(y, \texttt{Nil}) \qquad \mapsto \bot \tag{1}$$
$$\vdash \texttt{Cons}(\alpha^?, \texttt{Nil}) \equiv \texttt{Cons}(\beta^?, \texttt{Nil}) \qquad \mapsto \vdash \alpha^? \equiv \beta^? \tag{2}$$
$$\vdash \mathsf{List}(\mathsf{Int}) \equiv \mathsf{List}(\alpha^?) \qquad \mapsto \vdash \mathsf{Int} \equiv \alpha^? \tag{3}$$
$$x \vdash x.\texttt{ap}(e_1) \equiv x.\texttt{ap}(e_2) \qquad \mapsto x \vdash e_1 \equiv e_2 \tag{4}$$

Example 6.3 (Pattern and Copattern Matching)**.**

$$x \vdash x.\textbf{case}\ \{\texttt{T} \Rightarrow e_1, \texttt{F} \Rightarrow e_2\} \equiv x.\textbf{case}\ \{\texttt{T} \Rightarrow t_1, \texttt{F} \Rightarrow t_2\}^* \tag{1}$$
$$\mapsto x \vdash e_1 \equiv t_1 \land x \vdash e_2 \equiv t_2$$
$$\vdash \textbf{cocase}\ \{\texttt{ap}(x) \Rightarrow \alpha^?\} \equiv \textbf{cocase}\ \{\texttt{ap}(x) \Rightarrow \beta^?\} \tag{2}$$
$$\mapsto x \vdash \alpha^? \equiv \beta^?$$
$$\textbf{cocase}\ \{\texttt{fst} \Rightarrow 5, \texttt{snd} \Rightarrow \texttt{False}\} \equiv \alpha^? \tag{3}$$
$$\mapsto \vdash 5 \equiv \alpha^?.\texttt{fst} \land \vdash \texttt{False} \equiv \alpha^?.\texttt{snd}$$

*T and F are abbreviations for True and False.

Note that all of these examples are in normal form, i.e. they don't contain redexes. This means the only way for the terms on each side of the equations to be equivalent, is for them to be equivalent **syntactically**.

We want to formulate rules which help us simplify equations like these — equations without beta-redexes. We want remove redundant constraints, split constraints made up of composite terms to find solutions to their parts, and spot redundancies to stop the solving process as early as possible

Definition 6.2 (Decomposing Constraints)**.**

Removing redundancy

$\Psi \vdash x \equiv x \quad \mapsto_r \top$ (1)

$\Psi \vdash \alpha^? \equiv \alpha^? \quad \mapsto_r \top$ (2)

Decomposition

$\circ\ \Psi \vdash K(\overline{e}) \equiv K(\overline{t}) \quad \mapsto_d \overline{\Psi \vdash e \equiv t}$ (3)

$\circ\ \Psi \vdash e_1.d(\overline{e}) \equiv e_2.d(\overline{t})$ (4)

$\quad \mapsto_d \Psi \vdash e_1 \equiv e_2 \wedge \overline{\Psi \vdash e \equiv t}$

$*\ \Psi \vdash e_1.\mathbf{case}\ \{\overline{K(\overline{x}) \Rightarrow e}\} \equiv e_2.\mathbf{case}\ \{\overline{K(\overline{x}) \Rightarrow t}\}$ (5)

$\quad \mapsto_d \Psi \vdash e_1 \equiv e_2 \wedge \overline{\Psi, \overline{x} \vdash e \equiv t}$

$*\ \Psi \vdash \mathbf{cocase}\ \{\overline{d(\overline{x}) \Rightarrow e}\} \equiv \mathbf{cocase}\ \{\overline{d(\overline{x}) \Rightarrow t}\} \quad \mapsto_d \overline{\Psi, \overline{x} \vdash e \equiv t}$ (6)

$\Psi \vdash \mathbf{cocase}\ \{\overline{d(\overline{x}) \Rightarrow e}\} \equiv t \quad \mapsto_d \overline{\Psi, \overline{x} \vdash e \equiv t.d(\overline{x})}$ (7)

$\Psi \vdash t \equiv \mathbf{cocase}\ \{\overline{d(\overline{x}) \Rightarrow e}\} \quad \mapsto_d \overline{\Psi, \overline{x} \vdash t.d(\overline{x}) \equiv e}$ (8)

Eta-reduction

$\Psi \vdash \mathbf{cocase}\ \{\overline{d(\overline{x}) \Rightarrow e.d(\overline{x})}\} \equiv t \quad \mapsto_e \Psi \vdash e \equiv t$ (9)

Removing contradictions

$\Psi \vdash x \equiv y \quad \mapsto_c \bot$ (10)

non-matching con-/destructors or non-matching variables in $\circ$-equation $\quad \mapsto_c \bot$

non-matching con-/destructors in *-equation $\quad \mapsto_c \bot$ (12)

Rules 6-8 are taken from [1], adapted to our syntax.

For equations marked with *, we require the constructors (or destructors) to be equal to one another in each equation. We also require them to have the same list of variables as arguments, respectively (i.e. full syntactic equality). For equations marked with $\circ$, we only require the constructors (or destructors) to be equal to one another in each equation. (i.e. no syntactic equality among arguments required). This is because in *-equations, we expect variables,

whereas in ∘-arguments, we expect expressions (or **values** to be exact — since we expect terms in normal form).

In *-equations where the arguments are not equal syntactically, rule eq. (12) applies and we simplify to ⊥:

Example 6.4 (Contradiction in (co-)pattern matching)**.**

$$\vdash x.\mathbf{case}\ \{\mathtt{True} \Rightarrow \alpha^?, \mathtt{False} \Rightarrow \beta^?\} \equiv x.\mathbf{case}\ \{\mathtt{1} \Rightarrow \alpha^?, \mathtt{2} \Rightarrow \beta^?\} \quad \mapsto_c \bot \quad (1)$$

$$\vdash \mathbf{cocase}\ \{\mathtt{fst} \Rightarrow \mathtt{1}, \mathtt{snd} \Rightarrow \mathtt{2}\} \equiv \mathbf{cocase}\ \{\mathtt{ap}(x) \Rightarrow \mathtt{1}\} \quad \mapsto_c \bot \quad (2)$$

Also note that eq. (7) (as well as eq. (8) wich is just the mirror of eq. (7)) are possible through eta-conversion:

$$\begin{aligned} &\mathbf{cocase}\ \{\overline{d(\overline{x}) \Rightarrow e}\} \equiv t \\ &\mathbf{cocase}\ \{\overline{d(\overline{x}) \Rightarrow e}\} \equiv \mathbf{cocase}\ \{\overline{d(\overline{x}) \Rightarrow t.d(\overline{x})}\} \\ &\qquad\qquad \overset{(eq.\ (6))}{\longmapsto_d} \overline{e \equiv t.d(\overline{x})} \end{aligned}$$

6.3 Unification

What do we do when we can't simplify anymore? How do we actually find the solutions? Let's start with an example — take a look at the following unification problem:

Example 6.5 (Unification)**.**

$$\begin{aligned} \mathtt{Tup}(\alpha^?, \mathtt{False}) &\equiv \mathtt{Tup}(\mathtt{True}, \mathtt{False}) \\ \beta^? &\equiv \alpha^?.\mathbf{case}\ \{\mathtt{True} \Rightarrow \mathtt{1}, \mathtt{False} \Rightarrow \mathtt{5}\} \end{aligned}$$

I will demonstrate how we want to solve this, and in the process introduces what to do when we found a solution.

$$\begin{aligned} \mathcal{C} = \quad & \vdash \mathtt{Tup}(\alpha^?, \mathtt{F}) \equiv \mathtt{Tup}(\mathtt{T}, \mathtt{F}) && \wedge \vdash \beta^? \equiv \alpha^?.\mathbf{case}\ \{\mathtt{T} \Rightarrow \mathtt{1}, \mathtt{F} \Rightarrow \mathtt{5}\}^* \\ \overset{(eq.\ (3))}{\longmapsto_d} \quad & \vdash \alpha^? \equiv \mathtt{T} && \wedge \vdash \mathtt{F} \equiv \mathtt{F} \quad \wedge \vdash \beta^? \equiv \alpha^?.\mathbf{case}\ \{\mathtt{T} \Rightarrow \mathtt{1}, \mathtt{F} \Rightarrow \mathtt{5}\} \\ \overset{(eq.\ (3))}{\longmapsto_d} \quad & \vdash \alpha^? \equiv \mathtt{T} && \wedge \vdash \beta^? \equiv \alpha^?.\mathbf{case}\ \{\mathtt{T} \Rightarrow \mathtt{1}, \mathtt{F} \Rightarrow \mathtt{5}\} \\ \mapsto_u \quad & \vdash \alpha^? \equiv \mathtt{T} && \wedge \vdash \beta^? \equiv \mathtt{T}.\mathbf{case}\ \{\mathtt{T} \Rightarrow \mathtt{1}, \mathtt{F} \Rightarrow \mathtt{5}\} \\ \overset{\equiv^1_\beta}{=} \quad & \vdash \alpha^? \equiv \mathtt{T} && \wedge \vdash \beta^? \equiv \mathtt{1} \end{aligned}$$

What do we learn about unification steps from this example? When we have found an assignment to a unification variable, we want to keep it in our constraints! This is because we sometimes have to substitute that assignment in other constraints that contain the same unification variable. But this example tells us something else, too: Doing so might introduce redexes! This means that we might need to reduce terms to normal form again, even after applying unification steps.

There are very few unification steps:

Definition 6.3 (Unification)**.**

$$
\begin{array}{lll}
e \equiv \alpha^? & \mapsto_u \alpha^? \equiv e & (e \neq \beta^?) \quad (1) \\
\mathcal{C} \wedge \Psi \vdash \alpha^? \equiv e & \mapsto_u \mathcal{C}[e/\alpha^?] \wedge \Psi \vdash \alpha^? \equiv e & (\alpha^? \notin e) \quad (2) \\
\mathcal{C} \wedge \Psi \vdash \alpha^? \equiv e & \mapsto_u \bot & (\alpha^? \in e) \quad (3)
\end{array}
$$

Step 2 and 3 are the occurs check, which ensures that we don't infinitely substitute terms for unification variables.

We should be careful, because there is another thing we could introduce with unification steps:

Example 6.6 (Introducing simplifiable constraints through unficiation steps)**.**

$$
\begin{array}{lllll}
\mathcal{C} = & \vdash \textbf{cocase}\ \{\mathtt{ap}(x) \Rightarrow \alpha^?\} \equiv id & \wedge & \vdash \beta^?.\mathtt{ap}(x) \equiv \alpha^? \\
\overset{(7)}{\longmapsto}_d & x \vdash \alpha^? \equiv id.\mathtt{ap}(x) & \wedge & \vdash \beta^?.\mathtt{ap}(x) \equiv \alpha^? \\
\overset{(2)}{\longmapsto}_u & x \vdash \alpha^? \equiv id.\mathtt{ap}(x) & \wedge & \vdash \beta^?.\mathtt{ap}(x) \equiv id.\mathtt{ap}(x) \\
\overset{(4)}{\longmapsto}_d & x \vdash \alpha^? \equiv id.\mathtt{ap}(x) & \wedge & \vdash \beta^? \equiv id \wedge \vdash x \equiv x \\
\overset{(1)}{\longmapsto}_r & x \vdash \alpha^? \equiv id.\mathtt{ap}(x) & \wedge & \vdash \beta^? \equiv id
\end{array}
$$

Here, the unification step introduces another constraint we can simplify!

6.4 The algorithm

Having described all the steps of our algorithm, the given examples lead us to a question: How and when do we apply these steps?

Given a set of equations, we normalize the terms in them and are left with only normal forms. For each equation, we formulate a constraint and get the set of constraints through logical ands. Next, we decompose the constraints, removing redundancies and potentially terminating early since we find contradictions. We do this non-deterministically, since the order doesn't matter. This makes our algorithm dynamic, since we hold off on solving certain equations until we know more.

When we can't further simplify, we non-deterministically apply unification steps, beta-reduce our terms and apply decomposition steps. When we have found an assigned term for each unification variable, we can terminate and give the solution by formulating a substitution for each assigned term.

This algorithm works on any pattern unification problem, and in particular, any unification problem that can be reduced to a pattern unification problem.

7 Related works

In [4], the author presents codata types in the programming language ML

Codata in action [3] was an important motivating force to research codata more.

The calculus ND is taken from [2].

Martelli and Montanari presented the algorithm for first-order unification problems which is often used today. [5] Pattern unfication is first described in [6].

In [1], the authors describe a higher-order unification algorithm for dependent types and records.

8 Conclusion

In summary, I presented a higher-order unification algorithm solving pattern unification for languages with a strong data-codata duality. This is an important improvement on previously presented higher-order unification algorithms which don't take codata types apart from (dependent) function types into account. The higher-order unification algorithm presented is a step towards being able to use languages with strong dualities like the data-codata duality in practice, as it shows that type-checking and -verification is very much possible. I hope that this as well as the exploration of the syntax make others more inclined to consider and use codata.

References

[1] A. Abel and B. Pientka. Higher-order dynamic pattern unification for dependent types and records. In *Proceedings of the 10th International Conference on Typed Lambda Calculi and Applications*, TLCA'11, page 10–26, Berlin, Heidelberg, 2011. Springer-Verlag.

[2] D. Binder. *Programming with Symmetric Data and Codata Types*. Eberhard Karls Universität Tübingen, 2024.

[3] P. Downen, Z. Sullivan, Z. M. Ariola, and S. Peyton Jones. Codata in action. In L. Caires, editor, *Programming Languages and Systems*, pages 119–146, Cham, 2019. Springer International Publishing.

[4] T. Hagino. Codatatypes in ml. *Journal of Symbolic Computation*, 8(6):629–650, 1989.

[5] A. Martelli and U. Montanari. An efficient unification algorithm. *ACM Trans. Program. Lang. Syst.*, 4(2):258–282, Apr. 1982.

[6] D. Miller. A Logic Programming Language with Lambda-Abstraction, Function Variables, and Simple Unification. *Journal of Logic and Computation*, 1(4):497–536, 09 1991.

Selbständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Bachelorarbeit selbständig und nur mit den angegebenen Hilfsmitteln angefertigt habe und dass alle Stellen, die dem Wortlaut oder dem Sinne nach anderen Werken entnommen sind, durch Angaben von Quellen als Entlehnung kenntlich gemacht worden sind. Diese Bachelorarbeit wurde in gleicher oder ähnlicher Form in keinem anderen Studiengang als Prüfungsleistung vorgelegt.

Ort, Datum Unterschrift