

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

Write here your abstract.

## Acknowledgements

Write here your acknowledgements.

## Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Data and codata . . . . .	1
1.2	Higher-Order Unification . . . . .	1
<b>2</b>	<b>The Untyped Calculus ND</b>	<b>2</b>
2.1	Syntax of the Untyped Calculus ND . . . . .	3
2.2	How is this used? . . . . .	3
2.3	Dualities in data and codata . . . . .	5
2.4	Free Variables, Substitutions, Contexts . . . . .	6
2.5	Conversion . . . . .	8
<b>3</b>	<b>Simple Types</b>	<b>8</b>
3.1	Typing Rules . . . . .	9
<b>4</b>	<b>First-Order Unification</b>	<b>10</b>
<b>5</b>	<b>Higher-Order Unification</b>	<b>11</b>
<b>6</b>	<b>Pattern Unification</b>	<b>13</b>
<b>7</b>	<b>The Algorithm For Higher-Order Unification</b>	<b>16</b>
7.1	Decomposing Constraints . . . . .	16
7.2	Unification . . . . .	18
7.3	The algorithm . . . . .	18
<b>8</b>	<b>Literature review</b>	<b>19</b>

## List of Theorems

1.1	Example (Two ways to define the same tuple) . . . . .	1
1.2	Example (Streams) . . . . .	1
1.3	Example (First-order unification problems) . . . . .	2

1.4	Example (Higher-order unification problems)	2
2.1	Definition (Terms of the Calculus ND)	3
2.1	Example (Constructing Types with constructors)	3
2.2	Example (Instantiating data with constructors)	4
2.3	Example (Using pattern matching to take apart data)	4
2.4	Example (Using pattern matching to represent conditionals)	4
2.5	Example (Using copattern matching to instantiate codata)	4
2.6	Example (Using destructors to take apart codata)	5
2.7	Example (Using copattern matching for function definitions)	5
2.8	Example (Dualities in a tuple defined in two ways)	5
2.2	Definition (Free Variables)	6
2.3	Definition (Substitution)	6
2.4	Definition (Domain and Range of a Substitution)	6
2.5	Definition (Action of a Substitution)	6
2.6	Definition (Composition of Substitutions)	7
2.7	Definition (Idempotency)	7
2.8	Definition (More General)	7
2.9	Definition (Beta-Conversion)	8
2.10	Definition (Eta-Conversion for codata)	8
4.1	Definition (Solution)	10
4.2	Definition (Most General)	10
4.3	Definition (First-Order unification)	10
4.1	Theorem (Decidability of First-Order Unification)	10
4.4	Definition (Unification algorithm for First-Order Unification)	10
5.1	Definition (Higher-Order Unification)	11
5.1	Example	11
5.2	Example	12
5.2	Definition (Constraint)	12
5.1	Theorem (Decidability of Higher-Order Unification)	13
6.1	Definition (Pattern)	13
6.1	Theorem (Decidability of Pattern Unification)	14
6.1	Example	14
6.2	Definition (Normal Form)	14
6.2	Example	15
6.2	Theorem	15
7.1	Example (There are two kinds of variables!)	16
7.2	Example (Constructors and Destructors)	16

7.3	Example (Pattern and Copattern Matching) . . . . .	16
7.1	Definition (Decomposing Constraints) . . . . .	17
7.4	Example . . . . .	18
7.2	Definition (Unification) . . . . .	18





# 1 Introduction

I want you to understand roughly what all the terms in the title of, as well as the goal of this thesis in the next few paragraphs.

## 1.1 Data and codata

While data (as in algebraic data types) is a well-known concept, its brother codata is less talked about. Codata is dual to data, meaning any concept expressed using data may also be expressed through codata. On a high level: While data is concerned with how values are constructed or stored, codata is concerned with how values are destructured or used. For instance, look at these different ways to define the same tuple:

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

<code>Tup(1, 5)</code>	Data instantiation
<code>cocase{fst <math>\Rightarrow</math> 1, snd <math>\Rightarrow</math> 5}</code>	Codata instantiation

While the first instantiation 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).

$$Trues = \mathbf{cocase}\{\mathbf{fst} \Rightarrow \mathbf{True}, \mathbf{snd} \Rightarrow Trues\}$$

As you can see, we can also instantiate infinite codata. 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.

## 1.2 Higher-Order Unification

Most people probably first encounter unification problems in school, when solving simple math problems. It 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. For the type systems in most typed programming languages, first-order unification is enough. It is only when more complex types are introduced, like in the interactive theorem prover Coq or in the programming language Agda, higher-order unification is required.

**Example 1.3** (First-order unification problems).

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

Each of the given equations can be seen as their own unification problem. The first has an obvious solution  $[5/\alpha^?]$ , so substituting 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 second and third are both unification problems on types with 2 unification variables, and have many solutions.

**Example 1.4** (Higher-order unification problems). In this first example, we are looking for a function that when applied to 5, evaluates to 5:

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

We can think of two obvious solutions for this:  $[id/\alpha^?]$  and the constant function which always returns 5:  $[\mathbf{cocode}\{\mathbf{ap}(x) \Rightarrow 5\}/\alpha^?]$  As already implied before, sometimes our unification problems have more than one solution, but we are interested in a particular solution. This is the solution that is more general, the one from which the others can be generated. 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:

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

Here we have an example of pattern matching implementing an if-clause. If we were to substitute  $[\mathbf{True}/\alpha^?]$ , the right side would contract to 1, since the first case matches - solving the equation.

In this thesis, I will explore a calculus that has copattern matching and a focus on codata. As the goal is to present a higher-order pattern unification algorithm for this calculus, I go over some other smaller subsets of unification, and then explain all parts of the algorithm in more detail.

## 2 The Untyped Calculus ND

I will be introducing the Untyped Calculus ND, based on ...

## 2.1 Syntax of the Untyped Calculus ND

Some knowledge of notation is necessary to familiarize oneself with the syntax of the Untyped Calculus.  $X$  represents a (possibly empty) sequence  $X_1, \dots, X_i, \dots, X_n$ .

A pattern match  $e.\mathbf{case}\{\overline{K(\bar{x}) \Rightarrow e}\}$  matches a term  $e$  against a sequence of clauses, each clause consisting of a constructor and an expression. The expression associated with first constructor that matches the term is the result of the pattern match. For a copattern match  $\mathbf{cocase}\{\overline{d(\bar{x}) \Rightarrow e}\}$  the same rules apply, but instead of constructors, the term is matched against destructors.

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

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

In pattern and copattern matching, every con- or destructor may occur no more than once. This is so we don't match multiple clauses (since variables are interchangeable:  $\mathbf{Tup}(x_1, x_2)$  has the same meaning as  $\mathbf{Tup}(x, y)$ ).

## 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  $\mathbf{Int}, \mathbf{Bool}, \dots$  are constructors on an empty sequence, while there are of course Types that require arguments.

$\mathbf{Int}$   
 $\mathbf{Bool}$   
 $\mathbf{List}(\mathbf{Int})$   
 $\mathbf{Pair}(\mathbf{Int}, \mathbf{Bool})$

This is explored in detail and formally later on in ...

But we also use constructors for instantiating data types:

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

```
True, False
Tup(1, 2)
Date(27, 08, 2005)
Cons(True, Cons(False, Nil))
```

We can use constructors to define recursive data types, as you can see from the last example. Numbers are also implemented recursively, as Peano numbers. I use actual numbers as a shorthand throughout this thesis, though.

**Example 2.3** (Using pattern matching to take apart data).

```
Tup(1, 2).case{Tup(x, y) => x}
Cons(True, Cons(False, Nil)).case{Cons(x, y) => y}
```

This is the pattern matching you know and love. We will see later how this is evaluated to 1 and `Cons(False, Nil)`, respectively.

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

**Example 2.4** (Using pattern matching to represent conditionals).

```
t.case{True => e1, False => e2}           ≐ if t e1 else e2
e.case{Cons(True, x) => True, Nil => False, Cons(x, y) => False}
```

The second expression tests whether a given list starts with `True`.

**Example 2.5** (Using copattern matching to instantiate codata).

```
cocase{fst => 1, snd => 2}
TrafficLight = cocase{Green => Yellow, Yellow => Red, Red => Green}
Trues = cocase{fst => True, snd => Trues}
```

We have seen examples like this in the introduction; the first one is a tuple with the values 1 and 2.

In the second, we assume that there are destructors (on an empty sequence) for an algebraic data type that implements the colors of a traffic light. Notice again, that we could have defined our traffic light as what colors it displays (using a sum type), but chose to focus on how these colors change, what our traffic light *does*.

The third is the infinite stream of *Trues*.

**Example 2.6** (Using destructors to take apart codata).

```
cocase{fst  $\Rightarrow$  1, snd  $\Rightarrow$  2}.fst
TrafficLight.Green
id.ap(5)
```

These do exactly what you think they do: We select the first value of the tuple, and turn our traffic light Yellow, respectively. We will see how these examples resolve to 1, **Yellow** and 5 later on. `ap()` is the destructor that implements function application: It applies the function on which it is "called" to the argument in the brackets.

You might have noticed already 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 definition:

**Example 2.7** (Using copattern matching for function definitions).

```
id = cocase{ap(x)  $\Rightarrow$  x}
cocase{ap(x)  $\Rightarrow$  (ap(y)  $\Rightarrow$  x)}            $\hat{=}$   $\lambda x. \lambda y. x$ 
cocase{ap(x)  $\Rightarrow$  (ap(y)  $\Rightarrow$  y)}            $\hat{=}$   $\lambda x. \lambda y. y$ 
```

The second and third examples implement a function that takes two arguments, ignores one and gives 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 afterwards. The reason becomes clear when rereading the titles of the definitions: Constructors are used to instantiate data, pattern matching is used to take apart data. Copattern matching is used to instantiate codata, destructors are used to take apart codata. The order of the syntax definitions emphasizes these conceptual dualities. One more clean example to drive the point home:

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

	data	codata
instantiate	<code>Tup(1, 2)</code>	<code><b>cocase</b>{fst <math>\Rightarrow</math> 1, snd <math>\Rightarrow</math> 2}</code>
take apart	<code>.<b>case</b>{Tup(<i>x</i>, <i>y</i>) <math>\Rightarrow</math> <i>x</i>}</code>	<code>.fst</code>
example	<code>Tup(1, 2).<b>case</b>{Tup(<i>x</i>, <i>y</i>) <math>\Rightarrow</math> <i>x</i>}</code>	<code><b>cocase</b>{fst <math>\Rightarrow</math> 1, snd <math>\Rightarrow</math> 2}.fst</code>

## 2.4 Free Variables, Substitutions, Contexts

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

$$\begin{aligned}\text{FV}(x) &:= \{x\} \\ \text{FV}(K(e_1, \dots, e_n)) &:= \text{FV}(e_1) \cup \dots \cup \text{FV}(e_n) \\ \text{FV}(e.\text{case}\{\overline{K(\bar{x})} \Rightarrow e\}) &:= \text{FV}(e) \cup (\text{FV}(e_1) \setminus \bar{x}) \cup \dots \cup (\text{FV}(e_n) \setminus \bar{x}) \\ \text{FV}(\text{cocase}\{\overline{d(\bar{x})} \Rightarrow e\}) &:= (\text{FV}(e_1) \setminus \bar{x}) \cup \dots \cup (\text{FV}(e_n) \setminus \bar{x}) \\ \text{FV}(e.d(e_1, \dots, e_n)) &:= \text{FV}(e) \cup \text{FV}(e_1) \cup \dots \cup \text{FV}(e_n)\end{aligned}$$

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

$$\sigma ::= [e_1, \dots, e_n \setminus x_1, \dots, x_n]$$

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

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

$$\begin{aligned}\text{dom}([e_1, \dots, e_n \setminus x_1, \dots, x_n]) &:= \{x_1, \dots, x_n\} \\ \text{rng}([e_1, \dots, e_n \setminus x_1, \dots, x_n]) &:= \text{FV}(e_1) \cup \dots \cup \text{FV}(e_n)\end{aligned}$$

What is actually interesting is what happens when we apply a substitution to an expression

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

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

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

The composition of two substitutions  $\sigma_2 \circ \sigma_1$  which is equivalent to first applying the substitution  $\sigma_1$ , then the substitution  $\sigma_2$ .

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

$$\sigma_1 := [e_1, \dots, e_n / x_1, \dots, x_n], \quad \sigma_2 := [t_1, \dots, t_m / y_1, \dots, y_m]$$

Composition is defined as:

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

Where  $j, \dots, k$  is the greatest sub-range of indices  $1, \dots, m$  such that none of the variables  $y_j$  to  $y_k$  is in the domain of  $\sigma_1$

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

For example,  $[\mathbf{cocode}\{\mathbf{ap}(x) \Rightarrow x\}/y]$  is idempotent, since:

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

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

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

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

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

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

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

$$\tau \circ \sigma = [\text{Int}[\text{Int}/\alpha^?], \alpha^?[\text{Int}/\alpha^?]/\alpha^?, \beta^?] = [\text{Int}, \text{Int}/\alpha^?, \beta^?] = \theta$$

## 2.5 Conversion

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

$$\begin{aligned} K(\bar{e}).\mathbf{case}\{\dots, K(\bar{x}) \Rightarrow e, \dots\} &\equiv_\beta^1 e[\bar{e}/\bar{x}] & (\beta\text{-data}) \\ \mathbf{cocase}\{\dots, d(\bar{x}) \Rightarrow e, \dots\}.d(\bar{e}) &\equiv_\beta^1 e[\bar{e}/\bar{x}] & (\beta\text{-codata}) \end{aligned}$$

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

Some examples:

$$\begin{aligned} \mathbf{True}.\mathbf{case}\{\mathbf{False} \Rightarrow \mathbf{True}, \mathbf{True} \Rightarrow \mathbf{False}\} &\equiv_\beta^1 \mathbf{False} \\ \mathbf{cocase}\{\mathbf{ap}(x) \Rightarrow \mathbf{True}\}.\mathbf{ap}(x) &\equiv_\beta^1 \mathbf{True}[x/x] = \mathbf{True} \\ \mathbf{cocase}\{\mathbf{ap}(y) \Rightarrow y\}.\mathbf{ap}(x) &\equiv_\beta^1 y[x/y] = x \end{aligned}$$

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  $e_1 \equiv_\eta^1 e_2$  is defined as follows:

$$\mathbf{cocase}\{\overline{d(\bar{x}) \Rightarrow e.d(\bar{x})}\} \equiv_\eta^1 e \quad (\text{if } \bar{x} \notin \text{FV}(e)) \quad (\eta\text{-codata})$$

## 3 Simple Types

We want to be able to use the unification algorithm to discern types for given terms. To start, let's look at some examples terms and their types:

$$\begin{aligned} 5 &: \text{Int} \\ \mathbf{False} &: \text{Bool} \\ \mathbf{Tup}(1, \mathbf{True}) &: \text{Pair}(\text{Int}, \text{Bool}) \\ \mathbf{cocase}\{\mathbf{fst} \Rightarrow 1, \mathbf{snd} \Rightarrow \mathbf{True}\} &: \text{LPair}(\text{Int}, \text{Bool}) \\ \mathbf{Cons}(1, \mathbf{Cons}(2, \mathbf{Nil})) &: \text{List}(\text{Int}) \\ \mathbf{cocase}\{\mathbf{ap}(x) \Rightarrow x.\mathbf{case}\{\mathbf{True} \Rightarrow \mathbf{False}, \mathbf{False} \Rightarrow \mathbf{True}\}\} &: \text{Bool} \rightarrow \text{Bool} \\ \mathbf{Trues} = \mathbf{cocase}\{\mathbf{hd} \Rightarrow \mathbf{True}, \mathbf{tl} \Rightarrow \mathbf{Trues}\} &: \text{Stream}(\text{Bool}) \end{aligned}$$



### 3.1 Typing Rules

To be able to give our examples a formal basis, we will introduce typing rules. These will not cover all the types one could construct using our calculus, but instead all the basic composite types. This will be enough to construct examples covering each case we have to consider, and hopefully give an idea how to construct any type.

When writing  $\Gamma \vdash t : \tau$ , we mean that from the variables and their types in  $\Gamma$ , we can deduce that  $t$  has the type  $\tau$ .

$$\begin{array}{c}
\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \text{VAR} \quad \frac{}{\Gamma \vdash \text{True} : \text{Bool}} \text{TRUE} \quad \frac{}{\Gamma \vdash \text{False} : \text{Bool}} \text{FALSE} \\
\\
\frac{\Gamma \vdash t_1 : \tau_1 \quad \Gamma \vdash t_2 : \tau_2}{\Gamma \vdash \text{Tup}(t_1, t_2) : \text{Pair}(\tau_1, \tau_2)} \text{TUP} \\
\\
\frac{\Gamma \vdash t : \text{Pair}(t_1, t_2) \quad \Gamma, x \vdash : \tau_1, y : \tau_2, t' : \tau'}{\Gamma \vdash t.\text{case}\{\text{Tup}(t_1, t_2) \Rightarrow t'\} : \tau'} \text{CASE-PAIR} \\
\\
\frac{}{\Gamma \vdash \text{Nil} : \text{List}(\tau)} \text{NIL} \quad \frac{\Gamma \vdash t_1 : \tau \quad \Gamma \vdash t_2 : \text{List}(\tau)}{\Gamma \vdash \text{Cons}(t_1, t_2) : \text{List}(\tau)} \text{CONS} \\
\\
\frac{\Gamma \vdash t : \text{List}(\tau') \quad \Gamma \vdash t_1 : \tau \quad \Gamma, y : \tau', z : \text{List}(\tau') \vdash t_2 : \tau}{\Gamma \vdash t.\text{case}\{\text{Nil} \Rightarrow t_1, \text{Cons}(y, z) \Rightarrow t_2\} : \tau} \text{CASE-LIST} \\
\\
\frac{\Gamma \vdash t_1 : \tau \quad \Gamma \vdash t_2 : \text{Stream}(\tau)}{\Gamma \vdash \text{ccase}\{\text{hd} \Rightarrow t_1, \text{tl} \Rightarrow t_2\} : \text{Stream}(\tau)} \text{STREAM} \\
\\
\frac{\Gamma \vdash t : \text{Stream}(\tau)}{\Gamma \vdash t.\text{hd} : \tau} \text{HD} \quad \frac{\Gamma \vdash t : \text{Stream}(\tau)}{\Gamma \vdash t.\text{tl} : \text{Stream}(\tau)} \text{TL} \\
\\
\frac{\Gamma \vdash t_1 : \tau_1 \quad \Gamma \vdash t_2 : \tau_2}{\Gamma \vdash \text{ccase}\{\text{fst} \Rightarrow t_1, \text{snd} \Rightarrow t_2\} : \text{LPair}(\tau_1, \tau_2)} \text{LPAIR} \\
\\
\frac{\Gamma \vdash t : \text{Lpair}(\tau_1, \tau_2)}{\Gamma \vdash t.\text{fst} : \tau_1} \text{FST} \quad \frac{\Gamma \vdash t : \text{Lpair}(\tau_1, \tau_2)}{\Gamma \vdash t.\text{snd} : \tau_2} \text{SND} \\
\\
\frac{\Gamma \vdash t_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash t_2 : \tau_1}{\Gamma \vdash t_1.\text{ap}(t_2) : \tau_2} \text{APP} \\
\\
\frac{\Gamma, x : \tau_1 \vdash t : \tau_2}{\Gamma \vdash \text{ccase}\{\text{ap}(x) \Rightarrow t\} : \tau_1 \rightarrow \tau_2} \text{FUN}
\end{array}$$

## 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 concepts more to be able to talk about unification problems and describe our algorithm.

The Unification Problem is described by a set of equations with expressions on each side  $\bar{e} \equiv \bar{e}$  containing unknown unification variables  $\alpha_1^?, \alpha_2^?, \dots$ , where our goal is to find a simultaneous substitution  $[e_1, \dots, e_n / \alpha_1^?, \dots, \alpha_n^?]$  which substitutes expressions for unification variables, such that the sides of the given equations are the same (respectively).

**Definition 4.1** (Solution). A solution to a given unification problem is described by a simultaneous substitution  $[e_1, \dots, e_n / \alpha_1^?, \dots, \alpha_n^?]$  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 than all other solutions.

Now let's take a look at first-order unification, which is a pretty limited subproblem, but in many applications all we need, as in most type checking.

**Definition 4.3** (First-Order unification).

$$\begin{array}{ll}
 e, r, s, t ::= \alpha^?, \beta^? & \text{Unification variable} \\
 \quad \quad \quad | x & \text{Variable} \\
 \quad \quad \quad | K(\bar{e}) & \text{Constructor}
 \end{array}$$

Some examples:  $\alpha^? \equiv \mathbf{True}$  has the solution  $[\mathbf{True} / \alpha^?]$ .  $\mathbf{True} \equiv \mathbf{False}$  doesn't have a solution.

$\mathbf{Int} \rightarrow \alpha^? \equiv \mathbf{Int} \rightarrow \mathbf{Bool}$  has the solution  $[\mathbf{Bool} / \alpha^?]$ . On the other hand,  $\alpha^? \equiv \mathbf{List}(\alpha^?)$  should not have a solution. This is the negative example for the occur check rule below.

**Theorem 4.1** (Decidability of First-Order Unification). For first-order unification, there exists an algorithm on equations  $\bar{e} \equiv \bar{e}$ , 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).  $\perp$  is the symbol for fail. The algorithm is defined by non-deterministically applying the

below rules:

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

This algorithm is based on the version presented by Martelli and Montanari in [3], adapted to our syntax.

## 5 Higher-Order Unification

**Definition 5.1** (Higher-Order Unification).

$$\begin{array}{ll}
e, r, s, t ::= \alpha^? \sigma, \beta^? \sigma & \text{Unification Variables with substitution} \\
| x & \text{Variable} \\
| K(\bar{e}) & \text{Constructor} \\
| e.\mathbf{case}\{\overline{K(\bar{x}) \Rightarrow e}\} & \text{Pattern match} \\
| \mathbf{cocode}\{\overline{d(\bar{x}) \Rightarrow e}\} & \text{Cocode match} \\
| e.d(\bar{e}) & \text{Destructor}
\end{array}$$

Note that this is encompassed our syntax described in Section 2, but with the addition of unification variables with substitutions. To illustrate the need for this substitution, look at what problem arises when omitting the substitution: Take the following example:  $\mathbf{cocode}\{\mathbf{ap}(x) \Rightarrow \alpha^?\}. \mathbf{ap}(y) \equiv \beta^?$ . First focus on the left side and notice that there is a redex. What happens if we reduce it?

**Example 5.1.**

$$\mathbf{cocode}\{\mathbf{ap}(x) \Rightarrow \alpha^?\}. \mathbf{ap}(y) \equiv_{\beta}^1 \alpha^?[y/x]$$

If we find the following solution  $[x/\alpha^?]$  through another equation, we would actually need to substitute in  $\alpha^?$ !

This motivates our need for substitutions on unification variables: Since we don't know what the unification variable will solve to, 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 in 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, they 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 syntactically 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 externally.

Let's next consider an example again:

**Example 5.2.**

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

This problem has multiple solutions:

$$\begin{array}{ll} \sigma_1 = [\mathbf{cocode}\{\mathbf{ap}(x) \Rightarrow x\}/\alpha^?] & \text{identity function} \\ \sigma_2 = [\mathbf{cocode}\{\mathbf{ap}(x) \Rightarrow 5\}/\alpha^?] & \text{constant function, always returns 5} \end{array}$$

In this case, neither solution is more general. Why 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 to the identity function to get the constant function. When applying 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! This 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.

To be able to discuss unification equations in more depth, we need one more concept:

**Definition 5.2** (Constraint).

$$\begin{array}{ll} C ::= \top \mid \perp \mid \Psi \vdash e \equiv t & \text{where } \Psi = x_1, \dots, x_n \\ \mathcal{C} ::= C \wedge \mathcal{C} \end{array}$$

The variables in the Context  $\Psi$  need to be distinct.  $\Psi$  may also be empty.

$\Psi \vdash e \equiv t$  means that given the variables  $\Psi$ , we deduce that  $e \equiv t$ .  $\Psi$  may contain variables that occur in or are even bound in  $e$  or  $t$ .  $\Psi$  contains

those variables we have seen before and want to remember. This motivates the naming of  $\Psi$  as context, as well.

Why do we need a formal definition here? We don't solve our unification problem in one huge step, but work on it incrementally, solving parts of terms and remembering the information we have learned from these parts. We want to be able to talk about these processes formally.

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

For clarification,  $C \wedge \mathcal{C}$  is the regular logical and thus,  $\top \wedge \mathcal{C} = \mathcal{C}$  as well as  $\perp \wedge \mathcal{C} = \perp$

We can also apply substitutions on a set of constraints, which just means applying the substitution to the equation part of all the constraints in the set:  $\mathcal{C}\sigma = (C \wedge \mathcal{C}')\sigma = C\sigma \wedge \mathcal{C}'\sigma = \Psi \vdash e\sigma \equiv t\sigma \wedge \mathcal{C}'$ . Applying a substitution to  $\top$  or  $\perp$  changes nothing.

**Theorem 5.1** (Decidability of Higher-Order Unification). Higher-order unification includes unification problems containing higher-order terms (equivalent to lambda abstractions), and is covered by our introduced syntax. Higher-order unification is not decidable. This can be proven through reducing Hilbert's tenth problem to the unification problem.

## 6 Pattern Unification

Pattern unification, also sometimes called the pattern fragment is a subsection of Higher-Order Unification, with its solution being similarly simple as the one to first-order unification. It was described first by Miller in [4]. Since we are dealing with what amounts to an extension to the lambda calculus (as our syntax also contains constructors), we need to extend our definition to more than just function applications. This means that (in practice,) the pattern fragment described by Miller is a subset of our definition.

**Definition 6.1** (Pattern). A pattern is any term  $p$

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

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

If you are familiar with other definitions of patterns, you might have noticed that we have omitted the part about the variables being bound. We require this later on when looking at patterns as subterms. Some examples:  $\alpha^?.\mathbf{ap}(x).\mathbf{ap}(y) \equiv x$  is a pattern and has the solution  $[\mathbf{cocode}\{\mathbf{ap}(x) \Rightarrow$

$\mathbf{cocase}\{\mathbf{ap}(y) \Rightarrow x\}/\alpha^?$ .  $\alpha^?.fst \equiv 2$  is a pattern and has the solution  $[pair(2, x)/\alpha^?]$ .

**Theorem 6.1** (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 containing first-order terms that don't contain 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 6.1.**

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

We can name two solutions:

$$\begin{aligned}\sigma_1 &= [\mathbf{cocase}\{\mathbf{ap}(x) \Rightarrow \mathbf{cocase}\{\mathbf{ap}(y) \Rightarrow x\}\}/\alpha^?] \\ \sigma_2 &= [\mathbf{cocase}\{\mathbf{ap}(x) \Rightarrow \mathbf{cocase}\{\mathbf{ap}(y) \Rightarrow y\}\}/\alpha^?]\end{aligned}$$

(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. There exists no mgu for this problem. The solutions aren't unique because the variables aren't unique.

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

**Definition 6.2** (Normal Form). The normal form **NF** is defined as follows:

$$\begin{aligned}n &::= x \mid \alpha^? \mid n.d(\bar{v}) \mid n.\mathbf{case}\{\overline{K(\bar{x}) \Rightarrow v}\} \\ v &::= n \mid K(\bar{v}) \mid \mathbf{cocase}\{\overline{d(\bar{x}) \Rightarrow v}\}\end{aligned}$$

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 beta-redex is a term that can be transformed using a beta-conversion.

This is apparent when one tries to construct terms containing beta-redexes in normal form: To construct the first kind of beta-redex:  $K(\bar{e}).\mathbf{case}\{\dots, K(\bar{x}) \Rightarrow e\}$ , we start with the term  $n.\mathbf{case}\{\overline{K(\bar{x}) \Rightarrow v}\}$ , now wanting to substitute  $K(\bar{e})$  for  $n$ . This is not possible because we are limited

to neutral terms, which constructors are not part of. Similarly with the second kind of beta-redex:  $\mathbf{cocase}\{\dots, d(\bar{x}) \Rightarrow e, \dots\}.d(\bar{e})$ , where we could not substitute the cocase in  $n.d(\bar{e})$  because we are limited to neutral terms.

This underlines the intuition that neutral terms are terms that cannot be reduced because they contain an term we can't further evaluate (either a variable or a unification variable) in the front. Values, where we don't have this assurance cannot contain the building blocks for a beta-redex.

Some examples:

**Example 6.2.**

$$\begin{aligned} r &= \alpha^?.\mathbf{ap}(\mathbf{cocase}\{\mathbf{ap}(x) \Rightarrow x\}), \\ s &= \mathbf{cocase}\{\mathbf{ap}(x) \Rightarrow \mathbf{Cons}(x, \mathbf{Cons}(y, \mathbf{Nil}))\}, \\ t &= \alpha^?.\mathbf{ap}(x_1, x, y, z) \end{aligned}$$

$r$  and  $s$  are in normal form and  $t$  is a pattern in normal form.

**Theorem 6.2.** If a unification problem has a solution, that solution is the same for the normal form of that same problem. No normal form of a problem that doesn't have a solution has a solution either.

From this point on, we will only be looking at terms that have a normal form. This is enough in most applications.

This is helpful for us, since we only have to consider normal forms in our algorithm, as in: Met with a term, we first reduce it to its normal form, and then solve for our solution.

As an example, the problem  $\mathbf{True.case}\{\mathbf{True} \Rightarrow \alpha^?, \mathbf{False} \Rightarrow 2\} \equiv 3$  has the solution  $[3/\alpha^?]$ , but this is more obvious after using a beta-reduction on the left-hand side bringing it into normal form:

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

Let's take a look at another example:

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

To find out that  $[2/\beta^?]$  is the solution to the second equation, we first need to find out that  $[\mathbf{True}/\alpha^?]$  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 we don't have all the information for yet until we know more.

## 7 The Algorithm For Higher-Order Unification

### 7.1 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 7.1** (There are two kinds of variables!).

$$\begin{array}{ll} x \vdash x.\mathbf{fst} \equiv x.\mathbf{fst} & \mapsto \top \\ x \vdash x.\mathbf{fst} \equiv x.\mathbf{snd} & \mapsto \perp \\ x, y \vdash x.\mathbf{fst} \equiv y.\mathbf{fst} & \mapsto \perp \end{array}$$

These examples might be confusing at first. Looking at the second example, one might argue that there exists a pair, say  $x = \mathbf{cocode}\{\mathbf{fst} \Rightarrow 2, \mathbf{snd} \Rightarrow 2\}$ , such that  $x.\mathbf{fst} = x.\mathbf{snd}$ . 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  $x \vdash x.\mathbf{fst} \equiv x.\mathbf{snd}$  must hold for *any* variable  $x$ , since we cannot choose  $x$ ! Since for  $x = \mathbf{cocode}\{\mathbf{fst} \Rightarrow 2, \mathbf{snd} \Rightarrow 3\}$ , the equation does not hold, we can simplify to  $\perp$ .

**Example 7.2** (Constructors and Destructors).

$$\begin{array}{ll} \vdash \mathbf{Cons}(x, \mathbf{Nil}) \equiv \mathbf{Cons}(y, \mathbf{Nil}) & \mapsto \perp \\ \vdash \mathbf{Cons}(\alpha^?, \mathbf{Nil}) \equiv \mathbf{Cons}(\beta^?, \mathbf{Nil}) & \mapsto \alpha^? \equiv \beta^? \\ \vdash \mathbf{List}(\mathbf{Int}) \equiv \mathbf{List}(\alpha^?) & \mapsto \vdash \mathbf{Int} \equiv \alpha^? \\ x \vdash x.\mathbf{ap}(e_1) \equiv x.\mathbf{ap}(e_2) & \mapsto x \vdash e_1 \equiv e_2 \end{array}$$

**Example 7.3** (Pattern and Copattern Matching).

$$\begin{array}{ll} \vdash \mathbf{cocode}\{\mathbf{ap}(x) \Rightarrow \alpha^?\} \equiv \mathbf{cocode}\{\mathbf{ap}(x) \Rightarrow \beta^?\} & \mapsto x \vdash \alpha^? \equiv \beta^? \\ x \vdash x.\mathbf{case}\{\mathbf{T} \Rightarrow e_2, \mathbf{F} \Rightarrow e_2\} \equiv x.\mathbf{case}\{\mathbf{T} \Rightarrow t_1, \mathbf{F} \Rightarrow t_2\}^* & \\ \mapsto x \vdash e_1 \equiv t_1 \wedge x \vdash e_2 \equiv t_2 & \\ \mathbf{cocode}\{\mathbf{fst} \Rightarrow 5, \mathbf{snd} \Rightarrow \mathbf{False}\} \equiv \alpha^? & \\ \mapsto \vdash 5 \equiv \alpha^?.\mathbf{fst} \wedge \vdash \mathbf{False} \equiv \alpha^?.\mathbf{snd} & \end{array}$$



Note that none of these examples contain redexes, meaning 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.

**Definition 7.1** (Decomposing Constraints).

$$\begin{array}{ll}
\Psi \vdash x \equiv x & \mapsto_r \top \quad (1) \\
\Psi \vdash \alpha^? \equiv \alpha^? & \mapsto_r \top \quad (2) \\
\circ \Psi \vdash e_1.d(\bar{e}) \equiv e_2.d(\bar{t}) & \\
\mapsto_d \Psi \vdash e_1 \equiv e_2 \wedge \overline{\Psi \vdash e \equiv t} & (3) \\
* \Psi \vdash e_1.\mathbf{case}\{\overline{K(\bar{x}) \Rightarrow e}\} \equiv e_2.\mathbf{case}\{\overline{K(\bar{x}) \Rightarrow t}\} & \\
\mapsto_d \Psi \vdash e_1 \equiv e_2 \wedge \overline{\Psi, \bar{x} \vdash e \equiv t} & (4) \\
\circ \Psi \vdash K(\bar{e}) \equiv K(\bar{t}) & \mapsto_d \overline{\Psi \vdash e \equiv t} \quad (5) \\
\Psi \vdash \mathbf{cocode}\{\overline{d(\bar{x}) \Rightarrow e.d(\bar{x})}\} \equiv t & \mapsto_e \Psi \vdash e \equiv t \quad (6) \\
* \Psi \vdash \mathbf{cocode}\{\overline{d(\bar{x}) \Rightarrow e}\} \equiv \mathbf{cocode}\{\overline{d(\bar{x}) \Rightarrow t}\} & \mapsto_d \overline{\Psi, \bar{x} \vdash e \equiv t} \quad (7) \\
\Psi \vdash \mathbf{cocode}\{\overline{d(\bar{x}) \Rightarrow e}\} \equiv t & \mapsto_d \overline{\Psi, \bar{x} \vdash e \equiv t.d(\bar{x})} \quad (8) \\
\Psi \vdash t \equiv \mathbf{cocode}\{\overline{d(\bar{x}) \Rightarrow e}\} & \mapsto_d \overline{\Psi, \bar{x} \vdash t.d(\bar{x}) \equiv e} \quad (9) \\
\text{non-matching } * \text{- or } \circ \text{-equation} & \mapsto_c \perp \quad (10)
\end{array}$$

Rules 7-9 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).

(The reason for this discrepancy lies in the fact that we are only expecting terms in normal form!) If any of this does not hold in a given equation, rule 9 applies and we simplify to  $\perp$ .

In the special case of rule 5 where the constructors have no arguments, e.g.  $\mathbf{True} \equiv \mathbf{True}$ , we can simplify to  $\top$ . Note that 8 (as well as 9 with being just the mirror of 8) are possible through eta-conversion:

$$\begin{array}{ll}
\mathbf{cocode}\{\overline{d(\bar{x}) \Rightarrow e}\} \equiv t & \equiv_\eta \\
\mathbf{cocode}\{\overline{d(\bar{x}) \Rightarrow e}\} \equiv \mathbf{cocode}\{\overline{d(\bar{x}) \Rightarrow t.d(\bar{x})}\} & 6 \\
\mapsto \overline{e \equiv t.d(\bar{x})} &
\end{array}$$

---

\*T and F are stand-ins for **True** and **False**.

(Annotations on the right are supposed to show what rule is being used to get to the statement below.)

## 7.2 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 7.4.**

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

I will demonstrate how we want to solve this, introducing what to do when we found a solution.

$$\begin{array}{ll} \mathcal{C} = & \vdash \text{Tup}(\alpha^?, \text{False}) \equiv \text{Tup}(\text{True}, \text{False}) \quad \wedge \vdash \beta^? \equiv \alpha^?.\text{case}\{\text{True} \Rightarrow 1, \text{False} \Rightarrow 5\} \\ \xrightarrow{(5)}_d & \vdash \alpha^? \equiv \text{True}, \vdash \text{False} \equiv \text{False} \quad \wedge \vdash \beta^? \equiv \alpha^?.\text{case}\{\text{True} \Rightarrow 1, \text{False} \Rightarrow 5\} \\ \mapsto_u & \vdash \alpha^? \equiv \text{True} \quad \wedge \vdash \beta^? \equiv \text{True}.\text{case}\{\text{True} \Rightarrow 1, \text{False} \Rightarrow 5\} \\ \stackrel{1}{\equiv}_{\beta} & \vdash \alpha^? \equiv \text{True} \quad \wedge \vdash \beta^? \equiv 1 \end{array}$$

This is what the example tells us: 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. Doing so might introduce redexes!

**Definition 7.2** (Unification).

$$e \equiv \alpha^? \quad \mapsto_u \alpha^? \equiv e \quad (1)$$

$$\mathcal{C} \wedge \Psi \vdash \alpha^? \equiv e \quad \mapsto_u \mathcal{C}[e/\alpha^?] \wedge \Psi \vdash \alpha^? \equiv e \quad (\alpha^? \notin e) \quad (2)$$

$$\mathcal{C} \wedge \Psi \vdash \alpha^? \equiv e \quad \mapsto_u \perp \quad (\alpha^? \in e) \quad (3)$$

Step 2 and 3 are the occurs' check.

## 7.3 The algorithm

We have described steps of the algorithm so far. 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.

When we can't further simplify, we non-deterministically apply unification and 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.

## 8 Literature review

In [2], Huet’s algorithm is described.

## 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] G. Dowek. Higher-order unification and matching. In J. A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning (in 2 volumes)*, pages 1009–1062. Elsevier and MIT Press, 2001.
- [3] A. Martelli and U. Montanari. An efficient unification algorithm. *ACM Trans. Program. Lang. Syst.*, 4(2):258–282, Apr. 1982.
- [4] 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