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

List of Theorems

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 (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. While data is concerned with how values are constructed or stored, codata is concerned with how values are destructed or used.

For instance, look at these different ways to define the same tuple:

Example 1.1.

$$\texttt{Tup}(\texttt{1}, \texttt{5})$$
$$\textbf{cocase}\{fst \Rightarrow \texttt{1}, snd \Rightarrow \texttt{5}\}$$

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

We use copatterns (**cocase**{...}) to instantiate codata.

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 1.2 (First-order unification problems)**.**

$$\texttt{5} \equiv \alpha^?$$
$$\texttt{True} \equiv \texttt{False}$$
$$\mathsf{List}(\alpha^?) \equiv \mathsf{List}(\beta^?)$$
$$\texttt{Tup}(\texttt{1}, \alpha^?) \equiv \beta^?$$

Each of the equations can be seen as their own unification problem. The first has an obvious solution $[\mathtt{5}/\alpha^?]$, so substituting $\mathtt{5}$ for $\alpha^?$. 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 - I will discuss which is the one we want, later on.

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

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

The first example has the solution $[id/\alpha^?]$.

In this thesis, I will explore a calculus that has copatterns 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 $X_1, ...X_i, ...X_n$.
A pattern match $e.\mathbf{case}\{\overline{K(\overline{x}) \Rightarrow e}\}$ 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 $\mathbf{cocase}\{\overline{d(\overline{x}) \Rightarrow e}\}$ the same rules apply, but instead of constructors, the term is matched against destructors.

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

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

In pattern and copattern matches, every con- or destructor may occur no more than once.

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

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

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)**.**

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

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)**.**

$$\begin{aligned}&\texttt{Tup}(1, 2).\textbf{case}\{\texttt{Tup}(x, y) \Rightarrow x\}\\&\texttt{Cons}(\texttt{True}, \texttt{Cons}(\texttt{False}, \texttt{Nil})).\textbf{case}\{\texttt{Cons}(x, y) \Rightarrow y\}\end{aligned}$$

This is the pattern matching You know and love. We will see later how this is evaluated to `1` and $\texttt{Cons}(\texttt{False}, \texttt{Nil})$, 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)**.**

$$\begin{aligned}&t.\textbf{case}\{\texttt{True} \Rightarrow e_1, \texttt{False} \Rightarrow e_2\} && \widehat{=} \text{ if } t \text{ } e_1 \text{ else } e_2\\&e.\textbf{case}\{\texttt{Cons}(\texttt{True}, x) \Rightarrow \texttt{True}, \texttt{Nil} \Rightarrow \texttt{False}, \texttt{Cons}(x, y) \Rightarrow \texttt{False}\}\end{aligned}$$

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

Example 2.5 (Using Copattern Matching to instantiate Codata)**.**

$$\textbf{cocase}\{\texttt{fst} \Rightarrow \texttt{1}, \texttt{snd} \Rightarrow \texttt{2}\}$$
$$TrafficLight = \textbf{cocase}\{\texttt{Green} \Rightarrow \texttt{Yellow}, \texttt{Yellow} \Rightarrow, \texttt{Red} \Rightarrow \texttt{Green}\}$$

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*.

Example 2.6 (Using Destructors to take apart Codata)**.**

$$\textbf{cocase}\{\texttt{fst} \Rightarrow \texttt{1}, \texttt{snd} \Rightarrow \texttt{2}\}.\texttt{fst}$$
$$TrafficLight.\texttt{Green}$$
$$id.\texttt{Ap}(\texttt{5})$$

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)**.**

$$id = \textbf{cocase}\{\texttt{Ap}(x) \Rightarrow x\}$$
$$\textbf{cocase}\{\texttt{Ap}(x) \Rightarrow (\texttt{Ap}(y) \Rightarrow x)\} \qquad \widehat{=}\lambda x.\lambda y.x$$
$$\textbf{cocase}\{\texttt{Ap}(x) \Rightarrow (\texttt{Ap}(y) \Rightarrow y)\} \qquad \widehat{=}\lambda x.\lambda y.y$$

The second and third examples implement a function that takes two arguments, ignores one and gives back the other.

2.3 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:

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

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:

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

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:

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

What is actually interesting is what happens when we appy 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:

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

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.

The composition of two substitutions $\sigma_2 \circ \sigma_1$ 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 problem.

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

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

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

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

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

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

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

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

2.4 Conversion

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

$$K(\overline{e}).\mathbf{case}\{\dots, K(\overline{x}) \Rightarrow e, \dots\} \equiv^1_\beta e[\overline{e}/\overline{x}] \qquad (\beta\text{-Data})$$
$$\mathbf{cocase}\{\dots, d(\overline{x}) \Rightarrow e, \dots\}.d(\overline{e}) \equiv^1_\beta e[\overline{e}/\overline{x}] \qquad (\beta\text{-Codata})$$

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, i.e. terms that can't be evaluated.

Some examples:

$$\texttt{True}.\mathbf{case}\{\texttt{False} \Rightarrow \texttt{True}, \texttt{True} \Rightarrow \texttt{False}\} \equiv^1_\beta \texttt{False}$$
$$\mathbf{cocase}\{\texttt{Ap}(x) \Rightarrow \texttt{True}\}.\texttt{Ap}(x) \equiv^1_\beta \texttt{True}[x/x] = \texttt{True}$$
$$\mathbf{cocase}\{\texttt{Ap}(y) \Rightarrow y\}.\texttt{Ap}(x) \equiv^1_\beta y[x/y] = x$$

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:

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

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:

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

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 Γ, we can deduce that t has the type τ.

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

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

$$\frac{\Gamma \vdash t : \mathsf{Pair}(t_1, t_2) \qquad \Gamma, x \vdash : \tau_1, y : \tau_2, t : \tau}{\Gamma \vdash t.\mathbf{case}\{\mathtt{Tup}(t_1, t_2) \Rightarrow t\} : \tau}\ \textsc{Case-Pair}$$

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

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

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

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

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

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

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

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

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 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 $\overline{e \equiv e}$ containing unknown unification variables $\alpha_1^?, \alpha_2^?, ...$, 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 (respectively).

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 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)**.**

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

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

$\mathsf{Int} \to \alpha^? \equiv \mathsf{Int} \to \mathsf{Bool}$ has the solution $[\mathsf{Bool}/\alpha^?]$. On the other hand, $\alpha^? \equiv \mathsf{List}(\alpha^?)$ 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 $\overline{e \equiv e}$, 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)**.** $\perp$ is the symbol for fail. The algorithm is defined by non-deterministically applying the

below rules:

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

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)**.**

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

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: $\mathbf{cocase}\{\mathtt{Ap}(x) \Rightarrow \alpha^?\}.\mathtt{Ap}(y) \equiv \beta^?$. First focus on the left side and notice that there is a redex. What happens if we reduce it?

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

This motivates our need for substitutions: We may need to apply substitutions to unification variables. Note that this is *not* possible in first-order unification, since we don't create subsitutions 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 our 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 and/or are equivalent externally.

Let's next consider a few examples: The problem $\alpha^?.\texttt{Ap}(5) \equiv 5$ has multiple solutions: $[\textbf{cocase}\{\texttt{Ap}(x) \Rightarrow x\}/\alpha^?]$ and $[\textbf{cocase}\{\texttt{Ap}(x) \Rightarrow 5\}/\alpha^?]$,
where `Ap` is a function applicator. Thus, the two given solutions are the identity function and the constant function 5, which when consuming the argument 5, both give back 5. In this case there also doesn't exist an mgu.

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

Definition 5.2 (Constraint)**.**

$$C ::= \top \mid \bot \mid \Psi \vdash e \equiv t$$
$$\mathcal{C} ::= C \wedge \mathcal{C}$$

Ψ, called context is just a list of distinct variables $x_1, ...x_n$ and may be empty.

$\Psi \vdash e \equiv t$ means that given the variables Ψ, we deduce that $e \equiv t$. Ψ may contain variables that occur in or are even bound in e, t. Ψ contains those variables we have seen before and want to remember. This motivates the naming of Ψ 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 Ψ. We join them using ands: $\overline{e \equiv t}$ becomelowering für codata types hinzufügens: $\mathcal{C} = C_1 \wedge ... \wedge C_n = \vdash e_1 \equiv t_1 \wedge ... \wedge \vdash e_n \equiv t_n$

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

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 $\bot$ 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

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

where it holds that all $x_1, ...x_n$ 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: $\alpha^?.\mathtt{Ap}(x).\mathtt{Ap}(y) \equiv x$ is a pattern and has the solution $[\mathbf{cocase}\{\mathtt{Ap}(x) \Rightarrow \mathbf{cocase}\{\mathtt{Ap}(y) \Rightarrow x\}\}/\alpha^?]$. $\alpha^?.fst \equiv 2$ 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 problem $\alpha^?.\mathtt{Ap}(x).\mathtt{Ap}(x) \equiv x$ (where the variables are **not** distinct). We can name two solutions:

$$\sigma_1 = [\mathbf{cocase}\{\mathtt{Ap}(x) \Rightarrow \mathbf{cocase}\{\mathtt{Ap}(y) \Rightarrow x\}\}/\alpha^?]$$
$$\sigma_2 = [\mathbf{cocase}\{\mathtt{Ap}(x) \Rightarrow \mathbf{cocase}\{\mathtt{Ap}(y) \Rightarrow y\}\}/\alpha^?]$$

(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. 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:

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

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 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: $K(\overline{e}).\mathbf{case}\{\ldots, K(\overline{x}) \Rightarrow e\}$, we start with the term $n.\mathbf{case}\{\overline{K(\overline{x}) \Rightarrow v}\}$, now wanting to subtitute $K(\overline{e})$ 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: $\mathbf{cocase}\{\ldots, d(\overline{x}) \Rightarrow e, \ldots\}.d(\overline{e})$, where we could 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 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 6.1.

$$r = \alpha^?.\mathtt{Ap}(\mathbf{cocase}\{\mathtt{Ap}(x) \Rightarrow x\}),$$
$$s = \mathbf{cocase}\{\mathtt{Ap}(x) \Rightarrow \mathtt{Cons}(x, \mathtt{Cons}(y, \mathtt{Nil}))\},$$
$$t = \alpha^?.\mathtt{Ap}(x_1, x, y, z)$$

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

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

Let's take a look at another example:

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

To find out that $[\texttt{2}/\beta^?]$ 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 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 (Pairs and).

$$\begin{aligned}x \vdash x.\texttt{fst} \equiv x.\texttt{fst} \quad &\mapsto \top \\ x \vdash x.\texttt{fst} \equiv x.\texttt{snd} \quad &\mapsto \bot \\ x, y \vdash x.\texttt{fst} \equiv y.\texttt{fst} \quad &\mapsto \bot\end{aligned}$$

These example might be confusing at first. Looking at the second example, one might argue that there exists a pair, say $x = \texttt{Tup}(\texttt{2}, \texttt{2})$, 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 $x = (2, 3)$, the equation does not hold, we can simplify to $\bot$.

Similarly, for the third example, one might argue that there exists a solution, we just need to apply the substitution $[y/x]$ to the equation. Still, we may not choose our variables. Another way to look at it is this: Some other equations might "choose our variables" for us! If we were to apply the given substitution, we would have to apply it to all equations. What if we have another equation that imposes other constraints on our variables, like for example: $x \equiv$

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

$$\begin{aligned}
&\vdash \texttt{Cons(x, Nil)} \equiv \texttt{Cons(y, Nil)} && \mapsto \bot \\
&\vdash \texttt{Cons}(\alpha^?, \texttt{Nil}) \equiv \texttt{Cons}(\beta^?, \texttt{Nil}) && \mapsto \alpha^? \equiv \beta^? \\
&\vdash \mathsf{List}(\mathsf{Int}) \equiv \mathsf{List}(\alpha^?) && \mapsto \vdash \mathsf{Int} \equiv \alpha^? \\
&x \vdash x.\texttt{Ap}(e_1) \equiv x.\texttt{Ap}(e_2) && \mapsto x \vdash e_1 \equiv e_2
\end{aligned}$$

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

$$\begin{aligned}
&\vdash \mathbf{cocase}\{\texttt{Ap}(x) \Rightarrow \alpha^?\} \equiv \mathbf{cocase}\{\texttt{Ap}(x) \Rightarrow \beta^?\} && \mapsto x \vdash \alpha^? \equiv \beta^? \\
&x \vdash x.\mathbf{case}\{\texttt{T} \Rightarrow e_2, \texttt{F} \Rightarrow e_2\} \equiv x.\mathbf{case}\{\texttt{T} \Rightarrow t_1, \texttt{F} \Rightarrow t_2\}^* \\
&\mapsto x \vdash e_1 \equiv t_1 \land x \vdash e_2 \equiv t_2 \\
&\mathbf{cocase}\{\texttt{fst} \Rightarrow 5, \texttt{snd} \Rightarrow \texttt{False}\} \equiv \alpha^? \\
&\mapsto \vdash 5 \equiv \alpha^?.\texttt{fst} \land \vdash \texttt{False} \equiv \alpha^?.\texttt{snd}
\end{aligned}$$

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.

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

Definition 7.1 (Decomposing Constraints)**.**

$$\Psi \vdash x \equiv x \qquad \mapsto_r \top \tag{1}$$

$$\Psi \vdash \alpha^? \equiv \alpha^? \qquad \mapsto_r \top \tag{2}$$

$$\circ\ \Psi \vdash e_1.d(\overline{e}) \equiv e_2.d(\overline{t}) \quad \mapsto_d \Psi \vdash e_1 \equiv e_2 \wedge \overline{\Psi \vdash e \equiv t} \tag{3}$$

$$*\ \Psi \vdash e_1.\mathbf{case}\{\overline{K(\overline{x}) \Rightarrow e}\} \equiv e_2.\mathbf{case}\{\overline{K(\overline{x}) \Rightarrow t}\} \quad \mapsto_d \Psi \vdash e_1 \equiv e_2 \wedge \overline{\Psi, \overline{x} \vdash e \equiv t} \tag{4}$$

$$\circ\ \Psi \vdash K(\overline{e}) \equiv K(\overline{t}) \qquad \mapsto_d \overline{\Psi \vdash e \equiv t} \tag{5}$$

$$\Psi \vdash \mathbf{cocase}\{\overline{d(\overline{x}) \Rightarrow e.d(\overline{x})}\} \equiv t \qquad \mapsto_e \Psi \vdash e \equiv t \tag{6}$$

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

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

$$\Psi \vdash t \equiv \mathbf{cocase}\{\overline{d(\overline{x}) \Rightarrow e}\} \qquad \mapsto_d \overline{\Psi, \overline{x} \vdash t.d(\overline{x}) \equiv e} \tag{9}$$

$$\text{non-matching } *\text{- or } \circ\text{-equation} \qquad \mapsto_c \bot \tag{10}$$

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 ∘, 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 ⊥.

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

$$\begin{aligned} &\mathbf{cocase}\{\overline{d(\overline{x}) \Rightarrow e}\} \equiv t && \equiv_\eta \\ &\mathbf{cocase}\{\overline{d(\overline{x}) \Rightarrow e}\} \equiv \mathbf{cocase}\{\overline{d(\overline{x}) \Rightarrow t.d(\overline{x})}\} && 6 \\ &\qquad \mapsto \overline{e \equiv t.d(\overline{x})} \end{aligned}$$

(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 have a solution? ...

Take a look at the following unification problem:

Example 7.4.

$$\begin{aligned} \mathsf{List}(\alpha^?) &\equiv \mathsf{List}(\mathsf{Int}) \\ \beta^? &\equiv \alpha^?.\mathbf{case}\{\mathsf{Int} \Rightarrow \mathtt{True}\} \end{aligned}$$

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

$$\begin{array}{lll} \mathcal{C} = & \vdash \mathsf{List}(\alpha^?) \equiv \mathsf{List}(\mathsf{Int}) & \wedge \vdash \beta^? \equiv \alpha^?.\mathbf{case}\{\mathsf{Int} \Rightarrow \mathtt{True}\} \\ \overset{(5)}{\longmapsto}_d & \vdash \alpha^? \equiv \mathsf{Int} & \wedge \vdash \beta^? \equiv \alpha^?.\mathbf{case}\{\mathsf{Int} \Rightarrow \mathtt{True}\} \\ \mapsto_u & \vdash \alpha^? \equiv \mathsf{Int} & \wedge \vdash \beta^? \equiv \mathsf{Int}.\mathbf{case}\{\mathsf{Int} \Rightarrow \mathtt{True}\} \\ \overset{\equiv^1_\beta}{=} & \vdash \alpha^? \equiv \mathsf{Int} & \wedge \vdash \beta^? \equiv \mathtt{True} \end{array}$$

In this example, several things are expressed: When we found a solution for a unification variable, we want to keep it in our constraints, since we sometimes have to substitute already found unification variables in other equations (constraints). Doing so might introduce redexes.

Definition 7.2 (Unification)**.**

$$e \equiv \alpha^? \qquad \mapsto_u \alpha^? \equiv e \tag{1}$$

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

$$\mathcal{C} \wedge \Psi \vdash \alpha^? \equiv e \qquad \mapsto_u \bot \qquad (\alpha^? \in e) \tag{3}$$

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.

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