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 different subsections of higher-order unification problems have been discussed. 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, discussing their differences and decidability.

Finally, a framework for an algorithm and the algorithm for the aforementioned calculus and higher-order unification is presented, covering all pattern unification problems and those 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 a big focus on codata and data. This requires you 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 unification you probably already know. This is the goal for the next few paragraphs, as well as giving a quick overview of the different parts of this thesis.

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 (roughly) 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 destructed or used. For instance, look at these different ways to define the same tuple:

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

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

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

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 most likely the most familiar with:

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

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

This is the identity function, which is implemented through mapping a function applicator on the variable x to x.

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

$$\mathsf{5} \equiv \alpha^? \tag{1}$$

$$\mathtt{True} \equiv \mathtt{False} \tag{2}$$

$$\mathsf{List}(\alpha^?) \equiv \mathsf{List}(\beta^?) \tag{3}$$

$$\mathtt{Tup}(\mathtt{1}, \alpha^?) \equiv \beta^? \tag{4}$$

Each of the given equations can be seen as their own unification problem. The first has one obvious solution $[\mathsf{5}/\alpha^?]$, so substituting $\mathsf{5}$ 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 2 unification variables, and have many solutions.

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 $\mathsf{5}$, evaluates to $\mathsf{5}$:

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

We can think of two obvious solutions for this: $[id/\alpha^?]$ and the constant function which always returns $\mathsf{5}$: $[\mathbf{cocase}\{\mathtt{ap}(x) \Rightarrow \mathsf{5}\}/\alpha^?]$. As already implied before, sometimes our unification problems have more than one solution, even though we are interested in one particular solution. This is the solution that is more general, meaning 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 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 were to substitute $[\mathtt{True}/\alpha^?]$, the left side would contract to $\mathtt{1}$, since the first case matches - solving the equation.

1.3 Overview

Firstly, I will explore an (untyped) calculus with a heavy focus on the dualities between data and codata in 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. Furthermore, I will introduce some concepts we need so that we can start to play around with terms.

In 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 4, explaining the problem formally and giving an algorithm to solve it.

Then I explain higher-order unification (5) formally, as well as a more specific unification problem which is easier to solve.

Finally, in 6, I present the framework for our equations, and then go through our algorithm for higher-order unification problems in our calculus. It is made up of two parts, which I go though in more detail, then describing how they fit together to form our algorithm on a high level.

In the end (7) is a literature review, going over related works.

2 The Untyped Calculus ND

I will be introducing the Untyped Calculus ND, based on .... It 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 $e.\mathbf{case}\{\overline{K(\overline{x}) \Rightarrow t}\}$ matches an expressio n e against a sequence of clauses, each clause consisting of a constructor and an term t. For a copattern match $\mathbf{cocase}\{\overline{d(\overline{x}) \Rightarrow e}\}$ similar rules apply, but instead of constructors, the expression is matched against destructors.

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

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

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 $\texttt{Tup}(x_1, x_2)$ and $\texttt{Tup}(x, y)$ have the same meaning).

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 composite types of corse require arguments.

$$
\begin{array}{ll}
\mathsf{Int} & (1) \\
\mathsf{Bool} & (2) \\
\mathsf{List}(\mathsf{Int}) & (3) \\
\mathsf{Pair}(\mathsf{Int}, \mathsf{Bool}) & (4)
\end{array}
$$

Types are explored in detail and formally later on in 3.

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

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

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

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 actual numbers as a shorthand throughout this thesis, though.

When we have instantiated terms, we also need a way to take them apart. For some terms, this can be done using pattern matching:

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

$$\texttt{True}.\textbf{case}\{\texttt{True} \Rightarrow \texttt{False}, \texttt{False} \Rightarrow \texttt{True}\} \quad (1)$$
$$\texttt{Tup}(\texttt{1}, \texttt{2}).\textbf{case}\{\texttt{Tup}(x, y) \Rightarrow x\} \quad (2)$$
$$\texttt{Cons}(\texttt{True}, \texttt{Cons}(\texttt{False}, \texttt{Nil})).\textbf{case}\{\texttt{Cons}(x, y) \Rightarrow x\} \quad (3)$$

This is the pattern matching you probably know. We will see later (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)**.**

$$t.\textbf{case}\{\texttt{True} \Rightarrow e_1, \texttt{False} \Rightarrow e_2\} \quad (1)$$
$$e.\textbf{case}\{\texttt{Cons}(\texttt{True}, x) \Rightarrow \texttt{True}, \texttt{Nil} \Rightarrow \texttt{False}, \texttt{Cons}(x, y) \Rightarrow \texttt{False}\} \quad (2)$$

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

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

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

$$\textbf{cocase}\{\texttt{fst} \Rightarrow \texttt{1}, \texttt{snd} \Rightarrow \texttt{2}\} \quad (1)$$
$$TrafficLight = \textbf{cocase}\{\texttt{Green} \Rightarrow \texttt{Yellow}, \texttt{Yellow} \Rightarrow, \texttt{Red} \Rightarrow \texttt{Green}\} \quad (2)$$
$$Trues = \textbf{cocase}\{\texttt{fst} \Rightarrow \texttt{True}, \texttt{snd} \Rightarrow Trues\} \quad (3)$$

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 example is the infinite stream of $Trues$.

Now for how to take instantiations of codata apart:

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

$$\textbf{cocase}\{\texttt{fst} \Rightarrow \texttt{1}, \texttt{snd} \Rightarrow \texttt{2}\}.\texttt{fst} \quad (1)$$
$$TrafficLight.\texttt{Green} \quad (2)$$
$$id.\texttt{ap}(\texttt{5}) \quad (3)$$

These examples do exactly what you think they do: We select the first value of the tuple, turn our traffic light Yellow, and apply the identity function on `5`, respectively. We will see how these examples resolve to `1`, `Yellow` and `5` in 2.9.

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 definitions:

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

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

The first example actually implements the identity function. The second and third examples implement a functions that take two arguments, ignore one and give back the other (the first and second, respectively).

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 rereading the titles of the definitions: 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 definitions emphasizes these conceptual dualitites: Conceptually, data is dual to codata, but this is only because constructors are conceptually dual to copattern matching, and pattern matching is conceptually dual to destructors. One more clean example to drive this point home:

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

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

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:

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

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:

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

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 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:

$$\begin{aligned}
x[e_1, ...e_n/x_1, ...x_n] :=& e_i \quad (\text{if } x = x_i) \\
y\sigma :=& y \quad (\text{if } y \notin \mathtt{dom}(\sigma)) \\
(K(e_1, ...e_n))\sigma :=& K(e_1\sigma, \ldots, e_n\sigma) \\
(e.\mathbf{case}\{\overline{K(\overline{x}) \Rightarrow t}\})\sigma :=& (e\sigma).\mathbf{case}\{\overline{K(\overline{y} \Rightarrow (e\sigma')\sigma)}\} \\
(\mathbf{cocase}\{\overline{d(\overline{x}) \Rightarrow t}\})\sigma :=& \mathbf{cocase}\{\overline{(d(\overline{y}) \Rightarrow (e\sigma')\sigma)}\} \\
(e.d(e_1, ...e_n))\sigma :=& (e\sigma).d(e_1\sigma, \ldots, e_n\sigma)
\end{aligned}$$

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 need more than one substitution, or rather want to attach substitutions together. This process is calles composition of substitutions: $\sigma = \sigma_2 \circ \sigma_1$, 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 problem.

Consider these two examples of an idempotent substitution, and one 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:

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

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

Consider these examples of beta-conversion:

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

$$\mathbf{cocase}\{\mathtt{ap}(x) \Rightarrow \mathtt{True}\}.\mathtt{ap}(x) \equiv^1_\beta \mathtt{True}[x/x] = \mathtt{True} \qquad (1)$$
$$\mathbf{cocase}\{\mathtt{ap}(y) \Rightarrow y\}.\mathtt{ap}(x) \equiv^1_\beta y[x/y] = x \qquad (2)$$
$$\mathtt{True}.\mathbf{case}\{\mathtt{False} \Rightarrow \mathtt{True}, \mathtt{True} \Rightarrow \mathtt{False}\} \equiv^1_\beta \mathtt{False} \qquad (3)$$
$$\mathbf{cocase}\{\mathtt{fst} \Rightarrow \mathtt{1}, \mathtt{snd} \Rightarrow \mathtt{2}\}.\mathtt{fst} \equiv^1_\beta \mathtt{1} \qquad (4)$$

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:

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

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

Here are some examples of eta-conversion:

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

$$\mathbf{cocase}\{\mathtt{ap}(y) \Rightarrow id.\mathtt{ap}(y)\} \equiv^1_\eta id \qquad (1)$$
$$\mathbf{cocase}\{\mathtt{fst} \Rightarrow specificPair.\mathtt{fst}, \mathtt{snd} \Rightarrow specificPair.\mathtt{snd}\} \equiv^1_\eta e \qquad (2)$$
$$\text{where } specificPair = \mathbf{case}\{\mathtt{fst} \Rightarrow \mathtt{1}, \mathtt{snd} \Rightarrow \mathtt{2}\}$$

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 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:

Example 3.1 (Simple types)**.**

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

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 want 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.

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 (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. Even though first-order unification is a pretty limited subproblem of higher-order unification, in many application it is all that is needed - most simple types can be constructed from first-order terms.

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

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

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

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

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

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

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)**.** $\perp$ is the symbol for fail. The algorithm is defined by non-deterministically applying the below rules:

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

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)**.**

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

Note that this is encompassed our syntax described in 2.1, 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:

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 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.

In the introduction (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 more general solution)**.**

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

This problem has multiple solutions:

$\sigma_1 = [\mathbf{cocase}\{\mathtt{ap}(x) \Rightarrow x\}/\alpha^?]$ identity function

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

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 after 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!

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 includes unification problems containing higher-order terms (including functions), 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.

5.1 Pattern Unification

Pattern unification, also sometimes called the pattern fragment is a subsection of higher-order unification, with its path to a 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, we need to extend our definition to more than just function applications. This means that (conceptually) the pattern fragment described by Miller is a subset of our definition.

Definition 5.2 (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.

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

Example 5.3 (Pattern)**.**

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

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 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 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. 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 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 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: $K(\overline{e}).\textbf{case}\{\ldots, K(\overline{x}) \Rightarrow e\}$, we start with the term $n.\textbf{case}\{\overline{K(\overline{x}) \Rightarrow v}\}$, 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 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 an term we can't further evaluate (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. If and only if a unification problem has a solution, that solution is the same for the normal form of that problem. The same thing holds for the other way around: If and only if a normal form has a solution, that solution is the same for the extended expression of that normal form.

This is helpful for us, since we only have to consider normal forms in our algorithm. Concretely: Met with a term, we first reduce 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)**.**

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

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

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

Let's take a look at another example of a using beta-conversion:

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

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 we don't have all the information for yet until we know more.

6 The Algorithm For Higher-Order Unification

6.1 Constraints

Even though higher-order unification is undecidable, in many cases we can of corse still find solutions, or conclude that there is no solution. Since we don't only want to describe higher-order unification, but the steps necessary to solve higher-order unification problems, we need to have a framework of our equations; a framework 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)**.**

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

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 may contain variables that occur in or are even bound in e or t. Ψ contains those variables we have seen before and want to remember. This motivates the naming of Ψ as context, 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: $\overline{e \equiv t}$ 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 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: $\mathcal{C}\sigma = (C \wedge \mathcal{C}')\sigma = C\sigma \wedge \mathcal{C}'\sigma = \Psi \vdash e\sigma \equiv t\sigma \wedge \mathcal{C}'\sigma$. 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 con-

tradictory 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 $x = \textbf{cocase}\{\texttt{fst} \Rightarrow \texttt{2}, \texttt{snd} \Rightarrow \texttt{2}\}$, 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 $x = \textbf{cocase}\{\texttt{fst} \Rightarrow \texttt{2}, \texttt{snd} \Rightarrow \texttt{3}\}$, the equation does not hold, we can simplify to $\bot$.

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

$$\vdash \texttt{Cons}(x, \texttt{Nil}) \equiv \texttt{Cons}(y, \texttt{Nil}) \qquad \mapsto \bot \tag{1}$$
$$\vdash \texttt{Cons}(\alpha^?, \texttt{Nil}) \equiv \texttt{Cons}(\beta^?, \texttt{Nil}) \qquad \mapsto \vdash \alpha^? \equiv \beta^? \tag{2}$$
$$\vdash \mathsf{List}(\mathsf{Int}) \equiv \mathsf{List}(\alpha^?) \qquad \mapsto \vdash \mathsf{Int} \equiv \alpha^? \tag{3}$$
$$x \vdash x.\texttt{ap}(e_1) \equiv x.\texttt{ap}(e_2) \qquad \mapsto x \vdash e_1 \equiv e_2 \tag{4}$$
$$\vdash \alpha^?.\texttt{fst} \equiv \beta^?.\texttt{fst} \qquad \mapsto \vdash \alpha^? \equiv \beta^? \tag{5}$$

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

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

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 prematurely stop the solving process.

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

Definition 6.2 (Decomposing Constraints)**.**

Removing redundancy

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

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

$\Psi \vdash K() \equiv K() \quad \mapsto_r \top \quad (3)$

Decomposition

$\circ\ \Psi \vdash K(\overline{e}) \equiv K(\overline{t}) \quad \mapsto_d \overline{\Psi \vdash e \equiv t} \quad (4)$

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

$\quad \mapsto_d \Psi \vdash e_1 \equiv e_2 \wedge \overline{\Psi \vdash e \equiv t} \quad (5)$

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

$\quad \mapsto_d \Psi \vdash e_1 \equiv e_2 \wedge \overline{\Psi, \overline{x} \vdash e \equiv t} \quad (6)$

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

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

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

Eta-reduction

$\Psi \vdash \mathbf{cocase}\{\overline{d(\overline{x}) \Rightarrow e.d(\overline{x})}\} \equiv t \quad \mapsto_e \Psi \vdash e \equiv t \quad (10)$

Removing contradictions

$\Psi \vdash x \equiv y \quad \mapsto_c \bot \quad (11)$

$\Psi \vdash K_1() \equiv K_2() \quad \mapsto_c \bot \quad (12)$

non-matching *-equation $\quad \mapsto_c \bot \quad (13)$

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). This is because in *-equations, we expect variables, whereas in ∘-arguments, we expect values (since we expect terms in normal form). In *-equations where the arguments are not equal syntactically, rule 13 applies and we simplify to ⊥:

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

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

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

Note that 8 (as well as 9 with being just the mirror of 8) are possible through eta-conversion:

$$\begin{aligned}\mathbf{cocase}\{\overline{d(\overline{x}) \Rightarrow e}\} &\equiv t \\ \mathbf{cocase}\{\overline{d(\overline{x}) \Rightarrow e}\} &\equiv \mathbf{cocase}\{\overline{d(\overline{x}) \Rightarrow t.d(\overline{x})}\} \\ &\overset{(7)}{\longmapsto}_d \overline{e \equiv t.d(\overline{x})}\end{aligned}$$

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, introducing what to do when we found a solution.

$$\begin{array}{lllll}
\mathcal{C} = & \vdash \mathtt{Tup}(\alpha^?, \mathtt{F}) \equiv \mathtt{Tup}(\mathtt{T}, \mathtt{F}) & \wedge \vdash \beta^? \equiv \alpha^?.\mathbf{case}\{\mathtt{T} \Rightarrow \mathtt{1}, \mathtt{F} \Rightarrow \mathtt{5}\}^* \\
\overset{(4)}{\longmapsto}_d & \vdash \alpha^? \equiv \mathtt{T} & \wedge \vdash \mathtt{F} \equiv \mathtt{F} & \wedge \vdash \beta^? \equiv \alpha^?.\mathbf{case}\{\mathtt{T} \Rightarrow \mathtt{1}, \mathtt{F} \Rightarrow \mathtt{5}\} \\
\overset{(3)}{\longmapsto}_r & \vdash \alpha^? \equiv \mathtt{T} & & \wedge \vdash \beta^? \equiv \alpha^?.\mathbf{case}\{\mathtt{T} \Rightarrow \mathtt{1}, \mathtt{F} \Rightarrow \mathtt{5}\} \\
\mapsto_u & \vdash \alpha^? \equiv \mathtt{T} & & \wedge \vdash \beta^? \equiv \mathtt{T}.\mathbf{case}\{\mathtt{T} \Rightarrow \mathtt{1}, \mathtt{F} \Rightarrow \mathtt{5}\} \\
\overset{\equiv^1_\beta}{=} & \vdash \alpha^? \equiv \mathtt{T} & & \wedge \vdash \beta^? \equiv \mathtt{1}
\end{array}$$

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)**.**

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

Step 2 and 3 are the occurs check.

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

We should be careful, because there is another thing we could be introducing through unification steps, though:

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

$$
\begin{array}{lllll}
\mathcal{C} = & \vdash \mathbf{cocase}\{\mathtt{ap}(x) \Rightarrow \alpha^?\} \equiv id & \wedge & \vdash \beta^?.\mathtt{ap}(x) \equiv \alpha^? \\
\overset{(8)}{\longmapsto}_d & x \vdash \alpha^? \equiv id.\mathtt{ap}(x) & \wedge & \vdash \beta^?.\mathtt{ap}(x) \equiv \alpha^? \\
\overset{(2)}{\longmapsto}_u & x \vdash \alpha^? \equiv id.\mathtt{ap}(x) & \wedge & \vdash \beta^?.\mathtt{ap}(x) \equiv id.\mathtt{ap}(x) \\
\overset{(5)}{\longmapsto}_d & x \vdash \alpha^? \equiv id.\mathtt{ap}(x) & \wedge & \vdash \beta^? \equiv id \wedge \vdash x \equiv x \\
\overset{(1)}{\longmapsto}_r & x \vdash \alpha^? \equiv id.\mathtt{ap}(x) & \wedge & \vdash \beta^? \equiv id
\end{array}
$$

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.

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 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