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

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 $e.\textbf{case}\{\overline{K(\overline{x}) \Rightarrow e}\}$ 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 $\textbf{cocase}\{\overline{d(\overline{x}) \Rightarrow e}\}$ the same rules apply, but instead of constructors, the term is matched against destructors.

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

$$
\begin{array}{llr}
e ::= x & & \text{Variable} \\
\quad \mid K(\overline{e}) & & \text{Constructor} \\
\quad \mid e.d(\overline{e}) & & \text{Destructor} \\
\quad \mid e.\textbf{case}\{\overline{K(\overline{x}) \Rightarrow e}\} & & \text{Pattern match} \\
\quad \mid \textbf{cocase}\{\overline{d(\overline{x}) \Rightarrow e}\} & & \text{Copattern match}
\end{array}
$$

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

Let's look at some examples to ... All rudimentary datatypes can be constructed through constructors: The Booleans `True` and `False` are constructors on an empty sequence. We can of course represent integers as peano numbers using constructors.

Abstract data types like lists, arrays, records etc. can similarly be defined through constructors: `cons(True, cons(False, Nil))` and `Date(...)`

We can use Pattern matching to represent conditionals: Like if statements: if e_1 e_2 else e_3 which is analogous to: $e_1.\textbf{case}\{\texttt{True} \Rightarrow e_2, \texttt{False} \Rightarrow e_3\}$ Or the expression which tests whether a given list starts with `True` : $e.\textbf{case}\{\texttt{cons(True x)} \Rightarrow \texttt{True}, \texttt{Nil} \Rightarrow \texttt{False}, \texttt{cons(x y)} \Rightarrow \texttt{False}\}$

2.2 Data Types

There are of course endless constructors one can define using our syntax, but one can also use copattern matching and destructors to define the same constructs. Take the pair: $pair(x, y)$, which we can also define through its destructors *fst* and *snd*: $\mathbf{cocase}\{fst \Rightarrow ..., snd \Rightarrow ...\}$, as in $pair(\texttt{True}, \texttt{False}).\mathbf{cocase}\{fst \Rightarrow \texttt{True}, snd \Rightarrow \texttt{False}\}$ Streams $\mathbf{cocase}\{head \Rightarrow ..., tail \Rightarrow ...\}$ are a bit less intuitive, but take the example of the stream consisting of `Trues`: $\texttt{Trues} = \mathbf{cocase}\{head \Rightarrow \texttt{True}, tail \Rightarrow \texttt{Trues}\}$

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:

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

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:

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

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, $[\textbf{cocase}\{\texttt{Ap}(x) \Rightarrow x\}/y]$ is idempotent, since:

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

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

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

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:

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

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

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

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:

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

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:

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

Intuitively, beta-conversion means function application or the reduction under pattern or copattern matching.

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

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

3 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 $[e_1, ...e_n/\alpha^?_1, \dots, \alpha^?_n]$ which substitutes expressions for unification variables, such that the sides of the given equations are the same (respectively).

Definition 3.1 (Solution)**.** A solution to a given unification problem is described by a simultaneous substitution $[e_1, ...e_n/\alpha^?_1, \dots, \alpha^?_n]$ which when applied to the problem solves it, i.e. makes the sides of the equations equal

Definition 3.2 (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 3.3 (First-Order unification)**.**

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

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

$\mathit{Int} \to \alpha^? \equiv \mathit{Int} \to \mathit{Bool}$ has the solution $[\mathit{Bool}/\alpha^?]$. Note that we are dealing with type constructors, i.e. the type $\mathit{Int} \to \mathit{Bool}$ would be constructed as $\mathit{Type}(\mathit{Int}, \mathit{Bool})$ where $\mathit{Type}(\cdot)$ is a constructor.

On the other hand, $\alpha^? \equiv \mathit{List}(\alpha^?)$ should not have a solution. This is the negative example for the occur check rule below.

Theorem 3.1 (Decidability of First-Order Unification)**.** For first-order unification, there exists an algorithm on equations 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 3.4 (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.

4 Higher-Order Unification

Definition 4.1 (Higher-Order Unification).

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

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

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

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: $[\mathbf{cocase}\{\texttt{Ap}(x) \Rightarrow x\}/\alpha^{?}]$ and $[\mathbf{cocase}\{\texttt{Ap}(x) \Rightarrow 5\}/\alpha^{?}]$,
where $\texttt{Ap}$ 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.

Theorem 4.1 (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.

5 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 5.1 (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 5.1 (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 5.2 (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:

$$t_1 = \alpha^?.\mathtt{Ap}(\mathbf{cocase}\{\mathtt{Ap}(x) \Rightarrow x\}),$$
$$t_2 = \mathbf{cocase}\{\mathtt{Ap}(x) \Rightarrow \mathtt{cons(x\ cons(y\ Nil))}\},$$
$$t_3 = \alpha^?.\mathtt{Ap}(x_1, x_2, x_3)$$

t_1 and t_2 are in normal form and t_3 is a pattern in normal form.

Theorem 5.2. 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.

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

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

Let's take a look at another example:

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

To find out that $[2/\beta^?]$ is the solution to the second equation, we first need to find out that $[\texttt{True}/\alpha^?]$ 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 Decomposing Constraints

$$\begin{aligned} \textbf{cocase}\{\overline{d(\overline{x}) \Rightarrow e}\} \equiv \textbf{cocase}\{\overline{d(\overline{x}) \Rightarrow t}\} &\mapsto \overline{d(\overline{x}) = d(\overline{x})} \land \overline{e \equiv t} && 1,4 \\ \textbf{cocase}\{\overline{d(\overline{x}) \Rightarrow e}\} \equiv t &\mapsto \overline{e \equiv t.d(\overline{x})} && 2,5 \\ t \equiv \textbf{cocase}\{\overline{d(\overline{x}) \Rightarrow e}\} &\mapsto \overline{t.d(\overline{x}) \equiv e} && 3,6 \end{aligned}$$

These rules are taken from [1], adapted to our syntax.

Note that 2,5 (as well as 3,6 with being just the mirror of 2,5) are possible through eta-conversion:

$$\begin{aligned} &\textbf{cocase}\{\overline{d(\overline{x}) \Rightarrow e}\} \equiv t && \equiv_\eta \\ &\textbf{cocase}\{\overline{d(\overline{x}) \Rightarrow e}\} \equiv \textbf{cocase}\{\overline{d(\overline{x}) \Rightarrow t.d(\overline{x})}\} && 1,4 \\ &\qquad \mapsto \overline{e \equiv t.d(\overline{x})} \end{aligned}$$

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

(Nur weil ich eine Citation irgendwo brauche) 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