Karlsruhe Institute of Technology

Institute for Program Structures
and Data Organization (IPD)

Finding equations in functional programs

Bachelor Thesis of

Johannes Bader

at the Department of Informatics
Institute for Program Structures and Data Organization (IPD)

Reviewer: Prof. Dr.-Ing. Gregor Snelting
Advisor: Dipl.-Math. Dipl.-Inform. Joachim Breitner

Duration: 2014-06-02 – 2014-09-24

I declare that I have developed and written the enclosed thesis completely by myself, and have not used sources or means without declaration in the text, and have followed the rules of the KIT for upholding good scientific practice.

Karlsruhe, 2014-09-24

.....

(**Johannes Bader**)

Abstract

This thesis describes an algorithm finding and proving equations suitable to be used as rewrite rules, which have the potential to simplify a functional program.
To be independent from any specific functional programming language, a dialect of λ-calculus is introduced. It covers common features of such languages, including recursion, pattern matching and case-expressions.
The main focus of this work lies on putting expressions of this language in a partial order. Finally, a concrete strategy for finding rewrite rules using this partial order is specified.

Acknowledgements

I wish to thank my advisor Joachim Breitner for offering me this topic and for his patient assistance throughout the past few months. In any situation and every way, his remarks and thoughts guided me in the right direction.

Also I am very grateful to all my family and friends who encouraged and supported me throughout my life.

Contents

1 Introduction

1.1 Motivation

Optimization is a key feature of compilers. Functional programming languages are usually not as close to the hardware as imperative programming languages like C++. This is a consequence of both execution and data being treated in a much more abstract, mathematical way — rather than a way that reflects the details of underlying hardware. As a result, compilers of functional programming languages have to put great effort into optimization in order to minimize the overhead caused by the transformation from high-level functional to low-level imperative primitives.

One of the main features of purely functional programming languages like Haskell is the immutability of data. Functions designed to transform data (say, a list), will therefore always create brand new data instead of mutating existing objects. Under certain circumstances those functions might actually create an exact copy of an already existing object (e.g. Haskell's `fmap` function of `Functor` instances when passing the identity function as first argument).

However, whenever exact copies of an object are created (i.e. the object is cloned) one could instead just use the original object. As a result of immutability, clones are indistinguishable and thus are always redundant.

Furthermore, the same is true for any two objects that have different type but identical internal representation. While the programmer might be forced (with good reason) to explicitly cast between types (using constructors or pattern matching), those casts might turn out to have no effect under the hood and can therefore be removed during optimization, improving performance. Simply changing the type of an object without touching its internal representation is called *type coercion*.

GHC already knows how to identify such cases of safe type coercions [4]. The current approach has limits, though. Imagine a functor `F` and arbitrary types `A` and `B`. A programmer might want to convert an object of type `F A` to type `F B`, but has available only a method `conv` for converting from type `A` to type `B`. The obvious solution is using the functor's `fmap` function to convert the object, element by element.

GHC may know a safe coercion between `A` and `B` and between `F A` and `F B`. Still, if `conv` turns out to be a safe coercion, only `conv` itself will be a zero-cost operation. The call to `fmap` remains unchanged, whereas "ideal code" would make use of the safe coercion from `F A` to `F B`. The problem is, that the compiler does not recognize the behavior of `fmap conv`, or more specifically: If `fmap` is called with a coercion (ignoring types: an identity function) the result behaves like a coercion as well.

Note that a situation as described here is quite unlikely to occur in source code written by human programmers. However, they may occur in code emitted during optimization itself, i.e. in compiler generated code.

Manually written rewrite rules are currently the only option enabling advanced optimization of this kind. Unfortunately, the responsibility of proving the rewrite rule is laid

upon the programmer.

1.2 Goal

It is desirable for a compiler to automatically find the rewrite rules that programmers are currently required to provide. For a rewrite rule to be valid, it must replace expressions with equivalent ones. In other words, we are looking for a systematic way to find equalities between expressions, which can be expressed using rewrite rules.

In the previous section we were mainly talking about coercions arising from calling higher-order functions with a coercion as a parameter. In an untyped context, this is just a special case of equality having a function call on one side and one of the parameters (unmodified) on the other side of the equation. Pattern:

$$f\ ...\ e\ ... = e$$

We present an algorithm that, given a function (or rather its definition), will automatically search for rewrite rules of this simplifying kind. Intuitively, it will search for special cases of arguments, leading to *trivial* behavior of the overall expression.

First, we will work out a general way of *proving* equations and *finding* equations of a certain pattern. Since this task is relevant independently from any specific programming language, we will operate on an untyped λ-calculus with case-expressions, data constructors and a named fixed-point combinator (to enable easier detection of anonymous recursion).

Our proving algorithm will expect an equation (*claim*) as input, possibly with placeholders (making the equation really just a pattern of equations), and will return equations (*witnesses*) as output. Each witness must be successfully proved and contain none of the claim's placeholders. In other words, every output equation must imply the input equation.

1.3 Related Work

In the λ-calculus, proving the equality of two normalizing expressions is as easy as normalizing them and checking for α-equivalence. Infinitary equational reasoning as introduced in [9] can help dealing with divergent terms.

When it comes to functional programming languages, using logic programming (i.e. transform programs or equations to first-order logic) and theorem provers are the best choice for automated equational reasoning. A transformation from Haskell to first-order logic is presented in [14].

This thesis, in contrast, is meant to present a generic calculus and algorithm, usable independent from any specific functional programming language. Also, we focus on *finding* equations instead of proving given ones.

2 Language and syntax

2.1 Syntax

⟨*Constructor*⟩ ::= ⟨*upper case letter*⟩ [⟨*index*⟩]

⟨*Variable*⟩ ::= ⟨*lower case letter*⟩ [⟨*index*⟩]

⟨*QA*⟩ ::= ['∀' ⟨*QTail*⟩]

⟨*QE*⟩ ::= ['∃' ⟨*QTail*⟩]

⟨*QTail*⟩ ::= ⟨*Variable*⟩ ':'

⟨*PatternAlt*⟩ ::= ⟨*QA*⟩ ⟨*Expression*⟩ ' ↦ ' ⟨*Expression*⟩

⟨*Function*⟩ ::= '(' ⟨*PatternAlt*⟩ { ',' ⟨*PatternAlt*⟩ } ')'

⟨*Cases*⟩ ::= '[' ⟨*PatternAlt*⟩ { ',' ⟨*PatternAlt*⟩ } ']'

⟨*Expression*⟩ ::= 'fix' | ⟨*Constructor*⟩ | ⟨*Variable*⟩ | ⟨*Function*⟩ | ⟨*Cases*⟩
| ⟨*Expression*⟩ ⟨*Expression*⟩
| '(' ⟨*Expression*⟩ ')'

2.2 Expressions

The following constructs are available in our dialect of the λ-calculus:

Data constructor

Data constructors will be written as upper-case Latin letters, possibly together with an index. They are constant and unique.
Examples: A B C_0

Variable

Variables will be written as lower-case Latin letters (identifiers), possibly together with an index. They are named placeholders for expressions, either free or bound (for example by an abstraction).
Examples: a b x y z v_0
Remark: For meta variables, we will use e, f, g.

Application

The common syntax for applications of two expressions applies. That includes left-associativity of the (invisible) application operator and usage of parentheses.
Examples: $e\ f\ g$ $e\ (f\ g)$ $(e\ f)\ g$

Pattern Alternative

Pattern alternatives are generalizations of λ-abstractions. They are inspired by the ρ-calculus [8] and patterns as introduced in [10].

They consist of a list of universally quantified variables, a pattern expression (called "left-hand side" or LHS) and a result expression (called "right-hand side" or RHS).

Note the following points:

- Variables quantified in a pattern alternative and not appearing free in its LHS will not bind the corresponding free variables of the RHS.
- Variables appearing free in the LHS of a pattern alternative are not allowed to be bound by another (surrounding) pattern alternative. This ensures that functions are always monotonically increasing. (The importance of this is shown in section 8.1.7.)
- Unlike λ-abstractions, pattern alternatives themselves are no valid expressions and have no reduction semantics of their own. Instead, they are the key component of functions and case-expressions, which specify detailed reduction rules based on pattern alternatives.

Examples: $e \mapsto f$ $\quad \forall a : a \mapsto b\ a$ $\quad \forall a, b : C\ a\ b \mapsto b$

Function and Case-expression

Functions and case-expressions are both containers of pattern alternatives. Their semantics have subtle but important differences that will be described later. We call functions and case-expressions *methods*.

While the usage of patterns in case-expressions is nothing uncommon, their usage in functions may seem like useless, duplicate logic. In fact, functions and case-expressions have only subtle differences in their semantics, which we will rely on later.

Examples:

Functions	$(\forall a, b : C\ a\ b \mapsto b)$	$(\forall a : C\ a \mapsto a, \forall b : D\ b \mapsto b)$
Case-expressions	$[\forall a, b : C\ a\ b \mapsto b]$	$[\forall a : C\ a \mapsto a, \forall b : D\ b \mapsto b]$

Named fixed-point combinator

With the language being untyped, one can define a fixed-point combinator, e.g.:

$$(\forall f : f \mapsto (\forall x : x \mapsto f\ (x\ x))\ (\forall x : x \mapsto f\ (x\ x)))$$

Nevertheless we also provide a named one: fix

The main advantage is that our algorithms will be able to identify fix as being a fixed-point combinator, whereas it is hard or even undecidable to reliably determine whether an arbitrary expressions is a valid fixed-point combinator.

Expression

The following constructs are valid expressions:

- Variables
- Constructors
- Fixed-point combinator "fix"
- Functions

- Case-expressions

Be aware that constraints (like correctly bound pattern alternative variables) are not ensured by the grammar.

2.3 Notation

- $\bar{e}$ is an abbreviation for a finite range of expressions e_0, e_1, ...
- For any binary relation $\times$,
$$\bar{e} \times \bar{f}$$
holds iff $\bar{e}$ and $\bar{f}$ have same cardinality n and
$$\bigwedge_{i=0}^{n-1} e_i \times f_i$$
- Consecutive quantifiers of the same kind can freely be merged into a single quantifier:
$$\forall w, x : \exists y : \exists z : ... \quad \equiv \quad \forall w : \forall x : \exists y, z : ...$$

2.4 Subexpressions

An expression s is called *subexpressions* of an expression e, if it is an element of the set of e's subexpressions, defined as:

$$sub(e) = \{e\} \cup \begin{cases} sub(f) \cup sub(g), & \text{if } e = f\ g \text{ is an application} \\ sub(p) \cup sub(f), & \text{if } e = \forall \bar{v} : p \mapsto f \text{ is a map} \\ \bigcup_{i=0}^{n} sub(m_i), & \text{if } e = (m_0, ..., m_n) \text{ is a function} \\ \bigcup_{i=0}^{n} sub(m_i), & \text{if } e = [m_0, ..., m_n] \text{ is a case-expression} \\ \varnothing, & \text{otherwise} \end{cases}$$

2.5 Free variables

A variable is *bound*, if it is subexpression of the RHS of a pattern alternative that both quantifies the variable *and* freely contains it inside its LHS. Variables are considered "free" if they are not bound. We define the function $vars$ returning all free variables of an expression or pattern alternative:

$$vars(e) = \begin{cases} \{e\}, & \text{if } e \text{ is a variable} \\ vars(f) \cup vars(g), & \text{if } e = f\ g \text{ is an application} \\ \bigcup_{i=0}^{n} vars(m_i), & \text{if } e = (m_0, ..., m_n) \text{ is a function} \\ \bigcup_{i=0}^{n} vars(m_i), & \text{if } e = [m_0, ..., m_n] \text{ is a case-expression} \\ vars(f) \setminus (vars(p) \cap \bar{v}), & \text{if } e = \forall \bar{v} : p \mapsto f \text{ is a pattern alternative} \\ \varnothing, & \text{otherwise} \end{cases}$$

2.6 Substitutions

We attach curly braces containing substitution clauses to expressions to denote a substitution operation. A substitution clause has the conventional syntax: $variable := expression$

Multiple substitutions listed within the same curly braces are applied *simultaneously*, so the order in which they are mentioned does not matter. This justifies set notation. Chained substitution operators, on the other hand, are applied left to right. Examples:

$$\begin{aligned}
(a\ b\ c)\{a := b\}\{b := a\} &\equiv (a\ a\ c) \\
(a\ b\ c)\{b := a\}\{a := b\} &\equiv (b\ b\ c) \\
(a\ b\ c)\{a := b, b := a\} &\equiv (b\ a\ c) \\
(a\ b\ c)\{b := a, a := b\} &\equiv (b\ a\ c) \\
(a_0\ a_1\ a_2)\{a_n := a_{n+1} \mid n \in \mathbb{N}_0\} &\equiv (a_1\ a_2\ a_3)
\end{aligned}$$

Closed expressions do not contain any free variables.

2.7 Evaluation and equality

The evaluation rules are very close to those of regular λ-calculus. We will later formally define the evaluation relation $\rightsquigarrow$ (see section 4.1), which would not make sense before discussing other, more important details about expressions.

So far, we can think of $\rightsquigarrow$ as performing one step of ordinary β-reduction. It is undecidable whether repeated reduction leads to an expression that cannot be reduced any further (see discussion about normal form in section 4.4).

Equality of two expressions is defined as follows:

Reflexivity	$e = e$	
Transitivity	$e = f \wedge f = g \implies e = g$	
Constructors	$\bar{e} = \bar{f} \implies C\ \bar{e} = C\ \bar{f}$	
Eta-Conversion	$(\forall x : e\ x = f\ x) \iff e = f$	(if e and f do not freely contain x)
Beta-Reduction	$e \rightsquigarrow f \implies e = f$	

3 Partial order of expressions $\sqsubseteq$

There are several reasons for the equality relation to be the *wrong* relation for our algorithm to work on. In this section we motivate and define the partial order of expressions $\sqsubseteq$.

3.1 Motivation

An equality relation feels like the wrong choice in several situations:

3.1.1 Conservative approximation of possibly divergent expressions

Imagine two expressions e and f that our algorithm wants to compare. They probably do not appear similar "at first sight" (e.g. they are not structurally identical), so we try to normalize both of them.

e turns out to have a normal form, but f does not normalize after applying a certain number of reduction steps. (The halting problem forces the algorithm to impose hard limits on the number of performed reduction steps. Thus it may sometimes have to simply give up without knowing whether a normal form exists.)

At this point there are three possible situations:

- f is divergent and therefore not equal to e.
- f has a normal form that is not equal to e.
- f has a normal form that is equal to e.

As a result, we can neither claim $e = f$ nor $e \neq f$, as both of these statements could turn out to be wrong. Therefore, equality is not the right relation for making any statement here.

Instead, it would be more useful to have a relation that expresses something like: Either e equals to f or f is divergent.

3.1.2 Functions with patterns

Given a function with a certain pattern, what does applying this function to something that does *not* match the pattern evaluate to?

Consider two functions $e = (\forall x : x \mapsto x)$ and $f = (\forall x, y : P\ x\ y \mapsto P\ x\ y)$: Both of them are identities, but obviously operating in different domains. Are e and f equal? Certainly not! Still, they behave somewhat similar and it would be nice to have a relation that is able to express this similarity, instead of having to write $e \neq f$.

Instead of not defining an evaluation rule for the non-matching case, we could return a special value, representing this failure of pattern matching.

3.1.3 Case expressions

Although there are no types in our language, we want case-expressions to behave as if they were exhaustive. In other words, we expect the programmer to provide a case for every argument the expression could be applied to.

Consider an expression $e = [Z \mapsto S\ Z, \forall n : S\ n \mapsto S\ (S\ n)]$. Note that e is a valid "successor" function for natural numbers defined by the data constructors Z and S. How does e differ from the expression $f = S$?

As in the previous section, the domains of e and f are very different. While f will take any argument and wrap it inside an S, e makes sure it is actually dealing with a natural number. Since this is untyped lambda calculus, e and f are just not equal, but again, we would like to express their relationship.

3.2 Idea

As we are looking for a weaker relation than equality, equality of two expressions implies that they are also in relation $\sqsubseteq$ with each other. We also want $\sqsubseteq$ to be antisymmetric in order to be able to prove equality of expressions, but using only the partial order.

We design the partial order with the idea of inducing rewrite rules replacing e with f, whenever $e \sqsubseteq f$ holds.

3.2.1 Extended language

We extend our syntax with two special expressions $\top$ ("Top") and $\bot$ ("Bottom"):

$\langle \textit{Expression'} \rangle$ ::= $\langle \textit{Expression} \rangle$ | '$\top$' | '$\bot$'

3.2.2 $\bot$

Sometimes we might be unable to determine whether an expression is divergent or not (see motivation in section 3.1.1). Keeping rewrite rules in mind, we prefer replacing a divergent expression with a normalizing one to the other way round.

We therefore introduce $\bot$ as a representative of divergent expressions and define it to be *lower than* or equal to any other expression according to $\sqsubseteq$:

$$\bot = e \quad \text{for any divergent expression } e$$
$$\bot \sqsubseteq e \quad \text{for any expression } e$$

We will also use $\bot$ as the expression a case-expression will evaluate to once pattern matching fails throughout all alternatives (see motivation in section 3.1.3). This way, the following holds:

$$[Z \mapsto S\ Z, \forall n : S\ n \mapsto S\ (S\ n)] \sqsubseteq S$$

Note how this relationship would emit a reasonable rewrite rule — assuming that the programmer designed the case-expression to be exhaustive, the resulting optimization would not change a programs semantics!

3.2.3 $\top$

We introduce $\top$ (in some sense representing failure) as an expression dual to $\bot$, being *greater than* or equal to any other expression:

$$e \sqsubseteq \top \quad \text{for any expression } e$$

As an important result, there is no expression greater than $\top$. We chose $\top$ as the expression a function will evaluate to if pattern matching fails (see motivation in section 3.1.2). This way, the following holds:

$$(\forall x : x \mapsto x) \sqsubseteq (\forall x, y : P\ x\ y \mapsto P\ x\ y)$$

Formally, both correctness of deduction rules used by our algorithm (see † in section 8.2.1) and monotonicity of expressions (see † in section 8.1.5) strongly depend on this behavior of functions.

3.2.4 Intuition

The following ways of thinking about $\top$ and $\bot$ might help to understand the partial order better.

Set theory

We define a function $[\![\cdot]\!]$ mapping an expression e to the set of all divergent expressions and equal expressions:

$$[\![e]\!] = \{e' \mid \ e' \text{ has a normal form } e'_{NF} \implies e'_{NF} = e\ \}$$

Taking $\top$ and $\bot$ into account, the function behaves as follows:

$$[\![\bot]\!] = \{e' \mid e' \text{ does not have a normal form } \}$$
$$[\![\top]\!] = \{e \mid true\}$$

Now we could redefine $\sqsubseteq$ as follows:

$$e \sqsubseteq f \iff [\![e]\!] \subseteq [\![f]\!]$$

Type theory

Let's assume we were dealing with types instead of expressions and $\sqsubseteq$ would be the "assignable to" relation. Obviously, if two types are equivalent, they are compatible for assignment.

In this system, we would give $\top$ the broadest type (something like *object* in Java/.NET), whereas $\bot$ would have to be the narrowest type there is available, the one that is more specific than all the others (although you cannot explicitly name it in Java/.NET, it would probably be the type of *null*).

3.3 Definition

We now formally introduce the partial order $\sqsubseteq$. Note how only $\top$ and $\bot$ make this relation any different from equality:

Bottom-Axiom	$\bot \sqsubseteq e$
Top-Axiom	$e \sqsubseteq \top$

The following rules are just a generalization of the rules of equality between expressions (as defined in section 2.7):

Reflexivity	$e \sqsubseteq e$
Transitivity	$e \sqsubseteq f \wedge f \sqsubseteq g \implies e \sqsubseteq g$
Constructors	$\bar{e} \sqsubseteq \bar{f} \implies C\ \bar{e} \sqsubseteq C\ \bar{f}$
Eta-Conversion	$(\forall x : e\ x \sqsubseteq f\ x) \iff e \sqsubseteq f$ (if e and f do not freely contain x)
Beta-Reduction	$e \rightsquigarrow f \implies e = f$ (see section 2.7 and 4.1)

Also, we redefine the equality relation using $\sqsubseteq$, so expressions containing $\top/\bot$ are covered:

Antisymmetry	$e \sqsubseteq f \wedge f \sqsubseteq e \implies e = f$

Note that all variables occurring free in above statements are treated as if they were universally quantified, i.e. we do not make any assumptions about their values.

We will write $e \sqsubset f$ iff $e \sqsubseteq f \wedge e \neq f$ holds.

3.4 Monotonicity and composition rule

It turns out that expressions behave monotonically increasing, i.e.

$$f \sqsubseteq g \implies e\ f \sqsubseteq e\ g$$

holds. The proof of this rule (we refer to as *Monotonicity*) is given in the appendix (section 8.1).

In this section we will show that our definition of $\sqsubseteq$ implies a new, very general rule

$$e_1 \sqsubseteq e_2 \wedge f_1 \sqsubseteq f_2 \implies e_1\ f_1 \sqsubseteq e_2\ f_2$$

which we will refer to as *Composition* rule. The proof makes use of *Eta-Conversion* and *Monotonicity*:

$$\dfrac{\dfrac{f_1 \sqsubseteq f_2}{e_1\ f_1 \sqsubseteq e_1\ f_2}\ \text{Monotonicity} \qquad \dfrac{e_1 \sqsubseteq e_2}{e_1\ f_2 \sqsubseteq e_2\ f_2}\ \text{Eta-Conversion}}{e_1\ f_1 \sqsubseteq e_2\ f_2}\ \text{Transitivity}$$

Note that this rule implies the *Monotonicity* rule, the *Constructors* rule and η-expansion (making only the reduction part of the $Eta - Conversion$ rule necessary).

4 Semantics

4.1 Evaluation

Small-step semantics (operator $\rightsquigarrow$):

Fixed-point combinator

$$\text{fix } f \rightsquigarrow f \ (\text{fix } f)$$

Top/Bottom

$$\top \ f \rightsquigarrow \top$$

$$\bot \ f \rightsquigarrow \bot$$

Methods

The following definitions will greatly simplify the evaluation rules:

$$\begin{aligned}
\Gamma_{classic}(\bar{v_i}) &\equiv \quad c = \alpha_i(\bar{v_i}) \\
\Gamma_{min}(\bar{v_i}) &\equiv \quad c \sqsubseteq \alpha_i(\bar{v_i}) \\
\Gamma_{max}(\bar{v_i}) &\equiv \quad c \sqsupseteq \alpha_i(\bar{v_i}) \\
\Psi &\equiv \quad \forall\ vars(c) \cup vars(\alpha_i) \\
\Delta_{classic} &\equiv \quad \{\bar{v_i} \mid \Psi : \Gamma_{classic}(\bar{v_i})\} \\
\Delta_{min} &\equiv \quad \{\bar{v_i} \mid \Psi : \Gamma_{min}(\bar{v_i})\} \\
\Delta_{max} &\equiv \quad \{\bar{v_i} \mid \Psi : \Gamma_{max}(\bar{v_i})\}
\end{aligned}$$

$$(\forall \bar{v_0} : \alpha_0(\bar{v_0}) \mapsto \beta_0(\bar{v_0}), ...)\ c \rightsquigarrow \prod_{i=0}^{n} \begin{cases} \beta_i(\min\ \Delta_{min}), & \text{if } \Psi : \Gamma_{min}(\top) \\ \top, & \text{if } \Psi : \neg\Gamma_{min}(\top) \end{cases}$$

$$[\forall \bar{v_0} : \alpha_0(\bar{v_0}) \mapsto \beta_0(\bar{v_0}), ...]\ c \rightsquigarrow \bigsqcup_{i=0}^{n} \begin{cases} \beta_i(\max\ \Delta_{max}), & \text{if } \Psi : \Gamma_{max}(\bot) \\ \bot, & \text{if } \Psi : \neg\Gamma_{max}(\bot) \end{cases}$$

Remarks:

- α_i and β_i are *no* language constructs but expressions that depend on the corresponding pattern alternative's variables $\bar{v_i}$.
- Infimum and supremum operators are defined below (section 4.1.2).
- The Δ-sets are not empty iff pattern matching succeeds:
  - If pattern matching succeeds, at least the substitution of $\bar{v_i}$ that *caused* pattern matching to succeed is element of the set.
  - If the set is not empty, then *also* the pattern matching substitution must be in the set (due to monotonicity of substitution, see appendix 8.1.4).

- Compare the rules of "classical" pattern matching and unification ($\Gamma_{classic}$ and $\Delta_{classic}$) with those of our language. Basically, the only difference is the use of $\sqsubseteq$ instead of $=$. For unification, we select the lowest/greatest values $\bar{v_i}$ which let pattern matching succeed. Although we cannot necessarily always find a substitution for $\bar{v_i}$ that leads to an equality of two expressions, we will still call this process *unification*.
- The reduction rules are only exhaustive if Ψ does not quantify any variables, i.e. if c and α_i are *closed* expressions. If, on the other hand, none of those two cases apply, the reduction step cannot be taken (partial knowledge). Note that this is not a blocked situation, so the expression is also not in normal form. With the substitution of a free variable, a case may suddenly apply and the reduction step could be performed.

Applications

$$(a_1\ a_2)\ c \rightsquigarrow a\ c \quad \text{iff} \quad a_1\ a_2 \rightsquigarrow a$$

4.1.1 Notation

The following operators are added for convenience:

$\rightsquigarrow^*$ is the reflexive, transitive closure of $\rightsquigarrow$.

$\leftrightsquigarrow^*$ is the symmetric, reflexive, transitive closure of $\rightsquigarrow$.

Note how the transitivity and β-reduction rule (see section 2.7) lead to $\leftrightsquigarrow^*$ being a witness of equality. However, the opposite implication is false, i.e. $\leftrightsquigarrow^*$ is *not* implied by equality. This is due to the fact that $\rightsquigarrow$ does not operate on subexpressions whereas two expressions are also considered equal if their subexpressions are equal (*Composition*) or if they are equal after being applied to an argument (*Eta-Conversion*).

Example

$$\begin{gathered}
\bot\ D \rightsquigarrow \bot \\
\Longrightarrow \\
\bot\ D \rightsquigarrow^* \bot \\
\Longrightarrow \\
\bot\ D \leftrightsquigarrow^* \bot \\
\Longrightarrow \\
\bot\ D = \bot \\
\Longrightarrow \\
C\ (\bot\ D) = C\ \bot
\end{gathered}$$

Note how the resulting equality does *not* imply

$$C\ (\bot\ D) \leftrightsquigarrow^* C\ \bot$$

since there is simply no reduction rule for either of those two expressions!

4.1.2 Infimum and supremum

We define the infimum/supremum of a set of expressions as the greatest/lowest expression (according to $\sqsubseteq$) which is lower/greater than all elements of the set.

Note that only the infimum/supremum of closed expressions is guaranteed to also be closed. Otherwise, it may depend on the substitutions for free variables. Example:

$$C \sqcup x$$

where x is a free variable and C is a data constructor. Because of C being a constructor there are only two supremum candidates to begin with: C and $\top$

The result obviously depends on the concrete value of x:

$$C \sqcup x = \begin{cases} C & \text{, if } x \sqsubseteq C \\ \top & \text{, otherwise} \end{cases}$$

This result could be expressed using a pattern matching function (see section for 4.1 semantics):

$$C \sqcup x = (C \mapsto C)\ x$$

Note that generally, expressions containing free variables may have much more complex infima/suprema, which we are not interested in. For our algorithm, we therefore restrict our efforts to reliably handle closed expressions that have a normal form.

4.2 Examples

The following examples show correct usage of the evaluation semantics.

4.2.1 Equality due to semantics

Each of the following lists represents a set of equivalent expressions from the partially ordered set:

- $\bot$, $\bot\ a$, $\bot\ \top$, $(\top \mapsto \bot)\ b$, $(\forall x : x \mapsto \bot)\ c$, $[\forall x : D\ x \mapsto x]\ ((\forall x : x \mapsto C\ x)\ e)$
- $\top$, $\top\ a$, $\top\ \bot$, $[\bot \mapsto \top]\ b$, fix $\top$
- a, $(\forall y : y \mapsto y)\ a$, $(\forall y : y \mapsto a\ y)$, $(\forall y : y \mapsto a)\ b$, $[\forall y : C\ y \mapsto y]\ (C\ a)$

4.2.2 Pattern matching

Note how our two approaches of pattern matching complement each other to behave exactly like ordinary pattern matching (and resulting unification). As a result, whenever ordinary unification succeeds, so will both of our approaches, and vice versa.

The following two examples are an immediate comparison of the different semantics in action.

a) Pattern: $\forall x : C\ x$

Argument	ordinary unification	function semantics	case semantics
$C\ A$	success $(x := A)$	success $(x := A)$	success $(x := A)$
$C\ B$	success $(x := B)$	success $(x := B)$	success $(x := B)$
D	failure	failure $(\rightsquigarrow \top)$	failure $(\rightsquigarrow \bot)$
$\bot$	failure	success $(x := \bot)$	failure $(\rightsquigarrow \bot)$
$\top$	failure	failure $(\rightsquigarrow \top)$	success $(x := \top)$
$C\ \bot$	success $(x := \bot)$	success $(x := \bot)$	success $(x := \bot)$
$C\ \top$	success $(x := \top)$	success $(x := \top)$	success $(x := \top)$

b) Pattern: $\forall x : D\ x\ x$

Argument	ordinary unification	function semantics	case semantics
$D\ \bot\ \bot$	success $(x := \bot)$	success $(x := \bot)$	success $(x := \bot)$
$D\ \top\ \top$	success $(x := \top)$	success $(x := \top)$	success $(x := \top)$
$D\ A\ A$	success $(x := A)$	success $(x := A)$	success $(x := A)$
$D\ B\ A$	failure	success $(x := \top)$	success $(x := \bot)$
$D\ \bot\ A$	failure	success $(x := A)$	success $(x := \bot)$
$D\ \top\ A$	failure	success $(x := \top)$	success $(x := A)$

The first example shows a few "classical" cases where the algorithms behave the same. The biggest difference can be observed when passing just $\top$ or $\bot$ as an argument, each behaving as wildcards in either of our semantics. Note how function- and case-semantics are dual to each other.

4.2.3 Function evaluation

We analyze the expression $(\forall v : A\ v\ v \mapsto B\ v)\ x$

x	Unification using $x \sqsubseteq (A\ v\ v)$	Result of $(\forall v : A\ v\ v \mapsto B\ v)\ x$
$A\ \top\ \top$	true, emitting $v := \top \sqcup \top = \top$	$B\ \top$
$A\ C_1\ \top$	true, emitting $v := C_1 \sqcup \top = \top$	$B\ \top$
$A\ \top\ C_1$	true, emitting $v := \top \sqcup C_1 = \top$	$B\ \top$
$A\ C_1\ C_2$	true, emitting $v := C_1 \sqcup C_2 = \top$	$B\ \top$
$A\ C_1\ C_1$	true, emitting $v := C_1 \sqcup C_1 = C_1$	$B\ C_1$
$A\ C_1\ \bot$	true, emitting $v := C_1 \sqcup \bot = C_1$	$B\ C_1$
$A\ \bot\ 1$	true, emitting $v := \bot \sqcup C_1 = C_1$	$B\ C_1$
$A\ \bot\ \bot$	true, emitting $v := \bot \sqcup \bot = \bot$	$B\ \bot$
B	false	$\top$

4.2.4 Case evaluation

We analyze the expression $[\forall x : A\ x\ x \mapsto D\ V\ x, \forall y : A\ y\ V \mapsto D\ y\ V, B \mapsto C]\ w$

w	$(A\ x\ x) \sqsubseteq w$	$(A\ y\ V) \sqsubseteq w$	$B \sqsubseteq w$	$[...]\ w$
$A\ \top\ \top$	true, em. $x := \top \sqcap \top = \top$	true, em. $y := \top$	false	$D\ V\ \top \sqcup D\ \top\ V = D\ \top\ \top$
$A\ V\ \top$	true, em. $x := V \sqcap \top = V$	true, em. $y := V$	false	$D\ V\ V \sqcup D\ V\ V = D\ V\ V$
$A\ \top\ V$	true, em. $x := \top \sqcap V = V$	true, em. $y := \top$	false	$D\ V\ V \sqcup D\ \top\ V = D\ \top\ V$
$A\ V\ W$	true, em. $x := V \sqcap W = \bot$	false	false	$D\ V\ \bot$
$A\ V\ V$	true, em. $x := V \sqcap V = V$	true, em. $y := V$	false	$D\ V\ V \sqcup D\ V\ V = D\ V\ V$
$A\ V\ \bot$	true, em. $x := V \sqcap \bot = \bot$	false	false	$D\ V\ \bot$
$A\ \bot\ V$	true, em. $x := \bot \sqcap V = \bot$	true, em. $y := \bot$	false	$D\ V\ \bot \sqcup D\ \bot\ V = D\ V\ V$
$A\ \bot\ \bot$	true, em. $x := \bot \sqcap \bot = \bot$	false	false	$D\ V\ \bot$
B	false	false	true	C
C	false	false	false	$\bot$

4.3 Relationship to the λ-calculus

It is worth noting how terms of the λ-calculus correspond to terms of our language. We introduce the transformation operator T, converting λ-calculus terms into equivalent expressions of our language.

$$\begin{aligned} T[x] &= x \\ T[e\ f] &= T[e]\ T[f] \\ T[\lambda x.e] &= (\forall x : x \mapsto T[e]) \end{aligned}$$

Remark: The reverse transformation T^{-1} will fail for any construct of our language that has no matching counterpart in the λ-calculus.

Indeed, β-reduction of the λ-calculus matches our evaluation semantics:

$$(\lambda x.e)\ f\ \leadsto_\beta\ e\{x := f\}$$

$$\begin{aligned} T[(\lambda x.e)\ f] &= T[(\lambda x.e)]\ T[f] \\ &= (\forall x : x \mapsto T[e])\ T[f] \\ &\leadsto T[e]\{x := T[f]\} \\ &= T[e\{x := f\}] \end{aligned}$$

The evaluation step we took needs to be proved:

$$\begin{aligned} \alpha_0(x) &\equiv x \\ \beta_0(x) &\equiv T[e]\{x := x\} \\ \Gamma_{min}(x) &\equiv T[f] \sqsubseteq x \\ \Psi &\equiv \forall\ vars(T[f]) \\ \Delta_{min} &\equiv \{x \mid \Psi : \Gamma_{min}(x)\} \\ (\forall x : x \mapsto T[e])\ T[f]\ &\leadsto \begin{cases} T[e]\{x := \min\ \Delta_{min}\}, & \text{if } \Psi : \Gamma_{min}(\top) \\ \top, & \text{if } \Psi : \neg\Gamma_{min}(\top) \end{cases} \end{aligned}$$

The first evaluation case applies since $\Psi : T[f] \sqsubseteq \top$ matches our axiom for $\top$. As a result, we need to calculate the unification to substitute x:

$$\begin{aligned} \min\ \Delta_{min} &= \min\ \{x \mid \Psi : \Gamma_{min}(x)\} \\ &= \min\ \{x \mid \Psi : T[f] \sqsubseteq x\} \\ &= T[f] \end{aligned}$$

The result of the evaluation is therefore $T[e]\{x := T[f]\}$.

4.4 Normal form

Applying $\rightsquigarrow$ to an expression e over and over again might at some point result in an expression f that no rule applies to, i.e. the situation is *blocked*. In that case, f is called normal form of e. Whether a normal form exists or not is undecidable.

With an expressions e reducing to an expression f implying $e \sqsubseteq f$, the partial order must be undecidable as well. Otherwise we could decide whether e has the normal form f by deciding $e \sqsubseteq f$.

Note how $\rightsquigarrow$ relies on $\sqsubseteq$ in the case of function/case-expression reduction. Therefore, the result of a single evaluation step is also undecidable.

Normal forms have the following shape:

Atomic
: Anything that is not an application of two expressions cannot be reduced further since evaluation rules only deal with applications (see section 4.1).

  Summary of expressions: Variables, data constructors, fix, $\top$, $\bot$, methods

Composite
: When dealing with an application, evaluation rules will always try to recursively reduce the left expression. This terminates if the leftmost expression is atomic *and* there is no evaluation rule applying this atomic expression to an arbitrary argument. There are such rules for fix, $\top$, $\bot$, methods, but *not* for variables or data constructors.

  Summary of expressions: Applications that have a variable (or data constructor) as its leftmost subexpression.

4.5 Multi-argument functions and currying

At a later point it will come in handy to have multi-argument functions, i.e. functions consisting of a single pattern alternative that can consume multiple arguments. Special about multiple patterns per alternative is the fact that the patterns share the same set of universally quantified variables. For example, this allows the creation of a function accepting two arbitrary but *identical* arguments:

$$(\forall a : a \mapsto a \mapsto C)$$

We therefore extend our syntax of expressions with multi-argument functions and — instead of giving their semantics explicitly — provide rules for currying them into nested single-argument functions:

$$(\Psi\ p_0 \mapsto p_1 \mapsto ... \mapsto p_n \mapsto e)$$

$$\equiv$$

$$(\forall x_0 : x_0 \mapsto (\forall x_1 : x_1 \mapsto (... (\forall x_n : x_n \mapsto (\Psi\ C\ p_0\ p_1\ ...\ p_n \mapsto e)\ (C\ x_0\ x_1\ ...\ x_n))...))$$

with (optional) universal quantification Ψ, unique data constructor C, expressions $e, p_0, p_1, ..., p_n$, variables $x_0, x_1, ..., x_n$.

Arguments are simply collected one by one, packed into a single expression and passed to a function — which has all patterns packed into a single pattern in the exact same fashion.

The special case of zero-argument functions is also covered by above definition:

$$(\Psi\ e)$$

$$\equiv$$

$$(\Psi\ C \mapsto e)\ (C)$$

which can be reduced to just e.

4.5.1 Comparison with naive currying

One might wonder why arguments are not simply collected one by one and then mapped to the RHS expression e, but are instead first packed and then pattern matched *at once*.

As mentioned above, multi-argument functions are special in terms of quantification:

$$f = (\forall a : a \mapsto a \mapsto a)$$

This multi-argument function could mistakenly be "naively" curried in one of the following two ways:

$$f_1 = (\forall a : a \mapsto (a \mapsto a))$$
$$f_2 = (\forall a : a \mapsto (\forall a : a \mapsto a))$$

Both options can be ruled out quickly: f_1 violates our language specification (more specifically, the inner function contains a variable in its pattern that is bound by a surrounding pattern alternative; see sections 2 and 8.1.7), while f_2 behaves like a constant function since its first argument has no effect.

Compare this to the correct translation using the currying rule from the previous section:

$$f = (\forall x_0 : x_0 \mapsto (\forall x_1 : x_1 \mapsto (\forall a : C\ a\ a \mapsto a)\ (C\ x_0\ x_1)))$$

4.6 Regular methods

We call a function and especially case-expression "regular" iff the patterns of its alternatives

- are in normal form.
- are constructors, applied to an arbitrary number of distinct variables bound by the very same pattern alternative (matches the definition of a *linear* pattern in [11]).
- contain pairwise disjoint data constructors.

In other words, each pattern alternative has the following shape:

$$\forall v_0, v_1, ... : C\ v_0\ v_1\ ... \mapsto e$$

with data constructor C and an arbitrary expression e.

In the following, we focus on case-expressions as their regularity will be more important to us than that of functions. However, the same reasoning applies to functions.

4.6.1 Examples

Following above definition, the following case-expressions are regular.

$$\begin{aligned}&[N \mapsto n,\\ &\forall x :\ J\ x \mapsto j\ x]\end{aligned}$$

$$\begin{aligned}&[E \mapsto E,\\ &\forall h, t :\ C\ h\ t \mapsto C\ (f\ h)\ (r\ f\ t)]\end{aligned}$$

$$\begin{aligned}&[B \mapsto \bot,\\ &T \mapsto \top,\\ &\forall v :\ V\ v \mapsto v,\\ &\forall a\ b :\ A\ a\ b \mapsto (t\ a)\ (t\ b)]\end{aligned}$$

4.6.2 Evaluation

Recall the evaluation rules of case-expressions:

$$\begin{aligned}\Gamma_{max}(\bar{v_i}) \equiv &\quad c \sqsupseteq \alpha_i(\bar{v_i})\\ \Psi \equiv &\quad \forall\ vars(c) \cup vars(\alpha_i)\\ \Delta_{max} \equiv &\quad \{\bar{v_i} \mid \Psi : \Gamma_{max}(\bar{v_i})\}\end{aligned}$$

$$[\forall \bar{v_0} : \alpha_0(\bar{v_0}) \mapsto \beta_0(\bar{v_0}), ...]\ c \rightsquigarrow \bigsqcup_{i=0}^{n} \begin{cases}\beta_i(\max\ \Delta_{max}), & \text{if } \Psi : \Gamma_{max}(\bot)\\ \bot, & \text{if } \Psi : \neg\Gamma_{max}(\bot)\end{cases}$$

If the case-expression is regular, we know that

$$\alpha_i(\bar{v_i}) = C_i\ \bar{v_i}$$

for all $0 \leq i \leq n$ and pairwise distinct C_i. As a result, $\alpha_i(\bar{v_i})$ contains no free variables, i.e. $\bar{v_i}$ are bound by the pattern alternative. Therefore Ψ universally quantifies *only* the free variables of c.

We now take a closer look at the conditions for evaluating to $\beta_i(\bar{v_i})$ or $\bot$ and will find that the result of pattern matching is now very predictable (if c is in normal form).

We assume that c is in normal form and has the following shape:

$$c = h\ e_0\ e_1\ ...$$

The pattern matching condition is therefore

$$\Gamma_{max}(\bar{v_i}) \equiv \quad h\ \bar{e} \sqsupseteq C_i\ \bar{v_i}$$

Cases of c's left-most subexpression h and resulting behavior of a pattern alternative:

Variable x

x is free inside c (it is leftmost, so it is obviously not surrounded by a pattern alternative), so it is universally quantified by Ψ. As a result, none of both cases of evaluation match! As shown below, there is *always* a substitution for x that would result in a successful pattern match, as well as a substitution that would result in a mismatch, i.e. neither of both reduction cases apply!

Substitution that would let pattern matching succeed (besides from $x := \top$):

$$x := (\forall \bar{x} : \bar{x} \mapsto C_i\ \bar{y}) \quad \text{with } \bar{x} \text{ and } \bar{e} \text{ having the same cardinality}$$

$$\begin{aligned} \Gamma_{max}(\bar{y}) \equiv &\quad (\forall \bar{x} : \bar{x} \mapsto C_i\ \bar{y})\ \bar{e} \sqsupseteq C_i\ \bar{y} \\ \equiv &\quad C_i\ \bar{y} \sqsupseteq C_i\ \bar{y} \end{aligned}$$

Substitution that would let pattern matching fail (besides from $x := \bot$):

$$x := X$$

$$\begin{aligned} \Gamma_{max}(\bar{v_i}) \equiv &\quad X\ \bar{e} \sqsupseteq C_i\ \bar{v_i} \\ \equiv &\quad X\ \bar{e} \sqsupseteq C_i\ \bar{v_i} \\ \equiv &\quad \text{false for every substitution of } \bar{v_i} \end{aligned}$$

Data constructor $X \neq C_i$

The pattern alternative will evaluate to $\bot$. We proof the necessary statement:

$$\begin{aligned} & X \neq C_i \\ \Longrightarrow\ & \neg(X \sqsupseteq C_i) \\ \Longleftrightarrow\ & \forall \bar{e}, \bar{f} :\ \neg(X\ \bar{e} \sqsupseteq C_i\ \bar{f}) \\ \Longrightarrow\ & \forall \bar{e} :\ \neg(X\ \bar{e} \sqsupseteq C_i\ \bot\ \bot\ ...) \\ \Longleftrightarrow\ & \forall \bar{e} :\ \neg(\Gamma_{max}(\bot)) \\ \Longleftrightarrow\ & \Psi :\ \neg(\Gamma_{max}(\bot)) \end{aligned}$$

Data constructor $X = C_i$

The pattern alternative will evaluate to $\beta_i(\bar{e})$ iff $\bar{v_i}$ and $\bar{e}$ have the same cardinality. In that case, the unification is $\bar{v_i} := \bar{e}$ (element-wise):

$$\dfrac{\dfrac{\dfrac{}{\Psi : C_i\ \bar{e} \sqsupseteq C_i\ \bar{e}}\ \text{Refl}}{\Psi : \Gamma_{max}(\bar{e})}\ \equiv}{\Psi : \Gamma_{max}(\bot)}\ \text{weaken (substitution } \bar{v_i} := \bar{e})$$

If, on the other hand, cardinalities of $\bar{v_i}$ and $\bar{e}$ do *not* match, the pattern alternative will again evaluate to $\bot$. This is due to the fact that mismatching cardinalities require a constructor to equal or be lower/greater than itself, applied to a number of other expressions. This is impossible:

$$\begin{aligned} &\Psi : \Gamma_{max}(\bot) \\ &\quad \Longleftrightarrow \\ &\Psi : C_i\ \bar{e} \sqsupseteq C_i\ \bot\ \bot\ ... \\ &\quad \Longleftrightarrow \text{ (because of cardinality mismatch)} \\ \Psi : C_i \sqsupseteq C_i\ \bot\ \bot\ ... \quad &\vee \quad \Psi : C_i\ e_0\ e_1\ ... \sqsupseteq C_i \end{aligned}$$

None of those two statements are true: all expressions are in normal form, equality is obviously not given and none of the axioms of the partial order apply for corresponding subexpressions.

Top $\top$

Note that $h = \top$ implies $c = \top$, since c is in normal form.

The expression will reduce to $\beta_i(\top, \top, ...)$:

$$\dfrac{\dfrac{}{\Psi : \top \sqsupseteq C_i\ \top\ \top\ ...}\ \text{Top-Axiom}}{\Psi : \Gamma_{max}(\top)}\ \equiv$$

Bottom $\bot$ **or fix**

Note that c being in normal form means that $c = h$.

The pattern alternative evaluates to $\bot$ since c (being $\bot$ or fix) can never be greater than or equal to a data constructor.

Method

Note that c being in normal form means that $c = h$.

We try to η-reduce c and restart the algorithm on success. Otherwise, the pattern alternative evaluates to $\bot$ since c cannot be η-equivalent to an expression greater or equal to the pattern.

With this behavior of pattern alternatives in mind, the overall behavior of regular case-expressions can be summarized as follows:

- If the argument is $\top$, the expression evaluates to $\bigsqcup_{i=0}^{n} \beta_i(\top, \top, ...)$.
- If the argument is a data constructor with a number of arguments and the case-expression has a corresponding pattern (i.e. same data constructor and same number of arguments), the expression evaluates to the corresponding alternative's RHS — with the unification variables trivially substituted.
- If the argument is a method in normal form, try η-reduction and restart — on failure, evaluate to $\bot$.
- If the argument is *closed* and none of above three cases apply, the expression evaluates to $\bot$.

4.7 Well-formed statements

So far we have defined all the basic building blocks our algorithm will operate on. It will try to prove the relationship between expressions using the rules of $\sqsubseteq$ and $\rightsquigarrow$.

However, we have not yet given any further meaning to free variables, i.e. variables that are *not* bound by a pattern alternative. So far, we never made any assumptions about their actual values - we treated them as unknown atomic symbols. As a result, if we could prove $e \sqsubseteq f$ with e and/or f containing a free variable x, then also $e\{x := g\} \sqsubseteq f\{x := g\}$ would hold for *any* expression g.

4.7.1 Notation

To make explicit that we are not making any assumptions about the variable, we would from now on write $\forall x : e \sqsubseteq f$, i.e. universally quantify all free variables.

On the other hand, we may want to state, that there exists a certain value for a variable x, that would make a statement containing x true. In that case we will write $\exists x : e \sqsubseteq f$.

Conclusion: All free variables must be quantified. Quantified variables are considered bound. Well-formed statements contain only bound variables.

Syntax

$\langle \textit{Relation} \rangle$ ::= $\langle \textit{Expression} \rangle$ (‘ $\sqsubseteq$ ’ | ‘ $\sqsubset$ ’ | ‘ = ’) $\langle \textit{Expression} \rangle$

$\langle \textit{Statement} \rangle$ ::= $\langle \textit{Relation} \rangle$
| $\langle \textit{QE} \rangle$ $\langle \textit{QA} \rangle$ $\langle \textit{Statement} \rangle$

5 Proving statements

5.1 Statements as input and output

The algorithm will try to find and output statements that it can prove. Also recall that it will not just search for *any* well-formed statement but for statements of a certain pattern (see section 1.2):

Output statements
: For the purpose of rewrite rules, statements with existentially quantified variables are of no use. Therefore, the algorithm will try to find proofs for statements with only universally quantified variables.

Input statements
: On the other hand, existentially quantified variables are exactly what enables us to specify patterns of statements. For every existentially quantified variable, the algorithm will try to either find a specific expression/substitution that makes the statement true or even prove the statement correct for *any* substitution, i.e. *promoting* the variable to a universally quantified one.

Thus the overall behavior of the algorithm can be formalized like this:

$$output\ (witness) \implies input\ (claim)$$

with *output* being a statement arising from *input* by having all existentially quantified variables eliminated in one or the other way.

5.2 Deduction rules

As we have now formalized our notion of statements, we can now define the concrete steps our algorithm will take in order to prove such a statement. For this purpose we now collect all the rules we have come up with so far in the context of the partial order $\sqsubseteq$ or quantification.

The goal is to have a well-defined set of deduction rules for our algorithm to choose from. This greatly simplifies reasoning about the steps the algorithm may or may not perform in a given situation.

5.2.1 Notation

For conciseness the following abbreviations for quantifiers will be used:

Ψ is an arbitrary chain of universal quantifiers.

Φ on the other hand is a chain of arbitrary universal and/or existential quantifiers.

In proofs, we will sometimes use double lines instead of single lines to separate premises from conclusion. We use double lines to separate a reduction statement $e \rightsquigarrow f$ (conclusion) from its justification (premises; pattern matching and unification).

5.2.2 Relation

The following rules directly emerge from the definition of $\sqsubseteq$ and its implications (see section 3.3 and 3.4).

$$\frac{}{\forall\ vars(e) : e \sqsubseteq e} \quad Refl$$

$$\frac{\Psi\ e_1 \sqsubseteq e_2 \qquad \Psi\ e_2 \sqsubseteq e_3 \qquad ... \qquad \Psi\ e_{n-1} \sqsubseteq e_n}{\Psi\ e_1 \sqsubseteq e_n} \quad Trans$$

$$\frac{\Psi\ e \sqsubseteq f \qquad \Psi\ f \sqsubseteq e}{\Psi\ e = f} \quad Antisymm$$

$$\frac{\Phi\ e = f}{\Phi\ f = e} \quad Symm_{=}$$

$$\frac{\Phi\ e = f}{\Phi\ e \sqsubseteq f} \quad Weaken_{=}$$

$$\frac{}{\forall\ vars(e) : \bot \sqsubseteq e} \quad \bot$$

$$\frac{}{\forall\ vars(e) : e \sqsubseteq \top} \quad \top$$

$$\frac{\Psi\ e_1 \sqsubseteq e_2 \qquad \Psi\ f_1 \sqsubseteq f_2}{\Psi\ e_1\ f_1 \sqsubseteq e_2\ f_2} \quad Comp$$

$$\frac{\Psi\ \forall x : e\ x \sqsubseteq f\ x \quad (\text{with } x \notin vars(e) \cup vars(f))}{\Psi\ e \sqsubseteq f} \quad \eta$$

$$\frac{e \rightsquigarrow f}{\forall\ vars(e) \cup vars(f) : e \sqsubseteq f} \quad \beta'$$

Note that the definition of β', $Weaken_{=}$ and $Trans$ actually make $Refl$ redundant.

5.2.3 Quantification

The following rules describe how quantifiers can be introduced, removed or transformed into each other legally. This is generally accompanied by information loss. The rules are compliant with standard first-order logic.

$$\frac{\Phi\ e \sqsubseteq f \quad (\text{with } x \notin vars(e) \cup vars(f))}{\Phi\ \forall x : e \sqsubseteq f}\ Introduce_{\forall}$$

$$\frac{\Phi\ e \sqsubseteq f \quad (\text{with } x \notin vars(e) \cup vars(f))}{\exists x : \Phi\ e \sqsubseteq f}\ Introduce_{\exists}$$

$$\frac{\Phi_1\ \forall x : \Phi_2\ f \sqsubseteq g \quad (\Phi_2 \text{ does not quantify } x,\ \Phi_1 \text{ does not quantify any of } vars(e))}{\Phi_1\ \forall\ vars(e) : \Phi_2\ f\{x := e\} \sqsubseteq g\{x := e\}}\ Weaken_{\forall}$$

$$\frac{\Phi\ f\{x := e\} \sqsubseteq g\{x := e\} \quad (\text{with } x \notin vars(e))}{\exists x : \Phi\ f \sqsubseteq g}\ Weaken_{\exists}$$

$$\frac{\Phi_1\ \exists v : \forall w : \Phi_2\ e \sqsubseteq f}{\Phi_1\ \forall w : \exists v : \Phi_2\ e \sqsubseteq f}\ Swap_{weaken}$$

$$\frac{\Phi_1\ \forall v : \forall w : \Phi_2\ e \sqsubseteq f}{\Phi_1\ \forall w : \forall v : \Phi_2\ e \sqsubseteq f}\ Swap_{\forall}$$

$$\frac{\Phi_1\ \exists v : \exists w : \Phi_2\ e \sqsubseteq f}{\Phi_1\ \exists w : \exists v : \Phi_2\ e \sqsubseteq f}\ Swap_{\exists}$$

$$\frac{\Phi_1\ \exists x : \Phi_2\ e \sqsubseteq f \quad (\Phi_2 \text{ quantifies } x \text{ or } x \notin vars(e) \cup vars(f))}{\Phi_1\ \Phi_2\ a \sqsubseteq b}\ Hide_{\exists}$$

$$\frac{\Phi_1\ \forall x : \Phi_2\ e \sqsubseteq f \quad (\Phi_2 \text{ quantifies } x \text{ or } x \notin vars(e) \cup vars(f))}{\Phi_1\ \Phi_2\ e \sqsubseteq f}\ Hide_{\forall}$$

$$\frac{\begin{array}{cc}\text{it can be used that } P(e) & \text{it can be used that } \neg P(e) \\ \vdots & \vdots \\ \Phi\ f \sqsubseteq g & \Phi\ f \sqsubseteq g\end{array}}{\Phi\ f \sqsubseteq g}\ Cases$$

5.2.4 Admissible

The following rules are all redundant and implied by already defined rules. Nevertheless, they represent meaningful steps of reasoning as we would like to use them instead of always applying all their components one by one.

Furthermore, some of them are too complex for our algorithm to infer from the existing rules. This way we provide powerful rules and at the same time take the responsibility of proving them away from the algorithm (proofs in section 8.2).

$$\frac{e \leftrightsquigarrow^* f}{\forall\ vars(e) \cup vars(f) : e = f}\ \beta$$

$$\frac{\Psi\ f\ e_0\ e_1\ ...\ e_n \sqsubseteq g}{\Psi\ f \sqsubseteq (\Psi\ e_0 \mapsto e_1 \mapsto ... \mapsto e_n \mapsto g)}\ Lift$$

$$\frac{\Psi\ f\ (\Psi\ e_0 \mapsto ... \mapsto e_n \mapsto g)\ e_0\ ...\ e_n \sqsubseteq g \qquad \begin{array}{l}\text{it can be assumed that } a_i \sqsubset \top \\ \vdots \\ \Psi\ f\ \top\ e_0\ ...\ e_n \sqsubset \top\end{array}}{\Psi\ \text{fix}\ f\ e_0\ ...\ e_n \sqsubseteq g}\ Fix$$

$$\frac{\Phi_1\ \forall x : \Phi_2\ f \sqsubseteq g}{\exists x : \Phi_1\ \Phi_2\ f \sqsubseteq g}\ Weaken_{\forall\exists}$$

$$\frac{\Phi\ e_1 \sqsubseteq f \qquad \Psi\ e_1 = e_2}{\forall\ vars(e_2) : \Phi\ e_2 \sqsubseteq f}\ Combine_1$$

$$\frac{\Phi\ f \sqsubseteq e_1 \qquad \Psi\ e_1 = e_2}{\forall\ vars(e_2) : \Phi\ f \sqsubseteq e_2}\ Combine_2$$

The following two rules only work for *regular* case-expressions (see section 4.6):

$$\frac{\Phi_1\ \Phi_2\ \bigsqcup_{i=0}^{n}(\beta_i(\top,...)\{x := \top\}) \sqsubseteq e\{x := \top\} \quad \Phi_1\ \forall \bar{v_0} : \Phi_2\ \beta_0(\bar{v_0})\{x := \alpha_0(\bar{v_0})\} \sqsubseteq e\{x := \alpha_0(\bar{v_0})\} \quad ...}{\Phi_1\ \forall x : \Phi_2\ [\forall \bar{v_0} : \alpha_0(\bar{v_0}) \mapsto \beta_0(\bar{v_0}), ...]\ x \sqsubseteq e}\ CaseCombine_\forall$$

$$\frac{\Phi_1\ \exists \bar{v_i} : \Phi_2\ \beta_i(\bar{v_i})\{x := \alpha_i(\bar{v_i})\} \sqsubseteq e\{x := \alpha_i(\bar{v_i})\}}{\Phi_1\ \exists x : \Phi_2\ [\forall \bar{v_0} : \alpha_0(\bar{v_0}) \mapsto \beta_0(\bar{v_0}), ...]\ x \sqsubseteq e}\ CaseCombine_\exists$$

Sometimes, we will simply rewrite a statement into an equivalent one (e.g. applying a substitution). To emphasize such a step within a proof, we may use the following pseudo-rule:

$$\frac{\text{statement}}{\text{equivalent statement}}\ \equiv$$

5.3 Promises emitted by successful proofs

5.3.1 Motivation

Recall the evaluation semantics of methods (see section 4.1). Apart from proving whether pattern matching succeeds (Γ-conditions), we need to calculate the minimum/maximum of the Δ-sets, should pattern matching succeed (unification).

So far, we have no strategy for calculating these values, but as the following sections will show, this information can elegantly be determined together with a successful pattern matching proof.

5.3.2 Unification using $\top$ and $\bot$

According to the pattern matching rules, we try proving:

- $\Gamma(\top, \top, ...)$ when we are afterwards looking for the minimum $\bar{v}$ satisfying $\Gamma(\bar{v})$.
- $\Gamma(\bot, \bot, ...)$ when we are afterwards looking for the maximum $\bar{v}$ satisfying $\Gamma(\bar{v})$.

To express these "deferred" intentions, we annotate $\top$ and $\bot$ with the unification variable we would afterwards like to determine (e.g. $\top_v$ or $\bot_v$). The fact that we annotate the variables has no influence on evaluation semantics or deduction rules.

We now investigate how we could extract exactly the information we are later looking for from successful proofs of $\Gamma(\top_{v_0}, \top_{v_1}, ...)$ or $\Gamma(\bot_{v_0}, \bot_{v_1}, ...)$. When looking at a proof, there are only few rules that *rely* on the fact that $\bot_v$ behaves like $\bot$ or $\top_v$ behaves like $\top$. These rules are:

$\top$

Note that a statement $e \sqsubseteq \top_v$ does not require $\top_v \equiv \top$. In other words, it would still be provable if we had used v instead of $\top_v$, as long as $e \sqsubseteq v$ holds. Each time this rule is applied, we therefore *emit* e as a lower bound for v.

$\bot$

Note that a statement $\bot_v \sqsubseteq e$ does not require $\bot_v \equiv \bot$. In other words, it would still be provable if we had used v instead of $\bot_v$, as long as $v \sqsubseteq e$ holds. Each time this rule is applied, we therefore *emit* e as an upper bound for v.

β

The evaluation rules $\bot_v\ e \rightsquigarrow \bot$ and $\top_v\ e \rightsquigarrow \top$ rely on $\top_v$ behaving like $\top$ and $\bot_v$ behaving like $\bot$. Reduction of $\bot_v\ e$ therefore requires $v = \bot$ (when assuming we used v instead of $\bot_v$ all along). The same applies for $\top$.

To pin v to $\bot/\top$, we emit corresponding upper/lower bounds.

We make the following observations:

- The third rule (β) is emitting information, that is not perfect, i.e it does not attempt to allow other values of v that might have worked to prove the statement (an example of this can be found below; section 5.3.4). In other words, information is destroyed whenever this rule is applied.

  The consequences are rather dramatic: In a case where the proof for pattern matching relies on this kind of reasoning, we cannot give the correct answer for how v

must be chosen. Unification fails although pattern matching succeeds. Unfortunately, evaluation must be terminated as the result cannot be determined. The algorithm should therefore always try to find a proof not relying on this rule.

- Note how the first two rules ($\top$ and $\bot$) give us a *precise* lower/upper bound for v for the proof to still work. There are special cases, though: When looking at a statement like $\bot_a \sqsubseteq \bot_b$, emitting $\bot$ as an upper bound for a would be wrong. Instead, we would obviously use b. In general, when emitting an upper/lower bound, subexpressions $\bot_x/\top_x$ must be substituted by x for all variables x.
- We will need to take a closer look on reliability of bounds emitted by proofs. This will be done when discussing and formalizing the details of the algorithm (see section 6.5). For now, we are satisfied with roughly knowing which rules emit precise conditions of annotated variables.

In the successful case (having found a proof with only "safe" rules applied), we can reliably work with the bounds emitted for v. Note that only lower(upper) bounds of v are emitted if we prove a statement containing $\top_v(\bot_v)$. As a consequence, there can be no conflict like having both upper *and* lower bounds for a variable — that potentially no expression can respect at the same time.

We observe multiple cases of how many bounds could be emitted for a variable v:

None
: This means that indeed any value for v works. When looking for the lowest possible value for v, we will obviously choose $\bot$ and vice versa.

Single
: In this case, the perfect value for v is the boundary itself.

Multiple
: For v to respect multiple lower bounds at once, choose the supremum of all lower bounds. By definition of the supremum, it will respect all lower bounds by being greater (or equal) whilst still being the lowest possible value to do so. The same applies to upper bounds and taking their infimum.

As it turns out, infimum and supremum operators can also be used when one or zero bounds are emitted. They will behave exactly as desired.

5.3.3 Summary

For combining pattern matching and unification, prove $\Gamma(\top_{v_0}, \top_{v_1}, ...)$ or $\Gamma(\bot_{v_0}, \bot_{v_1}, ...)$. Finding a proof means that pattern matching succeeded. Finding a proof that does not rely on $\top/\bot$-reduction emits precise bounds for unification variables. These boundaries can be combined using infimum/supremum which gives the final values of all unification variables.

5.3.4 Examples

The following examples illustrate how information gets emitted by successful proofs as described above:

a) $C\ \bot_v \sqsubseteq C\ X$

The following is a valid proof:

$$\dfrac{\dfrac{}{C \sqsubseteq C}\ Refl \quad \dfrac{\text{emit!}}{\bot_v \sqsubseteq X}\ \bot}{C\ \bot_v \sqsubseteq C\ X}\ Comp$$

This emits the single upper bound X for v.

As a result of pattern matching for case-expression evaluation, this would result in the substitution $\{v := X\}$.

b) $D\ V\ W \sqsubseteq D\ \top_v\ \top_v$

The following is a valid proof:

$$\dfrac{\dfrac{}{D \sqsubseteq D}\ Refl \quad \dfrac{\text{emit!}}{V \sqsubseteq \top_v}\ \top \quad \dfrac{\text{emit!}}{W \sqsubseteq \top_v}\ \top}{D\ V\ W \sqsubseteq D\ \top_v\ \top_v}\ Comp$$

This emits the lower bounds V and W for v.

As a result of pattern matching for function evaluation, this would result in the substitution $\{v := \top\}$ since $V \sqcup W = \top$.

c) $\bot_v\ C \sqsubseteq C$

The following is a valid proof:

$$\dfrac{\dfrac{\bot\ C \rightsquigarrow \bot \quad \text{emit!}}{\bot_v\ C \sqsubseteq \bot}\ \beta \quad \dfrac{}{\bot \sqsubseteq C}\ \bot}{\bot_v\ C \sqsubseteq C}\ Trans$$

This emits the the upper bound $\bot$ for v. However, note that this result is not valid for unification, since the proof makes use of the $\bot$-reduction rule. In other words $\bot$ might not be the greatest possible upper bound for v.

We try an alternative proof:

$$\dfrac{\dfrac{\dfrac{\text{emit!}}{\bot_v \sqsubseteq (C \mapsto C)}\ \bot \quad \dfrac{}{C \sqsubseteq C}\ Refl}{\bot_v\ C \sqsubseteq (C \mapsto C)\ C}\ Comp \quad \dfrac{(C \mapsto C)\ C \rightsquigarrow C}{(C \mapsto C)\ C \sqsubseteq C}\ \beta}{\bot_v\ C \sqsubseteq C}\ Trans$$

This emits the the upper bound $(C \mapsto C)$ for v. Indeed, this result is valid for unification, as no unsafe rules are used: $\{v := (C \mapsto C)\}$

Also note that this proof is formally incomplete: The reduction used for the β-rule uses function evaluation. Thus it would require another pattern matching proof of its own.

5.3.5 General approach

Above approach might appear complex for what it achieves. Basically we are only searching for values of variables that satisfy a given statement — this is what existentially quantified variables are used for. The subtle but far-reaching difference is in fact that we are not looking for just any values satisfying the statement, but for the lowest/greatest ones. The main problem lies with our definition of a provable statement: It lacks any way to associate such extra requirements with existentially quantified variables.

Using annotated $\top$ and $\bot$ on the other hand naturally leads to a kind of proof that respects these extra wishes by emitting bounds for annotated variables. As we will see in section 6.5.3, this approach is indeed just a special case of a procedure that we will introduce to solve a much more general problem.

5.4 Proving different types of statements

Note that so far we have only presented the building blocks for proving statements of a very certain shape ($a \sqsubseteq b$ and thus also $a = b$). Indeed, this is not the only kind of statement we are dealing with:

- To ensure that pattern matching definitely fails (when evaluating a function or case-expression), we have to disprove a well-formed statement.
- The Fix rule requires us to prove $\sqsubset$. Fortunately, this can be reduced to proving and disproving one well-formed statement, as

$$e \sqsubset f \quad \Longleftrightarrow \quad e \sqsubseteq f \ \wedge \ \neg(f \sqsubseteq e)$$

5.4.1 Approach

A similar approach as in the previous section (5.3) about promises emitted by successful proofs can be taken.

Disproving a well formed statement is indeed possible when no weakening rules have been applied (i.e. no assumptions were made), and the statement is *obviously* incorrect (e.g. different data constructors are compared). The details of this will also be discussed in the definition of the algorithm (see section 6). Basically, passing a well-formed statement to the algorithm will always emit as much information about it as possible.

5.5 Examples of true/provable statements

Proofs for the following statements can be found in the appendix (section 8.3).

$$\exists n_0, j_0 : \forall x : (\forall n : n \mapsto (\forall j : j \mapsto [N \mapsto n, \forall s : J\ s \mapsto j\ s])) \ n_0\ j_0\ x \sqsubseteq x$$

$$\exists x : \forall n_0, j_0 : (\forall n : n \mapsto (\forall j : j \mapsto [N \mapsto n, \forall s : J\ s \mapsto j\ s])) \ n_0\ j_0\ x \sqsubseteq n_0$$

$$\exists f : \forall l : \text{fix}\ (\forall r : r \mapsto (\forall g : g \mapsto [E \mapsto E, \forall h, t : C\ h\ t \mapsto C\ (g\ h)\ (r\ g\ t)])) \ f\ l \sqsubseteq l$$

$$\text{fix}\ (\forall x : x \mapsto x, \top \mapsto C) = \bot$$

$$\begin{aligned}
\text{fix}\ \bot &= \bot \\
\text{fix}\ \top &= \top \\
\text{fix}\ (\top \mapsto \bot) &= \bot \\
\text{fix}\ (\bot \mapsto \top) &= \top \\
\text{fix}\ (\top \mapsto \top) &= \top
\end{aligned}$$

6 Algorithm

Recall the overall interface of the algorithm as defined in section 5.1.

It turns out that for every proof found, we can simply transform the claim into a witness according to the assumptions about variables made in the proof. In other words, if at any point in the proof we used the rule $Weaken_{\exists}$, the existentially quantified variable needs to be substituted accordingly. Otherwise, it can be promoted.

6.1 Challenges

For building a deterministic algorithm, there are a number of challenges we have to face:

- Whether a pair of expressions is in relation $\sqsubseteq$ or $\rightsquigarrow$ is undecidable (see section 4.4). Although the algorithm works with these relations, termination must still be guaranteed.

- Finding proofs means applying deduction rules backwards. Rules like $Trans$ have infinitely many ways to apply backwards. In other words, we have to be very smart about what rules to apply, and how.

- Note that most rules only work with statements that are free of existentially quantified variables. Also, there is no rule turning existentially quantified variables into universally quantified ones.

  As a result, all proofs can be divided into two parts: Proving a statement with only universally quantified variables and then weakening it into one with existentially quantified variables.

  Since the algorithm searches for proofs starting from the conclusion, this is a bad property: a generalization of the statement (reverse application of weakening rules) would be performed *before* having a proof of the resulting statement (premise). As there are usually a lot of ways to generalize a statement, the algorithm must be careful to find those it can afterwards prove.

- It might be the case that at some point during the process, the algorithm may be unable to prove or disprove a statement (for example in the context of pattern matching). In this case, we might still be able to go on with the proof: Using conservative approximations (see section 6.7).

- It must be possible to get guarantees about the minimality/maximality of instances found for existentially quantified variables (see section 5.3).

- It must be possible to disprove a statement (see section 5.4).

- We are interested in witnesses that are as general as possible, as the resulting rewrite rules are the most powerful. Furthermore, there is no point in returning witnesses implied by another (therefore more general) witness.

6.2 Most general output

One can imagine that a valid output statement may actually imply other valid statements if they are more specific. As statements implied by another statement are redundant (in terms of acting as rewrite-rules), it is desirable to only emit the most general statement and omit the implied ones.

6.2.1 Examples

Input	Possible output	Remarks
$\exists b : \forall a : b\ a \sqsubseteq a$	$\forall a : (\forall x : x \mapsto x)\ a \sqsubseteq a$	A concrete expression for b was found.
$\exists b : \forall a : a\ b \sqsubseteq a\ b$	$\forall b, a : a\ b \sqsubseteq a\ b$	b could be promoted.
$\exists b : \forall a : a\ b \sqsubseteq a\ b$	$\forall a : a\ C \sqsubseteq a\ C$	A concrete expression for b was found. Promoting b would have been more desirable, though.

6.3 Dealing with undecidability

There are several situations where we are trying to normalize an expression (e.g. to reliably prove/disprove equality or $\sqsubseteq$). Whenever reducing expressions, we must therefore limit the number of performed reduction steps. As soon as this limit is reached, evaluation is simply aborted (without information gain). So whatever caused the algorithm to perform a reduction that was aborted cannot draw any conclusion from it.

6.4 Formalization of input and output

With all the additional things we now expect the algorithm to do (or at least try), it makes sense to review and summarize the interface of our algorithm.

6.4.1 Required features

- Try to prove the provided statement. On success: Return what to do with existentially quantified variables (promote or substitute).
- Try to disprove the provided statement.
- If desired, find the lowest/greatest value for an existentially quantified variable. We generalize the approach elaborated in section 5.3, i.e. going back to using existentially quantified variables instead of annotated $\top/\bot$. On success: Return *bounds* of existentially quantified variables *and* whether precision of those boundaries can be guaranteed.

6.4.2 Encoding

The input is a statement to be analyzed. The output on the other hand can have one of the following shapes:

Unknown
: The algorithm could neither prove nor disprove the statement.

True (+ set of bounds of variables with "precision"-flags)
The statement could be proved. Witnesses can be constructed using the bounds.

False
The statement could be disproved.

6.5 Not losing information

As mentioned several times before the key of giving promises (going beyond confirmation of a statement) is not losing information in the process of proving. This is what "losing information" means intuitively:

- Applying a weakening rule to a provable statement s might result in a statement s' that is not provable. Disproving s' is therefore not sufficient for disproving s, because we *lost* the information that we actually wanted to prove something weaker.
- Applying rules like $Trans/Comp$ can split a provable statement into statements (premises) with at least one unprovable one. In other words, we lost the information that we actually wanted to prove a combination of all those statements.

Not losing information demands usage of only logical implications, i.e. rules where the premise(s) are all implied by the conclusion. We call those rules *safe*. Formally, disproving one of the premises of a safe rule is therefore sufficient to disprove the conclusion:

$$\begin{aligned} conclusion &\implies premise_1 \wedge premise_2 \wedge ... \\ &\text{iff} \\ \neg premise_1 \vee \neg premise_2 \vee ... &\implies \neg conclusion \end{aligned}$$

6.5.1 Safe deduction rules

Let's analyze our rules in terms of their ability to preserve all information.

$Refl$, $\bot$, $\top$
These rules are **safe**, because having no premises implicitly guarantees that all premises are a logical implication of the conclusion.

$Trans$
These rules are **unsafe**. As already discussed, having to make a decision about how to split the statement is based on *assumptions* lacking proof.

$Antisymm$, $Symm_=$, η, $Introduce_\forall$, $Introduce_\exists$, $Swap_\forall$, $Swap_\exists$
These rules are **safe**, since we even have deduction rules with the opposite effect (e.g. $Weaken_=$ corresponds to $Antisymm$).

$Comp$
This rules is **safe** when operating on normal forms with constructors or universally quantified variables as their leftmost expression. Such irreducible statements are in a relationship iff they can be decomposed into subexpressions that are in the same relationship.

Counter-example:

$$\bot\ C \sqsubseteq (\forall x : x \mapsto x)\ \bot$$

β'
: This rules is **unsafe**, as the conclusion is much weaker than the premise.

$Weaken_{=}$**,** $Weaken_{\forall}$**,** $Weaken_{\exists}$**,** $Weaken_{\forall\exists}$ **and** $Swap_{weaken}$
: These rules are **unsafe**. As their names suggest, these rules have weaker conclusion than premise. The same applies to $CaseCombine_{\exists}$, as it is defined using $Weaken_{\exists}$.

$Cases$
: This rules is **safe**. Even if the predicate P is chosen so poorly that it cannot be used for proving any of the premises (this is the worst case), the premises are still provable iff the conclusion is provable.

β
: This rules is **safe**. Equality arises from our evaluation semantics, so the conclusion implies the premise. Note that safe does *not* mean, that using the rule guarantees success: It can easily be impossible for the algorithm to find out whether $a \leftrightsquigarrow^* b$ holds or not. This is only guaranteed to work if a and b are in normal form, and likely to work if they at least **have** a normal form.

$Lift$**,** $CaseCombine_{\forall}$
: This rules is **safe**. Consists only consists of safe rules (see proof in section 8.2).

Fix
: This rules is **unsafe**. Basically only covers special cases. Counter-example:

$$\text{fix } \top \sqsubseteq \top$$

is obviously true but violates the second premise.

$Combine_1$ **and** $Combine_2$
: These rules are **safe** if we already know for sure that $e_1 = e_2$. In other words, we are always allowed to simply "squeeze in" known identities.

Again, a summary of safe deduction rules: $Refl$, $Antisymm$, $Symm_{=}$, $\bot$, $\top$, $Comp$ (for normal forms), η, $Swap_{\forall}$, $Swap_{\exists}$, $Cases$, β, $Lift$, $Introduce_{\forall}$, $Introduce_{\exists}$, $Combine_1/Combine_2$ (for known identities, e.g. single reduction steps).

6.5.2 Consequences

Reviewing the challenges we are facing, above set of rules are a good start for reasoning that is free of assumptions. But the problem remains, that existentially quantified variables must somehow still be replaced/removed at the very beginning of our reasoning. The deduction rules to replace/remove them are all *unsafe* (except $Introduce_{\exists}$, but it can only remove meaningless variables).

Having to apply one of these rules at the very beginning therefore basically turns our algorithm into a promise-free state immediately.

6.5.3 Delayed weakening and global restrictions

If unsafe rules eliminate the option of disproving a statement in the same run, weakening should be delayed as long as possible. Here is an example of an incorrect expression we would like to be able to disprove:

$$\forall c : \exists a : \forall b : c\ a\ b \sqsubseteq c\ b\ a$$

Note how the composition rule *Comp* looks like a good choice, but for a very good reason it cannot be applied on statements with existentially quantified variables:

$$\dfrac{\dfrac{}{\forall c : \exists a : \forall b : c \sqsubseteq c}\,Refl \qquad \dfrac{\dfrac{\dfrac{}{\forall c : \forall b : \bot \sqsubseteq b}\,\bot}{\exists a : \forall c : \forall b : a \sqsubseteq b}\,Weaken_{\exists}\ (a := \bot)}{\forall c : \exists a : \forall b : a \sqsubseteq b}\,Swap_{weaken} \qquad \dfrac{\dfrac{\dfrac{}{\forall c : \forall b : b \sqsubseteq \top}\,\top}{\exists a : \forall c : \forall b : b \sqsubseteq a}\,Weaken_{\exists}\ (a := \top)}{\forall c : \exists a : \forall b : b \sqsubseteq a}\,Swap_{weaken}}{\forall c : \exists a : \forall b : c\ a\ b \sqsubseteq c\ b\ a}\,\text{misused } Comp$$

Suddenly the incorrect statement would be provable! This example also makes very obvious the core problem of applying rules like *Comp* (or rather *any* rule that has more than one premise like $Trans$) before weakening. One and the same existentially quantified variable a could be weakened to/substituted by *different* statements (here: $\top$ and $\bot$).

In other words, rules like *Comp*, $Trans$, *Antisymm*, Fix or *Cases* have very good reason to only work with universally quantified variables: We cannot assume that an existentially quantified variable appearing in more than one premise is witnessed by the very same substitution throughout those premises (which is a crucial requirement for joining them to one and the same variable in the conclusion).

Thus, the key for delaying weakening *beyond* the use of such rules is to remember the relationship between existentially quantified variables to prevent conflicting substitutions like above. We will from now on treat existentially quantified variables as global, so "knowledge" (like substitutions) used/gained in one part of the proof *must* be visible to all parts of the proof. This is achieved by globally storing upper and lower bounds per existentially quantified variables. These bounds indicate the range of values for the variable to satisfy the statement. We also call those bounds *restrictions*.

6.5.4 Example

Here, the statement from above ($\forall c : \exists a : \forall b : c\ a\ b \sqsubseteq c\ b\ a$) is disproved reliably using globally visible restrictions:

- Initial statement: $\forall c : \exists a : \forall b : c\ a\ b \sqsubseteq c\ b\ a$
- Initial precise restrictions: $\bot \sqsubseteq a \sqsubseteq \top$
- As both expressions are normalized, apply *Comp*:
  - Statement 1: $\forall c : \exists a : \forall b : c \sqsubseteq c$
  - Apply $Refl$
  - Statement 1 proved without unprecise assumptions.
  - Statement 2: $\forall c : \exists a : \forall b : a \sqsubseteq b$

  - Substitute $a := \bot$ (provably within bounds), because a cannot depend on b (order of quantification). New precise restrictions: $\bot \sqsubseteq a \sqsubseteq \bot$
  - Statement 2 proved without unprecise assumptions.
  - Statement 3: $\forall c : \exists a : \forall b : b \sqsubseteq a$
  - Substitute $a := \top$ (provably not within bounds).
  - Statement 3 disproved.
- Statement disproved.

Note that this very general approach of preventing early weakening also covers the requirement for reliable promises introduced in section 5.3. Using $\top/\bot$ with annotations is just a special case of delayed weakening!

6.5.5 Working with global restrictions

Existentially quantified variables are initialized unrestricted, i.e. bounds are initialized to $\bot$ and $\top$. Along with trying to prove a statement, these bounds are updated (using infimum/supremum). As long as no unsafe rules are used, these bounds are known to be precise. Updating bounds in a way that would leave no values for the corresponding variable (i.e. if $bound_{lower} \sqsubseteq bound_{upper}$ was violated) leads to immediate cancellation of the (sub-)proof. If bounds are guaranteed to be precise up to this point, the initial statement was effectively disproved.

Note that this is the unification of precision as a whole: So far we found two different goals strongly depending on precision and therefore safety of used rules: giving promises (section 5.3) and disproving (section 5.4). Now both of these goals are direct implications of restrictions and their precision:

Promises
: If restrictions are precise, simply extract the required boundary.

Disproving
: If restrictions are precise and impossible (i.e. unfulfillable by any expression), the statement is incorrect.

6.6 Internal structure

Above specifications lead to the following internal structure of the algorithm.

6.6.1 Data

- For each existentially quantified variable, there are restrictions stored. They consist of an upper and lower bound, each with a flag indicating precision.
- There needs to be a flag associated with the overall state of the proof, indicating whether any unsafe assumptions were made so far. If the current state of the proof is unsafe all subsequent updates of variable bounds are not precise. Also, trivially false statements are only a valid disproof of the original statement, if the algorithm is still in a safe state.

6.6.2 Proving strategy

Since the algorithm is meant to work deterministically we will finally discuss exactly what rule to use in which situation. Making an assumption is often connected with making a (possibly wrong) substitution for one of the existentially quantified variables. Making such a substitution is only allowed if it respects the boundaries of the corresponding variable.

Notation:

- Whenever it matters, variables are annotated with $\forall/\exists$ to indicate their quantification.
- An application in normal form is abbreviated as "ApplicationNF".
- An application having a universally quantified variable or constructor as its leftmost expression is abbreviated as "ApplicationNFS".
- An application with fix as its leftmost expression is written as "ApplicationFIX".

The following cases are stated with descending priority, which is important as they may overlap.

$\bot \sqsubseteq$ **anything**
: Use the $\bot$-rule.

anything $\sqsubseteq \top$
: Use the $\top$-rule.

Constructor/$\top$**/fix** $\sqsubseteq$ **Constructor/**$\bot$**/fix**
: Use $Refl$ if expressions match (trivial), otherwise fail.

Constructor/$\top$**/fix** $\sqsubseteq$ **Variable** or

Variable $\sqsubseteq$ **Constructor/**$\bot$**/fix**
: If the variable is universally quantified, fail. Otherwise, update the corresponding boundary using the constructor/$\top$/$\bot$/fix (precise assumption).

Variable $\sqsubseteq$ **Variable**
: Pattern: $a \sqsubseteq b$
Several cases exist:

  $a = b$ **(trivial check)**
  : Use $Refl$.

  $a_\forall \sqsubseteq b_\forall$
  : Fail.

  $a_\forall \sqsubseteq b_\exists$
  : Update lower bound of b using a if b may depend on a (precise assumption). Otherwise, update it to $\top$ (precise assumption). Recall that an existentially quantified variable may depend on variables quantified left of them.

  $a_\exists \sqsubseteq b_\forall$
  : Update upper bound of a using b if a may depend on b (precise assumption). Otherwise, update it to $\bot$ (precise assumption).

$a_\exists \sqsubseteq b_\exists$
Depending on which variable may influence the other (i.e. their order of quantification), update one upper/lower bound of one variable using the other variable (precise assumption).

anything $\sqsubseteq$ **Method** or

Method $\sqsubseteq$ **anything**
Use η combined with *Cases* (so results of unification are already known).

ApplicationNFS $\sqsubseteq$ **ApplicationNFS**
Use *Comp*.

ApplicationNFS $\sqsubseteq$ **Constructor/**$\bot$**/fix** or

Constructor/$\top$**/fix** $\sqsubseteq$ **ApplicationNFS**
Fail.

ApplicationNF $\sqsubseteq$ **Variable** or

Variable $\sqsubseteq$ **ApplicationNF**
If the variable is universally quantified and the application does not have a variable as its leftmost subexpression, fail.

Otherwise: Try substituting the applications leftmost variable with functions extracting the variable from the applications subexpressions, e.g.:

- $(\forall x : x \mapsto x)$
- $(\forall \bar{x} : x_0 \mapsto x_1 \mapsto ... \mapsto x_i)$
- $(\forall x : x \mapsto x\ y)$ with fresh y
- There are infinitely many possibly useful substitutions. Our algorithm will try above substitutions, but any implementation trying additional substitutions is valid.

Otherwise: If the application contains any variable that the variable cannot depend on, fail.

Otherwise: Update the boundary of the variable using the application (precise assumption).

ApplicationNF $\sqsubseteq$ **ApplicationNF**
Assumptions will be made! Several strategies come to mind:

- Use *Comp*.
- Try substituting the leftmost variables with functions extracting subexpressions of the applications (see previous case for heuristics).
- Substitute the leftmost variables with functions trivially making the statement true (e.g. $\bot$, $\top$, constant methods).

ApplicationNF $\sqsubseteq$ **Constructor/**$\bot$**/fix** or

Constructor/$\top$**/fix** $\sqsubseteq$ **ApplicationNF**
Assumptions will be made! Several strategies come to mind:

- Try substituting the leftmost variable with functions extracting the constructor/$\bot$/fix from subexpressions of the application.
- Substitute the leftmost variables with functions trivially making the statement true.
- Use *Comp*.
- Try substituting the leftmost variables with functions extracting subexpressions of the applications.
- Substitute the leftmost variables with functions trivially making the statement true.

ApplicationFIX $\sqsubseteq$ anything
Assumptions will be made! Try using Fix: If existentially quantified variables are present, treat them as universally quantified (hoping for a promotion of the variables) when it comes to applying the Fix-rule. Should they, unfortunately, be substituted at a later point, the proof must be restarted from at the Fix-rule, but with the substitution applied. This is due to the fact that Fix has different premises depending on the set of universally quantified variables (see duplication of Ψ in section 5.2.4).

If all this fails, go to the next case.

Application $\sqsubseteq$ anything or

anything $\sqsubseteq$ Application
Try using $Combine_1/Combine_2/CaseCombine_\forall/CaseCombine_\exists$. This is also the place for making conservative approximations if none of the above rules work out (see section 6.7). If nothing succeeds, the correctness of the statement remains unknown.

6.7 Conservative approximations

In some situations, our algorithm might be unable to apply a safe rule for various reasons. For example, an expression may obviously be reducible, but we cannot perform the reduction step because required conditions (e.g. pattern matching) cannot be proved:

$$(C \mapsto C\ X, \top \mapsto C\ \bot)\ (\text{fix}\ (...)) \sqsubseteq C\ Y$$

The left expression is obviously a reducible application, but we cannot decide whether pattern matching succeeds for the first pattern alternative, i.e. if (fix (...)) $\sqsubseteq C$. Whenever facing evaluation, we can make conservative approximations about the outcome of the reduction step. As the evaluation is happening on the left side of "$\sqsubseteq$", a valid conservative approximation of the result would be anything greater than or equal to the actual result. Example of an approximation:

$$(C \mapsto C\ X, \top \mapsto C\ \bot)\ (\text{fix}\ (...)) \sqsubseteq \top$$

leading to

$$\top \sqsubseteq C\ Y$$

Formally, performing a conservative approximation is applying the $Trans$ rule to the statement in question:

$$\frac{(C \mapsto C\ X, \top \mapsto C\ \bot)\ (\text{fix}\ (...)) \sqsubseteq \top \qquad \top \sqsubseteq C\ Y}{(C \mapsto C\ X, \top \mapsto C\ \bot)\ (\text{fix}\ (...)) \sqsubseteq C\ Y}\ Trans$$

While this approximation was obviously too conservative ($\top \sqsubseteq C\ Y$ cannot be proved), there exists a better approximation:

$$(C \mapsto C\ X, \top \mapsto C\ \bot)\ (\text{fix}\ (...)) \sqsubseteq C\ \bot$$

leading to

$$C\ \bot \sqsubseteq C\ Y$$

How do we know about this approximation? Recall that function evaluation calculates the infimum of the results from each of its alternatives. The second alternative's pattern obviously matches any argument, so the overall evaluation result is the infimum of an unknown value and $C\ \bot$. Therefore, the result of evaluation must be lower than or equal to $C\ \bot$.

Remark: We already found out that $Trans$ is an unsafe rule (see section 6.5.1). Above example is a witness of this: The premises were chosen poorly so $\top \sqsubseteq C\ Y$ turning out wrong is no disproof of the original statement.

6.8 Example of proving

We will prove the following statement:

$$\exists n_0, j_0 : \forall x : (\forall n : n \mapsto (\forall j : j \mapsto [N \mapsto n, \forall s : J\ s \mapsto j\ s]))\ n_0\ j_0\ x \sqsubseteq x$$

We start the algorithm, initializing the boundaries of n_0 and j_0 with $\bot$ and $\top$ respectively. We are in a safe state. The following steps are performed:

Pattern: Application $\sqsubseteq$ anything

$Combine_1$ can be applied:

$$f_1 \equiv (\forall j : j \mapsto [N \mapsto n, \forall s : J\ s \mapsto j\ s])$$
$$f_2 \equiv (\forall j : j \mapsto [N \mapsto n_0, \forall s : J\ s \mapsto j\ s])$$

$$\frac{\exists n_0, j_0 : \forall x : f_2\ j_0\ x \sqsubseteq x \qquad \dfrac{\dfrac{\dfrac{\dfrac{\text{see 6.8.1}}{(\forall n : n \mapsto f_1)\ n_0 \rightsquigarrow f_2}}{(\forall n : n \mapsto f_1)\ n_0\ j_0 \rightsquigarrow f_2\ j_0}}{(\forall n : n \mapsto f_1)\ n_0\ j_0\ x \rightsquigarrow f_2\ j_0\ x}}{\forall n_0, j_0, x : (\forall n : n \mapsto f_1)\ n_0\ j_0\ x = f_2\ j_0\ x}\ \beta}{\exists n_0, j_0 : \forall x : (\forall n : n \mapsto f_1)\ n_0\ j_0\ x \sqsubseteq x}\ Combine_1$$

Pattern: Application $\sqsubseteq$ anything

$Combine_1$ can be applied:

$$f_1 \equiv [N \mapsto n_0, \forall s : J\ s \mapsto j\ s]$$
$$f_2 \equiv [N \mapsto n_0, \forall s : J\ s \mapsto j_0\ s]$$

$$\dfrac{\exists n_0, j_0 : \forall x : f_2\ x \sqsubseteq x \qquad \dfrac{\dfrac{\dfrac{\text{analogous to 6.8.1}}{(\forall j : j \mapsto f_1)\ j_0 \rightsquigarrow f_2}}{(\forall j : j \mapsto f_1)\ j_0\ x \rightsquigarrow f_2\ x}}{\forall n_0, j_0, x : (\forall j : j \mapsto f_1)\ j_0\ x = f_2\ x}\,\beta}{\exists n_0, j_0 : \forall x : (\forall j : j \mapsto f_1)\ j_0\ x \sqsubseteq x}\,Combine_1$$

Pattern: Application $\sqsubseteq$ anything

$CaseCombine_\forall$ can be applied:

$$\dfrac{\exists n_0, j_0 : \forall s : j_0\ s \sqsubseteq J\ s \qquad \exists n_0, j_0 : n_0 \sqsubseteq N}{\exists n_0, j_0 : \forall x : [N \mapsto n_0, \forall s : J\ s \mapsto j_0\ s]\ x \sqsubseteq x}\,CaseCombine_\forall$$

Pattern: Variable $\sqsubseteq$ Constructor/$\bot$/fix

Statement:

$$\exists n_0, j_0 : n_0 \sqsubseteq N$$

Update upper bound of n_0 to: $\top \sqcap N = N$

Pattern: ApplicationNF $\sqsubseteq$ ApplicationNF

Assumptions! We are now in an unsafe state. Statement:

$$\exists n_0, j_0 : \forall s : j_0\ s \sqsubseteq J\ s$$

Strategy: $Comp$

$$\dfrac{\exists n_0, j_0 : \forall s : j_0 \sqsubseteq J \qquad \exists n_0, j_0 : \forall s : s \sqsubseteq s}{\exists n_0, j_0 : \forall s : j_0\ s \sqsubseteq J\ s}\,Comp$$

Pattern: Variable $\sqsubseteq$ Variable

Statement:

$$\exists n_0, j_0 : \forall s : s \sqsubseteq s$$

Case: $a = b$

$$\dfrac{}{\exists n_0, j_0 : \forall s : s \sqsubseteq s}\,Refl$$

Pattern: Variable $\sqsubseteq$ Constructor/$\bot$/fix

Statement:

$$\exists n_0, j_0 : \forall s : j_0 \sqsubseteq J$$

Update upper bound of j_0 to: $\top \sqcap J = J$

6.8.1 Evaluation

We need to decide what the following obviously reducible expression reduces to:

$$(\forall n : n \mapsto (\forall j : j \mapsto [N \mapsto n, \forall s : J\ s \mapsto j\ s]))\ n_0$$

Trying to prove successful pattern match:

$$\forall n_0 : \exists n : (n_0 \sqsubseteq n\ \wedge\ \forall m : (n_0 \sqsubseteq m\ \Longrightarrow\ n \sqsubseteq m))$$

This can be achieved when finding a precise lower bound for n when proving

$$\forall n_0 : \exists n : n_0 \sqsubseteq n$$

This statement has the pattern **Variable** $\sqsubseteq$ **Variable** with subcase $a_\forall \sqsubseteq b_\exists$. As a result, the lower bound of n is updated to $\bot \sqcup n_0 = n_0$.

No assumptions were made, the algorithm was still in a safe state when updating the lower bound of n so the lower bound n_0 is guaranteed to be precise. Pattern matching was therefore successful and the substitution $\{n := n_0\}$ is applied to the pattern alternative's RHS

$$(\forall j : j \mapsto [N \mapsto n, \forall s : J\ s \mapsto j\ s])$$

leading to

$$(\forall j : j \mapsto [N \mapsto n_0, \forall s : J\ s \mapsto j\ s])$$

6.8.2 Check restrictions

Restrictions:

$$\bot \underset{(precise)}{\sqsubseteq} n_0 \qquad n_0 \underset{(precise)}{\sqsubseteq} N$$

$$\bot \underset{(precise)}{\sqsubseteq} j_0 \sqsubseteq J$$

Resulting statements to prove:

$$\bot \sqsubseteq N$$

$$\bot \sqsubseteq J$$

Obviously true. Boundaries are valid.

6.9 Example of disproving

We will disprove the following statement:

$$\forall a : \exists x : P\ x\ Y \sqsubseteq P\ a\ x$$

We start the algorithm, initializing the boundaries of x with $\bot$ and $\top$ respectively. We are in a safe state. The following steps are performed:

Pattern: ApplicationNFS $\sqsubseteq$ ApplicationNFS

Statement:

$$\forall a : \exists x : P\ x\ Y \sqsubseteq P\ a\ x$$

Comp can be applied:

$$\frac{\forall a : \exists x : P \sqsubseteq P \qquad \forall a : \exists x : x \sqsubseteq a \qquad \forall a : \exists x : Y \sqsubseteq x}{\forall a : \exists x : P\ x\ Y \sqsubseteq P\ a\ x}\ Trans$$

Pattern: Constructor/$\top$/fix $\sqsubseteq$ Constructor/$\bot$/fix

Statement:

$$\forall a : \exists x : P \sqsubseteq P$$

Refl can be applied:

$$\frac{}{\forall a : \exists x : P \sqsubseteq P}\ Refl$$

Pattern: Variable $\sqsubseteq$ Variable

Statement:

$$\forall a : \exists x : x \sqsubseteq a$$

Case: $a_\exists \sqsubseteq b_\forall$ Update upper bound of x using a since x may depend on a. New upper bound: $\top \sqcap a = a$

Pattern: Constructor/$\top$/fix $\sqsubseteq$ Variable

Statement:

$$\forall a : \exists x : Y \sqsubseteq x$$

Update lower bound of x using Y. New lower bound: $\bot \sqcup Y = Y$

6.9.1 Check restrictions

Restrictions:

$$Y \underset{(precise)}{\sqsubseteq} x \underset{(precise)}{\sqsubseteq} a$$

Resulting statements to prove:

$$\forall a : Y \sqsubseteq a$$

Pattern: Constructor/$\top$/fix $\sqsubseteq$ Variable

Fail, because a is universally quantified. Proof was in safe state, so statement is disproved.

The restriction is precise but impossible. The overall statement is disproved.

6.10 Finding rewrite rules

Recall that our main goal is finding rewrite rules of the following form (see section 1.2) for given definition of f:

$$f\ ...\ e\ ... = e$$

As it turned out, we will really search for proofs of

$$f\ ...\ e\ ... \sqsubseteq e$$

as it allows us to reason about things that we cannot have full knowledge about (see section 3.1).

6.10.1 Strategy

Given a function definition f, we will also need the arity n of the function. Then we will execute the algorithm against the following statements:

$$\exists a_0, a_1, ..., a_n : f\ a_0\ a_1\ ...\ a_n \sqsubseteq a_0$$
$$\exists a_0, a_1, ..., a_n : f\ a_0\ a_1\ ...\ a_n \sqsubseteq a_1$$
$$.$$
$$.$$
$$.$$
$$\exists a_0, a_1, ..., a_n : f\ a_0\ a_1\ ...\ a_n \sqsubseteq a_n$$

6.10.2 Interpretation of results

After getting results back from our algorithm for above statements, we have to interpret them to create statements with only universally quantified variables (i.e. rewrite rules).

For each existentially quantified variable:

- If it's boundaries are untouched (i.e. still $\bot$ and $\top$), promote it to a universally quantified variable.
- Otherwise, choose an arbitrary expression e respecting the boundaries and substitute the variable with e.

6.10.3 Relevant output

Recall that we introduced $\top/\bot$ (together with $\sqsubseteq$, see section 3) to enable reasoning about expressions we do not have full knowledge about. The source language is likely to have no equivalent counterpart for these expressions.

In other words: We have no use for output statements containing $\top$ or $\bot$, whereas both are perfectly legal to appear during calculations *leading* to that output.

7 Conclusion

To wrap everything up, we will now evaluate the capabilities of our algorithm and discuss future directions.

7.1 Evaluation

Our algorithm is capable of finding `fmap id` $\sqsubseteq$ `id` (as motivated in section 1.1) for arbitrary functor instances, as long as they do not rely on other function definitions we are not aware of. It has limits, though, which are caused by both the undecidability of some statements and the algorithm using only heuristics in some situations.

7.1.1 Functors

We are given an arbitrary functor instance (using Haskell syntax):

```
data F t = A_0 ā_0 | A_1 ā_1 | ...
```

with the types $\bar{a}_i$ being an arbitrary combination (with cardinality $c_i \in \mathbb{N}_0$) of `t` or functor instances parameterized by `t`. The corresponding `fmap` function should then be defined like this:

```
fmap f (A_0 x_0_0 x_0_1 ...)  = A_0 γ(x_0_0) γ(x_0_1) ...
fmap f (A_1 x_1_0 x_1_1 ...)  = ...
...
```

$$\text{with } \gamma(\texttt{x\_i\_j}) = \begin{cases} \texttt{f x\_i\_j} & \text{, if } a_{i,j} = \texttt{t} \\ \texttt{fmap f x\_i\_j} & \text{, otherwise} \end{cases}$$

Note that the function `fmap` used here references a *different* function, depending on the type $a_{i,j}$ of `x_i_j`. We do not know their definition, except for $a_{i,j}$ being of type `F t`, as we are currently looking at the corresponding function. We therefore assume that each $a_{i,j}$ is either `t` or `F t`. In the following, we translate `fmap` into our language using a regular case-expression and anonymous recursion:

$$\begin{aligned} fmap \equiv \text{fix } (\forall r : r \mapsto & \\ (\forall f : f \mapsto & \ [\ \forall \bar{x_0} : A_0\ \bar{x_0} \mapsto A_0\ \gamma'(x_{0,0})\ \gamma'(x_{0,1})\ ... \\ & \ ,\ \forall \bar{x_1} : A_1\ \bar{x_1} \mapsto A_1\ ... \\ & \ ,\ ...\ ])) \end{aligned}$$

$$\text{with } \gamma'(x_{i,j}) = \begin{cases} f\ x_{i,j} & \text{, if } a_{i,j} = \texttt{t} \\ r\ f\ x_{i,j} & \text{, if } a_{i,j} = \texttt{F t} \end{cases}$$

Our first goal is to prove $fmap\ (\forall x : x \mapsto x) \sqsubseteq (\forall x : x \mapsto x)$ using our deduction rules. Afterwards we will show that not only is our algorithm able to perform this proof but will also *find* the statement when given the pattern

$$\exists a, b : fmap\ a\ b \sqsubseteq b$$

(see section 6.10 for discussion). Note that the resulting witness would be

$$\forall e : fmap\ (\forall x : x \mapsto x)\ e \sqsubseteq e$$

, which is η-equivalent to

$$fmap\ (\forall x : x \mapsto x) \sqsubseteq (\forall x : x \mapsto x)$$

.

Proof

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\ldots \sqsubseteq \top}\top \quad \dfrac{}{\forall \bar{x_0} : A_0\ \bar{x_0} \sqsubseteq A_0\ \bar{x_0}}Refl}{\forall b : [\forall \bar{x_0} : A_0\ \bar{x_0} \mapsto A_0\ \bar{x_0}, \ldots]\ b \sqsubseteq b}\ldots\ CaseCombine_\forall \quad \text{(see below)}}{\forall b : (\forall r : r \mapsto \ldots)\ (\forall b : (\forall x : x \mapsto x) \mapsto b \mapsto b)\ (\forall x : x \mapsto x)\ b \sqsubseteq b}\ Combine_1 \quad (omitted)}{\dfrac{\forall b : fmap\ (\forall x : x \mapsto x)\ b \sqsubseteq b}{\exists a, b : fmap\ a\ b \sqsubseteq b}\ Weaken_{\forall\exists}, Weaken_\exists}\ Fix$$

We now proof the equality used for the $Combine_1$ rule:

$$\begin{aligned} & (\forall r : r \mapsto \ldots)\ (\forall b : (\forall x : x \mapsto x) \mapsto b \mapsto b)\ (\forall x : x \mapsto x)\ b \\ \rightsquigarrow^* & [\ \forall \bar{x_0} : A_0\ \bar{x_0} \mapsto A_0\ \gamma'(x_{0,0})\ \gamma'(x_{0,1})\ \ldots \\ & ,\ \forall \bar{x_1} : A_1\ \bar{x_1} \mapsto A_1\ \ldots \\ & ,\ \ldots\ ] \end{aligned}$$

$$\text{with } \gamma'(x_{i,j}) = \begin{cases} (\forall x : x \mapsto x)\ x_{i,j} & \text{, if } a_{i,j} = \texttt{t} \\ (\forall b : (\forall x : x \mapsto x) \mapsto b \mapsto b)\ (\forall x : x \mapsto x)\ x_{i,j} & \text{, if } a_{i,j} = \texttt{F t} \end{cases} = x_{i,j}$$

This results in the expression $[\forall \bar{x_0} : A_0\ \bar{x_0} \mapsto A_0\ \bar{x_0}, \ldots]$.

Algorithm

Input statement:

$$\exists a, b : fmap\ a\ b \sqsubseteq b$$

Note that we are initially in an "**ApplicationFIX $\sqsubseteq$ anything**" situation (see section 6.6.2), so there will be assumptions made. As a result, we are in an unsafe state and unable to disprove the statement or make precise guarantees — fortunately, neither is required in our case.

We start the algorithm, initializing the boundaries of a and b with $\bot$ and $\top$ respectively. According to the situation we are in, we try using Fix. Note that we *assume* that a and b are universally quantified and will need to come back to this point (†), should this assumption turn out to be wrong.

Now we need to prove:

$$\exists a, b : (\forall r : r \mapsto ...)\ (\forall a, b : a \mapsto b \mapsto b)\ a\ b \sqsubseteq b$$
$$\exists a, b : (\forall r : r \mapsto ...)\ \top\ a\ b \sqsubset \top$$

We omit the proving process of the second statement as it is trivial (reduce the left-hand side using $Combine_1$ and apply axioms — the algorithm will come up with that). Due to its shape (**Application** $\sqsubseteq$ **anything**), the first statement will be processed using $Combine_1$, resulting in:

$$\exists a, b : [\ \forall \bar{x_0} : A_0\ \bar{x_0} \mapsto A_0\ \gamma'(x_{0,0})\ \gamma'(x_{0,1})\ ...]\ b \sqsubseteq b$$

$$\text{with } \gamma'(x_{i,j}) = \begin{cases} a\ x_{i,j} & \text{, if } a_{i,j} = \texttt{t} \\ (\forall a, b : a \mapsto b \mapsto b)\ a\ x_{i,j}\ =\ x_{i,j} & \text{, if } a_{i,j} = \texttt{F t} \end{cases}$$

Due to its shape (**Application** $\sqsubseteq$ **anything**), this statement will be processed using $CaseCombine_\forall$ (note that the algorithm will also try using $Combine_1$ or $Combine_2$ and possibly find proofs for different statements! We will focus on the execution path leading to our desired result), resulting in:

$$... \sqsubseteq \top$$
$$\forall \bar{x_0} : A_0\ \bar{x_0} \sqsubseteq A_0\ \gamma'(x_{0,0})\ \gamma'(x_{0,1})\ ...$$
$$...$$

The first statement will be processed using the $\top$-rule, the remaining ones using $Comp$:

$$\forall \bar{x_i} : A_i \sqsubseteq A_i$$
$$\forall \bar{x_i} : x_{i,j} \sqsubseteq \gamma'(x_{i,j})$$

The first set of statements will be processed using $Refl$, the remaining ones depend on the types $a_{i,j}$:

$a_{i,j} = \texttt{F t}$
: The statement $(\forall \bar{x_0} : x_{i,j} \sqsubseteq x_{i,j})$ will be processed using $Refl$.

$a_{i,j} = \texttt{t}$
: The statement $(\forall \bar{x_0} : x_{i,j} \sqsubseteq a\ x_{i,j})$ is of the shape **Variable** $\sqsubseteq$ **ApplicationNF**. As a result, a is being substituted with a function extracting $x_{i,j}$ from its arguments: $a := (\forall x : x \mapsto x)$. Afterwards, $Refl$ can be applied.

Remark: We are done at this point if `F` does not make use of its type argument `t` a single time (i.e. there is no $a_{i,j} = \texttt{t}$). In that case, both a and b can be promoted and the witness $\forall a, b : fmap\ a\ b \sqsubseteq b$ is emitted. In the following, we assume that there *is* an $a_{i,j} = \texttt{t}$.

The substitution $a := (\forall x : x \mapsto x)$ violates the assumption we took back when applying Fix (see †), so we need to restart the algorithm from that point, but having the substitution applied. This time, using Fix requires us to prove

$$\exists b : (\forall r : r \mapsto ...)\ (\forall b : (\forall x : x \mapsto x) \mapsto b \mapsto b)\ (\forall x : x \mapsto x)\ b \sqsubseteq b$$

which will succeed. b will be promoted and therefore

$$\forall b : fmap\ (\forall x : x \mapsto x)\ b \sqsubseteq b$$

will be emitted as a witness.

We omit this part of the algorithms behavior as it is entirely analogous to the first run. After applying $Comp$, all the statements will be provable using $Refl$ this time.

Summary

It is worth noting how the proof tree we found before could also be constructed looking at the algorithms execution: Ignoring the weakening rules for a moment, the algorithm traversed and processed the statement in the exact opposite order we applied the rules in our proof (Fix first, axioms and $Refl$ last). Weakening (or rather the opposite) occurred in our algorithm in the form of promoting b after no restrictions were made (corresponds to $Weaken_{\forall\exists}$) and substituting a in order to prove some subexpression (corresponds to $Weaken_{\exists}$).

Also, the bounds of a and b were never touched, because no rule leading to an update was used.

7.1.2 λ-calculus

Recall how ordinary λ-calculus terms correspond to terms of our language (see section 4.3). Suppose we apply our algorithm to a statement, with expressions obtained from λ-calculus terms. We would transform back returned witness statements, or drop them, should the transformation fail.

We will now shortly investigate the shape of witnesses and discuss in which cases the reverse transformation will fail:

Starting from a statement that contains only expressions that can be transformed back to λ-calculus, our algorithm will never introduce case-expressions (as there is no deduction rule *introducing* a case-expression in one of its premises). Likewise, data constructors or fix will never be introduced by any rule. Note that Fix is the only rule introducing a pattern function — it will never be applied though, as fix will never appear in any expression. All in all, no methods with multiple pattern alternatives are introduced, so the only kind of method that could appear in a witness is the one matching λ-abstractions.

This already leaves us with only two language features that we cannot transform back: $\bot$ and $\top$. They may emerge as an evaluation result of pattern methods, using Fix or using $CaseCombine_{\forall}$. As neither of these situations can occur (according to our previous reasoning), they will not be introduced.
Note that witnesses containing $\bot$ or $\top$ would not be returned anyway (see section 6.10.3). Still, it is worth noting that applying deduction rules (backwards) will not introduce them in the first place.

To sum up, our algorithm is qualified to effectively operate on statements originating from the λ-calculus, as it will never make use of additional features of our dialect. This

reflects the fact that our semantics are consistent with those of the λ-calculus and that the language is therefore a proper extension of it.

7.1.3 Limitations

Our approach has a number of limitations and weaknesses we will describe in this section.

Finding substitutions

In some situations, the algorithm is forced to make a substitution for an existentially quantified variable, e.g. for statements like

$$\exists a : \forall b : b \sqsubseteq a\ b$$

(see section 6.6.2).

With such substitution being an assumption, we are not only eliminating our ability to disprove the statement reliably (should the subsequent statement be disproved), but also cannot ensure that it leads to a successful proof (should the original statement be provable).

In above statement, our algorithm will (among other things) substitute a with $(\forall x : x \mapsto x)$, leading to a successful proof.

In contrast,

$$\exists a : \forall b, c : a\ b\ (\forall x : x \mapsto c) \sqsubseteq a\ c\ (\forall x : x \mapsto x)$$

is true for a substitution like

$$a := (\forall x, y : x \mapsto y \mapsto y\ x)$$

whereas our algorithm (recall strategy for $\mathbf{ApplicationNF} \sqsubseteq \mathbf{ApplicationNF}$) will just come up with $a := \bot$, $a := \top$ and $a := (\forall x, y : x \mapsto y \mapsto c)$ for some constant c.

Disproving statements

There are a number of rules that put the algorithm in an unsafe state (one of them being Fix, as seen in the previous section about functors). As a result, we are often unable to reliably disprove a given statement.

Certainly, more effort could be put into disproving statements (e.g. an approach similar to stuck theory $\mathbb{T}_{stk}$ introduced in [8]). We put more effort into proving than disproving due to the fact that the latter is rarely required:

The algorithm is required to disprove a well-formed statement in the context of method evaluation and Fix (to prove $a \sqsubseteq b$). Method evaluation can only fail when dealing with methods with patterns.

Functions with patterns are rare (recall our main motivation to introduce them in the first place; section 2.2) and particularly occur in the context of applying the Fix rule to a statement. Fortunately, in this context we are mainly interested in reliably deciding that patterns *do* match (see typical example in section 7.1.1 about functors).

Case-expressions with patterns, on the other hand, are nothing uncommon. The admissible $CaseCombine$ rules help with this issue, as they cover common cases of statements containing case-expressions (they take away the responsibility of pattern matching proofs entirely; see section 5.2.4).

Termination

The fact that we deal with undecidability by aborting evaluation after a number of steps (see section 6.3) bears the risk of aborting the evaluation of an expression with normal form.

In other words, our approach of enforcing termination of the algorithm may cause it to fail proving an otherwise provable statement.

Universality

The algorithm we defined in this work is not very universal in its proving capabilities, but instead influenced a lot by our original goal to find rewrite rules. An example of this is our treatment of statements containing fix:

We have not defined a deduction rule for dealing with a statement having fix in its right-hand side. In other words, there is no equivalent of Fix (although we could define one, Fix is admissible after all).

Another example is our extra treatment of case-expressions in the form of $CaseCombine$-rules, although the same reasoning would apply to functions — case-expressions are simply the more common case.

7.2 Future Work

As it was the main motivation of this work, one of our goals is to implement this algorithm as an extension of GHC. The algorithm would be executed for every compiled function, emitting the corresponding rewrite rules in the right place (for future compilations to find and use them).

We are also investigating the ways a type system could benefit our analysis. Important aspects are the early rejection of impossible statements or type driven heuristics (e.g. when searching for a substitution for an existentially quantified variable as in section 7.1.3).

Other than that, this work might turn out useful in any scenario that requires estimation of the behavior of expressions involving guarded recursion, pattern matching and unknown or divergent subexpressions.

8 Appendix

8.1 Monotonicity

We now prove the *Monotonicity* rule (introduced in section 3.4):

$$f \sqsubseteq g \implies e\ f \sqsubseteq e\ g$$

We will do so via induction on the expression e:

8.1.1 Data constructor

Constructors obey the rule by definition (*Constructor* rule, see section 3.3).

8.1.2 Top/Bottom

$$\dfrac{\dfrac{\bot\ e \rightsquigarrow \bot}{\bot\ e \sqsubseteq \bot}\,\beta \qquad \dfrac{\bot\ f \rightsquigarrow \bot}{\bot \sqsubseteq \bot\ f}\,\beta}{\bot\ e \sqsubseteq \bot\ f}\,Trans$$

$$\dfrac{\dfrac{\top\ e \rightsquigarrow \top}{\top\ e \sqsubseteq \top}\,\beta \qquad \dfrac{\top\ f \rightsquigarrow \top}{\top \sqsubseteq \top\ f}\,\beta}{\top\ e \sqsubseteq \top\ f}\,Trans$$

8.1.3 Fixed-point combinator

We prove that

$$\text{fix} = (\forall f : f \mapsto (\forall x : x \mapsto f\ (x\ x))\ (\forall x : x \mapsto f\ (x\ x)))$$

. As a result, the monotonicity of fix is implied by the monotonicity of functions (see section 8.1.5).

$$\dfrac{\dfrac{\begin{array}{c}\forall f : f\ (f\ (f\ ...)) = f\ (f\ (f\ ...)) \\ \vdots \\ \forall f : \text{fix}\ f = (\forall x : x \mapsto f\ (x\ x))\ (\forall x : x \mapsto f\ (x\ x))\end{array}}{\forall f : \text{fix}\ f\ \bar{e} = (\forall f : f \mapsto (\forall x : x \mapsto f\ (x\ x))\ (\forall x : x \mapsto f\ (x\ x)))\ f}\,Combine_2}{\text{fix} = (\forall f : f \mapsto (\forall x : x \mapsto f\ (x\ x))\ (\forall x : x \mapsto f\ (x\ x)))}\,\eta$$

$Combine_1, Combine_2$

8.1.4 Substitution of a pattern alternative-bound variable

For proving consistency of functions and case-expressions, the following implication will be required:

$$a \sqsubseteq b \implies e\{v := a\} \sqsubseteq e\{v := b\}$$

A very important aspect is the fact that e is from inside a pattern alternative that binds v. This guarantees that e does not have a pattern alternative that freely contains v in its LHS as a subexpression (see section 2). We will rely on this fact (and without it, above implication would really be false).

We will prove the implication using structural induction on e:

Variable

If $e = v$:

$$\frac{a \sqsubseteq b}{e\{v := a\} \sqsubseteq e\{v := b\}} \equiv$$

If $e \neq v$:

$$\frac{\dfrac{}{e \sqsubseteq e}\, Refl}{e\{v := a\} \sqsubseteq e\{v := b\}} \equiv$$

Data constructor, $\top$, $\bot$ or fix

$$\frac{\dfrac{}{e \sqsubseteq e}\, Refl}{e\{v := a\} \sqsubseteq e\{v := b\}} \equiv$$

Function or case-expression

$$\frac{\dfrac{\dfrac{\text{inductive}}{\forall x : (e\ x)\{v := a\} \sqsubseteq (e\ x)\{v := b\}}}{\forall x : e\{v := a\}\ x \sqsubseteq e\{v := b\}\ x}\, \equiv}{e\{v := a\} \sqsubseteq e\{v := b\}}\, \eta \qquad (\dagger)$$

The statement above and below † are equivalent: Recall that we are guaranteed that e does not contain v freely in the patterns of any of its pattern alternatives. Therefore pattern matching is independent from the value of v and not influenced by the substitution. As a result, it does not matter if you apply the substitution to the pattern alternative before or after reduction.

Application

Assumption: $e = f\ g$

$$\frac{\dfrac{\text{inductive}}{f\{v := a\} \sqsubseteq f\{v := b\}} \quad \dfrac{\text{inductive}}{g\{v := a\} \sqsubseteq g\{v := b\}}}{\dfrac{f\{v := a\}\ g\{v := a\} \sqsubseteq f\{v := b\}\ g\{v := b\}}{e\{v := a\} \sqsubseteq e\{v := b\}}}\, Comp$$

8.1.5 Function/Cases containing a single alternative

Function

$$\frac{(statement_1) \qquad (statement_2)}{(\forall \bar{v} : \rho(\bar{v}) \mapsto \theta(\bar{v}))\ a \sqsubseteq (\forall \bar{v} : \rho(\bar{v}) \mapsto \theta(\bar{v}))\ b}\ Cases \text{ with } P(b) = b \text{ matches } \rho(\bar{v}) \text{ with } \bar{v} := \bar{v_b}$$

$$\frac{\dfrac{\dfrac{P(a)}{\dfrac{(\forall \bar{v} : \rho(\bar{v}) \mapsto \theta(\bar{v}))\ a \rightsquigarrow \theta(\bar{v}_a)}{(\forall \bar{v} : \rho(\bar{v}) \mapsto \theta(\bar{v}))\ a \sqsubseteq \theta(\bar{v}_a)}\,\beta} \qquad \dfrac{\dfrac{a \sqsubseteq b \qquad \dfrac{}{b \sqsubseteq \rho(\bar{v}_b)}\,P(b)}{a \sqsubseteq \rho(\bar{v}_b)}\,Trans \qquad P(a)}{\dfrac{\bar{v}_a \sqsubseteq \bar{v}_b}{\theta(\bar{v}_a) \sqsubseteq \theta(\bar{v}_b)}\,\text{see 8.1.4}} \qquad \dfrac{\dfrac{P(b)}{\dfrac{(\forall \bar{v} : \rho(\bar{v}) \mapsto \theta(\bar{v}))\ b \rightsquigarrow \theta(\bar{v}_b)}{\theta(\bar{v}_b) \sqsubseteq (\forall \bar{v} : \rho(\bar{v}) \mapsto \theta(\bar{v}))\ b}\,\beta}}{\dfrac{(\forall \bar{v} : \rho(\bar{v}) \mapsto \theta(\bar{v}))\ a \sqsubseteq (\forall \bar{v} : \rho(\bar{v}) \mapsto \theta(\bar{v}))\ b}{(statement_1)}\,\equiv}\,Trans$$

$$\frac{\dfrac{\dfrac{}{(\forall \bar{v} : \rho(\bar{v}) \mapsto \theta(\bar{v}))\ a \sqsubseteq \top}\,\top \qquad \dfrac{\dfrac{\neg P(b)}{\dfrac{(\forall \bar{v} : \rho(\bar{v}) \mapsto \theta(\bar{v}))\ b \rightsquigarrow \top}{\top \sqsubseteq (\forall \bar{v} : \rho(\bar{v}) \mapsto \theta(\bar{v}))\ b}\,\beta}\ (\dagger)}{(\forall \bar{v} : \rho(\bar{v}) \mapsto \theta(\bar{v}))\ a \sqsubseteq (\forall \bar{v} : \rho(\bar{v}) \mapsto \theta(\bar{v}))\ b}\,Trans}{(statement_2)}\,\equiv$$

We have used $P(a)$ in the part of the proof where only $P(b)$ is known to be true. This is legitimate:

$$\frac{\dfrac{a \sqsubseteq b \qquad \dfrac{P(b)}{b \sqsubseteq \rho(\bar{v}_b)}\,\equiv}{a \sqsubseteq \rho(\bar{v}_b)}\,Trans}{\dfrac{\exists \bar{v}_a : a \sqsubseteq \rho(\bar{v}_a)}{P(a)}\,\equiv}\,Weaken_{\exists}$$

Case-expression

The proof is analogous to above proof for a function.

8.1.6 Function/Cases containing multiple alternatives

We prove the property for functions with two pattern alternatives. The proofs for case-expressions and methods with more than two alternatives are analogous.

$$\frac{\dfrac{(statement_1) \qquad (statement_2)}{(m_1)\ a \sqcap (m_2)\ a \sqsubseteq (m_1)\ b \sqcap (m_2)\ b}\,\text{def. of infimum } (\forall c : (c \sqsubseteq a \wedge c \sqsubseteq b) \implies c \sqsubseteq a \sqcap b)}{(m_1, m_2)\ a \sqsubseteq (m_1, m_2)\ b}\,\text{def. of function evaluation}$$

$$\dfrac{\dfrac{(m_1)\ a \sqcap (m_2)\ a \sqsubseteq (m_1)\ a \qquad \dfrac{a \sqsubseteq b}{(m_1)\ a \sqsubseteq (m_1)\ b}\ 8.1.5}{(m_1)\ a \sqcap (m_2)\ a \sqsubseteq (m_1)\ b}\ Trans}{(statement_1)}\ \equiv$$

$$\dfrac{\dfrac{(m_1)\ a \sqcap (m_2)\ a \sqsubseteq (m_2)\ a \qquad \dfrac{a \sqsubseteq b}{(m_2)\ a \sqsubseteq (m_2)\ b}\ 8.1.5}{(m_1)\ a \sqcap (m_2)\ a \sqsubseteq (m_2)\ b}\ Trans}{(statement_2)}\ \equiv$$

8.1.7 Consequences of allowing alternative's pattern-variables to be bound by other pattern alternatives

The following expression would now be legal:

$$e = (\forall x : x \mapsto (x \mapsto \bot)\ \top)$$

Note how x in the LHS of the inner pattern alternative is bound by the outer pattern alternative. We observe the following reduction behavior:

$$e\ \bot \rightsquigarrow (\bot \mapsto \bot)\ \top \rightsquigarrow \top$$
$$e\ \top \rightsquigarrow (\top \mapsto \bot)\ \top \rightsquigarrow \bot$$

This allows us to prove:

$$\dfrac{\dfrac{\dfrac{e\ \bot \rightsquigarrow^* \top}{\top \sqsubseteq e\ \bot}\ \beta \quad \dfrac{\dfrac{}{e \sqsubseteq e}\ Refl \quad \dfrac{}{\bot \sqsubseteq \top}\ \bot}{e\ \bot \sqsubseteq e\ \top}\ Comp \quad \dfrac{e\ \top \rightsquigarrow^* \bot}{e\ \top \sqsubseteq \bot}\ \beta}{\top \sqsubseteq \bot}\ Trans \quad \dfrac{}{\bot \sqsubseteq \top}\ \bot}{\top = \bot}\ Antisymm$$

This would imply equality of all expressions! The root of the problem is the contravariance of the function "constructor" in its pattern-expression:

$$p_1 \sqsubseteq p_2 \implies (p_1 \mapsto e) \sqsupseteq (p_2 \mapsto e)$$

Proof (quantifiers for free variables of e, p_1 and p_2 omitted):

$$\dfrac{\dfrac{(statement_1) \qquad (statement_2)}{\forall x : (p_2 \mapsto e)\ x \sqsubseteq (p_1 \mapsto e)\ x}\ Cases \text{ with } P(x) = x \sqsubseteq p_1}{(p_2 \mapsto e) \sqsubseteq (p_1 \mapsto e)}\ \eta$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{x \sqsubseteq p_1 \qquad p_1 \sqsubseteq p_2}{x \sqsubseteq p_2}\ Trans}{(p_2 \mapsto e)\ x \rightsquigarrow e}}{\forall x : (p_2 \mapsto e)\ x \sqsubseteq e}\ \beta \qquad \dfrac{\dfrac{x \sqsubseteq p_1}{(p_1 \mapsto e)\ x \rightsquigarrow e}}{\forall x : e \sqsubseteq (p_1 \mapsto e)\ x}\ \beta}{\forall x : (p_2 \mapsto e)\ x \sqsubseteq (p_1 \mapsto e)\ x}\ Trans}{(statement_1)}\ \equiv$$

$$\dfrac{\dfrac{\dfrac{}{\forall x : (p_2 \mapsto e)\ x \sqsubseteq \top}\top \quad \dfrac{\dfrac{\neg(x \sqsubseteq p_1)}{(p_1 \mapsto e)\ x \rightsquigarrow \top}}{\forall x : \top \sqsubseteq (p_1 \mapsto e)\ x}\beta}{\forall x : (p_2 \mapsto e)\ x \sqsubseteq (p_1 \mapsto e)\ x}Trans}{(statement_2)}\equiv$$

8.2 Proofs for admissible rules

8.2.1 Lift

$$\dfrac{\dfrac{(statement_1) \qquad (statement_2)}{\forall \bar{x} : \Psi\ f\ \bar{x} \sqsubseteq (\Psi\ e_0 \mapsto ... \mapsto e_n \mapsto g)\ \bar{x}}Cases \text{ with } P(\bar{x}) = \bar{x} \sqsubseteq \bar{e}\{... := \top\}}{\Psi\ f \sqsubseteq (\Psi\ e_0 \mapsto ... \mapsto e_n \mapsto g)}\eta$$

$$\dfrac{\dfrac{\forall \bar{x} : \Psi\ f\ \bar{x} \sqsubseteq g \qquad \dfrac{\dfrac{\bar{x} \sqsubseteq \bar{e}\{... := \top\}}{\forall \bar{x} : (\Psi\ e_0 \mapsto ... \mapsto e_n \mapsto g)\ \bar{x} \rightsquigarrow g}\beta}{\forall \bar{x} : g \sqsubseteq (\Psi\ e_0 \mapsto ... \mapsto e_n \mapsto g)\ \bar{x}}}{\forall \bar{x} : \Psi\ f\ \bar{x} \sqsubseteq (\Psi\ e_0 \mapsto ... \mapsto e_n \mapsto g)\ \bar{x}}Trans}{(statement_1)}\equiv$$

$$\dfrac{\dfrac{\dfrac{}{\forall \bar{x} : \Psi\ f\ \bar{x} \sqsubseteq \top}\top \quad \dfrac{\dfrac{\neg(\bar{x} \sqsubseteq \bar{e}\{... := \top\})}{\forall \bar{x} : (\Psi\ e_0 \mapsto ... \mapsto e_n \mapsto g)\ \bar{x} \rightsquigarrow \top}\beta}{\top \sqsubseteq (\Psi\ e_0 \mapsto ... \mapsto e_n \mapsto g)\ \bar{x}}}{\forall \bar{x} : \Psi\ f\ \bar{x} \sqsubseteq (\Psi\ e_0 \mapsto ... \mapsto e_n \mapsto g)\ \bar{x}}Trans}{(statement_2)}\equiv \quad (\dagger)$$

Remark: The β reduction rule marked with † is a good example of our strong requirement of functions evaluating to $\top$ when pattern matching fails.

8.2.2 Fix

$$\dfrac{(statement_1) \qquad \Psi\ f\ (\Psi\ e_0 \mapsto ... \mapsto e_n \mapsto g)\ \bar{e} \sqsubseteq g}{\Psi \text{ fix } f\ \bar{e} \sqsubseteq g}Trans$$

$$\dfrac{\dfrac{(statement_2) \qquad \dfrac{}{\Psi\ e_0 \sqsubseteq e_0}Refl \qquad ...}{\Psi \text{ fix } f\ \bar{e} \sqsubseteq f\ (\Psi\ e_0 \mapsto ... \mapsto e_n \mapsto g)\ \bar{e}}Comp}{(statement_1)}\equiv$$

$$\dfrac{\dfrac{\dfrac{\text{fix } f \rightsquigarrow f\ (\text{fix } f)}{\Psi\ \text{fix } f \sqsubseteq f\ (\text{fix } f)}\,\beta \quad \dfrac{\dfrac{}{\Psi\ f \sqsubseteq f}\,Refl \quad \dfrac{\Psi\ \text{fix } f\ \bar{e} \sqsubseteq g}{\Psi\ \text{fix } f \sqsubseteq (\Psi\ e_0 \mapsto ... \mapsto e_n \mapsto g)}\,Lift}{\Psi\ f\ (\text{fix } f) \sqsubseteq f\ (\Psi\ e_0 \mapsto ... \mapsto e_n \mapsto g)}\,Comp}{\Psi\ \text{fix } f \sqsubseteq f\ (\Psi\ e_0 \mapsto ... \mapsto e_n \mapsto g)}\,Trans}{(statement_2)}\equiv$$

Note that this proof is recursive. The extra condition $\Psi\ f\ \top\ e_0\ e_1\ ...\ e_n \sqsubset \top$ (simplification of $\Psi\ f\ \top\ e_0\ e_1\ ...\ e_n \sqsubset \top\ e_0\ e_1\ ...\ e_n$) ensures that for an existing fixed point (represented by $\top$) and *all* possible arguments (e_0 .. e_n) one application of f actually performs an operation. In other words, each application of f narrows down the set of possible fixed-points passed in ($\top$ representing the set of all fixed-points).

8.2.3 Weaken$_{\forall\exists}$

We choose a distinct, fresh data constructor C.

$$\dfrac{\dfrac{\Phi_1\ \forall x : \Phi_2\ e \sqsubseteq f}{\Phi_1\ \Phi_2\ e\{x := C\} \sqsubseteq f\{x := C\}}\,Weaken_\forall}{\exists x : \Phi_1\ \Phi_2\ e \sqsubseteq f}\,Weaken_\exists$$

8.2.4 Combine$_1$

Let the statement

$$\Psi'\ e_1\{subst\} \sqsubseteq f\{subst\}$$

be the witness of

$$\Phi\ e_1 \sqsubseteq f$$

, therefore only containing universally quantifies variables that arise from substitutions $\{subst\}$ or promotions of existentially quantified variables.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Psi\ e_1 = e_2}{\Psi\ e_2 = e_1}\,Symm_=}{\Psi\ e_2\{subst\} = e_1\{subst\}}\,Weaken_=}{\Psi\ e_2\{subst\} \sqsubseteq e_1\{subst\}}\,Introduce_\forall* \quad \dfrac{\Psi'\ e_1\{subst\} \sqsubseteq f\{subst\}}{\Psi\ \Psi'\ e_1\{subst\} \sqsubseteq f\{subst\}}\,Introduce_\forall*, Swap_{weaken}}{\Psi\ \Psi'\ e_2\{subst\} \sqsubseteq f\{subst\}}\,Trans}{\Psi\ \Phi\ e_2 \sqsubseteq f}\,Weaken_\exists*}{\forall\ vars(e_2) : \Phi\ e_2 \sqsubseteq f}\,Hide_\forall/Hide_\exists*$$

8.2.5 Combine$_2$

Analogous to $Combine_1$.

8.2.6 CaseCombine$_\exists$

$$m_i \equiv [\forall \bar{v_i} : \alpha_i(\bar{v_i}) \mapsto \beta_i(\bar{v_i})]$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\Phi_1\ \exists \bar{v_i} : \Phi_2\ \beta_i(\bar{v_i})\{x := \alpha_i(\bar{v_i})\} \sqsubseteq e\{x := \alpha_i(\bar{v_i})\} \qquad \textit{(evaluation omitted)}}{\Phi_1\ \exists \bar{v_i} : \Phi_2\ [\forall \bar{v_i} : \alpha_i(\bar{v_i}) \mapsto \beta_i(\bar{v_i})\{x := \alpha_i(\bar{v_i})\}]\ \alpha_i(\bar{v_i}) \sqsubseteq e\{x := \alpha_i(\bar{v_i})\}}\ \textit{Combine}_1}{\Phi_1\ \exists \bar{v_i} : \Phi_2\ (m_i\ \alpha_i(\bar{v_i}))\{x := \alpha_i(\bar{v_i})\} \sqcup ([...\text{remaining...}]\ \alpha_i(\bar{v_i}))\{x := \alpha_i(\bar{v_i})\} \sqsubseteq e\{x := \alpha_i(\bar{v_i})\}}\ \text{see below}}{\Phi_1\ \exists \bar{v_i} : \Phi_2\ (m_i\ \alpha_i(\bar{v_i})) \sqcup ([...\text{remaining...}]\ \alpha_i(\bar{v_i})) \sqsubseteq e}\ \textit{Weaken}_\exists\ (x := \alpha_i(\bar{v_i}))$$

$$\dfrac{\dfrac{\Phi_1\ \exists x : \exists \bar{v_i} : \Phi_2\ (m_i\ x) \sqcup ([...\text{remaining...}]\ x) \sqsubseteq e}{\Phi_1\ \exists x : \Phi_2\ (m_i\ x) \sqcup ([...\text{remaining...}]\ x) \sqsubseteq e}\ \textit{Hide}_{\exists *}}{\Phi_1\ \exists x : \Phi_2\ [\forall \bar{v_0} : \alpha_0(\bar{v_0}) \mapsto \beta_0(\bar{v_0}), ...]\ x \sqsubseteq e}\ \text{def. of evaluation}$$

Above, at one point we make use of the resulting alternative's regularity: We know that $\alpha_i(\bar{v_i})$ matches the pattern of the ith alternative (perfectly). Thus it is an expression in normal form and with a constructor as leftmost subexpression. This cannot match any other alternative's pattern since constructors are pairwise distinct (see section 4.6 about regular methods).

8.2.7 CaseCombine$_\forall$

$$m \equiv [\forall \bar{v_0} : \alpha_0(\bar{v_0}) \mapsto \beta_0(\bar{v_0}), ...]$$

$$\dfrac{(\textit{statement}_1) \qquad (\textit{statement}_2)}{\Phi_1\ \forall x : \Phi_2\ m\ x \sqsubseteq e}\ \textit{Cases}\ \text{with}\ P(x) = (x = \top)$$

$$\dfrac{\dfrac{\Phi_1\ \Phi_2\ \bigsqcup_{i=0}^{n} (\beta_i(\top, ...)\{x := \top\}) \sqsubseteq e\{x := \top\} \qquad \textit{(evaluation omitted)}}{\Phi_1\ \Phi_2\ m\{x := \top\}\ \top \sqsubseteq e\{x := \top\}}\ \textit{Combine}_1}{(\textit{statement}_1)}\ \equiv$$

$$\dfrac{(\textit{statement}_3) \qquad (\textit{statement}_4)}{\dfrac{\Phi_1\ \forall x : \Phi_2\ m\ x \sqsubseteq e}{(\textit{statement}_2)}\ \equiv}\ \textit{Cases}\ \text{with}\ P(x) = (\alpha_0(\bar{v_0}) \sqsubseteq x)$$

$$\dfrac{\dfrac{\Phi_1\ \forall \bar{v_0} : \Phi_2\ \beta_0(\bar{v_0})\{x := \alpha_0(\bar{v_0})\} \sqsubseteq e\{x := \alpha_0(\bar{v_0})\}}{\Phi_1\ \forall x : \Phi_2\ m\ x \sqsubseteq e}}{(\textit{statement}_3)}\ \text{we know that } x = \alpha_0(\bar{v_0}) \text{ for some } \bar{v_0}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Phi_1\ \forall x : \Phi_2\ \bot \sqsubseteq e}\ \bot \quad \textit{(evaluation omitted)}}{\Phi_1\ \forall x : \Phi_2\ []\ x \sqsubseteq e}\ Combine_1}{\vdots}\ \text{we end up with a case with an empty case-expression}}{\Phi_1\ \forall x : \Phi_2\ [\forall \bar{v_1} : \alpha_1(\bar{v_1}) \mapsto \beta_1(\bar{v_1}), ...]\ x \sqsubseteq e}\ \textit{Cases}\text{, analogous to below cases, for all alternatives}}{\Phi_1\ \forall x : \Phi_2\ m\ x \sqsubseteq e}\ \text{we know that } x \neq \alpha_0(\bar{v_0}) \text{ for any } \bar{v_0}}{(statement_4)}\ \equiv$$

Again, we make heavy use of the regularity of the resulting case-expression (see section 4.6).

8.3 Proofs for example statements

Statement: $\exists x : \forall n_0, j_0 : (\forall n : n \mapsto (\forall j : j \mapsto [N \mapsto n, \forall s : J\ s \mapsto j\ s]))\ n_0\ j_0\ x \sqsubseteq n_0$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\forall n_0, j_0 : n_0 \sqsubseteq n_0}\ Refl}{\forall n_0, j_0 : [N \mapsto n_0, \forall s : J\ s \mapsto j_0\ s]\ N \sqsubseteq n_0}\ Combine_1}{\forall n_0, j_0 : (\forall n : n \mapsto (\forall j : j \mapsto [N \mapsto n, \forall s : J\ s \mapsto j\ s]))\ n_0\ j_0\ N \sqsubseteq n_0}\ Combine_1*}{\exists x : \forall n_0, j_0 : (\forall n : n \mapsto (\forall j : j \mapsto [N \mapsto n, \forall s : J\ s \mapsto j\ s]))\ n_0\ j_0\ x \sqsubseteq n_0}\ Weaken_\exists$$

Statement: $\exists n_0, j_0 : \forall x : (\forall n : n \mapsto (\forall j : j \mapsto [N \mapsto n, \forall s : J\ s \mapsto j\ s]))\ n_0\ j_0\ x \sqsubseteq x$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{N \sqcup (J\ \top) \sqsubseteq \top}\ \top \quad \dfrac{}{N \sqsubseteq N}\ Refl \quad \dfrac{}{\forall s : J\ s \sqsubseteq J\ s}\ Refl}{\forall x : [N \mapsto N, \forall s : J\ s \mapsto J\ s]\ x \sqsubseteq x}\ CaseCombine_\forall}{\forall x : (\forall n : n \mapsto (\forall j : j \mapsto [N \mapsto n, \forall s : J\ s \mapsto j\ s]))\ N\ J\ x \sqsubseteq x}\ Combine_1*}{\exists n_0, j_0 : \forall x : (\forall n : n \mapsto (\forall j : j \mapsto [N \mapsto n, \forall s : J\ s \mapsto j\ s]))\ n_0\ j_0\ x \sqsubseteq x}\ Weaken_\exists*$$

Statement: $\exists f : \forall l : \text{fix}\ (\forall r : r \mapsto (\forall g : g \mapsto [E \mapsto E, \forall h, t : C\ h\ t \mapsto C\ (g\ h)\ (r\ g\ t)]))\ f\ l \sqsubseteq l$

$$m \equiv (\forall r : r \mapsto (\forall g : g \mapsto [E \mapsto E, \forall h, t : C\ h\ t \mapsto C\ (g\ h)\ (r\ g\ t)]))$$
$$i \equiv (\forall x : x \mapsto x)$$

$$\dfrac{statement_{ind} \qquad \dfrac{\dfrac{\dfrac{}{l \sqsubset \top}}{\forall l : [E \mapsto E, \forall h, t : C\ h\ t \mapsto C\ h\ \top]\ l \sqsubseteq \top}\ 4.6}{\forall l : m\ \top\ i\ l \sqsubseteq \top}\ Combine_1*}{\forall l : \text{fix}\ m\ i\ l \sqsubseteq l}\ Fix$$
$$\dfrac{}{\exists f : \forall l : \text{fix}\ m\ f\ l \sqsubseteq l}\ Weaken_\exists$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{E \sqcup (C\ \top\ \top) \sqsubseteq \top}\top \quad \dfrac{}{E \sqsubseteq E}Refl \quad \dfrac{}{\forall h,t : C\ h\ t \sqsubseteq C\ h\ t}Refl}{\forall l : [E \mapsto E, \forall h,t : C\ h\ t \mapsto C\ h\ t)]\ l \sqsubseteq l}CaseCombine_\forall}{\forall l : [E \mapsto E, \forall h,t : C\ h\ t \mapsto C\ (i\ h)\ ((\forall l : i \mapsto l \mapsto l)\ i\ t)]\ l \sqsubseteq l}Combine_1*}{\forall l : (\forall g : g \mapsto [E \mapsto E, \forall h,t : C\ h\ t \mapsto C\ (g\ h)\ ((\forall l : i \mapsto l \mapsto l)\ g\ t)])\ i\ l \sqsubseteq l}Combine_1}{\forall l : m\ (\forall l : i \mapsto l \mapsto l)\ i\ l \sqsubseteq l}Combine_1}{statement_{ind}} \equiv$$

Statement: $\text{fix}\ (\forall x : x \mapsto x, \top \mapsto C) = \bot$

$$\dfrac{\dfrac{(\forall x : x \mapsto x, \top \mapsto C)\ \bot \rightsquigarrow \bot \sqcap C = \bot}{(\forall x : x \mapsto x, \top \mapsto C)\ \bot \sqsubseteq \bot}\beta \quad \dfrac{\dfrac{C \sqsubset \top}{\top \sqcap C \sqsubset \top} \equiv}{(\forall x : x \mapsto x, \top \mapsto C)\ \top \sqsubset \top}Combine_1}{\text{fix}\ (\forall x : x \mapsto x, \top \mapsto C) \sqsubseteq \bot}Fix$$

$$\dfrac{}{\bot \sqsubseteq \text{fix}\ (\forall x : x \mapsto x, \top \mapsto C)}\bot$$

Statement: $\text{fix}\ \bot = \bot$

$$\dfrac{\dfrac{\text{fix}\ \bot \rightsquigarrow \bot\ (\text{fix}\ \bot) \rightsquigarrow \bot}{\text{fix}\ \bot \sqsubseteq \bot}\beta \quad \dfrac{}{\bot \sqsubseteq \text{fix}\ \bot}\bot}{\text{fix}\ \bot = \bot}Antisymm$$

Alternative:

$$\dfrac{\dfrac{\dfrac{\bot\ \bot \rightsquigarrow \bot}{\bot\ \bot \sqsubseteq \bot}\beta \quad \dfrac{\bot \sqsubset \top}{\bot\ \top \sqsubset \top}}{\text{fix}\ \bot \sqsubseteq \bot}Fix \quad \dfrac{}{\bot \sqsubseteq \text{fix}\ \bot}\bot}{\text{fix}\ \bot = \bot}Antisymm$$

Statement: $\text{fix}\ \top = \top$

$$\dfrac{\dfrac{}{\text{fix}\ \top \sqsubseteq \top}\top \quad \dfrac{\text{fix}\ \top \rightsquigarrow \top\ (\text{fix}\ \top) \rightsquigarrow \top}{\top \sqsubseteq \text{fix}\ \top}\beta}{\text{fix}\ \top = \top}Antisymm$$

Statement: $\text{fix}\ (\top \mapsto \bot) = \bot$

$$\dfrac{\dfrac{}{\bot \sqsubseteq \text{fix}\ (\top \mapsto \bot)}\bot \quad \dfrac{\dfrac{\dfrac{}{\text{fix}\ (\top \mapsto \bot) \sqsubseteq \top}\top}{\text{fix}\ (\top \mapsto \bot) \rightsquigarrow (\top \mapsto \bot)\ (\text{fix}\ (\top \mapsto \bot)) \rightsquigarrow \bot}}{\text{fix}\ (\top \mapsto \bot) \sqsubseteq \bot}\beta}{\text{fix}\ (\top \mapsto \bot) = \bot}Antisymm$$

Alternative:

$$\dfrac{\dfrac{}{\bot \sqsubseteq \text{fix}\ (\top \mapsto \bot)}\ \bot \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{\bot \sqsubseteq \top}\ \top}{(\top \mapsto \bot)\ \bot \rightsquigarrow \bot}\ \beta}{(\top \mapsto \bot)\ \bot \sqsubseteq \bot} \qquad \dfrac{\bot \sqsubset \top}{(\top \mapsto \bot)\ \top \sqsubset \top}}{\text{fix}\ (\top \mapsto \bot) \sqsubseteq \bot}\ \mathit{Fix}}{\text{fix}\ (\top \mapsto \bot) = \bot}\ \mathit{Antisymm}$$

Statement: $\text{fix}\ (\bot \mapsto \top) = \top$

$$\dfrac{\dfrac{(\bot \mapsto \top) \text{ maps } \textit{everything} \text{ to } \top}{\text{fix}\ (\bot \mapsto \top) \rightsquigarrow (\bot \mapsto \top)\ (\text{fix}\ (\bot \mapsto \top)) \rightsquigarrow \top}\ \beta}{\top \sqsubseteq \text{fix}\ (\bot \mapsto \top)} \qquad \dfrac{}{\text{fix}\ (\bot \mapsto \top) \sqsubseteq \top}\ \top}{\text{fix}\ (\bot \mapsto \top) = \top}\ \mathit{Antisymm}$$

Statement: $\text{fix}\ (\top \mapsto \top) = \top$

$$\dfrac{\dfrac{(\top \mapsto \top) \text{ maps } \textit{everything} \text{ to } \top}{\text{fix}\ (\top \mapsto \top) \rightsquigarrow (\top \mapsto \top)\ (\text{fix}\ (\top \mapsto \top)) \rightsquigarrow \top}\ \beta}{\top \sqsubseteq \text{fix}\ (\top \mapsto \top)} \qquad \dfrac{}{\text{fix}\ (\top \mapsto \top) \sqsubseteq \top}\ \top}{\text{fix}\ (\top \mapsto \top) = \top}\ \mathit{Antisymm}$$

Bibliography

[1] S. Antoy. *Constructor-based conditional narrowing.* In Proc. PPDP'01, 199-206. ACM (2001)

[2] H. Barendregt. *Lambda Calculus: its Syntax and Semantics.* North Holland (1984)

[3] A. Bove, P. Dybjer, A. Sicard-Ramírez. *Combining interactive and automatic reasoning in first order theories of functional programs.* Foundations of Software Science and Computational Structures, Lecture Notes in Computer Science, Vol. 7213, 104-118 (2012)

[4] J. Breitner, R. A. Eisenberg, S. Peyton Jones, S. Weirich. *Safe Coercions.* ICFP (2014)

[5] H.-J. Bürckert. *Matching - A special case of unification?* Journal of Symbolic Computation, Vol. 8, Issue 5, 523–536 (1989)

[6] A. Church, J.B. Rosser. *Some Properties of Conversion.* Transactions of the American Mathematical Society, Vol. 39, No. 3, 472-482 (1936)

[7] H. Cirstea, C. Kirchner, L. Liquori. *Matching Power.* Rewriting Techniques and Applications, 12th International Conference, RTA 2001, Vol. 2051, 77-92. Springer (2001) Digital object available from author's homepage.

[8] H. Cirstea, L. Liquori, B. Wack. *Rewriting Calculus with Fixpoints: Untyped and First-order Systems.* Types for Proofs and Programs, International Workshop, TYPES 2003, Vol. 3085, 147-161. Springer (2004) Digital object available from author's homepage.

[9] J. Endrullis, H. Hvid Hansen, D. Hendriks, A. Polonsky, A. Silva. *A Coinductive Treatment of Infinitary Rewriting.* arXiv:1306.6224 (2013)

[10] B. Jay. *Pattern Calculus: Computing with Functions and Structures.* Springer (2009)

[11] J.W. Klop, V. van Oostrom, R. de Vrijer. *Lambda calculus with patterns.* Theoretical Computer Science 398, 16-31 (2008)

[12] S. Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice Hall (1987) Digital object available from author's homepage.

[13] Terese. *Term Rewriting Systems.* Cambridge Tracts in Theoretical Computer Science, Vol. 55. Cambridge University Press (2003)

[14] D. Vytiniotis, S. Peyton Jones, K. Claessen, D. Rosen. *HALO: Haskell to Logic through Denotational Semantics.* POPL '13 (2013)

[15] G. Winskel. *The formal semantics of programming languages - an introduction.* Foundation of computing series. MIT Press (1993)

[16] D.N. Xu. *Hybrid Contract Checking via Symbolic Simplification.* In Proc. PEPM'12, 107–116. ACM (2012)