Formalizing inference systems in Coq by means of type systems for Curry

Niels Bunkenburg

Bachelor's Thesis
Programming Languages and Compiler Construction
Department of Computer Science
Christian-Albrechts-University of Kiel

Advised by
Prof. Dr. Michael Hanus
M. Sc. Sandra Dylus

September 13, 2016

Abstract

content...

Contents

List of Figures

1 Introduction

2 Preliminaries

2.1 Curry

Curry[1] is a programming language that combines aspects of functional and logical programming. Created by an international initiative, Curry is predominantly aimed at providing a platform for research and teaching.Antoy and Hanus [2014] Curry's syntax is similar to Haskell with the addition of nondeterminism, that is, a function can return different values for the same input, and free variables, which allow the systematic search for unknown values.

The similarities to Haskell include the (interactive) compiler KICS2[2] that compiles Curry programs to Haskell code and an easily searchable[3], extensive module system with many predefined data structures and functions. In this short overview we will take a look at some language features that we will work with later, albeit not necessarily in the form of Curry code.

Curry programs have the ending `.curry` and consist of function and data type declarations. The simplest form of functions, such as `double x = x + x`, has a name, a possibly empty list of variables and an expression. Although it is not mandatory to explicitly state a function's type, in this case `double :: Int → Int → Int`, every function has a type that describes the arguments and the result of evaluating the function. When a function call is evaluated, the left-hand side is replaced with the right-hand side until there is no further evaluation possible, that is, only literal values or data structures remain. The evaluation can result in an infinite loop, for example `twos = 2 : twos`, or not compute any values. Infinite data structures are processed by computing the value of an expression only if it is actually needed, called *lazy* evaluation. In the example below the function `take` returns the first n elements of a list. The value of `twos` is only evaluated by one step when the application of `take` requires the values of the list's head. Thus, the infinite list can be accessed without computing the whole structure.

```
take 2 twos
take 2 (2 : twos)
2 : take 1 twos
2 : take 1 (2 : twos)
2 : 2 : take 0 twos
2 : 2 : []
```

```
take :: Int -> [a] -> [a]
take n l = if n <= 0 then [] else takep n l
  where takep _ []      = []
        takep m (x:xs) = x : take (m-1) xs
```

[1]http://curry-language.org/
[2]https://www-ps.informatik.uni-kiel.de/kics2/
[3]https://www-ps.informatik.uni-kiel.de/kics2/currygle/

More advanced functions use several rules and pattern matching to describe different computational paths. By using a pattern in the left-hand side of a rule, it is limited to arguments of a specific form. Thus, functions that process data types with multiple constructors can have a different rule for every constructor. While Haskell allows overlapping patterns, that is, multiple patterns that apply to the same argument, functions remain deterministic because only the first rule that matches the argument is evaluated. Curry does not limit the evaluation – a function with overlapping patterns can return every possible result an input evaluates to. On the other hand, patterns do not need to be exhaustive, that is, cover every possible input, because a failed computation is a valid result unlike the exception Haskell returns.

```
failed = head []
eight = failed ? 8
```

```
(?) :: a -> a -> a
x ? _ = x
_ ? y = y
```

The function **head** returns the first element of a list, but in this case the list is empty. Therefore, the computation fails and KICS2 returns an exclamation mark. Nevertheless, it is possible to apply the `?` operator that represents a non-deterministic choice between the arguments to both the failed computation and the number 8. Since only one argument contains a value, the expression evaluates to 8.

In some situations deterministic programs are inevitable, for example when working with IO actions. While it may not seem import if a function prints an 'a' or 'b' to the terminal, writing a file of non-deterministic size to the hard drive can cause severe problems. Fortunately, Curry enforces determinism when using IO actions. If one needs a more Haskell-esque evaluation of multiple patterns, the **case** expression applies:

```
maybeNot True  = False
maybeNot True  = True
maybeNot False = True
maybeNot False = False
```

```
notCase b =
  case b of
    True  -> False
    True  -> True
    False -> True
    False -> False
```

```
maybeNotCase b =
  case b of
    True  -> False ? True
    False -> True  ? False
```

Pattern matching in a function can be transformed to a corresponding case expression by replacing the pattern with a variable and creating a branch for every rule. The difference is that case expressions evaluate only the first matching branch. Thus, the second and fourth branch of `caseNot` are not reachable and the function is deterministic. If nondeterminism is needed in a **case** expression, the `?` operator is necessary.

Variables that are not bound to an expression are called *free variables*. They are declared by adding **where** $v_1 \ldots v_n$ **free** to a term containing free variables $v_1 \ldots v_n$. To evaluate such a term, the variables are instantiated with a value of appropriate type, for example `False && x where x free`, that is, the boolean conjunction of `False` and a free variable, evaluates to `{x = _x0} False`. The value of `x` is another free variable because the expression is false for every possible value of `x`.

Free variables can be used to compute the possible values of an expression, but there are more sophisticated applications: Curry can solve *constraints*, for example the equation

`1 + 1 = x where x free`. Solving the expression results in multiple values for `x` and corresponding the value of the equation:

```
{x = (-_x2)} False
{x = 0} False
{x = 1} False
{x = 2} True
{x = (2 * _x3 + 1)} False
{x = (4 * _x4)} False
{x = (4 * _x4 + 2)} False
```

Curry solves the equation by using *narrowing*, that is, guessing values that could satisfy the constraints. Since the sum of two positive numbers cannot be negative, only positive numbers are potential solutions. 2 fulfills the equation, but there could be more solutions. Therefore, odd numbers and even numbers greater than 2 are checked, both of which cannot satisfy the constraint. Since this includes all possible integer values, 2 remains the only solution. Free variables with non-basic types are evaluated similarly by trying every possible constructor.

An alternative approach to solving constraints is *residuation*: If an expression cannot be solved because of an unknown free variable, the evaluation is suspended until its value is known from evaluating other expressions. It is possible that there is not enough information to determine the variable's value, in which case the evaluation fails.

2.2 Coq

The formalization of Curry programs requires a language that allows us to express the code itself and the propositions we intend to prove. Coq[4] is an interactive proof management system that meets these requirements. Hence, it will be the main tool used in the following chapters. Pierce et al. [2016]

TODO: Kommasetzung?

2.2.1 Data types and functions

Coq's predefined definitions, contrary to e.g. Haskell's Prelude, are very limited. Nevertheless, being a functional language, there is a powerful mechanism for defining new data types. A definition of polymorphic lists could look like this:

```
Inductive list (X:Type) : Type :=
  | nil  : list X
  | cons : X -> list X -> list X.
```

We defined a type named `list` with two constructors: the constant `nil`, which represents an empty list, and a binary constructor `cons` that takes an element and a list of the same type as arguments. In fact, `nil` and `cons` have one additional argument, a type `X`. This is required, because we want polymorphic lists – but we do not want to explicitly state the type. Fortunately, Coq allows us to declare type arguments as implicit by enclosing them in curly brackets:

[4] https://coq.inria.fr/

```
Check (cons nat 8 (nil nat)). (*cons nat 8 (nil nat) : list nat*)
Arguments nil {X}.
Arguments cons {X} _ _.
```

Coq's type inference system infers the type of a list automatically now if possible. In some cases this does not work, because there is not enough information about the implicit types present.

```
Fail Definition double_cons x y z := (cons x (cons y z)).
Definition double_cons {A} x y z := (@cons A x (@cons A y z)).
```

The first definition does not work, as indicated by `Fail`[5], because Coq cannot infer the implicit type variable of `double_cons`, since `cons` does not have an explicit type either. By prefixing at least one `cons` with `@`, we can tell Coq to accept explicit expressions for all implicit arguments. This allows us to pass the type of `cons` on to `double_cons`, again as an implicit argument.

```
Check double_cons 2 4 []. (* : list nat *)
Fail Check (cons 2 (cons nil nil)).
(* Error: The term "cons nil nil" has type "list (list ?X0)"
while it is expected to have type "list nat". *)
```

Based on this we can write a function that determines if a list is empty:

```
Definition isEmpty {X : Type} (l : list X) : bool :=
  match l with
  | nil      => true
  | cons _ _ => false
  end.
```

Function definitions begin with the keyword `Definition`. `isEmpty` takes an (implicit) type and a list and returns a boolean value. To distinguish empty from non-empty lists, pattern matching can be used on n arguments by writing `match` $x_0, ..., x_{n-1}$ `with` $| \; p_0 \to e_0 \; | \; ... \; | \; p_{m-1} \to e_{m-1}$ for m pattern p, consisting of a sub-pattern for every x_i and expressions e_i.

The definition of recursive functions requires that the function is called with a smaller structure than before in each iteration, which ensures that the function eventually terminates. A recursive function is indicated by using `Fixpoint` instead of `Definition`.

```
Fixpoint app {X : Type} (l1 l2 : list X) : (list X) :=
  match l1 with
  | nil => l2
  | cons h t => cons h (app t l2)
  end.
```

In this case l_1 gets shorter with every iteration and thus the function terminates after a finite number of recursions.

Coq allows us to define notations for functions and constructors by using the keyword `Notation`, followed by the desired syntax and the expression.

[5] `Fail` checks if an expression does indeed cause an error and allows further processing of the file.

```
Notation "x :: y" := (cons x y) (at level 60, right associativity).
Notation "[ ]" := nil.
Notation "[ x ; .. ; y ]" := (cons x .. (cons y []) ..).
Notation "x ++ y" := (app x y) (at level 60, right associativity).
```

2.2.2 Propositions and proofs

Every claim that we state or prove has the type Prop. Propositions can be any statement, regardless of its truth. A few examples:

```
Check 1 + 1 = 2. (* : Prop *)
Check forall (X : Type) (l : list X), l ++ [] = l. (* : Prop *)
Check forall (n : nat), n > 0 -> n * n > 0. (* : Prop *)
Check (fun n => n <> 2). (* : nat -> Prop*)
```

The first proposition is a simple equation, while the second one contains an universal quantifier. This allows us to state propositions about every type of list, or, as shown in the third example, about every natural number greater than zero. Combined with implications we can premise specific properties that limit the set of elements the proposition applies to. The last example contains an anonymous function, which is used by stating the functions' arguments and an expression.

Now how do we prove these propositions? Proving an equation requires to show that both sides are equal, usually by simplifying one side until it looks exactly like the other. Coq allows us to do this by using tactics, which can perform a multitude of different operations.

```
Example e1 : 1+1=2.
Proof. simpl. reflexivity. Qed.
```

TODO: Highlighting für Proof und Qed

After naming the proposition as an example, theorem or lemma it appears in the interactive subgoal list that Coq provides. The `simpl` tactic performs basic simplification like adding two numbers in this case. The updated subgoal is now 2=2, which is obviously true. By using the `reflexivity` tactic we tell Coq to check both sides for equality, which succeeds and clears the subgoal list, followed by `Qed` to complete the proof.

```
Example e2 : forall (X : Type) (l : list X), [] ++ l = l.
Proof. intros X l. reflexivity. Qed.
```

Universal quantifiers allow us to introduce variables, the corresponding tactic is called `intros`. The new context contains a type `X` and a list `l`, with the remaining subgoal `[ ] ++ l = l`. Because we defined `app` to return the second argument if the first one is an empty list, `reflexivity` directly proves our goal. `reflexivity` is not useful for obvious equations only, it also simplifies and unfolds definitions until the flat terms match each other if possible.

To prove that the proposition `l ++ [] = l` holds, we need more advanced tactics, because we cannot just apply the definition. `app` works by iterating through the first list, but we need to prove the proposition for every list, regardless of its length. One possibility to solve this problem is by using structural induction.

```
Example e3 : forall (X : Type) (l : list X), l ++ [] = l.
Proof. intros X. induction l as [|l ls IH].
  reflexivity.
  simpl. rewrite IH. reflexivity.
Qed.
```

The proof begins by introducing type `X`, followed by the `induction` tactic applied to `l`. Coq names newly introduced variables by itself, which can be done manually by adding `as [c1|...|cn]` to the tactic. Each c_i represents a sequence of variable names, which will be used when introducing variables in the corresponding case. Cases are ordered as listed in the `Definition`.

Now we need to prove two cases: the empty list and a cons construct. The first case does not require any new variable names, therefore the first section in the squared brackets is empty. It is easily solved by applying `reflexivity`, because of the definition of `app`. The second case requires variables for the list's head and tail, which we call `l` and `ls` respectively. The variable name `IH` identifies the induction hypothesis `ls ++ [ ] = ls`, which Coq generates automatically. The goal changes as following:

```
(l :: ls) ++ [ ] = l :: ls
 l :: ls  ++ [ ] = l :: ls (* simpl *)
 l :: ls         = l :: ls (* rewrite with IH *)
```

The tactic `rewrite` changes the current goal by replacing every occurrence of the left side of the provided equation with the right side. Both sides are equal now and therefore `reflexivity` proves the last case.

Example `e4` is different from the other examples, in the sense that one cannot prove a function by itself and that only supplying an argument returns a verifiable inequality.

```
Example e4 : (fun n => n <> 2) 1.
Proof.
  simpl.        (* 1 <> 2 *)
  unfold not.   (* 1 = 2 -> False *)
  intros H.     (* H : 1 = 2, False *)
  inversion H. (* No more subgoals. *)
Qed.
```

This proof is not as straight forward as the other ones, mainly because of the inequality, which is a notation[6] for `not (x = y)`. Because `not` is the outermost term, we need to eliminate it first by applying `unfold`. This replaces `not` with its definition `fun A : Prop => A -> False`, where `False` is the unprovable proposition. Why does this work? Assuming that a proposition `P` is true, `not P` means that `P` implies `False`, which is false, because something true cannot imply something false. On the other hand, if `P` is false, then `False -> False` is true because anything follows from falsehood, as stated by the principle of explosion. **TODO:** verweis?

The current goal `4 = 8 -> False` is further simplified by introducing `4 = 8` as an hypothesis `H`, leaving `False` as the remaining goal. Intuitively we know that `H` is false,

[6]It is often useful to be able to look up notations, `Locate "<>"` returns the term associated with `<>`.

but Coq needs a justification for this claim. Conveniently the tactic `inversion` solves this problem easily by applying two core principles of inductively defined data types:

- Injectivity: `C n = C m` implies that n and m are equal for a constructor `C`.
- Disjoint constructors: Values created by different constructors cannot be equal.

By applying `inversion` to the hypothesis `2 = 1` we tell Coq to add all inferable equations as additional hypotheses. In this case we start with `2 = 1` or the Peano number representation `S(S(O)) = S(O)`. Injectivity implies that if the previous equation was true, `S(O) = O` must also be true. This is obviously false, since it would allow two different representations of nil. Hence, the application of `inversion` to `2 = 1` infers the current goal `False`, which concludes the proof.

Besides directly supplying arguments to functions that return propositions, there are other interesting applications for them, that we will discuss in the next section.

2.2.3 Higher-order constructs

Functions can be passed as arguments to other functions or returned as a result, they are first-class citizens in Coq. This allows us create higher-order functions, such as `map` or `fold`.

TODO: minted bug?

```
Fixpoint map {X Y : Type} (f : X -> Y) (l : list X) : (list Y) :=
  match l with
  | []     => []
  | h :: t => (f h) :: (map f t)
  end.
```

Function types are represented by combining two or more type variables with an arrow. Coq does not only allow higher-order functions but also higher-order propositions. A predefined example is `Forall`, which features a `A -> Prop` construct from the last section.

```
Forall : forall A : Type, (A -> Prop) -> list A -> Prop
```

`Forall` takes a *property* of `A`, which returns a `Prop` for any given `A`, plus a list of `A` and returns a proposition. It works by applying the property to every element of the given list and can be proven by showing that all elements satisfy the property.

```
Example e5 : Forall (fun n => n <> 8) [2;4].
Proof.
apply Forall_cons. intros H. inversion H.
(* Forall (fun n : nat => n <> 8) [4] *)
apply Forall_cons. intros H. inversion H.
(* Forall (fun n : nat => n <> 8) [ ] *)
apply Forall_nil.
Qed.
```

`Forall` is an inductively defined proposition, which requires rules to be applied in order to prove a certain goal. This will be further explained in the next section, for now it sufficient to know that `Forall` can be proven by applying the rules `Forall_cons` and `Forall_nil`, depending on the remaining list. Because we begin with a non-empty list, we have to apply `Forall_cons`. The goal changes to `2 <> 8`, the head of the list applied to the property. We have already proven this type of inequality before, `inversion` is actually able to do most of the work we did manually by itself. Next the same procedure needs to be done for the list's tail `[4]`, which works exactly the same as before. To conclude the proof, we need to show that the property is satisfied by the empty list. `Forall_nil` covers this case, which is trivially fulfilled.

2.2.4 Inductively defined propositions

Properties of a data type can be written in multiple ways, two of which we already discussed: Boolean equations of the form `b x = true` and functions that return propositions. For example the function `InB` returns `true` if a `nat` is contained in a list, the boolean function could look like this:

```
Fixpoint InB (x : nat) (l : list nat) : bool :=
match l with
| [] => false
| x' :: l' => if (beq_nat x x') then true else InB x l'
end.
Example e5 : InB 42 [1;2;42] = true.
Proof. reflexivity. Qed.
```

Because `InB` returns a boolean value, we have to check for equality with `true` in order to get a provable proposition. The proof is fairly simple, `reflexivity` evaluates the expression and checks the equation, nothing more needs to be done.

Properties are another approach that works equally well. This definition connects multiple equations by disjunction, noted as \/. The resulting proposition needs to contain a least one true equation to become true itself.

```
Fixpoint In (x : nat) (l : list nat) : Prop :=
match l with
| [] => False
| x' :: l' => x' = x \/ In x l'
end.
Example e6 : In 42 [1;2;42].
Proof.
  simpl. (* 1 = 42 \/ 2 = 42 \/ 42 = 42 \/ False *)
  right.           (* 2 = 42 \/ 42 = 42 \/ False *)
  right.                     (* 42 = 42 \/ False *)
  left.                      (* 42 = 42 *)
  reflexivity.
Qed.
```

Proving the same example as before, we need new tactics to work with logical connectives. By simplifying the original statement we get a disjunction of equations for every element in the list. If we want to show that a disjunction is true, we need to choose a side we believe to be true and prove it. `left` and `right` keep only the respective side as the current goal, discarding the other one. A similar tactic exists for the logical conjunction /\, with the difference that `split` keeps both sides as subgoals, since a conjunction is only true if both sides are true.

The last option to describe this property is by using inductively defined propositions. As already mentioned before, inductively defined propositions consist of rules that describe how an argument can satisfy the proposition. A useful notation for representing these rules are *inference rules* . They consist of an optional list of premises that needs to be fulfilled in order for the conclusion below the line to hold.

We can describe `In` with two rules:

$$\frac{}{\texttt{In n (n :: l)}}\ 1 \qquad\qquad \frac{\texttt{In n l}}{\texttt{In n (e :: l)}}\ 2$$

Rule one states that the list's head is an element of the list. Additionally, if an element is contained in a list, it is also an element of the the same list, prefixed by another element, as described in the second rule. This definition can be transferred to Coq:

```
Inductive InInd : nat -> list nat -> Prop :=
| Head : forall n l, InInd n (n :: l)
| Tail : forall n l, InInd n (tl l) -> InInd n l.

Example e7 : InInd 42 [2;42].
Proof.
  apply Tail.  (* InInd 42 (tl [2; 42]) *)
  simpl.       (* InInd 42 [42] *)
  apply Head.
Qed.
```

The interesting part about this proof is the deductive approach. Previously we started with a proposition and constructed evidence of its truth. In this case we use `InInd`'s rules "backwards": Because we want to show that `42` is an element of `[2;42]]`, we need to argue that it is contained within the list's tail. Since it is the head of `[42]`, we can then apply `Head` and conclude that the previous statement must also be true, because we required `42` to be contained in the list's tail, which is true.

Inductively defined propositions will play an important role in the following chapters, hence some more examples:

```
Inductive Forall (A : Type) (P : A -> Prop) : list A -> Prop :=
| Forall_nil : Forall P [ ]
| Forall_cons : forall (x : A) (l : list A), P x -> Forall P l -> Forall P (x :: l)
```

We already used `Forall` in the previous section without knowing the exact definition, the rules are fairly intuitive. According to `Forall_nil`, a proposition is always true for the empty list. If the list is non-empty, the first element and the list's tail have to satisfy the proposition, as stated in `Forall_cons`, in order for the whole list to satisfy the property. This pattern can be expanded to more complex inductive propositions, `Forall2` takes a binary property plus two lists and checks if $P\ a_i\ b_i$ holds for every `i < length l`.

```
Forall2 : forall A B : Type, (A -> B -> Prop) -> list A -> list B -> Prop
```

2.3 Theory

In functional languages a data type is a classification of applicable operators and properties of its members. There are base types that store a single date and more complex types that may have multiple constructors and type variables. Typing describes the process of assigning an expression to a corresponding type in order to avoid programming errors, for example calling an arithmetic function with a character.

Typing an expression requires a context that contains data type definitions, function/constructor/operator declarations and a map that assigns types to variables. Without a context, expressions do not have any useful meaning – 42 could be typed as a number, the character 'B', a string or the answer to everything. The majority of information in a context can be extracted from the source code of a program and is continually updated while typing expressions.

In the following chapters we are going to formalize two representations of Curry programs. This process consists of:

1. Creating
   - a Coq data structure that represents the program.
   - a context that contains all necessary information for typing expressions.
2. Formalizing typing rules with inductively defined propositions.
3. Parsing Curry code to Coq programs automatically.

To represent a program in Coq, we need to list all elements it can possibly contain and link them together in a meaningful way. In case of CuMin this is relatively easy; a program consists of several function declarations, which have a signature and a body. Signatures combine quantifiers and type variables, while the body contains variables and expressions. The resulting typing rules are straightforward, because types and expressions are very specific and some procedures are simplified, for example it is not allowed to supply more than one argument to a function at a time.

While FlatCurry is designed to accurately represent Curry code and therefore has a more abstract program structure, the basic layout is similar.

```
Definition total_map (K V : Type) := K -> V.

Definition partial_map (K V : Type) := total_map K (option V).

Definition tmap_empty {K V : Type} (v : V) : total_map K V := (fun _ => v).

Definition emptymap {K V :Type} : partial_map K V := tmap_empty None.

Definition t_update {K V : Type} (beq : K -> K -> bool) (m : total_map K V) (k : K) (v : V) :=
fun k' => if beq k k' then v else m k'.

Definition update {K V : Type} (beq : K -> K -> bool) (m : partial_map K V) (k : K) (v : V) :=
t_update beq m k (Some v).
```

3 CuMin

CuMin is a simplified sublanguage of Curry with a restricted syntax that allows more concrete typing rules and data types. Although it requires some transformations to substitute missing constructs, CuMin can express the majority of Curry programs. In the following sections we will take a look at CuMin's syntaxMehner et al. [2014], create a suitable context and discuss data types, followed by the formal definition and implementation of typing rules and some examples. In the concluding section of this chapter we will see some more advanced tactics that are able to fully automate simple proofs.

3.1 Syntax

The Backus–Naur Form (BNF) is a tool to formally describe context-free grammars and languages like CuMin. A BNF definition is a set of derivation rules $S ::= A_1 | \ldots | A_n$ where S is a *nonterminal*, that is, a symbol that can be replaced by any of the n sequences A_i on the right side of the ::=. A sequence is a combination of symbols and other characters that form an expression. If a symbol occurs only on the right side of a rule, it is called a *terminal* because it cannot be replaced.

$$
\begin{aligned}
P &::= D; P \mid D \\
D &::= f :: \kappa\tau; f\overline{x_n} = e \\
\kappa &::= \forall^{\epsilon}\alpha.\kappa \mid \forall^{*}\alpha.\kappa \mid \epsilon \\
\tau &::= \alpha \mid \mathrm{Bool} \mid \mathrm{Nat} \mid [\tau] \mid (\tau, \tau') \mid \tau \rightarrow \tau' \\
e &::= x \mid f_{\overline{\tau_m}} \mid e_1\ e_2 \mid \mathrm{let}\ x = e_1\ \mathrm{in}\ e_2 \mid n \mid e_1 + e_2 \mid e_1 \stackrel{\circ}{=} e_2 \\
&\quad \mid (e_1, e_2) \mid \mathrm{case}\ e\ \mathrm{of}\ \langle (x, y) \rightarrow e_1 \rangle \\
&\quad \mid \mathrm{True} \mid \mathrm{False} \mid \mathrm{case}\ e\ \mathrm{of}\ \langle \mathrm{True} \rightarrow e_1;\ \mathrm{False} \rightarrow e_2 \rangle \\
&\quad \mid \mathrm{Nil}_\tau \mid \mathrm{Cons}(e_1, e_2) \mid \mathrm{case}\ e\ \mathrm{of}\ \langle \mathrm{Nil} \rightarrow e_1;\ \mathrm{Cons}(x, y) \rightarrow e_2 \rangle \\
&\quad \mid \mathrm{failure}_\tau \mid \mathrm{anything}_\tau
\end{aligned}
$$

Figure 3.1: Syntax of CuMin

A program P is a list of function declarations D, which contain a function name f, a list of quantifiers κ, a type τ and a function definition. Quantifiers have a tag $t \in \{\epsilon, *\}$ that determines the valid types the variable α can be substituted with. Star-tagged type variables can only be specialized to non-functional types, while ϵ allows every specialization. The notation $\overline{x_n}$ in function definitions represents n variables $x_1, ..., x_n$ that occur after the function name and are followed by an expression e. A function's type τ can consist of type variables, basic Bool or Nat types, lists, pairs and functions. An example for a function is `fst`, which returns the first element of a pair:

$$\begin{array}{ll} \text{fst} :: \forall^*\alpha.\forall^*\beta.(\alpha,\beta) \to \alpha & \qquad \text{one} :: \text{Nat} \\ \text{fst}\, p = \text{case}\, p\, \text{of}\, \langle(u,v) \to u\rangle & \qquad \text{one} = \text{fst}_{Nat,Bool}\ (1, \text{True}) \end{array}$$

Polymorphic functions need to be explicitly specialized before they are applied to another expression, as shown by the function `one`, because there is no type inference. Besides function application, expressions can be literal boolean values and natural numbers, variables, arithmetic expressions, let bindings or case constructs and constructors for pairs and lists. The two remaining expressions arise from Curry's logical parts: anything_τ represents every possible value of type τ, similar to free variables. failure_τ represents a failed computation, for example $\text{fail} = \text{anything}_{Nat} \mathrel{\mathring{=}} \text{True}$. Since anything_{Nat} can be evaluated to natural numbers only, the equation always fails because Nat and Bool are not comparable.

The Coq implementation follows the theoretical description closely. Variables, quantifiers, functions and programs are identified by an `id` instead of a name to simplify comparing values. Case expressions for lists and pairs have two `id` arguments that represent the variables x and y, that is, the head/tail or left/right component of the expression e.

```
Inductive id : Type :=
| Id : nat -> id.

Inductive tag : Type :=
| tag_star  : tag
| tag_empty : tag.

Inductive quantifier : Type :=
| for_all : id -> tag -> quantifier.

Inductive ty : Type :=
| TVar  : id -> ty
| TBool : ty
| TNat  : ty
| TList : ty -> ty
| TPair : ty -> ty -> ty
| TFun  : ty -> ty -> ty.

Definition program := list func_decl.
Inductive func_decl : Type :=
| FDecl : id -> list quantifier -> ty -> list id -> tm -> func_decl.
```

```
Inductive tm : Type :=
  | tvar   : id -> tm
  | tapp   : tm -> tm -> tm
  | tfun   : id -> list ty -> tm
  | tlet   : id -> tm -> tm -> tm
  | ttrue  : tm
  | tfalse : tm
  | tfail  : ty -> tm
  | tany   : ty -> tm
  | tzero  : tm
  | tsucc  : tm -> tm
  | tadd   : tm -> tm -> tm
  | teqn   : tm -> tm -> tm
  | tpair  : tm -> tm -> tm
  | tnil   : ty -> tm
  | tcons  : tm -> tm -> tm
  | tcaseb : tm -> tm -> tm -> tm
  | tcasep : tm -> id -> id -> tm -> tm
  | tcasel : tm -> id -> id -> tm -> tm -> tm.
```

Shown below is the definition of `fst` in Coq syntax. All names are substituted by IDs, which do not necessarily need to be distinct from each other in general but within their respective domain. Quantifier's IDs are used in the function's type to represent type variables, following the above definition. The argument IDs of the function need to appear in the following term, in this case `Id 3` is passed to a case expression. The IDs `Id 4` and `Id 5` represent the left and right side of the pair `Id 3`, of which at least one needs to occur in the next term, otherwise the function is constant.

```
FDecl (Id 0)
      [for_all (Id 1) tag_star; for_all (Id 2) tag_star]
      (TFun (TPair (TVar (Id 1)) (TVar (Id 2))) (TVar (Id 1)))
      [Id 3]
      (tcasep (tvar (Id 3)) (Id 4) (Id 5) (tvar (Id 4))).
```

3.2 Context

As mentioned in section 2.3, we need a context in order to be able to type expressions. This basic version contains no program information and stores two partial maps: One maps type variable IDs to tags, the other variable IDs to types.

```
Inductive context : Type :=
| con : (partial_map id tag) -> (partial_map id ty) -> context.
```

There are two selector functions `tagcon` and `typecon` that allow accessing the corresponding maps of a context and two update functions `tag_update` and `type_update` that add or update values.

Since the program is not part of the context, we need another way to make it accessible. One option are variables, which are introduced by writing `Variable name : type`. They can be used in place of a regular function argument, for example as shown in the predefined `map` function:

```
Variables (A : Type) (B : Type).
Variable f : A -> B.
Fixpoint map (l:list A) : list B :=
  match l with
  | [] => []
  | a :: t => (f a) :: (map t)
  end.
```

Even though `A` and `B` are not introduced as types in the signature, they can be used to parametrize lists. Likewise, `f` can be applied to arguments despite the missing function argument `map` usually has. Although functions containing variables can be *defined* this way, they are only usable outside of the own section because the variables have a type but no value. Outside of the section all variables used in a definition are appended to its type, for example `map: list A -> list B` is added both types `A` and `B` plus a function type and becomes `forall A B : Type, (A -> B) -> list A -> list B`.

3.3 Data types

$$\Gamma, \alpha^* \vdash \alpha \in \text{Data} \qquad \Gamma \vdash \text{Bool} \in \text{Data} \qquad \Gamma \vdash \text{Nat} \in \text{Data}$$

$$\frac{\Gamma \vdash \tau \in \text{Data}}{\Gamma \vdash [\tau] \in \text{Data}} \qquad \frac{\Gamma \vdash \tau \in \text{Data} \quad \Gamma \vdash \tau' \in \text{Data}}{\Gamma \vdash (\tau, \tau') \in \text{Data}}$$

Figure 3.2: Rules for being a data type

CuMin does not allow data type constructs containing functions, for example a list of functions. Instead, data types can be constructed only by combining base types, polymorphic variables and lists or pairs. There is no syntax for explicitly naming data types or creating new constructors, therefore data types exist only as part of a function signature.

```
Reserved Notation "Gamma '|-' T '\is_data_type'" (at level 40).
Inductive is_data_type : context -> ty -> Prop :=
  | D_Var  : forall Gamma n,
               (tagcon Gamma) n  = Some tag_star ->
               Gamma |- (TVar n) \is_data_type
  | D_Bool : forall Gamma, Gamma |- TBool \is_data_type
  | D_Nat  : forall Gamma, Gamma |- TNat \is_data_type
  | D_List : forall Gamma T,
               Gamma |- T \is_data_type ->
               Gamma |- (TList T) \is_data_type
  | D_Pair : forall Gamma T T',
               Gamma |- T \is_data_type ->
               Gamma |- T' \is_data_type ->
               Gamma |- (TPair T T') \is_data_type
where "Gamma '|-' T '\is_data_type'" := (is_data_type Gamma T).
```

The inductively defined proposition `is_data_type` takes a context plus a type and yields a proposition, which can be proven using the provided rules if the type is indeed a data type. Coq allows notations to be introduced before they are actually defined by adding `Reserved` to a notation. The definition is specified after the last rule, prefaced by `where`. The syntax used is $\Gamma \vdash \tau$ \is_data_type, which means that in the context Γ the type τ is a data type.

Rules follow a common structure: First, all occurring variables need to be quantified. Then conditions can be stated, followed by an assignment of a type to an expression. The rules `D_Bool` and `D_Nat` simply state that basic types are data types. `D_Var` requires type variables to have a star tag in order to be a data type because, as mentioned above, nested function types are not allowed. Lists and pairs are data types if their argument type(s) are data types.

3.4 Typing

Typing requires a set of rules that covers every valid expression and assigns corresponding types. The following inference rules are composed of typing relations $\Gamma \vdash e :: \tau$ that state the type τ of an expression e in a context Γ. The notation $\Gamma, e_1 \mapsto \tau_1 \vdash e_2 :: \tau_2$ means that e_2 can only be typed to τ_2 if Γ maps e_1 to τ_1. As mentioned in subsection 2.2.4, the premises of an inference rule above the line need to be fulfilled in order for to conclusion below to hold, that is, an expression to be typed.

$$\Gamma, x \mapsto \tau \vdash x :: \tau \qquad \Gamma \vdash \mathrm{True} :: \mathrm{Bool} \qquad \Gamma \vdash \mathrm{False} :: \mathrm{Bool} \qquad \Gamma \vdash n :: \mathrm{Nat} \qquad \Gamma \vdash \mathrm{Nil}_\tau :: [\tau]$$

$$\frac{\Gamma \vdash e_1 :: \tau_1 \to \tau_2 \quad \Gamma \vdash e_2 :: \tau_1}{\Gamma \vdash e_1 e_2 :: \tau_2} \qquad \frac{\Gamma \vdash e_1 :: \tau_1 \quad \Gamma, x \mapsto \tau_1 \vdash e_2 :: \tau}{\Gamma \vdash \mathrm{let}\ x = e_1\ \mathrm{in}\ e_2 :: \tau} \qquad \frac{(f :: \forall^{v_1}\alpha_1. \cdots \forall^{v_m}\alpha_m.\tau; f\overline{x_n} = e) \in P}{\Gamma \vdash f_{\overline{\tau_m}} :: \tau[\overline{\tau_m/\alpha_m}]}\ \begin{array}{l}\text{if for all } i \text{ with}\\ v_i = * \text{ we have}\\ \Gamma \vdash \tau_i \in \mathrm{Data}\end{array}$$

$$\frac{\Gamma \vdash e_1 :: \mathrm{Nat} \quad \Gamma \vdash e_2 :: \mathrm{Nat}}{\Gamma \vdash e_1 + e_2 :: \mathrm{Nat}} \qquad \frac{\Gamma \vdash e_1 :: \mathrm{Nat} \quad \Gamma \vdash e_2 :: \mathrm{Nat}}{\Gamma \vdash e_1 \doteq e_2 :: \mathrm{Bool}} \qquad \frac{\Gamma \vdash e_1 :: \tau_1 \quad \Gamma \vdash e_2 :: \tau_2}{\Gamma \vdash (e_1, e_2) :: (\tau_1, \tau_2)} \qquad \frac{\Gamma \vdash e_1 :: \tau \quad \Gamma \vdash e_2 :: [\tau]}{\Gamma \vdash \mathrm{Cons}(e_1, e_2)}$$

$$\frac{\Gamma \vdash e :: [\tau'] \quad \Gamma \vdash e_1 :: \tau \quad \Gamma, h \mapsto \tau', t \mapsto [\tau'] \vdash e_2 :: \tau}{\Gamma \vdash \mathsf{case}\ e\ \mathrm{of}\ \langle \mathrm{Nil} \to e_1; \mathrm{Cons}(h, t) \to e_2 \rangle :: \tau} \qquad \frac{\Gamma \vdash e :: (\tau_1, \tau_2) \quad \Gamma, l \mapsto \tau_1, r \mapsto \tau_2 \vdash e_1 :: \tau}{\Gamma \vdash \mathsf{case}\ e\ \mathrm{of}\ \langle (l, r) \to e_1 \rangle :: \tau}$$

$$\frac{\Gamma \vdash e :: \mathrm{Bool} \quad \Gamma \vdash e_1 :: \tau \quad \Gamma \vdash e_2 :: \tau}{\Gamma \vdash \mathsf{case}\ e\ \mathrm{of}\ \langle \mathrm{True} \to e_1; \mathrm{False} \to e_2 \rangle :: \tau} \qquad \Gamma \vdash \mathrm{failure}_\tau :: \tau \qquad \frac{\Gamma \vdash \tau \in \mathrm{Data}}{\Gamma \vdash \mathrm{anything}_\tau :: \tau}$$

Figure 3.3: Typing rules for CuMin

The first row of rules holds unconditionally: Basic expressions like boolean values and natural numbers have the type `Bool` and `Nat` respectively and there is an empty list of every type. Variables can only be typed if there is an entry in the context that binds the variable to a type; these bindings are created in `let` and `case` expressions.

The second row begins with the application of two expressions, which requires the first one to have a functional type and the second term to match the function's argument type. The resulting type may be another function or a basic type, depending on the arity of the original function. A `let` construct binds a variable x to an expression e_1 that is used within the expression e_2 and needs to be added to the context in order to type e_2.

The row's last inference rule describes typing a function call with specific types $\overline{\tau_m}$: The program P needs to contain a matching function declaration with a list of quantified type variables $\overline{\alpha_m}$. For every α_i the corresponding τ_i needs to be a data type if their quantifier has a star tag because we must ensure that these variables are replaced by non-functional types, which data types fulfill by definition. The type of a function call is represented by the expression $\tau[\overline{\tau_m/\alpha_m}]$, which is a type substitution of every occurrence of α_i in τ with τ_i.

The third row contains arithmetic operations and constructors. Both $+$ and $\doteq$ can only be applied to natural numbers, while the first returns a `Nat` and the latter a `Bool`. In aspect of constructors, pairs can be constructed from two expressions of arbitrary types τ_1 and τ_2, the resulting type is a pair (τ_1, τ_2). The list constructor `Cons` takes two expressions e_1 and e_2, the first of which needs to be a head element of type τ and the second a tail list of type $[\tau]$, which results in a list of τ.

Case expressions work similarly but have type specific properties. The first argument has to be of the case's type, for example `Bool` for the boolean case expressions. Depending on the constructor of the term, the corresponding branch expression is returned. The list case returns either e_1 if the list is empty or e_2 otherwise. In the latter case, bindings for the list's head and tail need to be added to the context in order to type e_2. This is also necessary in the pair case, however, since there is only one constructor, there is no choice of different terms to return. While this may be unusual for case expressions, the construct serves a purpose nevertheless: accessing a pair's individual components. The last case expression for boolean values works like an if-then-else construct; depending on the first argument either e_1 or e_2 is returned.

Finally, there is a `failure` of every type, that can be returned in place of a value if the computation fails and an `anything` of every data type. The restriction of `anything` to non-functional types is necessary because functions are not enumerable.

To implement the above rules, we begin by introducing an inductively defined proposition, similar to `\is_data_type`, with an additional argument. Since we want to assign types τ to expressions e within a context Γ, we use a ternary proposition `has_type` Γ e τ that represents the typing relation $\Gamma \vdash e :: \tau$ used above. Since `::` is the `cons` constructor in Coq, we will use \in instead. Another detail is the usage of a `Variable` to represent the program. As mentioned earlier, variables are appended to a definition's type, that is, `has_type` has the type `program -> context -> tm -> ty -> Prop`, although the below definition is missing the `program`.

```
Variable Prog : program.
Reserved Notation "Gamma '|-' t '\in' T" (at level 40).
Inductive has_type : context -> tm -> ty -> Prop :=
  | T_Var : forall Gamma x T, (typecon Gamma) x = Some T ->
              Gamma |- tvar x \in T
  | ...
where "Gamma '|-' t '\in' T" := (has_type Gamma t T).
```

Supplying arguments to a function is limited to one at a time, that is, we apply a functional expression $e_1 :: \tau_1 \to \tau_2$ to the expression $e_2 :: \tau_1$. Because we supplied e_2 with its first argument, the resulting type is the return type of the function that may be of functional type, since a recursive definition is possible.

```
T_App : forall Gamma e1 e2 T1 T2,
          Gamma |- e1 \in (TFun T1 T2) -> Gamma |- e2 \in T1 ->
          Gamma |- (tapp e1 e2) \in T2
```

The `tlet` expression has three arguments: an ID that represents the variable bound to the expression e_1 in e_2. Because we introduce a new variable x that occurs in e_2, we need to update Γ with the type of e_1 associated to x, for instance the expression `tlet (Id 0) 4 (tadd (tvar (Id 0) 8))` is only typeable if $\Gamma\,(\mathrm{Id}\,0) = \mathrm{TNat}$.

```
T_Let : forall Gamma e1 e2 x T1 T2,
          Gamma |- e1 \in T1 -> (type_update Gamma x T1) |- e2 \in T2 ->
          Gamma |- (tlet x e1 e2) \in T2
```

There are case expressions for bools, lists and pairs. The boolean case works like a if-then-else construct that returns e_1 if e is true and e_2 otherwise. Both expressions must have the same type, since the case expression would otherwise have multiple types depending on the condition. `T_CaseP` has only one case e_1, which is useful to access the first and second component of a pair by introducing variables, similar to the `tlet` expression. The last case expression works with lists and returns e_1 if e is the empty list. In case of a non-empty list, two variables for the list's head and tail are introduced and e_2 is returned. Since there are no possible variables in `nil`, this is only necessary for typing e_2.

```
T_CaseB : forall Gamma e e1 e2 T,
            Gamma |- e \in TBool -> Gamma |- e1 \in T -> Gamma |- e2 \in T ->
            Gamma |- (tcaseb e e1 e2) \in T
T_CaseL : forall Gamma e e1 e2 h t T T',
            Gamma |- e \in (TList T') -> Gamma |- e1 \in T ->
            (type_update (type_update Gamma h T') t (TList T')) |- e2 \in T ->
            Gamma |- (tcasel e h t e1 e2) \in T
```

Function specialization is the most complex rule, since it involves looking up the function's declaration in a program and checking the specialized type.

```
T_Fun : forall Gamma id tys T,
          let fd := fromOption default_fd (lookup_func Prog id) in
          specialize_func fd tys = Some T ->
          Forall (is_data_type Gamma) (fd_to_star_tys fd tys) ->
          Gamma |- (tfun id tys) \in T
```

The lookup function uses the predefined `find` that takes a boolean predicate plus a list and returns an optional of the first (and only, since functions are named uniquely) element that fulfills the predicate or `None` otherwise. An anonymous function is used to compare the function's ID to every entry's ID until a match is found. Because the search is not guaranteed to succeed, an `option func_decl` is returned.

There are two options to handle variables like `fd` in rules: quantification or let expressions. Via `forall` quantified variables are easy to work with and allow limited pattern matching, for instance `forall id fd, lookup_func Prog id = Some fd -> ...` This definitions ensures that `lookup_func` succeeds and binds the function declaration to `fd`, which saves us from using `fromOption` to extract the optional value.

The big disadvantage of this solution is the effort and redundancy arising from using quantified variables: If a variable is not found explicitly in the conclusion of a rule, that is, `Gamma |- e \in T`, it needs to be instantiated manually when applying the rule in a proof.[1] This is especially cumbersome for function declarations since this information is

[1]This is actually not completely true, there are automated tactics that are able to infer this information, as shown in section 3.6

already contained in the program. Because of this limitation, we will use `let` instead, which is not as comfortable but makes proofs significantly shorter.

The next step is to specialize the function with `specialize_func`. It takes a function declaration plus a list of types and checks if the length of the list of quantified type variables in the function declaration matches the length of the provided type list. Subsequently the substitution begins: for every pair $(\forall^t \alpha_i, \tau_i)$ the type variable α_i is replaced with τ_i in the function type τ. The substitution itself is a recursive function that takes an ID, a replacement type τ and the type τ'. Basic types and type variables with a different ID in τ' remain unchanged; if the ID of a type variable matches the provided ID, it is replaced by τ. Because types are nested structures, the substitution of functions, lists and pairs is recursively applied to the argument types. Technically, this can produce invalid substitutions since variables are replaced, regardless of their tag. Thus, this requirement is checked by the next premise.

The last condition of the rule is a `Forall` construct. We need to check that every type in `tys` is a data type if its quantifier has a star tag: `fd_to_star_tys` returns these types from a function declaration and a list of types. To check the data type property, we use `Forall` with `is_data_type Gamma`, that is, a function `ty -> Prop`. Every element of the type list is applied to the property by `Forall`; the returned propositions need do be proven when using this rule.

One last technicality needs to be mentioned: Coq does not allow notations in a section, such as used with `has_type`, to be exported. While modules allow this, they do not have the same semantics regarding variables. Hence, we need to use both in combination to circumvent this issue. Additionally, notations defined in modules cannot be exported unless they are part of a scope. Scopes are a list of notations and their interpretations, which can be named and imported.

```
Module TypingNotation.
  Notation "Prog > Gamma '|-' t '\in' T" := (has_type Prog Gamma t T)
  (at level 40) : typing_scope.
End TypingNotation.
```

This combination of sections, modules and scopes works the following way: The module is outside of `has_type`'s section and thus `has_type` has an additional `program` argument. A new notation similar to the original one with a new program parameter is defined, followed by the definition of a `typing_scope`. The notation can now be imported by writing `Import TypingNotation. Open Scope typing_scope.`
This concludes the segment about typing rules in CuMin. The following section will demonstrate the usage in proofs based on a few examples.

3.5 Examples

```
Example t3 : e_prog > empty |- (tlet (Id 5) tzero
                                (tadd (tsucc tzero)
                                      (tvar (Id 5)))) \in TNat.
Proof.
```

```
  apply T_Let with (T1 := TNat).
  apply T_Zero.
  apply T_Add.
  apply T_Succ. apply T_Zero.
  apply T_Var. reflexivity.
Qed.
```

The first example proves that `let x = 0 in 1 + x` is a natural number. Because the outermost term is a let expression, we need to apply the corresponding rule. `T_Let` has quantified variables for both expressions, their types and the variable ID. As shown in Figure 3.4, all variables can be matched to a part of the expression, except for `T1`, that is, the type of e_1.

```
T_Let : forall Gamma x e1 e2 T1 T2,

empty |- (tlet (Id 5) tzero (tadd (tsucc tzero) (tvar (Id 5)))) \in TNat.
```

Figure 3.4: Matching quantified variables with arguments

The application of `T_Let` requires manually supplied arguments because `T1` is not explicitly stated in the expression. Since `tzero` is of type `TNat`, we can tell Coq to assume the type of `T1` by writing `with (T1 := TNat)` after the tactic.

In the following proof we need to show that 0, 1 and the addition $1 + x$ are `TNat`s, which is mostly done by applying the corresponding rules. Since `T_Let` adds a binding of x (`Id 5` in Coq) to the type of e_1 to the context, we can apply `T_Var` successfully in the context `type_update empty (Id 5) TNat`.

The next example demonstrates the application $\text{app} = (\text{union}_{[Nat]}\ u)\ v$ of two variables to the function `union`. We do not consider the source of the variable bindings, they are assumed to be natural numbers, that is, $\Gamma, u \mapsto \text{Nat}, v \mapsto \text{Nat}, t_\alpha = * \vdash \text{app} :: \text{Nat}$.

$$\text{union} :: \forall^* \alpha.(\alpha \to (\alpha \to \alpha))$$
$$\text{union}\ x\ y = \text{case anything}_{\text{Bool}} \text{ of } \langle \text{True} \to x; \text{False} \to y \rangle$$

The function `union` is comparable to Curry's `?` operator, since anything_{Bool} can be either `True` or `False`; the boolean case expression becomes a non-deterministic choice between both arguments. Now we want to prove that applying two `Nat` expressions to `union` results in a natural number.

```
Definition app := tapp (tapp (tfun (Id 1) [TNat])
                             (tvar (Id 3)))
                       (tvar (Id 4)).
Example t7 : prog > cntxt |- app1 \in TNat.
Proof.
 apply T_App with (T1 := TNat). apply T_App with (T1 := TNat).
   * apply T_Fun.
```

```
    - reflexivity.
    - apply Forall_cons.
      -- apply D_Nat.
      -- apply Forall_nil.
  * apply T_Var. reflexivity.
  * apply T_Var. reflexivity.
Qed.
```

Proofs can be structured using bullet points, which makes it easier to follow the reasoning in writing. Valid options are: $+, -, *$ or a concatenation of up to 3 of the same symbols listed. When a rule generates multiple subgoals, every subgoal needs to be marked using the same bullet point.

We begin by applying `T_App` twice, once for each of the given arguments. In both cases we need to explicitly supply `T1`, that is, the type of the argument applied, for the same reason as in the previous example. The resulting subgoal list has three entries: the specialization of `union` (with ID 1 in Coq) needs to match both argument's types and the variables must be bound to the correct types in the context.

```
______________________________________(1/3)
prog > cntxt |- tfun (Id 1) [TNat] \in TFun TNat (TFun TNat TNat)
______________________________________(2/3)
prog > cntxt |- tvar (Id 3) \in TNat
______________________________________(3/3)
prog > cntxt |- tvar (Id 4) \in TNat
```

The first subgoal is a `tfun` expression and therefore we use `T_Fun` and get two additional subgoals to prove, the first of which states that `specialize_func` applied to `union`'s function declaration and `TNat` must be equal to the arguments supplied. The specialized type is computed by looking up the declaration in the program, removing the `option` from the result and substituting α with `TNat` in the function type. Since only computable functions are used, `reflexivity` solves this goal directly.

```
______________________________________(1/2)
specialize_func (fromOption default_fd (lookup_func prog (Id 1))) [TNat] =
Some (TFun TNat (TFun TNat TNat))
______________________________________(2/2)
Forall (is_data_type cntxt)
(fd_to_star_tys (fromOption default_fd (lookup_func prog (Id 1))) [TNat])
```

The second subgoal requires all argument types to be data types if the corresponding quantifier has a star tag. Since this applies to α, we need to prove that `TNat` is a data type, which is shown by `D_Nat`. We complete the first $*$-subgoal with `Forall_nil` and have two subgoals regarding variables left. Since we assumed these variables to be bound in the context initially, applying `T_Var` and `reflexivity` finishes the proof.

3.6 Automated proofs

The examples shown in the previous section have a common structure: They all work with `has_type` propositions and can be proven by applying rules of the inductive definition.

A tactic that proves an arbitrary `has_type` expression would need to try every rule but would find a match eventually. Unfortunately this is not sufficient because we already saw that some variables cannot be instantiated from the supplied expression and thus require supplying the regarding variables explicitly.

Reconsidering the last example, one may notice the following: It is possible to infer the missing types of the arguments because the specialized function type contains the arguments' types. The usage of `apply` entails that all variables need to be instantiated immediately, even if the information may appear later. We need a way to postpone assigning values to variables until it is necessary – the tactic `eapply` does precisely this.

```
______________________________________(1/3)
prog > cntxt |- tfun (Id 1) [TNat] \in TFun ?T10 (TFun ?T1 TNat)
______________________________________(2/3)
prog > cntxt |- tvar (Id 3) \in ?T10
______________________________________(3/3)
prog > cntxt |- tvar (Id 4) \in ?T1
```

Using `eapply` replaces unknown values with variables, indicated by a question mark, without the need for manual input. When we apply `T_Fun`, the type `TFun ?T10 (TFun ?T1 TNat)` is matched with `TFun TNat (TFun TNat TNat)`, that is, the specialized function type. Consequently, all necessary information was inferred automatically.

The tactic `constructor` tests every constructor of an inductive type, which works for inductively defined propositions. To automate the entire proof, we need to combine both ideas: A tactic that tries every rule and replaces unknown values with variables until they are known – `econstructor`. This is not just a combination of 'e' and constructor but an existing, powerful tactic. It is possible to prove the example by using `econstructor` eleven times, but luckily there is the tactic `repeat t` that applies the supplied tactic `t` to every subgoal and recursively to every additional generated subgoal until it fails or there is no more progress.

```
Example t7a : prog > cntxt |- app1 \in TNat.
Proof.
  repeat econstructor.
Qed.
```

This results in a fully automated, single-line proof and shows a small part of the powerful tactics Coq offers.

Summarizing this chapter, we began by transferring the formal definition of CuMin's syntax to Coq, followed by the definition of a context that contains variable bindings and tag information to enable typing of expressions. We used inductively defined propositions to describe data types and created a proposition that maps expressions to types with respect to a context. In the final sections we proved some examples and had a more detailed look at how proofs in Coq work and how they can be automated.

In the following chapter we will follow the same procedure with FlatCurry, omitting the identical parts and focussing on the differences. Furthermore, we will look into transferring Curry programs to Coq and some examples.

4 FlatCurry

FlatCurry is a flat representation of Curry code that enables meta-programming, that is, the transformation of Curry programs in Curry.Hanus [editor] et al. [2016] Since we want to reason about Curry programs in Coq, the respective module `FlatCurry.Types` will be the foundation of the Coq implementation. The generation of FlatCurry code involves two transformations: Firstly, lambda lifting is applied, that is, local function definitions, for example introduced by `where` or `let` clauses, are replaced by top-level definitions. Secondly, pattern matching is substituted by case and or expressions, the latter in case of overlapping patterns.

TODO: Kapitelüberblick

4.1 Syntax

The syntax we use in Coq is similar to the Curry code[1] and thus we discuss the Coq implementation only. Generally, Curry allows data types and constructors to have identical names, which is not possible in Coq; likewise `Type` is a reserved keyword, as we have seen multiple times.

There are three type synonyms that identify variables and other names, for example functions:

- `VarIndex`: Variables in expressions are represented by a `nat`.
- `TVarIndex`: Type variables in type expressions are also represented by a `nat` but have a different name. The distinction between the two variable types is useful to prevent mistakes.
- `QName`: A qualified name is a pair of strings: the module name and the name the function, data type, etc. Thus, the same name can occur in multiple modules and still be uniquely addressable.

We will discuss the relevant elements of the FlatCurry syntax in a top-down approach, beginning with a program. Besides its name, a program contains a list of imports and lists for type, function and operator declarations. Imports need to be handled manually when working with multiple modules.

TODO: Verlinkung praktische Nutzung

```
Inductive TProg : Type :=
| Prog : string -> list string ->
         list TypeDecl -> list FuncDecl -> list OpDecl -> TProg.
```

[1] https://www-ps.informatik.uni-kiel.de/kics2/lib/FlatCurry.Types.html

Types in FlatCurry are more abstract compared to CuMin, but there are similar elements: Type variables and function types are almost the same, except the index of the variable. We do not distinguish types and data types as we did before, therefore there is no need for an `is_data_type` property or a context that maps types to tags.

```
Inductive TypeExpr : Type :=
  | TVar     : TVarIndex -> TypeExpr
  | FuncType : TypeExpr  -> TypeExpr      -> TypeExpr
  | TCons    : QName     -> list TypeExpr -> TypeExpr.
```

The big difference is the absence of explicit types, for example `Nat` or `Bool`. Every type is represented by a `TCons` construct, which consists of a qualified name and a list of types. The latter contains type parameters, for example the expression `Left 42` has the type `Either Int a`, which is represented in FlatCurry by the qualified name `("Prelude", "Either")` and a list containing `Int` plus a type variable. Base types like `Int` do not have type parameters, hence the empty list.

```
(TCons ("Prelude","Either") [(TCons ("Prelude","Int") [] ), (TVar 0)])
```

The next construct are declarations of functions, types and constructors. They are identified by qualified names and have a visibility, which determines if the declaration is visible when the module is imported in another program. A function's arity, that is, the number of arguments, is represented by a natural number; this information is useful when working with partial function applications. It is followed by the function's type, represented by a `TypeExpr`. Lastly, rules encapsulate a list of variables and an expression, which is similar to CuMin's syntax.

```
Inductive FuncDecl : Type :=
  | Func : QName -> nat -> Visibility -> TypeExpr -> TRule -> FuncDecl.

Inductive TypeDecl : Type :=
  | Typec   : QName -> Visibility -> list TVarIndex -> list ConsDecl -> TypeDecl
  | TypeSyn : QName -> Visibility -> list TVarIndex -> TypeExpr      -> TypeDecl.

Inductive ConsDecl : Type :=
  | Cons :  QName -> nat -> Visibility -> list TypeExpr -> ConsDecl.
```

Type declarations are a new construct since it is not possible to define data types in CuMin. There are two constructors: type synonyms, for example `type IntL = [Int]`, and constructor to create new types. While both have a list of type variables, the synonym takes a type expression and the new type a list of constructor declarations.

```
data BTree a = Leaf | Branch a (BTree a) (BTree a)

Typec ("MyProg","BTree") Public  [0]
[(Cons ("MyProg","Leaf") 0 Public  [] ), (Cons ("MyProg","Branch") 3 Public
                                           [(TVar 0),
                                            (TCons ("MyProg","BTree") [(TVar 0)] ),
                                            (TCons ("MyProg","BTree") [(TVar 0)] )])]
```

The definition of binary trees has a type variable a that is represented by the number nil. The list of constructor declarations contains `Leaf`, a constructor without any arguments, and a constructor `Branch` that takes a decoration of type a and two subtrees, both of type `BTree a`.

FlatCurry expressions share some common elements with the CuMin definition: There are expressions for variables and literals, let and case constructs, the application of functions and a way to express nondeterminism. Nevertheless, most expressions are more abstract and require more complex typing rules, for example, there is only one generic case expression instead of a specific definition for every type.

```
Inductive Expr : Type :=
  | Var   : VarIndex -> Expr
  | Lit   : Literal -> Expr
  | Comb  : CombType -> QName -> list Expr -> Expr
  | Let   : list (prod VarIndex Expr) -> Expr -> Expr
  | Free  : list VarIndex -> Expr -> Expr
  | Or    : Expr -> Expr -> Expr
  | Case  : CaseType -> Expr -> list BranchExpr -> Expr
  | Typed : Expr -> TypeExpr -> Expr
```

Literals Integers, characters or floating point numbers can be literals. While there is a `nat` type in Coq, the other types cannot be represented that easily. The module `Ascii` contains data structures and notations for characters, but it is not possible to use escaped symbols like `\n` without converting them to a decimal, three digit representation, in this case 010. Similarly, the module `Reals` can represent floats, but the dot notation that FlatCurry uses (for example 1.2) is not available, instead numbers need to be written as fractions. Because we are mainly interested in typing expressions, we avoid this problem by using strings to represent chars and floats. That is not to say that an accurate representation is not feasible, but rather that it adds few value in the context of typing expressions.

Function application `Comb` represents the application of functions and constructors. Its first argument is a `CombType`, which is either a function/constructor call with all arguments provided (`FuncCall`/`ConsCall`) or a partial call (FuncPartCall/ConsPartCall). The latter has an integer value that is the number of missing arguments. The other arguments of `Comb` are a qualified name of a function or constructor and a list of expressions, that it is applied to.

```
Comb FuncCall ("Prelude","map")
     [(Comb (FuncPartCall 1) ("MyProg","double") []),
      (Comb ConsCall ("Prelude",":") [(Lit (Intc 1)),
                                      (Comb ConsCall ("Prelude","[]") [] )] )]
```

The application of `map` to `double` and `[1]` results in a complete function call of map, since all necessary arguments have been provided. Because `double` is missing its argument, the function call is partial with one argument remaining. The list `[1]` is represented by the application of `cons` to the integer literal 1 and the empty list, which is a constructor without arguments.

Let While CuMin allows only one binding in a `let` expression, this limitation is not present in FlatCurry. Multiple bindings are represented by a list of (`VarIndex`, `Expr`) pairs that bind a variable to an expression, the last argument is the expression that the bindings occur in. The expression `let` $x = y + 1$, $y = z + 2$, $z = 3$ in $x + y$ results in the following code:

```
Let [(1,(Comb FuncCall ("Prelude","+") [(Lit (Intc 3)), (Lit (Intc 2))] )),
     (2,(Comb FuncCall ("Prelude","+") [(Var 1),(Lit (Intc 1))] ))]
    (Comb FuncCall ("Prelude","+") [(Var 2),(Var 1)] )
```

FlatCurry misses unnecessary variables and bindings are sorted hierarchically, that is, if a variable occurs in another binding, its position in the list is ahead of the binding.

Case There are two instances that result in a case expression in FlatCurry: An explicit case expression and pattern matching, for example in functions. The `CaseType` is either `rigid`, in case of an explicit case, or `flexible` for transformed pattern matching. It modifies the evaluation strategy of free variables in the case expression; flexible cases use *narrowing*, which evaluates function calls with unknown arguments non-deterministically, while rigid cases delay function calls if they cannot be evaluated deterministically, which is called *residuation*.Hanus [2013] The evaluation strategy does not affect the result's type. Hence, we do not need to distinguish both cases.

TODO: Stimmt das?

The two remaining arguments are an expression that determines the branch to be taken and a list of `BranchExpr`, which consist of a pattern and an expression. Patterns may be literals or a constructor and a list of variables, as shown in the following example. The original function `fromMaybe` is transformed to a case expression in order eliminate pattern matching.

```
fromMaybe :: a -> Maybe a -> a
fromMaybe _ (Just x) = x
fromMaybe d Nothing  = d
```

```
fromMaybeCase :: a -> Maybe a -> a
fromMaybeCase d m = case m of
                      Just x  -> x
                      Nothing -> d
```

```
Case Flex (Var 2) [(Branch (Pattern ("Prelude","Just") [3] )(Var 3)),
                   (Branch (Pattern ("Prelude","Nothing") [] )(Var 1))]
```

The resulting FlatCurry code looks nearly identical for both definitions, only the `CaseType` differs because `fromMaybeCase` is an explicit case expression instead of a transformation. Unlike CuMin's `case`, this definition can be used with every type. Hence, typing a `case` expression is more complex in FlatCurry.

The remaining expressions work as expected: `Free` introduces a list of free variables in an expression, `Or` returns one of the two expressions supplied and `Typed` assigns a type to an expression.

TODO: Weglassen?

4.2 Context

The context we use for typing FlatCurry contains the familiar partial map from `VarIndex` to `TypeExpr` but is missing the `tag` map, since we do not have data types anymore. In addition to the information about variables, the context contains function and constructor declarations in form of two partial maps that map a qualified name to a pair of the full type and a list of type variables, for example `Just` has the entry ($a \to \text{Maybe } a$, $[a]$). The list of type variables simplifies specializing a function because every type variable needs to be substituted with a concrete type. To work with contexts there are selectors `vCon`, `fCon` and `cCon` to access the respective contexts and update functions to add values, similar to the CuMin context.

We want to be able to work with possibly large Curry programs. Hence, we need to create a parser that extracts the required information from a FlatCurry program and adds it to a context. Parsing function declarations is simple because it contains the function's type explicitly and we need to extract the type variables only.

```
Fixpoint extractTVars (t : TypeExpr) : list TVarIndex :=
  match t with
  | TVar i      => [i]
  | TCons _ tys => concat (map extractTVars tys)
  | FuncType argT retT => (extractTVars argT) ++ (extractTVars retT)
  end.
```

Because `extractTVars` lists every occurrence of a type variable, the result needs to be deduplicated. Together with the function type the pair is added to the context. By using `fold_right`, a list of function declarations is added to the same context.

Parsing a constructor declaration is more complicated because the type is not as easily accessible. Additionally, the declarations are contained in a type declaration and we need do incorporate this information in the constructor's type. We begin by parsing the list with the data type supplied as an additional parameter, for example `Maybe` a for the declaration of `Just`. Unfortunately the type parameters of a constructor are stored as a list, therefore we need a function that transforms a list of types to a function type, for example `[Int, Bool, Int]` to `Int -> Bool -> Int`.

```
Fixpoint tyListFunc (tys : list TypeExpr) : TypeExpr :=
  match tys with
  | [t]     => t
  | t :: ts => FuncType t (tyListFunc ts)
  | []      => TCons ("Coq","NoType") []
  end.
```

The function `tyListFunc` recursively creates function types and adds the supplied types until the list contains only one type, which is the return type of the function. In case of an empty list we need a default type because Coq enforces exhaustive pattern matching. Now we can use `tyListFunc` to assemble the full type:

```
tyListFunc (args ++ [TCons tqn (map TVar vis)])
```

The information obtained from the type declaration is its qualified name `tqn` and type variables `vis`. By constructing a `TypeExpr` with `TCons` and concatenating it to the list of type variables `args` of the constructor, we get the full type by applying `tyListFunc` to it.

```
tyListFunc ([TVar 0] ++ [TCons ("Prelude", "Maybe") (map TVar [0])])
```

In case of `Just` this results in the type $a \to \text{Maybe } a$, which is added to the context together with the list of type variables of the data type. This procedure is applied to every constructor declaration and, together with the parsed function declarations, results in a context that contains the full type and a list of type variables for every function and constructor.

4.3 Typing

TODO: :: durch $\mapsto$ bei CuMin ersetzen?

The FlatCurry typing rules are expand the concepts we discussed in the previous chapter. Functions can be applied to multiple arguments at once, there are user-defined data types, generic `case` expressions and `let` expressions with multiple bindings. This results in more complex typing rules, because conditions need to hold for multiple elements, instead of one, and are not restricted to a specific data type.

We begin by defining names for common types, for example `Int`, `Char` and `Float`, which represent `TCons ("Prelude", "Int") []` for the respective type. In the following rules the same syntax $\Gamma \vdash e :: \tau$ as before is used to express that in a context Γ the expression e has the type τ. We will discuss inference rules and, if not easily transferable, the implementation for FlatCurry expressions.

$$\Gamma \vdash \text{Lit (Intc } i) :: \text{Int} \qquad \Gamma \vdash \text{Lit (Floatc } f) :: \text{Float} \qquad \Gamma \vdash \text{Lit (Charc } c) :: \text{Char}$$

$$\Gamma, x \mapsto \tau \vdash x :: \tau$$

Figure 4.1: Typing rules for literals and variables

FlatCurry does have fewer literals than CuMin; the type of a literal depends on its argument. The other literals we defined explicitly in CuMin are replaced by `TCons` constructs, for example `TCons ("Prelude", "[]") []` represents the empty list. Typing variables works as before: If the context Γ has an entry for a variable, its type is the value returned by Γ.

$$\frac{\Gamma \vdash e_1 :: \tau_1[\ \overline{x_k \mapsto t_k}\ ] \qquad \dots \qquad \Gamma \vdash e_n :: \tau_n[\ \overline{x_k \mapsto t_k}\ ]}{\Gamma, \{f \mapsto (\tau_1 \to \dots \to \tau_n \to \tau, \overline{x_k})\} \vdash f\ e_1 \dots e_n :: \tau[\ \overline{x_k \mapsto t_k}\ ]}$$

Figure 4.2: Typing rule for function applications

TODO: CuMin Vergleich

TODO: Überschriften?

Applying a function f to n arguments $e_1 \dots e_n$ requires the arguments to have types $\tau_1 \dots \tau_n$ that match the types in the function type returned by the context. Looking up the function f, denoted by curly brackets, yields the pair $(\tau_1 \to \cdots \to \tau_n \to \tau, \overline{x_k})$ where the first component is the type of f and the second is a list of type variables, as defined in section 4.2. The notation $[\ \overline{x_k \mapsto t_k}\ ]$ represents a type substitution of the type variable x_i with the type t_i. The types t_i need to be supplied explicitly, because there is no type inference. For instance, the application of `map` to `IntToChar` and `[1,2,3]` yields the pair $((a \to b) \to [a] \to [b], [a, b])$ from the context, which results in the premises $\Gamma \vdash \texttt{IntToChar} :: (a \to b)[a \mapsto \texttt{Int}, b \mapsto \texttt{Char}]$ and $\Gamma \vdash [1, 2, 3] :: [a][a \mapsto \texttt{Int}, b \mapsto \texttt{Char}]$.

TODO: Grafik statt inline Code?

TODO: Neue Versionen von Comb einpflegen

The implementation is a set of rules in an inductively defined proposition, as before:

```
Inductive hasType : context -> Expr -> TypeExpr -> Prop :=
  T_Comb_Fun : forall Gamma qname exprs substTypes T,
                 let funcT := fromOption defaultTyVars ((fCon Gamma) qname) in
                 let specT := multiTypeSubst (snd funcT) substTypes (fst funcT)
                  in funcPart specT None = T ->
                     Forall2 (hasType Gamma) exprs (fst (funcTyList specT)) ->
               Gamma |- (Comb FuncCall qname exprs) \in T
```

Firstly, the qualified name of the function is looked up in the function context and the `option` removed, which yields the pair `funcT`. The variable `substTypes`, that is, a list of types that replace the type variables, is supplied by the user when using the rule. It is used to replace the type variables in the function type, which yields the specialized type `specT`. Now we need to determine the type T of the function application, which is computed by `funcPart`. It takes a function type plus an optional number and yields the return type of the function if no number is supplied. Otherwise `funcPart` removes the first n types of the function, for example `funcPart (Some 2) (a -> b -> c)` yields `c`. This is useful for computing the type of partial applications, in this case we need only the return type, since all arguments are supplied.

The last premise checks that the expressions have the type of the respective part of the specialized function part by transforming the specialized type into a list of types, minus the return type that is contained in the second part of the pair `funcTyList` returns. `Forall2` checks if the i-th element of the expression list has the i-th type of the type list, which represents the conditions of the inference rule in Figure 4.2.

$$\frac{\Gamma \vdash e_1 :: \tau_1[\ \overline{x_k \mapsto t_k}\ ] \qquad \dots \qquad \Gamma \vdash e_n :: \tau_n[\ \overline{x_k \mapsto t_k}\ ]}{\Gamma, \{C \mapsto (\tau_1 \to \cdots \to \tau_n \to \tau, \overline{x_k}\} \vdash C\ e_1 \dots e_n :: \tau[\ \overline{x_k \mapsto t_k}\ ]}$$

Figure 4.3: Typing rule for constructor applications

The application of a constructor works identical to a function application, which is a consequence of the way we constructed the context and the type transformation associated with adding a constructor. The only difference is the context to look up the constructor because we distinguish between constructors and functions.

$$\frac{\Gamma \vdash e_1 :: \tau_1[\ \overline{x_i \mapsto t_i}\ ] \qquad \ldots \qquad \Gamma \vdash e_k :: \tau_k[\ \overline{x_i \mapsto t_i}\ ]}{\Gamma, F \vdash f\ e_1 \ldots e_k :: (\tau_{k+1} \to \cdots \to \tau_n \to \tau)[\ \overline{x_i \mapsto t_i}\ ]} \text{ with } k < n$$

$$\text{with } F = \{f \mapsto (\tau_1 \to \cdots \to \tau_k \to \tau_{k+1} \to \cdots \to \tau_n \to \tau, \overline{x_i})\}$$

Figure 4.4: Typing rule for partial function applications

The partial application of a n-ary function f to k arguments works similar to the full application. The type of f returned by Γ is split in two parts: the first k types are checked against the argument's types and the last $n+1-k$ types represent the new type of f. The substitution types t_i need to be specified only for type variables that occur in the supplied arguments, for example a function $f :: a \to a \to b$ applied to the number 4 would require the substitution $[a \mapsto \texttt{Int}, b \mapsto b]$, because we do not want to specialize types of arguments that have not been applied yet.

```
T_Comb_PFun : forall Gamma qname exprs substTypes remArg T,
                let funcT := fromOption defaultTyVars ((fCon Gamma) qname) in
                let specT := multiTypeSubst (snd funcT) substTypes (fst funcT) in
                let     k := funcArgCnt (fst funcT) - remArg
                let argTs := firstn (@length Expr exprs) (fst (funcTyList specT))
                 in funcPart specT (Some k) = T ->
                    Forall2 (hasType Gamma) exprs argTs ->
              Gamma |- (Comb (FuncPartCall remArg) qname exprs) \in T
```

Similarly to the implementation of the full application, we begin with looking up the function type in the context and specializing it with the supplied types. As mentioned above, it is technically necessary to supply type variables in `substTypes` for every type variable that is not contained in the argument's types, however, `multiTypeSubst` works by using `zip` to pair the substitute types and type variables. If `zip` is applied to lists of different size, the longer list's tail is discarded, which is, in this case, the list of type variables that we do not want to replace.

To compute the new type of f, we need to consider the number of remaining arguments supplied in the partial function call. The function `funcArgCnt` yields the number of arguments a function type has, which we subtract the number of remaining arguments from – this results in the number k of arguments applied to the function. We previously used `funcPart` with `None` to obtain the return type of a function, now we use it with `Some n`. This removes the first k types, that is, the arguments that were supplied to the function, and yields the new type of f.

Lastly, we check that the argument's types match the types of the specialized function type. Only the first k types are relevant and thus we use `firstn` to discard the other types and use `Forall2` with the result and the supplied arguments.

$$\frac{\Gamma, x_1 \mapsto \tau_1, \ldots, x_n \mapsto \tau_n \vdash e_1 :: \tau_1 \qquad \ldots \qquad \Gamma, x_1 \mapsto \tau_1, \ldots, x_n \mapsto \tau_n \vdash e_n :: \tau_n \qquad \Gamma, x_1 \mapsto \tau_1, \ldots, x_n \mapsto \tau_n \vdash e :: \tau}{\Gamma \vdash \text{let } x_1 = e_1, \ldots, x_n = e_n \text{ in } e :: \tau} \text{ with } n > 0$$

Figure 4.5: Typing rule for let expressions

Typing `let` expressions differs, compared to CuMin, in the arbitrary number of bindings allowed. A variable x can occur in other bindings `(y = x + 1)` or even its own, for example in infinite lists `(x = [2] ++ x)`. As a consequence, context entries for the variables $x_1 \ldots x_n$ are required in order to type the expressions $e_1 \ldots e_n$ and e. While it is a syntactically valid to write `let` expressions without bindings, for example `one = let in 1`, we require at least one binding because such an occurrence is replaced by its value in FlatCurry.

```
T_Let : forall Gamma ve ves tyexprs e T,
          let vexprs    := (ve :: ves) in
          let exprs     := map snd vexprs in
          let vtyexprs := replaceSnd vexprs tyexprs in
          let Delta     := multiTypeUpdate Gamma vtyexprs
           in Forall2 (hasType Delta) exprs tyexprs ->
              Delta |- e \in T ->
          Gamma |- (Let (ve :: ves) e) \in T
```

We use pattern matching to verify that the list of bindings `vexrps`, that is, pairs of variables and expressions, is not empty. It is possible to use `length vexrps > 0` instead, but pattern matching has two advantages: The rule fails directly when applied to an empty list and, if this is not the case, the proof is shorter because we do not need to prove the inequality. As before, `tyexprs` is explicitly supplied and represents the types of the expressions used in the bindings. The function `replaceSnd` takes a list of pairs plus another list and replaces the second component of every pair with the corresponding item in the list; in this context we replace expressions with types, resulting in pairs of variables and types, which we can use to create an updated context Δ with `multiTypeUpdate`. Lastly, we check $\Delta \vdash e_i :: \tau_i$ with `Forall2` and the type of the expression e, that is, the type of the `let` expression.

$$\frac{\Gamma, \overline{x_i \mapsto \tau_{1_i}[\,\overline{v_j \mapsto t_j}\,]} \vdash e_1 :: \tau' \qquad \ldots \qquad \Gamma, \overline{x_i \mapsto \tau_{k_i}[\,\overline{v_j \mapsto t_j}\,]} \vdash e_k :: \tau' \qquad \Gamma \vdash e :: \tau[\,\overline{v_j \mapsto t_j}\,]}{\Gamma, Cs \vdash \text{(f)case e of } \{C_1\ x_1 \ldots x_n \to e_1; \ldots; C_k\ x_1 \ldots x_m \to e_k\} :: \tau'} \text{ with } k > 0$$

$$\text{with } Cs = \{C_1 \mapsto (\tau_{1_1} \to \cdots \to \tau_{1_n} \to \tau, \overline{v_i}), \ldots, C_k \mapsto (\tau_{k_1} \to \cdots \to \tau_{k_m} \to \tau, \overline{v_i})\}$$

Figure 4.6: Typing rule for case expressions

FlatCurry has one generic `case` expression compared to CuMin's specific versions for every type, which results in the most complex rule so far. The expression e can have any type with an arbitrary number of constructors, all of which need to be looked up in the context and specialized with the supplied types, followed by updating the context with new variables. A clarifying example:

```
fromMaybeCase d m = case m of
                      Just x   -> x
                      Nothing -> d
example = fromMaybeCase 0 (Just 2)
```

Looking up `Just` and `Nothing` yields $(a \to \text{Maybe } a, [a])$ and $(\text{Maybe } a, [a])$. We called the function with integer values and thus we specialize a in both types with `Int`, resulting in $(\text{Int} \to \text{Maybe Int})$ and (Maybe Int). In order to type the expression x, we need to add $x \mapsto \text{Int}$ to Γ; because we type the case expression only, we can assume d to have an entry in the context. Now both expressions x and d can be typed to $\tau' = \text{Int}$ and satisfy the conditions of the rule, that is, $(\Gamma, x \to a[a \mapsto \text{Int}] \vdash x :: \text{Int})$, $(\Gamma \vdash d :: \text{Int})$ and $(\Gamma \vdash \text{Just } 2 :: \text{Maybe } a[a \mapsto Int])$.

The implementation makes heavy use of pairs and lists. Therefore, many calls of `map`, `fst` and `snd` are necessary. Again, we use pattern matching to ensure that the `case` expression has a least one branch because, unlike `let`, the expression would otherwise not be typeable.

TODO: Camel-Case im Code

```
T_Case : forall Gamma ctype e substTypes T Tc p vis brexprs',
           let  brexprs := Branch p vis :: brexprs' in
           let   pattps := map pattSplit (brexprsToPatterns brexprs) in
           let contyvis := map (compose (fromOption defaultTyVars) (cCon Gamma))
                               (map fst pattps) in
           let     tvis := snd (fromOption defaultTyVars (cCon Gamma (fst (pattSplit p))))
           let   specTs := map (multiTypeSubst tvis substTypes)
                               (map fst contyvis) in
           let  vistysl := zip (map snd pattps)
                               (map (compose fst funcTyList) specTs) in
           let    Delta := multiListTypeUpdate Gamma vistysl
            in Forall (flip (hasType Delta) T) (brexprsToExprs brexprs) ->
               Forall (fun ty => ty = Tc) (map ((flip funcPart) None) specTs) ->
               Gamma |- e \in Tc ->
         Gamma |- (Case ctype e (Branch p vis :: brexprs')) \in T
```

The implementation might not be clear at first sight, therefore we will use the example above to illustrate the different steps. We begin with a list of branch expressions `brexprs`, that is, a pattern and an expression. `Var 3` represents the variable introduced in the `Just` pattern and `Var 1` is the default value d.

```
[(Branch (Pattern ("Prelude","Just")    [3])(Var 3));
 (Branch (Pattern ("Prelude","Nothing") [])(Var 1))]
```

The function `pattSplit` takes a pattern and yields a pair of the qualified name of the constructor and its variables. Applied to the list of patterns, which is returned by `brexprsToPatterns`, it returns the list `pattps` of (`QName`, `[TVarIndex]`) pairs.

```
[("Prelude", "Just", [3]); ("Prelude", "Nothing", [])]
```

By composing `fromOption` and `cCon` we have a function that is applied to every qualified name, that is, constructor name, yielding the list `contyvis` of types and type variable pairs.

```
[(FuncType (TVar 0) (TCons ("test", "Maybe") [TVar 0]), [0]);
                    (TCons ("test", "Maybe") [TVar 0],  [0])]
```

The type variables are equal for every constructor since they return the same type. To avoid using `head` and its default element, we use additional pattern matching in the last line to obtain the first pattern and apply the previous procedure again, yielding the list of type variables `tvis = [0]` that need to be specialized. This type specialization is applied to every constructor type, which is represented by the list `specTs`.

```
[FuncType Int (TCons ("test", "Maybe") [Int]);
              TCons ("test", "Maybe") [Int]]
```

To update the variables of the constructors in the context, they are merged with their types by using `zip`, resulting in the list `vistysl`.

```
[([3], [Int]); ([], [])]
```

Lastly, the list of pairs are added to the context by using `multiListTypeUpdate`. The function takes a list of pairs of lists, that is, a pair that contains lists of variables and the corresponding types for every constructor, and adds them to the context. The pairs represents `Just` with its variable x and `Nothing` without variables.

$$\frac{\Gamma \vdash e_1 :: \tau \qquad \Gamma \vdash e_2 :: \tau}{\Gamma \vdash e_1 \text{ or } e_2 :: \tau} \qquad \frac{\Gamma, x_1 \mapsto \tau_1, \dots, x_n \mapsto \tau_n \vdash e :: \tau}{\Gamma \vdash \text{let } x_1, \dots, x_n \text{ free in } e :: \tau} \qquad \frac{\Gamma \vdash e :: T \qquad T \Rightarrow \tau}{\Gamma \vdash (e ::: \tau) :: \tau}$$

(a) (b) (c)

Figure 4.7: Typing rules for (a) or, (b) free and (c) typed expressions

TODO: Unterschiedliche Höhe...

The typing rules for `or` and `free` expressions are short:

- The `or` expression requires both expressions to have the same type τ, so that the result's type is coherent.
- When introducing free variables $x_1 \dots x_n$ in an expression e the types $\tau_1 \dots \tau_n$ need to be added to the context on order to type e.

Typed expressions require that the stated type τ of the expression e can be specialized to T, that is, $\exists t_1 \dots t_n : T[\,\overline{x_i \mapsto t_i}\,] = \tau$ where $x_1 \dots x_n$ are type variables in τ, notated as $T \Rightarrow \tau$. This requirement is necessary because polymorphic functions, such as `id :: a → a`, can be typed to a more specific type, for example `intId = id :: Int → Int`.

The Coq equivalent of $\tau \Rightarrow \tau$ is an inductively defined proposition `isSpecializableTo` that takes two type expressions and can be used to prove that T is specializable to τ.

```
Reserved Notation "T '==>' t" (at level 40).
Inductive isSpecializableTo : TypeExpr -> TypeExpr -> Prop :=
  | T_Eq   : forall T, T  ==> T
  | T_Spec : forall t T substTypes,
               multiTypeSubst (funcTVars T) substTypes T = t ->
               T ==> t
where "T '==>' t" := (isSpecializableTo T t).
```

`T_Eq` states that any type can be "specialized" to itself, which is useful for expressions like `(42 :: Int)`, where no specialization is necessary. If we want to prove that a type T can be specialized to τ, we need to substitute all type variables in T with explicitly supplied types and the result must be equal to τ.

4.4 Conversion of FlatCurry to Coq

Reasoning about Curry programs in Coq requires a way to transform a `.curry` file into the data structure discussed in section 4.1. The module `FlatCurry.Files` provides functions to read and transform Curry files, which can be viewed in FlatCurry syntax by using the `showFlatProg` function of the `FlatCurry.ShowFlat` module. We modeled the syntax in Coq deliberately as similar as possible to the original syntax to minimize the conversion effort necessary. The differences we need to address are:

- Type declarations use the constructor `Type`, which is a keyword in Coq.
- Floating point number and character notations are not available or different.
- The " character needs to be escaped because characters are enclosed in double quotes themselves.
- List elements are separated by semicolons instead of commas.

To accomplish the above conversion one may come up with the idea of parsing the FlatCurry string directly, which requires counting brackets to distinguish list from pair separators and replacing certain substrings and characters in a large case expression. While this is a viable way of achieving the goal, it is neither an easy maintainable nor very elegant solution.

A better approach works with the FlatCurry data structure and a modified version of `ShowFlat`. Most functions in the module work by building the FlatCurry structure from string snippets and values like names and numbers. The modified functions are denoted by an appended "Coq"; we will take a look at the relevant excerpts.

```
showFlatTypeCoq :: TypeDecl -> String
showFlatTypeCoq (Type name vis tpars consdecls) =
  "\n  (Typec " ++ show name ++ showFlatVisibility vis
               ++ showFlatList show tpars
               ++ showFlatList showFlatCons consdecls ++ ")"
```

The function showFlatTypeCoq returns the string representation of a type declaration by using either show directly or an additional formatting function to transform the values into strings, which are then concatenated and prefixed by a newline and the constructor name. To prevent Coq from mistaking the constructor for a keyword, we use "Typec" instead of "Type".

```
showFlatLit (Floatc f) = "(Floatc \"" ++ show f ++ "\")"
showFlatLit (Charc  c) = "(Charc \"" ++ escQuote (filter (/='\'') (show c))
                          ++ "\")"
```

Literals are prefixed by the respective constructor and use show to create the string representing the value. In case of floats, it is sufficient to enclose the value in double quotes since we treat them as strings. Characters require slightly more effort: show returns a string, for example 'a', that contains single quotes, which we remove by filtering out the character only. Since Curry denotes strings with single quotes, they can contain unescaped double quotes. Hence, we use the function escQuote to escape every occurrence by adding another double quote.The Coq Development Team [2016]

```
showFlatListElems :: (a->String) -> [a] -> String
showFlatListElems format elems = concat (intersperse ";" (map format elems))
```

The first argument of showFlatListElems is a formatting function that prints a specific structure. Thus, showFlatListElems can transform arbitrary lists to strings by mapping the formatting function over it, interspersing a separator and concatenating the resulting list. In accordance with Coq's list notations, we separate the list's elements with semicolons, instead of commas.

The above modifications allow us to write a Curry program that transforms .curry files to Coq files. The function showFlatCoq is called with a file path s and returns an IO action that prints the transformed program.

```
showFlatCoq :: String -> IO ()
showFlatCoq s = do flatProg  <- readFlatCurry s
                   let (Prog modname _ _ _ _) = flatProg
                       flatProgS = (showFlatProgCoq flatProg)
                       coqProg   = headerString ++ importString
                                   ++ defString modname ++ flatProgS
                                   ++ ['.','\n']
                    in putStr coqProg
```

The supplied file is parsed by readFlatCurry, which results in a Prog, that is, the Curry data structure representing a FlatCurry program. We use a let expression to obtain the program name and apply the modified showFlatProgCoq function to flatProg. Then, the flat program string is concatenated with a header and import string. The imports required are the Coq definition of FlatCurry and Lists.List, that is, a module containing list notations. As usual, the program needs a definition and subsequently a name. Thus, we use defString with the modname we got earlier; the result is a string of the form `Definition modname_coq :=`. Together with the actual program and a concluding dot, we have all parts of the finished Coq program that can now be imported in other programs.

4.5 Examples

In the previous sections we have discussed the FlatCurry syntax, context and typing rules. Additionally, we created a program that transforms Curry programs into a Coq representation; now we will take a look at some examples.

```
double :: Int -> Int
double x = x + x

example1 :: [Int] -> [Int]
example1 = map double

example2 :: Bool -> Bool
example2 = id :: (Bool -> Bool)
```

```
example3 :: Int
example3 = case x of
             Just n  -> n
             Nothing -> 0
           where x free
```

The first example is the partial application of `map` to `double`. Since double has the type `Int` → `Int`, both type variables of `map` are specialized to `Int` in the resulting type. The second example shows explicit typing of the polymorphic identity function and the last example is a case expression with a free variable.
Using the transformation introduced in section 4.4 yields the following definition:

```
Definition MyProg_coq :=  (Prog "MyProg"
["Prelude"] (* imported modules *)
[] (* data type declarations *)
[ (* function declarations *)
      (* qualified name,  arity and visibility *)
(Func ("MyProg","double") 1          Public
      (* full type *)
      (FuncType Int Int)
      (* variable list and expression *)
      (Rule  [1] (Comb FuncCall ("Prelude","+") [(Var 1);(Var 1)] )));
(Func ("MyProg","example1") 0  Public
      (FuncType (List Int) (List Int))
      (Rule  [] (Comb (FuncPartCall 1) ("Prelude","map") [(Comb (FuncPartCall 1)
      ("MyProg","double") [] )] )));
(Func ("MyProg","example2") 0  Public
      (FuncType Bool Bool)
      (Rule  [] (Typed (Comb (FuncPartCall 1) ("Prelude","id") [] )
      (FuncType Bool Bool))));
(Func ("MyProg","example3") 0  Public
      Int
      (Rule  [] (Free  [1] (Case Rigid (Var 1)
        [(Branch (Pattern ("Prelude","Just") [2] )(Var 2));
         (Branch (Pattern ("Prelude","Nothing") [] )(Lit (Intc 0)))] ))))
]
[] (* operator declarations *)
```

```
).

Definition cntxt := parseProgram MyProg_coq.
```

The context is created by applying `parseProgram` to the program definition, as shown in section 4.2. In the following proofs `exp` variables represent the expression of the respective example's `Rule`. Unfortunately, the usage of `let` expressions in the definition of `hasType`'s rules creates large, flat terms that contribute few to the explanation. Therefore, we will discuss proofs in a more abstract manner.

The proof for `example1` contains two applications of the rule `T_Comb_Part`: The first is the application of `map` to `double`, but `double` itself is also a partial application, even though no arguments are supplied. A function application without arguments yields the original function and is necessary to supply functional arguments to higher-order functions. `T_Comb_Part` creates two subgoals when applied: Specializing the return type of the function with the supplied `substTypes`, that is, the types that replace the type variables, needs to match the type supplied in the proposition. The second subgoal requires the arguments to have the corresponding types of the specialized function type.

```
Definition example1 : cntxt |- exp1 \in (FuncType (List Int) (List Int)).
Proof.
  apply T_Comb_Part with (substTypes := [Int; Int]).
  * reflexivity.
  * apply Forall2_cons.
    - apply T_Comb_Part with (substTypes := []).
      -- reflexivity.
      -- apply Forall2_nil.
    - apply Forall2_nil.
Qed.
```

The first subgoal is easily proven by applying `reflexivity`. There is, however, more to it than meets the eye: `map` is looked up in the context, which yields the polymorphic type `map :: (a → b) → [a] → [b]`. The specialization `[a ↦ Int, b ↦ Int]` is applied and the first argument type removed, yielding `[a] → [b]`, that is, the expected type.

The second subgoal checks if the supplied argument matches the previously removed argument type, that is, if `double` has the type `(Int → Int)`. In order to do so, we apply `T_Comb_Part` once again, this time without any `substTypes` since `double` is not a polymorphic function. The generated subgoals are similar to the first application, with the exception that we can use `Forall2_nil` directly since we do not apply any arguments to `double`. This proves that the first and only argument of `map` matches its corresponding type. Thus, the proof is finished by applying `Forall2_nil`.

Proving the full application of functions or constructors works identical because all differences are contained in the computational part that `reflexivity` solves.

In the second example `id :: (Bool -> Bool)` we take a look at explicitly typed expressions and the proposition `isSpecializableTo`.

```
Definition example2 : cntxt |- exp2 \in FuncType Bool Bool.
Proof.
  eapply T_Typed.
    * apply T_Comb_Part with (substTypes := []);
      econstructor.
    * apply T_Spec with (substTypes := [Bool]).
      reflexivity.
Qed.
```

We begin with e-applying `T_Typed`. Without existential variables, the actual type `T` of the expression would need to be specified, but in this case it can be inferred from the line `FuncType (TVar 0) (TVar 0) = ?T` that occurs while typing `id`. Since we already used `T_Comb_Part` before, we solve the generated subgoals automatically with `econstructor`. To apply a tactic `T'` to every subgoal generated by the tactic `T`, we write `T; T'`, which avoids repeating tactics for similar subgoals.

The second subgoal states that `(a → a) ⇒ (Bool → Bool)`, that is, the actual type of `id` can be specialized to the explicitly stated type. We can prove this by applying `T_Spec` with a list of specialized types for every type variable. Now, one may ask if it is possible to use this rule without supplying the substitution explicitly since it seems fairly obvious that the type variable `a` needs to be specialized to `Bool`. The answer arises from the following situation:

```
multiTypeSubst (funcTVars (FuncType (TVar 0) (TVar 0))) ?substTypes
(FuncType (TVar 0) (TVar 0)) = FuncType Bool Bool
```

The rule `T_Spec` uses the function `multiTypeSubst` to specialize the polymorphic type. We do not know the types that replace the type variables, but we know the result. A Curry analogy could look like this:

```
replChar :: Char -> Char -> String -> String
replChar a b s = map (\c -> if (c == a) then b else c) s

replChar 'a' b "a -> a" == "B -> B" where b free
```

The function `replChar` replaces a character in a string, similar to replacing a type in a larger construct of types. Curry allows us to use a free variable in place of the replacement character and can solve the equation, that is, `b = 'B'`. Coq is able to instantiate variables in such a scenario too, as we saw in section 3.6: The equation is simplified until both sides contain similar data structures and then individual components are matched to variables, for example `?a → Int = Char → Int` would result in the instantiation of `?a` with `Char`.

The actual cause of the problem is the definition of `multiTypeSubst` or rather that it replaces multiple variables. The above Curry function replaces only one character, a definition equal to `multiTypeSubst` that replaces multiple characters in a string could look like this:

```
replChars :: [Char] -> [Char] -> String -> String
replChars cs ds s =  foldr (\cd t -> case cd of (c,d) -> replChar c d t)
```

```
                          s (zip cs ds)

replChars ['a', 'b'] cs "a -> b" == "I -> C" where cs free
```

Curry is still able to solve the equation, but the result `cs = ('I':'C':_xk)` contains a free variable because the substitution discards the remaining tail of the longer list. Therefore, the result is ambiguous! Coq needs to instantiate the variable with a specific value, but since choosing one solution could lead to multiple outcomes depending on the choice, the proof fails. Therefore, we need to supply the `substTypes` and can finish the proof with `reflexivity`.

The last example, that is, the case expression with a free variable, expands on the usage of automation in proofs. As we already discussed, there are some limitations that require explicitly supplying some values, but the proof is still shortened significantly by using the `;` and `econstructor`.

```
Definition example3 : cntxt |- exp3 \in Int.
  Proof.
  apply T_Free with (tyexprs := [TCons ("Prelude", "Maybe") [Int]]);
  eapply T_Case;
  try instantiate (1 := [Int]);
  repeat econstructor.
Qed.
```

Typing an expression e containing free variables requires explicitly supplying the type of the variables when applying `T_Free`. In this case the return type is `Int` and the `case` patterns are `Maybe a` constructors. Hence, the free variable must be of type `Maybe Int`. In order to type e, that is, a `case` expression, we use `eapply` with `T_Case` which introduces two existential variables `?substTypes` and `?Tc`. Although we know that `?substTypes` needs to be explicitly supplied, it is better to use `eapply` nevertheless because `?Tc`, that is, the type of the free variable we already typed above, can be inferred.

Now we need to supply a value for `?substTypes` after the application of `T_Case`. The tactic `instantiate` instantiates existential variables but works with numbers instead of variable names to simplify automation. The variables are enumerated by their first occurrence from right to left in the expression, which may be hard to perceive in large expressions. Fortunately, the tactic `Show Existentials` prints a list of all existential variables and the corresponding types.

```
Existential 1 = ?substTypes : [ |- list TypeExpr]
Existential 2 = ?Tc : [ |- TypeExpr]
```

Thus, we know that `?substTypes` has the number 1 and can instantiate it with the type `Int` to specialize `Maybe a`. We already used `;` before, but in this case we use it between all tactics. This applies every tactic to all subgoals generated by the previous tactic, which concludes the proof with a single application of `repeat econstructor`. There are downsides to this approach, however: `T_Case` generates three subgoals, all of which `instantiate` would be applied to. Existential variables are instantiated globally, that is, in all subgoals, and would therefore fail. In cases where a tactic is only useful for some subgoals, `try` allows the proof to proceed even if the application of the tactic fails.

5 Conclusion

Bibliography

Sergio Antoy and Michael Hanus. Curry: A Tutorial Introduction, 2014.

M. Hanus. Functional Logic Programming: From Theory to Curry. In *Programming Logics - Essays in Memory of Harald Ganzinger*, pages 123–168. Springer LNCS 7797, 2013.

Michael Hanus [editor], Bernd Braßel, Björn Peemöller, and Fabian Reck. KiCS2: The Kiel Curry System (Version 2) User Manual, August 2016.

Stefan Mehner, Daniel Seidel, Lutz Straßburger, and Janis Voigtländer. Parametricity and Proving Free Theorems for Functional-Logic Languages. December 2014. URL `https://hal.inria.fr/hal-01092357;https://hal.inria.fr/hal-01092357/document;https://hal.inria.fr/hal-01092357/file/ppdp2014-final.pdf`.

Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, and Brent Yorgey. *Software Foundations*. Electronic textbook, 2016. Version 4.0. `http://www.cis.upenn.edu/~bcpierce/sf`.

The Coq Development Team. *The Coq Proof Assistant Reference Manual – Version V8.5pl2*, 2016. URL `https://coq.inria.fr/refman/`. `https://coq.inria.fr/refman/`.