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 25, 2016

Erklärung der Urheberschaft

Ich erkläre hiermit an Eides statt, dass ich die vorliegende Arbeit ohne Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Aus fremden Quellen direkt oder indirekt übernommene Gedanken sind als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form in keiner anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.

Ort, Datum	Unterschrift

Contents

List of Figures

1. Introduction

Does a program behave as intended? This question is a fundamental part of the formal verification of software. One aspect of formal verification is deductive verification, that is, formally representing and proving properties about programs.[Filliâtre, 2011] Interactive theorem provers, such as Coq, offer a specification language to represent programs and verification logic to argue about properties of the program. Figure 1.1 shows CoqIde, a graphical interface for Coq. The specification is written on the left-hand side; the right-hand side contains an overview of the remaining goals in a proof.

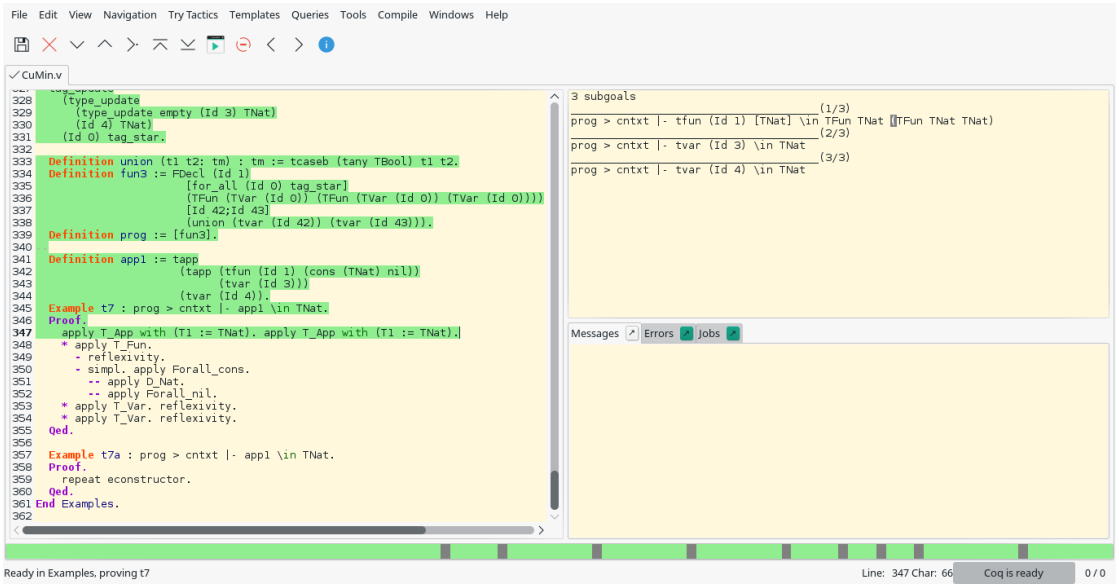

Figure 1.1.: CoqIde, a graphical interface for Coq

In the following chapters, we will use Coq to formalize inference systems, particularly type systems for the functional logic programming language Curry. Type systems are an important concept of strongly typed languages like Curry. Every function and expression has a specific type that, if not explicitly stated, is inferred and checked by the compiler. The process of typing expressions follows a set of inference rules that define conditions, which need to be met in order for an expression to be of a certain type. The formalization of these rules is one aspect that is presented in the following chapters.

The goal of this thesis is to find an appropriate representation of Curry code in Coq and to implement the inference rules of the type system in order to verify properties of Curry

code and to provide a basis for the formalization of further aspects of Curry.

This thesis is structured the following way: We begin with preliminaries in chapter 2, which includes a basic introduction to Curry and Coq. The last section of the chapter gives a theoretical overview of the approach used in the following chapters.

Both chapters 3 and 4 are structured similarly: Chapter 3 features the language CuMin, a minimal representation of Curry mostly used in theoretical contexts, while chapter 4 focuses on FlatCurry, a more practical representation that is used as an intermediary language for some Curry compilers. The common approach in these chapters includes discussing and implementing the syntax of both languages and the type system. Chapter 3 ends with a few examples and a section about automated proofs, while chapter 4 is concluded with a program that transforms Curry programs into a Coq-compatible form.

The last chapter presents the results and possible expansion thereof, as well as related work.

2. Preliminaries

This chapter introduces the two main programming languages used in this thesis: Coq and Curry. The introduction of Coq is based on *Software Foundations* by Pierce et al. [2016] and gives an overview of modeling and proofs in Coq. In case of Curry, we focus on the extended features compared to Haskell.

2.1. Coq

The formalization of Curry programs requires a language that allows us to express the code itself and propositions we intend to prove. Coq is an interactive proof management system created by The Coq Development Team [2016] that meets these requirements with its functional modeling language and integrated verification logic.

We already saw CoqIde in Figure 1.1, but this is not the only Coq front-end. There is also the Emacs package *Proof General*[1] that integrates multiple interactive theorem provers like Coq into the text editor.

Regardless of the interface chosen, the structure is similar to the following: The left-hand side contains the code, while the right-hand side shows information about proofs and error messages. Definitions in Coq are always completed by a dot that marks a segment. Depending on the front-end, there may be buttons or shortcuts to load one segment at a time, multiple segments or the whole file. A loaded segment is usually highlighted green, which means that the definition contains no errors.

Because of this approach, definitions that reference other definitions need to be in the correct order, that is, a function or data structure needs to be defined before it can be used in an expressions.

2.1.1. Data types and functions

Coq's predefined definitions, contrary to e.g. Haskell's Prelude, are rather limited. Nevertheless, being a functional language, there is a powerful mechanism for defining new data types. The following example shows the definition of a polymorphic list.

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

The definition begins with the keyword `Inductive`, followed by the name of the data type and its arguments, that is, a `Type` in this case. The return type, marked by a colon,

[1] https://proofgeneral.github.io/

is also a `Type`. The inductive definition `list` has 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`.

```
> Check cons nat 8 (nil nat).
cons nat 8 (nil nat) : list nat
```

This is required because the list is supposed to be polymorphic, 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 in the original definition or using `Arguments` followed by the name of the function and the arguments to do this afterwards.

```
Arguments nil {X}.
Arguments cons {X} _ _.

> Check cons 8 nil.
cons 8 nil: list nat
```

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`[2], because Coq cannot infer the implicit type variable of `double_cons`, since `cons` does not have a 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 nil.
double_cons 2 4 nil : list nat
> 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 definition of lists we can write a polymorphic function that determines if a list is empty, as shown in the next example.

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

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

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` $x_0, ..., x_m$ `with` b `end.` where $m < n$ and b is a list of branches. Because the arguments need to be explicitly stated in order to be pattern matched, it is possible to use pattern matching only on a subset of the arguments. Multiple patterns are separated by commas and thus, patterns do not need to be enclosed in brackets.

A branch contains a pattern and an expression. Multiple branches are separated by vertical lines and need to be exhaustive, that is, cover every constructor of the argument. If there are overlapping patterns, Coq prints an error message stating that the respective clause is redundant.

A recursive function must be explicitly marked by using `Fixpoint` instead of `Definition`. 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. In the following example `app` appends two lists. As mentioned before, only the first list is pattern matched because we do not need to access the second list's structure.

```
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. Sometimes it is necessary to state the level, that is, the priority when used with other notations, and associativity of the notation.

```
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).
```

2.1.2. Propositions and proofs

Every claim that we state or want to prove has the type `Prop`. Propositions can be any statement, regardless of its truth.

```
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 the universal quantifier `forall`. This allows us to state propositions about, for example, every type of list, or, as shown in the third example, about every natural number greater than

zero. Combined with implications that are represented by the symbol `->` we can premise specific properties that limit the set of elements the proposition applies to. The last example contains an anonymous function, which is indicated by the keyword `fun` followed by variable binders. The expression after the `=>` can be a simple expression or pattern matching, for example `(fun p => match p with (x,y) => x end)`.

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.
```

After naming the proposition as an example, theorem or lemma it appears in the interactive subgoal list that Coq provides.

```
1 subgoal
______________________________________(1/1)
1 + 1 = 2
```

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` and is used with the names of the variables that should be introduced. The new context contains a type `X` and a list `l`and the remaining goal.

```
1 subgoal
X : Type
l : list X
______________________________________(1/1)
[ ] ++ 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 only useful for obvious equations, it also simplifies and unfolds definitions until the flat terms match each other if possible, otherwise it fails with an error message.

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, that is, showing that every constructor of a data type satisfies the proposition.

```
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 the 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 subgoal. For every constructor a subgoal is generated, ordered as listed in the definition of the data type.

```
X : Type
______________________________________(1/2)
[ ] ++ [ ] = [ ]
______________________________________(2/2)
(l :: ls) ++ [ ] = l :: ls
```

Now we need to prove that the proposition holds for two subgoals: the empty list and a cons construct. The first subgoal does not require any new variable names since `l` is replaced by the empty list. Therefore, the first section in the squared brackets is empty. The subgoal 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.

```
X : Type
l : X
ls : list X
IH : ls ++ [ ] = ls
______________________________________(1/1)
(l :: ls) ++ [ ] = l :: ls
```

The remaining proof is finished by using `simpl` and `rewrite IH`. The tactic `rewrite` changes the current goal by replacing every occurrence of the left side of the provided equation with the right side.

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

Both sides are equal now and therefore `reflexivity` proves the last subgoal.

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 to the function returns a verifiable inequality. Therefore, we apply the function to 1 and prove that $1 \neq 2$. This proof is not as simple as the other ones, mainly because of the inequality, which is a notation[3] for `not (x = y)`.

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

```
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.
```

Because `not` is the outermost term, we need to eliminate it first by applying `unfold`, which 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 `P -> 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.

After using `simpl` and `unfold not`, the current goal is `1 = 2 -> False`. Introducing `1 = 2` as an hypothesis `H` with `intros` leaves `False` as the remaining goal. Intuitively we know that `H` is false, but Coq needs a justification for this claim. Conveniently, the tactic `inversion` solves this problem 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, using `inversion` with the hypothesis `2 = 1` infers the current goal `False`, which concludes the proof.

2.1.3. 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 like `map`, which applies a function to every element of a list.

```
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`, that is, a function that 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.

2.1.4. 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 `f x = true` and functions that return propositions. For example, the function `inB` returns `true` if a `nat` is contained in a list.

```
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. The following definition connects multiple equations by disjunction, noted as `\/`. This operator is not the boolean disjunction, but a disjunction of propositions. It can be proven by showing that either one of the arguments is true. In the example at least one equation needs to be true in order for the whole disjunction to be true.

```
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.

```
Inductive InInd : nat -> list nat -> Prop :=
  | head : forall n l, InInd n (n :: l)
  | tail : forall n l e, InInd n l -> InInd n (e :: l).
```

The rule `head` states that the list's head is an element of the list. The second rule `tail` covers the case that if an element is contained in a list, it is also an element of the same list prefixed by another element.

```
Example e7 : InInd 42 [2;42].
Proof.
  apply Tail.  (* 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 every element in the list's tail have to satisfy the property, 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 like `Forall2`, which takes a binary property plus two lists a, b and checks if $P\ a_i\ b_i$ holds for every `i < length l`.

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

To summarize this section, we began with inductive data types that we expanded to have implicit type arguments. We defined functions and discussed the restrictions Coq enforces in order to prevent infinite loops, followed by notations that allow us to use list constructors conveniently. Then, we discussed propositions and proved some examples, in which we used some more advanced tactics like `induction` and `inversion`. Lastly, we learned about higher-order constructs and inductively defined propositions like `Forall`.

2.2. Curry

CurryHanus [editor] et al. [2016] 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. Curry's syntax is similar to the functional programming language 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 an (interactive) compiler named KiCS2[4] that compiles Curry programs to Haskell code and an easily searchable[5], 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, for example `double :: Int → Int → Int`, every function has a type that describes the arguments and the result of evaluating the function.

[4] https://www-ps.informatik.uni-kiel.de/kics2/
[5] https://www-ps.informatik.uni-kiel.de/kics2/currygle/

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 the infinite list of twos `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.

Besides functions, there is another important element of functional programming: data types. A data type is generally of the form `data Name` $v_1 \dots v_n$ `=` $C_1 T_1 | \dots | C_m T_m$ where v_i are type variables, C_i are constructor names and T_i are lists of types, which can include the type variables v_i. Simple data types like `Bool` do not have type variables, but polymorphic types like `Maybe` can contain any data type to represent optional values.

```
data Bool = True | False
data Maybe a = Nothing | Just a
```

Nondeterminism More advanced functions use multiple 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 strictly necessary, 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, as shown in the function `maybeNotCase`.

Free Variables Unknown values are represented in Curry by free variables, that is, variables that are not bound to an expression. They are declared by adding `where` $v_1 \dots v_n$ `free` to a term containing free variables $v_1 \dots v_n$. To evaluate such a term, the variables are instantiated with a value of appropriate type.

```
> False && x where x free
{x = _x0} False
```

In the above example, the boolean conjunction of `False` and a free variable evaluates to another free variable `_x0` 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 the corresponding 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[Hanus, 2013]. 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.

In summary, we looked at the basic structure of a Curry program and function definitions. Regarding functions, we discussed lazy evaluation and non-determinism as well as its consequences when evaluating functions. Finally, we used free variables and learned about the different approaches to evaluating such expressions: narrowing and residuation.

2.3. Theory

In this section we will discuss the theoretical basics of the following chapters. First and foremost, what is typing? 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 declarations and a map that assigns types to variables. Without a context, expressions do not have any useful meaning – `A 42` could be typed as a character and a number, the application of a function `A` to 42, a variable, etc. 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 the typing systems of 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. Transferring programs into Coq syntax and using typing rules to prove propositions about the program.

Representation 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 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 simple 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. A program consists of a name, imported modules and declarations of functions, data types and operators. The latter two are not present in CuMin, but function declarations have a similar structure with additional information, such as the function's arity and visibility.

Types and expressions in FlatCurry are not as specific as in CuMin, for example, CuMin has an explicitly defined data type `Bool` with a `case` expression that works with `Bool` values only, while FlatCurry uses the data type constructor to represent the `Bool` type. FlatCurry's `case` expression works with arbitrary values and therefore the typing rules are more complex.

Context Both CuMin and FlatCurry need a context that maps keys to values, that is, variables to types or names to declarations. There are many options to implement a map, such as a list of pairs, but we will use functions instead because it resembles the formal notation we will use later more closely.

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

A total map is a function that takes a key value of type `K` and returns a value of type `V`. Because we define maps for arbitrary keys and values, the same definition can be used for CuMin and FlatCurry. A context contains only information about certain variables or functions, not for every possible key value. Thus, we define a partial map by adding an `option` to the value, that is, known keys yield `Some v` and others `None`.

An empty total map is a function that returns the same value `v` for every key. In case of the partial map, the default value is `None`.

```
Definition emptytmap {K V : Type} (v : V) : total_map K V := (fun _ => v).
Definition emptymap {K V :Type} : partial_map K V := emptytmap None.
```

Updating a total map works by calling `t_update` with a boolean equality function `beq` between keys, a total map `m`, key `k` and value `v`. To update a partial map with the value `v`, we use `t_update` with `Some v`.

```
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).
```

The result of `t_update` is a function that takes a key `k'` and applies `beq` to `k'` and `k`. If the keys are equal, the value is returned, otherwise the key is applied to the map `m`.

```
pmap = update beq_nat 1 Int emptymap
--> fun k' => if (beq_nat 1 k') then (Some Int) else (emptymap k')
pmap2 = update beq_nat 2 Char pmap
--> fun k' => if (beq_nat 2 k') then (Some Char) else (pmap k')
```

This implementation of a map is constructed by linking multiple functions together. Each function compares the supplied key with its key and if they differ, passes it to the next function. When looking up a key in a map, the key is compared to the last added entry first. An unknown key is checked by every function until finally `emptymap` returns `None`.

Inference rules In the following chapters we will work with typing rules, that is, a rule that states conditions which need to be satisfied in order for an expression to have a certain type. In Coq we use such rules as part of an inductively defined proposition, but this usually includes implementation-specific details. Therefore, we use a more formal notation: *inference rules*. An inference rule consists of an optional list of premises that needs to be fulfilled in order for the conclusion below the line to hold, similar to an implication.

```
Inductive InInd : nat -> list nat -> Prop :=
  | gead : forall n l, InInd n (n :: l)
  | tail : forall n l e, InInd n l -> InInd n (e :: l).
```

We can describe the inductively defined proposition `InInd` we used in subsection 2.1.4 that represents the occurrence of an element in a list with two inference rules as shown in Figure 2.1.

$$\frac{}{\texttt{In n (n :: l)}}\ \texttt{In\_H} \qquad\qquad \frac{\texttt{In n l}}{\texttt{In n (e :: l)}}\ \texttt{In\_T}$$

Figure 2.1.: Inference rules for occurrence of an element in a list

`In_H` does not have a premise because a list's head is always an element of the list. Thus, the line can be omitted. The rule `In_T` states that if the premise `In n l` is true, then the conclusion `In n (e :: l)` holds. Inference rules can have multiple premises $p_1 \dots p_n$ that are treated as multiple implications $p_1 \to \dots \to p_n \to c$ with the conclusion c.

Even though the line may imply that premises and the conclusion are independent from each other or that there is a chronological order, this is not necessarily the case. The premises can contain variables that are bound to a value only after evaluating the conclusion.

TODO: Beispiel

3. CuMin

CuMin (for *Curry Minor*) is a simplified sublanguage of Curry introduced by Mehner et al. [2014]. The syntax of CuMin is restricted and thus allows more concrete types and typing rules, compared to FlatCurry. 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 syntax, create a suitable context and discuss data types, followed by the formal definition and implementation of typing rules as well as 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 formalism to 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-hand side. A sequence is a combination of symbols and other characters that form an expression. If a symbol occurs only on the right-hand side of a rule, it is called a *terminal* because it cannot be replaced. Figure 3.1 shows the syntax of CuMin in BNF.

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

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 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 types like Bool or Nat, lists, pairs and functions. The following example shows function definitions for `fst`, which returns the first element of a pair, and the application of `fst` to a pair.

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

Polymorphic functions like `fst` need to be explicitly specialized before they are applied to another expression, as shown by the function `one`, because there is no type inference. Both type variables α and β in the signature of `fst` need to be specialized explicitly. Thus, `fst` is called with Nat and Bool, that is, the types of the first and second component of the pair.

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 $\text{anything}_{Nat} + 2 \stackrel{\circ}{=} 0$. Since anything_{Nat} can be evaluated to positive numbers only, the equation always fails.

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 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 CuMin expressions. This basic version contains no program information and stores two partial maps: the first 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 `tag_con` and `type_con` 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 local 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. The scope of a variable is limited by the *section* it is defined in. Sections are used to structure source files and to allow local declarations. In this case, `map` is contained in the section `Map`.

```
Section Map.
  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.
End Map.
```

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 variables have a type but no value. Outside of the section all variables used in a definition are appended to the type, for example, the type of `map` has additional type and function arguments outside of the section `Map`.

```
> Check map. (* inside of section *)
map : list A -> list B

> Check map. (* outside of section *)
map : 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 basic 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 'in_Data'" (at level 40).
Inductive is_data_type : context -> ty -> Prop :=
  | D_Var  : forall Gamma n,
               (tag_con Gamma) n  = Some tag_star ->
               Gamma |- (TVar n) in_Data
  | D_Bool : forall Gamma, Gamma |- TBool in_Data
  | D_Nat  : forall Gamma, Gamma |- TNat in_Data
  | D_List : forall Gamma T,
               Gamma |- T in_Data ->
               Gamma |- (TList T) in_Data
  | D_Pair : forall Gamma T T',
               Gamma |- T in_Data ->
               Gamma |- T' in_Data ->
               Gamma |- (TPair T T') in_Data
where "Gamma '|-' T 'in_Data'" := (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 $\Gamma \vdash \tau \setminus \text{is\_data\_type}$, 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 application of the inductively defined proposition to variables or values.

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 nested function types are not allowed. The last two rules state that 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 section 2.3, the premises of an inference rule above the line need to be fulfilled in order for the conclusion below to hold, that is, an expression to be typed.

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

$$\frac{\Gamma \vdash e_1 :: \tau_1 \to \tau_2 \quad \Gamma \vdash e_2 :: \tau_1}{\Gamma \vdash e_1 e_2 :: \tau_2} \qquad \frac{\Gamma \vdash e_1 :: \tau_1 \quad \Gamma, x \mapsto \tau_1 \vdash e_2 :: \tau}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 :: \tau} \qquad \frac{(f :: \forall^{v_1}\alpha_1. \cdots \forall^{v_m}\alpha_m.\tau; f\overline{x_n} = e) \in P}{\Gamma \vdash f_{\overline{\tau_m}} :: \tau[\overline{\tau_m/\alpha_m}]}\ \bigstar$$

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

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

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

$\bigstar$ if for all i with $v_i = *$ we have $\Gamma \vdash \tau_i \in \text{Data}$

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` or `Nat` respectively and there is an empty list of every type as well. 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. Using the variable within the expression e_2 requires the binding to be added to the context.

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 $\hat{=}$ can only be applied to natural numbers; $+$ returns a `Nat` and $\hat{=}$ 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 have specific properties depending on the type. 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 Curry does not allow free variables in place of functions.

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 ::: 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 ':::' T" (at level 40).
Inductive has_type : context -> tm -> ty -> Prop :=
  | T_Var  : forall Gamma x T, (type_con Gamma) x = Some T ->
             Gamma |- tvar x ::: T
  | T_True : forall Gamma, Gamma |- ttrue ::: TBool
  | ...
where "Gamma '|-' t ':::' T" := (has_type Gamma t T).
```

The first rules reflect the simplicity of the formal counterpart. `T_Var` looks up a variable's type in the type context, the result must be some value `T` in order for the variable to

have the type `T`. Rules like `T_True` without premises are, apart from the notation, equal to the corresponding inference rule. Therefore, we will omit these rules.

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 ::: (TFun T1 T2) -> Gamma |- e2 ::: T1 ->
          Gamma |- (tapp e1 e2) ::: 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 bound 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 ::: T1 -> (type_update Gamma x T1) |- e2 ::: T2 ->
          Gamma |- (tlet x e1 e2) ::: 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 ::: TBool -> Gamma |- e1 ::: T -> Gamma |- e2 ::: T ->
            Gamma |- (tcaseb e e1 e2) ::: T
T_CaseL : forall Gamma e e1 e2 h t T T',
            Gamma |- e ::: (TList T') -> Gamma |- e1 ::: T ->
            (type_update (type_update Gamma h T') t (TList T')) |- e2 ::: T ->
            Gamma |- (tcasel e h t e1 e2) ::: 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) ::: 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, a value of type `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 ::: T`, it needs to be instantiated manually when applying the rule in a proof.[1] This restriction is especially cumbersome for function applications since the declaration is already contained in the program but needs to be explicitly supplied anyway. 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`, that is, replacing the type variables of the function type with the supplied types. `specialize_func` 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. If both lists have the same length, the substitution begins.

```
Definition specialize_func (fd : func_decl) (tys : list ty) : option ty :=
  match fd with
  | (FDecl _ qs t _ _) => if (beq_nat (length qs) (length tys))
                         then Some (multi_ty_subst (zip qs tys) t)
                         else None
  end.
```

The substitution `multi_ty_subst` works as follows: 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 `ty_subst` 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.

```
Fixpoint ty_subst (k: id) (t: ty) (t': ty) : ty :=
  match t' with
  | TVar i      =>  if (beq_id i k) then t else TVar i
  | TBool       => TBool
  | TNat        => TNat
  | TList T     => TList (ty_subst k t T)
  | TPair TF TS => TPair (ty_subst k t TF) (ty_subst k t TS)
  | TFun  TA TR => TFun  (ty_subst k t TA) (ty_subst k t TR)
  end.
```

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

```
Definition multi_ty_subst (qtys : list (quantifier * ty)) (t : ty) : ty :=
  fold_right (fun qty t => match qty with
                           | (for_all id _, ty) => ty_subst id ty t
                           end)
             t qtys.
```

Technically, the substitution can produce invalid types since variables are replaced, regardless of their tag. While this could be checked by `ty_subst`, we use a more explicit way in the last condition of the rule, that is, a `Forall` construct. We need to check that every type in `tys` is a data type if its quantifier has a star-tag. The function `fd_to_star_tys` takes a function declaration plus a list of types and returns the types whose matching quantifiers have a star-tag, for example, $[\forall^{\epsilon}\alpha, \forall^{*}\beta]$ and [Int, Bool] would result in the list [Bool] because only β has a star-tag.

To check the data type property, we use `Forall` with `is_data_type Gamma`, that is, a function `ty -> Prop`. Every element of the list of `ty` is applied to the property by `Forall`; the returned propositions need do be proven when using this rule.

Exporting Notations Coq does not allow notations in a section, such as used with `has_type`, to be exported. While notations in a modules can be exported, 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.

```
Section Typing.
  Reserved Notation "Gamma '|-' t ':::' T" (at level 40).
  Inductive has_type : context -> tm -> ty -> Prop :=
    | T_Nil : forall Gamma T, Gamma |- (tnil T) ::: (TList T)
    | ...
  where "Gamma '|-' t ':::' T" := (has_type Gamma t T) : typing_scope.
End Typing.

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

Import TypingNotation. Open Scope typing_scope.
Section Examples. ...
```

This combination of sections, modules and scopes works the following way: The module `TypingNotation` 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 and used in other sections or programs.
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

In this section we take a look at some examples that demonstrate the usage of the the typing rules we discussed above. The first example proves that `let x = 0 in 1 + x` is a natural number. Since there are no functions in this expression that could potentially need to be looked up, we use the empty program `e_prog`. Furthermore, because there are no external bindings necessary to type the expression, the context is initially empty too.

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

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.

The outermost term of the example is a let expression. Therefore, we need to apply the corresponding rule `T_Let`. It 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.

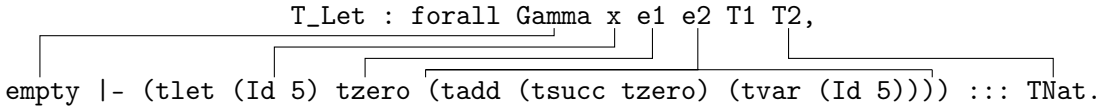

Figure 3.4.: Matching quantified variables with arguments

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.

```
______________________________________(1/2)
e_prog > empty |- tzero ::: TNat
______________________________________(2/2)
e_prog > type_update empty (Id 5) TNat |- tadd (tsucc tzero)
                                                (tvar (Id 5)) ::: TNat
```

The application of `T_Let` generates two subgoals that we need to prove, that is, show that 0 and $1 + x$ have the type `TNat`. We apply `T_Zero`, a rule without premises, to

directly prove the first goal. The remaining subgoal is the addition $1 + x$, but compared to the first subgoal, the context does contain an entry for the variable with the ID 5. This arises from the definition of the rule: The type we supplied for `x` and proved to be correct in the first subgoal is now added to the context in order to use the variable in the addition, which requires the application of `T_Add`. Two new subgoals are generated:

```
_____________________________________(1/2)
e_prog > type_update empty (Id 5) TNat |- tsucc tzero ::: TNat
_____________________________________(2/2)
e_prog > type_update empty (Id 5) TNat |- tvar (Id 5) ::: TNat
```

An addition has the type `TNat` if both summands have, which is expressed by the subgoal list. The first subgoal states that `S(O)` is a `TNat`. Because the successor of zero is also a `TNat` by definition, we apply `T_Succ` and `T_Zero` to prove this goal. Lastly, we need to prove that the variable, which was added to the context, also has the type `TNat`. Applying `T_Var` yields a type lookup of the variable with the ID 5, which is results in the expected type `TNat` and is therefore proven by applying `reflexivity`.

The next example demonstrates the application app $=$ (union$_{[Nat]}$ u) v of two variables to the function `union`.

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

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)).
Definition cntxt := tag_update
                     (type_update
                       (type_update empty (Id 3) TNat)
                     (Id 4) TNat)
                   (Id 0) tag_star.
```

We do not consider the source of the variable bindings for `Id 3` and `Id 4`; they are assumed to be natural numbers bound by a function or `let` expression. The tag-entry for `Id 0`, that is, the type variable in the type of `union`, is created by parsing the program initially, which we also do not consider for now.

```
Example t7 : prog > Gamma |- app1 ::: 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.
```

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] ::: TFun TNat (TFun TNat TNat)
______________________________________(2/3)
prog > cntxt |- tvar (Id 3) ::: TNat
______________________________________(3/3)
prog > cntxt |- tvar (Id 4) ::: 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 as well as `Forall` and can be proven by applying rules of a 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 that.

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

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

The tactic `constructor` tests every constructor of an inductive type, which works for inductively defined propositions. Considering the above proof, we used `reflexivity` to evaluate `specialize_func` and to prove the equation. At first glance, one may question why `constructor` is able to use `reflexivity` if it works by trying constructors. The answer is found in the Coq reference manualThe Coq Development Team [2016]: `reflexivity` is equivalent to `apply refl_equal`.

```
> Print refl_equal.
Notation refl_equal := @eq_refl
> Print eq_refl.
Inductive eq (A : Type) (x : A) : A -> Prop :=  eq_refl : x = x
```

Coq tells us that `eq_refl` is a constructor of the inductive definition `eq`, which is a notation for the equality sign. Thus, `reflexivity` is actually the application of a constructor and can be tried by `constructor`.

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 ::: 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.

4. FlatCurry

FlatCurry[1] is an intermediate representation of Curry code that is used by compilers like KiCS2 and enables meta-programming, that is, the transformation of Curry programs in Curry. Since we want to reason about Curry programs in Coq, the respective module `FlatCurry.Types` will be the foundation of the Coq implementation. Generating 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.

```
addTwo n = addOne (addOne n)
  where addOne m = m + 1
```

```
addTwo n = addOne (addOne n)
addTwo.addOne m = m + 1
```

Secondly, pattern matching is substituted by `case` and `or` expressions, the latter in case of overlapping patterns. We already discussed this transformation in section 2.2.

In this chapter we pursue the same approach used with CuMin, that is, implementing FlatCurry's syntax, a context and typing rules. Some aspects of the FlatCurry implementation are similar to CuMin and therefore will not be described as detailed as before, instead we discuss additional possibilities like automatically creating and parsing FlatCurry programs.

4.1. Syntax

The syntax we use in Coq is similar to the Curry code[2] 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. Thus, these limitations need to be considered when transforming the syntax.
There are three type synonyms that identify variables and other named entities, 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.

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

- `QName`: A qualified name is a pair of strings: the module name and the name of 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.

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

Types in FlatCurry are more abstract compared to CuMin, but there are similar elements: Type variables and function types are similar to CuMin, except the index of variable is a `TVarIndex` instead of an ID. 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` as well as a type variable `TVar x`. Base types like `Int` do not have type parameters, as shown in the example below.

```
(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 CuMin does not have user-defined data types. There are two constructors: type synonyms, for example `type IntL = [Int]`, and a constructor to create new types. While both have a list of type variables, `TypeSyn` has a type expression and the `Typec` a list of constructor declarations. The following example shows the FlatCurry representation of binary trees in Curry, that is, a data type `BTree` with two constructors: a `Leaf` without further arguments and a `Branch` with a *decoration* of some type as well as two binary trees.

```
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 with the index `0`, which is used in place of the decoration's type and supplied to the subtrees in order to allow polymorphism.

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 than in CuMin, 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 argu-

ments supplied (`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 as well as a list of expressions, that the function 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 full function call of map since all arguments have been supplied. 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 of a let expression is the term that the bindings occur in. The following example shows the expression `let` $x = y + 1,\ y = z + 2,\ z = 3$ in $x + y$ in FlatCurry syntax.

```
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 replaces unnecessary variables like z with its value. Bindings are sorted hierarchically, that is, if a variable occurs in another binding, its position in the list is ahead of the binding that references the variable.

Case There are two instances that result in a case expression in FlatCurry: explicit case expressions 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 while rigid cases are evaluated by residuation, as we discussed in section 2.2. As the evaluation strategy does not affect the result's type, we do not need to distinguish both cases.

The two remaining arguments are the 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 are simple: `Free` has a list of variables indexes that represent free variables, which can be used in the supplied expression in place of variables or values. `Or` represents the non-deterministic choice between two expressions, similar to the `?` operator. Lastly, `Typed` explicitly assigns a type to an expression, for example `2 :: Int`.

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 variable, function or constructor context 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 a declaration contains the function's type explicitly and we only need to extract the type variables.

```
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.
```

`extractTVars` works by recursively concatenating the indexes of a type expression. Because the function lists every occurrence of a type variable, the result needs to be deduplicated. Together with the function type, the list of type variables is added to the context by the function `addFunc`. To add a list of function declarations to the same context, `parseFuncs` uses `fold_right`.

```
Definition parseFuncs (con : context) (fdecls : list FuncDecl) : context :=
  fold_right (fun f c => addFunc c f) con fdecls.
```

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 to incorporate this information in the constructor's type. We begin by parsing the list of constructor declarations and pass the data type as an additional parameter to the processing functions, 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 by adding 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, even though this situation cannot occur normally because the type of a constructor contains a least the data type, for example `Nothing :: Maybe a`. Now we can use `tyListFunc` to assemble the full type of a constructor.

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

The information obtained from the type declaration is the 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 $a \to$ 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

The FlatCurry typing rules expand concepts we discussed in the previous chapter. Functions can be applied to multiple arguments at the same time, 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 are represented by a type constructor call `TCons ("Prelude", "Int") []` for the

respective type. In the following rules the same syntax $\Gamma \vdash e :: \tau$ is used as before to express that in a context Γ the expression e has the type τ. We will discuss the different inference rules and, if not easily transferable, the FlatCurry implementation of the rule.

Literals and Variables FlatCurry does have fewer literals than CuMin, all of which are contained in the `Lit` expression. Therefore, the type of a literal depends on the constructor of its argument.

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

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

Figure 4.1.: Typing rules for literals and variables

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.

Function Application Applying a function f to n arguments $e_1 \dots e_n$ with types $\tau_1 \dots \tau_n$ requires the types to match the argument types in the function type returned by the context. Looking up the function f, denoted by curly brackets, yields the pair $(\tau_1 \to \dots \to \tau_n \to \tau, \overline{x_k})$ where the first component is the type of f and the second is a list of type variables, as defined in section 4.2.

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

Figure 4.2.: Typing rule for function applications

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 `app = map IntToChar [1,2,3]` with an arbitrary function `IntToChar :: (Int → Char)` would result in the typing rule below.

$$\frac{\Gamma \vdash \texttt{IntToChar} :: (a \to b)[a \mapsto t_1, b \mapsto t_2] \qquad \Gamma \vdash [1,2,3] :: [a][a \mapsto t_1, b \mapsto t_2]}{\Gamma, \{\texttt{map} \mapsto ((a \to b) \to [a] \to [b], [a, b])\} \vdash \texttt{app} :: [b][a \mapsto t_1, b \mapsto t_2]}$$

Figure 4.3.: Typing rule for `map IntToChar [1,2,3]`

In this example the $\overline{x_i}$ are instantiated with a and b by looking up `map` in the context. The type information obtained from the context is divided into the types of the respective arguments. Since these types can still contain type variables, the substitution types t_1 and t_2 need to be instantiated appropriately, in this case with `Int` and `Char`, in order satisfy the premises of the rule.

Similar to CuMin, we use the inductively defined proposition `hasType` to describe typing rules. The respective types are adapted to reflect the FlatCurry syntax, but apart from this change, the proposition works the same. The implementation of function applications is shown below.

```
Inductive hasType : context -> Expr -> TypeExpr -> Prop :=
T_Comb : forall Gamma qname exprs cType substTypes T,
           let funcT := fromOption failType ((combCon cType 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 cType qname exprs) ::: T
```

The rule `T_Comb` can type full function and constructor applications since the only difference is the context that the function or constructor is looked up in, which is decided by `combCon` depending on the supplied `combType`. The result is `funcT`, a pair of the function's type and type variables.

Next, the variable `substTypes` (the specialization types supplied by the user) is used to replace the type variables in the function type with `multiTypeSubst`, which yields the specialized type `specT` the same way the function works in the CuMin implementation.

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. but 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.

Partial Function Application 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 divided into 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.

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

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

Figure 4.4.: Typing rule for partial function applications

The 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_Part: forall Gamma qname exprs substTypes cType T,
               let funcT := fromOption failType ((combCon cType Gamma) qname) in
               let specT := multiTypeSubst (snd funcT) substTypes (fst funcT) in
               let     k := funcArgCnt (fst funcT) - (partCombArgs cType) in
               let argTs := firstn (@length Expr exprs) (fst (funcTyList specT))
                in funcPart specT (Some n) = T ->
                   Forall2 (hasType Gamma) exprs argTs ->
               Gamma |- (Comb cType qname exprs) ::: 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 of the function, 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, of which we subtract `partCombArgs cType`, that is, the number of remaining arguments contained in the `combType`. 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 k`. 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.

Let Expression 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 $x_1 \dots x_n$ are required in order to type the expressions $e_1 \dots e_n$ and e.

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

Figure 4.5.: Typing rule for let expressions

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 vTys   := replaceSnd vExprs tyexprs in
          let Delta  := multiTypeUpdate Gamma vTys
           in Forall2 (hasType Delta) exprs tyExprs ->
              Delta |- e ::: T ->
          Gamma |- (Let (ve :: ves) e) ::: 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.

The variable `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 the types of the expressions used in the bindings with `Forall2` and use the updated context to type e.

Case Expression 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.

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

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

Figure 4.6.: Typing rule for case expressions

Because every branch expression needs a context containing the variables of its constructor, there are two options: either every expression is typed in its own context that contains only the relevant variable entries, or all variables are added to the same context that then can type every expression. The latter option is used here because, even if variables are named the same in a Curry program, FlatCurry uses unique variable names and therefore there is no risk of a name clash. Unfortunately, the result of this approach is the rather obscure formal representation with twice overlined expressions.

TODO: CamelCase im Code

```
T_Case : forall Gamma ctype e substTypes T Tc p vis brExprs',
           let  brExprs := Branch p vis :: brExprs' in
           let   patPs  := map pattSplit (brExprsToPatterns brExprs) in
           let consTvis := map (compose (fromOption defaultTyVars) (cCon Gamma))
                               (map fst patPs) in
           let     tvis := snd (fromOption defaultTyVars
                                           (cCon Gamma (fst (pattSplit p)))) in
           let   specTs := map (multiTypeSubst tvis substTypes)
                               (map fst consTyVis) in
           let  visTys  := zip (map snd patPs)
                               (map (compose fst funcTyList) specTs) in
           let    Delta := multiListTypeUpdate Gamma visTys
            in Forall (flip (hasType Delta) T) (brExprsToExprs brExprs) ->
               Forall (fun ty => ty = Tc) (map ((flip funcPart) None) specTs) ->
               Gamma |- e ::: Tc ->
         Gamma |- (Case ctype e (Branch p vis :: brExprs')) ::: T
```

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. The implementation might not be clear at first sight. Therefore, the example below, that is, a function that retrieves the value of a `Maybe` or returns a default value, is used to illustrate the computation.

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

```
example = fromMaybeCase 0 (Just 2)
```

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 `patPs` 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 `consTvis` 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]]
```

The second component we need in the premises later is `visTys`, a list of pairs that assigns all variables used in constructors their type, according to the specialized types.

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

The pairs represents `Just` with its variable x and `Nothing` without variables. Lastly, `visTys` is 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.

Now, the computational part of the rule is finished and we can begin to check the premises. The first `Forall` proposition checks if all branch expressions have the same type `T`. In this case we would need to check if 0 and 2 are integers, which is obviously true. The next `Forall` proposition checks if the specialized data type `Tc` of all constructors matches the case expression's type, that is, `Maybe Int`, which is also true. Therefore, the case expression has the type `Tc`.

Or, Free and Typed 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.

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

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

Typed expressions require that the (polymorphic) type T of the expression e can be specialized to the stated type τ, 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 T, 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 $T \Rightarrow \tau$ 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. Transformation of FlatCurry into 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 double quote character needs to be escaped because characters are enclosed in " 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".

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.

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

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, a string can contain unescaped double quotes. Furthermore, we use the function `escQuote` to escape double quotes by adding another double quote to it.

```
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 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, that is, the term that is contained in the `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`, the second `double` itself, 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 function type with the supplied `substTypes` 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 ::: (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 proven by applying `reflexivity`. However, there is 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 `[Int] → [Int]`, 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 because there are no arguments. 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 ::: 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 subgoal

`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 the example below.

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

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)
> replChar 'a' b "a -> a" == "B -> B" where b free
{b = 'B'} True
```

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. 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, `FuncType ?a Int = FuncType Char Int` would result in the instantiation of `?a` with `Char`.

```
> replChars ['a', 'b'] cs "a -> b" == "I -> C" where cs free
{cs = ('I':'C':_xk)} True
```

The actual cause of the problem is the definition of `multiTypeSubst` or rather that it replaces multiple variables. The above function `replChars` is similar to the Coq implementation of `multiTypeSubst` and while Curry is able to find a solution of the equation, Coq cannot instantiate the list `?substTypes`.

```
match ?substTypes with
| [] => FuncType (TVar 0) (TVar 0)
| t :: _ => FuncType t t
end = FuncType Bool Bool
```

One possible problem of the Coq implementation is the usage of `zip` in `multiTypeSubst` because lists of different length are handled by discarding the longer list's tail. As we can see in Curry's solution, using `zip` entails that lists of arbitrary length satisfy the equation. However, this behavior is actually quite useful because it allows us to specify types only for supplied arguments in partial a application.
Implementations that consider the lists' length and fail if it differs do not solve this problem either; they are stuck at the same step as shown above, even though the solution is unique. One probable cause of this is that Coq would need to try every branch of the function and introduce further variables (`t` in this example), which this tactic is not capable of.

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 ::: 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 free variable has the 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`, the latter being the type of the free variable. Although we know that `?substTypes` needs to be explicitly supplied, it is preferable to use `eapply` nevertheless because `?Tc` can be inferred.

In order to further proceed in the proof, we need to supply a value for `?substTypes`. The tactic `instantiate` instantiates existential variables but works with numbers, instead of variable names, to simplify automation. The variables are enumerated[3] by their first occurrence from right to left in the expression. We instantiate `?substTypes` with the type `Int` to specialize `Maybe a`.

We already used `;` before, but in this case we use it between *all* tactics. However, there are downsides to this approach: `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. Because we accumulated all current subgoals with `;` and instantiated the existential variables, a single application of `repeat econstructor` concludes the last proof of the FlatCurry chapter.

[3] `Show Existentials` can be used in a proof to list existential variables and the corresponding numbers.

5. Conclusion

To conclude this thesis, we begin with recapitulating previous chapters and compare the initial goals with the outcome. Then, we discuss related work as well as future expansions of the implementation.

5.1. Summary and Results

The first goal of this thesis was to find an appropriate representation of Curry code in Coq and to implement the type system. After familiarizing us with Coq, Curry and the theoretical basics, we began formalizing CuMin, a minimal representation of Curry. Because of its simplicity, CuMin was a good way to gain a deeper understanding of the structure of programming languages and type systems. We transferred CuMin's syntax from a BNF representation to Coq, created a context that stores information about variables and used inductively defined propositions to formalize the inference rules of CuMin's type system. Lastly, we proved some examples and discussed advanced Coq tactics that allowed us to automate proofs in many cases.

Due to CuMin's theoretical nature, the implementation became a precursor to the formalization of the more practically oriented language FlatCurry. We used the same approach as with CuMin, beginning with the transfer of the syntax from the Curry implementation to Coq. We added information about function types and type variables to the context and created a parser that extracts this information from a FlatCurry program.

The implementation of FlatCurry's typing rules reused some functions and structures from CuMin, but the higher complexity of the rules affected the clarity of the implementation in some places, compared to CuMin. This is particularly apparent in rules with many `let` expressions, which result in large, flat terms when using the rule in a proof. In contrast, quantified variables would require the user to often explicitly specify redundant information and would add more steps proofs. These problems might be mitigated by solely using tactics that introduce existential variables, but this compromises the understandability of proofs, especially on paper. Thus, the current implementation favors the understandability of proofs over the readability of the subgoal list.

In the last sections of the FlatCurry chapter we adapted the `FlatCurry.Show` module to incorporate some Coq-specific changes. Based on this, we created a program that transforms a Curry file into a FlatCurry definition that can be saved and imported as a Coq file. To conclude the chapter, we transformed a Curry program and proved some examples, supplemented by a short digression about Coq's inference capabilities.

5.2. Related and Future Work

The second goal of this thesis was to provide a basis for the formalization of further aspects of Curry. While the type system is an important part of a programming language, there are few interesting properties to be proven about specific expressions or functions because the type always remains the same and is checked by the compiler. However, there are other applications like type inference, as shown below.

```
Fixpoint inferType (c : context) (e : Expr) : option TypeExpr :=
  match e with
  | Var i => (vCon c) i
  | Lit l => Some (litType l)
  | ...
  end.

Theorem typeInferenceWorks : forall Gamma e, Gamma |- e ::: inferType Gamma (Some e).
```

The function `inferType` returns the type of an expression in a context and would work like `hasType`. One could proof that such a function yields the same types as the inductively defined proposition and incorporate it into `hasType` to remove types that need to be explicitly supplied currently.

Another interesting aspect to formalize is the semantics, that is, rules that determine the evaluation, of Curry. A simple demonstration of the formalization of a toy language has been created by Pierce et al. [2016]. It involves representing the syntax in Coq and formalizing the semantics in form of inference rules. Together with a formalized type system, one can argue about the *type soundness* of terms. This means that a well-typed term is either a value or can be further evaluated while preserving its type.

An approach for imperative languages has been implemented by Blazy et al. [2006]. The paper presents the formal verification of a compiler front-end[1] that translates *Clight*, a subset of the imperative programing language C, into the *Cminor* intermediate language. Clight's syntax is represented in Coq by inductive data types and the semantics and typing rules are transformed into inductively defined propositions to argue about the languages and the translation.

In conclusion, there are many more aspects of Curry that could be formalized to expand the functionality of this implementation.

[1] https://github.com/AbsInt/CompCert/tree/master/cfrontend

A. Usage

The source code presented in this thesis can be found at `https://git.informatik.uni-kiel.de/stu114713/curroqe`. The repository is structured as follows.

- **code** contains Coq and Curry source files.
  - **typing** contains files regarding CuMin and FlatCurry typing.
    - **Basics.v**: General function definitions like `zip` or `elem`.
    - **CuMin.v**: CuMin syntax definition and typing rules.
    - **FlatCurry.v**: FlatCurry syntax definition.
    - **FlatCurryType.v**: FlatCurry typing rules.
    - **Maps.v**: Polymorphic maps.
  - **tools** contains transformation programs.
    - **showFlatCoq.curry**: Transforms Curry files to FlatCurry definitions in Coq syntax.
    - **FlatCurry** contains Curry modules.
      - **ShowCoq.curry**: Adapted `Show.curry` module.
  - **_CoqProject**: Used to generate a `Makefile`.
- **text** contains LaTeX files.

Compilation In order to compile all Coq files of this project, the `_CoqProject` file can be used.

```
coq_makefile -f _CoqProject -o Makefile
```

This generates a `Makefile` that can be executed with `make`. The generated `Makefile` resolves dependencies and calls the Coq compiler `coqc`. Subsequently, the files can be opened and imported.
The transformation program can be compiled by using KiCS2 in the respective directory:

```
kics2 :load ShowCoq.curry :quit
kics2 :load showFlatCoq.curry :save :quit
```

The first command compiles the `ShowCoq` module, the second generates a binary that can be used with a Curry source file as follows.

- `./showFlatCoq <name>.curry` prints the transformed FlatCurry program to the terminal. The output can be written to a file by adding `> <name>.v` to the command.
- `./showFlatCoq -c <name>.curry` prints a `_CoqProject` file that can be used to compile the transformed FlatCurry definition file. Similarly, the output can be written to a file by adding `> _CoqProject` to the command and then processed by `coq_makefile` as shown above.

Transformation The complete process of transforming a Curry file begins with compiling the program and using `showFlatCoq` to generate the definition and `_CoqProject` files.

```
kics2 :load MyProg.curry :quit
./showFlatCoq MyProg.curry > MyProg.v
./showFlatCoq -c MyProg.curry > _CoqProject
```

If one wants to work in a separate Coq file `MyProgProofs.v`, the file name needs to be appended to the `_CoqProject` file with `echo "MyProgProofs.v" >> _CoqProject`. Additionally, the FlatCurry definition and the typing rules needs to be imported with `Require Export CQE.MyProg.` in the `MyProgProofs.v` file. Then, the `Makefile` can be created and the Coq files are compiled.

```
coq_makefile -f _CoqProject -o Makefile
make
```

The new Coq file can now be opened in a Coq front-end. The import of `CQE.MyProg` loads the definition of the Curry program, which is named after the Curry file with `_coq` appended, in this example `MyProg_coq`.

```
> Check MyProg_coq.
MyProg_coq : FlatCurry.TProg
```

Finally, the definition can be used in the new Coq file, for example with `parseProgram`.

Bibliography

Sandrine Blazy, Zaynah Dargaye, and Xavier Leroy. Formal verification of a C compiler front-end. In *FM 2006: Int. Symp. on Formal Methods*, volume 4085 of *Lecture Notes in Computer Science*, pages 460–475. Springer, 2006. URL `http://gallium.inria.fr/~xleroy/publi/cfront.pdf`.

Jean-Christophe Filliâtre. Deductive software verification. *International Journal on Software Tools for Technology Transfer*, 13(5):397–403, 2011.

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], Sergio Antoy, Bern Braßel, Herbert Kuchen, Francisco J. López-Fraguas, Wolfgang Lux, Juan José Moreno Navarro, Björn Peemöller, and Frank Steiner. Curry: An Integrated Functional Logic Language, January 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/`.