

Christian-Albrechts-Universität zu Kiel

Bachelor Thesis

TODO:
Weitere Formalitäten...

Title: TBD

Niels Bunkenburg

SS 2016

Contents

1	Preliminaries	1
1.1	Coq	1
1.1.1	Data types and functions	1
1.1.2	Propositions and proofs	2
1.1.3	Higher-order constructs	5
1.1.4	Inductively defined propositions	6
1.2	FlatCurry	8
2	Theory	9
3	CuMin	11
3.1	Syntax	11
3.2	Context	13
3.3	Data types	13
3.4	Typing	14
4	Curry	18
4.1	FlatCurry	18
4.2	Typing	18
4.2.1	Differences to CuMin	18
4.2.2	Conversion of FlatCurry to Coq	18
5	Conclusion	19

1 Preliminaries

1.1 Coq

The formalization of Curry programs requires a language that allows us to express the code itself and the propositions we intend to prove. Coq¹ is an interactive proof management system that meets these requirements, hence it will be the main tool used in the following chapters.

TODO:
Kommasetzung?

1.1.1 Data types and functions

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

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

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

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

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

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

The first definition does not work, as indicated by `Fail`², because Coq cannot infer the implicit type variable of `double_cons`, since `cons` does not have an explicit type either.

¹<https://coq.inria.fr/>

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

By prefixing at least one `cons` with `@`, we can tell Coq to accept explicit expressions for all implicit arguments. This allows us to pass the type of `cons` on to `double_cons`, again as an implicit argument.

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

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

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

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

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

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

In this case l_1 gets shorter with every iteration, thus the function terminates after a finite amount 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.

```
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).
```

1.1.2 Propositions and proofs

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

1 Preliminaries

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

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

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

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

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

TODO:
Highlighting
für Proof
und Qed

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

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

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

```
Example e3 : forall (X : Type) (l : list X), l ++ [] = l.
Proof. intros X. induction l as [| l1 ls IH].
reflexivity.
simpl. rewrite IH. reflexivity.
Qed.
```

The proof begins by introducing type `X`, followed by the `induction` tactic applied to `l`. Coq names newly introduced variables by itself, which can be done manually by adding

`as [c1|...|cn]` to the tactic. Each c_i represents a sequence of variable names, which will be used when introducing variables in the corresponding case. Cases are ordered as listed in the [Definition](#).

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

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

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

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

Example e4 : `(fun n => n <> 2) 1.`

Proof.

```
simpl.      (* 1 <> 2 *)
unfold not. (* 1 = 2 -> False *)
intros H.   (* H : 1 = 2, False *)
inversion H. (* No more subgoals. *)
```

Qed.

This proof is not as straight forward as the other ones, mainly because of the inequality, which is a notation³ for `not (x = y)`. Because `not` is the outermost term, we need to eliminate it first by applying `unfold`. This replaces `not` with its definition `fun A : Prop => A -> False`, where `False` is the unprovable proposition. Why does this work? Assuming that a proposition `P` is true, `not P` means that `P` implies `False`, which is false, because something true cannot imply something false. On the other hand, if `P` is false, then `False -> False` is true because anything follows from falsehood, as stated by the principle of explosion.

The current goal `4 = 8 -> False` is further simplified by introducing `4 = 8` as an hypothesis `H`, leaving `False` as the remaining goal. Intuitively we know that `H` is false, but Coq needs a justification for this claim. Conveniently the tactic `inversion` solves this problem easily by applying two core principles of inductively defined data types:

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

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

TODO:
verweis?

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 $S(S(0)) = S(0)$. Injectivity implies that if the previous equation was true, $S(0) = 0$ must also be true. This is obviously false, since it would allow two different representations of nil. Hence, the application of `inversion` to $2 = 1$ infers the current goal `False`, which concludes the proof.

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

1.1.3 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, such as `map` or `fold`.

TODO:
minted bug?

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

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

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

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

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

`Forall` is an inductively defined proposition, which requires rules to be applied in order to prove a certain goal. This will be further explained in the next section, for now it is 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 $2 \neq 8$, 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.

1.1.4 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 `b x = true` and functions that return propositions. For example the function `InB` returns `true` if a `nat` is contained in a list, the boolean function could look like this:

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

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

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

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

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

\wedge , with the difference that `split` keeps both sides as subgoals, since a conjunction is only true if both sides are true.

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

We can describe `In` with two rules:

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

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

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

```
Example e7 : InInd 42 [2;42].
```

```
Proof.
```

```
  apply Tail. (* InInd 42 (tl [2; 42]) *)
  simpl.      (* InInd 42 [42] *)
  apply Head.
```

```
Qed.
```

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

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

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

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

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

1.2 FlatCurry

2 Theory

In functional languages a data type is a classification of applicable operators and properties of its members. There are base types that store a single data 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 which contains data type definitions, function/constructor/operator declarations and a map that assigns types to variables. Without a context, expressions do not have any useful meaning – 42 could be typed as a number, the character 'B', a string or the answer to everything. The majority of information in a context can be extracted from the source code of a program and is continually updated while typing expressions.

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

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

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

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

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

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

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

2 Theory

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

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

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

3 CuMin

3.1 Syntax

CuMin is a simplified sublanguage of Curry, which restricts the syntax to allow more concrete typing rules and data types. Although it requires some transformations to substitute missing constructs, CuMin can express the majority of Curry programs. Figure 3.1 shows the syntax of CuMin in Backus–Naur Form:

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

Figure 3.1: Syntax of CuMin

A program P is a list of function declarations D , which contain a function name f , a list of quantifiers κ , a type τ and a function definition. Quantifiers have a tag $t \in \{\epsilon, *\}$ that determines the valid types of the variable α . 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, \dots, 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, primitive `Bool` or `Nat` types, lists, pairs and functions. An example for a CuMin function is `fst`, which returns the first element of a pair:

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

It is important to notice that polymorphic functions need to be explicitly specialized

before they are applied to another expression, as shown in `one`, because there is no type inference.

Besides function application, there are literal boolean values and natural numbers, variables, arithmetic expressions, let bindings and case constructs and constructors for pairs and lists. The two remaining expressions arise from Curry's logical parts: `anythingτ` represents every possible value of type τ , similar to free variables. `failureτ` represents a failed computation, for example `fail = anythingNat \doteq True`. Since `anythingNat` can be evaluated to natural numbers only, the equation always fails because `Nat` and `Bool` are not comparable.

The Coq implementation follows the theoretical description closely. Variables, quantifiers, functions and programs are identified by an `id` instead of a name, to simplify comparing values. Case expressions for lists and pairs have two `id` arguments that represent the variables x and y , the head/tail or left/right side of the term 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
| TFunc : 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
| tcase1 : tm -> id -> id -> tm -> tm -> tm.

```

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

`FDecl (Id 0)`

TODO: label?

```
[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]
(tcseep (tvar (Id 3)) (Id 4) (Id 5) (tvar (Id 4))).
```

3.2 Context

As mentioned in chapter 2, we need a context in order to be able to type expressions. This basic version contains no program information and stores two partial maps: One maps type variable IDs to tags, the other variable IDs to types.

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

There are two selector functions `tagcon` and `typecon` to access the corresponding maps of a context and two update functions `tag_update` and `type_update` to update values.

TODO:
Mehr
schreiben?

3.3 Data types

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

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

The inductively defined proposition `is_data_type` takes a context plus a type and returns a proposition, which can be proven using the provided rules if the type is indeed a data type. Coq allows notations to be announced 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 $\Gamma \vdash \tau \backslash \text{is_data_type}$, 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 assignment of a type to an expression. The rules **D_Bool** and **D_Nat** simply state that basic types are data types. **D_Var** requires type variables to have a star tag in order to be a data type, because as mentioned above, nested function types are not allowed. Lists and pairs are data types if their argument type(s) are data types.

3.4 Typing

Typing requires a set of rules that covers every valid expression and assigns corresponding types. The following inference rules [1] are composed of typing relations $\Gamma \vdash e :: \tau$ that state the type τ of an expression e in a context Γ . The notation $\Gamma, e_1 :: \tau_1 \vdash e_2 :: \tau_2$ means that e_2 can only be typed to τ_2 if Γ contains the information $e_1 :: \tau_1$. As mentioned in subsection 1.1.4, the premises of an inference rule above the line need to be fulfilled in order for the conclusion below the line to hold.

$$\begin{array}{c}
\Gamma, x :: \tau \vdash x :: \tau \quad \Gamma \vdash \text{True} :: \text{Bool} \quad \Gamma \vdash \text{False} :: \text{Bool} \quad \Gamma \vdash n :: \text{Nat} \quad \Gamma \vdash \text{Nil}_\tau :: [\tau] \\
\\
\frac{\Gamma \vdash e_1 :: \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash e_2 :: \tau_1}{\Gamma \vdash e_1 e_2 :: \tau_2} \quad \frac{\Gamma \vdash e_1 :: \tau_1 \quad \Gamma, x :: \tau_1 \vdash e_2 :: \tau}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 :: \tau} \quad \frac{(f :: \forall^{v_1} \alpha_1 \dots \forall^{v_m} \alpha_m. \tau; f \overline{x_n} = e) \in P \quad \text{if for all } i \text{ with } v_i = * \text{ we have } \Gamma \vdash \tau_i \in \text{Data}}{\Gamma \vdash f_{\overline{\tau_m}} :: \tau[\overline{\tau_m}/\overline{\alpha_m}]} \\
\\
\frac{\Gamma \vdash e_1 :: \text{Nat} \quad \Gamma \vdash e_2 :: \text{Nat}}{\Gamma \vdash e_1 + e_2 :: \text{Nat}} \quad \frac{\Gamma \vdash e_1 :: \text{Nat} \quad \Gamma \vdash e_2 :: \text{Nat}}{\Gamma \vdash e_1 \doteq e_2 :: \text{Bool}} \quad \frac{\Gamma \vdash e_1 :: \tau_1 \quad \Gamma \vdash e_2 :: \tau_2}{\Gamma \vdash (e_1, e_2) :: (\tau_1, \tau_2)} \quad \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 :: \tau', t :: [\tau'] \vdash e_2 :: \tau}{\Gamma \vdash \text{case } e \text{ of } \langle \text{Nil} \rightarrow e_1; \text{Cons}(h, t) \rightarrow e_2 \rangle :: \tau} \quad \frac{\Gamma \vdash e :: (\tau_1, \tau_2) \quad \Gamma, l :: \tau_1, r :: \tau_2 \vdash e_1 :: \tau}{\Gamma \vdash \text{case } e \text{ of } \langle (l, r) \rightarrow 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} \rightarrow e_1; \text{False} \rightarrow e_2 \rangle :: \tau} \quad \Gamma \vdash \text{failure}_\tau :: \tau \quad \frac{\Gamma \vdash \tau \in \text{Data}}{\Gamma \vdash \text{anything}_\tau :: \tau}
\end{array}$$

Figure 3.2: Typing rules for CuMin

The first row of rules holds unconditionally, basic expressions like boolean values and natural numbers have the type **Bool** and **Nat** respectively, there is also an empty list of every type. Variables can only get 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 , which is used within the expression e_2 and needs to be added to the context in order to type e_2 . The row's last inference rule describes typing a function call with specific types $\overline{\tau_m}$. The program P needs to contain a matching function declaration with a list

of quantified type variables $\overline{\alpha_m}$. For every α_i the corresponding τ_i needs to be a data type if their quantifier has a star tag, because we must ensure that these variables are replaced by non-functional types, which data types fulfill by definition. The type of a function call is represented by the expression $\tau[\tau_m/\alpha_m]$, which is a type substitution of every occurrence of α_i in τ with τ_i .

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

Case expressions work similarly but have type specific properties. The first argument has to be of the case's type, for example **Bool** for the boolean case expressions. Depending on the form of the term, the corresponding branch expression, all of which need to have the same type, 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 to non-functional types is necessary, because functions are not enumerable.

TODO: Redundanzen entfernen

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` which represents the typing relation $\Gamma \vdash e :: \tau$ used above. Since `::` is the **cons** constructor in Coq, we will use `\in` instead. Another interesting detail is the usage of a **Variable** to represent the program. Variables are locally declared within a **Section** and can be used like any other defined identifier. When a definition containing a variable is used outside of the section, the variable will be appended to the type, for example `has_type` has the type `program -> context -> tm -> ty -> Prop`, although the below definition is missing the `program`.

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

Supplying arguments to a function is limited to one at a time, hence we apply an ex-

pression $e_2 :: \tau_1$ to a functional expression $e_1 :: \tau_1 \rightarrow \tau_2$. Because we supplied e_2 with its first argument, the resulting type is the return type of the function, which may be of functional type itself, since a recursive definition is possible.

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

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

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

There are case expressions for booleans, 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` is no usual case expression, because there is only one case, e_1 . This construct 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 return. 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  $\lambda$ in TBool -> Gamma |- e1  $\lambda$ in T -> Gamma |- e2  $\lambda$ in T ->
  Gamma |- (tcaseb e e1 e2)  $\lambda$ in T
T_CaseL : forall Gamma e e1 e2 h t T T',
  Gamma |- e  $\lambda$ in (TList T') -> Gamma |- e1  $\lambda$ in T ->
  (type_update (type_update Gamma h T') t (TList T')) |- e2  $\lambda$ in T ->
  Gamma |- (tcaseL e h t e1 e2)  $\lambda$ in T
```

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

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

The lookup function uses the predefined `find`, which takes a boolean predicate plus a list and returns all elements that fulfill the condition. An anonymous function is

used to compare the function's ID to every entry's ID until a match is found. Because the search is not guaranteed to succeed, an `option func_decl` is returned. At this point there are two options to handle variables: 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 definition 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, $\Gamma \vdash e \text{ in } T$, it needs to be instantiated manually when applying the rule in a proof.¹ This is especially cumbersome for function declarations, since this information is already contained in the program. Because of this limitation, we will use `let` instead, which is not as comfortable, but makes proofs significantly shorter.

TODO:
Verlinkung
in Fußnote

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

¹This is actually not completely true, there are automated tactics that are able to infer this information. See...

4 Curry

4.1 FlatCurry

4.2 Typing

4.2.1 Differences to CuMin

4.2.2 Conversion of FlatCurry to Coq

5 Conclusion

Bibliography

- [1] S. Mehner, D. Seidel, L. Straßburger, and J. Voigtländer. Parametricity and proving free theorems for functional-logic languages. Dec. 18 2014.