Compiling Alba

Helmut Brandl

(firstname dot lastname at gmx dot net)

Abstract

How to compile Alba programs.

Contents

1 Calculus

1.1 Terms and Contexts

The terms are generated by the grammar

$$
\begin{array}{rcll}
t & ::= & \mathcal{P} & \text{propositional sort} \\
 & | & \mathcal{U} & \text{computational sort} \\
 & | & c & \text{constants (numbers, strings, \ldots)} \\
 & | & x & \text{variables} \\
 & | & \overset{?}{m} & \text{metavariables} \\
 & | & \mathcal{M} & \text{constructors} \\
 & | & f\mathbf{a} & \text{application} \\
 & | & \lambda \mathbf{x}^{\mathbf{A}}.e & \text{abstractions} \\
 & | & \Pi \mathbf{x}^{\mathbf{A}}.B & \text{products} \\
 & | & \text{let}[\mathbf{x} := \mathbf{a} \mid e] & \text{let expressions} \\
 & | & \text{type}[\Gamma \mid I^K \mid \mathbf{C}] & \text{inductive types} \\
 & | & \text{case}[f^F \mid \mathbf{c} \mid t] & \text{case expressions} \\
 & | & \text{cta}[f, t, \mathbf{u}, p] & \text{case tree application}
\end{array}
$$

and contexts are generated by the grammar

$$
\begin{array}{rcll}
\Gamma & ::= & [] & \text{empty context} \\
 & | & \Gamma, x^A & \text{local variable} \\
 & | & \Gamma, (x^A := a) & \text{definition} \\
 & | & \Gamma, \overset{?}{m}^M & \text{meta variable} \\
 & | & \Gamma, (\overset{?}{m}^M := a) & \text{meta variable instantiation}
\end{array}
$$

1.2 Head Reductions

The following head reductions are possible:

$$
\begin{array}{lcl}
\Gamma \vdash (\lambda \mathbf{x}^{\mathbf{A}}.e)\mathbf{ab} & \overset{h}{\rightsquigarrow} & \Gamma, [\mathbf{x}^{\mathbf{A}} := \mathbf{a}] \vdash e\mathbf{b} \\
\Gamma \vdash \text{let}[\mathbf{x}^{\mathbf{A}} := \mathbf{a}\ , e]\mathbf{b} & \overset{h}{\rightsquigarrow} & \Gamma, [\mathbf{x}^{\mathbf{A}} := \mathbf{a}] \vdash e\mathbf{b} \\
\Gamma \vdash \text{case}[*, *, t]\mathbf{ab} & \overset{h}{\rightsquigarrow} & \Gamma \vdash \text{cta}[\text{case}[*, *, t], t, \varepsilon, p.f = a_1]\mathbf{b} \\
\Gamma \vdash \text{cta}[*, t_1, \mathbf{u}_1, p_1]\mathbf{b} & \overset{h}{\rightsquigarrow} & \Gamma \vdash \text{cta}[*, t_2, \mathbf{u}_2, p_2]\mathbf{b} \\
\Gamma \vdash \text{cta}(\text{pm}, e, \mathbf{u}, \varepsilon)\mathbf{b} & \overset{h}{\rightsquigarrow} & \Gamma, [f, \mathbf{x} := \text{pm}, \mathbf{u}] \vdash e\mathbf{b} \\
 & & \\
\Gamma \vdash x\ \mathbf{b} & \overset{h}{\rightsquigarrow} & \Gamma \vdash a\mathbf{b}
\end{array}
$$

The last reduction is an unfolding of a definition i.e. $(x := a) \in \Gamma$ is assumed. A case tree application can become stuck if the focal term does not allow a decision.

Note that the head reductions only make substitutions of head terms. Instead of substitutions variables with definitions are introduced.

1.3 Typing Rules

We introduce two relations.

$$\Gamma \vdash t : T$$

Read: In the context Γ the term t has type T.

$$\Gamma \vdash A \leq B$$

Read: A and B are welltyped in the context Γ and A is a subtype of B.

The rules for $\Gamma \vdash t : T$:

axiom $$[] \vdash \mathcal{P} : \mathcal{U}_0 \qquad \frac{i < j}{[] \vdash \mathcal{U}_i : \mathcal{U}_j}$$

var add $$\frac{\Gamma \vdash A : s}{\Gamma, x^A \vdash x : A} \qquad \frac{\Gamma \vdash a : A}{\Gamma, [x^A := a] \vdash x : A}$$

def elim $$\frac{\Gamma, [x^A := a] \vdash t : T}{\Gamma \vdash t[a/x] : T[a/x]} \qquad \frac{\Gamma, [x^A := a] \vdash t : T}{\Gamma, x^A \vdash t : T}$$

product $$\frac{\begin{array}{l}\Gamma \vdash A : s \\ \Gamma, x^A \vdash B : \mathcal{P}\end{array}}{\Gamma \vdash \Pi x^A.B : \mathcal{P}} \qquad \frac{\begin{array}{l}\Gamma \vdash A : \mathcal{U}_i \\ \Gamma, x^A \vdash B : \mathcal{U}_j\end{array}}{\Gamma \vdash \Pi x^A.B : \mathcal{U}_{\max(i,j)}}$$

pi elim $$\frac{\Gamma, [x^A := a], \Delta \vdash \Pi x^A.B}{\Gamma, [x^A := a], \Delta \vdash B}$$

lambda $$\frac{\Gamma, x^A \vdash e : B}{\Gamma \vdash \lambda x^A.e := \Pi x^A.B}$$

app $$\frac{\Gamma, [x^A := a], \Delta \vdash f : \Pi x^A.B}{\Gamma, [x^A := a], \Delta \vdash fx : B}$$

weaken $$\frac{\begin{array}{l}\Gamma \vdash t : T \\ \Gamma \vdash A : s\end{array}}{\Gamma, x^A \vdash t : T} \qquad \frac{\begin{array}{l}\Gamma \vdash t : T \\ \Gamma \vdash a : A\end{array}}{\Gamma, [x^A := a] \vdash t : T}$$

subtype $$\frac{\begin{array}{l}\Gamma \vdash t : T \\ \Gamma \vdash T \leq U\end{array}}{\Gamma \vdash t : U}$$

The rules for $\Gamma \vdash A \leq B$:

axiom $\quad [] \ \vdash \mathcal{P} \leq \mathcal{U}_0 \qquad \dfrac{i \leq j}{[] \ \vdash \mathcal{U}_i \leq \mathcal{U}_j}$

product $\quad \dfrac{\begin{array}{l}\Gamma \ \vdash A_2 \leq A_1 \\ \Gamma, x^{A_2} \vdash B_1 \leq B_2\end{array}}{\Gamma \ \vdash \Pi x^{A_1}.B_1 \leq \Pi x^{A_2}.B_2}$

inductive MISSING

weaken $\quad \dfrac{\begin{array}{l}\Gamma \vdash T \leq U \\ \Gamma \vdash A : s\end{array}}{\Gamma, x^A \vdash T \leq U} \qquad \dfrac{\begin{array}{l}\Gamma \vdash T \leq U \\ \Gamma \vdash a : A\end{array}}{\Gamma, [x^A := a] \vdash T \leq U}$

unfold $\quad \dfrac{\begin{array}{l}\Gamma \vdash a : A \\ \Gamma, [x^A := a], \Delta \vdash T \leq U\end{array}}{\Gamma, \Delta[a/x] \vdash T[a/x] \leq U[a/x]}$

same $\quad \dfrac{\Gamma \vdash A : s}{\Gamma \vdash A \leq A}$

equiv $\quad \dfrac{\begin{array}{l}\Gamma \vdash A \sim B \\ \Gamma \vdash B \leq C \\ \Gamma \vdash C \sim D\end{array}}{\Gamma \vdash A \leq D}$

2 Inductive Types

2.1 Form of an inductive type:

$$\text{type}[\Gamma \mid T^K \mid C_{\ell_1}, \ldots, C_{\ell_n}] : \Pi\Gamma.K$$

Parameters Γ Context of parameters. It binds all free variables in K and C_{ℓ_i}.

Kind K It reduces to $\Pi\mathbf{x}^{\mathbf{A}}.s$. It is a function type. It has an arity which is possibly zero and a sort s as its result type.

Constructor types C_{ℓ_i} Each constructor type has a label. All labels must be different. The constructor types have the form

$$\Pi\Delta.T\mathbf{a}$$

where the context Δ might be empty and T can occur in Δ only positively i.e. Δ has the structure

$$[y_1^{\Pi B_1.R_1}, \ldots, y_n^{\Pi B_m.R_m}]$$

and T can occur only in R_i and if it occurs in R_i, then R_i has the form $T\mathbf{a}$. I.e. the arguments of constructors are functions (arity zero included) where T occurs only in the result type and not in the argument types.

Recursive An inductive type is recursively defined if T occurs in the result type of any constructor argument.

2.2 Examples

boolean	$\text{type}[B^{\mathcal{U}} \mid B, B]$
peano numbers	$\text{type}[N^{\mathcal{U}} \mid N, N \to N]$
list	$\text{type}[A^{\mathcal{U}} \mid L^{\mathcal{U}} \mid L, A \to L \to L]$
equality	$\text{type}[A^{\mathcal{U}} \mid E^{A \to A \to \mathcal{P}} \mid \Pi x^A.Exx]$
accessibles	$\text{type}[A^{\mathcal{U}}, R^{A \to A \to \mathcal{P}} \mid T^{A \to \mathcal{P}} \mid \Pi x^A.(\Pi y^A.Ryx \to Ty) \to Tx]$

In peano numbers the first constructor type has no arguments. The second has one recursive argument.

2.3 Constructors

$$\mathcal{M}^{\mathcal{I}}_{\ell_i}\mathbf{pb} : \mathcal{I}\mathbf{pa}$$

where $\mathcal{I}$ is the inductive type, i marks the ith constructor, $\mathbf{p}$ are the parameter arguments and $\mathbf{b}$ are the constructor arguments, a are the index arguments which depend on the constructor arguments.

The peano number 2 looks like $\mathcal{M}^N_1(\mathcal{M}^N_1\mathcal{M}^N_0)$.

A constructor has the type

$$\mathcal{M}^{\mathcal{I}}_{\ell_i} : \Pi\Gamma\Delta_{\ell_i}.T\mathbf{a}_{\ell_i}$$

2.4 Typing

Let $\mathcal{I} = [\Gamma \mid T^K \mid C_{\ell_1}, \dots, C_{\ell_n}]$ be a wellformed inductive type. Then we have the following typing rules.

Inductive type

$$\frac{\begin{array}{l} K \overset{\beta}{\sim} \Pi\Gamma_K.s \\ \forall i.\ \Gamma, T^K \vdash C_{\ell_i} : s \end{array}}{[]\ \vdash \mathcal{I} : \Pi\Gamma.K}$$

Constructor Let $C_{\ell_i} = \Pi\Delta.T\mathbf{a}$.

$$\frac{[] \vdash \mathcal{I} : \Pi\Gamma.K}{\mathcal{M}^{\mathcal{I}}_{\ell_i} : \Gamma\Delta.T\mathbf{a}}$$

2.5 Mutually defined Inductive Types

It is just an array of inductive types where all inductive types share the same parameters.

$$\text{type}\left[\Gamma \mid \left[\begin{array}{l} T_1^{K_1} \mid C_{11}, \dots, C_{1n_1} \\ \dots \\ T_m^{K_m} \mid C_{m1}, \dots, C_{mn_m} \end{array}\right]\right]$$

All kinds K_i are valid in the context Γ and all constructors of the ith type construct an object of type $T_i\mathbf{a}$ but can use any other objects of type $T_j\mathbf{a}$ as arguments. All constructors are valid types in the context $\Gamma, T_1^{K_1},\ \dots, T_m^{K_m}$.

3 Pattern Match

3.1 Basics

Fully elaborated pattern match expression:

$$\mathrm{case}(f^F, [c_1, \dots, c_n], t)$$

where F is a function type of the form

$$\Pi x_1^{A_1} \dots x_k^{A_k}.R$$

and where each clause c_i has the form

$$(\Delta, [p_1^{P_1}, \dots, p_m^{P_m}], e^E)$$

where the context Δ contains all pattern variables introduced in the pattern, p_i is the i-th pattern and P_i is the corresponding type of the pattern, e is the body of the clause, E its corresponding type and t is a case tree (see below).

Each clause defines the function

$$\lambda \Delta . e^E$$

The number of toplevel pattern in all clauses of a case expression must be the same. The number of toplevel pattern in the clauses must not exceed the number of arguments of the corresponding function type $\Pi x_1^{A_1} \dots x_k^{A_k}.R$.

The patterns are terms generated from the grammar

$$p ::= x \mid x := c \mid x := \mathcal{M}_\ell^I \mathbf{q}\mathbf{p}$$

where c ranges over constants (strings, characters, numbers, etc.), x ranges over variables and p ranges over pattern. The expression $\mathcal{M}_\ell^I \mathbf{q}\mathbf{p}$ is the constructor with label ℓ of the inductive type I applied to its parameter arguments and to its arguments.

Patterns are trees. Each node of the tree is labelled by a pattern variable and either by a constant or a constructor $\mathcal{M}_\ell^I \mathbf{q}$. A pattern clause has a sequence of trees.

The pattern variables are assumed to be distinct in all clauses.

3.2 Welltyped

A clause of the pattern match expression is welltyped if

$$\begin{array}{lcl} \Delta & \vdash & p_i : P_i \\ \Delta & \vdash & e : E \end{array}$$

and

$$\begin{array}{lcl} P_i & \leq & A_i[p_1, \ldots, p_{i-1}/x_1, \ldots, x_{i-1}] \\ E & \leq & R[p_1, \ldots, p_m/x_1, \ldots, x_m] \end{array}$$

where $F = \Pi x_1^{A_1} \ldots x_m^{A_m}.R$.

3.3 Recursion

The bound variable f in a case expression $[f^F \mid \mathbf{c} \mid t]$ might occur in one of the right hand sides e of the case clauses in the form $f\mathbf{a}$ where the number of arguments is the same as the number of toplevel pattern in the corresponding case clause. In that case we have a recursive call where termination has to be guaranteed.

For each clause we have a list of pattern variables $\Delta = [y_1^{B_1}, y_2^{B_2}, \ldots]$. We attach to each pattern variable the number pair i/j where i is the position of the toplevel argument from which it comes and j is the level i.e. the number of constructors used to uncover it.

For each recursive call we attach to each argument which is a pattern variable (or a pattern variable applied to arguments (see below wellfounded recursion) coming from the same toplevel argument one of the symbols 0 or $-$ indicating if the argument is the toplevel argument or it has been structurally decreased. All other arguments get a $+$ which means *don't know*.

The recursive calls are valid if there are is a sequence of arguments such that they decrease in lexicograhic order with each recursive call.

This is possible only if there are candidates for the leftmost argument in the sequence of arguments. A candidate for the leftmost argument is an argument which either stays the same or decreases on all calls.

A candidate for the next argument in the sequence is an argument which decreases or stays the same when the previous stays the same.

A candidate for the first or next argument is a last argument in the sequence if it decreases in all calls where the previous (if present) stays the same.

The trivial case is when there is one argument which decreases in all recursive calls. In this case the sequence consists of one element only.

An argument which does not decrease in some calls cannot be part of the sequence.

Trying all possible candidates for the first and subsequent arguments either finds a sequence or fails. In the failure case termination is not guaranteed.

Example Ackermann's function

```
ack: Nat -> Nat := case
   \ zero  ,      n      := succ n

   \ succ m,      zero   := ack m (succ zero)
--        1/1                   -  +

   \ k := succ m, succ n := ack m (ack k n)
--   1/0       1/1     2/1             0 -
--                              -  +
--  recursive calls:
--   - +
--   - +
--   0 -
```

From this signature we see that either the first argument decreases (i.e. level > 0) or the first argument stays at the same level (i.e. level 0) and the second argument decreases.

3.4 Wellfounded Recursion

```
type Finite: Nat -> Prop :=
   finite {x}: (all {y}: y < x -> Finite y) -> Finite x
```

A pattern match on an object of type `Finite n` uncovers with the pattern `finite f` a function `f` and not another object of type `Finite n`. In that case the function `f` is considered structurally smaller than `Finite n` and all applications `f {y} (lt: y < n)` of type `Finite y` are structurally smaller than the object of type `Finite n`.

We can prove that all natural numbers are finite. In order to do this we need the helper theorems

```
splitLE: all {a b: Nat}: a <= b  ->  a < b \/ a = b
replace: all {A: Any} {P: A -> Prop} {a b}: a = b -> P a -> P b
```

and derive the proof

```
natFinite: all {n: Nat}: Finite n := case
   \ {zero}   := fin notLTZ
   \ {succ n} :=
let
    g: all {y}: Finite n -> (y < n \/ y = n) -> Finite y := case
        \ finite f, left  lt   := f lt
        \ finN,     right eq   := replace (flip eq) finN
    f: all {y}: y < succ n -> Finite y := case
        \ next le :=
            g natFinite (splitLE le)
:=
    finite f
```

Since all natural numbers are finite we can write recursive functions where the recursive calls are not necessarly structurally decreasing of one of the arguments of an inductive type, but where some measure which depends on the arguments decreases within the relation <.

Assume we want to write a function with one argument where some measure on the argument decreases in each recursive call. Then we equip the function with an additional argument stating that the measure is finite.

```
f: all (a: A): Finite (measure a) -> R := case
    \ a, finite g :=
        f a0 (g (lt: measure a0 < measure a))
```

Within the function we have to generate a proof that the measure really decreases. Since all natural numbers are finite, the function can be called on all arguments.

The same applies to functions with more arguments.

3.5 Mutual Recursion

A pattern match expression can be mutual recursive.

$$\text{case}\begin{bmatrix} f_1^{F_1} \mid \mathbf{c}_1 \mid t_1 \\ f_2^{F_2} \mid \mathbf{c}_2 \mid t_2 \\ \ldots \end{bmatrix}$$

Any f_i can occur in the right hand sides of any of the clauses of $\mathbf{c}_j$.

A recursive call happens if there is a cycle e.g. f_1 calls f_2 calls ... calls f_1.

In order to check termination we try to find a sequence of arguments as a subset of all arguments of all f_i such that the sequence decreases lexicographically in each cycle.

We work for each pattern variable with triples $i/j/k$ where i represents the function f_i from which it comes, j is the argument from which it comes and k is the number of constructors used to uncover it.

Following a cycle it is clear if the first function of the cylce receives in the recursive call an argument which is unchanged, decreased or potentially increased. With this information we can establish a signature of $+$, 0, and $-$ for each argument and try to find a sequence which decreases lexicographically for each cycle.

Example: Flatten a tree:

```
type Tree (A: Any): Any :=
    node: A -> List (Tree A) -> Tree A

flatT: Tree A -> List A := case
    \ node a ts := a :: flatF ts

flatF: List (Tree A) -> List A := case
    \ []         :=  []
    \ t :: ts    :=  flatT t + flatF ts
```

In this mutual recursion we have two functions each with one argument. The set of all arguments is $\{t, f\}$ where t represents the argument of `flatT` and f the argument of `flatF`. There are two cycles possible:

```
flatT: flatF, flatT             -- indirect recursion
flatF: flatF                    -- direct recursion
```

For each cycle we can find the signature:

```
--                                 t           f
flatT: flatF, flatT                -           0
flatF: flatF                       0           -
```

In the first cycle t decreases (by two) and f is not involved, in the second cycle f decreases (by one) and t is not involved. I.e. any sequence of t and f decreases lexicographically in each cycle.

3.6 Syntactical Pattern

The pattern in the source code are generated from the grammar

$$\begin{array}{rcll} p_s & := & x & \text{pattern variable} \\ & \mid & \ell \mathbf{p}_s & \text{constructor + arguments} \\ & \mid & c & \text{constant} \\ & \mid & x := \ell \mathbf{p}_s & \\ & \mid & x := c & \end{array}$$

A constructor label ℓ without arguments and a variable name x are indistiguishable in the source code. They are just names. But since the required type is known it is clear if the name is one of the labels of the corresponding inductive type. In that case the name is a label and not a pattern variable.

3.7 Case Tree

A case expression is a function which can be applied to arguments

$$\text{case}[\ldots]a_1 \ldots a_n$$

The task of a valid case tree is to

- split the arguments into a series of subterms $u_1, u_2, \ldots$
- decide which case clause is applicable (if sufficient information is available)
- return the result $e[u_1, u_2, \ldots /.]$ where e is the right hand side of the applicable case clause.

A case tree is defined by the grammar

$$\begin{array}{rcl} t & ::= & e \\ & | & [c_1t_1, \ldots, c_nt_n \mid \ell_1t_1, \ldots, \ell_mt_m \mid d] \end{array}$$

where $n + m > 0$ and

t	case tree
d	optional case tree (default or catch all)
e	expression on the right hand side of a case clause
c	constant
ℓ	label of a constructor

We can abbreviate the inner node by $[\mathbf{ct} \mid \ell\mathbf{t} \mid d]$.

No duplicate constants and no duplicate labels are allowed in an inner node.

Note that an inner node of the form $[\varepsilon \mid \varepsilon \mid \varepsilon]$ is a leave node.

Exhaustiveness of Inner Nodes An inner node of the form $[\mathbf{ct} \mid \ell\mathbf{t} \mid d]$ is exhaustive if it has a default tree or if all labels of constructors which can construct an object of the corresponding type are present (none of the missing constructors can construct an object of type T).

In order to verify that a constructor cannot construct an object of the required type T we create for all constructor arguments metavariables and unify the constructed type with the required type. Unification has to fail definitively (and not just for lack of knowledge).

An empty inner node of the form $[\varepsilon \mid \varepsilon \mid \varepsilon]$ is possible if T is an inductive type and none of the constructors of the inductive type can construct an object of type T.

Exhaustiveness of a case tree A case tree is exhaustive if all its nodes are exhaustive.

3.8 Application of a Case Tree

A pattern match expression is a function. It can be applied to arguments.

$$\text{case}[f^F, \mathbf{c}, t]\ \mathbf{a}$$

where $|\mathbf{a}|$ is the number of arguments needed by the pattern match expression.

We define the *spine-tree* form s of the arguments $\mathbf{a}$ by the grammar

$$\begin{array}{lcll} s & ::= & \varepsilon & \text{no arguments} \\ & \mid & [\mathcal{M}_\ell \mathbf{q}\mathbf{a}, s, s] & \text{constructor term} \\ & \mid & [x, s] & \text{non constructor term} \end{array}$$

and generate the spine-tree form of an argument list $\mathbf{a}$ by $f_{\text{st}}(\mathbf{a}, \varepsilon)$ using the recursive function

$$\begin{array}{lcl} f_{\text{st}}(t,s) & := & \left\{ \begin{array}{lcl} f_{\text{st}}(\mathcal{M}_\ell \mathbf{q}\mathbf{a}, s) & := & [\mathcal{M}_\ell \mathbf{q}\mathbf{a}, f_{\text{st}}(\mathbf{a}, s), s] \\ f_{\text{st}}(x, s) & := & [x, s] \end{array} \right. \\ f_{\text{st}}(\mathbf{a},s) & := & \left\{ \begin{array}{lcl} f_{\text{st}}(\varepsilon, s) & := & \varepsilon \\ f_{\text{st}}(\mathbf{a}t, s) & := & f_{\text{st}}(\mathbf{a}, f_{\text{st}}(t, s)) \end{array} \right. \end{array}$$

REWORK REWORK REWORK!!

The following needs rework using spine trees of arguments.

REWORK REWORK REWORK!!

A pattern match expression applied to the arguments $\mathbf{ab}$ where $\mathbf{a}$ are the arguments needed by the case expression can be reduced to a case tree application

$$\text{cta}(\text{pm}, t, \mathbf{u}, p)\mathbf{b}$$

where

1. pm: Pattern match expression $\text{case}[f^F, \mathbf{c}, t]$.
2. t: Case tree.
3. $\mathbf{u} = [u_1, u_2, \ldots]$: Collected subterms of the arguments $\mathbf{a}$. Initially empty ε.
4. p: Pointer into the arguments $\mathbf{a}$: Initially p points to the first argument of $\mathbf{a}$.

A pointer points to its focal term $p.f$ which is the subexpression of the arguments $\mathbf{a}$ which is currently to be matched.

$p.n$ is a pointer to next term after the focal point (or ε if there are no more subterms).

$p.a_i$ is a pointer to the i-th index argument of the focal term. This is possible only if the focal term is a constructur expression. Note the $p.a_1$ is the first argument after the parameter arguments of the constructor.

The case tree application is completed if the case tree is the right hand side $t = e$ of one of the case clauses and the pointer points to nowhere $p = \varepsilon$. In that case the reduced form of the case tree application is

$$e[\text{pm}, \mathbf{u}/.]$$

Note that the right hand side might contain the free variable f and the pattern variables. The free variable f is replaced by the pattern match expression and the pattern variables are replaced by the corresponding subterms. In the recursive case the right hand side e contains the free variable and therefore the reduct contains the pattern match expression again.

The case tree application scans via the pointer the list of arguments left to write. Semantic actions associated to the nodes of a case tree:

- e: The pointer has to point beyond the last argument of the arguments. The list of collected subexpression $u_1, u_2, \dots$ cover all free variables in the expression e. Action: Return the value $e[\text{case}[f^F \mid \mathbf{c} \mid t], u_1, u_2, \dots /.]$.
- $[\mathbf{ct} \mid \ell\mathbf{t} \mid d]$: The pointer has to point to some subexpression of the arguments. The subexpression has to reduce to a head normal form which has either a constant or a constructor in the head position. From the constant or the constructor it is clear which case tree is the next to apply. Actions:
  - Append the subexpression to the list $u_1, u_2, \dots$.
  - If the default tree is the next tree then apply it to the next pointer position.
  - If the subexpression is a constant then apply the corresponding next tree to the next pointer position.
  - If the subexpression is a constructor without index arguments then apply the corresponding next tree to the next pointer position.

  - If the subexpression is a constructor with arguments then apply the corresponding next tree to the pointer pointing at the first index argument of the constructor (skip the parameter arguments).

3.9 Case Tree of a Clause

First we construct a case tree from a cause clause

$$[\Delta \mid \mathbf{p} \mid e]$$

.

The case tree corresponding to e is e. The case tree $f_{\mathrm{ct}}(p,t)$ of a pattern p where t is a partially constructed case tree is defined recursively

$$\begin{array}{lll}
f_{\mathrm{ct}}(p,t) & := & \left\{ \begin{array}{lll} f_{\mathrm{ct}}(x,t) & := & [\varepsilon,\varepsilon,t] \\ f_{\mathrm{ct}}(x := c,t) & := & [ct,\varepsilon,\varepsilon] \\ f_{\mathrm{ct}}(x := \mathcal{M}_\ell^{\mathcal{I}}\mathbf{qp},t) & := & [\varepsilon,\ell f_{\mathrm{ct}}(\mathbf{p},t),\varepsilon] \end{array} \right. \\
f_{\mathrm{ct}}(\mathbf{p},t) & := & \left\{ \begin{array}{lll} f_{\mathrm{ct}}(\varepsilon,t) & := & t \\ f_{\mathrm{ct}}(\mathbf{p}q,t) & := & f_{\mathrm{ct}}(\mathbf{p},f_{\mathrm{ct}}(q,t)) \end{array} \right.
\end{array}$$

The case tree of a case clause $[\Delta \mid \mathbf{p} \mid e]$ is defined as $f_{\mathrm{ct}}(\mathbf{p},e)$.

Note that the case tree of a cause clause has no branching. It is a path from the root to a leave.

3.10 Merge Case Trees

We have to be able to merge two case trees t_1 and t_2 to $m(t_1,t_2)$ where the first one might an already merged case tree and the second one has been constructed from a clause of the pattern match expression.
Error cases:

1. $m(e,t)$: This case is ambiguous. The first case tree would return e, the second would apply t.
2. $m(t,e)$: This cas is ambiguous as well. The first tree t would be applied to deliver some result the second tree would immdiately return e.
3. $m([. \mid . \mid t_1],[. \mid . \mid t_2]$: Ambiguous: Which default shall be taken?
4. $m([. \mid \ell\mathbf{t}_1 \mid \varepsilon],[\varepsilon \mid \varepsilon \mid t_2])$ where ℓ covers all possible constructors for the corresponding inductive type: Default case is redundant.

5. $m([. \mid \ell\mathbf{t}_1 \mid .], [ct_t \mid \varepsilon \mid \varepsilon])$ where ℓ has at least one constructor: Out of order. All constants have to be merged before the first constructor case.

6. $m([. \mid . \mid t], [ct_2 \mid \varepsilon \mid \varepsilon])$ or $m([. \mid . \mid t], [. \mid \ell t_2 \mid \varepsilon])$: Out of order. All constants and constructors have to be merged before any default case.

Success cases of merging a case tree from a case clause into another case tree:

1. Merge a default tree: In the first case tree the top node must not be exhaustive.

2. Merge an already available constant: Merge the subtrees corresponding to the constant.

3. Merge a new constant: The top node of the first case tree must not yet have any constructors. Add the new constant and its case trees to the available constants and case trees.

4. Merge an alreay available constructor: Merge the subtree corresponding to the constructor.

5. Merge a new constructor: Add the new constructor with its case tree to the available constructors and their case trees.

3.11 Algorithm to Construct a Case Tree

1. Construct a case tree from the first clause.

2. For all remaining clauses: Construct a case tree from the clause and merge it into the existing case tree.

3. Check if the constructed case tree is exhaustive.

No case clauses This is allowed only if in the type $\Pi x_1^{A_1} \dots x_k^{A_k}.R$ of the case expression there is one A_i which is an inductive type which cannot be constructed (e.g. False or zero $=$ succ n).

3.12 Code Examples

Unbounded Loop

```
type Decision (P: Prop): Any :=
    true:  P     -> Decision
    false: Not P -> Decision
```

```
Decider {A: Any} (P: A -> Prop): Any :=
    all x: Decision (P x)

type Finite: Nat -> Prop :=
    fin {x}: (all y: y < x -> Finite y) -> Finite x

type Refine {A: Any} (P: A -> Prop): Any :=
    (,) x: P x -> Refine

LB (P: Nat -> Prop) (x: Nat): Prop :=
    all {y}: P y -> x <= y

Least (P: Nat -> Prop) (x: Nat): Prop :=
    LB P x /\ P x

natFinite: all {n}: Finite n :=
    -- induction proof

zeroLB {P: Nat -> Prop}: LB P zero := ...

lbSucc {P: Nat -> Prop}:
    all {n}: LB P n -> Not (P n) -> LB P (succ n)
:= ...

invLT {a b c: Nat}: a <= c -> b < a -> c - a < c - b
:= ...

leRefl {a: Nat}: a <= a :=
:= ... -- induction proof

find {P: Nat -> Prop} (d: Decider P): Exist P -> Refine (Least P)
:= case
    \ (w, pW) :=
        let
            aux: all n: LB P n -> Decision (P n) -> Finite (w - n)
                -> Refine (Least P)
            := case
                \ n, lbN, true pN, ? :=
                    (n, (lbN, pN))
                \ n, lbN, false npN, fin f :=
                    let
                        lbSN: LB P (succ n) :=
                            lbSucc lbN npN
                        ltWmSN: w - succ n < w - n :=
                            invLT (lbSN pW) leRefl
                    :=
                        aux (succ n) lbSN (d (succ n)) (f ltWmSN)
        :=
            aux zero zeroLB (d zero) natFinite where
```

Half

```
type Half: Nat -> Any :=
    even n: Half (n + n)
    odd  n: Half (succ (n + n))
```

```
half: all n: Half n := case
    \ zero          :=   even zero
    \ succ zero     :=   odd zero
    \ succ (succ n) :=
        match half n case
            : Half n -> Half (succ (succ n)) -- type usually inferred
            \ even h := even (succ h)
            \ odd  h := odd  (succ h))
```

This function does not typecheck. Let's look into the cases:

```
--  Required types                      Actual types
    Half (succ (succ (h + h)))          Half (succ (h + succ h))
    Half (succ (succ (succ (h + h))))   Half (succ (succ (h + succ h))
```

We have to pull out the successor constructor of the second argument out of the parenthesis. With a proof of

```
succ (h + h) = h + succ h
```

the required types and the actual type would be unifiable.

```
value {n}: Half n -> Nat := case
    \ even h := h
    \ odd  h := h

halfLT {n} (lt: 0 < n):  value (half n) < n :=
    let
        aux {n} {h: Half n}: 0 < n  ->  value h < n := case
            \ {even h},       lt :=
            -- : Half (h+h)   : 0 < h + h
                ?: h < h + h
            \ {odd h},              ? :=
            --: Half (succ (h+h))
                ?: h < succ (h + h)
    := aux {half n} lt
```

Exponentiation

```
(^) (a b: Nat): Nat :=
    let
        exp: all n: Half n -> Finite n -> Nat := case
            \ zero, ?, ? :=
                succ zero
            \ succ i, even h, fin f :=
                let
                    ltH: h < succ i := ...
                    r := exp h (f ltH)
                :=
                    r * r
            \ succ i, odd h, fin f :=
                let
                    ltH: h < succ i := ...
                    r := exp h (f ltH)
                :=
                    a * r * r
    :=
        exp b (half b) natFinite
```

Ackermann Function

```
ack: Nat -> Nat -> Nat := case
    \ zero  ,       n      :=  succ n
    \ succ m,       zero   :=  ack m (succ zero)
    \ k := succ m, succ n :=  ack m (ack k n)
```

Mutual Recursion

```
type Tree (A: Any): Any :=
    node: A -> List (Tree A) -> Tree A

preT: Tree A -> List A := case
    \ node a ts := a :: preF ts

preF: List (Tree A) -> List A := case
    \ []          :=  []
    \ t :: ts     :=  preT t + preF ts
```

Vector

```
type Vec (A: Any): Nat -> Any :=
    []:  Vec zero
    (::) {n: Nat}: A -> Vec n -> Vec (succ n)

zip {A B: Any}
: all {n}: Vec A n -> Vec B n -> Vec (A,B) n
:= case
    \ nil,     nil     := nil
    \ x :: xs, y :: ys := (x, y) :: zip xs ys

map {A B C} (f: A -> B -> C)
    : all {n}
      : Vec A n -> Vec B n -> Vec C n
:= case
    \ {zero},   [],            [] :=
        []
    \ {succ n}, (::) {n} x xs, (::) {n} y ys :=
        (::) {n} (f x y) (map {n} xs ys)
    -- without implicits
    \ [] []            := []
    \ x :: xs, y :: ys := (f x y) (map xs ys)
```

Less Equal on natural numbers

```
type (<=): Nat -> Nat -> Prop :=
    start {n}:    0 <= n
    next  {n m}:  n <= m -> succ n <= succ m
```

```
leRefl: all {n: Nat}: n <= n := case
    \ {zero}   := start {zero}
    \ {succ n} := next {n} {n} (leRefl {n})

    -- without implicits
    \ {zero}   := start
    \ {succ n} := next leRefl
```

Pattern match on implicits is allowed in this case, because the result type is a proposition!

Equality

```
type (=) (A: Any): A -> A -> Prop :=
    same {x}: x = x

zeroNeSucc: all {n: Nat}: zero = succ n -> False :=
    case
        -- no case clauses
```

The compiler has to verify that no match is possible. The pattern match expression is the two argument function with type $\Pi n^N . 0 = 1 + n \to \text{False}$.

<=?

```
type Nat := [zero, succ: Nat -> Nat]

(<=?): Nat -> Nat -> Nat := case
    \ zero,   _      :=  true
    \ succ _, zero   :=  false
    \ succ n, succ m :=  n <=? m

-- as case tree:
case
    zero          :=   \ _ := true
    succ n :=
        case
            zero   :=   false
            succ m :=   n <=? m
```

Parity

```
type Parity: Nat -> Any :=
    even n: Parity (n + n)
    odd  n: Parity (succ (n + n))

parity: all n: Parity n := case
    \ zero: Parity zero :=
        even
    \ succ n: Parity (succ n) :=
        match parity n case
            \ even nh :=
                odd nh
            \ odd nh :=
                even (succ nh)

natToBin: Nat -> List Bool := case
    \ zero :=
        []
    \ succ n :=
        match parity n case
            \ even nh :=
                false :: natToBin nh
            \ odd nh :=
                true :: natToBin nh
```

4 Ambiguity

4.1 Simplified Types

Simplified terms are defined by the grammar

$$
\begin{array}{lll}
F, A, B := & & \\
\quad | & S & \text{sort} \\
\quad | & U & \text{unknown} \\
\quad | & {}^{?}X & \text{metavariable} \\
\quad | & C & \text{constant, type constructor} \\
\quad | & F\,A & \text{application} \\
\quad | & A \to B & \text{function type} \\
\quad | & \{A\} \to B & \text{function with implicit argument}
\end{array}
$$

Code Examples:

```
-- dependent type                        simplified type

String -> String -> String               String -> String -> String

all {A: Any} (a: A) (n: Nat): Vec A n    {S} -> U -> Nat -> Vec U U

all {A: Any}: A                          {S} -> U

all {A: Any}: A -> A                     {S} -> U -> U

all {A: Any} {P: A -> Any} (a: A) (f: all x: P x): P a
                                         {S} -> {U -> S} -> U -> (U -> U U) -> U U

Nat -> Nat -> Prop                       Nat -> Nat -> S

all {n: Nat}: zero <= n                  {Nat} -> (<=) U U
```

Assume that a term is in head normal form and h is a function mapping a term to its head normal form without unfolding definitions then we can define the recursive function f mapping terms from head normal form to simplified terms.

$$
\begin{array}{llll}
f(\mathcal{P}) & := & S & \\
f(\mathcal{U}) & := & S & \\
\text{cta}[*]\mathbf{a} & := & U\mathbf{a} & \\
f(x\mathbf{a}) & := & U f(h(\mathbf{a})) & x \text{ is type used} \\
f(x\mathbf{a}) & := & x f(h(\mathbf{a})) & \\
f(\Pi x^A.B) & := & O \to f(h(B)) & x^A \text{ is implicit} \\
f(\Pi x^A.B) & := & f(h(A)) \to f(h(B)) & \\
f(\lambda x^A.e) & := & U &
\end{array}
$$

A variable x is type used if it has been introduced by mapping $\Pi x^A.B$.

The context Γ of the terms has been omitted for clarity but clearly the mapping of a term $\Pi x^A.B$ and the mapping to head normal form does introduce variables into the context.

Note that mapping a term into a simplified term introduces variables without definition only by mapping $\Pi x^A.B$. The mapping to head normal form only introduces definitions.

5 Find Terms

5.1 Basics

There is often the need starting from a specific term to find an abstract term (i.e. a term with variables) and a substitution such that the abstract term when substituting the substitution for the variables matches the specific term.

Let t be the specific term and u the abstract term and $\mathbf{a}$ is a substitution then

$$t = u[\mathbf{a}/.]$$

must be valid.

The abstract terms are generated from the grammar

$$\begin{array}{lll} u & ::= & x\mathbf{u} & \text{variable applied} \\ & | & c\mathbf{u} & \text{constant applied} \\ & | & \mathcal{M}_\ell \mathbf{qu} & \text{constructor applied} \end{array}$$

Note that the number of arguments might be zero.

The following is not limited to terms satisfying this grammar. The grammar can be extented to full fledged terms. However the above choice is the usual form, because we store terms in head normal form, all the Πs, λ and cases all treated in different manners.

As in the case of pattern match we can define decision trees generated by the grammar

$$\begin{array}{llll} t & :: & [\Delta \mid u] & \text{abstract term with variables} \\ & | & [\mathbf{xt} \mid \mathbf{ct} \mid \ell\mathbf{t}] & \text{inner node} \end{array}$$

Like for pattern match expression we can generate a decision tree from an abstract term by starting from the rear end to the front end and collecting all variables and using each headterm as an inner node. A tree generated from a term is a linear decision tree where each inner node has only one branch.

Decision trees can be merged to obtain a real tree structure.

We apply a decision tree to a specific term by scanning it from left to write, using the current symbol in the decision tree to find the next tree and in case of a variable collect the corresponding subterm as a substitution term for this variable. If a variable already has a substitution term, then both have to be equivalent (which is identical in normal form).

In case that the application of a decision tree fails at some point of the scanning then the term is not represented by the decision tree.

5.2 Propositions

Finding terms by decision trees can help to prove assertions. E.g. we have the general assertion

$$\Pi a^N b^N u^N . a < b \to b \leq u \to u - b < u - a$$

and the goal

$$w - \text{succ } i < w - i$$

(see code example of unbounded search). It is easy to see that the goal matches the result type of the general assertion. The application of a decision tree would point to the result type with the substitutions $i, \text{succ i}, w$ for a, b, u.

We can use the substitution to derive the premises $i <$ succ i and succ i $\leq w$ for the goal.

6 Unification

6.1 Description of the Problem

Whenever a function is applied to an actual argument then the type of the actual argument A has to be a subtype of the type of the formal argument R i.e. $A \leq R$ has to be valid. Otherwise the actual argument is not a legal argument to the function. Without metavariables and performance considerations the solution is quite simple:

- Transform both types into normal form. Since both are types their normal forms have to be products with zero or more arguments and the result type is either a sort or a base term i.e. they have one of the forms

$$\begin{array}{l} \Pi x_1^{A_1} \ldots x_n^{A_n}.x\mathbf{a} \\ \Pi x_1^{A_1} \ldots x_n^{A_n}.s \end{array}$$

- Check that both have the same number of arguments and all argument types are identical.
- The result types have to be either both base terms or sorts. In case of base terms both have to be identical. In case of sorts the sort of the actual argument type has to be a subtype of the sort of the formal argument type.

There are two problems with this approach:

- The source code is not fully annotated. Metavariables are introduced during elaboration for terms missing in the source code. The elaborator has to instantiate these metavariables.
- A complete normalization performs a lot of function unfolding, case expression reductions, let term reductions and beta reductions. Operation of let and beta reduction is variable substitution in terms. Since variables can occur several times in terms the normalized terms can be considerably large even growing exponentially.

  Furthermore case term reductions can grow exponentially in the worst case.

  Therefore reduction to complete normal form can be a performance problem.

We work with contexts

$$\Gamma ::= [] \mid \Gamma, x^A \mid \Gamma, x^A := a \mid \Gamma, \overset{?}{m}^M \mid \Gamma, \overset{?}{m}^M := a$$

Clearly duplicate names are not allowed.

The unification problem looks like

$$\Gamma \vdash A \leq R$$

where A and R are welltyped in the context Γ.

6.2 Normalization

MISSING!!!

6.3 Metavariables

Metavariables are introduced when there is something unknown in a certain context (e.g. a missing type, an implicit variable). Each metavariable has a type $\Gamma \vdash \overset{?}{m} : M$. I.e. metavariables are required to be welltyped.

The type M of a metavariable is a required type. The actual type used in instantiations can be a subtype of M.

If we have two metavariables $\overset{?}{m}_1$ and $\overset{?}{m}_2$ and both are separated in the context by a free variable (i.e. variable without definition $\Gamma = [\ldots, \overset{?}{m}_1^{M_1}, \ldots x^A, \ldots, \overset{?}{m}_2^{M_2}, \ldots]$) then the meta variable m_2 is an *inner metavariable* with respect to $\overset{?}{m}_1$. Metavariables which are not separated by free variables are in the same group.

There are the following reasons to introduce metavariables:

1. Missing type annotation: In the source code a variable is introduced without an explicit type (e.g. types in global or local definitions, argument type of an a abstraction or a product, argument type of a let binding). The usage of the variable shall finally instantiate the metavariable standing for the type.

2. Implicit actual argument: In a function application the implicit arguments can be ommitted in the source code. For each implicit actual argument a metavariable is introduced. Usually the implicit argument is used in one of the subsequent argument types or in the result type. Having these types the implicit argument can be instantiated.

3. Interleaved elaboration: In the elaboration of an application fa the function term f and the argument term a can be elaborated in parallel. A metavariable for the argument a and its type is introduced. The elaboration of the function term f can fill the argument type and the elaboration of the argument can fill the argument and the type.
4. Wildcard: In the source code a wildcard can be used to ask the compiler to fill the wildcard from information it already has.
5. Constructor arguments in pattern of case expressions.
6. The unification might introduce metavariables.

6.4 Flex-Rigid Constraints

A *flex-rigid* constraint looks like

$$\Gamma_0, \Gamma_1 \vdash {}^?m\mathbf{a} \sim \begin{cases} h\mathbf{b} \\ \Pi x^A.B \\ \lambda x^A.e \end{cases}$$

where Γ_1 is the context where the metavariable ${}^?m$ has been introduced and the typing conditions $\Gamma_0 \vdash {}^?m : \Pi\mathbf{y}^{\mathbf{A}}.R$ and $\Gamma_0, \Gamma_1 \vdash$ *right hand side* $: T$ are valid.

The left hand side is called flexible because its structure can be changed by substitutions and reductions, the right hand side is called *rigid* because its structure cannot be changed by substitutions and reductions.

The left and right hand side are in head normal form.

The constraint is satisfiable when the typing condition

$$\Gamma_0, \Gamma_1 \vdash T \leq R[\mathbf{x}/\mathbf{y}]$$

is valid which implies that all free variables of the type of the right hand side T are within $\mathrm{FV}(\Gamma_0) \cup \mathbf{x}$. The typing condition is a subtype because metavariables are introduced with the most general type and can be instantiated by an object of a more specific type.

6.4.1 Ambiguity

In higher order unification there is no *most general unifier*. On the contrary, multiple unifiers can exist. The simplest example to demonstrate the ambiguity is the contraint ${}^?mx \sim x$ where $x \in \Gamma_1$. It has the solutions

$$\,^?m := \begin{cases} \lambda y^A.x & \text{constant function} \\ \lambda y^A.y & \text{identity function} \end{cases}$$

Both solutions satisfy the constraint. Multiple solutions might cause backtracking which we want to avoid.

6.4.2 Huet's Algorithm

Gérard Huet's higher order unification algorithm gives a general procedure to solve constraints of the form

$$\Gamma_0, \Gamma_1 \vdash \,^?m\mathbf{a} \sim h\mathbf{b}$$

where h is either a constant of a bound variable. In our setting all variables $x_i \in \Gamma_1$ and all global variables correspond to constants and all $x_j \in \Gamma_2$ correspond to bound variables.

Any valid instantiation of $^?m$ must have the form $\lambda\mathbf{y}^{\mathbf{B}}.e$ such that $^?m\mathbf{a} = e[\mathbf{a}/\mathbf{y}] \rightsquigarrow h\mathbf{b}$. This is possible only if $e = h\ldots$ (*imitation*) or $e = y_i\ldots$ (*projection*) i.e. either e has h is its head term or $a_i \ldots \rightsquigarrow h \ldots$.

In order to get the most general instantiation some fresh metavariables $^?h_1, ^?h_2, \ldots$ valid in the context Γ_1 are introduced

$$\,^?m := \lambda\mathbf{y}^{\mathbf{B}}.\begin{cases} h(^?h_1\mathbf{y})\ldots(^?h_n\mathbf{y}) & \text{imitation} \\ y_i(^?h_1\mathbf{y})\ldots(^?h_m\mathbf{y}) & \text{projection} \end{cases}$$

where n is the arity of h and m is the arity of y_i, giving one of the constraints

$$\begin{array}{lcll} h(^?h_1\mathbf{a})\ldots(^?h_n\mathbf{a}) & \sim & h\mathbf{b} & \text{imitation} \\ a_i(^?h_1\mathbf{a})\ldots(^?h_m\mathbf{a}) & \sim & h\mathbf{b} & \text{projection} \end{array}$$

If h is a bound variable (i.e. $h \in \Gamma_2$), then imitation is not possible. Otherwise the instantiation of $^?m$ would contain a bound variable from an inner context.

The constraint in the imitation case has the rigid-rigid form and immediately results in the simpler problems

$$\Gamma_0, \Gamma_1 \vdash \,^?h_i\mathbf{a} \sim b_i$$

whilst the projection case might introduce new redexes therefore might lead to more complex problems. The projection case is the reason why Huet's algorithm might not terminate and is therefore only a semidecision procedure.

The general solution of a flex-rigid constraint in Huet's algorithm might generate up to $1 + n$ alternatives where n is the arity of the metavariable $\overset{?}{m}$.

6.4.3 Algorithm with Restriction

In order to avoid backtracking we try to instantiate the metavariable $\overset{?}{m}$ only if the constraint is in pattern form

$$\Gamma_0, \Gamma_1 \vdash \overset{?}{m}\mathbf{x} \sim \begin{cases} h\mathbf{b} \\ \Pi z^C.D \\ \lambda z^C.e \end{cases}$$

where $\Gamma_0 \vdash \overset{?}{m} : \Pi\mathbf{y}^{\mathbf{A}}.R$, $\Gamma_0, \Gamma_1 \vdash$ *right hand side* $: T$ and $\mathbf{x} \in \Gamma_0, \Gamma_1$ are different variables (multiple occurrences of the same variable would be a quasi pattern form). This has the advantage that most of the solutions are unique.

1. $\overset{?}{m} \sim$ *right hand side*:

   This is the trivial case which has (provided the type condition is satisfied and the right hand side has no variables from the inner context Γ_1) the solution

   $$\overset{?}{m} := \textit{right hand side}$$

2. $\overset{?}{m}\mathbf{x} \sim x_i$ where $x_i \in \Gamma_0$: This is the only case which creates an ambiguity. It has the solutions

   $$\overset{?}{m} := \begin{cases} \lambda\mathbf{y}.x_i & \text{imitation} \\ \lambda\mathbf{y}.y_i & \text{projection} \end{cases}$$

   This disambiguity can be resolved if we encounter another constraint of one of the two forms

   $$\overset{?}{m}\mathbf{z} \sim \begin{cases} x_i & \text{where } z_i \neq x_i \\ z_i & \text{where } z_i \neq x_i \end{cases}$$

   The first form requires imitation i.e. $\overset{?}{m}\mathbf{z}$ ignores the arguments and always returns x_i. The second form requires projection i.e. $\overset{?}{m}\mathbf{z}$ always projects onto the ith argument.

3. $\,^?m\mathbf{x} \sim x_i$ where $x_i \in \Gamma_1$: The only possible solution is projection, because any solution must not contain variables from inner contexts
$$\,^?m := \lambda \mathbf{y}.y_i$$
4. $\,^?m\mathbf{x} \sim h\mathbf{b}$ where $h \in \mathbf{x} \rightarrow (h \in \Gamma_0 \wedge |b| > 0)$: The only solution is imitation.

   We introduce fresh metavariables $\,^?h_1, \ldots \,^?h_n$ with the types $\Gamma_0 \vdash \,^?h_i : \Pi \mathbf{y}^{\mathbf{A}} : B_i$ where n is the arity of h and B_i is the type of b_i. This requires that B_i is a valid type in the context Γ_0, Δ where Δ is Γ_1 restricted to the variables $\mathbf{x}$.

   The constraint can be solved by
$$\,^?m := \lambda \mathbf{y}^{\mathbf{A}}.h(\,^?h_1\mathbf{y}) \ldots (\,^?h_n\mathbf{y})$$
   which converts the constraint into
$$h(\,^?h_1\mathbf{x}) \ldots (\,^?h_n\mathbf{x}) \sim h\mathbf{b}$$
   which is rigid-rigid and can be simplified. Note that simplification leads to new constraints of the form $\,^?h_i\mathbf{x} \sim b_i$ which are in pattern form.
5. $\,^?m\mathbf{x} \sim \Pi z^C.D$:

   We introduce fresh metavariables $\,^?h_1 : \Pi \mathbf{y}^{\mathbf{A}}.\mathcal{U}_\infty$ and $\,^?h_2 : \Pi \mathbf{y}^{\mathbf{A}} z^{\,^?h_1\mathbf{y}}.\mathcal{U}_\infty$. Then the constraint can be solved by the instantiation
$$\,^?m := \lambda \mathbf{y}^{\mathbf{A}}.\Pi z^{\,^?h_1\mathbf{y}}.\,^?h_2\mathbf{y}z$$
   This immediately generates the constraint
$$\Gamma_0, \Gamma_1 \vdash \Pi z^{\,^?h_1\mathbf{x}}.\,^?h_2\mathbf{x}z \sim \Pi z^C.D$$
   which can be simplified.
6. $\,^?m\mathbf{x} \sim \lambda z^C.e$:

   We introduce fresh metavariables $\,^?h_1 : \Pi \mathbf{y}^{\mathbf{A}}.\mathcal{U}_\infty$, $\,^?h_2 : \Pi \mathbf{y}^{\mathbf{A}} z^{\,^?h_1\mathbf{y}}.\mathcal{U}_\infty$ and $\,^?h_3 : \Pi \mathbf{y}^{\mathbf{A}} z^{\,^?h_1\mathbf{y}}.\,^?h_2\mathbf{y}z$. Then the constraint can be solved by the instantiation
$$\,^?m := \lambda \mathbf{y}^{\mathbf{A}} z^{\,^?h_2\mathbf{y}}.\,^?h_3\mathbf{y}z$$
   This immediately generates the constraint
$$\Gamma_0, \Gamma_1 \vdash \lambda z^{\,^?h_1\mathbf{x}}.\,^?h_3\mathbf{x}z \sim \lambda z^C.e$$
   which can be simplified.

6.4.4 Code Example

Suppose we have the code

```
type (=) (A: Any): A -> A -> Prop :=
    same {x}: x = x

flip {A: Any} {a b: A}: a = b -> b = a :=
    ...

adapt {A B: Any} {a b: A} {F: A -> B}: F a -> F b :=
    ...

(,) {A: Any} {a b c: A} (ab: a = b) (bc: b = c): a = c :=
    adapt {A} {b} {a} {?F} (flip ab) bc
```

and we want to instantiate ?F by unification.

```
-- unification constraints
?F b   ~   b = c
?F a   ~   a = c

-- instantiation from the first flex-rigid constraint
?F := (\ x := ?h1 x = ?h2 x)

-- after simplification
?h1 b  ~  b             -- ambiguous: imitation or projection
?h2 b  ~  c             -- only imitation is possible

?h2 := (\ x := c)

-- instantiation of ?F and ?h2 into the second constraint
?h1 a = c    ~    a = c

-- after simplification
?h1 a  ~  a             -- now only projection is possible
                        -- ambiguity resolved
?h1 := (\ x := x)

-- instantiation of ?F
?F x := (\ x := x) x = (\ x := c) x

?F x := x = c   -- reduced
```

6.5 Quasipattern Constraints

MISSING!!!

6.6 Rigid-Rigid Constraints

Rigid-rigid constraints have one of the forms

$$\begin{array}{lcll}
h_a\mathbf{a} & \leq & h_b\mathbf{b} & \text{base terms} \\
\Pi x^{A_1}.B_1 & \leq & \Pi x^{A_2}.B_2 & \text{products} \\
\lambda x^{A_1}.e_1 & \sim & \lambda x^{A_2}.e_2 & \text{abstractions} \\
\lambda x^{A}.e & \sim & h\mathbf{b} & \text{abstraction and base term}
\end{array}$$

A rigid-rigid constraint can be resolved only if the left and the right side have the same structure (except the last one which can be valid only if η-reduction is allowed). A lambda abstraction can never be a type, therefore a subtype constraint is not meaningful for lambda abstractions.

6.6.1 Base Terms

$$\Gamma \vdash h_1\mathbf{a}_1 \leq h_2\mathbf{a}_2$$

MISSING!!!!

6.6.2 Products

The constraint $\Gamma \vdash \Pi x^{A_1}.B_1 \leq \Pi x^{A_2}.B_2$ can be simplified to

$$\begin{array}{lcll}
\Gamma & \vdash & A_2 \leq A_1 & \text{arguments contravariant} \\
\Gamma, x^{A_2} & \vdash & B_1 \leq B_2 & \text{results covariant}
\end{array}$$

The constraint $A_2 \leq A_1$ has to be resolved before the second one, otherwise B_1 would not be welltyped in the context Γ, x^{A_2}.

Why do we use A_2 as the type of x? We have the general typing rule specialized for variables

$$\frac{\begin{array}{c}\Gamma \vdash x : T \\ T \leq U\end{array}}{\Gamma \vdash x : U}$$

If a variable has type T which is a subtype of type U, then the variable has type U as well. Therefore in the above requirement for the result type we have to use the stronger (subtype) of the types A_1 and A_2.

6.6.3 Abstractions

The constraint $\Gamma \vdash \lambda x^{A_1}.e_1 \sim \lambda x^{A_2}.e_2$ can be simplified to

$$\begin{array}{lcl} \Gamma & \vdash & A_1 \sim A_2 \\ \Gamma, x^{A_1} & \vdash & e_1 \sim e_2 \end{array}$$

The constraint $A_1 \sim A_2$ has to be resolved before the second one, otherwise e_2 would not be welltyped in the context Γ, x^{A_1}.

6.6.4 Abstractions and Base Terms

The constraing $\Gamma \vdash \lambda x^{A}.e \sim h\mathbf{b}$ can be simplified to

$$\Gamma, x^{A} \vdash e \sim h\mathbf{b}x$$

provided that the left and right hand side of the contraint have the same type.

7 Elaboration

7.1 Abstractions

$$\Gamma \vdash \lambda x^A.e^E : R$$

The type annotations A and E are optional.

An elaboration can be successful only if it elaborates a term which satifies the requirement R. There are only two possibilities to satisfy the requirement.

1. R represents a function type i.e. it has the form $\Pi x^{A_r}.B_r$. In that case the requirement can flow down to give requirements for A, E and e.
2. R has a metavariable ${}^?M$ as a head term and R is a pattern i.e. $R = {}^?M\mathbf{y}$ where $\mathbf{y}$ are pairwise distinct free variables. In that case the requirement can flow up. I.e. the elaboration elaborates a term with the type $\Pi x^A.B$ and the unification of the type with R instantiates ${}^?M$.

1. $R = \Pi x^{A_r}.B_r$: I.e. the required type is a function type.
   (a) If A is present then elaborate A with $\Gamma \vdash A_r \leq A$. Otherwise use $A = A_r$.
   (b) If E is present then elaborate E with $\Gamma, x^A \vdash E \leq B_r$. Otherwise use $E = B_r$.
   (c) Elaborate e with $\Gamma, x^A \vdash e : E$.
   (d) Form $\Gamma \vdash \lambda x^A.e^E : \Pi x^A.E$ where $\Pi x^A.E$ is guaranteed to be a subtype of $\Pi x^{A_r}.B_r$.
   The forming of $\lambda x^A.e^E$ requires that all metavariables belonging to the context Γ, x^A are instantiated. This usually requires waiting (e.g. e is usually an application where the metavariables representing arguments have to be instantiated).
2. $R = {}^?M\mathbf{a}$: The required type is represented by an uninstantiated metavariable ${}^?M$.
3. All other cases are error cases.

   Open question: What happens with a stuck case tree application?

7.2 Applications

$$fa$$

The elaborator of fa starts with a signature requirement $[R]$ and an optional required type.

Subtasks

1. E_f: Elaborate f
2. E_a: Elaborate a
3. U_A: Unify the type of a as a subtype of the argument type of f
4. U_R: Unify the type of $^?f\,^?a$ as a subtype of the required type of fa.
5. E_{fa}: Elaborate fa

The subtasks E_f, U_A, U_R and E_{fa} have to be executed in sequence. The subtask E_a can run interleaved.

Algorithm

1. Make holes:
   (a) Make the unkown signature element $^?U$.
   (b) Make a hole $^?f$ for f with signature requirement $[^?U, R]$.
   (c) Make a hole $^?A$ for the type of a with the signature requirement $[S]$.
   (d) Make a hole $^?a$ for a with signature requirement $[^?U]$ and the required type $^?A$.
2. Waiting tasks:
   (a) Put U_A into the wait queue for $^?f$.
   (b) Put U_R into the wait queue for U_A.
   (c) Put E_{fa} into the wait queue for U_R.
3. Ready tasks:
   (a) Push E_a into the ready queue.
   (b) Push E_f into the ready queue.

Remarks

- The elaborators of f and a don't have any preconditions. For the result of the elaboration the sequence is not important. However it is preferable to start the elaboration of f before the elaboration of a. The elaboration of a has a better chance to be successful without becoming stuck if the elaboration of f has finished and the required type ${}^{?}A$ for a is available.
- The term fa can be built as soon as the type of a is unified with the argument type of f and the type of ${}^{?}f{}^{?}a$ is unified with the required result type. Before that it is not evident that fa is welltyped and satisfies its requirement.

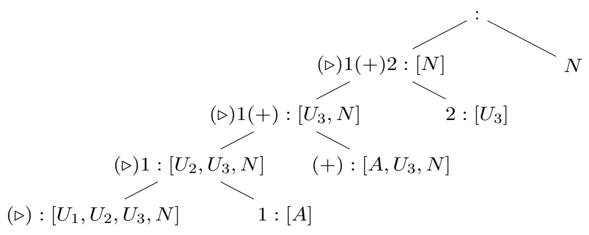
Example Elaborate the term

```
(|>) 1 (+) 2: Nat

-- equivalent to
(1 |> (+)) 2: Nat

-- in global context
(+): Nat -> Nat -> Nat
(+): String -> String -> String)

(|>) {A: Any} {P: A -> Any} (a: A) (f: all x: P x): P a
:=
    f a
```



:
$(\triangleright)1(+)2 : [N]$
N
$(\triangleright)1(+) : [U_3, N]$
$2 : [U_3]$
$(\triangleright)1 : [U_2, U_3, N]$
$(+) : [A, U_3, N]$
$(\triangleright) : [U_1, U_2, U_3, N]$
$1 : [A]$

We start with the elaboration of $(\triangleright)1(+)2$ with the signature requirement $[N]$ and the required type N.

- The term is an application with the function term $(\triangleright)1(+)$ and the argument 2. We start the elaboration of the function term with the signature requirement $[U_3, N]$ and no required type.
- The term is again an application with the function term $(\triangleright)1$ and the argument $(+)$. We start the elaboration of the function term with the signature requirement $[U_2, U_3, N]$ and no required type.
- The term is again an application with the function term $(\triangleright)$ and the argument 1. We start the elaboration of the function term

with the signature requirement $[U_1, U_2, U_3, N]$ and no required type.

- The term $(\triangleright)$ is a global name with the signature $[I, I, A, [A, P], P]$. Unification with the required signature $[U_1, U_2, U_3, N]$ results in

$$\begin{array}{lcl} U_1 & := & A \\ U_2 & := & [A, U_3, N] \\ P & := & [U_3, N] \end{array}$$

  Because of the two implicit arguments the term is elaborated as $(\triangleright)AP$ with the metavariables A and P.

- Next the argument term 1 has to be elaborated with the required signature $[A]$ and the required type A. This elaboration gets stuck on A, because the elaborator cannot decide the number type.

- Next the argument term $(+)$ is tried with the required signature $[A, U_3, N]$. The global name is ambiguous, but the ambiguity can be resolved by the result type. The signature is $[N, N, N]$. Unification with the required signature results in

$$\begin{array}{lcl} A & := & N \\ U_3 & := & N \end{array}$$

- Now the complete term can be elaborated as

$$(\triangleright)N(N \to N)\,{}^?a(+)\,{}^?b$$

  leaving the two holes ${}^?a$ and ${}^?b$ for the remaining arguments. However by unification the metavariables ${}^?A$ and ${}^?U_3$ are instantiated by N and therefore the elaboration of the remaining arguments is unblocked and finally will succeed.

8 Bound Variables

The function type

$$\Pi x^A.R$$

has a bound variable x of type A and a result type R which might depend on x.

A bound variable has the following attributes.

1. Implicit: Let $f : \Pi x^A.R$ be a function and $f\,a$ an application. If the bound variable is implicit, then the compiler is instructed to infer the argument a from the context in case it is missing.

2. Dependent: A variable is dependent if the result type depends on it.

   Open point: What happens in

   $$\Pi P^{A \to \mathcal{U}} x^A.Px$$

   if $P = \lambda x^A.N$? Is x dependent?

3. Ghost: A ghost variable means that its value cannot be used in the runtime code. It can be

   - used in types
   - used as an argument to a function which expects a ghost argument
   - pattern matched on to make decisions if the result type of the decision is a proposition
   - used as an argument to any function whose return type is a propositon.

A term which is not a type (i.e. its type is not a sort) are propositional or non-propositional. A term is propositional if its type is a proposition. Otherwise it is non-propositional. Non-propositional terms represent potential runtime objects.

If a constructor for a proposition uses non-propositional bound variables, then the non-propositonal bound variable is a ghost variable. A pattern match uncovering it can use its value only as a ghost value. A decision cannot be made on pattern match unless the result type of the pattern match is a proposition.

9 Universes

9.1 Universes and Sorts

We have the following sorts or universes (sorts and universes are synonymous):

1. $\mathcal{P}$: Impredicative universe of propositions
2. $\mathcal{U}_i$: Predicative universes of types for $i \in \{0, 1, 2, \ldots\}$
3. $\mathcal{U}_\infty$: Top universe

and the builtin type $\mathcal{L}$ with the typing judgements which are treated as axioms:

$$\begin{array}{l} \mathcal{L} : \mathcal{U}_\infty \\ \mathcal{P} : \mathcal{U}_0 : \mathcal{U}_1 \ldots \end{array}$$

All sorts except the top sort $\mathcal{U}_\infty$ have types. Therefore it is possible to introduce type variables $X : s$ for all sorts except the top sort.

Furthermore since $\mathcal{L}$ has type $\mathcal{U}_\infty$ it is possible to introduce universe variables $u : \mathcal{L}$.

The type of a type is always a sort. The sort of a type defines the universe of the type. I.e. all types live in a universe. The typing judgement $T : s$ says that the type T lives in the universe s.

There is a subtle difference between the universe of a type and the universe of an object. An object has a type and its type lives in a universe. We say the an *object o lives in a universe s if its type T lives in the universe s*. I.e. the typing judgement $o : T : s$ must be valid. In other words o has type T and T has type s.

A type T can be regarded as a type. Then its type is a sort s with $T : s$ and it therefore lives in the universe s as a type.

However a type can be regarded as an object as well. Then there is the typing judgement $T : s_1 : s_2$. We say the object T lives in the universe s_2.

Some examples: The type $\mathbb{N}$ lives in the universe $\mathcal{U}_0$. The object 1 has type $\mathbb{N}$. Therefore the object 1 lives in the universe $\mathcal{U}_0$. The object $\mathbb{N}$ has type $\mathcal{U}_0$ which has the type $\mathcal{U}_1$. Therefore the object $\mathbb{N}$ lives in the universe $\mathcal{U}_1$. By the same reasoning the object $\mathcal{U}_0$ lives in the universe $\mathcal{U}_2$.

All types regarded as objects live in a universe higher than the types regarded as types.

9.2 Typing Rules

Typing rules for products:

1. Propositional Products:

$$\frac{\begin{array}{l}\Gamma \vdash A : s\\ \Gamma, x^A \vdash B : \mathcal{P}\\ A \neq \mathcal{L}\end{array}}{\Gamma \vdash \Pi x^A.B : \mathcal{P}}$$

2. Predicative Products:

$$\frac{\begin{array}{l}\Gamma \vdash A : \mathcal{U}_i\\ \Gamma, x^A \vdash B : \mathcal{U}_i\end{array}}{\Gamma \vdash \Pi x^A.B : \mathcal{U}_i}$$

3. Universe Products:

$$\frac{\begin{array}{l}\Gamma \vdash u : \mathcal{L}\\ \Gamma, u^{\mathcal{L}} \vdash B : s\\ A \neq \mathcal{L}\end{array}}{\Gamma \vdash \Pi u^{\mathcal{L}}.B : \mathcal{U}_\infty}$$

These rules guarantee that the type $\Pi u^{\mathcal{L}}.B$ of a universe polymorphic object cannot be used as an argument to a function.

Furthermore we have the cumulativity rule

$$\frac{\begin{array}{l}\Gamma \vdash t : T\\ \Gamma \vdash U : s\\ T \leq U\end{array}}{\Gamma \vdash t : U}$$

The cumulativity rule for predicative universes says that whenever $T : \mathcal{U}_i$ is valid, then $T : \mathcal{U}_j$ is valid for all $i \leq j$ since $\mathcal{U}_i \leq \mathcal{U}_j$ is valid. Together with the predicative product rule we get

$$\frac{\begin{array}{l}\Gamma \vdash A : \mathcal{U}_i\\ \Gamma, x^A \vdash B : \mathcal{U}_j\\ i \leq k\\ j \leq k\end{array}}{\Gamma \vdash \Pi x^A.B : \mathcal{U}_k}$$

or maybe more specific

$$\frac{\begin{array}{l}\Gamma \vdash A : \mathcal{U}_i\\ \Gamma, x^A \vdash B : \mathcal{U}_j\end{array}}{\Gamma \vdash \Pi x^A.B : \mathcal{U}_{\max(i,j)}}$$

9.3 Constraints

Universe variables are always implicit. The programmer usually ommits universe variables in the source code i.e. instead of $\mathcal{U}_u$ the source code contains only $\mathcal{U}$. The compiler generates for each sort $\mathcal{U}$ in the source code a metavariable $^?u : \mathcal{L}$ and interprets the sort $\mathcal{U}$ as $\mathcal{U}_{^?u}$.

There are the following two sources of constraints between universes.

1. Predicative type argument $T : \mathcal{U}_u$: If T is used as an actual argument e.g. in the expression fT, then the function f must have the type $\Pi X^{\mathcal{U}_v}.R$. This creates the constraint $u \le v$.
2. Predicative inductive type $\Pi \mathbf{x}^{\mathbf{A}}.\mathcal{U}_v$: Any instance of such an inductive type lives in the universe $\mathcal{U}_v$.

   This requires that all arguments of all contructors must not live in a higher universe. Reason: Pattern matches of an indutive type living in the universe $\mathcal{U}_v$ must not uncover any object living in a higher universe. In other words: *Only smaller (or equal) things can be used to create greater things.*

   I.e. any constructor argument living as an object in the universe $\mathcal{U}_u$ creates the constraint $u \le v$.

9.4 Predicative Inductive Types

In Alba new types can be created as inductive types. The simplest inductive types have neither parameter nor index arguments.

$$\begin{array}{lcl} B & := & [T^{\mathcal{U}_0} \mid T, T] \\ N & := & [T^{\mathcal{U}_0} \mid N, N \to N] \end{array}$$

The types of booleans and natural numbers live in the universe $\mathcal{U}_0$ because of the typing judgements $B : \mathcal{U}_0$ and $N : \mathcal{U}_0$. Objects of type boolean do not contain anything. Objects of type N might contain other natural numbers (second constructor). There is no possibility for such simple types to contain objects living in a higher universe than $\mathcal{U}_0$.

Polymorphic types like lists and pairs have type parameters. The type parameters can live in any universe. Therefore they need an universe level parameter and a type parameter whhich depents on the uinverse level.

$$\begin{array}{lcllll} L & := & [i^{\mathcal{L}}, A^{\mathcal{U}_i} & \mid T^{\mathcal{U}_i} & \mid T, A \to T \to T] \\ P & := & [i^{\mathcal{L}}, A^{\mathcal{U}_i}, B^{\mathcal{U}_i} & \mid T^{\mathcal{U}_i} & \mid A \to B \to T] \end{array}$$

Having that we can construct a list of natural numbers $L\,0\,N : \mathcal{U}_0$ living in the universe level 0. However it is possible to construct list of types $L\,(i+1)\,\mathcal{U}_i$ which might live in the universe level i. The types as types live in the universe level i, but the types as objects live in the universe level $i+1$. Therefore the whole list which contains objects from the universe level $i+1$ lives in the universe level $i+1$ as well.

A more complex type is the dependent list.

$$\text{DL} \;:=\; [i^{\mathcal{L}}, A^{\mathcal{U}_i}, P^{A \to \mathcal{U}_i} \quad \mid T^{LiA \to \mathcal{U}_i} \quad \mid T[], \Pi a^A \text{as}^{LiA}.Pa \to T\text{as} \to T(a :: \text{as})]$$

A dependent list might contain objects of type A, LiA, Pa and Tas. All these object live in the universe level i. Therefore the whole dependent list can live in the universe level i.

Furthermore we can construct hetergeneous lists.

$$\text{HL} \;:=\; [i^{\mathcal{L}} \quad \mid T^{L(i+1)\mathcal{U}_i \to \mathcal{U}_{i+1}} \quad \mid T[], \Pi A^{\mathcal{U}_i} \text{As}^{L(i+1)\mathcal{U}_i}.A \to T\text{As} \to T(A :: \text{As})]$$

A heterogeneous list contains objects A and As with the typing judgements $A : \mathcal{U}_i : \mathcal{U}_{i+1}$, $\text{As} : L(i+1)\mathcal{U}_i : \mathcal{U}_{i+1}$. These objects live in the universe level $i+1$. Therefore the whole heterogeneous list object must live in the universe level $i+1$ as well.

Here is a rather strange type

$$\text{Ex} \;:=\; [i^{\mathcal{L}}, P^{\mathcal{U}_i \to \mathcal{P}} \quad \mid T^{\mathcal{U}_{i+1}} \quad \mid \Pi X^{\mathcal{U}_i}.PX \to T]$$

9.5 No Universe Polymorphism

No universe polymorphis means that all user entered $\mathcal{U}$ are interpreted as $\mathcal{U}_0$. Therefore all types live as types at the universe level 0 and all objects which are not types live as well at the universe level 0. Types as objects live at the universe level 1.

This has an important consequence: *Type objects cannot be contained within other objects.* There is no list of types, no pair of types etc.

Of the above examples nearly all types are possible without universe polymorphism. Only heterogeneous lists HL and the strange type Ex are not possible without universe polymorphism.

Without universe polymorphism we are basically in the calculus of constructions with inductive types. All construction which require an infinite set of stratified universes with cumulativity are not possible.

10 Monads

10.1 Basics

A monadic type is a type constructor with at least one type argument.

$$M : \Pi\Gamma A^{\mathcal{U}}\Delta.\mathcal{U}$$

Type of a monadic bind operation:

$$\Pi A^{\mathcal{U}} B^{\mathcal{U}} \ldots . M\mathbf{a}A\mathbf{b} \to (A \to M\mathbf{c}B\mathbf{d}) \to M\mathbf{e}B\mathbf{f}$$

10.2 Recursion

Form of monadic types where recursion is possible:

$$\begin{array}{l} S \to R(DA)S \\ S \to (DA \to S \to Z) \to Z \\ S_1 \to \ldots \to S_n \to ((DA) \to S_1 \to \ldots \to S_n \to Z) \to Z \end{array}$$

where D is an inductive type constructor with two constructors (one contains A, the other not) and R is an inductive record type constructor (only one constructor.

Usually D is either `Maybe` or `Result` and R is a tuple.

```
S -> (Maybe A, S)

S -> (Maybe A -> S -> Z) -> Z

S1 -> S2 -> ... ->  (Maybe A -> S1 -> S2 -> ... -> Z) -> Z
```

10.3 Code Examples

State with Success

```
State (A S: Any): Any :=
    S -> (Maybe A, S)

return {A S: Any} (a: A): State A S :=
    \ s := (just a, s)

(>>=) {A B S: Any} (m: State A S) (f: A -> State B S): State B S :=
    \ s1 :=
        match m s1 case
            \ (just a, s2) :=  (f a) s2     -- potential recursive call!
            \ (empty,  s2) :=  (empty, s2)
```

Continuation with State and Success

```
Cont (S A R: Any): Any :=
    S -> ((Maybe A, S) -> R) -> R

return {S A R: Any} (a: A): Cont S A R :=
    \ s k := k s (just a)

(>>=) {S A B R: Any} (m: Cont S A R) (f: A -> Cont S B R): Cont S B R :=
    \ s kb :=
        m s1
          (case
            \ (just a, s2)  := f a s2 kb        -- potential recursive call
            \ (empty,  s2)  := kb (empty, s2))

run {S A Any} (s: S) (m: Cont S (Maybe A) (Maybe A)): Maybe A :=
    m s (\ x := x)
```

State

```
State (A S: Any): Any :=
    S -> (A, S)

return {A S: Any} (a: A): State A S :=
    \ s := (a, s)

(>>=) {A B S: Any} (m: State A S) (f: A -> State B S): State B S :=
    \ s :=
        match m s case
            \ (a, s2)  := f a s2

map {A B S: Any} (f: A -> B) (m: State A S): State B S :=
        -- 'State' is covariant in 'A'
    \ s :=
        match m s case
            \ (a, s2) := (f a, s2)

mapS {A S1 S2: Any} (f: S1 -> S2) (g: S2 -> S1) (m: State A S1): State A S2 :=
        -- 'State' is not variant in 'S'
    \ (s2: S2) :=
        match m (g s2)  case
            \ (a, s1) := (a, f s1)
```

Continuation

```
Cont (A R: Any): Any :=
    (A -> R) -> R

return {A R: Any} (a: A): Cont A R :=
    \ (k: A -> R) := k a

(>>=) {A B R: Any} (m: Cont A R) (f: A -> Cont B R): Cont B R :=
    \ (k: B -> R) :=
        m (\ a := f a k)
```

```
map {A B R: Any} (f: A -> B) (m: Cont A R): Cont B R :=
        -- 'Cont' is covariant in 'A'
    \ (k: B -> R) :=
        m (\ a := k (f a))

mapR {A R1 R2: Any} (f: R2 -> R1) (m: Cont A R1): Cont A R2 :=
        -- 'Cont' is contravariant in 'R'
    \ (k: A -> R2) :=
        m (\ a :=  f (k a))


run {A: Any} (m: Cont A A): A :=
    m (\ x := x)
```

State with Continuation

```
SCont (S A R: Any): Any :=
    S -> (A -> S -> R) -> R
    -- or equivalent
    S -> ((A, S) -> R) -> R

return {S A R: Any} (a: A): SCont S A R :=
    \ s k := k a s

(>>=) {S A B R: Any} (m: SCont S A R) (f: A -> SCont S B R): SCont S B R :=
    \  s1 kb :=
        m s1 (\ a s2 := f a s2 kb)

map {S A B R: Any} (f: A -> B) (m: SCont S A R): SCont S B R :=
        -- 'Cont' is covariant in 'A'
    \ s1 kb
        m s1 (\ a s2 := kb (f a) s2)
```

Function

```
Fun (X A: Any): Any :=
    X -> A

return {X A: Any} (a: A): Fun X A :=
    \ _ := a

(>>=) {X A B: Any} (m: Fun X A) (f: A -> Fun X B): Fun X B :=
    \ x :=
        f (m x)

map {X A B: Any} (f: A -> B) (m: Fun X A): Fun X B :=
        -- 'Fun' is covariant in the result type
    \ x := f (m x)

mapX {X1 X2 A: Any} (f: X2 -> X1) (m: Fun X1 A): Fun X2 A :=
        -- 'Fun' is contravariant in the argument type
    \ (x: X2) :=
        m (f x)
```