UNIVERSITÀ DEGLI STUDI DI UDINE

Dipartimento di Scienze Matematiche, Informatiche e Fisiche

Corso di Laurea Magistrale in Matematica

Tesi di Laurea

A PROTOTYPE IMPLEMENTATION OF CUBICAL TYPE THEORY WITH EXPLICIT PARTIAL AND RESTRICTION TYPES

Relatori:
Prof. FABIO ALESSI
Prof. FURIO HONSELL

Correlatore:
Prof. THIERRY COQUAND

Laureando:
MATTIA FURLAN

ANNO ACCADEMICO 2021-2022

Introduction

1   Intuitionistic type theory

The topic of this thesis is type theory, a branch of mathematics and computer science that traces its origins back to Russell's theory of types in the early 1900's, developed to be a solid foundation for mathematics which avoided the paradoxes. More precisely, we focus on some recent developments of intuitionistic type theory [ML75], introduced by Per Martin-Löf as a formal system meant to be a foundation for constructive mathematics. It is based on the Brouwer-Heyting-Kolmogorov interpretation of intuitionistic logic, which explains the meaning of the logical connectives in terms of computations, and on the so called Curry-Howard correspondence (or *propositions-as-types* [How80]), whose motto is «a proposition is the type of its proofs». The main feature of intuitionistic type theory is the fact that it can also be seen as a functional programming language, based on the typed lambda calculus, with a type system so rich that indeed it can interpret (higher order) logic. It should be noted that the type system of intuitionistic type theory is much more evolved than the ones found in ordinary programming languages, as they usually lack *dependent types*.[1] The problem with dependent types is that to keep type checking decidable, the language must be strongly normalizing, that is every computation must halt, and this implies the non-Turing completeness of the language.[2] Having a very expressive type system allows to state formally (i.e. inside the type system itself) a specification of a program, so that developing a program to fulfil its task and proving its correctness can be done at the same time. Although the non-Turing completeness be can a problem, it can be argued that dependent types remedy a bit, at least for pratical purposes,

---

[1]Dependent types are types that depend on a value: a standard example is $\texttt{vector}(n)$, which is the type of vectors of length n. The dependent type is $\texttt{vector}$ itself, not $\texttt{vector}(n)$.

[2]The normalizing property is strongly connected to the consistency of the underlying (intuitionistic) logic: a non terminating computation would be interpreted logically as a proof of falsum.

since often it's possible to avoid blocking the computation: an example is the function `head` which returns the first element of a list, as it must handle the empty list case somehow, usually in some artificial way, e.g. by reporting an I/O error and halting; with dependent types it can be solved easily, by passing not only the list but also a proof (that is, a term witnessing...) that the length of the list is not zero;[3] by doing this, the empty list case is automatically excluded. The propositions-as-types paradigm, which may be more appropriately called proofs-as-programs, is actually practically implemented in some proof assistants such as Coq and Agda, as one can extract automatically a concrete program, written for example in Haskell, that computes the function corresponding to a given proof. To make a possibly trivial example, from a Coq formalization of Euclid's proof of the infinity of prime numbers,[4] one can extract an Haskell program that, given a natural number, outputs a prime greater than that.

Turning back to the foundations, one of the most important features of intuitionistic type theory is the treatment of equality, of which there exists two main notions. The first is the *judgmental* or *definitional* equality, which is a relation between the terms of the theory but is external to it, and expresses the fact that two terms are convertible to each other, by simplifying the terms and unfolding the definitions.[5] In this thesis we will always write $a \equiv b$ to mean that a and b are judgmentally equal. The second notion of equality, often called *definitional* equality, is more subtle and is internal to the theory itself: for each pair of terms a and b of the same type A, one can form the type $\text{Id}_A(a, b)$, often written more simply as $a =_A b$; an element p of type $a =_A b$ is then a witness that a and b are (propositionally) equal, but this does not imply that $a \equiv b$.[6] Propositional equality can be seen as an inductive relation, but is more simply modelled by an introduction rule asserting reflexivity of equality, and by an elimination rule (also called J-rule) which allows to prove a predicate about two equal elements a and b of type A, eventually depending also on the proof that they are equal, by proving only the simpler case where the two ele-

---

[3]With the notation that we will introduce in chapter 1, the signature of the function, for lists of naturals, would be $\text{head} : [n : \text{N}] \, (n > 0) \rightarrow \text{vector}_\text{N}(n) \rightarrow \text{N}$.

[4]I think it's worth to spend a few words to say that albeit the proof seems not to be really constructive, since the law of the excluded middle (LEM) is used to test whether the number $N = p_1 \cdots p_n + 1$ is prime or not, that instance of LEM is intuitionistically valid, as the primeness property is obviously decidable.

[5]The strong normalization property implies the decidability of the judgmental equality.

[6]We say that the equality is *intensional*; it's possible to add the equality reflection rule (i.e. propositional equality implies judgmental equality), but it would destroy the decidability of the type-checking. Note that this implies not being able to even (computationally) recognize a valid proof.

ments are judgementally the same and the equality is given by reflexivity. This seemingly simple treatment is the bridge to the homotopical interpretation of type theory, as it gives rise to the so called (weak) ∞-groupoid structure,[7] due to the possibility of iterating the equality type: given $a, b : A$ and two proofs $p, q : a =_A b$, due to intensionality it shall not always hold that $p \equiv q$, and indeed it makes sense to form the type $p =_{(a=_A b)} q$ of identifications between proofs of equality between a and b; then we can continue: given two different identifications $\alpha, \beta : p =_{(a=_A b)} q$, we can form the type $\alpha =_{(p=_{(a=_A b)} q)} \beta$, and so on. The aforementioned structure of ∞-groupoid is given by the groupoid operations (identity, inverse, composition) induced by the elimination rule for propositional equality. The crucial fact is that this structure does not collapse, as one may intuitively expect that there is only one way to prove an equality, a principle known as *uniqueness of identity proofs* (UIP); [HS96] first proved that UIP does not hold in general, by exploiting the so called *groupoid model*, where types are interpreted by groupoids, the elements of the type are the elements of the corresponding groupoid and equality is modelled by arrows in that groupoid; a counterexample to UIP is then a groupoid with more than an arrow between two elements.

2   The homotopical interpretation

The groupoid model suggests that types should not be treated just as sets of points, and indeed an interpretation should exploit the ∞-groupoid structure. In the homotopical interpretation types are seen as topological spaces and identities between points as topological paths, i.e. continuos functions from the unit interval to the type/space. As I said in the previous section, using the J-rule it is possible to define the operations of *path inversion*, $(a =_A b) \to (b =_A a)$, $p \mapsto p^{-1}$, and *path composition*, $(a =_A b) \to (b =_A c) \to (a =_A c)$, $p, q \mapsto p \cdot q$, which satisfy the groupoid laws, but only up to (higher order) propositional equality,[8] e.g. for $p : a =_A b$, $p \cdot p^{-1}$ need not be judgmentally equal to $\mathtt{refl}_A(a)$, but it can be proved that $p \cdot p^{-1} =_{(a=_A a)} \mathtt{refl}_A(a)$. This is easily explained topologically: the concatenation of a path with its inverse is almost never pointwise equal to the trivial path at its beginning point, but is always homotopic to it (note that an homotopy is nothing else than path between two paths). The connection between homotopy theory, higher-dimensional category theory and type theory arose around 2006 by the works of Awodey and Warren [AW09] and of Voevodsky [Voe06], resulting most notably

---

[7]A groupoid is a category where every arrow is invertible, or equivalently is a group where the operation is a partial function.

[8]This is why the structure is usually called *weak* ∞-groupoid.

in Voevodsky's model of type theory in Kan simplicial sets [KL21].

3   Univalence and cubical type theory

The simplicial model of Voevodsky satisfies the so called *univalence property* (now called univalence axiom), which is about equality of types (i.e. paths inside the universe of types), as it says informally that equivalent types are actually (propositionally) equal. This is important because it adds more structure to the type-theoretic universe and because it makes possible to identify equivalent structures not in the informal way that it's usually done in mathematics, but explicitly using the language of type theory.[9] To make an example,[10] suppose we define two types `Nat` and `Nat'` inductively with the constructors `Z : Nat`, `S : Nat → Nat` and `zero : Nat'`, `suc : Nat' → Nat'`; these two types are obviously equivalent in some intuitive sense, and suppose for now that this equivalence is formally witnessed by a suitable term `e : Equiv(Nat, Nat')`; the axiom of univalence allows us to obtain a term `ua(e)` : $\text{Nat} =_{\mathcal{U}} \text{Nat'}$, which means we can *transport* theorems (i.e. terms) from `Nat` to `Nat'` and viceversa.[11] E.g. let us suppose that we have derived $[n, m : \text{Nat}]\,(n + m =_{\text{Nat}} m + n)$; then we get 'for free' also $[n, m : \text{Nat'}]\,(n +' m =_{\text{Nat'}} m +' n)$. The univalence axiom is so called because it is not derivable from the other rules of intuitionistic type theory (and indeed in does not hold in the groupoid model). In general it is possible to add axioms to intuitionistic type theory, or to be more precise, to postulate the existence of terms with a given type (i.e. `ex-middle` : $[A : \text{U}](A + \neg A)$ for the excluded middle). Note that although the excluded middle is consistent with intuitionistic type theory,[12] adding it makes no more possible to extract a program from a classical proof. Similarly, the problem with univalence is that it has to be postulated as an axiom, so proofs using it lose their computational content; the same happens with function extensionality, which is consistent with the theory (although it is implied

---

[9]Moreover, univalence implies that every construction in the type theory is homotopy invariant; this, along with *higher inductive types* (see [Uni13] chapter 6), allows to develop a synthetic approach to homotopy theory (see [Uni13] chapter 8).

[10]For a less trivial one, see [Uni13] 2.14.

[11]The transport is a basic principle of homotopy type theory, derivable from the J-rule, which is nothing else than Leibniz's principle of indiscernibles, i.e. for a type family $P : A → \text{U}$ and a path $p : a =_A b$, we have $\text{transport}^P(p, -) : P(a) → P(b)$.

[12]But not with the univalence axiom, see [Uni13] 3.2.7. Indeed, the 'right' formulation of LEM is not the naive one we presented, but $[A : \text{U}]\,\text{isProp}\,A → A + \neg A$, which is consistent, as it holds in the simplicial model. The predicate `isProp` classifies the so called *mere propositions*, or (-1)-types, which allow to develop classical logic in univalent foundations (see chapter 3 of [Uni13]).

by univalence). We have to be precise by what we mean with the possibility of extracting a program from a proof, which known as the *canonicity property*: we mean that every closed term that has the type of natural numbers reduces to a numeral in a computable way.[13] It is now clear that finding a constructive justification of univalence is (or, has been) a central topic in type theory. Cubical type theory, recently introduced in [CCHM15], allows to prove univalence while still keeping the computational content of the theory.

The key difference between CTT (cubical type theory) and HOTT (homotopy type theory) is the treatment of equality: whereas in the latter equality is inductively defined, in the former it is modelled in a more primitive, topological manner, i.e. as functions (paths) from a formal unit interval $\mathbb{I}$ into a specified type. To make things more concrete, suppose that a and b have type A, then a witness of equality between a and b is a function p, called path, of type $\mathbb{I} \to A$, such that $p\,0 \equiv a_1$ and $p\,1 \equiv a_2$, where 0 and 1 are the formal representation of zero and one in the unit interval. We use the notation $\mathtt{Path}_A(a, b)$ for paths between elements a, b of type A. The key point is that $\mathbb{I}$ is not a type, although it is possible to abstract over an element of $\mathbb{I}$ and to apply a path to elements of $\mathbb{I}$. It should be kept clear from the beginning that path types do not correspond exactly to identity types in HOTT, as for example the elimination rule for path types holds only up to homotopy and not judgmentally.[14]

An enlightening example with path types is function extensionality, which can be proved in a surprisingly simple way: suppose that we have a (dependent) path $p : [x : A]\,\mathtt{Path}_B(f\,x, g\,x)$ between two functions $f, g : A \to B$. Then the term

$$\beta \equiv [i : \mathbb{I}][x : A]\,p\,x\,i$$

is a path in $A \to B$ between f and g, because

$$\beta\,0 \equiv [x : A]\,p\,x\,0 \equiv [x : A]\,f\,x \equiv f$$

and similarly $\beta\,1 \equiv g$. Note that what we did is basically just swapping $[i : \mathbb{I}]$ with $[a : A]$ to get from p to β.

Equalities can be visualized as paths between the two endpoints, whose line structure can be thought as a degenerate '1-dimensional' hypercube, described by the formula $i = 0 \lor i = 1$.

---

[13]This is how the idea of a theory with computational content is formalized: with a suitable coding every other term of the theory can be thought as a function $\mathbb{N} \to \mathbb{N}$.

[14]Anyway, it is possible to define a data type $\mathtt{Id}$ in CTT from path types so that they correspond exactly to HOTT's identity types, making it possible to interpret HOTT inside CTT (see [CCHM15] 9.1, due to A. Swan).

$$p \ \mathtt{0} \xrightarrow{\quad p \quad} p \ \mathtt{1}$$

Homotopies, that is paths of paths, can be viewed as squares, or in other words 2-dimensional hypercubes. In the figure below, α is an homotopy between the paths α $\mathtt{0}$ (or $[j : \mathbb{I}] \ \alpha \ \mathtt{0} \ j$, with endpoints $\alpha \ \mathtt{0} \ \mathtt{0}$ and $\alpha \ \mathtt{0} \ \mathtt{1}$) and α $\mathtt{1}$ (or $[j : \mathbb{I}] \ \alpha \ \mathtt{1} \ j$, with endpoints $\alpha \ \mathtt{1} \ \mathtt{0}$ and $\alpha \ \mathtt{1} \ \mathtt{1}$). We explicitly represent the axes for clarity; note that the structure can be described by the formula $i = \mathtt{0} \vee i = \mathtt{1} \vee j = \mathtt{0} \vee j = \mathtt{1}$.

$$
\begin{array}{ccc}
\alpha \ \mathtt{0} \ \mathtt{1} & \xrightarrow{[i:\mathbb{I}] \ \alpha \ i \ \mathtt{1}} & \alpha \ \mathtt{1} \ \mathtt{1} \\
\big\uparrow{\scriptstyle \alpha \ \mathtt{0}} & & \big\uparrow{\scriptstyle \alpha \ \mathtt{1}} \\
\alpha \ \mathtt{0} \ \mathtt{0} & \xrightarrow[{[i:\mathbb{I}] \ \alpha \ i \ \mathtt{0}}]{} & \alpha \ \mathtt{1} \ \mathtt{0}
\end{array}
$$

Disposing only of the interval does not bring us very far: the real power of CTT is the compositional structure (or *Kan structure*) that each type is equipped with. Indeed, the syntax of CTT is modelled in a way that reflects a model of type theory in cubical sets [Hub16], where each type has to satisfy the so-called *uniform Kan condition*, which is a generalization of the homotopy-theoretic Kan condition which informally says that for each n-dimensional hypercube missing one 'face' there is a complete hypercube that extends it. For example, suppose we have a term of type A defined on all the sides of a square except the top (we say that the type of that term is a *partial type*), that is on the *extent* $i = \mathtt{0} \vee i = \mathtt{1} \vee j = \mathtt{0}$; then we can obtain a term defined on whole square (the *filler*) which extends the partial one. One easy application of composition is the concatenation of two paths, which we show in section 1.2.2. Another application is the derivation of the *transport* function, which allows to transfer properties along paths, i.e. paths $\mathtt{Path}_A(x, y)$ induce functions $P \ x \to P \ y$ for each P. Composition is defined inductively on the type family, which requires great care especially for the universe of types.

We use the notation $[\varphi]A$ for the type of partial elements of A defined on the extent φ and $[\psi \mapsto t]A$ for the type of (total) elements of A which agree with t on ψ. We talk respectively of partial and restriction types; the latter is used for example to assert that the composition agrees with the given partial term under the incomplete hypercube, or in other words that the composition extends the partial term. We remark that although the concept of partial type

is very important for the presentation, it remains implicit in [CCHM15] and in the `cubicaltt` implementation.[15]

4 Contributions of the thesis

In this thesis we investigate and implement a modification of cubical type theory with explicit partial and restriction types, that is a type theory with:

- Basic type formers of intuitionistic type theory: empty and unit types, naturals, (dependent) products, (dependent) sums, coproducts.[16]

- A universe of types U, such that U : U.[17]

- A formal unit interval $\mathbb{I}$ with endpoints 0 and 1. Although not being considered a type, it is possible to define and apply functions with domain $\mathbb{I}$, i.e. paths.

- Formulas built from atomic ones of the kind $(i = 0)$, $(i = 1)$ and $(i = j)$, where $i, j : \mathbb{I}$ are called dimension names, using the operations $\wedge$ and $\vee$.[18] Formulas are used only in partial and restriction types and in compositions; note that a conjunctive formula describes a face of a hypercube and a formula in disjunctive normal form describes a union of faces, that is a sub-polyhedra of an hypercube.

- Partial types of the form $[\varphi]A$, where A is a type and φ is a formula in disjunctive normal form. The meaning of $[\varphi]A$ is that of a type defined only of the extent φ. To define partial elements, we use *systems*, written as $[\psi_1 \mapsto t_1, \ldots, \psi_n \mapsto t_n]$, where φ is the disjunction of the ψ's, and t_i is the term associated to the face described by the conjunctive formula ψ_i (note that we have to ensure that the terms agree whenever the faces overlap, that is on the edges, described by the conjunction of the formulas of the two faces).

---

[15]https://github.com/mortberg/cubicaltt

[16]Unit, sums and products is all that is needed to define non-inductive ADTs (algebraic data types). The empty and unit types will actually be derived types, the latter defined using a partial version of the naturals.

[17]The fact that the universe is an element of itself makes the theory inconsistent (Girard's paradox); we chose to do so because it really simplifies the implementation, otherwise one must introduce an infinite hierarchy of universes, and then handle subtyping, which is highly non-trivial to implement. We must then be careful that what we input is not inconsistent.

[18]Formulas of the form $(i = j)$ are needed for higher inductive types, which we do not treat in this thesis. We decided to handle this kind of formulas anyway, in case of future upgrades of the implementation.

- Restriction types of the form $[\psi_1 \mapsto t_1, \ldots, \psi_n \mapsto t_n]A$, where A is a type, and for each i, ψ_i is a conjunctive formula and t_i is a term of type $[\psi_i]A$. The meaning of $[\psi_1 \mapsto t_1, \ldots, \psi_n \mapsto t_n]A$ is that of the type of elements of A that on each ψ_i are convertible to the given t_i. Analogously to systems, terms on overlapping formulas must be checked.

The handling of formulas requires special attention during type-checking and evaluation, managing individually the special cases of partial and restriction types. In particular, during type-checking we must pay attention to 'propagate' partial and restriction types (e.g. from $f : [\psi \mapsto g](A \to B)$ and $a : A$, the application $f\ a$ shall have type $[\psi \mapsto g\ a]B$; inferring just type B results in a loss of information); during evaluation, some terms which we call *neutral* (i.e. terms for which we don't have a definition at that moment of evaluation) have to be annotated with their type, and this has to reflect the typing rules. The need for annotation is that a neutral term with restriction type may have a true formula, which means that the whole term shall be reduced to the term associated with the true formula. Several choices about the exact rules to be admitted have been made and are presented in chapter 1.

After introducing the theory in the first chapter, in the second one we describe a category-theoretic semantics for this type theory using presheaf models; in particular we explain how to model partial and restriction types, also describing the connections with the subobject classifier available in any presheaf category, as is the category of *cubical sets*. We do not deal with the full semantics needed to handle univalence (i.e. Kan cubical sets [Hub16]).

Finally in chapter 3 we describe the details of the Haskell implementation of this type theory, using most notably a bidirectional type-checker and the normalization by evaluation technique. The result is a program which, given in input a file or a list of declarations (i.e. of the form `name : type`) and definitions (i.e. of the form `name : type = def`), type-checks it and then starts a REPL loop where it is possible to input new declarations and definitions and to infer the type of the given terms.

We remark that the focus of this thesis is the treatment of explicit partial and restriction types, and the purpose is not a full implementation of cubical type theory; indeed, in the implementation we have not developed composition for the universe of types, so that univalence is not yet provable; moreover, an efficient and smart implementation of composition is still to be engineered. This may be a direction for future work.

Contents

Chapter 1

Theory

1.1 The basic type theory

We first present the basic intuitionistic type theory underlying our implementation, which we will then extend with the tools of cubical type theory (interval, systems, partial and restriction types).

First of all we define the syntax of the terms of the basic theory.[1] It's worth pointing out that what we call terms is different (more general) than terms usually intended in non-dependently typed λ-calculi, as there one can distinguish between types and terms (that is, the syntactic objects that can be assigned a type) regardless of the type-checking. Instead in dependently typed λ-calculi one cannot do this distinction, but can only define what is a type in a context (A is a type in context Γ if $\Gamma \vdash A : \mathtt{U}$) and what is a term of a given type in a context (a is a term of type A in context Γ if A is a type in context Γ and $\Gamma \vdash a : A$). Therefore, here by term we mean a syntactic object of the language, which can also be ill-typed, and if not so, can be a term in the less general meaning or a type.

Definition 1.1.1 (Terms)**.** *Terms are inductively defined by:*

1. *Variables $x, y, z, \ldots$*

2. *A universe of types, $\mathtt{U}$.*

3. *A 'let-definition' mechanism of the form $[x : A = M]M'$, where A, M and M' are terms; it means 'let x be of type A and judgmentally equal*

---

[1]This syntax is not exactly the same as the one used in the source files for the implementation (concrete syntax), but is very similar.

to M inside M'''.

4. *For Π-types:*

    (a) *Application $(M_1\, M_2)$, where M_1 and M_2 are terms.*

    (b) *Abstraction $[x : A]M$, where x is a variable and A, M are terms. It is used both for the λ-abstraction for terms (i.e. $\lambda x : A.M$) and for dependent product types (i.e. $\Pi_{x:A}.M$). The context in which a term of that form is declared clears the ambiguity. When x does not appear in M, we shall simply write $A \to M$.*

5. *For Σ-types:*

    (a) *Projections $M.1$ and $M.2$, where M is a term.[2]*

    (b) *Pair constructor (M_1, M_2), where M_1 and M_2 are terms.*

    (c) *Dependent sum types $\langle x : A \rangle M$, where x is a variable and A, M are terms. When x does not appear in M, we shall simply write $A * M$.*

6. *For coproduct types:*

    (a) *Left and right injections `inl` M and `inr` M, where M is a term.*

    (b) *Coproduct types $A + B$, where A and B are terms.*

    (c) *Elimination rule for coproduct types: given terms A (type family), M_l (left case), M_r (right case) and M, `split` $A\ M_l\ M_r\ M$ is a term.*

7. *The type of natural numbers, `N`, along with the constant `Z` and the unary constructor `S`. Moreover, for terms A (type family), M_0 (base case), M_s (successor case) and N, `ind` $A\ M_0\ M_s\ N$ is a term, which represents the inductor principle (i.e. the elimination rule) for `N`.*

For brevity we do not include the empty type, as it can be defined using the impredicativity of the universe,[3] and the unit type since it will be possible to derive it later too using cubical concepts (section 1.2).

The syntax can be presented more succinctly with a formal grammar, as follows, where x is a generic variable and A, B, M (eventually subscripted) are

---

[2]The two projections allows to define the elimination rule for Σ-types; otherwise one can take the elimination rule as a primitive, and then define the two projections.

[3]The empty type, logically corresponding to falsum, is defined as `empty` $: U = [X : U]X$. The eliminator principle for the empty type is nothing else that the 'ex falso quodlibet' law (false implies anything), which is straightforward to derive: `ind-empty` $: [A : U]$ `empty` $\to A = [A : U][b : $ `empty` $]\ b\ A$.

meta-variables for terms.

$$A, B, M ::= \mathtt{U} \mid x$$

$$\mid [x : A = N]M \qquad\qquad\qquad \text{(Let-definition)}$$
$$\mid M_1\ M_2 \mid [x : A]M \qquad\qquad\qquad \text{(\Pi-types)}$$
$$\mid \langle x : A \rangle M \mid (M_1, M_2) \mid M.1 \mid M.2 \qquad \text{(\Sigma-types)}$$
$$\mid A + B \mid \mathtt{inl}\ M \mid \mathtt{inr}\ M \mid \mathtt{split}\ A\ M_l\ M_r\ M \quad \text{(Coproducts)}$$
$$\mid \mathtt{N} \mid \mathtt{0} \mid \mathtt{S} \mid \mathtt{ind}\ A\ M_0\ M_s\ M \qquad\qquad \text{(Naturals)}$$

It is important, especially for the implementation, to isolate a subset of the terms, known as *neutral terms* and denoted with K, which is composed by terms in β-normal form, i.e. terms that do not contain any redex, that is eliminators (function application, projections .1 and .2, $\mathtt{split}$ and $\mathtt{ind}$) applied to constructors. Neutral terms are: variables, all the terms whose main 'term-constructor' is not an eliminator,[4] and eliminators applied to a neutral terms only. Therefore, the grammar for neutral terms is the following:

$$A, B, M ::= \mathtt{U} \mid K$$

$$\mid [x : A]M \qquad\qquad\qquad\qquad \text{(\Pi-types)}$$
$$\mid \langle x : A \rangle M \mid (M_1, M_2) \qquad\qquad \text{(\Sigma-types)}$$
$$\mid A + B \mid \mathtt{inl}\ M \mid \mathtt{inr}\ M \qquad \text{(Coproducts)}$$
$$\mid \mathtt{N} \mid \mathtt{0} \mid \mathtt{S} \mid \mathtt{ind}\ A\ M_0\ M_s\ M \quad \text{(Naturals)}$$

$$K ::= x \mid K\ M \mid \mathtt{ind}\ A\ M_0\ M_s\ K$$
$$\mid K.1 \mid K.2 \mid \mathtt{split}\ A\ M_l\ M_r\ K \qquad \text{(Neutrals)}$$

We also consider values, i.e. the results of the evaluation of terms (evaluation will be discussed in 1.1.4), as neutral terms, excluding the abstractions $[x : A]B$ and $\langle x : A \rangle B$, extended with a closure operator of the form $\mathsf{closure}(M, \Gamma)$, where M is an (unevaluated) term of the form $[x : A]B$ or $\langle x : A \rangle B$ and the closure environment is specified by the context Γ. We write meta-variables representing values in bold, i.e. $\mathbf{v}$, to distinguish them from terms.

---

[4]By the main term-constructor I mean the root of the tree associated to a term. For example, the term $([x : \mathtt{N}]\ \mathtt{plus}\ x\ (\mathtt{S}\ \mathtt{Z}))\ \mathtt{Z}$ has function application as its main term-constructor.

The only change we make in the abstract syntax is that we annotate neutral values with their (evaluated) type, written as $\mathbf{v^t}$.[5] Neutral values get annotated during evaluation, so the EVAL function needs the context to lookup the types; instead of having to give both the context and the environment, we use only the context, adding values bindings to it. As will become clear later, type annotations are required in order to simplify cubical expressions: for example if the variable x has type $[i : \mathbb{I}][(i = 0) \mapsto a, (i = 1) \mapsto b]A$, then $x\ i$ shall reduce to a under the binding $i \mapsto 0$.

Having defined terms and values, we give the rules to form a whole 'program', that is a list of declarations, definitions and examples,[6] which we formalize in the following grammar, where A, B, M are meta-variables for terms. Note that this part, albeit being practically important, is formally superfluous, since a sequence of definitions and declarations may be replaced by a single term using the abstractors $[x : A]$ and $[x : A = M]$;

$$
\begin{aligned}
Program &::= [TopLevel] \\
TopLevel &::= Definition \mid Declaration \mid Example \\
Definition &::= x : A = M \\
Declaration &::= x : A \\
Example &::= M
\end{aligned}
$$

1.1.1 Syntactical constraints

It's well known that one has to be careful with variables when performing reduction of terms, because there might occur a 'capture of variable', even in seemingly 'safe' cases, as this example shows (from [vBJ77]): suppose that we have declared

$$
\begin{aligned}
x &: A & b &: A \to A \to A \\
y &: A & u &: (A \to A \to A) \to A \to A \to A \\
z &: A \to A \to A \quad & a &: (A \to A) \to A \to A
\end{aligned}
$$

Then the term $([u : A]\ u\ (u\ b))([z, y, x : A]\ a\ (z\ x)\ y)$ is well typed and the reduction chain starts as follows:

$$
\begin{aligned}
&([u : A]\ u\ (u\ b))([z, y, x : A]\ a\ (z\ x)\ y) \\
&\to_\beta ([z, y, x : A]\ a\ (z\ x)\ y)(([z, y, x : A]\ a\ (z\ x)\ y)\ b)
\end{aligned}
$$

---

[5]In the implementation, we use a field `Neutral Value Value` for annotated values.

[6]An 'example' is a term whose type will be inferred, if well-typed.

$$\to_\beta ([z, y, x : A] \ a \ (z \ x) \ y)([y, x : A] \ a \ (b \ x) \ y)$$
$$\to_\beta [y, x : A] \ a \ (([y, x : A] \ a \ (b \ x) \ y) \ x) \ y$$

At this point it's necessary to rename one of the bounded variables x, otherwise the term would reduce to

$$[y, x : A] \ a \ ([x : A] \ a \ (b \ x) \ x)) \ y$$

which is not correct. The point of this example is that the capture of variable problem might happen even if in the starting term each binding variable is declared only once and the variables used in the sub-terms forming function applications are all different.

This problem must be handled with great care if one implements β-reduction syntactically, which is moreover inefficient; instead, we use the normalization-by-evaluation technique (see section 1.1.4), which defers actual substitutions until the body of a function is free of abstractions, so that the when the substitution is made, only the variables that were originally bound in the body of the function will be substituted by the corresponding terms.

There is a second subtlety that we must address, that is the eventual shadowing of names. Consider the following context:

$$x : \mathtt{N}$$
$$y : \mathtt{N} = x$$
$$x : \mathtt{N} \to \mathtt{N}$$

If this has to be accepted, the program must handle somehow the fact that y of type $\mathtt{N}$ seems to get assigned an object of type $\mathtt{N} \to \mathtt{N}$; for example, Agda allows this by using indexes to discriminate between variables with the same name,[7] but this approach can still be confusing to the user. Instead, we decided to forbid name shadowing, so that the implementation is simpler. Note that the name shadowing check is done every time a context is expanded, i.e. with each new 'top-level' declaration or definition and when type-checking an abstraction. It has to made clear that the following context is valid, as the two variables x are not in the same scope:

$$fun1 : \mathtt{N} \to \mathtt{N} = [x : \mathtt{N}] \ x$$
$$fun2 : \mathtt{N} \to \mathtt{N} = [x : \mathtt{N}] \ \mathtt{S} \ x$$

---

[7]A technique called *namespaced De Bruijn indices*, see `https://www.haskellforall.com/2021/08/namespaced-de-bruijn-indices.html`.

1.1.2 Contexts

To state the typing rules of the theory, we need a formal notion of context, which is thought as a list of declarations and definitions; moreover, we augment it with value bindings of the form $x \mapsto \mathbf{t}$, which are using only during evaluation.[8] Note that the evaluation function also needs the list of declarations and definitions to annotate neutral terms with their type.

Definition 1.1.2 (Context).

1. $()$ *is the empty context;*

2. $\Gamma, x : A$ *extends the context Γ with a declaration, where the type A is a term;*

3. $\Gamma, x : A = M$ *extends the context Γ with a definition, where both the type A and the body M are terms;*

4. $\Gamma, x \mapsto \mathbf{t}$ *extends the context Γ binding the variable x to the value $\mathbf{t}$.*

Although being used for the formal recursive definition, we will obviously omit the $()$ symbol for the empty context.

The utility function LOOKUPTYPE retrieves the (evaluated) type of an identifier from the given context; keep in mind that evaluation is done only after type-checking, so that an eventual variable with no declaration is reported as an error before evaluation takes place. The function LOOKUPTYPE can be specified formally by recursion on the structure of the context (where the last '$-$' means 'every other possibility'):

$$\text{LOOKUPTYPE}((\Gamma, x : A), x) = (A)_\Gamma$$
$$\text{LOOKUPTYPE}((\Gamma, x : A = B), x) = (A)_\Gamma$$
$$\text{LOOKUPTYPE}((\Gamma, -), x) = \text{LOOKUPTYPE}(\Gamma,, x)$$

1.1.3 Typing rules

We use a technique called *bidirectional type checking*, that is we define and implement two relations, one for type inference (written $\Gamma \vdash t \Rightarrow \mathbf{ty}$) and one for type checking (written $\Gamma \vdash t \Leftarrow \mathbf{ty}$), which allow to type-check programs in a fairly straightforward manner while maintaining a close correspondence with

---

[8]Therefore our notion of context is not the usual one; it may be more appropriately called 'extended context'.

the syntax and also producing good error messages.[9] Moreover, this approach requires type annotations only at the 'top-level' of the declarations and not in every sub-term, allowing e.g. to write only `inl` a instead of $\text{inl}_{A,B}\ a$, (a,b) instead of $(a,b)_{A,B}$ and even λ-abstractions without mentioning the type, as it has to be already present in the top-level signature (e.g. in $f : \text{N} \to \text{N} = [n]\ \text{S}\ n$, n must have type N due to the signature).[10]

We present now the rules for a bidirectional type-checker for the basic theory; later, to handle partial and restriction types, we will use not only the context Γ for type-checking but also a *directions environment* Θ (see section 1.2.3).

Type inference

$$\overline{\Gamma \vdash x \Rightarrow \text{LOOKUPTYPE}(\Gamma, x)} \qquad\qquad\qquad \text{(Var)}$$

$$\overline{\Gamma \vdash \text{U} \Rightarrow \text{U}} \qquad\qquad\qquad\qquad \text{(Universe)}$$

$$\frac{\Gamma \vdash f \Rightarrow \text{closure}([x:ty]e, \Gamma') \quad \Gamma \vdash a \Leftarrow (ty)_{\Gamma'}}{\Gamma \vdash f\ a \Rightarrow \text{APP}(\text{closure}([x:ty]e, \Gamma'), (a)_{\Gamma})} \qquad \text{(App)}$$

$$\frac{\Gamma \vdash p \Rightarrow \text{closure}(\langle x:ty \rangle e, \Gamma')}{\Gamma \vdash p.1 \Rightarrow (ty)_{\Gamma'}} \qquad\qquad \text{(Sigma-1)}$$

$$\frac{\Gamma \vdash p \Rightarrow \text{closure}(\langle x:ty \rangle e, \Gamma')}{\Gamma \vdash p.2 \Rightarrow (e)_{\Gamma', x \mapsto (p.1)}} \qquad\qquad \text{(Sigma-2)}$$

$$\frac{\begin{array}{c} \Gamma \vdash t \Rightarrow \mathbf{ty_1} + \mathbf{ty_2} \quad \Gamma \vdash f_1 \Leftarrow ([x:ty_1]\ ty\ (\text{inl}\ x))_{\Gamma} \\ \Gamma \vdash f_2 \Leftarrow ([x:ty_2]\ ty\ (\text{inr}\ x))_{\Gamma} \end{array}}{\Gamma \vdash \text{split}\ ty\ f_1\ f_2\ t \Rightarrow (ty\ t)_{\Gamma}} \ (x = \text{NEWVAR}(\Gamma))$$
$$\text{(Split)}$$

---

[9]Compare with the Hindley-Milner type-checking algorithm, which uses the unification algorithm. Error reporting there can be difficult to interpret.

[10]However, to make the syntax more uniform and simple, we require to always include the type in abstractions.

$$\overline{\Gamma \vdash \mathtt{N} \Rightarrow \mathtt{U}} \tag{Nat}$$

$$\overline{\Gamma \vdash \mathtt{Z} \Rightarrow \mathtt{N}} \tag{Zero}$$

$$\frac{\Gamma \vdash n \Leftarrow \mathtt{N}}{\Gamma \vdash \mathtt{S}\ n \Rightarrow \mathtt{N}} \tag{Succ}$$

$$\frac{\begin{array}{ll} \Gamma \vdash F \Leftarrow \mathtt{N} \to \mathtt{U} & \Gamma \vdash c_0 \Leftarrow \mathrm{APP}((F)_\Gamma, \mathtt{Z}) \\ \Gamma \vdash n \Leftarrow \mathtt{N} & \Gamma \vdash c_s \Leftarrow ([m : \mathtt{N}]\ F\ m \to F\ (\mathtt{S}\ m))_\Gamma \end{array}}{\Gamma \vdash \mathtt{ind}\ F\ c_0\ c_s\ n \Rightarrow \mathrm{APP}((F)_\Gamma, (n)_\Gamma)} \tag{Ind}$$

Type checking

$$\frac{\Gamma \vdash ty \Leftarrow \mathtt{U} \quad \Gamma \vdash e \Leftarrow (ty)_\Gamma \quad \Gamma, e : ty = e \vdash t \Leftarrow \mathbf{A}}{\Gamma \vdash [x : ty = e]t \Leftarrow \mathbf{A}} \tag{Def}$$

$$\frac{\Gamma \vdash ty \Leftarrow \mathtt{U} \quad \Gamma, x : ty \vdash e \Leftarrow \mathtt{U}}{\Gamma \vdash [x : ty]e \Leftarrow \mathtt{U}} \tag{Pi}$$

$$\frac{\Gamma \vdash ty \Leftarrow \mathtt{U} \quad \Gamma, x : ty \vdash e \Leftarrow \mathtt{U}}{\Gamma \vdash \langle x : ty \rangle e \Leftarrow \mathtt{U}} \tag{Sigma}$$

$$\frac{\begin{array}{l} \Gamma \vdash ty \Leftarrow \mathtt{U} \qquad (ty)_\Gamma \sim_{\tau(\Gamma)} (ty_1)_{\Gamma_1} \\ \Gamma, x : ty, x \mapsto v \vdash e \Leftarrow \mathrm{APP}(\mathsf{closure}([x_1 : ty_1]e_1, \Gamma_1)\rangle, v^{(ty_1)_{\Gamma_1}}) \end{array}}{\Gamma \vdash [x : ty]e \Leftarrow \mathsf{closure}([x_1 : ty_1]e_1, \Gamma_1)} \ (v = \mathrm{NEWVAR}(\Gamma_1, x)) \tag{Abstraction}$$

$$\frac{\Gamma \vdash s \Leftarrow (ty)_{\Gamma'} \qquad \Gamma \vdash t \Leftarrow (e)_{\Gamma', x \mapsto (s)}}{\Gamma \vdash (s, t) \Leftarrow \mathsf{closure}(\langle x : ty \rangle e, \Gamma')} \tag{Pair}$$

$$\frac{\Gamma \vdash A \Leftarrow \mathtt{U} \qquad \Gamma \vdash B \Leftarrow \mathtt{U}}{\Gamma \vdash A + B \Leftarrow \mathtt{U}} \tag{Coproduct}$$

$$\frac{\Gamma \vdash t_1 \Leftarrow \mathbf{A}}{\Gamma \vdash \mathtt{inl}\ t_1 \Leftarrow \mathbf{A} + \mathbf{B}} \tag{Inl}$$

$$\frac{\Gamma \vdash t_2 \Leftarrow \mathbf{B}}{\Gamma \vdash \mathtt{inr}\ t_2 \Leftarrow \mathbf{A} + \mathbf{B}} \tag{Inr}$$

$$\frac{\Gamma \vdash e \Rightarrow \mathbf{A} \quad \mathbf{A} \sim_{\tau(\Gamma)} \mathbf{A}'}{\Gamma \vdash e \Leftarrow \mathbf{A}'} \tag{Infer}$$

1.1.4 Operational semantics and evaluation

Given a context Γ, the function EVAL gets a term as input and gives the corresponding value as output. Instead of writing $\mathrm{EVAL}_\Gamma(t)$, we shall simply write $(t)_\Gamma$. The key point of evaluation is that of using closures to defer computation in the case of abstractions of the form $[x : A]M$ or $\langle x : A\rangle M$; as already explained in 1.1, in this way the structure of the body is preserved until it becomes free of abstractions, and only at that point the substitutions are actually performed.

Evaluation is then defined by recursion on the structure of the terms.

$$
\begin{aligned}
(x)_\Gamma &= x^{\mathrm{LOOKUPTYPE}(\Gamma, x)} \\
(\mathtt{U})_\Gamma &= \mathtt{U} \\
([x : ty = e]t)_\Gamma &= (t)_{\Gamma, x:ty=e} \\
([x : ty]e)_\Gamma &= \mathsf{closure}([x : ty]e, \Gamma) \\
(f\ a)_\Gamma &= \mathrm{APP}((f)_\Gamma, (a)_\Gamma) \\
(\langle x : ty\rangle e)_\Gamma &= \mathsf{closure}(\langle x : ty\rangle e, \Gamma) \\
(t_1, t_2)_\Gamma &= ((t_1)_\Gamma, (t_2)_\Gamma) \\
(t.1)_\Gamma &= \mathrm{FST}((t)_\Gamma) \\
(t.2)_\Gamma &= \mathrm{SND}((t)_\Gamma) \\
(A + B)_\Gamma &= (A)_\Gamma + (B)_\Gamma \\
(\mathtt{inl}\ t_1)_\Gamma &= \mathtt{inl}\ (t_1)_\Gamma \\
(\mathtt{inr}\ t_2)_\Gamma &= \mathtt{inr}\ (t_2)_\Gamma \\
(\mathtt{split}\ ty\ f_1\ f_2\ t)_\Gamma &= \mathrm{SPLIT}((ty)_\Gamma, (f_1)_\Gamma, (f_2)_\Gamma, (t)_\Gamma) \\
(\mathtt{N})_\Gamma &= \mathtt{N}
\end{aligned}
$$

$$(\text{Z})_\Gamma = \text{Z}$$
$$(\text{S } n)_\Gamma = \text{S } (n)_\Gamma$$
$$(\text{ind } F \ c_0 \ c_s \ n)_\Gamma = \text{IND}((F)_\Gamma, (c_0)_\Gamma, (c_s)_\Gamma, (n)_\Gamma)$$

Note that the result of the evaluation of a variable is always a neutral term, which is the variable itself annotated with its type; moreover, each eliminator is handled individually using an helper function (written in small caps), as we need to handle carefully the case when the argument is neutral. We specify them below, but we will need to expand them to handle restriction types in 1.4.

- APP applied to $\mathbf{f}$ and $\mathbf{a}$ first checks if $\mathbf{f}$ is a closure, and if so it evaluates it, otherwise $\mathbf{f} \ \mathbf{a}$ must be a neutral application which gets annotated with its type.[11]

$$\text{APP}(\mathsf{closure}([x : ty]e, \Gamma), \mathbf{a}) = (e)_{\Gamma, x \mapsto \mathbf{a}}$$
$$\text{APP}(\mathbf{f}^{\mathsf{closure}([x:ty]e,\Gamma)}, \mathbf{a}) = (\mathbf{f} \ \mathbf{a})^{\text{APP}(\mathsf{closure}([x:ty]e,\Gamma)}$$

- IND evaluates an induction by pattern-matching on the argument n; if n is neutral, then the whole induction is neutral and so it gets annotated with its type.

$$\text{IND}(\mathbf{F}, \mathbf{c}_0, \mathbf{c}_s, \text{Z}) = \mathbf{c}_0$$
$$\text{IND}(\mathbf{F}, \mathbf{c}_0, \mathbf{c}_s, \text{S } \mathbf{m}) = \text{APP}(\text{APP}(\mathbf{c}_s, \mathbf{m}), \text{IND}(\mathbf{F}, \mathbf{c}_0, \mathbf{c}_s, \mathbf{m}))$$
$$\text{IND}(\mathbf{F}, \mathbf{c}_0, \mathbf{c}_s, \mathbf{n}^{\mathbb{N}}) = (\text{ind } F \ \mathbf{c}_0 \ \mathbf{c}_s \ \mathbf{n})^{\text{APP}(\mathbf{F}, \mathbf{n})}$$

- FST and SND return respectively the first and the second component of a pair, if the value is not neutral; otherwise they annotate the whole value with the corresponding type.

$$\text{FST}((\mathbf{v}_1, \mathbf{v}_2)) = \mathbf{v}_1$$
$$\text{FST}(\mathbf{v}^{\mathsf{closure}(\langle x:ty \rangle e, \Gamma)}) = (\mathbf{v}.1)^{(ty)_\Gamma}$$
$$\text{SND}((\mathbf{v}_1, \mathbf{v}_2)) = \mathbf{v}_2$$
$$\text{SND}(\mathbf{v}^{\mathsf{closure}(\langle x:ty \rangle e, \Gamma)}) = (\mathbf{v}.2)^{(e)_{\Gamma, x \mapsto \text{FST}(\mathbf{v})}}$$

- SPLIT does by-case analysis on the given argument, if it is not neutral, otherwise the whole term becomes neutral.

$$\text{SPLIT}(\mathbf{F}, \mathbf{f}_1, \mathbf{f}_2, \text{inl } \mathbf{v}_1) = \text{APP}(\mathbf{f}_1, \mathbf{v}_1)$$

---

[11]Remember that terms are type-checked before they get evaluated, so the types are assumed to be correct, and in particular in this case the type of $\mathbf{f}$ has to be a function type.

$$\text{SPLIT}(\mathbf{F}, \mathbf{f}_1, \mathbf{f}_2, \texttt{inr } \mathbf{v}_2) = \text{APP}(\mathbf{f}_2, \mathbf{v}_2)$$

$$\text{SPLIT}(\mathbf{F}, \mathbf{f}_1, \mathbf{f}_2, \mathbf{v}^{\mathbf{ty}'}) = (\texttt{split } \mathbf{F} \ \mathbf{f}_1 \ \mathbf{f}_2 \ \mathbf{v})^{\text{APP}(\mathbf{F}, \mathbf{v})}$$

Reading back

Having discussed evaluation, we now describe the function $\mathcal{R}$ which converts values back into terms, that is by recursively evaluating the closures. In that way, starting from a term, by first evaluating and then reading back, one gets the normal form of that term.

The read-back function $\mathcal{R}$ keeps track of the list of already used names ns, to avoid name conflicts when reading back closures. For all the values except closures, $\mathcal{R}$ is defined inductively in the obvious manner.

$$\mathcal{R}_{ns}(\mathbf{x}) = x$$
$$\mathcal{R}_{ns}(\mathtt{U}) = \mathtt{U}$$
$$\mathcal{R}_{ns}(\mathbf{k} \ \mathbf{a}) = \mathcal{R}_{ns}(\mathbf{k}) \ \mathcal{R}_{ns}(\mathbf{a})$$
$$\mathcal{R}_{ns}(\mathbf{v}_1, \mathbf{v}_2) = (\mathcal{R}_{ns}(\mathbf{v}_1), \mathcal{R}_{ns}(\mathbf{v}_2))$$
$$\mathcal{R}_{ns}(\mathbf{k}.1) = \mathcal{R}_{ns}(\mathbf{k}).1$$
$$\mathcal{R}_{ns}(\mathbf{k}.2) = \mathcal{R}_{ns}(\mathbf{k}).2$$
$$\mathcal{R}_{ns}(\mathbf{A} + \mathbf{B}) = \mathcal{R}_{ns}(\mathbf{A}) + \mathcal{R}_{ns}(\mathbf{B})$$
$$\mathcal{R}_{ns}(\texttt{inl } \mathbf{v}_1) = \texttt{inl } \mathcal{R}_{ns}(\mathbf{v}_1)$$
$$\mathcal{R}_{ns}(\texttt{inr } \mathbf{v}_2) = \texttt{inr } \mathcal{R}_{ns}(\mathbf{v}_2)$$
$$\mathcal{R}_{ns}(\texttt{split ty } \mathbf{f}_1 \ \mathbf{f}_2 \ \mathbf{k}) = \text{SPLIT}(\mathcal{R}_{ns}(\mathbf{ty}), \mathcal{R}_{ns}(\mathbf{f}_1), \mathcal{R}_{ns}(\mathbf{f}_2), \mathcal{R}_{ns}(\mathbf{k}))$$
$$\mathcal{R}_{ns}(\mathtt{N}) = \mathtt{N}$$
$$\mathcal{R}_{ns}(\mathtt{Z}) = \mathtt{Z}$$
$$\mathcal{R}_{ns}(\mathtt{S} \ n) = \mathtt{S} \ \mathcal{R}_{ns}(n)$$
$$\mathcal{R}_{ns}(\texttt{ind } F \ c_0 \ c_s \ k) = \text{IND}(\mathcal{R}_{ns}(F), \mathcal{R}_{ns}(c_0), \mathcal{R}_{ns}(c_s), \mathcal{R}_{ns}(k))$$

Finally for closures we read-back the argument type and the evaluated body (with an eventual fresh variable x').

$$\mathcal{R}_{ns}(\mathsf{closure}([x : ty]e, \Gamma)) = [x' : ty']e'$$
$$\mathcal{R}_{ns}(\mathsf{closure}(\langle x : ty \rangle e, \Gamma)) = \langle x' : ty' \rangle e'$$

where

$$\begin{cases} x' & = \text{NEWVAR}(ns, x) \\ ty' & = \mathcal{R}_{ns}((ty)_\Gamma) \\ e' & = \mathcal{R}_{ns, x'}((e)_{\Gamma, x \mapsto x'(ty)_\Gamma}) \end{cases}$$

1.1.5 Examples

We report some brief examples which can be written in this basic type theory.

We can define, using the well known recursive definitions, the basic arithmetic operations.

$\texttt{idN} : \texttt{N} \to \texttt{U} = [x : \texttt{N}] \ \texttt{N}$

$\texttt{plus} : \texttt{N} \to \texttt{N} \to \texttt{N} = [m : \texttt{N}][n : \texttt{N}] \ \texttt{ind idN} \ m \ ([n' : \texttt{N}][mPn' : \texttt{N}] \ \texttt{S} \ mPn') \ n$

$\texttt{mult} : \texttt{N} \to \texttt{N} \to \texttt{N} = [m : \texttt{N}][n : \texttt{N}] \ \texttt{ind idN} \ \texttt{Z} \ ([n' : \texttt{N}][mTn' : \texttt{N}] \ \texttt{plus} \ mTn' \ m) \ n$

$\texttt{exp} : \texttt{N} \to \texttt{N} \to \texttt{N} = [m : \texttt{N}][n : \texttt{N}] \ \texttt{ind idN} \ (\texttt{S} \ \texttt{Z}) \ ([n' : \texttt{N}][mEn' : \texttt{N}] \ \texttt{mult} \ mEn' \ m) \ n$

These functions obviously satisfy the identities between numerals judgmentally, and it is possible to prove (using path types, see 1.2.1) the usual properties by induction (commutativity, associativity, etc).

We assume to have a unit type (which we will introduce in 1.6) with a single constructor and the induction/elimination principle:

$$\texttt{tt} : \texttt{unit}$$

$$\texttt{unitInd} : [C : \texttt{unit} \to \texttt{U}] \ C \ \texttt{tt} \to [x : \texttt{unit}] \ C \ x$$

Using $\texttt{unit}$ and coproducts we can form the $\texttt{bool}$ type and derive its induction/elimination principle:

$$\texttt{bool} : \texttt{U} = \texttt{unit} + \texttt{unit}$$

$$\texttt{false} : \texttt{bool} = \texttt{inl tt}$$

$$\texttt{true} : \texttt{bool} = \texttt{inr tt}$$

$$\texttt{boolInd} : [C : \texttt{bool} \to \texttt{U}] \ C \ \texttt{false} \to C \ \texttt{true} \to [x : \texttt{bool}] \ C \ x$$
$$= [C : \texttt{bool} \to \texttt{U}][cf : C \ \texttt{false}][ct : C \ \texttt{true}][x : \texttt{bool}]$$
$$\texttt{split} \ C \ ([u : \texttt{unit}] \ \texttt{unitInd} \ ([u' : \texttt{unit}] \ C \ (\texttt{inl} \ u')) \ cf \ u)$$
$$([u : \texttt{unit}] \ \texttt{unitInd}([u' : \texttt{unit}] \ C \ (\texttt{inr} \ u')) \ ct \ u) \ x$$

$$\texttt{boolRec} : [C : \texttt{U}] \ C \to C \to \texttt{bool} \to C$$
$$= [C : \texttt{U}][f : C][t : C][x : \texttt{bool}] \ \texttt{boolInd} \ (\texttt{bool} \to C) \ f \ t \ x$$

Then we can define the boolean operations and some relations (e.g. equality, less-or-equal) between naturals using the recursive characterization.[12]

$$\texttt{If} : [C : \texttt{U}] \ \texttt{bool} \to C \to C \to C$$

---

[12] We define by induction on $m : \texttt{N}$ a function $\texttt{N} \to \texttt{bool}$, thought as $(m = -)$. When $m \equiv \texttt{Z}$, the function returns $\texttt{true}$ on $\texttt{Z}$ and $\texttt{false}$ on successors; when $m \equiv \texttt{S} \ m'$, the function returns $\texttt{false}$ on $\texttt{Z}$ and on successors $n \equiv \texttt{S} \ n'$ it computes the previous value (i.e. $m' = n'$, the inductive hypothesis). Similarly for 'less-or-equal'.

$$= [C : \mathtt{U}][x : \mathtt{bool}][t : C][f : C] \ \mathtt{boolRec} \ C \ f \ t \ x$$
$$\mathtt{not} : \mathtt{bool} \to \mathtt{bool} = [b : \mathtt{bool}] \ \mathtt{If} \ \mathtt{bool} \ b \ \mathtt{false} \ \mathtt{true}$$
$$\mathtt{and} : \mathtt{bool} \to \mathtt{bool} \to \mathtt{bool} = [b1, b2 : \mathtt{bool}] \ \mathtt{If} \ \mathtt{bool} \ b1 \ b2 \ \mathtt{false}$$
$$\mathtt{or} : \mathtt{bool} \to \mathtt{bool} \to \mathtt{bool} = [b1, b2 : \mathtt{bool}] \ \mathtt{If} \ \mathtt{bool} \ b1 \ \mathtt{true} \ b2$$
$$\mathtt{natEq} : \mathtt{N} \to \mathtt{N} \to \mathtt{bool} = [m : \mathtt{N}] \ \mathtt{ind} \ (\mathtt{N} \to \mathtt{N} \to \mathtt{bool})$$
$$([n : \mathtt{N}] \ \mathtt{ind} \ (\mathtt{N} \to \mathtt{bool}) \ \mathtt{true} \ (\mathtt{N} \to \mathtt{bool} \to \mathtt{false}) \ n)$$
$$([m' : \mathtt{N}][m'Eq : \mathtt{N} \to \mathtt{bool}][n : \mathtt{N}]$$
$$\mathtt{ind} \ (\mathtt{N} \to \mathtt{bool}) \ \mathtt{false} \ ([n' : \mathtt{N}] \ \mathtt{bool} \to m'Eq \ n') \ n) \ m$$
$$\mathtt{natNeq} : \mathtt{N} \to \mathtt{N} \to \mathtt{bool} = [m, n : \mathtt{N}] \ \mathtt{not} \ (\mathtt{natEq} \ m \ n)$$
$$\mathtt{natLeq} : \mathtt{N} \to \mathtt{N} \to \mathtt{bool} = [m : \mathtt{N}] \ \mathtt{ind} \ (\mathtt{N} \to \mathtt{N} \to \mathtt{bool}) \ (\mathtt{N} \to \mathtt{true})$$
$$([m' : \mathtt{N}][m'Leq : \mathtt{N} \to \mathtt{bool}][n : \mathtt{N}]$$
$$\mathtt{ind} \ (\mathtt{N} \to \mathtt{bool}) \ \mathtt{false} \ ([n' : \mathtt{N}] \ \mathtt{bool} \to m'Leq \ n') \ n) \ m$$
$$\mathtt{natLe} : \mathtt{N} \to \mathtt{N} \to \mathtt{bool} = [m, n : \mathtt{N}] \ \mathtt{and} \ (\mathtt{natLeq} \ m \ n) \ (\mathtt{natNeq} \ m \ n)$$

Using `bool` and function types we can define an alternative product type, which is the type of functions $\mathtt{bool} \to \mathtt{U}$ mapping `false` to the first type and `true` to the second one. In section 1.6, after introducing path types and transport, we will prove the propositional η-equality and the inductive/elimination principle for `prod`, which will finally allow us to prove that $\mathtt{prod} \ A \ B$ is equivalent to $A * B$.

$$\mathtt{prod} : \mathtt{U} \to \mathtt{U} \to \mathtt{U} = [A, B : \mathtt{U}][x : \mathtt{bool}] \ \mathtt{boolRec} \ \mathtt{U} \ A \ B \ x$$
$$\mathtt{couple} : [A, B : \mathtt{U}] \ A \to B \to \mathtt{prod} \ A \ B$$
$$= [A, B : \mathtt{U}][a : A][b : B] \ \mathtt{boolInd} \ ([x : \mathtt{bool}] \ \mathtt{boolRec} \ \mathtt{U} \ A \ B \ x) \ a \ b$$
$$\mathtt{pi1} : [A, B : \mathtt{U}] \ \mathtt{prod} \ A \ B \to A = [A, B : \mathtt{U}][p : \mathtt{prod} \ A \ B] \ p \ \mathtt{false}$$
$$\mathtt{pi2} : [A, B : \mathtt{U}] \ \mathtt{prod} \ A \ B \to B = [A, B : \mathtt{U}][p : \mathtt{prod} \ A \ B] \ p \ \mathtt{true}$$

As a last example, we can prove the so called 'type-theoretic axiom of choice' (see [Uni13] 1.6 and 2.15.7) and the induction/elimination principle for Σ-types,[13] but note that these need η-conversion to type-check. From the inductor principle we can deduce the recursion (i.e. non-dependent) principle.

$$\mathtt{AC} : [A : \mathtt{U}][B : \mathtt{U}][C : A \to B \to \mathtt{U}]$$
$$([x : A]\langle y : B\rangle \ C \ x \ y) \to \langle f : A \to B\rangle[x : A] \ C \ x \ (f \ x)$$
$$= [A : \mathtt{U}][B : \mathtt{U}][C : A \to B \to \mathtt{U}][h : [x : A]\langle y : B\rangle \ C \ x \ y]$$
$$(([x : A](h \ x).1), ([x : A](h \ x).2))$$

---

[13]Recall that we choose to use projections as primitives in our language.

$$\texttt{indSigma} : [A : \texttt{U}][B : A \to \texttt{U}][C : (\langle x : A \rangle B \ x) \to \texttt{U}]$$
$$([a : A][b : B \ a]C((a, b))) \to [p : \langle x : A \rangle \ B \ x] \ C \ p$$
$$= [A : \texttt{U}][B : A \to \texttt{U}][C : (\langle x : A \rangle \ B \ x) \to \texttt{U}]$$
$$[h : [a : A][b : B \ a] \ C \ (a, b)][p : \langle x : A \rangle \ B \ x] \ h \ p.1 \ p.2$$

$$\texttt{recSigma} : [A : \texttt{U}][B : \texttt{U}][C : A * B \to \texttt{U}]$$
$$([a : A][b : B] \ C \ (a, b)) \to [p : A * B]C \ p$$
$$= [A : \texttt{U}][B : \texttt{U}][C : A * B \to \texttt{U}][h : [a : A][b : B] \ C \ (a, b)]$$
$$\texttt{indSigma} \ A \ (A \to B) \ C \ h$$

1.2 Extension with partial and restriction types

We now extend the basic type theory discussed so far with the constructs of cubical type theory, that is the interval, formulas, partial and restriction types.

The syntax is expanded as follows:

$$A, B, M ::= \ \ldots$$

$$\mid \mathbb{I} \mid \texttt{0} \mid \texttt{1} \qquad\qquad\qquad\qquad \text{(Interval)}$$
$$\mid [\psi_1 \mapsto M_1, \ldots, \psi_n \mapsto M_n] \qquad\qquad \text{(System)}$$
$$\mid [\varphi]A \qquad\qquad\qquad\qquad\qquad \text{(Partial type)}$$
$$\mid [\psi_1 \mapsto M_1, \ldots, \psi_n \mapsto M_n]A \qquad \text{(Restriction type)}$$

$$K ::= \ \ldots$$
$$\mid \texttt{comp} \ K \ \varphi \ M_0 \ M_u \ M_b \ M_i \qquad\qquad \text{(Composition)}$$

However, care must be taken in the case of `comp`, as it may become neutral when the first argument (type family) is not neutral but M_b is.[14]

We also introduce a syntax for formulas (we use i and j for generic names for *interval variables*).

$$\chi ::= (i = 0) \mid (i = 1) \mid (i = j) \qquad\qquad \text{(Atomic)}$$
$$\psi ::= \chi \mid \chi \wedge \psi \qquad\qquad\qquad\qquad \text{(Conjunction)}$$

---

[14]For example for coproducts and naturals.

$$\varphi ::= \psi \mid \psi \vee \varphi \qquad\qquad \text{(Disjunctive n.f.)}$$

We chose not to allow arbitrary formulas in the syntax, but only conjunctive formulas in systems and restriction types and formulas in disjunctive normal form in partial types, because it makes little sense to write programs breaking that rule and because this constraint simplifies the implementation.

We will often use a standard vector notation for systems and restriction types, i.e. writing just

$$[\vec{\psi} \mapsto \vec{M}]$$

instead of

$$[\psi_1 \mapsto M_1, \ldots, \psi_n \mapsto M_n].$$

When also write for example

$$[\vec{\psi} \mapsto \vec{f}\ a], \qquad [\vec{\psi} \mapsto \vec{p}.1]$$

respectively for

$$[\psi_1 \mapsto f_1\ a, \ldots, \psi_n \mapsto f_n\ a], \qquad [\psi_1 \mapsto (p_1).1, \ldots, \psi_n \mapsto (p_n).1].$$

1.2.1 Path types

There is no need for a primitive syntax for path types in the language, as is in [CCHM15], because we can exploit restriction types to define path types as a derived notion. Indeed, a path $p : \texttt{Path}_A(a, b)$ is nothing else than a $p : \mathbb{I} \to A$ such that $p\ 0 \equiv a$ and $p\ 1 \equiv b$, that is

$$\texttt{Path}_A(a, b) \equiv [i : \mathbb{I}][i = 0 \mapsto a, i = 1 \mapsto b]A.$$

In our implementation $\texttt{Path}$ is actually defined like so:

$$\texttt{Path} \equiv [A : \texttt{U}][a, b : A][i : \mathbb{I}][i = 0 \mapsto a, i = 1 \mapsto b]A.$$

The trivial path at $a : A$, which corresponds to a proof of reflexivity, is then defined as the constant path at a, or more generally:

$$\texttt{refl} : [A : U][a : A]\ \texttt{Path}_A(a, a) = [A : U][a : A][i : I]\ a.$$

It's trivial to see that every inference rule of ([CCHM15] section 3.1) holds if path types are defined this way.

1.2.2   Syntax for composition

We use the following notation for compositions: $\mathtt{comp}\ F\ \varphi\ i_0\ u\ b\ i$, where:

- $F : \mathbb{I} \to \mathtt{U}$ is a type family;

- φ is the formula describing the sides of the composition figure;

- $i_0 : \mathbb{I}$ is the starting point of the composition;

- $u : [x : \mathbb{I}][\varphi]F\ x$ is the partial term, defined on φ, which will be extended by the composition;

- $b : [\varphi \mapsto u\ i_0]F\ i_0$ is the base case, defined on the starting point i_0; the term b must agree with $u\ i_0$ on φ;

- $i : \mathbb{I}$ is the end point of the composition.

The composition has type $[i = i_0 \mapsto b, \varphi \mapsto u\ i]F\ i$, i.e. it extends b on $i = i_0$ and u on φ. Note that it cannot be just $[\varphi \mapsto u\ i]F\ i$, as under $i = i_0$ the type would be $[\varphi \mapsto u\ i_0]F\ i_0$, the same as b, but we still have to assert that the composition and b are convertible in that case.

The inference rule for composition shall be therefore the following:[15]

$$\frac{\begin{array}{ccc} \Gamma \vdash \varphi : \mathbb{F} & \Gamma \vdash u : [x : \mathbb{I}][\varphi]\ F\ x & \Gamma \vdash i_0 : \mathbb{I} \\ \Gamma \vdash F : \mathbb{I} \to \mathtt{U} & \Gamma \vdash b : [\varphi \mapsto u\ i_0]F\ i_0 & \Gamma \vdash i\ : \mathbb{I} \end{array}}{\Gamma \vdash \mathtt{comp}\ F\ \varphi\ i_0\ u\ b\ i : [i = i_0 \mapsto b \mid \varphi \mapsto u\ i]F\ i}\ (x = \textsc{newVar}(\Gamma))$$

We make an example to make clear how composition works, demonstrating how to obtain the concatenation of two paths.

Suppose we are given a type $A : \mathtt{U}$ and three points $a, b, c : A$, with paths $p : \mathtt{Path}_A(a, b)$ and $q : \mathtt{Path}_A(b, c)$; we want to obtain a path $pq : \mathtt{Path}_A(a, c)$.

We draw a square having p as the base, so $i_0 \equiv \mathtt{0}$ is the starting point and $p\ i$ is the base case, with the constant path $\mathtt{refl}_A(a)$ as the left side and the path q as the right side. With composition (at the end point $\mathtt{1}$) we finally obtain a path from a to c. In more detail, keeping in mind that we quantify over $i : \mathbb{I}$ before the composition,

- F is the constant type family $\mathbb{I} \to A$;

- φ is $i = \mathtt{0} \vee i = \mathtt{1}$ (left and right sides of the square);

---

[15]In section 1.3 we will state it more precisely, conforming to the bidirectional type-checking mechanism and using the directions environment in addition to the context.
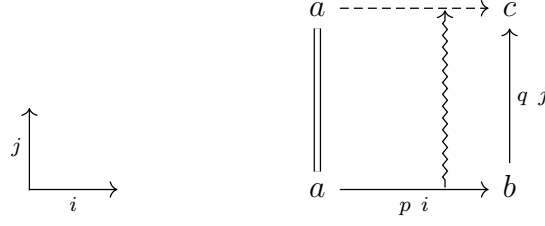
Figure 1.1: Concatenation of two paths. The result is the dotted line (lid of the square), which is a path connecting a to c. The twisted line only represents the act of composition (along the direction j), obtaining a point on the lid from the base point $p\ i$, for each point $i : \mathbb{I}$ which we quantify over.

- i_0 is the starting point 0;

- u is the partial term defined only on the sides, given by $[j : \mathbb{I}][i = 0 \mapsto a, i = 1 \mapsto q\ j]$;[16]

- b is the base case given by $p\ i$;

- The end point is 1.[17]

The code for concatenation is therefore:

$$\texttt{concat} : [A : \texttt{U}][a, b, c : A]\ \texttt{Path}\ A\ a\ b \to \texttt{Path}\ A\ b\ c \to \texttt{Path}\ A\ a\ c$$
$$= [A : \texttt{U}][a, b, c : A][p : \texttt{Path}\ A\ a\ b][q : \texttt{Path}\ A\ b\ c][i : \mathbb{I}]$$
$$\texttt{comp}\ (\mathbb{I} \to A)\ (i = 0 \vee i = 1)\ 0\ ([j : \mathbb{I}][i = 0 \mapsto a, i = 1 \mapsto q\ j])\ (p\ i)\ 1$$

Indeed we can verify, now informally using the evaluation rules presented in section 1.4, that the concatenation is a path with the right endpoints:

$$(\texttt{concat}\ A\ a\ b\ c\ p\ q)\ 0$$
$$\equiv \texttt{comp}\ (\mathbb{I} \to A)\ (0 = 0 \vee 0 = 1)\ 0\ ([j : \mathbb{I}][0 = 0 \mapsto a, 0 = 1 \mapsto q\ j])\ (p\ 0)\ 1$$
$$\equiv \texttt{comp}\ (\mathbb{I} \to A)\ \texttt{True}\ 0\ ([j : \mathbb{I}][\texttt{True} \mapsto a])\ a\ 1$$
$$\equiv ([j : \mathbb{I}][\texttt{True} \mapsto a])\ 0$$
$$\equiv a$$

---

[16]We could also have written $\texttt{refl}_A(a)\ j$ instead of a in the system.

[17]We use 1 because we are only interested in the lid, not on the whole square (filler), which we would get by using i as the end point.

And similarly:

$$
\begin{aligned}
&(\texttt{concat}\ A\ a\ b\ c\ p\ q)\ \texttt{1}\\
&\equiv \texttt{comp}\ (\mathbb{I} \to A)\ (\texttt{1} = \texttt{0} \vee \texttt{1} = \texttt{1})\ \texttt{0}\ ([j : \mathbb{I}][\texttt{1} = \texttt{0} \mapsto a, \texttt{1} = \texttt{1} \mapsto q\ j])\ (p\ \texttt{1})\ \texttt{1}\\
&\equiv \texttt{comp}\ (\mathbb{I} \to A)\ \texttt{True}\ \texttt{0}\ ([j : \mathbb{I}][\texttt{True} \mapsto q\ j])\ b\ \texttt{1}\\
&\equiv ([j : \mathbb{I}][\texttt{True} \mapsto q\ j])\ \texttt{1}\\
&\equiv q\ \texttt{1}\\
&\equiv c
\end{aligned}
$$

1.2.3 Directions environment

The type-checking rules (section 1.3) require the use of formulas in contexts, e.g. when type-checking a declaration of the form $\Gamma \vdash e : [\psi \mapsto t]A$, one has to check that $\Gamma, \psi \vdash t : A$, that is, t has to be of type A when subject to the constraint ψ; the context Γ, ψ has to be thought as a restricted context.

Instead of adding formulas to contexts, we use a different context-like structure to store information about formulas, called *directions environment*, because it makes the implementation and the presentation simpler, as the context and the directions environment serve different purposes which are handled distinctly.[18] The directions environment is used only during type-checking and $\alpha\eta$-conversion, and not during evaluation; it's important to remark that the management of formulas never yield an actual syntactical substitution.

Moreover, note that the meaning of a judgement of the form $\Gamma, \psi_1 \vee \ldots \psi_n \vdash J$ is that $\Gamma, \psi_1 \vdash J, \ldots, \Gamma, \psi_n \vdash J$, so that the directions environment only need to store information about a conjunction of atomic formulas.

To manage and simplify conjunctive formulas efficiently, we use a particular data structure which is formed by:

- A list (*zeros*) of names which shall be replaced by $\texttt{0}$.

- A list (*ones*) of names which shall be replaced by $\texttt{1}$.

- A list of partitions (called diagonals), that is, a list of lists of names; each partition represents the names which shall be identified.

---

[18]Note that if Γ and Δ are contexts without formulas, then a judgment in $\Gamma, \psi_1, \Delta, \psi_2$ is equivalent to a judgement in $\Gamma, \Delta, \psi_1 \wedge \psi_2$, i.e. the constraints are global in the context. This mean that we can always split the 'full' context into two contexts $\Gamma; \Theta$, where Γ does not contain any formula and Θ is the directions environment.

For example, the formula $(i = 0) \wedge (j = k) \wedge (w = r) \wedge (w = 1)$ is stored as $([i], [w, r], [[j, k]])$, meaning that i is mapped to 0, w, r to 1 and j, k are identified.

Even if it is implemented using the aforementioned lists, we will use a simple context-like notation for the directions environment, as if it were a list of bindings.

Definition 1.2.1 (Directions environment)**.**

1. $()$ *is the empty directions environment;*

2. $\Theta, i \mapsto 0$ *(or $i \mapsto 1$) extends the directions environment Θ binding the name i to 0 (or to 1);*

3. $\Theta, i \mapsto j$ *extends the directions environment Θ binding the name i to the name j (diagonal).*

1.3 Typing rules

We now present the type-checking rules for the full system, extending the ones of 1.1.3. The main difference is that now judgments are of the form

$$\Gamma; \Theta \vdash J$$

i.e. we use not only the context Γ, but also the directions environment Θ, which is used when type-checking under a formula (constraint).

We also need other kinds of judgments to state the rules, which are:

- $\Gamma \vdash \varphi : \mathbb{F}$ means that φ is a formula whose variables are declared in Γ.

- $\mathbf{v}_1 \sim^{\Theta}_{\tau(\Gamma)} \mathbf{v}_2$ means that values $\mathbf{v}_1$ and $\mathbf{v}_2$ are $\alpha\eta$-equivalent modulo the constraints in Θ; $\tau(\Gamma)$ is the list of names declared in the context Γ.[19]

- $\psi_1 \leq^{\Theta} \psi_2$ means that ψ_1 logically implies ψ_2 modulo the constraints in Θ.

- $\psi_1 \sim^{\Theta} \psi_2$ means that ψ_1 and ψ_2 are logically equivalent modulo the constraints in Θ, i.e. that $\psi_1 \leq^{\Theta} \psi_2$ and $\psi_2 \leq^{\Theta} \psi_1$.

The implementation of the three judgments regarding the formulas is relatively straightforward and is discussed in section 3.5.

---

[19]See section 1.5.

We allow only a very restricted form of subtyping, specified by the rules (Weak-Partial) and (Weak-Restr). This means that subtyping works only at the 'top-level', e.g.

$$x : [i = 0 \ \vee \ i = 1]\mathtt{N}$$
$$y : [i = 0]\mathtt{N} = x$$

is admitted but

$$f : [i : \mathbb{I}][i = 0 \ \vee \ i = 1]\mathtt{N}$$
$$g : [i : \mathbb{I}][i = 0]\mathtt{N} = f$$

is rejected.[20]   Note that we can still define g as follows:

$$g' : [i : \mathbb{I}][i = 0]\mathtt{N} = [i : \mathbb{I}] \ f \ i$$

Type inference

$$\frac{}{\Gamma; \Theta \vdash x \Rightarrow \textsc{lookupType}(\Gamma, x)} \qquad \text{(Var)}$$

$$\frac{}{\Gamma; \Theta \vdash \mathtt{U} \Rightarrow \mathtt{U}} \qquad \text{(Universe)}$$

$$\frac{\Gamma; \Theta \vdash f \Rightarrow \mathsf{closure}([x : ty]e, \Gamma') \quad \Gamma; \Theta \vdash a \Leftarrow (ty)_{\Gamma'}}{\Gamma; \Theta \vdash f \ a \Rightarrow \textsc{app}(\mathsf{closure}([x : ty]e, \Gamma'), (a)_\Gamma)} \qquad \text{(App)}$$

$$\frac{\Gamma; \Theta \vdash f \Rightarrow [\vec{\psi} \mapsto \vec{\mathbf{g}}]\mathsf{closure}([x : ty]e, \Gamma') \quad \Gamma; \Theta \vdash a \Leftarrow (ty)_{\Gamma'}}{\Gamma; \Theta \vdash f \ a \Rightarrow [\vec{\psi} \mapsto \textsc{app}(\vec{\mathbf{g}}, (a)_\Gamma)]\textsc{app}(\mathsf{closure}([x : ty]e, \Gamma'), (a))} \quad \text{(App-Restr)}$$

$$\frac{\Gamma; \Theta \vdash f \Rightarrow [\varphi]\mathsf{closure}([x : ty]e, \Gamma') \quad \Gamma; \Theta \vdash a \Leftarrow (ty)_{\Gamma'}}{\Gamma; \Theta \vdash f \ a \Rightarrow [\varphi]\textsc{app}(\mathsf{closure}([x : ty]e, \Gamma'), (a))} \qquad \text{(App-Partial)}$$

---

[20]Even $y' : [i = 0]\mathtt{N} = [i = 0 \mapsto \mathtt{Z}, i = 1 \mapsto \mathtt{S \ Z}]$ is rejected, as it makes no sense to write it that way (compare with y).

$$\frac{\Gamma; \Theta \vdash p \Rightarrow \mathsf{closure}(\langle x : ty \rangle e, \Gamma')}{\Gamma; \Theta \vdash p.1 \Rightarrow (ty)_{\Gamma'}} \qquad \text{(Sigma-1)}$$

$$\frac{\Gamma; \Theta \vdash p \Rightarrow \mathsf{closure}(\langle x : ty \rangle e, \Gamma')}{\Gamma; \Theta \vdash p.2 \Rightarrow (e)_{\Gamma', x \mapsto (p.1)}} \qquad \text{(Sigma-2)}$$

$$\frac{\Gamma; \Theta \vdash p \Rightarrow [\vec{\psi} \mapsto \vec{\mathbf{q}}]\mathsf{closure}(\langle x : ty \rangle e, \Gamma')}{\Gamma; \Theta \vdash p.1 \Rightarrow [\vec{\psi} \mapsto \mathrm{FST}(\vec{\mathbf{q}})](ty)_{\Gamma'}} \qquad \text{(Sigma-1-Restr)}$$

$$\frac{\Gamma; \Theta \vdash p \Rightarrow [\vec{\psi} \mapsto \vec{\mathbf{q}}]\mathsf{closure}(\langle x : ty \rangle e, \Gamma')}{\Gamma; \Theta \vdash p.2 \Rightarrow [\vec{\psi} \mapsto \mathrm{SND}(\vec{\mathbf{q}})](e)_{\Gamma', x \mapsto (p.1)}} \qquad \text{(Sigma-2-Restr)}$$

$$\frac{\Gamma; \Theta \vdash p \Rightarrow [\varphi]\mathsf{closure}(\langle x : ty \rangle e, \Gamma')}{\Gamma; \Theta \vdash p.1 \Rightarrow [\varphi](ty)_{\Gamma'}} \qquad \text{(Sigma-1-Partial)}$$

$$\frac{\Gamma; \Theta \vdash p \Rightarrow [\varphi]\mathsf{closure}(\langle x : ty \rangle e, \Gamma')}{\Gamma; \Theta \vdash p.2 \Rightarrow [\varphi](e)_{\Gamma', x \mapsto (p.1)}} \qquad \text{(Sigma-2-Partial)}$$

$$\frac{\begin{array}{ll} \Gamma; \Theta \vdash F \Leftarrow \mathbf{ty_1} + \mathbf{ty_2} \to \mathtt{U} & \Gamma; \Theta \vdash f_1 \Leftarrow ([x : ty_1]\ F\ (\mathtt{inl}\ x))_{\Gamma} \\ \Gamma; \Theta \vdash t \Rightarrow \mathbf{ty_1} + \mathbf{ty_2} & \Gamma; \Theta \vdash f_2 \Leftarrow ([x : ty_2]\ F\ (\mathtt{inr}\ x))_{\Gamma} \end{array}}{\Gamma; \Theta \vdash \mathtt{split}\ F\ f_1\ f_2\ t \Rightarrow (F\ t)_{\Gamma}} \ (x = \mathrm{NEWVAR}(\Gamma))$$
$$\text{(Split)}$$

$$\frac{\begin{array}{ll} \Gamma; \Theta \vdash F \Leftarrow \mathbf{ty_1} + \mathbf{ty_2} \to \mathtt{U} & \Gamma; \Theta \vdash f_1 \Leftarrow ([x : ty_1]\ F\ (\mathtt{inl}\ x))_{\Gamma} \\ \Gamma; \Theta \vdash t \Rightarrow [\vec{\psi} \mapsto \vec{\mathbf{t}}](\mathbf{ty_1} + \mathbf{ty_2}) & \Gamma; \Theta \vdash f_2 \Leftarrow ([x : ty_2]\ F\ (\mathtt{inr}\ x))_{\Gamma} \end{array}}{\Gamma; \Theta \vdash \mathtt{split}\ F\ f_1\ f_2\ t \Rightarrow [\vec{\psi} \mapsto (F)\ \vec{\mathbf{t}}](F\ t)_{\Gamma}} \ (x = \mathrm{NEWVAR}(\Gamma))$$
$$\text{(Split-Restr)}$$

$$\frac{\begin{array}{ll} \Gamma; \Theta \vdash F \Leftarrow \mathbf{ty_1} + \mathbf{ty_2} \to \mathtt{U} & \Gamma; \Theta \vdash f_1 \Leftarrow ([x : ty_1]\ F\ (\mathtt{inl}\ x))_{\Gamma} \\ \Gamma; \Theta \vdash t \Rightarrow [\varphi](\mathbf{ty_1} + \mathbf{ty_2}) & \Gamma; \Theta \vdash f_2 \Leftarrow ([x : ty_2]\ F\ (\mathtt{inr}\ x))_{\Gamma} \end{array}}{\Gamma; \Theta \vdash \mathtt{split}\ F\ f_1\ f_2\ t \Rightarrow [\varphi](F\ t)_{\Gamma}} \ (x = \mathrm{NEWVAR}(\Gamma))$$
$$\text{(Split-Partial)}$$

$$\frac{\begin{array}{ll}\Gamma \vdash \varphi : \mathbb{F} & \Gamma; \Theta \vdash u \Leftarrow ([x : \mathbb{I}][\varphi]\ F\ x)_\Gamma \quad \Gamma; \Theta \vdash i_0 \Leftarrow \mathbb{I} \\ \Gamma; \Theta \vdash F \Leftarrow \mathbb{I} \to \mathtt{U} \quad \Gamma; \Theta \vdash b \Leftarrow ([\varphi \mapsto u\ i_0]\ F\ i_0)_\Gamma \quad \Gamma; \Theta \vdash i \Leftarrow \mathbb{I}\end{array}}{\Gamma; \Theta \vdash \mathtt{comp}\ F\ \varphi\ i_0\ u\ b\ i \Rightarrow ([i = i_0 \mapsto b\ |\ \varphi \mapsto u\ i]\ F\ i)_\Gamma} \ (x = \textsc{newVar}(\Gamma))$$

$$\text{(Comp)}$$

$$\frac{}{\Gamma; \Theta \vdash \mathtt{N} \Rightarrow \mathtt{U}} \qquad \text{(Nat)}$$

$$\frac{}{\Gamma; \Theta \vdash \mathtt{Z} \Rightarrow \mathtt{N}} \qquad \text{(Zero)}$$

$$\frac{\Gamma; \Theta \vdash n \Leftarrow \mathtt{N}}{\Gamma; \Theta \vdash \mathtt{S}\ n \Rightarrow \mathtt{N}} \qquad \text{(Succ)}$$

$$\frac{\begin{array}{ll}\Gamma; \Theta \vdash F \Leftarrow \mathtt{N} \to \mathtt{U} & \Gamma; \Theta \vdash c_0 \Leftarrow \textsc{app}((F)_\Gamma, \mathtt{Z}) \\ \Gamma; \Theta \vdash x \Leftarrow \mathtt{N} & \Gamma; \Theta \vdash c_s \Leftarrow ([m : \mathtt{N}]\ F\ m \to F\ (\mathtt{S}\ m))_\Gamma\end{array}}{\Gamma; \Theta \vdash \mathtt{ind}\ F\ c_0\ c_s\ x \Rightarrow (F\ x)_\Gamma} \ (m = \textsc{newVar}(\Gamma))$$

$$\text{(Ind)}$$

$$\frac{\begin{array}{ll}\Gamma; \Theta \vdash F \Leftarrow \mathtt{N} \to U & \Gamma; \Theta \vdash c_0 \Leftarrow \textsc{app}((F)_\Gamma, \mathtt{Z}) \\ \Gamma; \Theta \vdash x \Leftarrow [\vec{\psi} \mapsto \vec{\mathbf{m}}]\mathtt{N} & \Gamma; \Theta \vdash c_s \Leftarrow ([m : \mathtt{N}]\ F\ m \to F\ (\mathtt{S}\ m))_\Gamma\end{array}}{\Gamma; \Theta \vdash \mathtt{ind}\ F\ c_0\ c_s\ x \Rightarrow [\vec{\psi} \mapsto (F)_\Gamma\ \vec{\mathbf{m}}](F\ x)_\Gamma} \ (m = \textsc{newVar}(\Gamma))$$

$$\text{(Ind-Restr)}$$

$$\frac{\begin{array}{ll}\Gamma; \Theta \vdash F \Leftarrow \mathtt{N} \to U & \Gamma; \Theta \vdash c_0 \Leftarrow \textsc{app}((F)_\Gamma, \mathtt{Z}) \\ \Gamma; \Theta \vdash x \Leftarrow [\varphi]\mathtt{N} & \Gamma; \Theta \vdash c_s \Leftarrow ([m : \mathtt{N}]\ F\ m \to F\ (\mathtt{S}\ m))_\Gamma\end{array}}{\Gamma; \Theta \vdash \mathtt{ind}\ F\ c_0\ c_s\ x \Rightarrow [\varphi](F\ x)_\Gamma} \ (m = \textsc{newVar}(\Gamma))$$

$$\text{(Ind-Partial)}$$

$$\frac{}{\Gamma; \Theta \vdash \mathtt{0} \Rightarrow \mathbb{I}} \qquad \text{(Interval-0)}$$

$$\frac{}{\Gamma; \Theta \vdash \mathtt{1} \Rightarrow \mathbb{I}} \qquad \text{(Interval-1)}$$

Type checking

$$\frac{\Gamma \vdash ty \Leftarrow \mathtt{U} \quad \Gamma, x : ty \vdash e \Leftarrow \mathtt{U}}{\Gamma \vdash [x : ty]e \Leftarrow \mathtt{U}} \tag{Pi}$$

$$\frac{\Gamma, x : \mathbb{I} \vdash e \Leftarrow \mathtt{U}}{\Gamma \vdash [x : \mathbb{I}]e \Leftarrow \mathtt{U}} \tag{Pi-$\mathbb{I}$}$$

$$\frac{\Gamma \vdash ty \Leftarrow \mathtt{U} \quad \Gamma, x : ty \vdash e \Leftarrow \mathtt{U}}{\Gamma \vdash \langle x : ty \rangle e \Leftarrow \mathtt{U}} \tag{Sigma}$$

$$\frac{\begin{array}{c} \Gamma; \Theta \vdash ty \Leftarrow \mathtt{U} \qquad (ty)_\Gamma \sim_{\tau(\Gamma)} (ty_1)_{\Gamma_1} \\ \Gamma, x : ty, x \mapsto v; \Theta \vdash e \Leftarrow \mathrm{APP}(\mathsf{closure}([x_1 : ty_1]e_1, \Gamma_1)\rangle, v^{(ty_1)_{\Gamma_1}}) \end{array}}{\Gamma; \Theta \vdash [x : ty]e \Leftarrow \mathsf{closure}([x_1 : ty_1]e_1, \Gamma_1)} \; (v = \mathrm{NEWVAR}(\Gamma_1, x)) \tag{Abstraction}$$

$$\frac{\Gamma; \Theta \vdash s \Leftarrow (ty)_{\Gamma'} \qquad \Gamma; \Theta \vdash t \Leftarrow (e)_{\Gamma', x \mapsto (s)}}{\Gamma; \Theta \vdash (s, t) \Leftarrow \mathsf{closure}(\langle x : ty \rangle e, \Gamma')} \tag{Pair}$$

$$\frac{\Gamma; \Theta \vdash A \Leftarrow \mathtt{U} \qquad \Gamma; \Theta \vdash B \Leftarrow \mathtt{U}}{\Gamma; \Theta \vdash A + B \Leftarrow \mathtt{U}} \tag{Sum}$$

$$\frac{\Gamma; \Theta \vdash t_1 \Leftarrow \mathbf{A}}{\Gamma; \Theta \vdash \mathtt{inl}\ t_1 \Leftarrow \mathbf{A} + \mathbf{B}} \tag{Inl}$$

$$\frac{\Gamma; \Theta \vdash t_2 \Leftarrow \mathbf{B}}{\Gamma; \Theta \vdash \mathtt{inr}\ t_2 \Leftarrow \mathbf{A} + \mathbf{B}} \tag{Inr}$$

$$\frac{\Gamma \vdash \psi_i : \mathbb{F} \quad \varphi \sim^\Theta (\psi_1 \vee \cdots \vee \psi_n) \quad \Gamma; \Theta, \psi_i \vdash t_i \Leftarrow \mathbf{A} \quad (t_i)_\Gamma \sim_{\tau(\Gamma)}^{\Theta, \psi_i \wedge \psi_j} (t_j)_\Gamma}{\Gamma \vdash [\psi_1 \mapsto t_1, \ldots, \psi_n \mapsto t_n] \Leftarrow [\varphi]\mathbf{A}} \; (1 \le i, j \le n) \tag{System}$$

$$\frac{\Gamma \vdash \psi_i : \mathbb{F} \quad \Gamma; \Theta \vdash A \Leftarrow \mathtt{U}}{\Gamma; \Theta \vdash [\psi_1 \vee \cdots \vee \psi_n]A \Leftarrow \mathtt{U}} \; (1 \le i \le n) \tag{Partial}$$

$$\frac{\Gamma;\Theta \vdash A \Leftarrow \mathtt{U} \quad \Gamma \vdash \psi_i : \mathbb{F} \quad \Gamma;\Theta,\psi_i \vdash t_i \Leftarrow (A)_\Gamma}{\Gamma;\Theta \vdash [\psi_1 \mapsto t_1, \ldots, \psi_n \mapsto t_n]A \Leftarrow \mathtt{U}} \; (1 \le i \le n) \qquad \text{(Restr-U)}$$

$$\frac{\Gamma;\Theta \vdash e \Leftarrow \mathbf{A} \quad (e)_\Gamma \sim^{\Theta,\psi_i}_{\tau(\Gamma)} \mathbf{t}_i}{\Gamma;\Theta \vdash e \Leftarrow [\psi_1 \mapsto \mathbf{t}_1, \ldots, \psi_n \mapsto \mathbf{t}_n]\mathbf{A}} \; (1 \le i \le n) \qquad \text{(Restr)}$$

$$\frac{\Gamma \vdash e \Rightarrow [\psi']\mathbf{A}' \quad \psi \le^\Theta \psi' \quad \mathbf{A} \sim^\Theta_{\tau(\Gamma)} \mathbf{A}'}{\Gamma \vdash e \Leftarrow [\psi]\mathbf{A}} \qquad \text{(Weak-Partial)}$$

$$\frac{\Gamma \vdash e \Rightarrow [\vec{\psi'} \mapsto \vec{\mathbf{t}'}]\mathbf{A}' \quad [\vec{\psi} \mapsto \vec{\mathbf{t}}] \sim^{\Theta,\psi_1 \vee \cdots \vee \psi_n}_{\tau(\Gamma)} [\vec{\psi'} \mapsto \vec{\mathbf{t}'}] \quad \mathbf{A} \sim^\Theta_{\tau(\Gamma)} \mathbf{A}'}{\Gamma \vdash e \Leftarrow [\vec{\psi} \mapsto \vec{\mathbf{t}}]\mathbf{A}}$$
$$\text{(Weak-Restr)}$$

$$\frac{\Gamma;\Theta \vdash e \Rightarrow \mathbf{A} \quad \mathbf{A} \sim^\Theta_{\tau(\Gamma)} \mathbf{A}'}{\Gamma;\Theta \vdash e \Leftarrow \mathbf{A}'} \qquad \text{(Infer)}$$

1.4   Evaluation

We now extend evaluation, first presented in section 1.1.4, to handle the new cubical constructs.

We denote with $(\varphi)_\Gamma$ the substitution in the formula φ of the bindings stored in Γ,[21] which is thought as the evaluation of the formula φ.

$$(\mathbb{I})_\Gamma = \mathbb{I}$$
$$(\mathtt{0})_\Gamma = \mathtt{0}$$
$$(\mathtt{1})_\Gamma = \mathtt{1}$$
$$([\psi_1 \mapsto M_1, \ldots, \psi_n \mapsto M_n])_\Gamma = \text{SIMPLIFY}_{()}\big([(\psi_1)_\Gamma \mapsto (M_1)_\Gamma, \ldots, (\psi_n)_\Gamma \mapsto (M_n)_\Gamma]\big)$$
$$([\varphi]A)_\Gamma = \text{FOLDPARTIAL}\big([(\varphi)_\Gamma](A)_\Gamma\big)$$
$$([\psi_1 \mapsto M_1, \ldots, \psi_n \mapsto M_n]A)_\Gamma = \text{FOLDRESTR}\big([(\psi_1)_\Gamma \mapsto (M_1)_\Gamma, \ldots, (\psi_n)_\Gamma \mapsto (M_n)_\Gamma](A)_\Gamma\big)$$

---

[21]This is used also for interval variables renaming, which is needed since, for example, if $q : \mathtt{Path}\ A\ b\ c \equiv [i:I][i=0 \mapsto b, i=1 \mapsto c]A$, then $q\ j$ should be annotated with the type $[j=0 \mapsto b, j=1 \mapsto c]A$.

$$(\texttt{comp}\ F\ \varphi\ i_0\ u\ b\ i)_\Gamma = \text{COMP}\big(\Gamma, F, \varphi, i_0, u, b, i\big)$$

The evaluation function uses some helpers:

- SIMPLIFY applied to a system returns the value corresponding to a true formula, if there is one, otherwise it does nothing.[22] The directions environment Θ is not used during the evaluation, but we give a more general definition with it since we will use the procedure SIMPLIFY later too and we will need Θ there.

$$\text{SIMPLIFY}_\Theta\big([\psi_1 \mapsto \mathbf{t}_1, \ldots, \psi_n \mapsto \mathbf{t}_n]\big) = \mathbf{t}_i \qquad (\textit{if } \psi_i \textit{ is true in } \Theta)$$
$$\text{SIMPLIFY}_\Theta\big(\mathbf{v}\big) = \mathbf{v} \qquad\qquad\qquad (\textit{otherwise})$$

  One may expect SIMPLIFY to be recursive, which would not be wrong; we choose not to define it recursively, i.e. calling $\text{SIMPLIFY}_\Theta(\mathbf{t}_i)$ instead of just returning $\mathbf{t}_i$, as SIMPLIFY, except this unnecessary case in evaluation, is used only during conversion (see section 1.5) and recursion already happens there.

- The routine FOLDRESTR collapses all the terms or values the form

$$[\vec{\psi_1} \mapsto \vec{M_1}] \ldots [\vec{\psi_n} \mapsto \vec{M_n}]A,$$

  where A is not a restriction type, to

$$[\vec{\psi_1} \mapsto \vec{M_1}, \ldots, \vec{\psi_n} \mapsto \vec{M_n}]A.$$

  The routine FOLDPARTIAL works analogously. This simplifies the type-checker since we put restriction and partial types in a kind of normal form, so that we don't need explicit rules to handle this problem.

- SIMPLIFY-NV gets as input a neutral value, and if its type is a restriction type with a true formula, then the corresponding value is returned, otherwise the value is left untouched. Note that SIMPLIFY-NV does not depend on the directions environment, and indeed SIMPLIFY-NV is used solely during evaluation.

$$\text{SIMPLIFY-NV}\big(\mathbf{v}^{[\psi_1 \mapsto \mathbf{t}_1, \ldots, \psi_n \mapsto \mathbf{t}_n]\mathbf{A}}\big) = \mathbf{t}_i \qquad (\textit{if } \psi_i \textit{ is true})$$
$$\text{SIMPLIFY-NV}\big(\mathbf{v}\big) = \mathbf{v} \qquad\qquad\qquad (\textit{otherwise})$$

---

[22] The type-checking assures that if more than a formula is true, then the corresponding values are convertible one to each other.

We now have to revisit the helpers introduced in 1.1.4, handling the possibility that the argument is a neutral value with a partial or restriction type, which requires great care.

- APP applied to $\mathbf{f}$ and $\mathbf{a}$ first checks if $\mathbf{f}$ is a closure, in which case it evaluates it, otherwise $\mathbf{f}\ \mathbf{a}$ must be a neutral application which gets annotated with its type, handling the eventual partial or restriction type. The need for SIMPLIFY-NV is easily explained: suppose that $\mathbf{p}$ is neutral with type `Path` $A\ a\ b$, that is $[i : \mathbb{I}][i = 0 \mapsto \mathbf{a}, i = 1 \mapsto \mathbf{b}]\mathbf{A}$. Then the application $\text{APP}(\mathbf{p}, 0)$ shall reduce to $(\mathbf{p}\ 0)$ annotated with the type $[\texttt{True} \mapsto \mathbf{a}, \texttt{False} \mapsto \mathbf{b}]\mathbf{A}$, which has finally to be simplified to $\mathbf{a}$. When the function is neutral with a restriction type, e.g. $\mathbf{f}^{[\psi \mapsto \mathbf{g}](\mathbf{A} \to \mathbf{B})}$, then $\text{APP}(\mathbf{f}, \mathbf{a})$ shall reduce to $(\mathbf{f}\ \mathbf{a})$ annotated with the type $[\psi \mapsto \text{APP}(\mathbf{g}, a)]\mathbf{B}$; we invoke FOLDRESTR to handle the case where $\mathbf{B}$ is itself a restriction type.

  $$\text{APP}(\mathsf{closure}([x : ty]e, \Gamma), \mathbf{a}) = (e)_{\Gamma, x \mapsto \mathbf{a}}$$
  $$\text{APP}(\mathbf{f}^{\mathsf{closure}([x:ty]e,\Gamma)}, \mathbf{a}) = \text{SIMPLIFY-NV}\left((\mathbf{f}\ \mathbf{a})^{\text{APP}(\mathsf{closure}([x:ty]e,\Gamma),\mathbf{a})}\right)$$
  $$\text{APP}(\mathbf{f}^{[\vec{\psi} \mapsto \vec{\mathbf{g}}]\mathsf{closure}([x:ty]e,\Gamma)}, \mathbf{a}) = \text{SIMPLIFY-NV}\left((\mathbf{f}\ \mathbf{a})^{\text{FOLDRESTR}([\vec{\psi} \mapsto \text{APP}(\vec{\mathbf{g}},\mathbf{a})]\text{APP}(\mathsf{closure}([x:ty]e,\Gamma),\mathbf{a}))}\right)$$

  A drawback of our unified syntax for λ- and Π-abstractions is that the APP function must handle both, which may be a bit confusing to understand. We have then to add the case when a restricted neutral Π-type is applied to an argument.[23]

  $$\text{APP}([\vec{\psi} \mapsto \vec{\mathbf{g}}]\mathbf{f}, \mathbf{a}) = \text{FOLDRESTR}([\vec{\psi} \mapsto \text{APP}(\vec{\mathbf{g}}, \mathbf{a})]\text{APP}(\mathbf{f}, \mathbf{a}))$$

  Lastly we handle partial types, that is the cases of a system of functions, and of a neutral function with partial type.

  $$\text{APP}([\vec{\psi} \mapsto \vec{\mathbf{f}}], \mathbf{a}) = [\vec{\psi} \mapsto \text{APP}(\vec{\mathbf{f}}, \mathbf{a})]$$
  $$\text{APP}(\mathbf{f}^{[\varphi]\mathsf{closure}([x:ty]e,\Gamma)}, \mathbf{a}) = (\mathbf{f}\ \mathbf{a})^{\text{FOLDPARTIAL}([\varphi]\text{APP}(\mathsf{closure}([x:ty]e,\Gamma),\mathbf{a}))}$$

- IND evaluates an induction by pattern-matching on the argument n; if n is neutral, then the whole induction is neutral and so it gets annotated with its type. The partial and restriction type cases are handled analogously.

  $$\text{IND}(\mathbf{F}, \mathbf{c}_0, \mathbf{c}_s, \mathsf{Z}) = \mathbf{c}_0$$
  $$\text{IND}(\mathbf{F}, \mathbf{c}_0, \mathbf{c}_s, \mathsf{S}\ \mathbf{m}) = \text{APP}(\text{APP}(\mathbf{c}_s, \mathbf{m}), \text{IND}(\mathbf{F}, \mathbf{c}_0, \mathbf{c}_s, \mathbf{m}))$$
  $$\text{IND}(\mathbf{F}, \mathbf{c}_0, \mathbf{c}_s, \mathbf{n}^{\mathbb{N}}) = \text{SIMPLIFY-NV}\left((\mathsf{ind}\ \mathbf{F}\ \mathbf{c}_0\ \mathbf{c}_s\ \mathbf{n})^{\text{APP}(\mathbf{F},\mathbf{n})}\right)$$

$$\text{IND}\big(\mathbf{F}, \mathbf{c}_0, \mathbf{c}_s, \mathbf{n}^{[\vec{\psi} \mapsto \vec{\mathbf{m}}]\mathbb{N}}\big) = \text{SIMPLIFY-NV}\left(\big(\texttt{ind F } \mathbf{c}_0 \texttt{ } \mathbf{c}_s \texttt{ } \mathbf{n}\big)^{\text{FOLDRESTR}([\vec{\psi} \mapsto \text{APP}(\mathbf{F}, \vec{\mathbf{m}})]\text{APP}(\mathbf{F}, \mathbf{n}))}\right)$$

$$\text{IND}\big(\mathbf{F}, \mathbf{c}_0, \mathbf{c}_s, [\vec{\psi} \mapsto \vec{\mathbf{n}}]\big) = [\vec{\psi} \mapsto \text{IND}(\mathbf{F}, \mathbf{c}_0, \mathbf{c}_s, \vec{\mathbf{n}})]$$

$$\text{IND}\big(\mathbf{F}, \mathbf{c}_0, \mathbf{c}_s, \mathbf{n}^{[\varphi]\mathbb{N}}\big) = \text{IND}(\mathbf{F}, \mathbf{c}_0, \mathbf{c}_s, \mathbf{n})^{\text{FOLDPARTIAL}([\varphi]\text{APP}(\mathbf{F}, \mathbf{n}))}$$

- FST and SND return respectively the first and the second component of a pair, if the value is not neutral; otherwise it annotates the whole value with the corresponding type, handling the eventual partial or restriction type. Similarly to APP, the case of neutral values requires some explanation: suppose that $\mathbf{v}$ is neutral with type $([\psi_1 \mapsto \mathbf{a}]A) * ([\psi_2 \mapsto \mathbf{b}]B)$, where ψ_1 is true. Then $\text{FST}(\mathbf{v})$ should be neutral with type $([\psi_1 \mapsto \mathbf{a}]A)$ and so the whole value shall reduce to $\mathbf{a}$; this is why we need to invoke SIMPLIFY-NV. Similarly for $\text{FST}(\mathbf{v})$ when ψ_2 is true. When $\mathbf{v}$ has a restricted type, e.g. $[\vec{\psi} \mapsto \vec{\mathbf{w}}](\mathbf{A} * \mathbf{B})$ for simplicity, then $\text{FST}(\mathbf{v})$ shall have type $[\vec{\psi} \mapsto \text{FST}(\vec{\mathbf{w}})]\mathbf{A}$; we call FOLDRESTR as $\mathbf{A}$ may be a restriction type too, and finally we still invoke SIMPLIFY-NV for the same reason as the simpler (i.e. non-restricted) case.

$$\text{FST}\big((\mathbf{v}_1, \mathbf{v}_2)\big) = \mathbf{v}_1$$

$$\text{FST}\big(\mathbf{v}^{\text{closure}(\langle x:ty \rangle e, \Gamma)}\big) = \text{SIMPLIFY-NV}\left(\big(\mathbf{v}.1\big)^{(ty)_\Gamma}\right)$$

$$\text{FST}\big(\mathbf{v}^{[\vec{\psi} \mapsto \vec{\mathbf{w}}]\text{closure}(\langle x:ty \rangle e, \Gamma)}\big) = \text{SIMPLIFY-NV}\left(\big(\mathbf{v}.1\big)^{\text{FOLDRESTR}([\vec{\psi} \mapsto \text{FST}(\vec{\mathbf{w}})](ty)_\Gamma)}\right)$$

$$\text{FST}\big([\vec{\psi} \mapsto \vec{\mathbf{v}}]\big) = [\vec{\psi} \mapsto \text{FST}(\vec{\mathbf{v}})]$$

$$\text{FST}\big(\mathbf{v}^{[\varphi]\text{closure}(\langle x:ty \rangle e, \Gamma)}\big) = \big(\mathbf{v}.1\big)^{\text{FOLDPARTIAL}([\varphi](ty)_\Gamma)}$$

$$\text{SND}\big((\mathbf{v}_1, \mathbf{v}_2)\big) = \mathbf{v}_2$$

$$\text{SND}\big(\mathbf{v}^{\text{closure}(\langle x:ty \rangle e, \Gamma)}\big) = \text{SIMPLIFY-NV}\left(\big(\mathbf{v}.2\big)^{(e)_{\Gamma, x \mapsto \text{FST}(\mathbf{v})}}\right)$$

$$\text{SND}\big(\mathbf{v}^{[\vec{\psi} \mapsto \vec{\mathbf{w}}]\text{closure}(\langle x:ty \rangle e, \Gamma)}\big) = \text{SIMPLIFY-NV}\left(\big(\mathbf{v}.2\big)^{\text{FOLDRESTR}([\vec{\psi} \mapsto \text{SND}(\vec{\mathbf{w}})](e)_{\Gamma, x \mapsto \text{FST}(\mathbf{v})})}\right)$$

$$\text{SND}\big([\vec{\psi} \mapsto \vec{\mathbf{v}}]\big) = [\vec{\psi} \mapsto \text{SND}(\vec{\mathbf{v}})]$$

$$\text{SND}\big(\mathbf{v}^{[\varphi]\text{closure}(\langle x:ty \rangle e, \Gamma)}\big) = \big(\mathbf{v}.2\big)^{\text{FOLDPARTIAL}([\varphi](e)_{\Gamma, x \mapsto \text{FST}(\mathbf{v})})}$$

- SPLIT does by-case analysis on the given argument, if not neutral, otherwise the whole term becomes neutral, handling the neutral cases analogously to the previous helpers.

$$\text{SPLIT}(\mathbf{F}, \mathbf{f}_1, \mathbf{f}_2, \texttt{inl } \mathbf{v}_1) = \text{APP}(\mathbf{f}_1, \mathbf{v}_1)$$

$$\text{SPLIT}(\mathbf{F}, \mathbf{f}_1, \mathbf{f}_2, \texttt{inr } \mathbf{v}_2) = \text{APP}(\mathbf{f}_2, \mathbf{v}_2)$$

$$\text{SPLIT}(\mathbf{F}, \mathbf{f}_1, \mathbf{f}_2, \mathbf{v^{A+B}}) = \text{SIMPLIFY-NV}\left((\texttt{split } \mathbf{F}\ \mathbf{f}_1\ \mathbf{f}_2\ \mathbf{v})^{\text{APP}(\mathbf{F},\mathbf{v})}\right)$$

$$\text{SPLIT}(\mathbf{F}, \mathbf{f}_1, \mathbf{f}_2, \mathbf{v}^{[\vec{\psi} \mapsto \vec{\mathbf{w}}]\mathbf{A+B}}) = \text{SIMPLIFY-NV}\left((\texttt{split } \mathbf{F}\ \mathbf{f}_1\ \mathbf{f}_2\ \mathbf{v})^{\text{FOLDRESTR}([\vec{\psi}\mapsto\text{APP}(\mathbf{F},\vec{\mathbf{w}})]\text{APP}(\mathbf{F},\mathbf{v}))}\right)$$

$$\text{SPLIT}(\mathbf{F}, \mathbf{f}_1, \mathbf{f}_2, [\vec{\psi} \mapsto \vec{\mathbf{v}}]) = [\vec{\psi} \mapsto \text{SPLIT}(\mathbf{F}, \mathbf{f}_1, \mathbf{f}_2, \vec{\mathbf{v}})]$$

$$\text{SPLIT}(\mathbf{F}, \mathbf{f}_1, \mathbf{f}_2, \mathbf{v}^{[\varphi](\mathbf{A+B})}) = (\texttt{split } \mathbf{F}\ \mathbf{f}_1\ \mathbf{f}_2\ \mathbf{v})^{\text{FOLDPARTIAL}([\varphi]\text{APP}(\mathbf{F},\mathbf{v}))}$$

- COMP is more complex than the previous helpers: the main problem with evaluating composition is that one must do pattern matching on the first argument F (the type family), which is inside an $\mathbb{I}$-abstraction, and that means that one must first introduce a fresh variable $v : \mathbb{I}$, evaluate $F\ v$ and then pattern match; to call COMP recursively, one must then read-back to get the new type family. This naive approach is inefficient because of this problem, but an efficient implementation has still to be engineered. Reporting the actual working of COMP would complicate tremendously the notation, hence we now give the rules for the various type formers only in an informal way (i.e. only syntactical). We refer to chapter 3 for the actual implementation. As already said, the definition is by structural induction of the type former F inside the $\mathbb{I}$-abstraction, i.e. $[v : \mathbb{I}]F$.

  - $F \equiv [x : A]B$ (with $A \neq \mathbb{I}$). Given $b : [\varphi \mapsto u\ i_0][x : A(v/i0)]B(v/i0)$, the result of the composition shall be the function

  $$A(v/i) \rightarrow [i = i_0 \mapsto b, \varphi \mapsto u\ i_0]B(x/y_i, v/i)$$
  $$y_i \mapsto \texttt{comp } ([v : I]B(x/\tilde{y}))\ \varphi\ i_0\ u'\ (b\ \tilde{y}_0)\ i$$

  where

  $$\tilde{y}_0 \equiv \texttt{comp } ([v : \mathbb{I}]A)\ (\texttt{False})\ i\ (\mathbb{I} \rightarrow [])\ y_i\ i_0$$
  $$\tilde{y} \equiv \texttt{comp } ([v : \mathbb{I}]A)\ (\texttt{False})\ i\ (\mathbb{I} \rightarrow [])\ y_i\ v$$

  u' is obtained by applying the functions to $\tilde{y}$ inside the abstraction. Note that what we do is first transporting y_i to get $\tilde{y}_0$ (the other end at i_0) and the whole filler $\tilde{y}$, so that we obtain a value $b\ \tilde{y}_0$ in $B(v/i0)$ and we can do transport in B.

  - $F \equiv [x : \mathbb{I}]B$. Given $b : [\varphi \mapsto u\ i_0][x : \mathbb{I}]B(v/i0)$, the result of the composition shall be the function

  $$\mathbb{I} \rightarrow [i = i_0 \mapsto b, \varphi \mapsto u\ i_0]B(x/y_i, v/i)$$
  $$x \mapsto \texttt{comp } ([v : \mathbb{I}]B)\ \varphi\ i_0\ u'\ (b\ x)\ i$$

  u' is obtained by applying the functions to x inside the abstraction.

– $F \equiv \langle x : A \rangle B$. Given $b : [\varphi \mapsto u \ i_0]\langle x : A(v/i0)\rangle B(v/i0)$, the result of composition shall be a pair (c_1, c_2), i.e. composition is done component-wise. We put:

$$\texttt{comp} \ ([v : \mathbb{I}]\langle x : A \rangle B) \ \varphi \ i_0 \ u \ b \ i \equiv (c_0, c_1)$$

with

$$c_1 \equiv \texttt{comp} \ ([v : \mathbb{I}]A) \ \varphi \ i_0 \ u_1 \ (b.1) \ i$$
$$\tilde{c}_1 \equiv \texttt{comp} \ ([v : \mathbb{I}]A) \ \varphi \ i_0 \ u_1 \ (b.1) \ v$$
$$c_2 \equiv \texttt{comp} \ ([v : \mathbb{I}]B(x/\tilde{c}_1)) \ \varphi \ i_0 \ u_2 \ (b.2) \ i$$

u_1 and u_2 are obtained from u by applying the projections inside the abstraction.

- $F \equiv A + B$. Given $b : [\varphi \mapsto u \ i_0]A(v/i0) + B(v/i0)$, the composition is done 'inside the injection' if the value is not neutral, otherwise the whole composition becomes neutral.

$$\texttt{comp} \ ([v : \mathbb{I}]A + B) \ \varphi \ i_0 \ u \ (\texttt{inl} \ b) \ i \equiv \texttt{inl} \ \big(\texttt{comp} \ ([v : \mathbb{I}]A) \ \varphi \ i_0 \ u' \ b \ i\big)$$
$$\texttt{comp} \ ([v : \mathbb{I}]A + B) \ \varphi \ i_0 \ u \ (\texttt{inr} \ b) \ i \equiv \texttt{inr} \ \big(\texttt{comp} \ ([v : \mathbb{I}]B) \ \varphi \ i_0 \ u'' \ b \ i\big)$$

u' and u'' are obtained from u by removing respectively the left and right injection inside the abstraction.[24]

- $F \equiv \texttt{N}$. Given $b : [\varphi \mapsto u \ i_0]\texttt{N}$, the composition is done by induction if the value is not neutral, otherwise the whole composition becomes neutral.

$$\texttt{comp} \ (\mathbb{I} \to \texttt{N}) \ \varphi \ i_0 \ u \ \texttt{Z} \ i \equiv \texttt{Z}$$
$$\texttt{comp} \ (\mathbb{I} \to \texttt{N}) \ \varphi \ i_0 \ u \ (\texttt{S} \ b) \ i \equiv \texttt{S} \ \big(\texttt{comp} \ (\mathbb{I} \to \texttt{N}) \ \varphi \ i_0 \ u' \ b \ i\big)$$

u' is obtained from u by removing the left-most $\texttt{S}$ inside the abstraction.[25]

- $F \equiv [\vec{\psi} \mapsto \vec{w}]A$ (with $i \notin \texttt{vars}(\vec{\psi})$). The constraint on the formula requires some explanation: if we allowed it, then by considering the family $F \equiv [i = 0 \wedge j = 0 \mapsto \texttt{true}, i = 0 \wedge j = 1 \mapsto \texttt{false}]\texttt{bool}$, we could transport $\texttt{true}$ from $F(i/0) \equiv \texttt{bool}$ to $F(i/1) \equiv \bot$, which is inconsistent.

Therefore we put:

$$\texttt{comp} \ ([v : \mathbb{I}]\langle x : A \rangle B) \ \varphi \ i_0 \ u \ b \ i$$
$$\equiv \texttt{comp} \ ([v : \mathbb{I}] \ A) \ (\varphi \vee \vec{\psi}) \ i_0 \ [i : \mathbb{I}][\varphi \mapsto u \ i, \vec{\psi} \mapsto \vec{w}(v/i)] \ b \ i$$

---

[24]If the base point is of the form $\texttt{inl} \ b$, then the body of u shall be a system of the form $[\vec{\psi} \mapsto \texttt{inl} \ \vec{v}]$, due to type-checking. Similarly for $\texttt{inr}$.

[25]u has to be of that form, for the same reason (that is, type-checking) explained for coproducts.

Using the above computational rules for composition one may easily derive the rule for path types as given in [CCHM15], i.e.

$$\texttt{comp}([v : \mathbb{I}] \texttt{ Path } A \ w_1 \ w_2) \ \varphi \ i_0 \ u \ p_0 \ i$$
$$\equiv [x : \mathbb{I}] \texttt{ comp}([v : \mathbb{I}] \ A) \ (\varphi \vee x = \texttt{0} \vee x = \texttt{1}) \ i_0$$
$$[j : \mathbb{I}][\varphi \mapsto u \ j \ x, x = \texttt{0} \mapsto w_1(v/j), x = \texttt{1} \mapsto w_2(v/j)] \ (p_0 \ x) \ i$$

Reading back

We now extend straightforwardly the read-back function $\mathcal{R}$ introduced in section 1.1.4 to handle the new syntactical constructs.

$$\mathcal{R}_{ns}(\mathbb{I}) = \mathbb{I}$$
$$\mathcal{R}_{ns}(\texttt{0}) = \texttt{0}$$
$$\mathcal{R}_{ns}(\texttt{1}) = \texttt{1}$$
$$\mathcal{R}_{ns}([\psi_1 \mapsto M_1, \ldots, \psi_n \mapsto M_n]) = [\psi_1 \mapsto \mathcal{R}_{ns}(M_1), \ldots, \psi_n \mapsto \mathcal{R}_{ns}(M_n)]$$
$$\mathcal{R}_{ns}([\varphi]A) = \textsc{foldPartial}\big([\varphi](\mathcal{R}_{ns}(A))\big)$$
$$\mathcal{R}_{ns}([\psi_1 \mapsto M_1, \ldots, \psi_n \mapsto M_n]A) = \textsc{foldRestr}\big([\psi_1 \mapsto \mathcal{R}_{ns}(M_1), \ldots, \psi_n \mapsto \mathcal{R}_{ns}(M_n)](\mathcal{R}_{ns}(A))\big)$$
$$\mathcal{R}_{ns}(\texttt{comp } F \ \varphi \ i_0 \ u \ b \ i) = \texttt{comp } \mathcal{R}_{ns}(F) \ \varphi \ \mathcal{R}_{ns}(i_0) \ \mathcal{R}_{ns}(u) \ \mathcal{R}_{ns}(b) \ \mathcal{R}_{ns}(i)$$

1.5 $\alpha\eta$-conversion

The function $\cdot \sim_{ns}^{\Theta} \cdot$ tests α- and η-conversion between two values, given a directions environment Θ; it keeps track of the list of already used names, to avoid conflicts when reading back closures.

The rules assume that the two values being tested have the same type.

Non-neutral values

$$\overline{\texttt{U} \sim_{ns}^{\Theta} \texttt{U}}$$

$$\frac{(ty)_\Gamma \sim_{ns}^{\Theta} (ty')_{\Gamma'} \quad (e)_{\Gamma, x \mapsto v^{(ty)_\Gamma}} \sim_{ns,v}^{\Theta} (e')_{\Gamma', y \mapsto v^{(ty')_{\Gamma'}}}}{\textsf{closure}([x : ty]e, \Gamma) \sim_{ns}^{\Theta} \textsf{closure}([y : ty']e', \Gamma')} \ (v = \textsc{newVar}(ns \texttt{++} \Gamma', x))$$

$$\frac{(e)_{\Gamma, x \mapsto v^{(ty)_\Gamma}} \sim_{ns,v}^{\Theta} \textsc{app}(\textsc{simplify}_\Theta(\mathbf{v}_2), v^{(ty)_\Gamma})}{\textsf{closure}([x : ty]e, \Gamma) \sim_{ns}^{\Theta} \mathbf{v}_2} \ (v = \textsc{newVar}(ns, x))$$

$$\frac{\text{APP}\big(\text{SIMPLIFY}_\Theta(\mathbf{v}_1), v^{(ty)_\Gamma}\big) \sim_{ns,v}^\Theta (e)_{\Gamma, x \mapsto v^{(ty)_\Gamma}}}{\mathbf{v}_1 \sim_{ns}^\Theta \mathsf{closure}([x:ty]e, \Gamma)} \ (v = \text{NEWVAR}(ns, x))$$

$$\frac{(ty)_\Gamma \sim_{ns}^\Theta (ty')_{\Gamma'} \quad (e)_{\Gamma, x \mapsto v^{(ty)_\Gamma}} \sim_{ns,v}^\Theta (e')_{\Gamma', y \mapsto v^{(ty')_{\Gamma'}}}}{\mathsf{closure}(\langle x:ty \rangle e, \Gamma) \sim_{ns}^\Theta \mathsf{closure}(\langle y:ty' \rangle e', \Gamma')} \ (v = \text{NEWVAR}(ns{+}{+}\Gamma', x))$$

$$\frac{\mathbf{s} \sim_{ns}^\Theta \mathbf{t}}{\mathbf{s}.1 \sim_{ns}^\Theta \mathbf{t}.1}$$

$$\frac{\mathbf{s} \sim_{ns}^\Theta \mathbf{t}}{\mathbf{s}.2 \sim_{ns}^\Theta \mathbf{t}.2}$$

$$\frac{\mathbf{s} \sim_{ns}^\Theta \mathbf{s}' \quad \mathbf{t} \sim_{ns}^\Theta \mathbf{t}'}{(\mathbf{s}, \mathbf{t}) \sim_{ns}^\Theta (\mathbf{s}', \mathbf{t}')}$$

$$\frac{\text{FST}\big(\text{SIMPLIFY}_\Theta(\mathbf{v})\big) \sim_{ns}^\Theta \mathbf{s} \quad \text{SND}\big(\text{SIMPLIFY}_\Theta(\mathbf{v})\big) \sim_{ns}^\Theta \mathbf{t}}{\mathbf{v} \sim_{ns}^\Theta (\mathbf{s}, \mathbf{t})}$$

$$\frac{\mathbf{s} \sim_{ns}^\Theta \text{FST}\big(\text{SIMPLIFY}_\Theta(\mathbf{v})\big) \quad \mathbf{t} \sim_{ns}^\Theta \text{SND}\big(\text{SIMPLIFY}_\Theta(\mathbf{v})\big)}{(\mathbf{s}, \mathbf{t}) \sim_{ns}^\Theta \mathbf{v}}$$

$$\frac{\mathbf{A} \sim_{ns}^\Theta \mathbf{A}' \quad \mathbf{B} \sim_{ns}^\Theta \mathbf{B}'}{\mathbf{A} + \mathbf{B} \sim_{ns}^\Theta \mathbf{A}' + \mathbf{B}'}$$

$$\frac{\mathbf{v} \sim_{ns}^\Theta \mathbf{v}'}{\mathtt{inl} \ \mathbf{v} \sim_{ns}^\Theta \mathtt{inl} \ \mathbf{v}'}$$

$$\frac{\mathbf{v} \sim_{ns}^\Theta \mathbf{v}'}{\mathtt{inr} \ \mathbf{v} \sim_{ns}^\Theta \mathtt{inr} \ \mathbf{v}'}$$

$$\frac{}{\mathtt{N} \sim^\Theta_{ns} \mathtt{N}} \qquad \frac{}{\mathtt{Z} \sim^\Theta_{ns} \mathtt{Z}} \qquad \frac{\mathbf{n} \sim^\Theta_{ns} \mathbf{n}'}{\mathtt{S}\ \mathbf{n} \sim^\Theta_{ns} \mathtt{S}\ \mathbf{n}'}$$

$$\frac{}{\mathbb{I} \sim^\Theta_{ns} \mathbb{I}} \qquad \frac{}{0 \sim^\Theta_{ns} 0} \qquad \frac{}{1 \sim^\Theta_{ns} 1}$$

$$\frac{\psi_i \sim^\Theta \mathtt{True} \quad \mathbf{t}_i \sim^\Theta_{ns} \mathbf{v}}{[\psi_1 \mapsto \mathbf{t}_1, \ldots, \psi_n \mapsto \mathbf{t}_n] \sim^\Theta_{ns} \mathbf{v}} \qquad \frac{\psi_i \sim^\Theta \mathtt{True} \quad \mathbf{v} \sim^\Theta_{ns} \mathbf{t}_i}{\mathbf{v} \sim^\Theta_{ns} [\psi_1 \mapsto \mathbf{t}_1, \ldots, \psi_n \mapsto \mathbf{t}_n]}$$

$$\frac{(\psi_1 \vee \cdots \vee \psi_n) \sim^\Theta (\psi'_1 \vee \cdots \vee \psi'_n) \quad \mathbf{t}_i \sim^{\Theta, \psi_i \wedge \psi'_j}_{ns} \mathbf{t}'_j}{[\psi_1 \mapsto \mathbf{t}_1, \ldots, \psi_n \mapsto \mathbf{t}_n] \sim^\Theta_{ns} [\psi'_1 \mapsto \mathbf{t}'_1, \ldots, \psi'_m \mapsto \mathbf{t}'_m]}\ (1 \leq i \leq n, 1 \leq j \leq m)$$

$$\frac{\varphi \sim^\Theta \varphi' \quad \mathbf{A} \sim^\Theta_{ns} \mathbf{A}'}{[\varphi]\mathbf{A} \sim^\Theta_{ns} [\varphi']\mathbf{A}'}$$

$$\frac{[\psi_1 \mapsto \mathbf{t}_1, \ldots, \psi_n \mapsto \mathbf{t}_n] \sim^\Theta_{ns} [\psi'_1 \mapsto \mathbf{t}'_1, \ldots, \psi'_m \mapsto \mathbf{t}'_m] \quad \mathbf{A} \sim^\Theta_{ns} \mathbf{A}'}{[\psi_1 \mapsto \mathbf{t}_1, \ldots, \psi_n \mapsto \mathbf{t}_n]\mathbf{A} \sim^\Theta_{ns} [\psi'_1 \mapsto \mathbf{t}'_1, \ldots, \psi'_m \mapsto \mathbf{t}'_m]\mathbf{A}'}$$

Neutral values

$$\frac{}{\mathbf{x} \sim^\Theta_{ns} \mathbf{x}}$$

$$\frac{(x = y) \sim^\Theta \mathtt{True}}{\mathbf{x}^\mathbb{I} \sim^\Theta_{ns} \mathbf{y}^\mathbb{I}} \quad \frac{(x = 0) \sim^\Theta \mathtt{True}}{\mathbf{x}^\mathbb{I} \sim^\Theta_{ns} 0} \quad \frac{(x = 1) \sim^\Theta \mathtt{True}}{\mathbf{x}^\mathbb{I} \sim^\Theta_{ns} 1} \quad \frac{(x = 0) \sim^\Theta \mathtt{True}}{0 \sim^\Theta_{ns} \mathbf{x}^\mathbb{I}} \quad \frac{(x = 1) \sim^\Theta \mathtt{True}}{1 \sim^\Theta_{ns} \mathbf{x}^\mathbb{I}}$$

$$\frac{\mathbf{f} \sim^\Theta_{ns} \mathbf{f}' \quad \mathbf{a} \sim^\Theta_{ns} \mathbf{a}'}{\mathbf{f}\ \mathbf{a} \sim^\Theta_{ns} \mathbf{f}'\ \mathbf{a}'}$$

$$\frac{\mathbf{F} \sim^\Theta_{ns} \mathbf{F}' \quad \mathbf{c}_0 \sim^\Theta_{ns} \mathbf{c}'_0 \quad \mathbf{c}_s \sim^\Theta_{ns} \mathbf{c}'_s \quad \mathbf{n} \sim^\Theta_{ns} \mathbf{n}'}{\mathtt{ind}\ \mathbf{F}\ \mathbf{c}_0\ \mathbf{c}_s\ \mathbf{n} \sim^\Theta_{ns} \mathtt{ind}\ \mathbf{F}'\ \mathbf{c}'_0\ \mathbf{c}'_s\ \mathbf{n}'}$$

$$\frac{\mathbf{ty} \sim_{ns}^{\Theta} \mathbf{ty}' \quad \mathbf{f}_1 \sim_{ns}^{\Theta} \mathbf{f}_1' \quad \mathbf{f}_2 \sim_{ns}^{\Theta} \mathbf{f}_2' \quad \mathbf{x} \sim_{ns}^{\Theta} \mathbf{x}'}{\texttt{split } \mathbf{ty} \ \mathbf{f}_1 \ \mathbf{f}_2 \ \mathbf{x} \sim_{ns}^{\Theta} \texttt{split } \mathbf{ty}' \ \mathbf{f}_1' \ \mathbf{f}_2' \ \mathbf{x}'}$$

$$\frac{\mathbf{F} \sim_{ns}^{\Theta} \mathbf{F}' \quad \varphi \sim^{\Theta} \varphi' \quad \mathbf{i}_0 \sim_{ns}^{\Theta} \mathbf{i}_0' \quad \mathbf{u} \sim_{ns}^{\Theta} \mathbf{u}' \quad \mathbf{b} \sim_{ns}^{\Theta} \mathbf{b}' \quad \mathbf{i} \sim_{ns}^{\Theta} \mathbf{i}'}{\texttt{comp } \mathbf{F} \ \varphi \ \mathbf{i}_0 \ \mathbf{u} \ \mathbf{b} \ \mathbf{i} \sim_{ns}^{\Theta} \texttt{comp } \mathbf{F}' \ \varphi' \ \mathbf{i}_0' \ \mathbf{u}' \ \mathbf{b}' \ \mathbf{i}'}$$

$$\frac{\psi_i \sim^{\Theta} \texttt{True} \quad \mathbf{t}_i \sim_{ns}^{\Theta} \mathbf{v}}{\mathbf{k}^{[\psi_1 \mapsto \mathbf{t}_1, \dots, \psi_n \mapsto \mathbf{t}_n]\mathbf{A}} \sim_{ns}^{\Theta} \mathbf{v}} \qquad \frac{\psi_i \sim^{\Theta} \texttt{True} \quad \mathbf{v} \sim_{ns}^{\Theta} \mathbf{t}_i}{\mathbf{v} \sim_{ns}^{\Theta} \mathbf{k}^{[\psi_1 \mapsto \mathbf{t}_1, \dots, \psi_n \mapsto \mathbf{t}_n]\mathbf{A}}}$$

$$\frac{\psi_i \not\sim^{\Theta} \texttt{True} \quad \psi_j' \not\sim^{\Theta} \texttt{True} \quad \mathbf{k} \sim_{ns}^{\Theta} \mathbf{k}'}{\mathbf{k}^{[\psi_1 \mapsto \mathbf{t}_1, \dots, \psi_n \mapsto \mathbf{t}_n]\mathbf{A}} \sim_{ns}^{\Theta} \mathbf{k}'^{[\psi_1' \mapsto \mathbf{t}_1', \dots, \psi_m' \mapsto \mathbf{t}_m']\mathbf{A}'}} \quad \begin{matrix}(1 \le i \le n)\\(1 \le j \le m)\end{matrix}$$

$$\frac{\textsc{proofIrrelevant}_{\Theta}(\mathbf{A})}{\mathbf{k}^{\mathbf{A}} \sim_{ns}^{\Theta} \mathbf{k}'^{\mathbf{A}'}}$$

Proof irrelevance

When two values have a type that can be simplified (i.e. a restriction type with a true formula), then both values shall reduce to the same value and so they are indeed $\alpha\eta$-equivalent.[26] We generalize this by introducing a predicate called $\textsc{proofIrrelevant}$: when we test $\alpha\eta$-equivalence between two neutral values, if their type is proof irrelevant it means that they shall reduce to the same value, and so they are automatically convertible (from this comes the name 'proof irrelevance', since we don't have to look inside the value/proof).

$$\frac{\psi_i \sim^{\Theta} \texttt{True}}{\textsc{proofIrrelevant}([\psi_1 \mapsto \mathbf{t}_1, \dots, \psi_n \mapsto \mathbf{t}_n]\mathbf{A})}$$

$$\frac{\textsc{proofIrrelevant}(\mathbf{A})}{\textsc{proofIrrelevant}([\psi_1 \mapsto \mathbf{t}_1, \dots, \psi_n \mapsto \mathbf{t}_n]\mathbf{A})}$$

$$\frac{\textsc{proofIrrelevant}((e)_{\Gamma, x \mapsto v})}{\textsc{proofIrrelevant}(\mathsf{closure}([x : ty]e, \Gamma))} \quad (v = \textsc{newVar}(\Gamma, x))$$

---

[26]This works because when we test $\alpha\eta$-equivalence, the two values being tested are assumed to have the same type.

$$\frac{\text{PROOFIRRELEVANT}((ty)_\Gamma) \quad \text{PROOFIRRELEVANT}((e)_{\Gamma, x \mapsto v})}{\text{PROOFIRRELEVANT}(\text{closure}(\langle x : ty \rangle e, \Gamma))} \; (v = \text{NEWVAR}(\Gamma, x))$$

1.6   Examples

We can now show some examples exploiting the full theory developed so far.

1.6.1   Equivalence of two product types

First of all, we show how to model the unit type, as promised in section 1.1.5. For the unit type we choose $[\texttt{True} \mapsto \texttt{Z}]\texttt{N}$, i.e. the subtype of naturals which 'contain' only $\texttt{Z}$. As we don't have a constant $\texttt{True}$ in the syntax, we introduce a dummy variable $\_i$ which we don't use anywhere else, so that $\texttt{True}$ can be replaced by $\_i = \_i$.[27]   The induction principle is then trivial to prove: if $x : \texttt{unit}$, then x is convertible to $\texttt{Z}$ and so C $\texttt{tt}$ is convertible to C x, which is the type of p. We also prove by induction the propositional uniqueness principle for $\texttt{unit}$.

$$
\begin{aligned}
&\_i : \mathbb{I} \\
\texttt{unit} : \texttt{U} &= [\_i = \_i \to \texttt{Z}]\texttt{N} \\
\texttt{tt} : \texttt{unit} &= \texttt{Z} \\
\texttt{unitInd} : &[C : \texttt{unit} \to \texttt{U}] \; C \; \texttt{tt} \to [x : \texttt{unit}] \; C \; x \\
&= [C : \texttt{unit} \to \texttt{U}][p : C \; \texttt{tt}][x : \texttt{unit}] \; p \\
\texttt{unitEq} : &[x : \texttt{unit}] \; \texttt{Path unit tt } x \\
&= [x : \texttt{unit}] \; \texttt{unitInd} \; ([y : \texttt{unit}] \; \texttt{Path unit tt } y) \; (\texttt{refl unit tt}) \; x
\end{aligned}
$$

As remarked in the introduction, we can prove function extensionality, even in the dependent case.[28]

$$
\begin{aligned}
\texttt{funext} : &[A, B : \texttt{U}][f, g : A \to B] \\
&([x : A] \; \texttt{Path } B \; (f \; x) \; (g \; x)) \to \texttt{Path } (A \to B) \; f \; g \\
&= [A, B : \texttt{U}][f, g : A \to B][p : [x : A] \; \texttt{Path } B \; (f \; x) \; (g \; x)] \\
&\quad [i : \mathbb{I}][x : A] \; (p \; x \; i)
\end{aligned}
$$

$$\texttt{dfunext} : [A : U][B : A \to \texttt{U}][f, g : [x : A] \; B \; x]$$

---

[27]We chose not to add the constant $\texttt{True}$ to the language, as it would be useful only in this case, and it would complicate the implementation.

$$([x : A] \text{ Path } (B \ x) \ (f \ x) \ (g \ x)) \to \text{Path } ([x : A] \ B \ x) \ f \ g$$
$$= [A : U][B : A \to U][f, g : [x : A] \ B \ x][p : [x : A] \text{ Path } (B \ x) \ (f \ x) \ (g \ x)]$$
$$[i : \mathbb{I}][x : A] \ (p \ x \ i)$$

The first use of composition is in proving *transport*,[29] which can be thought as the case of extension in a 1-cube, i.e. a structure with only two points. We first derive the function `transp`, which is very general, and then we use it to define transport in the usual formulation.

$$\text{transp} : [F : \mathbb{I} \to U] \ F \ 0 \to F \ 1$$
$$= [F : \mathbb{I} \to U][a : F \ 0] \text{ comp } F \ () \ 0 \ (\mathbb{I} \to []) \ a \ 1$$

$$\text{transport} : [B : U][P : B \to U][x, y : B] \text{ Path } B \ x \ y \to P \ x \to P \ y$$
$$= [B : U][P : B \to U][x, y : B][pB : \text{Path } B \ x \ y]$$
$$[u : P \ x] \text{ transp } ([i : \mathbb{I}] \ P \ (pB \ i)) \ u$$

Using (dependent) function extensionality, we can prove the propositional η-law and the induction/elimination principle for `prod` types (introduced in 1.1.5).

$$\text{prodEta} : [A, B : U][p : \text{prod } A \ B] \text{ Path } (\text{prod } A \ B)$$
$$(\text{couple } A \ B \ (\text{pi1 } A \ B \ p) \ (\text{pi2 } A \ B \ p)) \ p$$
$$= [A, B : U][p : \text{prod } A \ B] \text{ dfunext bool } ([x : \text{bool}] \text{ boolRec } U \ A \ B \ x)$$
$$(\text{couple } A \ B \ (\text{pi1 } A \ B \ p) \ (\text{pi2 } A \ B \ p)) \ p$$
$$(\text{boolInd } ([y : \text{bool}] \text{ Path } (\text{boolRec } U \ A \ B \ y)$$
$$(\text{couple } A \ B \ (\text{pi1 } A \ B \ p) \ (\text{pi2 } A \ B \ p) \ y) \ (p \ y))$$
$$(\text{refl } A \ (p \ \text{false})) \ (\text{refl } B \ (p \ \text{true})))$$

$$\text{prodInd} : [A, B : U][C : \text{prod } A \ B \to U]$$
$$([a : A][b : B] \ C \ (\text{couple } A \ B \ a \ b)) \to [p : \text{prod } A \ B] \ C \ p$$
$$= [A, B : U][C : \text{prod} AB \to U][f : [a : A][b : B] \ C \ (\text{couple } A \ B \ a \ b)]$$
$$= [p : \text{prod } A \ B] \text{ transport } (\text{prod } A \ B) \ C$$
$$(\text{couple } A \ B \ (\text{pi1 } A \ B \ p) \ (\text{pi2 } A \ B \ p)) \ p \ (\text{prodEta } A \ B \ p)$$
$$(f \ (\text{pi1 } A \ B \ p)(\text{pi2 } A \ B \ p))$$

We now want to prove that the built-in product type $A * B$ and the derived one `prod` $A \ B$ are equivalent, in the sense of [Uni13] 2.4.

$$\text{id} : [A : U] \ A \to A = [A : U][x : A] \ x$$

$$\mathtt{fcomp} : [A, B, C : \mathtt{U}][f : B \to C][g : A \to B]\ A \to C$$
$$= [A, B, C : \mathtt{U}][f : B \to C][g : A \to B][a : A]\ f\ (g\ a)$$

$$\mathtt{isEquiv} : [A, B : \mathtt{U}]\ (A \to B) \to U$$
$$= [A, B : \mathtt{U}][f : A \to B]$$
$$(\langle g : B \to A \rangle\ \mathtt{Path}\ (B \to B)\ (\mathtt{fcomp}\ B\ A\ B\ f\ g)\ (\mathtt{id}\ B))$$
$$* (\langle h : B \to A \rangle\ \mathtt{Path}\ (A \to A)\ (\mathtt{fcomp}\ A\ B\ A\ h\ f)\ (\mathtt{id}\ A))$$

$$\mathtt{equiv} : \mathtt{U} \to \mathtt{U} \to \mathtt{U} = [A, B : \mathtt{U}]\langle f : A \to B \rangle\ \mathtt{isEquiv}\ A\ B\ f$$

Now we define two functions $\mathtt{to}$ and $\mathtt{from}$, and then we prove that they are one the inverse of the other.

$$\mathtt{to} : [A, B : \mathtt{U}]\ A * B \to \mathtt{prod}\ A\ B$$
$$= [A, B : \mathtt{U}][p : A * B]\ \mathtt{couple}\ A\ B\ (p.1)\ (p.2)$$

$$\mathtt{from} : [A, B : \mathtt{U}]\ \mathtt{prod}\ A\ B \to A * B$$
$$= [A, B : \mathtt{U}][p : \mathtt{prod}\ A\ B]\ (\mathtt{pi1}\ A\ B\ p, \mathtt{pi2}\ A\ B\ p)$$

First we prove that applying $\mathtt{from}$ and then $\mathtt{to}$ is (propositionally) pointwise equal to the identity.

$$\mathtt{to\_from}' : [A, B : \mathtt{U}][p : \mathtt{prod}\ A\ B]\ \mathtt{Path}\ (\mathtt{prod}\ A\ B)\ (\mathtt{to}\ A\ B\ (\mathtt{from}\ A\ B\ p))\ p$$
$$= [A, B : \mathtt{U}]\ \mathtt{prodInd}\ A\ B$$
$$([p' : \mathtt{prod}\ A\ B]\ \mathtt{Path}\ (\mathtt{prod}\ A\ B)\ (\mathtt{to}\ A\ B\ (\mathtt{from}\ A\ B\ p'))\ p')$$
$$([a : A][b : B]\ \mathtt{refl}\ (\mathtt{prod}\ A\ B)\ (\mathtt{couple}\ A\ B\ a\ b))$$

And then we use function extensionality to get half of what we need to prove the aforementioned equivalence.

$$\mathtt{to\_from} : [A, B : \mathtt{U}]\ \mathtt{Path}\ (\mathtt{prod}\ A\ B \to \mathtt{prod}\ A\ B)$$
$$(\mathtt{fcomp}\ (\mathtt{prod}\ A\ B)\ (A * B)\ (\mathtt{prod}\ A\ B)\ (\mathtt{to}\ A\ B)\ (\mathtt{from}\ A\ B))$$
$$(\mathtt{id}\ (\mathtt{prod}\ A\ B))$$
$$= [A, B : \mathtt{U}]\ \mathtt{funext}\ (\mathtt{prod}\ A\ B)\ (\mathtt{prod}\ A\ B)$$
$$(\mathtt{fcomp}\ (\mathtt{prod}\ A\ B)\ (A * B)\ (\mathtt{prod}\ A\ B)\ (\mathtt{to}\ A\ B)\ (\mathtt{from}\ A\ B))$$
$$(\mathtt{id}\ (\mathtt{prod}\ A\ B))\ (\mathtt{to\_from}'\ A\ B)$$

Viceversa now we show that applying `to` and then `from` is (propositionally) pointwise equal to the identity.

$$\begin{aligned}
\texttt{from\_to}' : [A, B : \texttt{U}]&[p : A * B] \texttt{ Path } (A * B) \texttt{ (from } A \ B \texttt{ (to } A \ B \ p)) \ p \\
= [A, B : \texttt{U}]& \texttt{ recSigma } A \ B \\
&([p' : A * B] \texttt{ Path } (A * B) \texttt{ (from } A \ B \texttt{ (to } A \ B \ p')) \ p') \\
&([a : A][b : B] \texttt{ refl } (A * B) \ (a, b))
\end{aligned}$$

And then we can prove the other half of the equivalence.

$$\begin{aligned}
\texttt{from\_to} : [A, B : \texttt{U}]& \texttt{ Path } (A * B \to A * B) \\
&(\texttt{fcomp } (A * B) \texttt{ (prod } A \ B) \ (A * B) \texttt{ (from } A \ B) \texttt{ (to } A \ B)) \\
&(\texttt{id } (A * B)) \\
= [A, B : \texttt{U}]& \texttt{ funext } (A * B) \ (A * B) \\
&(\texttt{fcomp } (A * B) \texttt{ (prod } A \ B) \ (A * B) \texttt{ (from } A \ B) \texttt{ (to } A \ B)) \\
&(\texttt{id } (A * B)) \texttt{ (from\_to' } A \ B)
\end{aligned}$$

Finally, using what we proved till now, we get the equivalence between the two product types.

$$\begin{aligned}
\texttt{prodEquiv} : [A, B : \texttt{U}]& \texttt{ equiv } (A * B) \texttt{ (prod } A \ B) \\
= [A, B : \texttt{U}]& \texttt{ (to } A \ B, \\
&((\texttt{from } A \ B, \texttt{to\_from } A \ B), (\texttt{from } A \ B, \texttt{from\_to } A \ B)))
\end{aligned}$$

1.6.2 Path operations

We have already defined transport and the concatenation operation of two paths (see section 1.6); we give now some more examples, following [Ben22], beginning with the path inversion operation, which we derive by composition.

$$\begin{aligned}
\texttt{inv} : [A : \texttt{U}]&[a, b : A] \texttt{ Path } A \ a \ b \to \texttt{Path } A \ b \ a \\
= [A : U]&[a, b : A][p : \texttt{Path } A \ a \ b][i : \mathbb{I}] \\
&\texttt{comp } (\mathbb{I} \to A) \ (i = 0 \land i = 1) \ 0 \ ([j : \mathbb{I}][i = 0 \to p \ j, i = 1 \to a]) \ a \ 1
\end{aligned}$$

We shall now prove the right and left unit laws; the former is easier, since it's already given by the filler of the composition operation (when $q \equiv \texttt{refl } A \ b$), which we basically already derived in 1.2.2.
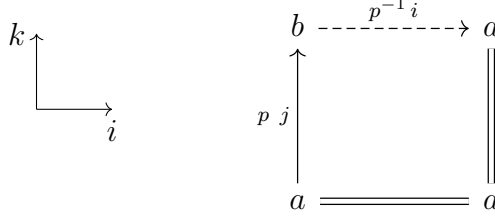
Figure 1.2: Inversion of a path, $p \mapsto p^{-1}$.
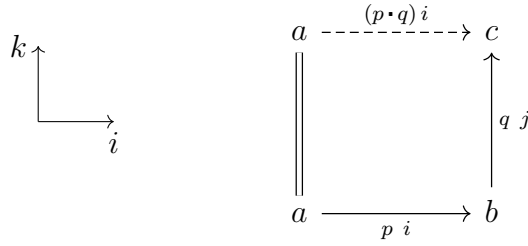
Figure 1.3: Concatenation of two paths.

$\texttt{concatFill} : [A : \texttt{U}][a, b, c : A][p : \texttt{Path } A\ a\ b][q : \texttt{Path } A\ b\ c]$
$\qquad [i, j : \mathbb{I}][j = 0 \mapsto p\ i, i = 0 \mapsto a, i = 1 \to q\ j]A$
$\quad = [A : \texttt{U}][a, b, c : A][p : \texttt{Path } A\ a\ b][q : \texttt{Path } A\ b\ c][i, j : \mathbb{I}]$
$\qquad \texttt{comp } (\mathbb{I} \to A)\ (i = 0 \vee i = 1)\ \texttt{0}\ ([j' : \mathbb{I}][i = 0 \mapsto a, i = 1 \mapsto q\ j'])\ (p\ i)\ j$

Therefore to get $\texttt{rightUnit}$ we only have to swap the dimensions and specialize q to $\texttt{refl } A\ b$.

$\texttt{rightUnit} : [A : \texttt{U}][a, b, c : A][p : \texttt{Path } A\ a\ b]$
$\qquad \texttt{Path } (\texttt{Path } A\ a\ b)\ p\ (\texttt{concat } A\ a\ b\ b\ p\ (\texttt{refl } A\ b))$
$\quad = [A : \texttt{U}][a, b, c : A][p : \texttt{Path } A\ a\ b][i, j : \mathbb{I}]$
$\qquad \texttt{concatFill } A\ a\ b\ b\ p\ (\texttt{refl } A\ b)\ j\ i$

The derivation of the left unit law is more intricate; we shall first state a lemma, i.e. we obtain the filler of a composition figure which we will need later. The idea is to mimic the meet connection of the interval which is built-in in De Morgan CTT [CCHM15] but not in the Cartesian one, that is we want a function $p\,(-\vee-)$ given by the figure 1.4.
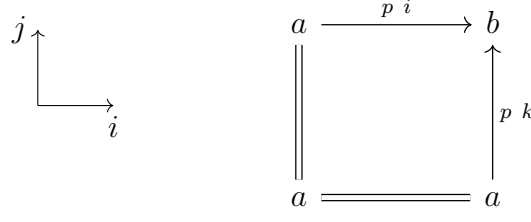
Figure 1.4: Weak meet.

We call it 'weak meet' because the computational rules hold only up to homotopy and not judgmentally. To derive it, we need a 2-dimensional composition, so we have to work on a 3-dimensional hypercube; putting the square we want to obtain on the $j = 1$ face, we compose from $j = 0$ to $j = 1$, defining all the other faces of the cube as reflexivity apart from the $i = 1$ and $k = 1$ faces which are given by the filler of `concat`, as one may check by comparing with the figure 1.3.
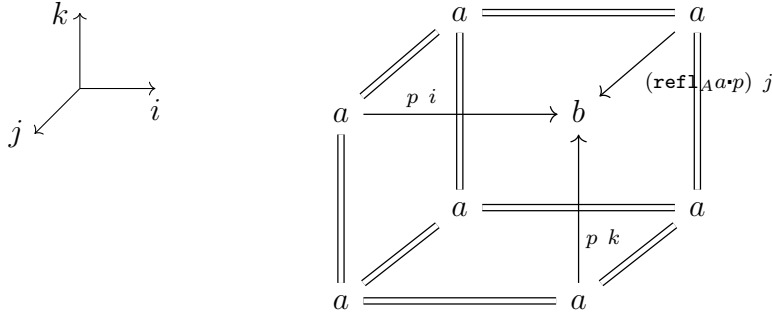
Figure 1.5: Composition for `weakMeet`.

$$
\begin{aligned}
&\texttt{weakMeet} : [A : \texttt{U}][a, b : A][p : \texttt{Path}\ A\ a\ b][i, k : \mathbb{I}] \\
&\qquad [i = 0 \mapsto a, i = 1 \to p\ k, k = 0 \mapsto a, k = 1 \mapsto p\ i]A \\
&\quad = [A : \texttt{U}][a, b : A][p : \texttt{Path}\ A\ a\ b][i, k : \mathbb{I}] \\
&\qquad \texttt{comp}\ (\mathbb{I} \to A)\ (i = 0 \vee i = 1 \vee k = 0 \vee k = 1)\ \texttt{0} \\
&\qquad ([j : \mathbb{I}][i = 0 \mapsto a, \\
&\qquad\qquad i = 1 \mapsto \texttt{concatFill}\ A\ a\ a\ b\ (\texttt{refl}\ A\ a)\ p\ j\ k, \\
&\qquad\qquad k = 0 \mapsto a, \\
&\qquad\qquad k = 1 \mapsto \texttt{concatFill}\ A\ a\ a\ b\ (\texttt{refl}\ A\ a)\ p\ j\ i])
\end{aligned}
$$

$$a\ 1$$

Note that the type of `weakMeet` can be seen as a *dependent path* between `refl` $A\ a$ and p, as they both start at a, and their endpoint follows $p\ i$.

$$\texttt{PathD} : [A : \mathbb{I} \to \texttt{U}]\ A\ 0 \to A\ 1 \to \texttt{U}$$
$$= [A : \mathbb{I} \to \texttt{U}][a0 : A\ 0][a1 : A\ 1][i : \mathbb{I}][i = 0 \to a0, i = 1 \to a1]A\ i$$
$$\texttt{weakMeetDP} : [A : \texttt{U}][a, b : A][p : \texttt{Path}\ A\ a\ b]\ \texttt{PathD}\ ([i : \mathbb{I}]\ \texttt{Path}\ A\ a\ (p\ i))\ (\texttt{refl}\ A\ a)\ p$$

Turning back to the left unit law, as before we work in a 3-dimensional hypercube, with the $j = 1$ face (which is the one that we want to obtain with composition) being a path between p (at $k = 0$) and `refl` $A\ a \cdot p$ (at $k = 1$). We complete the cube with the filler of the inverse on the $j = 0$ face, reflexivity for the $i = 0$ face, the filler of concatenation on the $k = 1$ face, $p\ i$ on the $k = 0$ face and finally a term γ, which we shall derive again by composition, on the $i = 1$ face.
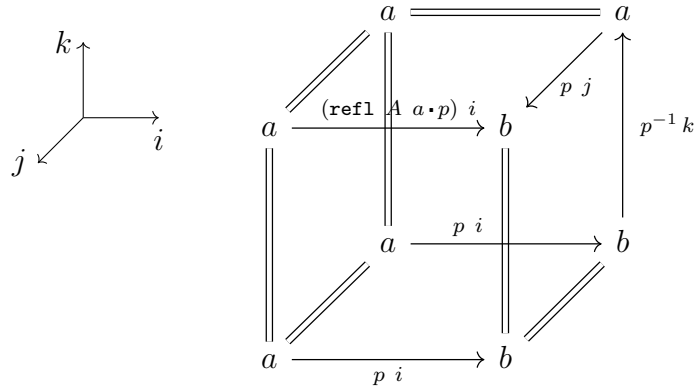
Figure 1.6: Composition for the left unit law.

$$\texttt{leftUnit} : [A : \texttt{U}][a, b, c : A][p : \texttt{Path}\ A\ a\ b]$$
$$\texttt{Path}\ (\texttt{Path}\ A\ a\ b)\ p\ (\texttt{concat}\ A\ a\ a\ b\ (\texttt{refl}\ A\ a)\ p)$$
$$= [A : \texttt{U}][a, b, c : A][p : \texttt{Path}\ A\ a\ b][k, i : \mathbb{I}]$$
$$\texttt{comp}\ (\mathbb{I} \mapsto A)\ (i = 0 \vee i = 1 \vee k = 0 \vee k = 1)\ 0$$
$$([j : \mathbb{I}][i = 0 \mapsto a,$$
$$i = 1 \mapsto \texttt{gamma}\ A\ a\ b\ p\ j\ k,$$
$$k = 0 \mapsto p\ i,$$

$$k = 1 \mapsto \texttt{concatFill}\ A\ a\ a\ b\ (\texttt{refl}\ A\ a)\ p\ i\ j])$$
$$(\texttt{invFill}\ A\ a\ b\ p\ k\ i)\ 1$$

We now show how to derive the square γ; as usual we work in a 3-dimensional hypercube (figure 1.7), defining its sides using the faces we already have at disposal.
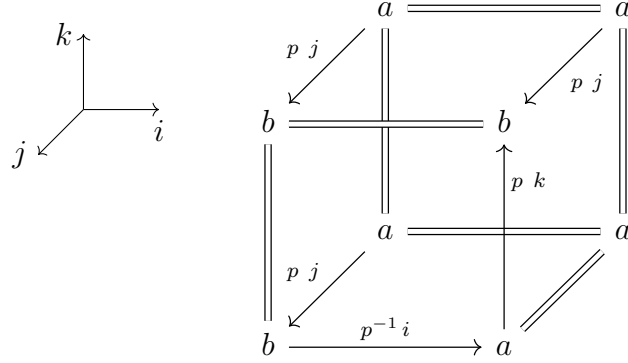
Figure 1.7: Composition for γ.

$$\texttt{gamma} : [A : \texttt{U}][a, b : A][p : \texttt{Path}\ A\ a\ b][k, i : \mathbb{I}]$$
$$[k = 0 \mapsto \texttt{inv}\ A\ a\ b\ p\ i, k = 1 \mapsto b, i = 0 \mapsto b, i = 1 \mapsto p\ k]A$$
$$= [A : \texttt{U}][a, b : A][p : \texttt{Path}\ A\ a\ b][k, i : \mathbb{I}]$$
$$\texttt{comp}\ (\mathbb{I} \mapsto A)\ (i = 0 \vee i = 1 \vee k = 0 \vee k = 1)\ 0$$
$$([j : \mathbb{I}][i = 0 \mapsto p\ j,$$
$$i = 1 \mapsto \texttt{weakMeet}\ A\ a\ b\ p\ j\ k,$$
$$k = 0 \mapsto \texttt{invFill}\ A\ a\ b\ p\ i\ j,$$
$$k = 1 \mapsto p\ j])$$
$$a\ 1$$

As a last example, we show how to derive the induction rule for path types, which asserts that for each predicate $C : [x : A]\ \texttt{Path}\ A\ a\ x \to \texttt{U}$, from $c : C\ a\ (\texttt{refl}\ A\ a)$ one may get $C\ x\ p$ for each $x : A$ and $p : \texttt{Path}\ A\ a\ x$. The idea is simply to transport c along a 'type line' $C\ a\ (\texttt{refl}\ A\ a) \to C\ x\ p$, that is we have to find a $D : \mathbb{I} \to \texttt{U}$ such that $D\ 0 \equiv C\ a\ (\texttt{refl}\ A\ a)$ and $D\ 1 \equiv C\ x\ p$. Using the weak meet operation, we choose

$$D \equiv [i : \mathbb{I}]\ C\ (p\ i)\ ([j : \mathbb{I}]\ p\ (i \wedge j)),$$

so that

$$D \; \mathtt{0} \equiv C \; (p \; \mathtt{0}) \; ([j : \mathbb{I}] \; p \, (\mathtt{0} \wedge j)) \equiv C \; a \; ([j : I] \; a) \equiv C \; a \; (\mathtt{refl} \; A \; a),$$
$$D \; \mathtt{1} \equiv C \; (p \; \mathtt{1}) \; ([j : \mathbb{I}] \; p \, (\mathtt{1} \wedge j)) \equiv C \; x \; ([j : I] \; p \; j) \equiv C \; x \; p$$

Therefore, we obtain path induction.

$$
\begin{aligned}
\mathtt{pathInd} : \; & [A : U][a : A][C : [x : A] \; \mathtt{Path} \; A \; a \; x \to U] \\
& [x : A][p : \mathtt{Path} \; A \; a \; x] \; C \; a \; (\mathtt{refl} \; A \; a) \to C \; x \; p \\
= \; & [A : U][a : A][C : [x : A]\mathtt{Path} \; A \; a \; x \to U] \\
& [x : A][p : \mathtt{Path} \; A \; a \; x][c : C \; a \; (\mathtt{refl} \; A \; a)] \\
& \mathtt{transp} \; ([i : \mathbb{I}] \; C \; (\mathtt{weakMeet} \; A \; a \; x \; p \; i \; \mathtt{1}) \; ([j : \mathbb{I}] \; \mathtt{weakMeet} \; A \; a \; x \; p \; i \; j)) \; c
\end{aligned}
$$

It's also possible to derive the propositional computational rule for path induction, which is shown in [Ben22].

Chapter 2

Semantics

In this chapter we study the semantics of dependent type theory with partial elements. First we introduce the notion of 'categories with families' [Hof97], which is an abstract semantics based on category theory and is now the de facto standard in type theory semantics. Most interpretations of type theory (e.g. groupoid [HS96], simplicial [KL21], cubical [CCHM15]) can be seen as particular cases of the aforementioned abstract semantics. We also describe presheaf models, that is categories with families induced by the category of presheaves over a (base) category. The category of cubical sets (cf. section 2.3) is an example of a presheaf category. Lastly, we show how to model partial and restriction types in presheaf models, using the category-theoretic concept of subobject classifier.

2.1 Categories with families

First of all we define the category of families of sets, which is used to model dependency.

Definition 2.1.1 (Fam). *The category Fam of families of sets is defined by:*

1. *Objects of the form (A, B), where A is a set and $B = (B_a)_{a \in A}$ is a family of sets indexed by A.*

2. *Morphisms of the form $(f, g) : (A, B) \to (A', B')$, where $f : A \to A'$ is a function and $g = \{g_a : a \in A\}$ is a family of functions $g_a : B_a \to B'_{f(a)}$ indexed by A. Composition is defined pointwise on the second component.*

Before giving the main definition, let us fix the postfix notation for the composition of functions, i.e. writing fg for $g \circ f$, which will make the notation

more intuitive as morphisms in CwFs are substitutions.

Definition 2.1.2 (CwF)**.** *A category with families (CwF) is given by:*

1. *A category $\mathcal{C}$ with objects $\Gamma, \Delta, \ldots$ called* contexts, *morphisms $\sigma, \tau, \ldots$ called* substitutions *and a terminal object 1.[1] We write $\Gamma \vdash$ to mean that Γ is a context of $\mathcal{C}$.[2]*

2. *A functor $\mathcal{F} : \mathcal{C}^{op} \to \mathsf{Fam}$. Given a context Γ, we write the indexed family $\mathcal{F}(\Gamma)$ as $(Type(\Gamma), (Term(\Gamma, A))_{A \in Type(\Gamma)})$. If $A \in Type(\Gamma)$, we say that A is a type in context Γ and write $\Gamma \vdash A$; if $a \in Term(\Gamma, A)$, we say that a is a term of type A in context Γ and write $\Gamma \vdash a : A$. For the morphisms of $\mathcal{C}$, given a substitution $\sigma : \Delta \to \Gamma$, contravariance yields $\mathcal{F}(\sigma) : \mathcal{F}(\Gamma) \to \mathcal{F}(\Delta)$, so $\mathcal{F}$ acts on types $\Gamma \vdash A$ as $\Delta \vdash A\sigma$ and on terms $\Gamma \vdash a : A$ as $\Delta \vdash a\sigma : A\sigma$. Since $\mathcal{F}$ is a functor, we also have:[3]*

$$A\mathbf{1} = A, \quad (A\sigma)\tau = A(\sigma\tau), \quad a\mathbf{1} = a, \quad (a\sigma)\tau = a(\sigma\tau).$$

3. *An operation of* context extension*: if $\Gamma \vdash A$, then there exists a context $\Gamma.A$ with a substitution $\mathsf{p}_{\Gamma,A} : \Gamma.A \to \Gamma$ and a term $\Gamma.A \vdash \mathsf{q}_{\Gamma,A} : A\mathsf{p}_{\Gamma,A}$ such that for each context Δ, substitution $\sigma : \Delta \to \Gamma$ and term $\Delta \vdash a : A\sigma$, there exists a substitution $(\sigma, a) : \Delta \to \Gamma.A$ such that:*

$$\mathsf{p}_{\Gamma,A}(\sigma, a) = \sigma, \quad \mathsf{q}_{\Gamma,A}(\sigma, a) = a, \quad (\sigma, a)\tau = (\sigma\tau, a\tau), \quad (\mathsf{p}_{\Gamma,A}, \mathsf{q}_{\Gamma,A}) = \mathbf{1}_{\Gamma.A}.$$

Usually we will simply write p and q if Γ and A are clear.

The definitions given above for a CwF can be presented in a rule form, and this is the notation we will use from now on. Pay attention not to confuse them with the formal syntactical rules of type theory.

- Context formation:

$$\frac{}{1 \vdash} \qquad \frac{\Gamma \vdash \quad \Gamma \vdash A}{\Gamma.A \vdash}$$

- Substitution in types and terms:

$$\frac{\Gamma \vdash A \quad \sigma : \Delta \to \Gamma}{\Delta \vdash A\sigma} \qquad \frac{\Gamma \vdash a : A \quad \sigma : \Delta \to \Gamma}{\Delta \vdash a\sigma : A\sigma}$$

- Context manipulation:

$$\frac{\Gamma \vdash A}{\mathsf{p} : \Gamma.A \to \Gamma} \qquad \frac{\Gamma \vdash A}{\Gamma.A \vdash \mathsf{q} : A\mathsf{p}} \qquad \frac{\Gamma \vdash A \quad \Delta \vdash a : A\sigma \quad \sigma : \Delta \to \Gamma}{(\sigma, a) : \Delta \to \Gamma.A}$$

---

[1]We denote by $\mathbf{1}_A$, or even by $\mathbf{1}$, the constant substitution $A \to A$.

[2]$\mathcal{C}$ will always be clear from the context.

[3]It's clear that the functor $\mathcal{F}$ serves only for presentation purposes, so we will not use it.

Using these rules, we can derive the first 'operation', which is the substitution by a term, as shown by the following remark.

Remark 2.1.3 (Substitution by a term). *If we have a type $\Gamma \vdash A$ and a dependent type over A, i.e. $\Gamma.A \vdash B$, then there is a 'substitution by a' operation in the CwF; indeed, given the identity substitution $\mathbf{1} : \Gamma \to \Gamma$, using the context extension property we can form $(\mathbf{1}, a) : \Gamma \to \Gamma.A$, usually denoted by $[a] : \Gamma \to \Gamma.A$, which gives $\Gamma \vdash B[a]$. Similarly for terms, i.e. from $\Gamma.A \vdash b : B$ we obtain $\Gamma \vdash b[a] : B[a]$.*

The following is a technical remark which we will use in the subsequent arguments; it formalizes the intuitive concept of extending a substitution.

Remark 2.1.4. *If we have a type $\Gamma \vdash A$ and $\sigma : \Delta \to \Gamma$, then we can find a context morphism $\Delta.A\sigma \to \Gamma.A$ as follows: first we form $\sigma\mathsf{p}_{\Delta,A\sigma} : \Delta.A\sigma \to \Gamma$, with $\Delta.A\sigma \vdash \mathsf{q}_{\Delta,A\sigma} : A\sigma\mathsf{p}_{\Delta,A\sigma}$; then we use context comprehension to get $(\sigma\mathsf{p}_{\Delta,A\sigma}, \mathsf{q}_{\Delta,A\sigma}) : \Delta.A\sigma \to \Gamma.A$.*

We now explain how to model type formers in a CwF, by first defining Π-types.

Definition 2.1.5 (Π-types). *A CwF supports Π-types if the following rules are admissible*

$$\frac{\Gamma \vdash A \quad \Gamma.A \vdash B}{\Gamma \vdash \Pi_A B} \qquad \frac{\Gamma.A \vdash b : B}{\Gamma \vdash \lambda\,b : \Pi_A B} \qquad \frac{\Gamma \vdash u : \Pi_A B \quad \Gamma \vdash a : A}{\Gamma \vdash \mathsf{app}(u, a) : B[a]}$$

and they satisfy the β-rule

$$\mathsf{app}(\lambda\,b, a) = b[a]$$

and the laws for commutation with substitutions

$$(\Pi_A B)\sigma = \Pi_{A\sigma} B(\sigma\mathsf{p}, \mathsf{q}), \qquad (\lambda\,b)\sigma = \lambda\,b(\sigma\mathsf{p}, \mathsf{q}), \qquad \mathsf{app}(u, a)\sigma = \mathsf{app}(u\sigma, a\sigma).$$

Remark 2.1.6. *The above three equations can be justified by the following considerations,[4] given $\sigma : \Delta \to \Gamma$.*

- *If*

$$\frac{\Gamma \vdash A \quad \Gamma.A \vdash B}{\Gamma \vdash \Pi_A B}$$

  then

$$\frac{\dfrac{\Gamma \vdash A}{\Delta \vdash A\sigma} \quad \dfrac{\Gamma.A \vdash B}{\Delta.A\sigma \vdash B(\sigma\mathsf{p}, \mathsf{q})}}{\Delta \vdash \Pi_{A\sigma} B(\sigma\mathsf{p}, \mathsf{q})}$$

  which is exactly how we defined $(\Pi_A B)\sigma$.

---

[4]For brevity we do this only for the Π-type former, as it can done analogously for the others type formers.

- *If*

$$\frac{\Gamma.A \vdash b : B}{\Gamma \vdash \lambda\,b : \Pi_A B}$$

  then

$$\frac{\dfrac{\Gamma.A \vdash b : B}{\Delta.A\sigma \vdash b(\sigma\mathsf{p}, \mathsf{q}) : B(\sigma\mathsf{p}, \mathsf{q})}}{\Delta \vdash \lambda\,b(\sigma\mathsf{p}, \mathsf{q}) : \Pi_{A\sigma} B(\sigma\mathsf{p}, \mathsf{q})}$$

  The last line is the same as $\Delta \vdash (\lambda\,b)\sigma : (\Pi_A B)\sigma$.

- *If*

$$\frac{\Gamma \vdash u : \Pi_A B \quad \Gamma \vdash a : A}{\Gamma \vdash \mathsf{app}(u, a) : B[a]}$$

  then

$$\frac{\dfrac{\Gamma \vdash u : \Pi_A B}{\Delta \vdash u\sigma : \Pi_{A\sigma} B(\sigma\mathsf{p}, \mathsf{q})} \quad \dfrac{\Gamma \vdash a : A}{\Delta \vdash a\sigma : A\sigma}}{\Delta \vdash \mathsf{app}(u\sigma, a\sigma) : B(\sigma\mathsf{p}, \mathsf{q})[a]}$$

  but $B(\sigma\mathsf{p}, \mathsf{q})[a\sigma] = B(\sigma\mathsf{p}, \mathsf{q})(\mathbf{1}, a\sigma) = B(\sigma\mathsf{p}(\mathbf{1}, a\sigma), \mathsf{q}(\mathbf{1}, a\sigma)) = B(\sigma, a\sigma)$, meaning that the last line in the derivation is the same as $\Delta \vdash (\mathsf{app}(u, a))\sigma : B[a]\sigma$, since $B[a]\sigma = B(\mathbf{1}, a)\sigma = B(\mathbf{1}\sigma, a\sigma) = B(\sigma, a\sigma)$ too.

Similarly to Π-types, we now introduce Σ-types, whose definition is a bit simpler.

Definition 2.1.7 (Σ-types)**.** *A CwF supports Σ-types if the following rules are admissible*

$$\frac{\Gamma \vdash A \quad \Gamma.A \vdash B}{\Gamma \vdash \Sigma_A B} \qquad \frac{\Gamma \vdash a : A \quad \Gamma \vdash b : B[a]}{\Gamma \vdash \langle a, b \rangle : \Sigma_A B}$$

$$\frac{\Gamma \vdash u : \Sigma_A B}{\Gamma \vdash \pi_1 u : A} \qquad \frac{\Gamma \vdash u : \Sigma_A B}{\Gamma \vdash \pi_2 u : B[\pi_1 u]}$$

and they satisfy the β-rules

$$\pi_1 \langle a, b \rangle = a, \qquad \pi_2 \langle a, b \rangle = b$$

and the laws for commutation with substitutions

$$(\Sigma_A B)\sigma = \Sigma(A\sigma)(B(\sigma\mathsf{p}, \mathsf{q})), \qquad \langle a, b \rangle\sigma = \langle a\sigma, b(\sigma\mathsf{p}, \mathsf{q}) \rangle,$$
$$(\pi_1 u)\sigma = \pi_1(u\sigma), \qquad (\pi_2 u)\sigma = \pi_2(u\sigma).$$

One can define analogously the rules for the other type formers, as coproducts, naturals, etc. For brevity, here and in the following sections, we treat only the cases of Π and Σ types. The semantic interpretation of the universe is trickier, which we refer to [Hub16] chapter 1.

2.2 Presheaf models

In this section we show how the presheaf category on a fixed category $\mathcal{C}$ induces a CwF; this will be used in section 2.3 to handle cubical constructs by suitably choosing the base category $\mathcal{C}$.

Definition 2.2.1. *A* presheaf *on a category $\mathcal{C}$ is a functor $\mathcal{C}^{op} \to$ Set. The category $\hat{\mathcal{C}}$ of presheaves on $\mathcal{C}$ is the functor category $[\mathcal{C}^{op}, \text{Set}]$, which has presheaves on $\mathcal{C}$ as objects and natural transformations between them as arrows.*

A presheaf Γ will be thought as a context $\Gamma \vdash$, with sets $\Gamma(I)$ for each $I \in \mathcal{C}$ and functions $\Gamma_f : \Gamma(I) \to \Gamma(J)$ for each $f : J \to I$, usually written $\rho \mapsto \rho f$ instead of $\rho \mapsto \Gamma_f(\rho)$, satisfying the functor laws

$$\rho\mathbf{1} = \rho, \qquad (\rho)fg = \rho(fg).$$

We now explain how to define a CwF from $\hat{\mathcal{C}}$:

- Contexts are given by presheaves, with the terminal object 1 defined by the constant presheaf $\mathbb{1}$ such that $\mathbb{1}(I) = \{\star\}$ for all $I \in \mathcal{C}$.[5]

- A substitution $\sigma : \Delta \to \Gamma$ is a morphism between functors Δ and Γ, i.e. a natural transformation $\sigma = (\sigma_I : \Delta(I) \to \Gamma(I))_{I \in \mathcal{C}}$, such that the naturality condition holds:

$$\Gamma_f \circ \sigma_I = \sigma_J \circ \Delta_f$$

usually written as

$$(\sigma\rho)f = \sigma(\rho f)$$

for all $\rho \in \Delta(I)$.

- A type $\Gamma \vdash A$ is given by sets $A\rho$ for all $I \in \mathcal{C}$ and $\rho \in \Gamma(I)$ and by functions $A\rho \to A(\rho f)$ for each $f : J \to I$, as usual written as $a \mapsto af$, such that the functorial laws hold:

$$a\mathbf{1} = a, \qquad (af)g = a(fg).$$

Substitutions on types are defined as follows: given $\Gamma \vdash A$ and $\sigma : \Delta \to \Gamma$, $\Delta \vdash A\sigma$ is given by sets $(A\sigma)\rho = A(\sigma\rho)$ and the induced map $(A\sigma)\rho = A(\rho\sigma) \to A((\rho\sigma)f) = A(\sigma(\rho f)) = (A\sigma)(\rho f)$

---

[5]The set $\{\star\}$ represents 'the' singleton set.

- A term $\Gamma \vdash a : A$ is given by an element $a\rho \in A\rho$ for each $I \in \mathcal{C}$ and $\rho \in \Gamma(I)$, such that the following law hold for all $f : J \to I$:

$$(a\rho)f = a(\rho f).$$

Substitutions on terms are defined as follows: given $\Gamma \vdash a : A$ and $\sigma : \Delta \to \Gamma$, $\Delta \vdash a\sigma : A\sigma$ is given by sets $(a\sigma)\rho = a(\sigma\rho)$.

- The context extension $\Gamma.A \vdash$, given $\Gamma \vdash A$, is defined by the sets

$$(\Gamma.A)(I) = \{(\rho, u) : \rho \in \Gamma(I), u \in A\rho\}$$

for each $I \in \mathcal{C}$ and by the functions $(\Gamma.A)_f : (\Gamma.A)(I) \to (\Gamma.A)(J)$ defined by
$$(\rho, u)f = (\rho f, uf)$$

for each morphism $f : J \to I$. Projections $\mathsf{p} : \Gamma.A \to \Gamma$ and $\Gamma.A \vdash \mathsf{q} : A\mathsf{p}$ are defined as to satisfy the required equations, i.e.

$$\mathsf{p}(\rho, u) = \rho,$$
$$\mathsf{q}(\rho, u) = u.^6$$

It is straightforward now to verify that the equations hold:

- $\mathsf{p}(\sigma, a) = \sigma$ by definition;

- $\mathsf{q}(\sigma, a) = a$ by definition;

- $(\sigma, a)\tau = (\sigma\tau, a\tau)$ by definition;

- $(\mathsf{p}, \mathsf{q}) = \mathbf{1}$, since $(\mathsf{p}, \mathsf{q})(\rho, u) = (\mathsf{p}(\rho, u), \mathsf{q}(\rho, u)) = (\rho, u)$.

2.2.1   Π-types

First of all, following [Hub16], we give a motivation for the definition of Π-types in presheaf categories, by exploiting the intuitive idea that function types should be modelled by the categorical counterpart which are exponentials. If Γ and Δ are presheaves and the exponential Δ^Γ exists, then by the Yoneda lemma (see appendix A) and the fact that exponentiation is right adjoint to product, we get

$$(\Delta^\Gamma)(I) \cong \mathsf{Hom}_{\hat{\mathcal{C}}}(\mathsf{y}I, \Delta^\Gamma) \cong \mathsf{Hom}_{\hat{\mathcal{C}}}(\mathsf{y}I \times \Gamma, \Delta).$$

---

[6]Remember that to give a term $\Gamma.A \vdash \mathsf{q} : A\mathsf{p}$ we must specify an element $\mathsf{q}(\rho, u) \in A\rho$ for each $(\rho, u) \in \Gamma.A$.

This can then be taken as a definition, so that the elements of the presheaf Δ^Γ are natural transformations $w : \mathsf{y}I \times \Gamma \to \Delta$, i.e. $w = (w_J : \mathsf{Hom}_\mathcal{C}(J, I) \times \Gamma(J) \to \Delta(J))_{J \in \mathcal{C}}$ satisfying the naturality condition

$$\Delta_g \circ w_J = w_K \circ ((\mathsf{y}I)_g, \Gamma_g)$$

for each $g : K \to J$, which means that for every $(f, \rho) \in \mathsf{Hom}_\mathcal{C}(J, I) \times \Gamma(J)$, the following holds:

$$(w_J(f, \rho))g = w_K(fg, \rho g).$$

$$
\begin{array}{ccc}
\mathsf{Hom}_\mathcal{C}(J, I) \times \Gamma(J) & \xrightarrow{\ (\mathsf{y}I)_g\ } & \mathsf{Hom}_\mathcal{C}(K, I) \times \Gamma(K) \\
\Big\downarrow{\scriptstyle w_J} & & \Big\downarrow{\scriptstyle w_K} \\
\Delta(J) & \xrightarrow[\ \Delta_g\ ]{} & \Delta(K)
\end{array}
$$

Figure 2.1: Naturality condition for w.

Now we show how a presheaf category supports Π-types in the induced CwF:

1. (Type formation) Given $\Gamma \vdash A$ and $\Gamma.A \vdash B$, we shall define the type $\Gamma \vdash \Pi_A B$. Let $I \in \mathcal{C}$ and $\rho \in \Gamma(I)$; we define the set $(\Pi_A B)\rho$ as composed by the families $w = (w_f)_{J \in \mathcal{C}, f : J \to I}$, such that for each $u \in A(\rho f)$,

$$w_f u \in B(\rho f, u)$$

and for each $g : K \to J$, the following holds:

$$(w_f u)g = w_{fg}(ug).$$

The last equation is well typed since $(w_f u)g \in B(\rho f, u)g = B(\rho f g, ug)$. We also have to specify a function $(\Pi_A B)\rho \to (\Pi_A B)(\rho f)$ for each $f : J \to I$, mapping $w \in (\Pi_A B)\rho \mapsto wf \in (\Pi_A B)(\rho f)$. The latter is defined by

$$(wf)_g u = w_{fg} u \in B(\rho f g, u)$$

for each $g : K \to J$ and $u \in A(\rho f g)$. This definition satisfies the required functorial equations:

- $w\mathbf{1} = w$, since $(w\mathbf{1})_g u = w_{\mathbf{1}g} u = w_g u$.

- $(wf)g = w(fg)$, since $((wf)g)_h u = (wf)_{gh} u = w_{fgh} u$ and $(w(fg))_h u = w_{fgh} u$.

2. (Abstraction) Given $\Gamma.A \vdash b : B$, we shall define a term $\Gamma \vdash \lambda\, b : \Pi_A B$ by giving a set $(\lambda\, b)\rho \in (\Pi_A B)\rho$ for each $I \in \mathcal{C}$ and $\rho \in \Gamma(I)$; if $f : J \to I$ and $u \in A(\rho f)$, we put

$$((\lambda\, b)\rho)_f u = b(\rho f, u) \in B(\rho f, u).^7$$

We have to check that $((\lambda\, b)\rho)f = (\lambda\, b)(\rho f)$, indeed

$$(((\lambda\, b)\rho)f)_g u = ((\lambda\, b)\rho)_{fg} u = b(\rho f g, u)$$

which is equal to $((\lambda\, b)(\rho f))_g u = b(\rho f g, u)$ by definition.

3. (Application) Given $\Gamma \vdash u : \Pi_A B$ and $\Gamma \vdash v : A$, we shall define a term $\Gamma \vdash \mathsf{app}(u, v) : B[v]$, remembering that $B[v]$ is a shortcut for $B(1, v)$. So for each $I \in \mathcal{C}$ and $\rho \in \Gamma(I)$ we define the set $\mathsf{app}(u, v)\rho : B(1, v)\rho$, i.e. of type $B(\rho, v\rho)$, by

$$\mathsf{app}(u, v)\rho = (u\rho)_{\mathbf{1}}(v\rho) \in B(\rho\mathbf{1}, v\rho) = (B[v])\rho$$

since $u\rho \in (\Pi_A B)\rho$ and $v\rho \in A\rho$.

It's now easy to check that the CwF equations for Π types hold:

- (β-reduction) $\mathsf{app}(\lambda\, b, v) = b[v]$ since

$$\mathsf{app}(\lambda\, b, v)\rho = ((\lambda\, b)\rho)_{\mathbf{1}}(v\rho) = b(\rho, v\rho) = b[v]\rho.$$

- (Substitution 1) $(\Pi_A B)\sigma = \Pi_{A\sigma} B(\sigma\mathsf{p}, \mathsf{q})$ holds since $((\Pi_A B)\sigma)\rho = (\Pi_A B)(\sigma\rho)$ and any element w of that set is such that for all $u \in A(\sigma\rho f) = A(\sigma(\rho f))$, $w_f u \in B(\sigma\rho f, u = B(\sigma\mathsf{p}, \mathsf{q})(\rho f, u)$, which means that w is also an element of $\Pi_{A\sigma} B(\sigma\mathsf{p}, \mathsf{q})$. The other inclusion in proved in the same manner.

- (Substitution 2) $(\lambda\, b)\sigma = \lambda\, b(\sigma\mathsf{p}, \mathsf{q})$ holds since

$$\begin{aligned}(((\lambda\, b)\sigma)\rho)_f u &= ((\lambda\, b)(\sigma\rho))_f u = b(\sigma\rho f, u)\\ &= b(\sigma\mathsf{p}, \mathsf{q})(\rho f, u) = ((\lambda\, b(\sigma\rho f, u))\rho)_f u.\end{aligned}$$

- (Substitution 3) $\mathsf{app}(u, v)\sigma = \mathsf{app}(u\sigma, v\sigma)$ holds since

$$\begin{aligned}(\mathsf{app}(u, v)\sigma)\rho &= \mathsf{app}(u, v)(\sigma\rho) = (u(\sigma\rho))_{\mathbf{1}}(v(\sigma\rho))\\ &= ((u\sigma)\rho)_{\mathbf{1}}((v\sigma)\rho) = \mathsf{app}(u\sigma, v\sigma)\rho.\end{aligned}$$

---

[7] Remember that $\Gamma.A \vdash b : B$ and $(\rho f, u) \in (\Gamma.A)(J)$, so that $b(\rho f, u) \in B(\rho f, u)$.

2.2.2 Σ-types

We show how a presheaf category supports Σ-types in the induced CwF:

1. (Type formation) Given $\Gamma \vdash A$ and $\Gamma.A \vdash B$, we shall define the type $\Gamma \vdash \Sigma_A B$. For $I \in \mathcal{C}$ and $\rho \in \Gamma(I)$, let

$$(\Sigma_A B)\rho = \{(a, b) : a \in A\rho, b \in B(\rho, a)\}$$

   with the mapping $(\Sigma_A B)\rho \to (\Sigma_A B)(\rho f)$ given by

$$(a, b)f \mapsto (af, bf).$$

2. (Pairing) Given $\Gamma \vdash a : A$ and $\Gamma.A \vdash b : B[a]$, we shall define the term $\Gamma \vdash \langle a, b \rangle : \Sigma_A B$. For $I \in \mathcal{C}$ and $\rho \in \Gamma(I)$, let

$$\langle a, b \rangle \rho = \langle a\rho, b\rho \rangle.$$

3. (Projections) Given $\Gamma \vdash u : \Sigma_A B$, we shall define the two projections $\Gamma \vdash \pi_1 u : A$ and $\Gamma \vdash \pi_2 u : B[\pi_1 u]$ by

$$(\pi_1 u)\rho = a, \qquad (\pi_2 u)\rho = b,$$

   where $u\rho = \langle a, b \rangle \in (\Sigma_A B)\rho$.

It can be easily shown that β-reduction and commutation with substitution laws hold, analogously to what we have done in section 2.2.1 for Π-types.

2.3 Cubical sets

In this section we introduce cubical sets, which is the presheaf category on the category of cubes $\mathcal{C}$.

Definition 2.3.1 (Category of cubes). *Fix a discrete and countably infinite set of names, usually denoted by $x, y, z, \ldots$. The* category of cubes $\mathcal{C}$ *has as objects the finite subsets of names, usually denoted by $I, J, K, \ldots$, and as morphisms $f : I \to J$ functions $\tilde{f} : I \to J \cup \mathbf{2}$ which are injective on the set $\mathsf{def}(f) = \{x \in I : \tilde{f}(x) \notin \mathbf{2}\}$. Composition of morphisms is defined as*

$$fg(x) = \begin{cases} fg(x) & \text{if } x \in \mathsf{def}(f) \\ f(x) & \text{otherwise} \end{cases}$$

Example 2.3.2. *Using the notation $I, x_1, \ldots, x_n$ for $I \cup \{x_1, \ldots, x_n\}$ and $I - x_1, \ldots, x_n$ for $I \setminus \{x_1, \ldots, x_n\}$, we name a few particular morphisms:*

- *Face maps $(x = 0), (x = 1) : I \to I - x$ which are the identity except that they map x to 0 or 1. It hold that $\mathsf{def}(x0) = \mathsf{def}(x1) = I - x$.*

- *Degeneracy map $i_x : I \to I, x$ for $x \notin I$, which is the inclusion map $I \subseteq I, x$.*

- *Renaming map $(x = y) : I, x \to I, y$ for $x, y \notin I$ which is the identity except that x gets mapped to y.*

Definition 2.3.3. *A cubical set X is a functor $X : \mathcal{C} \to \mathsf{Set}$, where $\mathcal{C}$ is the category of cubes. In other words, a cubical set is a presheaf on $\mathcal{C}^{op}$.*

As any other presheaf category, $\widehat{\mathcal{C}^{op}}$ induces a CwF, as described in section 2.2.

First we show how to model the formal interval $\mathbb{I}$ as a cubical set.

Example 2.3.4. *The interval $\mathbb{I}$ is defined on objects by $\mathbb{I}(I) = I \cup \{0, 1\}$ and on morphisms $f : I \to J$ by $\mathbb{I}_f : I \cup \{0, 1\} \to J \cup \{0, 1\}$ obtained extending f with $\mathbb{I}_f(0) = 0$ and $\mathbb{I}_f(1) = 1$. It's important to note that $\mathbb{I}$ is a representable functor, as*

$$\mathsf{y}\{x\} = \mathsf{Hom}_{\mathcal{C}}(\{x\}, -) \cong \mathbb{I}$$

since for example $\mathsf{y}\{x\}\{y_1, \ldots, y_n\} = \mathsf{Hom}_{\mathcal{C}}(\{x\}, \{y_1, \ldots, y_n\})$ which is composed by the morphisms $(x = 0)$, $(x = 1)$ and $(x = y_i)$ for each i, which is in bijection with $\mathbb{I}(\{y_1, \ldots, y_n\})$. More generally, instead of defining the square, the cube, etc. 'manually', we can use the Yoneda embedding (see appendix A) to define them using y, so we set

$$\square_I = \mathsf{y}I,$$

for any set of names I, called the standard I-cube. By the Yoneda lemma, it holds that

$$\mathsf{Hom}_{\widehat{\mathcal{C}^{op}}}(\square_I, X) \cong X(I).$$

2.3.1 The subobject classifier

We now introduce a categorical construction, called *subobject classifier*, that generalizes the set of truth values $\{0, 1\}$ in the category of sets to arbitrary categories, allowing to describe 'subobjects' of any object X as morphisms from X to the subobject classifier, in the same manner as subsets of a set X can be described by their characteristic function, i.e. a function from X to $\{0, 1\}$.

We follow the description given in [MLM92].

Definition 2.3.5 (Subobject). *If X is an object in a category $\mathcal{C}$, a subobject of C is an equivalence class of monomorphisms $m : U \rightarrowtail X$.[8] The set of subobjects of X is denoted with $\mathsf{Sub}_{\mathcal{C}}(X)$.*

Definition 2.3.6 (Subobject classifier). *Let $\mathcal{C}$ be a category with a terminal object 1. An object $\Omega \in \mathcal{C}$ is said to be a subobject classifier for $\mathcal{C}$ if there exists a morphism $\mathsf{true} : 1 \to \Omega$ such that for each monomorphism $j : U \rightarrowtail X$ there exists an unique $\phi : X \to \Omega$ such that the commutative diagram*

$$
\begin{array}{ccc}
U & \xrightarrow{\ !\ } & 1 \\
{\scriptstyle j}\downarrow & & \downarrow{\scriptstyle \mathsf{true}} \\
X & \xrightarrow{\ \phi\ } & \Omega
\end{array}
$$

is a pullback.[9]

The subobject classifier, if it exists, is unique up to isomorphism (see [Gol79]) and it holds that

$$\mathsf{Sub}_{\mathcal{C}}(X) \cong \mathsf{Hom}_{\mathcal{C}}(X, \Omega).$$

Example 2.3.7. *In the category of sets, the subobject classifier is the set of truth values, i.e. $\Omega = \{0, 1\}$. The function true 'chooses' the true value, that is $\mathsf{true}(\star) = 1$. If $j : U \rightarrowtail X$, then $j(U) \subseteq X$ and we can identify $j(U)$ with U; the morphism $X \to \Omega$ is obviously the characteristic function of U, which makes the inner diagram commute. It's easy to see that the diagram is a pullback.*

$$
\begin{array}{ccc}
U & \xrightarrow{\ !\ } & \{\star\} \\
{\scriptstyle j}\downarrow & & \downarrow{\scriptstyle \mathsf{true}} \\
X & \xrightarrow{\ \chi_U\ } & \{0, 1\}
\end{array}
$$

We now start to investigate the existence of the subobject classifier in presheaf categories; first we introduce a useful concept which is the particular case of subobjects in functor categories.

Definition 2.3.8 (Subfunctor). *Let $\mathcal{C}$ be a small category, with presheaves $Q, P \in \hat{\mathcal{C}}$. Then Q is said to be a subfunctor of P if:*

---

[8]With a common abuse of language, we also say that m is a subobject when we really mean the equivalence class of m.

[9]We denote with '!' the unique morphism from an object to the terminal object 1.

1. *For each $I \in \mathcal{C}$, $Q(I) \subseteq P(I)$;*

2. *For each $f : J \to I$, $Q_f = P_f \restriction_{Q(I)}$.*

Remark 2.3.9. *If Q is a subfunctor of P, it's then clear that the inclusion $i : Q \hookrightarrow P$ is a monomorphism. Vice versa, if $\theta : Q \rightarrowtail P$ is a natural transformation which is a monomorphism, then $\theta(I) : Q(I) \to P(I)$ is injective. Let Q' be the functor defined by $Q'(I) = \operatorname{Im}\theta(I)$ on objects and by the induced maps on morphisms; Q' is equivalent as a subobject to Q and is a subfunctor of P.*

Suppose that there exists a subobject classifier Ω in the presheaf category $\hat{\mathcal{C}}$; then it must hold that

$$\mathsf{Sub}_{\hat{\mathcal{C}}}(\mathsf{Hom}_{\mathcal{C}}(-, I)) \cong \mathsf{Hom}_{\hat{\mathcal{C}}}(\mathsf{Hom}_{\mathcal{C}}(-, I), \Omega) \cong \Omega(I)$$

where the last step is justified by the Yoneda lemma (see appendix A). Therefore,
$$\Omega(I) \cong \{Q : Q \text{ is a subfunctor of } \mathsf{Hom}_{\mathcal{C}}(-, I)\}.$$

To better understand this definition, let us introduce an alternative presentation of subfunctors, called *sieves*.

Definition 2.3.10 (Sieve). *A sieve on $I \in \mathcal{C}$ is a set of morphisms with codomain I such that for all $h : K \to J$, $f \in S \implies hf \in S$.*

$$K \xrightarrow{\quad h \quad} J \xrightarrow{\quad f \quad} I$$

We now show that every sieve on I corresponds to a subfunctor of $\mathsf{Hom}_{\mathcal{C}}(-, I)$, and vice versa.

- If Q is a subfunctor of $\mathsf{Hom}_{\mathcal{C}}(-, I)$, then $S = \{f : \exists J \in \mathcal{C}\ f : J \to I, f \in Q(J)\}$ is a sieve; to see this, let $f \in S$, with $f : J \to I$, $f \in Q(J)$; if $k : K \to J$, then $Q_h : Q(J) \to Q(K)$, so that $Q_h(f) = hf \in Q(K)$ and $hf \in S$.

- If S is a sieve on I, let Q be the functor defined by $Q(J) = f \in S : f : J \to I \subseteq \mathsf{Hom}_{\mathcal{C}}(J, I)$ on objects and by the induced composition map on morphisms. Then Q is obviously a subfunctor of $\mathsf{Hom}_{\mathcal{C}}(-, I)$.

Therefore it makes sense to assert that

$$\begin{aligned}\Omega(I) &\cong \{Q : Q \text{ is a subfunctor of } \mathsf{Hom}_{\mathcal{C}}(-, I)\} \\ &\cong \{S : S \text{ is a sieve on } I\}.\end{aligned}$$

Remark 2.3.11. *If* $g : J \to I$ *then a subobject of* $\mathsf{Hom}_{\mathcal{C}}(-, I)$ *induces a subobject of* $\mathsf{Hom}_{\mathcal{C}}(-, J)$, *or in other words, a sieve* S *on* I *induces a sieve on* J *defined by*

$$S \cdot g = \{h : hg \in S\}.$$

Having already established that $\Omega(I) \cong \{S : S \text{ is a sieve on } I\}$, using this last remark we finally define the subobject classifier Ω con the presheaf category $\hat{\mathcal{C}}$.

Definition 2.3.12 (Subobject classifier on $\hat{\mathcal{C}}$).
Let $\Omega \in \hat{\mathcal{C}}$ *be defined by*

$$\Omega(I) = \{S : S \text{ is a sieve on } I\}$$

on objects and by

$$\Omega_g : \Omega(I) \to \Omega(J), \qquad S \mapsto S \cdot g$$

on morphisms $g : J \to I$.

The object Ω comes along with the natural transformation $\mathsf{true} : 1 \to \Omega$ defined by

$$\mathsf{true}_I : \{\star\} \to \Omega(I), \qquad \star \mapsto \mathsf{Hom}_{\mathcal{C}}(-, I),$$

which associates to each object its *maximal sieve*. It's easy to see that true is a natural transformation, since

$$
\begin{array}{ccc}
1 & \xrightarrow{\;\mathsf{true}_I\;} & \Omega(I) \qquad\qquad I \\
\downarrow{\scriptstyle \mathbf{1}} & & \downarrow{\scriptstyle \Omega_g} \qquad\qquad \uparrow{\scriptstyle g} \\
1 & \xrightarrow{\;\mathsf{true}_J\;} & \Omega(J) \qquad\qquad J
\end{array}
$$

$\Omega_g(\mathsf{true}_I(\star)) = \mathsf{true}_I(\star) \cdot g = \mathsf{true}_J(\star)$ holds.

We now prove that Ω really is the subobject classifier of $\hat{\mathcal{C}}$.

Theorem 2.3.13. *The object* Ω *just defined is a subobject classifier of* $\hat{\mathcal{C}}$.

Proof. Let $j : Q \rightarrowtail P$ be a monomorphism, so that Q is a subfunctor of P. If $f : J \to I$, then $P_f : P(I) \to P(J)$ is a function, so we put

$$\phi_I(x) = \{f \mid f : J \to I, P_f(x) \in Q(J)\}.$$

$$Q \xrightarrow{\quad ! \quad} 1$$

(diagram: $Q \xrightarrow{!} 1$, with vertical map j on the left from Q to P, vertical map true on the right from 1 to Ω, and $P \xrightarrow{\phi} \Omega$ along the bottom)

It's easily seen that:

- $\phi_I(x)$ is a sieve: let $f \in \phi_I(x)$, with $f : J \to I$ and $P_f(x) \in Q(J)$. Let $h : K \to J$ so that $P_{hf} : P(I) \to P(K)$; then

$$P_{hf}(x) = P_h(\underbrace{P_f(x)}_{\in Q(J)}) \in Q(I)$$

  since Q_h is the restriction of P_h to $Q(J)$. It follows that $hf \in \phi_I(x)$.

- ϕ is a natural transformation, as

$$
\begin{array}{ccc}
P(I) & \xrightarrow{\phi_I} & \Omega(I) \\
\downarrow{\scriptstyle P_g} & & \downarrow{\scriptstyle \Omega_g} \\
P(J) & \xrightarrow{\phi_J} & \Omega(J)
\end{array}
\qquad
\begin{array}{c}
I \\
\uparrow{\scriptstyle g} \\
J
\end{array}
$$

$$\Omega_g(\phi_I(x)) = \phi_I(x){\cdot}g = \{h : K \to J \mid hg \in \phi_I(x)\} = \{h : K \to J \mid P_{hg} \in Q(K)\}$$

  and

$$\phi_J(P_g(x)) = \{h : K \to J \mid P_f(P_g(x)) \in Q(K)\} = \{h : K \to J \mid P_{hg} \in Q(K)\}.$$

- The diagram is a pullback: if $i : R \rightarrowtail P$ is a monomorphism, then R is a subfunctor of Q and so by letting $x \in R(I) \subseteq P(I)$, we have (since the outer diagram commutes)

$$\mathsf{true}_I(\star) = \phi_I(x),$$

  that is $P_f(x) \in Q(J)$ for all $J \in \mathcal{C}$ and $f : J \to I$. Then $x \in Q(I)$, as $P_{\mathbf{1}_I}(x) = x \in Q(I)$, so that R is a subfunctor of Q and $m : R \rightarrowtail Q$ is the inclusion map. Indeed for the maps it holds that

$$Q_f \upharpoonright_{R(I)} = (P_f \upharpoonright_{Q(I)}) \upharpoonright_{R(I)} = P_f \upharpoonright_{(Q(I) \cap R(I))} = P_f \upharpoonright_{R(I)} = R_f.$$
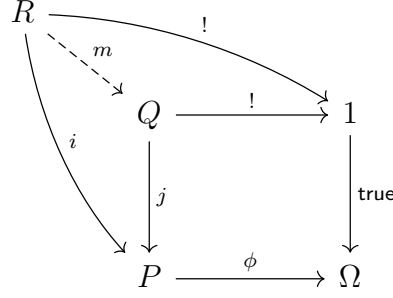
- ϕ is the unique natural transformation $\theta : P \to \Omega$ making the diagram a pullback: if $x \in P(I)$ and $f : J \to I$, then commutativity means that $P_f(x) \in Q(I)$ iff $\theta_J(P_f(x)) = \mathsf{true}_J(\star)$, i.e. $\theta_I(x) \cdot f = \mathsf{true}_J(\star)$ by naturality, which is equivalent to $f \in \theta_I(x)$. Therefore,

$$\theta_I(x) = \{f \mid f : J \to I, P_f(x) \in Q(J)\} = \phi_I(x)$$

and $\theta = \phi$.

$\square$

2.4 Modelling partial and restriction types

In this section we show how to model partial and restriction types in cubical sets.

First of all note that every cubical set X can be seen as a type $\Gamma \vdash X'$, putting $X'\rho = X(I)$ for each $I \in \mathcal{C}$ and $\rho \in \Gamma(I)$; for simplicity we shall write X for both.

The set of formulas induces a cubical set $\mathbb{F}$ defined by the sets $\mathbb{F}(I) = \{\varphi \mid \mathtt{vars}(\varphi) \subseteq I\}$ on objects and on morphisms $f : I \to J$ by $\mathbb{F}_f : \mathbb{F}(I) \to \mathbb{F}(J)$, $\varphi \mapsto \varphi f$ which substitutes the symbols $i \in I$ by $f(i) \in J$. Note that we can define a natural transformation $\hat{\cdot} : \mathbb{F} \to \Omega$ such that for any $I \in \mathcal{C}$, each $\varphi \in \mathbb{F}(I)$ is sent into the sieve $\hat{\varphi} \in \Omega(I)$ of all the $f : J \to I$ that make φ true, i.e. $\varphi f = \mathtt{True}$.

Now for each formula $\Gamma \vdash \varphi : \mathbb{F}$ we define a subsingleton type $\Gamma \vdash [\varphi]$ whose inhabitation corresponds to φ being true. Keeping in mind that for each $I \in \mathcal{C}$ and $\rho \in \Gamma(I)$ we have a $\varphi\rho \in \mathbb{F}(I)$, we put

$$[\varphi]\rho = \begin{cases} \{\star\} & \text{if } \varphi\rho = \mathtt{True} \\ \emptyset & \text{otherwise} \end{cases}$$

The restriction $[\varphi]\rho \to [\varphi](\rho f)$ is defined in the obvious manner.

Note what a context extension with a formula looks like:

$$(\Gamma.[\varphi])(I) = \{(\rho, \_) \mid \rho \in \Gamma(I), \_ \in [\varphi]\rho\} \cong \{\rho \mid \rho \in \Gamma(I), \varphi\rho = \mathtt{True}\}$$

This means that $\Gamma, [\mathtt{True}] = \Gamma$ and $\Gamma, [\mathtt{False}] = ()$.

Given a type $\Gamma \vdash A$ and a formula $\Gamma \vdash \varphi : \mathbb{F}$, we define $\Gamma \vdash [\varphi]A$ to be the function type $\Gamma \vdash [\varphi] \to A$. More explicitly, as explained in 2.2.1, each set $([\varphi]A)\rho$ is given by families

$$u = (u_f \in A(\rho f) \mid (\varphi\rho)f = \mathtt{True}, (u_f)g = u_{fg} \; \forall g : K \to J)_{f:J\to I}$$

Systems $\Gamma \vdash [\vec{\psi} \mapsto \vec{t}] : [\vec{\psi}]A$ are interpreted as follows: each $[\vec{\psi} \mapsto \vec{t}]\rho$ is given by the family

$$\left((t_i\rho)_f \mid (\psi_i\rho)f = \mathtt{True}, (t_i\rho)_f g = (t_i\rho)_{fg} \; \forall g : K \to J\right)_{f:J\to I, i=1\ldots n}$$

where each $(t_i\rho)_f$ is given by the term $\Gamma \vdash t_i : [\psi_i]A$; the uniqueness condition $\Gamma.[\psi_i \wedge \psi_j] \vdash t_i = t_j$, i.e. $(t_i\rho)_f = (t_j\rho)f$ for each $f : J \to I$ such that $\psi_i\rho f = \psi_j\rho f = \mathtt{True}$, assures that the definition is well posed.

Lastly, for restriction types of the form $\Gamma \vdash [\vec{\psi} \mapsto \vec{t}]A$, we define $([\vec{\psi} \mapsto \vec{t}]A)\rho$ to be the subset of the $u \in A\rho$ such that $uf = (t_i\rho)_f$ for each i and $f : J \to I$ such that $\psi_i\rho f = \mathtt{True}$. The restriction $([\vec{\psi} \mapsto \vec{t}]A)\rho \to ([\vec{\psi} \mapsto \vec{t}]A)(\rho h)$, for $h : I \to J$, is the induced map $A\rho \to A(\rho h)$, $u \mapsto uh$, because if $(\psi_i\rho h)f = \mathtt{True}$, then $(\psi_i\rho)(hf) = \mathtt{True}$ and so $(uh)f = u(hf) = (t_i\rho)_{hf} = (t_i\rho)(hf) = (t_i\rho h)f$.

As an example, we can justify the rules:

1.

$$\frac{\Gamma \vdash r : [\psi \mapsto s]\Sigma_A B}{\Gamma \vdash \pi_1 r : [\psi \mapsto \pi_1 s]A} \qquad \frac{\Gamma \vdash r : [\psi \mapsto s]\Sigma_A B}{\Gamma \vdash \pi_2 r : [\psi \mapsto \pi_2 s](B[\pi_1 r])}$$

as follows: $\Gamma \vdash r : [\psi \mapsto s]\Sigma_A B$ means that for each $I \in \mathcal{C}$ and $\rho \in \Gamma(I)$, we have an $r\rho \in (\Sigma_A B)\rho$ such that $r\rho f = (s\rho)_f$ for each $f : J \to I$ such that $\psi\rho f = \mathtt{True}$. Then $r\rho$ can be written as a pair $(r_1\rho, r_2\rho)$ with $r_1\rho \in A\rho$ and $r_2\rho \in B(\rho, r_1\rho)$. Moreover, by the formation rules it must hold that $\Gamma \vdash s : [\psi]\Sigma_A B$, that is for each $\rho \in \Gamma(I)$ we have a family

$$s\rho = (s\rho_f \in (\Sigma_A B)(\rho f) \mid (\psi\rho)f = \mathtt{True}, (s\rho_f)g = (s\rho)_{fg} \; \forall g : K \to J)_{f:J\to I}.$$

Each $s\rho_f$ can be written as a pair $(s_1\rho_f, s_2\rho_f)$, with $s_1\rho_f \in A\rho f$ and $s_2\rho_f \in B(\rho f, s_1\rho_f)$, from which we get two families $s_1\rho$ and $s_2\rho$; it follows

immediately that $\Gamma \vdash s_1 : [\psi]A$ and $\Gamma \vdash s_2 : [\psi](B[s_1])$. From $r\rho f = (s\rho)_f$ we get

$$r\rho f = (r_1\rho f, r_2\rho f) = (s\rho)_f = (s_1\rho_f, s_2\rho_f),$$

which implies $r_1\rho f = s_1\rho_f$ and $r_2\rho f = s_2\rho_f$. These equalities witness the judgements

$$\Gamma \vdash \pi_1 r : [\psi \mapsto \pi_1 s]A, \qquad \Gamma \vdash \pi_2 r : [\psi \mapsto \pi_2 s](B[\pi_1 r]).$$

2.

$$\frac{\Gamma \vdash f : [\varphi]\Pi_A B \quad \Gamma \vdash a : A}{\Gamma \vdash f\ a : [\varphi](B[a])}$$

From $\Gamma \vdash f : [\varphi]\Pi_A B$, we are given for each $I \in \mathcal{C}$ and $\rho \in \Gamma(I)$ a family

$$f\rho = (f\rho_h \in (\Pi_A B)(\rho h) \mid \varphi\rho h = \text{True}, (f\rho_h)g = f\rho_{hg}\ \forall g : K \to J)_{h:J \to I},$$

where by definition of $\Pi_A B$, each $f\rho_h$ is given by a family

$$f\rho_h = ((f\rho_h)_l \mid \forall u \in A(\rho h l)\ (f\rho_h)_l u \in B(\rho h l, u))_{j:K \to J}.$$

From $\Gamma \vdash a : A$, we are given for each $I \in \mathcal{C}$ and $\rho \in \Gamma(I)$ an element $a\rho \in A\rho$. We define $\Gamma \vdash b : [\varphi](B[a])$, putting

$$b\rho = (b\rho_h \mid \varphi\rho h = \text{True})_{h:J \to I}$$
$$b\rho_h = (f\rho_h)_{\mathbf{1}}(a\rho h) \in B(\rho h, a\rho h) = (B[a])\rho h$$

We have to verify that $(b\rho_h)g = b\rho_{hg}$:

$$(b\rho_h)g = ((f\rho_h)_{\mathbf{1}}(a\rho h))g = (f\rho_h)_g(a\rho hg) = (f\rho_{hg})(a\rho hg) = b\rho_{hg}.$$

Therefore, we get, $\Gamma \vdash b : [\varphi](B[a])$, which we define to be the interpretation of $f\ a$.

Chapter 3

Implementation

In this chapter we discuss the actual software implementation of the type theory described in chapter 1, using the Haskell programming language. The full code can be found at `https://github.com/mattia-furlan/CTT`.

We use the tools Alex and Happy to handle the lexical analyzer and the parser.[1] Instead of writing them from scratch, we initially wrote a simple grammar using the BNFC tool, which then automatically produces the files (one `.x` and one `.y`) which are given in input to Alex and Happy.[2]

The files which do all the main work are the following:[3]

- In `Interval.hs` we define formulas and the directions environment structure, along with all the utility functions needed to handle them (e.g. updating the directions environment, substitution in a formula).

- In `CoreCTT.hs` we define the abstract syntax of terms and values, the context and the functions which check term shadowing.

- In `Eval.hs` we implement the evaluation function for terms and formulas, with an handler for each type eliminator (e.g. function application, projections). Moreover, there we also implement the printing functions, the read-back function and the convertibility (i.e. $\alpha\eta$-conversion) predicate.

---

[1]Alex and Happy are the Haskell version of the celebrated Lex and Yacc for C.

[2]BNFC is useful especially for the lexer, as it does under the hood all the hard-work of treating special characters and Unicode. Moreover, from the basic parser produced by BNFC one obtains a sketch which will then be refined manually, handling for example the shift/reduce conflicts.

[3]Listed in order of dependencies (e.g. `TypeChecker.hs` needs `Interval.hs`, `CoreCTT.hs` and `Eval.hs`).

- In `TypeChecker.hs` we implement the bidirectional type-checker, with one function for type-inference and one for type-checking. We also handle type-checking under a formula.

- In `MainCTT.hs` we implement the main REPL loop which reads declarations from the input (or from some files) and processes them.

3.1 Lexical analysis and parsing

As explained in the introduction of the chapter, we use a lexer automatically generated by BNFC, which we don't report here due to the excess of technical details.[4] Instead we give here the code for the parser, initially generated by BNFC and then refined manually. The syntax that the parser recognizes is the one shown in chapter 1, allowing also multiple identifiers in abstractions (e.g. one can write $[a, b, c : A] B$ instead of $[a : A][b : A][c : A] B$).

The abstract syntax is defined in `CoreCTT.hs` for terms and top-levels, and in `Interval.hs` for formulas.

All the files use a wrapper data structure for strings, called `Ident`, defined in the homonym file.

Source file `Ident.hs`.

```
{-# LANGUAGE GeneralizedNewtypeDeriving #-}

module Ident where

import Data.String

newtype Ident = Ident String
  deriving (Eq, Ord, IsString)

instance Show Ident where
    show (Ident s) = s
```

Source file `ParCTT.y`.

```
{
module ParCTT ( happyError, myLexer, pProgram,pToplevel, pTerm )
where

import Prelude
import qualified Data.Map as Map
```

---

[4]The file is `LexCTT.x`.

```
import Ident
import qualified CoreCTT
import Interval
import LexCTT
}

%name pProgram Program
%name pToplevel Toplevel
%name pTerm Term
%monad { Err } { (>>=) } { return }
%tokentype {Token}
%token
  '(' { PT _ (TS _ 1) }
  ')' { PT _ (TS _ 2) }
  '*' { PT _ (TS _ 3) }
  '+' { PT _ (TS _ 4) }
  ',' { PT _ (TS _ 5) }
  '->' { PT _ (TS _ 6) }
  '.1' { PT _ (TS _ 7) }
  '.2' { PT _ (TS _ 8) }
  '/\\' { PT _ (TS _ 9) }
  '0' { PT _ (TS _ 10) }
  '1' { PT _ (TS _ 11) }
  ':' { PT _ (TS _ 12) }
  ';' { PT _ (TS _ 13) }
  '<' { PT _ (TS _ 14) }
  '=' { PT _ (TS _ 15) }
  '>' { PT _ (TS _ 16) }
  'I' { PT _ (TS _ 17) }
  'N' { PT _ (TS _ 18) }
  'S' { PT _ (TS _ 19) }
  'U' { PT _ (TS _ 20) }
  'Z' { PT _ (TS _ 21) }
  '[' { PT _ (TS _ 22) }
  '\\/' { PT _ (TS _ 23) }
  ']' { PT _ (TS _ 24) }
  'comp' { PT _ (TS _ 25) }
  'ind' { PT _ (TS _ 26) }
  'inl' { PT _ (TS _ 27) }
  'inr' { PT _ (TS _ 28) }
  'split' { PT _ (TS _ 29) }
  '|' { PT _ (TS _ 30) }
  L_Ident  { PT _ (TV $$) }

%%

Ident :: { Ident }
Ident  : L_Ident { Ident $1 }
```

```
ListIdent :: { [Ident] }
ListIdent  : Ident { [$1] }
           | Ident ',' ListIdent { (:) $1 $3 }


Program :: { CoreCTT.Program }
Program : ListToplevel { CoreCTT.Program $1 }


Term :: { CoreCTT.Term }
Term : Term1 '->' Term { CoreCTT.Abst (Ident "") $1 $3 }
     | '['  Ident ':' Term '=' Term ']' Term { CoreCTT.TDef ($2,$4,$6) $8 }
     | '[' Ident ':' Term ']' Term { CoreCTT.Abst $2 $4 $6 }
     | '[' ListIdent ':' Term ']' Term
       { foldr (\i e -> CoreCTT.Abst i $4 e) $6 $2 }
     | Term1 { $1 }


Term1 :: { CoreCTT.Term }
Term1 : Term2 '+' Term1 { CoreCTT.Sum $1 $3 }
      | Term2 '*' Term1 { CoreCTT.Sigma (Ident "") $1 $3 }
      | '<' ListIdent ':' Term '>' Term
        { foldr (\i e -> CoreCTT.Sigma i $4 e) $6 $2 }
      | '[' DisjFormula ']' Term1 { CoreCTT.Partial $2 $4 }
      | System Term1 { CoreCTT.Restr $1 $2 }
      | Term2 ',' Term2 { CoreCTT.Pair $1 $3 }
      | Term2 { $1 }


Term2 :: { CoreCTT.Term }
Term2 : Term2 Term3 { CoreCTT.App $1 $2 }
      | 'ind' Term3 Term3 Term3 Term3 { CoreCTT.Ind $2 $3 $4 $5 }
      | 'comp' Term3 '(' DisjFormula ')' Term3 Term3 Term3 Term3
        { CoreCTT.Comp $2 $4 $6 $7 $8 $9 }
      | 'comp' Term3 '(' ')' Term3 Term3 Term3 Term3
        { CoreCTT.Comp $2 fFalse $5 $6 $7 $8 }
      | 'S' Term3 { CoreCTT.Succ $2 }
      | 'inl' Term3 { CoreCTT.InL $2 }
      | 'inr' Term3 { CoreCTT.InR $2 }
      | 'split' Term3 Term3 Term3 Term3 { CoreCTT.Split $2 $3 $4 $5 }
      | Term3 { $1 }


Term3 :: { CoreCTT.Term }
Term3 : Ident { CoreCTT.Var $1 }
      | 'U' { CoreCTT.Universe }
      | 'N' { CoreCTT.Nat }
      | 'Z' { CoreCTT.Zero }
      | 'I' { CoreCTT.I }
      | Term3 '.1' { CoreCTT.Fst $1 }
      | Term3 '.2' { CoreCTT.Snd $1 }
      | System { CoreCTT.Sys $1 }
      | '0' { CoreCTT.I0 }
      | '1' { CoreCTT.I1 }
```

```
        | '(' Term ')' { $2 }


Toplevel :: { CoreCTT.Toplevel }
Toplevel : Ident ':' Term '=' Term { CoreCTT.Definition $1 $3 $5 }
         | Ident ':' Term { CoreCTT.Declaration $1 $3 }
         | Term { CoreCTT.Example $1 }

ListToplevel :: { [CoreCTT.Toplevel] }
ListToplevel : {- empty -} { [] }
             | Toplevel ';' ListToplevel { (:) $1 $3 }


AtomicFormula :: { AtomicFormula }
AtomicFormula : Ident '=' '0' { Eq0 $1 }
              | Ident '=' '1' { Eq1 $1 }
              | Ident '=' Ident { Diag $1 $3 }

ConjFormula1 :: { [AtomicFormula] }
ConjFormula1 : AtomicFormula { [$1] }
             | AtomicFormula '/\\' ConjFormula1 { $1 : $3 }

ConjFormula :: { ConjFormula }
ConjFormula : ConjFormula1 { Conj $1 }
            | '(' ConjFormula ')' { $2 }

DisjFormula1 :: { [ConjFormula] }
DisjFormula1 : ConjFormula { [$1] }
             | ConjFormula '\\/' DisjFormula1 { $1 : $3 }

DisjFormula :: { DisjFormula }
DisjFormula : DisjFormula1 { Disj $1 }

System :: { CoreCTT.System }
System : '[' ListSysElem ']' { $2 }

SysElem :: { (ConjFormula,CoreCTT.Term) }
SysElem : ConjFormula '->' Term { ($1,$3) }

ListSysElem :: { [(ConjFormula,CoreCTT.Term)] }
ListSysElem : {- empty -} { [] }
            | SysElem { [$1] }
            | SysElem '|' ListSysElem { $1 : $3 }

{
type Err = Either String

happyError :: [Token] -> Err a
happyError ts = Left $
```

```
    "syntax error at " ++ tokenPos ts ++
    case ts of
      []      -> []
      [Err _] -> " due to lexer error"
      t:_     -> " before `" ++ (prToken t) ++ "'"

myLexer :: String -> [Token]
myLexer = tokens
}
```

3.2 Interval and formulas

In this file first we define three types of formulas: atomic ones (of the kind
$i = 0$, $i = 1$ or $i = j$), conjunctive ones (that is, lists of atomic formulas)
and disjunctive ones (that is, lists of conjunctive formulas).[5] Next we define
the directions environment structure, called `DirEnv`, made of three fields: one
storing the variables set to zero, one storing the variables set to one, and
the last one storing the equivalence classes of 'diagonals' (see 1.2.3). We then
implement some utility functions needed to work with directions environments,
as adding a conjunction, checking if such a structure makes a formula true,
etc.

Source file `Interval.hs`.

```
{-# LANGUAGE FlexibleInstances #-}

module Interval where

import Data.List (intercalate,delete)
import Ident

-- Atomic formulas are of the kind `i = 0`, `i = 1` or `i = j`
data AtomicFormula
  = Eq0 Ident
  | Eq1 Ident
  | Diag Ident Ident
  deriving (Ord)

-- Equality between atomic formulas
instance Eq AtomicFormula where
  af1 == af2 = case (af1,af2) of
    (Eq0 s1,Eq0 s2) -> s1 == s2
    (Eq1 s1,Eq1 s2) -> s1 == s2
    (Diag s1 s2,Diag s3 s4) -> (s1 == s3 && s2 == s4)
```

---

[5]More precisely, the last type are the formulas in disjunctive normal form.

```
        || (s1 == s4 && s2 == s3)
    otherwise -> False

-- A conjunctive formula is a list of atomic formulas
newtype ConjFormula = Conj [AtomicFormula]
  deriving (Eq,Ord)

-- A disjunctive formula is a list of conjunctive formulas
newtype DisjFormula = Disj [ConjFormula]
  deriving (Eq,Ord)

{- Pretty printing of atomic/conjunctive/disjunctive formulas -}

instance Show AtomicFormula where
  show af = case af of
    Eq0 s -> show s ++ " = 0"
    Eq1 s -> show s ++ " = 1"
    Diag s1 s2 -> show s1 ++ " = " ++ show s2

instance Show ConjFormula where
  show (Conj cf)
    | null cf   = "True"
    | otherwise =  intercalate " /\\ " (map show cf)

instance Show DisjFormula where
  show disj@(Disj df)
    | disj == fFalse = "False"
    | disj == fTrue  = "True"
    | otherwise      = intercalate " \\/ " $
      map (\cf -> "(" ++ show cf ++ ")") df

{- Helpers -}

-- True (disjunctive) formula: just one empty conjunction
fTrue :: DisjFormula
fTrue = Disj [Conj []]

-- False formula: empty disjunction
fFalse :: DisjFormula
fFalse = Disj []

isTrue :: DisjFormula -> Bool
isTrue = (== fTrue)

isFalse :: DisjFormula -> Bool
isFalse = (== fFalse)

{- Implication and equivalence -}
```

```haskell
{- A disjunctive formula implies another one if each of its conjunctions
   makes the second formula true. The case where the second formula is false
   must be handled separately. The first two checks are unnecessary, used only
   for efficiency -}
impDisj :: DirEnv -> DisjFormula -> DisjFormula -> Bool
impDisj dirs (Disj df1) disj2 = isFalse (Disj df1) || isTrue disj2 ||
  if isFalse disj2 then
    all (inconsistent . addConj dirs) df1 -- First formula must be false
  else
    all (\cf1 -> addConj dirs cf1 `makesTrueDisj` disj2) df1

-- Two formulas are equal is each one imply the other
eqFormulas :: DirEnv -> DisjFormula -> DisjFormula -> Bool
eqFormulas dirs disj1 disj2 = impDisj dirs disj1 disj2 && impDisj dirs disj2 disj1

{- Directions environment -}

-- A `DirEnv` stores the list of zeros, ones and the diagonals partitions
type DirEnv = ([Ident],[Ident],[[Ident]])

emptyDirEnv :: DirEnv
emptyDirEnv = ([],[],[])

-- A `DirEnv` is inconsistent if there is an identifier
-- which is set both to zero and one
inconsistent :: DirEnv -> Bool
inconsistent (zeros,ones,diags) =
  any (`elem` ones) zeros || any (`elem` zeros) ones

-- Find the partition which contains `s`, if it exists.
-- Otherwise return the fake partition [s]
findPartition :: [[Ident]] -> Ident -> [Ident]
findPartition diags s = case filter (s `elem`) diags of
    [] -> [s]
    l  -> head l

-- Set an identifier to zero. Any eventual identifier which is in the same
-- partition must then be set to zero, and that partition is removed
addZero :: DirEnv -> Ident -> DirEnv
addZero (zeros,ones,diags) s =
  let toadd = findPartition diags s
  in (toadd ++ zeros,ones,delete toadd diags)

-- Set an identifier to one. Any eventual identifier which is in the same
-- partition must then be set to one, and that partition is removed
addOne :: DirEnv -> Ident -> DirEnv
addOne (zeros,ones,diags) s =
  let toadd = findPartition diags s
  in (zeros,toadd ++ ones,delete toadd diags)
```

```haskell
-- Add a new diagonal `s1 = s2`
addDiag :: DirEnv -> Ident -> Ident -> DirEnv
addDiag dirs@(zeros,ones,diags) s1 s2
  | s1 == s2         = dirs              -- Trivial, nothing to do
  | s1 `elem` zeros = addZero dirs s2 -- `s1` already zero -> set `s2` to zero
  | s2 `elem` zeros = addZero dirs s1 -- `s2` already zero -> set `s1` to zero
  | s1 `elem` ones  = addOne dirs s2  -- `s1` already one  -> set `s2` to one
  | s2 `elem` ones  = addOne dirs s1  -- `s2` already one  -> set `s1` to zero
  | otherwise = let
    -- Add `s1` and `s2` to the existing partitions if it's the case:
    -- it means that e.g if partition `set` contains `s1`, then `s2`
    -- shall be added to `set` too
    diags' = [if s1 `elem` set then s2 : set else if s2 `elem` set then s1 : set
        else set | set <- diags]
    -- Add a new partition if `s1` and `s2` are new (= not found in the partitions)
    diags'' = diags' ++
      [[s1,s2] | not (s1 `elem` concat diags' || s2 `elem` concat diags')]
    par1 = findPartition diags'' s1
    par2 = findPartition diags'' s2
    -- Eventually join the two partitions
    -- (e.g. [i,k] [j,k,l] gets joined into [i,j,k,l])
    diags''' = if par1 /= par2 then
        delete par2 (delete par1 diags'') ++ [par1 ++ par2]
      else
        diags''
  in (zeros,ones,diags''')

-- Add a conjunction to a `DirEnv`
addConj :: DirEnv -> ConjFormula -> DirEnv
addConj dirs (Conj conj) = foldl addAtomic dirs conj
  where
    addAtomic :: DirEnv -> AtomicFormula -> DirEnv
    addAtomic dirs' ff = case ff of
      Eq0 s    -> addZero dirs' s
      Eq1 s    -> addOne dirs' s
      Diag s s' -> addDiag dirs' s s'

-- Conversion from a conjunction to a `DirEnv`
conjToDirEnv :: ConjFormula -> DirEnv
conjToDirEnv = addConj emptyDirEnv

-- Test if a `DirEnv` makes an atomic formula true.
-- A diagonal is true iff both are zero, or both are true, or if
-- they are in the same partition
makesTrueAtomic :: DirEnv -> AtomicFormula -> Bool
(zeros,ones,diags) `makesTrueAtomic` af = case af of
  Eq0 s -> s `elem` zeros
  Eq1 s -> s `elem` ones
```

```haskell
  Diag s1 s2 -> s1 == s2 || bothIn zeros || bothIn ones || any bothIn diags
    where bothIn set = s1 `elem` set && s2 `elem` set

-- A conjunction is true iff all of its atomic formulas are true
makesTrueConj :: DirEnv -> ConjFormula -> Bool
makesTrueConj dirs (Conj cf) = all (dirs `makesTrueAtomic`) cf

-- A disjunction is true iff one of its conjunctive formulas is true
makesTrueDisj :: DirEnv -> DisjFormula -> Bool
makesTrueDisj dirs (Disj df) = any (dirs `makesTrueConj`) df

-- Substitute `s'` for `s` in an atomic formula
substAtomic :: (Ident,Ident) -> AtomicFormula -> AtomicFormula
substAtomic (s,s') af = case af of
  Eq0 x | s == x -> Eq0 s'
  Eq1 x | s == x -> Eq1 s'
  Diag x y -> Diag (if x == s then s' else x) (if y == s then s' else y)
  otherwise -> af

-- Substitute into each atomic formula of the conjunction
substConj :: (Ident,Ident) -> ConjFormula -> ConjFormula
substConj (s,s') (Conj cf) = Conj $ map (substAtomic (s,s')) cf

-- Concatenation (logical AND) between two conjunctive formulas
meet :: ConjFormula -> ConjFormula -> ConjFormula
(Conj cf1) `meet` (Conj cf2) = Conj $ cf1 ++ cf2
```

3.3   Core language

In this file we first define the abstract syntax for terms/values and for top-levels
(i.e. declarations, definitions and examples). We then define some functions
to handle these syntactical aspects, e.g. getting free variables and checking
eventual term shadowing. Next we define contexts as lists of pairs of the
form (s,Decl ty), (s,Def ty def) or (s,Val v), along with some utility
functions.

Source file CoreCTT.hs.

```haskell
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE LambdaCase #-}

module CoreCTT where

import Data.Maybe (fromJust)

import Ident
import Interval
```

```haskell
{- Syntax (terms/values) -}

data Term
  = Var Ident
  | Universe
  {- Let-definition `[x:ty = e]t` -}
  | TDef (Ident,Term,Term) Term
  {- Π types -}
  | Abst Ident Term Term
  | App Term Term
  {- Σ types -}
  | Sigma Ident Term Term
  | Pair Term Term
  | Fst Term
  | Snd Term
  {- Coproducts -}
  | Sum Term Term
  | InL Term
  | InR Term
  --       fam   f_0   f_1    x
  | Split Term Term Term Term
  {- Naturals -}
  | Nat
  | Zero
  | Succ Term
  --     fam   c_0   c_s     n
  | Ind Term Term Term Term
  {- Cubical -}
  | I | I0 | I1
  | Sys System
  | Partial DisjFormula Term
  | Restr System Term
  --      fam       phi      i0    u    base   i
  | Comp Term DisjFormula Term Term Term Term
  {- For values only: -}
  | Closure Term Ctx
  --          val   type
  | Neutral Value Value
  deriving (Eq, Ord)

type Value = Term

newtype Program = Program [Toplevel]

data Toplevel = Definition Ident Term Term
              | Declaration Ident Term
              | Example Term
  deriving (Eq, Ord)
```

```
-- Generate a fresh name starting from 'x'
newVar :: [Ident] -> Ident -> Ident
newVar used x
  | x `elem` used = newVar used (Ident $ show x ++ "'")
  | otherwise     =  x

-- For printing purposes: e.g. collectApps ((App (App f x_1) x_2) x_3) []
-- returns (f,[x_1,x_2,x_3])
collectApps :: Term -> [Term] -> (Term,[Term])
collectApps t apps = case t of
  App t1 t2' -> collectApps t1 (t2' : apps)
  otherwise  -> (t,apps)

-- Generic class for objects (terms,values,top-levels,formulas,etc.)
-- which contain variables
class SyntacticObject a where
  containsVar :: Ident -> a -> Bool
  containsVar s x = s `elem` freeVars x
  vars :: a -> [Ident]
  freeVars :: a -> [Ident]

instance SyntacticObject Ident where
  vars s = [s]
  freeVars s = [s]

instance SyntacticObject System where
  vars sys = concatMap vars (keys sys) ++ concatMap vars (elems sys)
  freeVars = vars

-- For terms only and not for values (which means we don't
-- define `vars` and `freeVars` for closures and neutral values)
instance SyntacticObject Term where
  vars = \case
    Var s                -> [s]
    Universe             -> []
    TDef (s,t,e) t'      -> s : vars t ++ vars e ++ vars t'
    Abst s t e           -> s : vars t ++ vars e
    App fun arg          -> vars fun ++ vars arg
    Sigma s t e          -> s : vars t ++ vars e
    Pair t1 t2           -> vars t1 ++ vars t2
    Fst t                -> vars t
    Snd t                -> vars t
    Sum ty1 ty2          -> vars ty1 ++ vars ty2
    InL t1               -> vars t1
    InR t2               -> vars t2
    Split ty f1 f2 x     -> vars ty ++ vars f1 ++ vars f2 ++ vars x
    Nat                  -> []
    Zero                 -> []
```

```
    Succ t                  -> vars t
    Ind ty b s n            -> vars ty ++ vars b ++ vars s ++ vars n
    I                       -> []
    I0                      -> []
    I1                      -> []
    Sys sys                 -> vars sys
    Partial phi t           -> vars phi ++ vars t
    Restr sys t             -> vars sys ++ vars t
    Comp fam phi i0 u b i -> vars fam ++ vars phi ++ vars i0 ++ vars u
      ++ vars b ++ vars i

  freeVars = \case
    Var s                   -> [s]
    Universe                -> []
    TDef (s,t,e) t'         -> freeVars t ++ filter (/= s)
      (freeVars e ++ freeVars t')
    Abst s t e              -> freeVars t ++ filter (/= s) (freeVars e)
    App fun arg             -> freeVars fun ++ freeVars arg
    Sigma s t e             -> freeVars t ++ filter (/= s) (freeVars e)
    Pair t1 t2              -> freeVars t1 ++ freeVars t2
    Fst t                   -> freeVars t
    Snd t                   -> freeVars t
    Sum ty1 ty2             -> freeVars ty1 ++ freeVars ty2
    InL t1                  -> freeVars t1
    InR t2                  -> freeVars t2
    Split ty f1 f2 x        -> freeVars ty ++ freeVars f1 ++ freeVars f2
      ++ freeVars x
    Nat                     -> []
    Zero                    -> []
    Succ t                  -> freeVars t
    Ind ty b s n            -> freeVars ty ++ freeVars b ++ freeVars s
      ++ freeVars n
    I                       -> []
    I0                      -> []
    I1                      -> []
    Sys sys                 -> freeVars sys
    Partial phi t           -> freeVars phi ++ freeVars t
    Restr sys t             -> freeVars sys ++ freeVars t
    Comp fam phi i0 u b i -> freeVars fam ++ freeVars phi ++ freeVars i0
      ++ freeVars u ++ freeVars b ++ freeVars i

instance SyntacticObject AtomicFormula where
  vars af = case af of
    Eq0 s       -> [s]
    Eq1 s       -> [s]
    Diag s1 s2  -> [s1,s2]
  freeVars = vars

instance SyntacticObject ConjFormula where
```

```
  vars (Conj cf) = concatMap vars cf
  freeVars = vars

instance SyntacticObject DisjFormula where
  vars (Disj df) = concatMap vars df
  freeVars = vars

checkTermShadowing :: [Ident] -> Term -> Bool
checkTermShadowing used term = case term of
  Var _              -> True
  Universe           -> True
  TDef (s,t,e) t'    ->
    s `notElem` used && checkTermShadowing used t &&
    checkTermShadowing (if s == Ident "" then used else s : used) e &&
    checkTermShadowing (if s == Ident "" then used else s : used) t'
  Abst s t e         -> s `notElem` used && checkTermShadowing used t &&
        checkTermShadowing (if s == Ident "" then used else s : used) e
  App fun arg        -> checkTermShadowing used fun &&
    checkTermShadowing used arg
  Sigma s t e        -> s `notElem` used && checkTermShadowing used t &&
    checkTermShadowing (if s == Ident "" then used else s : used) e
  Pair t1 t2         -> checkTermShadowing used t1 &&
    checkTermShadowing used t2
  Fst t              -> checkTermShadowing used t
  Snd t              -> checkTermShadowing used t
  Sum ty1 ty2        -> checkTermShadowing used ty1 &&
    checkTermShadowing used ty2
  InL t1             -> checkTermShadowing used t1
  InR t2             -> checkTermShadowing used t2
  Split ty f1 f2 x   ->
    checkTermShadowing used ty && checkTermShadowing used f1 &&
    checkTermShadowing used f2 && checkTermShadowing used x
  Nat                -> True
  Zero               -> True
  Succ n             -> checkTermShadowing used n
  Ind ty b s n       ->
    checkTermShadowing used ty && checkTermShadowing used b &&
    checkTermShadowing used s  && checkTermShadowing used n
  I                  -> True
  I0                 -> True
  I1                 -> True
  Sys sys            -> all (checkTermShadowing used) (elems sys)
  Partial _ t        -> checkTermShadowing used t
  Restr sys t        -> all (checkTermShadowing used) (elems sys) &&
    checkTermShadowing used t
  Comp fam _ i0 u b i  ->
    checkTermShadowing used fam && checkTermShadowing used i0 &&
    checkTermShadowing used u   && checkTermShadowing used b   &&
    checkTermShadowing used i
```

```
    otherwise -> error "[checkTermShadowing] got non-term"

{- Printing functions are in 'Eval.hs' -}

type ErrorString = String

{- Generic association lists utilities -}

extend :: Ctx -> Ident -> CtxEntry -> Ctx
extend ctx s e = if s == Ident "" then ctx else (s,e) : ctx

keys :: [(k,a)] -> [k]
keys = map fst

elems :: [(k,a)] -> [a]
elems = map snd

at :: (Eq k) => [(k,a)] -> k -> a
al `at` s = fromJust $ lookup s al

{- Contexts -}

type Ctx = [(Ident,CtxEntry)]

data CtxEntry = Decl Term       -- Type
              | Def Term Term   -- Type and definition
              | Val Value       -- Value binding for `eval`
  deriving (Eq, Ord)

emptyCtx :: Ctx
emptyCtx = []

-- Extract the value bindings from a context
getBindings :: Ctx -> [(Ident,Value)]
getBindings = concatMap $
  \(s,entry) -> case entry of Val v -> [(s,v)]; _ -> []

-- Shall not be called with values in the context
-- (it is used only in `removeFromCtx`)
instance SyntacticObject CtxEntry where
  vars entry = case entry of
    Decl t      -> vars t
    Def ty def -> vars ty ++ vars def
  freeVars entry = case entry of
    Decl t      -> freeVars t
    Def ty def -> freeVars ty ++ freeVars def

-- Remove an identifier from the context and also all the others
-- (recursively) which depend on it
```

```haskell
removeFromCtx :: Ctx -> Ident -> Ctx
removeFromCtx ctx s = if s `elem` keys ctx then
    let dep  = map fst $ filter (\(_,entry) -> s `elem` freeVars entry) ctx
        ctx' = filter (\(s',_) -> s /= s') ctx
    in foldl removeFromCtx ctx' dep
  else
    ctx


{- Systems -}


type System = [(ConjFormula,Term)]

-- Get the disjunction of the (conjunctive) formulas of the system
getSystemFormula :: System -> DisjFormula
getSystemFormula = Disj . map fst

-- Utility `map` function for systems: it applies a function
-- to the values inside the system
mapSys ::  (Value -> Value) -> System -> System
mapSys f = map (\(psi,v) -> (psi,f v))

-- Split a type into the form [phi]A, with `phi` eventually trivial
-- It is used in the function `compTypes` of `TypeChecker.hs`
split :: Value -> (DisjFormula,Value)
split v = case v of
  Partial phi ty -> (phi,ty)
  Restr   _    ty -> (fTrue,ty)
  otherwise       -> (fTrue,v)
```

3.4   Evaluation

In this file we first define the evaluation function, by recursion on the structure
of terms; we also handle evaluation of formulas. Next we implement a function
for each type eliminator (e.g. `doApp` for function application, `doSplit` for
the coproduct types eliminator), which have to handle neutral values with
eventually a partial or restriction type (see 1.4). Lastly we implement the read-
back function, the printing utilities and the conversion (i.e. $\alpha\eta$-conversion)
predicate between two values.

Source file `Eval.hs`.

```haskell
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE LambdaCase #-}
{-# LANGUAGE BlockArguments #-}

module Eval where
```

```haskell
import Data.List (intercalate,nub,find)
import Data.Foldable (foldrM)
import Data.Maybe (mapMaybe,fromJust)

import Ident
import Interval
import CoreCTT

import Debug.Trace

-- For debug purposes only
debug :: Bool
debug = False

myTrace :: String -> a -> a
myTrace s x = if debug then trace s x else x

-- Retrieve the evaluated type of a variable from the context
lookupType :: Ident -> Ctx -> Value
lookupType s [] = error $ "[lookupType] got unknown identifier " ++ show s
lookupType s ((s',entry):ctx) = if s == s' then
    case entry of
      Decl ty    -> eval ctx ty
      Def  ty _  -> eval ctx ty
      Val _      -> lookupType s ctx
  else
    lookupType s ctx

-- Evaluate a term in the given context
eval :: Ctx -> Term -> Value
eval ctx term = case term of
  Var s -> case lookup s ctx of
    Nothing        -> error $ "[eval] not found var `" ++ show s ++ "` in ctx"
    Just (Val v)   -> v
    Just (Decl ty) -> simplNeutralValue $ Neutral (Var s) (eval ctx ty)
    Just (Def _ e) -> eval ctx e
  Universe          -> Universe
  TDef (s,t,e) t'   -> eval (extend ctx s (Def t e)) t'
  Abst{}            -> Closure term ctx
  App e1 e2         -> doApply (eval ctx e1) (eval ctx e2)
  Sigma{}           -> Closure term ctx
  Pair t1 t2        -> Pair (eval ctx t1) (eval ctx t2)
  Fst t             -> doFst (eval ctx t)
  Snd t             -> doSnd (eval ctx t)
  Sum ty1 ty2       -> Sum (eval ctx ty1) (eval ctx ty2)
  InL t1            -> InL (eval ctx t1)
  InR t2            -> InR (eval ctx t2)
  Split ty f1 f2 x  ->
    doSplit (eval ctx ty) (eval ctx f1) (eval ctx f2) (eval ctx x)
```

```
  Nat                    -> Nat
  Zero                   -> Zero
  Succ t                 -> Succ (eval ctx t)
  Ind ty base step n -> doInd (eval ctx ty) (eval ctx base)
    (eval ctx step) (eval ctx n)
  I  -> I
  I0 -> I0
  I1 -> I1
  Sys sys                  -> simpl emptyDirEnv (Sys $ evalSystem ctx sys)
  Partial phi t            -> foldPartial (evalDisjFormula ctx phi) (eval ctx t)
  Restr sys t              -> foldRestr (evalSystem ctx sys) (eval ctx t)
  Comp fam phi i0 u b i -> doComp ctx fam phi i0 u b i
  otherwise                -> error $ "[eval] got " ++ show term

-- Evaluate a conjunctive formula
evalConjFormula :: Ctx -> ConjFormula -> Maybe ConjFormula
evalConjFormula ctx conj = conj'
  where
    -- Get the bindings which concern the formula's variables
    entries' = filter (\(s,_) -> s `elem` vars conj) (getBindings ctx)
    -- Get only the last one of each binding
    entries = map (\s -> fromJust $ find (\(s',_) -> s' == s) entries')
      (nub $ keys entries')
    -- Get the renamings from the entries (i.e. the ones of the form i -> j)
    renamings = concatMap (\case { (s,Neutral (Var s') I) -> [(s,s')] ;
      _ -> []}) entries
    -- Apply renamings to the conjuction
    renamedConj = foldr substConj conj renamings
    -- Apply value substitutions to the renamed conjuction
    vals       = filter (\(_,v) -> v == I0 || v == I1) entries
    conj'      = foldrM evalConj renamedConj vals

-- Evaluate a single conjuction by replacing `s` with 0
-- Returns `Nothing` if the resulting formula is false
evalConj :: (Ident,Value) -> ConjFormula -> Maybe ConjFormula
evalConj (s,I0) conj@(Conj cf) =
  if conjToDirEnv conj `makesTrueAtomic` Eq1 s -- Inconsistent cases
    || inconsistent (conjToDirEnv conj') then
    Nothing
  else
    Just conj' -- Substitute into each atomic formula
  where
    conj' = Conj . nub $ concatMap (\case
      Eq0 s' | s == s' -> [];
      Diag s1 s2 -> if s == s1 && s1 == s2 then []
        else (if s == s1 then Eq0 s2
          else if s == s2 then Eq0 s1 else Diag s1 s2);
      af -> [af]) cf
```

```haskell
-- Same as before, now replacing `s` with 1
evalConj (s,I1) conj@(Conj cf) =
  if conjToDirEnv conj `makesTrueAtomic` Eq0 s -- Inconsistent cases
    || inconsistent (conjToDirEnv conj') then
    Nothing
  else
    Just conj' -- Substitute into each atomic formula
  where
    conj' = Conj . nub $ concatMap (\case
      Eq1 s' | s == s' -> [];
      Diag s1 s2 -> if s == s1 && s1 == s2 then []
        else [if s == s1 then Eq1 s2
          else if s == s2 then Eq1 s1 else Diag s1 s2];
      af -> [af]) cf

-- Evaluate a disjunctive formula, by first checking if it contains
-- a true conjunction
evalDisjFormula :: Ctx -> DisjFormula -> DisjFormula
evalDisjFormula ctx (Disj df) = if Conj [] `elem` df then
    fTrue
  else -- Otherwise evaluate each conjunction, disregarding the false ones
    Disj $ mapMaybe (evalConjFormula ctx) df

-- Helper function, used below to delete duplicates
-- in restriction and partial types
nubFst :: Eq a => ([a],b) -> ([a],b)
nubFst (as,b) = (nub as,b)

-- Simplify cascading restriction types into one restriction type
-- E.g. [psi_1 -> t_1]([psi_2 -> t_2]([psi_3 -> t_3]A)) becomes
-- [psi_1 -> t_1 | psi_2 -> t_2 | psi_3 -> t_3]A
-- I call `nubFst` to delete duplicate formulas
foldRestr :: System -> Value -> Value
foldRestr sys0 v0 = uncurry Restr . nubFst $ foldRestr' sys0 v0
    where
        foldRestr' :: System -> Value -> (System,Value)
        foldRestr' sys v = case v of
            Restr sys' v' -> foldRestr' (sys ++ sys') v'
            otherwise     -> (sys,v)

-- Simplify cascading partial types into one partial type
-- E.g. [phi_1]([phi_2]([phi_3]A)) becomes
-- [phi /\ psi_2 /\ psi_3]A
-- I call `nubFst` to delete duplicate formulas
foldPartial :: DisjFormula -> Value -> Value
foldPartial (Disj df0) v0 = (\(df,v) -> Partial (Disj df) v) .
    nubFst $ foldPartial' df0 v0
    where
        foldPartial' :: [ConjFormula] -> Value -> ([ConjFormula],Value)
```

```
        foldPartial' df v = case v of
            Partial (Disj df') v' -> foldPartial' (dnf df' df) v'
            otherwise             -> (df,v)
        -- Get the conjuction of the two disjunctive formulas,
        -- in disjunctive normal form
        dnf :: [ConjFormula] -> [ConjFormula] -> [ConjFormula]
        dnf df1 df2 = [cf1 `meet` cf2 | cf1 <- df1, cf2 <- df2]

-- Evaluate a system
evalSystem :: Ctx -> System -> System
evalSystem ctx sys =
  concatMap (\(phi,t) -> evalConjFormula' phi (eval ctx t)) sys
  where
    -- Need to handle the case of false formulas
    evalConjFormula' phi v = case evalConjFormula ctx phi of
      Nothing -> []
      Just cf -> [(cf,v)]

-- Split the Abst/Sigma constructor and the arguments from a value
-- inside a closure
extract :: Value -> (Ident -> Term -> Term -> Value,Ident,Term,Term)
extract (Abst  s t e) = (Abst ,s,t,e)
extract (Sigma s t e) = (Sigma,s,t,e)
extract v             = error $ "[extract] got " ++ show v

-- Evaluate a closure, extending the context by assigning the variable to
-- the given value. In case of non-dependent abstractions, i.e. empty
-- variable, we don't need to extend the context
evalClosure :: Value -> Value -> Value
evalClosure (Closure (Abst  s _ e) ctx) arg =
  eval (if s == Ident "" then ctx else extend ctx s (Val arg)) e
evalClosure (Closure (Sigma s _ e) ctx) arg =
  eval (if s == Ident "" then ctx else extend ctx s (Val arg)) e
evalClosure v arg = error $ "[evalClosure] got non-closure " ++ show v
  ++ " applied to " ++ show arg

-- Handler of `App` (function application, i.e. Π-type eliminator)
doApply :: Value -> Value -> Value
-- Standard case: do β-reduction
doApply fun@(Closure Abst{} _) arg = evalClosure fun arg
-- Restricted abstraction case, which requires to apply the function
-- inside the restriction too
doApply (Restr sys fun@Closure{}) arg = foldRestr sys' (doApply fun arg)
  where sys' = mapSys (`doApply` arg) sys
-- Standard neutral case
doApply fun@(Neutral _ fty@Closure{}) arg =
  simplNeutralValue $ Neutral (App fun arg) (doApply fty arg)
-- Restricted neutral case
doApply fun@(Neutral _ (Restr sys cl@Closure{})) arg =
```

```
    simplNeutralValue $ Neutral (App fun arg) (foldRestr sys' (doApply cl arg))
  where sys' = mapSys (`doApply` arg) sys
-- System case
doApply (Sys sys) arg = Sys $ mapSys (`doApply` arg) sys
-- Partial type case
doApply fun@(Neutral _ (Partial phi cl@Closure{})) arg =
  Neutral (App fun arg) (foldPartial phi (doApply cl arg))
doApply v arg = error $ "[doApply] got " ++ show v ++ ", " ++ show arg

-- Handler of `Ind` (i.e. Nat eliminator)
doInd :: Value -> Value -> Value -> Value -> Value
doInd fam base step m = case m of
  -- Standard base and inductive cases: do β-redution
  Zero      -> base
  Succ n'  -> doApply fun prev
    where
      fun = doApply step n'
      prev = doInd fam base step n'
  -- Stardard neutral case; need to compute the type
  Neutral n Nat -> simplNeutralValue $ Neutral (Ind fam base step n)
    (doApply fam (Neutral n Nat))
  -- Restricted neutral case
  Neutral n (Restr sys Nat) ->
    simplNeutralValue $ Neutral (Ind fam base step n)
      (foldRestr sys' (doApply fam (Neutral n Nat)))
    where sys' = mapSys (doApply fam) sys
  -- System case
  Sys sys -> Sys $ mapSys (doInd fam base step) sys
  -- Partial type case
  Neutral n (Partial phi Nat) ->
    Neutral (Ind fam base step n) (foldPartial phi (doApply fam (Neutral n Nat)))
  otherwise -> error $ "[doInd] got " ++ show m

-- Handler of `Fst` (i.e. Σ-type first projection)
doFst :: Value -> Value
doFst v = case v of
  -- Standard case: do β-redution
  Pair v1 _ -> v1
  -- Stardard neutral case; need to compute the type
  Neutral _ (Closure (Sigma _ t _) ctx) -> simplNeutralValue $ Neutral (Fst v)
    (eval ctx t)
  -- Restricted neutral case
  Neutral x (Restr sys cl@(Closure (Sigma _ t _) ctx)) ->
    simplNeutralValue $ Neutral (Fst (Neutral x cl)) (foldRestr sys' (eval ctx t))
    where sys' = mapSys doFst sys
  -- System case
  Sys sys -> Sys $ mapSys doFst sys
  -- Partial type case
  Neutral x (Partial phi cl@(Closure (Sigma _ t _) ctx)) ->
```

```
      Neutral (Fst (Neutral x cl)) (foldPartial phi (eval ctx t))
    otherwise -> error $ "[doFst] got " ++ show v

-- Handler of `Snd` (i.e. Σ-type second projection)
doSnd :: Value -> Value
doSnd v = case v of
  -- Standard case: do β-redution
  Pair _ v2 -> v2
  -- Stardard neutral case; need to compute the type
  Neutral _ ty@(Closure Sigma{} _) -> simplNeutralValue $ Neutral (Snd v)
    (evalClosure ty (doFst v))
  -- Restricted neutral case
  Neutral x (Restr sys cl@(Closure Sigma{} _)) ->
    simplNeutralValue $ Neutral (Snd (Neutral x cl))
      (foldRestr sys' (evalClosure cl (doFst (Neutral x cl))))
    where sys' = mapSys doSnd sys
  -- System case
  Sys sys -> Sys $ mapSys doSnd sys
  -- Partial type case
  Neutral x (Partial phi cl@(Closure Sigma{} _)) ->
    Neutral (Snd (Neutral x cl))
      (foldPartial phi (evalClosure cl (doFst (Neutral x cl))))
  otherwise -> error $ "[doSnd] got " ++ show v

-- Handler of `Split` (i.e. (+)-type eliminator)
doSplit :: Value -> Value -> Value -> Value -> Value
doSplit fam f1 f2 x = case x of
  -- Standard cases (left/right injection): do β-redution
  InL x1 -> doApply f1 x1
  InR x2 -> doApply f2 x2
  -- Stardard neutral case; need to compute the type
  Neutral _ (Sum _ _) -> simplNeutralValue $
    Neutral (Split fam f1 f2 x) (doApply fam x)
  -- Restricted neutral case
  Neutral _ (Restr sys (Sum ty1 ty2)) -> simplNeutralValue $
    Neutral (Split fam f1 f2 (Neutral x (Sum ty1 ty2)))
      (foldRestr sys' (doApply fam x))
    where sys' = mapSys (doApply fam) sys
  -- System case
  Sys sys -> Sys $ mapSys (doSplit fam f1 f2) sys
  -- Partial type case
  Neutral _ (Partial phi (Sum ty1 ty2)) ->
    Neutral (Split fam f1 f2 (Neutral x (Sum ty1 ty2)))
      (foldPartial phi (doApply fam x))
  otherwise -> error $ "[doSplit] got " ++ show x

-- Utility function to handle eventually empty strings
ifEmpty :: Ident -> String -> Ident
ifEmpty (Ident "") s = Ident s
```

```
ifEmpty i            _ = i

-- Handler of composition (which is not an eliminator!)
doComp :: Ctx -> Term -> DisjFormula -> Term -> Term -> Term -> Term -> Value
doComp ctx fam phi i0 u b i =
  if isTrue phiV then  -- 1° trivial case: `phi` is True
    doApply uV iV
  else if conv (keys ctx) emptyDirEnv i0V iV then -- 2° trivial case: i = i_0
    bV
  else
    -- Fresh variables: `var` to pattern-match the type family
    --                  `var2` to handle the partially defined `u`
    let var  = newVar (keys ctx) (Ident "_i")
        var2 = case uV of
          Closure (Abst v _ _) _ -> newVar (keys ctx) (ifEmpty v "j")
        emptySys = Abst (Ident "") I (Sys [])
    -- Evaluate the type-family, pattern-matching inside the closure
    in case doApply famV (Neutral (Var var) I) of
      -- Π-type `[x:ty]e`, with `ty` a type
      cl@(Closure (Abst x ty _) ctx') | eval ctx' ty /= I ->
        Closure (Abst yi tyc comp) ctx
        where
          -- Variable of the target type `tyc`, i.e. `ty(i)`
          yi   = newVar (var : var2 : keys ctx) (ifEmpty x "u")
          -- Transport of that variable at `i0` and `var'`
          yt0  = Comp (Abst var I ty1) fFalse i emptySys (Var yi) i0
          yt   = Comp (Abst var I ty1) fFalse i emptySys (Var yi) (Var var)
          -- Resulting composition
          comp = Comp (Abst var I e') phi i0 u' (App b yt0) i
          ty1  = readBack (keys ctx) $ eval ctx' ty
          e'   = readBack (keys ctx) $ evalClosure cl (eval ctx yt)
          -- Target type: `ty(i)`
          tyc  = readBack (keys ctx) $ case doApply famV iV of
            Closure (Abst _ ty' _) ctx'' -> eval ctx'' ty'
          -- Apply functions to `yt` inside `u`
          u' = case isFalse phiV of
            True  -> u
            False -> Abst var2 I (App (App u (Var var2)) yt)

      -- Π-type `[x:I]e`
      cl@(Closure (Abst x ty _) ctx') | eval ctx' ty == I ->
        Closure (Abst x' I comp) ctx
        where
          -- Fresh nterval variable
          x'   = newVar (var : var2 : keys ctx) (ifEmpty x "i")
          -- Resulting composition
          comp = Comp (Abst var I e') phi i0 u' (App b (Var x')) i
          e'   = readBack (keys ctx) $ evalClosure cl (Neutral (Var x') I)
          -- Apply functions to `x'` inside `u`
```

```
    u'   = case isFalse phiV of
      True  -> u
      False -> Abst var2 I (App (App u (Var var2)) (Var x'))

-- Σ-type `<x:ty>e`
cl@(Closure (Sigma _ ty _) ctx') -> Pair c1 c2
  where
    -- Composition on each pair
    c1  = doComp ctx (Abst var I ty1) phi i0 u0 (Fst b) i
    c2  = doComp ctx (Abst var I ty2) phi i0 u1 (Snd b) i
    ty1 = readBack (keys ctx) $ eval ctx' ty
    -- The type family of `c2` needs the comp. on the first component
    c1' = doComp ctx (Abst var I ty1) phi i0 u0 (Fst b) (Var var)
    ty2 = readBack (keys ctx) $ evalClosure cl c1'
    -- Apply projections inside `u`
    (u0,u1) = case isFalse phiV of
      True  -> (u,u)
      False -> (Abst var2 I (Fst (App u (Var var2))),
                Abst var2 I (Snd (App u (Var var2))))

-- Coproduct `ty1 + ty2`
Sum ty1 ty2 ->
  -- If `b` is neutral, the result is neutral
  case bV of
    Neutral{} -> doNeutralComp
    otherwise -> inj comp
  where (inj,bV',ty) = case bV of
      -- Extract the injection, the inner value and the type
      InL b1 -> (InL,b1,ty1)
      InR b2 -> (InR,b2,ty2)
      -- Resulting composition
      comp = doComp ctx (Abst var I ty') phi i0 u' b' i
      b' = readBack (keys ctx) bV'
      ty'  = readBack (keys ctx) ty
      -- Remove the outer injections from `u`
      u' = case isFalse phiV of
        True  -> u
        False -> Abst var2 I (readBack (keys ctx) sys')
        where app = App u (Var var2)
              sys' = case eval (extend ctx var2 (Decl I)) app of
                Sys sysV -> Sys $ map (\case {(psi,InL q) -> (psi,q);
                  (psi,InR q) -> (psi,q)}) sysV
                InL q    -> q
                InR q    -> q

-- Naturals
Nat -> case bV of
  Zero    -> Zero
  Succ b' -> Succ $ doComp ctx fam phi i0 u' b' i
```

```
                        -- Remove the outer `S` from `u`
                  where u' = case isFalse phiV of
                      True  -> u
                      False -> Abst var2 I (readBack (keys ctx) sys')
                        where app  = App u (Var var2)
                              sys' = case eval (extend ctx var2 (Decl I)) app of
                                Sys sysV -> Sys $
                                  map (\case (psi,Succ m) -> (psi,m)) sysV
                                Succ m    -> m
              Neutral{} -> doNeutralComp

            -- Restriction type `[phi]ty`
          Restr sysR ty | var `notElem` concatMap vars (keys sys) ->
              doComp ctx (Abst var I ty') formula i0 u' b i
            where
              formula = Disj $ phi' ++ psis
              phi' = case phi of Disj df -> df
              psis = keys sysR
              ty'  = readBack (keys ctx) ty
              u'   = Abst var2 I (Sys sys')
              -- Concatenate the two systems
              sys' = map (\conj -> (conj,App u (Var var2))) phi'
                ++ case doApply famV (Neutral (Var var2) I) of
                  Restr sys'' _ ->
                    map (\(conj,v) -> (conj,readBack (keys ctx) v)) sys''

            -- Neutral type family; the result of the composition is neutral too
          otherwise -> doNeutralComp
      where
        -- Values computed for each argument (remember that Haskell is lazy)
        famV = eval ctx fam
        phiV = evalDisjFormula ctx phi
        i0V  = eval ctx i0
        uV   = eval ctx u
        bV   = eval ctx b
        iV   = eval ctx i
        -- Compute the type of the composition, and prepare the neutral value
        doNeutralComp = simplNeutralValue $
          Neutral (Comp famV phiV i0V uV bV iV) (eval ctx compTy)
        sys     = getCompSys phi i0 u b i
        compTy = Restr sys (App fam i)

-- Simplify a neutral value if the type is a restriction type with a true
-- formula, otherwise do nothing. It is similar to `simplRestr`, used only
-- for `conv`, except that here we do not use the directions environment.
-- `simplNeutralValue` is used only in evaluation.
simplNeutralValue :: Value -> Value
simplNeutralValue neu@(Neutral _ ty) = case ty of
  Restr sys _ | isSimplSys emptyDirEnv sys ->
```

```
      simplSys emptyDirEnv sys
    otherwise -> neu

-- Read-back function which converts values back into terms
-- The first argument is the list of already used names
-- The only non-trivial case is that of closures
readBack :: [Ident] -> Value -> Term
readBack used val = case val of
  App fun arg -> App (readBack used fun) (readBack used arg)
  Succ v -> Succ (readBack used v)
  Fst v -> Fst (readBack used v)
  Snd v -> Snd (readBack used v)
  Pair v1 v2 -> Pair (readBack used v1) (readBack used v2)
  Sum v1 v2 -> Sum (readBack used v1) (readBack used v2)
  InL v -> InL (readBack used v)
  InR v -> InR (readBack used v)
  Split ty f1 f2 x -> Split (readBack used ty) (readBack used f1)
    (readBack used f2) (readBack used x)
  Sys sys -> Sys $ mapSys (readBack used) sys
  Partial phi ty -> foldPartial phi (readBack used ty)
  Restr sys ty -> foldRestr (mapSys (readBack used) sys) (readBack used ty)
  Ind ty b e n -> Ind (readBack used ty) (readBack used b) (readBack used e)
    (readBack used n)
  Comp fam phi i0 u b i -> Comp (readBack used fam) phi (readBack used i0)
    (readBack used u) (readBack used b) (readBack used i)
  -- Closure case: first evaluate the body with a fresh name, then read-back
  cl@(Closure f ctx) -> let
    -- `constr` is `Abst` or `Sigma`
    (constr,s,t,e) = extract f
    s'      = newVar used s
    eVal    = evalClosure cl (Neutral (Var s') (eval ctx t))
    e'      = readBack (s' : used) eVal
    t'      = readBack used (eval ctx t)
    in constr s' t' e'
  -- For neutrals, read-back the value, ignoring the type
  Neutral v _ -> readBack used v
  otherwise -> val

-- Normalization means first evaluating and then reading-back
normalize :: Ctx -> Term -> Term
normalize ctx e = readBack (keys ctx) (eval ctx e)

{- Printing utilities (should be in AbsCTT but these need 'readBack') -}

-- Print function for a term or value (which is read-back into
-- a term, except when debugging)
instance Show Term where
  show t = printTerm' 0 (if debug then t else readBack [] t)
```

```
-- Helper function; the first argument `i` measures the depth
-- of the term (but is reset in some cases), which is used to
-- avoid unnecessary parentheses
printTerm' :: Int -> Term -> String
printTerm' i = \case
  Var s        -> show s
  Universe     -> "U"
  TDef (s,t,e) t' ->
    "[" ++ show s ++ ":" ++ printTerm' 0 t ++ " = "
      ++ printTerm' 0 e ++ "]" ++ printTerm' 0 t'
  Abst s t e   -> par1 ++ abstS ++ par2
    where abstS = if not (containsVar s e)
        then -- A -> B (no dependency)
          printTerm' (i+1) t ++ " -> " ++ printTerm' 0 e
        else
          "[" ++ show s ++ ":" ++ printTerm' 0 t ++ "]" ++ printTerm' 0 e
  Sigma s t e  -> par1 ++ abstS ++ par2
    where abstS = if not (containsVar s e)
        then -- A * B (no dependency)
          printTerm' (i+1) t ++ " * " ++ printTerm' 0 e
        else
          "<" ++ show s ++ ":" ++ printTerm' 0 t ++ ">"
            ++ printTerm' 0 e
  Pair t1 t2 -> par1 ++ printTerm' i t1 ++ "," ++ printTerm' i t2 ++ par2
  Fst t -> par1 ++ printTerm' (i + 1) t ++ ".1" ++ par2
  Snd t -> par1 ++ printTerm' (i + 1) t ++ ".2" ++ par2
  Sum ty1 ty2          -> par1 ++ printTerm' (i + 1) ty1 ++ " + "
    ++ printTerm' (i + 1) ty2 ++ par2
  InL t1               -> par1 ++ "inl " ++ printTerm' (i + 1) t1 ++ par2
  InR t2               -> par1 ++ "inr " ++ printTerm' (i + 1) t2 ++ par2
  Split ty f1 f2 x     -> par1 ++ "split " ++ printTerm' (i+1) ty ++ " "
    ++ printTerm' (i+1) f1 ++ " " ++ printTerm' (i+1) f2  ++ " "
      ++ printTerm' (i+1) x ++ par2
  App fun arg -> par1 ++ printTerm' (i+1) inner
    ++ " " ++ unwords printedArgs ++ par2
    where (inner,args) = collectApps (App fun arg) []
        printedArgs  = map (printTerm' (i+1)) args
  Nat             -> "N"
  Zero            -> "Z"
  Succ t          -> par1 ++ "S " ++ printTerm' (i+1) t ++ par2
  Ind ty b s n    -> par1 ++ "ind-N " ++ printTerm' (i+1) ty ++ " "
    ++ printTerm' (i+1) b ++ " " ++ printTerm' (i+1) s ++ " "
      ++ printTerm' (i+1) n ++ par2
  I               -> "I"
  I0              -> "0"
  I1              -> "1"
  Sys sys         -> showSystem sys
  Partial phi t   -> "[" ++ show phi ++ "]" ++ printTerm' (i+1) t
  Restr sys t     -> showSystem sys ++ printTerm' (i+1) t
```

```haskell
  Comp fam phi i0 u b i' -> par1 ++ "comp " ++ printTerm' (i+1) fam
    ++ " (" ++ show phi ++ ") " ++ printTerm' (i+1) i0 ++ " "
      ++ printTerm' (i+1) u ++  " " ++ printTerm' (i+1) b
        ++  " " ++ printTerm' (i+1) i' ++ par2
  -------- Used only when debugging, to print proper values
  Closure cl _ -> "Cl(" ++ show cl ++  ")"
  Neutral v t  -> "N{" ++ printTerm' i v ++ "}:" ++ printTerm' (i+1) t
  -- Parentheses are not needed if `i` is zero
  where (par1,par2) = if i == 0 then ("","") else ("(",")")

-- Print a context (in one line)
showCtx :: Ctx -> String
showCtx ctx = "[" ++ intercalate ", " (map showEntry (reverse ctx)) ++ "]"

-- Print a single context entry
showEntry :: (Ident,CtxEntry) -> String
showEntry (s,Decl ty) = show s ++ " : " ++ show ty
showEntry (s,Def ty val) = show s ++ " : " ++ show ty ++ " = " ++ show val
showEntry (s,Val val) = show s ++ " => " ++ show val

-- Print a system
showSystem :: System -> String
showSystem sys = "[" ++ sysS ++ "]"
  where sysS = intercalate " | " $
      map (\(ff,t) -> show ff ++ " -> " ++ show t) sys

-- Get the system of the restriction type of a composition
getCompSys :: DisjFormula -> Term -> Term -> Term -> Term -> System
getCompSys (Disj df) i0 u b i = eq ++ map (\conj -> (conj,App u i)) df
  where
    -- Extract the variables from eventual values
    i'  = case i  of { Neutral (Var x) _ -> Var x ; _ -> i  }
    i0' = case i0 of { Neutral (Var x) _ -> Var x ; _ -> i0 }
    -- Translate `i = i0` into a real formula
    eq  = case (i0',i') of
      (I0,I0)        -> [(Conj [],b)]
      (I0,I1)        -> []
      (I0,Var s)     -> [(Conj [Eq0 s],b)]
      (Var s,I0)     -> [(Conj [Eq0 s],b)]
      (Var s,I1)     -> [(Conj [Eq1 s],b)]
      (Var s,Var s') -> [(Conj [Diag s s'],b)]
      (I1,I0)        -> []
      (I1,I1)        -> [(Conj [],b)]
      (I1,Var s)     -> [(Conj [Eq1 s],b)]
      _              -> error $ "[getCompSys] got " ++ show (i,i0)

-- Generic class for objects that allow a notion of α-conversion
class Convertible a where
  conv :: [Ident] -> DirEnv -> a -> a -> Bool
```

```haskell
-- Check convertibility under a conjunctive formula
-- If the formula is false, the values are trivally convertible
convPartialConj :: [Ident] -> ConjFormula -> DirEnv -> Value -> Value -> Bool
convPartialConj used conj dirs v1 v2 =
  let dirs' = addConj dirs conj
  in inconsistent dirs' || conv used dirs' v1 v2

-- Two values are convertible under a disjunctive formula iff
-- they are so under each conjunction
convPartialDisj :: [Ident] -> DisjFormula -> DirEnv -> Value -> Value -> Bool
convPartialDisj used (Disj df) dirs v1 v2 =
  all (\conj -> convPartialConj used conj dirs v1 v2) df

-- Check if two Π/Σ-abstractions are of the same kind
sameKind :: Term -> Term -> Bool
sameKind Abst{}  Abst{}  = True
sameKind Sigma{} Sigma{} = True
sameKind _       _       = False

-- Check if a restriction type contains a true formula
-- under the directions environment `dirs`
isSimplRestr :: DirEnv -> Value -> Bool
isSimplRestr dirs ty = case ty of
  Restr sys _ -> isSimplSys dirs sys
  otherwise   -> False

-- Simplify a restriction type if it contains a true formula
-- under the directions environment `dirs`
simplRestr :: DirEnv -> Value -> Value
simplRestr dirs (Restr sys _) = simplSys dirs sys

-- Check if a system contains a true formula
-- under the directions environment `dirs`
isSimplSys :: DirEnv -> System -> Bool
isSimplSys dirs = any ((dirs `makesTrueConj`) . fst)

-- Simplify a system if it contains a true formula
-- under the directions environment `dirs`
simplSys :: DirEnv -> System -> Value
simplSys dirs sys = snd . fromJust $
  find ((dirs `makesTrueConj`) . fst) sys

-- Simplify a system if possibile, otherwise do nothing
simpl :: DirEnv -> Value -> Value
simpl dirs (Sys sys) | isSimplSys dirs sys = simplSys dirs sys
simpl dirs v = v

-- Check if the type can be simplified; in that case two values
```

```haskell
-- of that type are automatically convertible, because they
-- shall reduce to the same value. That is, we don't need
-- to look inside the terms, i.e. we can ignore the proof
proofIrrelevant :: DirEnv -> Value -> Bool
proofIrrelevant dirs ty = case ty of
  Restr sys ty' -> isSimplRestr dirs ty
    || proofIrrelevant dirs ty'
  Closure cl ctx -> let
    -- Fresh variable to evaluate closures
    varV :: Ident -> Value -> CtxEntry
    varV s t = Val $ Neutral (Var $ newVar (keys ctx) s) (eval ctx t)
    in case cl of
      Abst s t e  -> proofIrrelevant dirs
        (eval (extend ctx s (varV s t)) e)
      Sigma s t e -> proofIrrelevant dirs (eval ctx t) &&
        proofIrrelevant dirs (eval (extend ctx s (varV s t)) e)
  otherwise -> False

-- αη convertibility for values, which are supposed to have the
-- same type. For efficiency, we first test exact syntactical equality
instance Convertible Value where
  conv used dirs v1 v2 =
    v1 == v2 || let cnv = conv used dirs in case (v1,v2) of
      (Universe,Universe) -> True
      -- Π/Σ closures
      (Closure cl1 ctx1,Closure cl2 ctx2) | sameKind cl1 cl2 -> let
        (_,s1,t1,_) = extract cl1
        (_,_ ,t2,_) = extract cl2
        var = newVar used s1
        t1V = eval ctx1 t1
        t2V = eval ctx2 t2
        e1' = evalClosure v1 (Neutral (Var var) t1V)
        e2' = evalClosure v2 (Neutral (Var var) t2V)
        in cnv t1V t2V && conv (var : used) dirs e1' e2'
      -- η-rule for Π (first case)
      (Closure (Abst s1 t1 _) ctx1,Neutral _ (Closure Abst{} _)) -> let
        var = newVar used s1
        t1V = eval ctx1 t1
        e1' = evalClosure v1 (Neutral (Var var) t1V)
        e2' = doApply (simpl dirs v2) (Neutral (Var var) t1V)
        in conv (var : used) dirs e1' e2'
      -- η-rule for Π (second case)
      (Neutral _ (Closure Abst{} _),Closure (Abst s2 t2 _) ctx2) -> let
        var = newVar used s2
        t2V = eval ctx2 t2
        e2' = evalClosure v2 (Neutral (Var var) t2V)
        e1' = doApply (simpl dirs v1) (Neutral (Var var) t2V)
        in conv (var : used) dirs e1' e2'
      {- Sigma types -}
```

```
(Fst v,Fst v') -> cnv v v'
(Snd v,Snd v') -> cnv v v'
(Pair vp1 vp1',Pair vp2 vp2') -> cnv vp1 vp2 &&
  cnv vp1' vp2'
-- η-rule for Σ (first case)
(vp,Pair v v') -> cnv (doFst $ simpl dirs vp) v &&
  cnv (doSnd $ simpl dirs vp) v'
-- η-rule for Σ (second case)
(Pair v v',vp) -> cnv v (doFst $ simpl dirs vp) &&
  cnv v' (doSnd $ simpl dirs vp)
{- Coproduct types -}
(Sum ty1 ty2,Sum ty1' ty2') -> cnv ty1 ty1' && cnv ty2 ty2'
(InL v,InL v') -> cnv v v'
(InR v,InR v') -> cnv v v'
{- Naturals -}
(Nat,Nat)            -> True
(Zero,Zero)          -> True
(Succ n1,Succ n2)    -> cnv n1 n2
{- Cubical -}
(I,I)                -> True
(I0,I0)              -> True
(I1,I1)              -> True
-- Systems. We have to check is the system is simplifiable
-- to avoid an infinite loop
(Sys sys,_) | isSimplSys dirs sys ->
  cnv (simpl dirs v1) v2
(_,Sys sys) | isSimplSys dirs sys ->
  cnv v1 (simpl dirs v2)
(Sys sys,Sys sys') -> conv used dirs sys sys'
(Partial phi v,Partial phi' v') -> eqFormulas dirs phi phi' &&
  cnv v v'
(Restr sys t,Restr sys' t') -> conv used dirs sys sys' && cnv t t'
{- Neutrals -}
(Var s1,Var s2) -> s1 == s2
(App f1 a1,App f2 a2) -> cnv f1 f2 && cnv a1 a2
(Ind ty1 b1 s1 n1,Ind ty2 b2 s2 n2) ->
  cnv ty1 ty2 && cnv b1 b2 &&
  cnv s1  s2  && cnv n1 n2
(Split ty1 f1 g1 x1,Split ty2 f2 g2 x2) -> cnv ty1 ty2
  && cnv f1 f2 && cnv g1 g2 && cnv x1 x2
(Comp fam1 phi1 i01 u1 b1 i1,Comp fam2 phi2 i02 u2 b2 i2) ->
  cnv fam1 fam2 && eqFormulas dirs phi1 phi2 &&
  cnv i01 i02 && cnv u1 u2 && cnv b1 b2 && cnv i1 i2
-- Neutral values with a simplifiable restriction type
(Neutral _ ty1,_) | isSimplRestr dirs ty1 ->
    cnv (simplRestr dirs ty1) v2
(_,Neutral _ ty2) | isSimplRestr dirs ty2 ->
    cnv v1 (simplRestr dirs ty2)
-- Interval names
```

```
        (Neutral (Var x1) I,Neutral (Var x2) I) ->
          dirs `makesTrueAtomic` Diag x1 x2
        (Neutral (Var x1) I,I0) ->
          dirs `makesTrueAtomic` Eq0 x1
        (Neutral (Var x1) I,I1) ->
          dirs `makesTrueAtomic` Eq1 x1
        (I0,Neutral (Var x2) I) ->
          dirs `makesTrueAtomic` Eq0 x2
        (I1,Neutral (Var x2) I) ->
          dirs `makesTrueAtomic` Eq1 x2
        -- Last case, two neutral values
        -- The types must be convertible, due to type-checking
        (Neutral v ty,Neutral v' _) ->
          proofIrrelevant dirs ty || cnv v v'
        otherwise -> False

-- Convertibility between two systems
instance Convertible System where
  conv used dirs sys1 sys2 =
    eqFormulas dirs (getSystemFormula sys1) (getSystemFormula sys2)
    && all (\(conj,t1,t2) -> convPartialConj used conj dirs t1 t2) meets
    where meets = [(conj1 `meet` conj2, sys1 `at` conj1, sys2 `at` conj2) |
                     conj1 <- keys sys1, conj2 <- keys sys2]
```

3.5   Type-checking

In this file we implement the bidirectional type checker, made up of the two
functions `inferType` and `checkType`; the former gets a term as input and
returns the evaluated type as output, whereas the latter gets as input a term
and the evaluated type, and returns a boolean. To be more precise, the two
functions use the `Either` monad to handle errors, using `Left` *err* to report an
error and quit, and `Right` *v* to successfully return a value of the specified type.

Source file `TypeChecker.hs`.

```
module TypeChecker where

import Control.Monad

import Ident
import Interval
import CoreCTT
import Eval

-- Infer the type of a term, in the given context and directions environment
inferType :: Ctx -> DirEnv -> Term -> Either ErrorString Value
inferType ctx dirs term = case term of
```

```haskell
-- Variables: look up the type in the context
Var s -> Right $ lookupType s ctx
-- Universe
Universe -> Right Universe
-- Function application: the type of the function is inferred
App fun arg -> do
  funTy <- inferType ctx dirs fun
  -- Handle restriction and partial types: `box` is used
  -- to put the resulting type in the eventual restriction
  -- or partial type
  let (funTy',box) = case funTy of
        Restr sys v   -> (v,makeRestr sys)
        Partial phi v -> (v,makePartial phi)
        otherwise     -> (funTy,curry snd)
      makeRestr :: System -> Value -> Value -> Value
      makeRestr sys val = foldRestr $ mapSys (`doApply` val) sys
      makePartial :: DisjFormula -> Value -> Value -> Value
      makePartial phi _ = foldPartial phi
  -- The type must be a Π-type
  case funTy' of
    c@(Closure (Abst _ t _) ctx1) -> do
      checkType ctx dirs arg (eval ctx1 t)
      let argVal = eval ctx arg
      return $ box argVal (doApply c argVal)
    otherwise -> Left $
      "term '" ++ show fun ++ "' has type '" ++ show funTy
        ++"' , which is not a product"

-- First projection: the type of the argument is inferred
Fst p -> do
  ty <- inferType ctx dirs p
  -- Handle restriction and partial types
  let (ty',box) = case ty of
        Restr sys t   -> (t,makeRestr sys)
        Partial phi v -> (v,foldPartial phi)
        otherwise     -> (ty,id)
      makeRestr :: System -> Value -> Value
      makeRestr = foldRestr . mapSys doFst
  -- The type must be a Σ-type
  case ty' of
    Closure (Sigma _ t _) ctx1 -> do
      return $ box (eval ctx1 t)
    otherwise -> Left $
      "term '" ++ show term ++ "' has type '" ++ show ty
        ++ "' , which is not a sum"

-- Second projection: the type of the argument is inferred
Snd p -> do
  ty <- inferType ctx dirs p
```

```
   --Handle restriction and partial types
   let (ty',box) = case ty of
         Restr sys t    -> (t,makeRestr sys)
         Partial phi v -> (v,foldPartial phi)
         otherwise      -> (ty,id)
       makeRestr :: System -> Value -> Value
       makeRestr = foldRestr . mapSys doSnd
   -- The type must be a Σ-type
   case ty' of
     c@(Closure Sigma{} ctx1) -> do
       return . box $ evalClosure c (doFst $ eval ctx p)
     otherwise -> Left $
       "term '" ++ show term ++ "' has type '" ++ show ty
         ++ "' , which is not a sum"

-- Coproduct eliminator: the type of the argument is inferred
Split fam f1 f2 x -> do
  ty <- inferType ctx dirs x

   --Handle restriction and partial types
   let (ty',box) = case ty of
         Restr sys t    -> (t,makeRestr sys)
         Partial phi v -> (v,foldPartial phi)
         otherwise      -> (ty,id)
       famV = eval ctx fam
       makeRestr :: System -> Value -> Value
       makeRestr = foldRestr . mapSys (famV `doApply`)
   -- The type must be a coproduct
   case ty' of
     Sum{} -> do
       let sty@(Sum sty1 sty2) = readBack (keys ctx) ty'
           var = newVar (keys ctx) (Ident "a")
       checkType ctx dirs fam
         (Closure (Abst (Ident "_") sty Universe) ctx)
       checkType ctx dirs f1
         (eval ctx $ Abst var sty1 (App fam (InL (Var var))))
       checkType ctx dirs f2
         (eval ctx $ Abst var sty2 (App fam (InR (Var var))))
       return . box $ eval ctx (App fam x)
     otherwise -> Left $ "expected a sum type, got term '" ++ show x
         ++ "' of type '" ++ show ty ++ "' instead"

-- Naturals
Nat -> Right Universe
Zero -> Right Nat
Succ n -> do
  checkType ctx dirs n Nat
  Right Nat
-- Induction for naturals
```

```
Ind fam base step n -> do
  -- Check that `n` is a natural (eventually partial/restricted)
  nTyVal <- inferType ctx dirs n
  isNat n nTyVal
  -- Check that `fam` has type N -> U
  checkType ctx dirs fam (makeFunTypeVal Nat Universe)

  -- Handle restriction and partial types
  let box = case nTyVal of
          Restr sys Nat -> makeRestr sys
          Partial phi _ -> foldPartial phi
          Nat           -> id
      famV = eval ctx fam
      makeRestr :: System -> Value -> Value
      makeRestr = foldRestr . mapSys (famV `doApply`)

  -- Evaluate the type-family `fam`, checking that `base` has
  -- type `fam Z`
  let tyVal  = eval ctx fam
      tyVal0 = doApply tyVal Zero
  checkType ctx dirs base tyVal0

  -- Checking that the "inductive step" `step` has type
  -- [n : nat] fam n -> fam (suc n)
  let varname = newVar (keys ctx) (Ident "n")
      var     = Var varname
      ctx'    = extend ctx varname (Decl Nat)
  checkType ctx dirs step (eval ctx'
    (Abst varname Nat
      (Abst (Ident "") (App fam var)
        (App fam (Succ var)))
    )) -- [n : nat] fam n -> fam (suc n)

  return . box $ doApply tyVal (eval ctx n)

-- Interval endpoints
I0 -> Right I
I1 -> Right I

-- Composition
Comp fam phi@(Disj df) i0 u b i -> do
  -- Checking the type-family `fam`, point `i_0` and formula `phi`
  checkType ctx dirs fam (makeFunTypeVal I Universe) -- I -> U
  checkType ctx dirs i0 I
  checkDisjFormula ctx phi
  -- Checking that `u` has the correct type
  let var = newVar (keys ctx) (Ident "_i")
  checkType ctx dirs u (eval (extend ctx var (Decl I))
    (Abst var I (Partial phi (App fam (Var var)))))
```

```
    -- Checking that `b` has type `[phi -> u i0](fam i0)`
    checkType ctx dirs b $
      eval ctx (Restr (map (\psi -> (psi,App u i0)) df) (App fam i0))
    -- Return the evaluated type, without the restriction if it's empty
    let sys = getCompSys phi i0 u b i
    return $ eval ctx $ (if null sys then id else Restr sys) (App fam i)

  -- Failed type inference
  _  -> Left $ "don't know how to infer type of '" ++ show term ++ "'"

-- Check if a term has Nat type
isNat :: Term -> Value -> Either ErrorString ()
isNat _ Nat            = Right ()
isNat _ (Restr _ Nat)  = Right ()
isNat _ (Partial _ Nat) = Right ()
isNat t v              = Left $ "expected type nat, got term '" ++ show t ++
  "' of type '" ++ show v ++ "' instead"

-- Utility function to get type values of the form A -> B
makeFunTypeVal :: Term -> Term -> Value
makeFunTypeVal ty e = eval emptyCtx (Abst (Ident "") ty e)

-- Check the type of a term under a conjunction
-- If the conjunction is false, type-check is trivially true
checkTypePartialConj :: ConjFormula -> Ctx -> DirEnv -> Term
  -> Value -> Either ErrorString ()
checkTypePartialConj conj ctx dirs e v = do
  let dirs' = addConj dirs conj
  unless (inconsistent dirs') $
    checkType ctx dirs' e v

-- Check the type of a term under a disjunction, that is
-- under each conjunction
checkTypePartialDisj :: DisjFormula -> Ctx -> DirEnv -> Term
  -> Value -> Either ErrorString ()
checkTypePartialDisj (Disj df) ctx dirs e v =
  mapM_ (\conj -> checkTypePartialConj conj ctx dirs e v) df

-- Check the type of a term agains a given type
-- The type must be a value (i.e. in β-normal form)
checkType :: Ctx -> DirEnv -> Term -> Value -> Either ErrorString ()
checkType ctx dirs term v = case (term,v) of
  -- Let-definition
  (TDef (s,t,e) t',_) -> do
    checkType ctx dirs t Universe
    checkType (extend ctx s (Decl t)) dirs e (eval ctx t)
    checkType (extend ctx s (Def t e)) dirs t' v

  -- Π-type former
```

```
(Abst s t e,Universe) -> do
  -- Handle also path types
  unless (t == I) $ checkType ctx dirs t Universe
  checkType (extend ctx s (Decl t)) dirs e Universe

-- Σ-type former
(Sigma s t e,Universe) -> do
  checkType ctx dirs t Universe
  checkType (extend ctx s (Decl t)) dirs e Universe

-- λ- or Π- abstraction
(Abst s t e,Closure (Abst _ t1 _) ctx1) -> do
  -- Handle also I-abstractions
  unless (t == I) $ checkType ctx dirs t Universe
  let tVal  = eval ctx t
      t1Val = eval ctx1 t1
  unless (conv (keys ctx) dirs tVal t1Val) $
    Left $ "type '" ++ show tVal ++ "' is not convertible to type '"
      ++ show t1Val ++ "' (while checking term '" ++ show (Abst s t e)
      ++ "' against type '" ++ show v ++ "')"
  -- Introduce a fresh variable and check the body
  let var   = newVar (keys ctx1) s
      e1Val = doApply v (Neutral (Var var) t1Val)
      ctx'  = if s == var then
        extend ctx s (Decl t)
      else
        extend (extend ctx s (Decl t)) s (Val (Neutral (Var var) tVal))
  checkType ctx' dirs e e1Val

-- Σ-type constructor (pair)
(Pair p1 p2,Closure (Sigma _ t1 _) ctx1) -> do
  let t1Val = eval ctx1 t1
      e1Val = evalClosure v (eval ctx p1)
  -- Check each component
  checkType ctx dirs p1 t1Val
  checkType ctx dirs p2 e1Val

-- Coproduct type former
(Sum ty1 ty2,Universe) -> do
  checkType ctx dirs ty1 Universe
  checkType ctx dirs ty2 Universe

-- Σ-type left injection
(InL t1,Sum ty1 _) -> do
  checkType ctx dirs t1 ty1

-- Σ-type right injection
(InR t2,Sum _ ty2) -> do
  checkType ctx dirs t2 ty2
```

```
-- Restriction type
(e,Restr sys ty) -> do
  let eVal = eval ctx e
      phi  = getSystemFormula sys
  checkType ctx dirs e ty
  -- Check for conversion under the formulas
  unless (convPartialDisj (keys ctx) phi dirs eVal (Sys sys)) $
    Left $ "term '" ++ show e ++ "' does not agree with '" ++
      show (Sys sys) ++ "' on " ++ show phi

-- System
(Sys sys,Partial phi ty) -> do
  -- Check that the formulas match
  let psis = keys sys
  mapM_ (checkConjFormula ctx) psis
  unless (eqFormulas dirs phi (Disj psis)) $
    Left $ show phi ++ " is not logically equivalent to "
      ++ show (Disj psis)
  -- Check conversion the partial elements in the system
  mapM_ (\(psi,t) -> checkTypePartialConj psi ctx dirs t ty) sys
  -- Check conversion at the intersections
  let eq_check = all (\((psi1,t1),(psi2,t2)) ->
      convPartialConj (keys ctx) (psi1 `meet` psi2) dirs
        (eval ctx t1) (eval ctx t2)
        ) [(x1,x2) | x1 <- sys, x2 <- sys, x1 /= x2]
  unless eq_check $
    Left "values are not adjacent"

-- Partial type former
(Partial phi ty,Universe) -> do
  checkDisjFormula ctx phi
  checkType ctx dirs ty Universe

-- Restriction type former
(Restr sys ty,Universe) -> do
  checkType ctx dirs ty Universe
  let tyVal = eval ctx ty
      phi   = getSystemFormula sys
  checkDisjFormula ctx phi
  -- Check the elements in the system
  mapM_ (\(conj,t) -> checkTypePartialConj conj ctx dirs t tyVal) sys

-- If no other rule match, try inferring the type and
-- check if it's compatible
otherwise -> do
  ty <- inferType ctx dirs term
  -- Check for sub-typing: `v` more general than `ty`
  unless (compTypes (keys ctx) dirs ty v) $
```

```
        Left $ "type '" ++ show v ++ "' expected, got term '" ++ show term
          ++ "' of type '" ++ show ty ++ "' instead"

-- Check compatibility (subtyping) between two partial or restriction types,
-- as specified by the typing rules: `v1` more general than `v2`
compTypes :: [Ident] -> DirEnv -> Value -> Value -> Bool
compTypes used dirs v1 v2 =
  -- Split the type and get eventual partial type formulas
  -- (by default `True` in the other cases)
  let (iphi,ity) = split v1
      (vphi,vty) = split v2
      syscheck   = case (v1,v2) of
        (Restr isys _,Restr vsys _) -> convPartialDisj used
          (getSystemFormula vsys) dirs (Sys isys) (Sys vsys)
        otherwise -> True
    -- Check sub-typing compatibility for restrictions, for the base
    -- types and for formulas in the case of partial types
  in syscheck && conv used dirs ity vty && impDisj dirs vphi iphi

-- Check that the variables of the conjunctive formula are in the context
checkConjFormula :: Ctx -> ConjFormula -> Either ErrorString ()
checkConjFormula ctx cf = do
  let dom     = keys ctx
      support = vars cf
  unless (all (`elem` dom) support) $
    Left $ "formula '" ++ show cf ++ "' contains undeclared names"

-- Check that the variables of the disjunctive formula are in the context
checkDisjFormula :: Ctx -> DisjFormula -> Either ErrorString ()
checkDisjFormula ctx (Disj df) = mapM_ (checkConjFormula ctx) df
```

3.6   Main program

This last file is the one that handles input-output, using all the previous files to produce a full working interpreter of the type theory. We use the `State` monad to store some information across the iterations of the REPL loop, that is the current context, the last checked term and the list of locked names.[6] For each declaration or definition given in input (typed from the user or from a file), first the parser is called to produce the abstract syntax, then `checkSingleToplevel` checks name shadowing and lastly `checkSingleToplevel'` calls the type-checker to ascertain that the input is correct, finally adding the declaration or definition to the context. When an example is given, that is simply a term, after the syntactical checks the type-checker is called to infer its type.

---

[6]If a name is locked, then its definition in the context is temporarily erased, keeping only the type declaration. By unlocking the name, the definition is added again.

The program supports some commands, starting with ':', such as .

- :help (print help)

- :q (quit the program)

- :ans (print the last term checked)

- :ctx (print the current context)

- :clear $id_1 \ldots id_n$ (remove the given identifiers from the context, recursively)

- :lock $id_1 \ldots id_n$ (lock the given names)

- :unlock $id_1 \ldots id_n$ (unlock the given names)

- :unlockall (unlock every currently locked name)

- :printlock (show all locked names)

Source file `MainCTT.hs`.

```haskell
module Main where

import System.IO ( hFlush, stdout )
import System.Environment ( getArgs )
import System.Exit        ( exitSuccess )
import Control.Monad.State
import Data.List ( intercalate )
import Data.Maybe ( isJust )

import ParCTT   ( pTerm, pToplevel, pProgram, myLexer )

import Ident
import Interval
import CoreCTT
import Eval
import TypeChecker

type Err = Either String

-- Current context, last term checked, list of locked names
type ReplState = (Ctx,Term,[Ident])

-- Initial state
initReplState :: ReplState
initReplState = ([],Zero,[])

-- Read from a file and call `run`
runFile :: FilePath -> StateT ReplState IO Bool
```

```haskell
runFile f = do
  printLnIO $ "Reading file " ++ f
  contents <- liftIO . readFile $ f
  res <- run contents
  liftIO . when res . putStrLn $ "\nFile " ++ f ++ " loaded successfully"
  return res

-- Parse and call `checkProgram`
run :: String -> StateT ReplState IO Bool
run s = case pProgram ts of
  Left err -> do
    liftIO $ putStrLn "\nParse failed!"
    liftIO $ showErr err
    return False
  Right program -> do
    checkProgram program
  where
    ts = myLexer s

-- Check a whole program, by type-checking every top-level
-- declaration
checkProgram :: Program -> StateT ReplState IO Bool
checkProgram (Program []) = return True
checkProgram (Program (toplevel : decls)) = do
  res <- checkSingleToplevel toplevel
  if res then
    checkProgram (Program decls)
  else
    return False

-- Check if a term contains undeclared variables (True = OK)
checkVars :: Ctx -> Term -> Bool
checkVars ctx term = case term of
  Var s               -> isJust $ lookup s ctx
  Universe            -> True
  Abst s t e          -> checkVars ctx t &&
    checkVars (extend ctx s (Decl {-dummy-}Universe)) e
  TDef (s,t,e) t'     -> checkVars ctx t &&
    checkVars (extend ctx s (Decl {-dummy-}Universe)) e &&
    checkVars (extend ctx s (Def t e)) t'
  App fun arg         -> checkVars ctx fun && checkVars ctx arg
  Sigma s t e         -> checkVars ctx t &&
    checkVars (extend ctx s (Decl {-dummy-}Universe)) e
  Pair t1 t2          -> checkVars ctx t1 && checkVars ctx t2
  Fst t               -> checkVars ctx t
  Snd t               -> checkVars ctx t
  Sum ty1 ty2         -> checkVars ctx ty1 && checkVars ctx ty2
  InL t1              -> checkVars ctx t1
  InR t2              -> checkVars ctx t2
```

```
  Split ty f1 f2 x    -> checkVars ctx ty && checkVars ctx f1 &&
    checkVars ctx f2 && checkVars ctx x
  Nat                 -> True
  Zero                -> True
  Succ t              -> checkVars ctx t
  Ind ty b s n        -> checkVars ctx ty && checkVars ctx b &&
    checkVars ctx s && checkVars ctx n
  I                   -> True
  I0                  -> True
  I1                  -> True
  Sys sys             -> all (\phi -> all (`elem` keys ctx) (vars phi))
    (keys sys) && all (checkVars ctx) (elems sys)
  Partial phi t       -> all (`elem` keys ctx) (vars phi) &&
    checkVars ctx t
  Restr sys t         -> checkVars ctx (Sys sys) && checkVars ctx t
  Comp fam phi i0 u b i -> checkVars ctx fam &&
    all (`elem` keys ctx) (vars phi) && checkVars ctx i0 &&
    checkVars ctx u && checkVars ctx b && checkVars ctx i

-- Check a single top-level declaration, calling `checkSingleToplevel'`
-- Here we mostly check variables
checkSingleToplevel :: Toplevel -> StateT ReplState IO Bool
-- Example: infer its type
checkSingleToplevel (Example t) = do
  (ctx,_,_) <- get
  if not (checkTermShadowing (keys ctx) t) then do
    liftIO . showErr $ "term '" ++ show t ++ "' contains shadowed variables"
    return False
  else if not (checkVars ctx t) then do
    liftIO . showErr $ "term '" ++ show t ++ "' contains undeclared variables"
    return False
  else
    checkSingleToplevel' (Example t)
-- Add a declaration to the context
checkSingleToplevel decl@(Declaration s t) = do
  (ctx,_,_) <- get
  if not (checkTermShadowing (keys ctx) t) then do
    liftIO . showErr $ "term '" ++ show t ++ "' contains shadowed variables"
    return False
  else if not (checkVars ctx t) then do
    liftIO . showErr $ "term '" ++ show t ++ "' contains undeclared variables"
    return False
  else case lookup s ctx of
    Nothing -> checkSingleToplevel' decl
    Just _  -> do
      liftIO . showErr $ "context already contains name '" ++ show s ++ "'"
      return False
-- Add a definition to the context
checkSingleToplevel def@(Definition s t e) = do
```

```haskell
  (ctx,_,_) <- get
  if not (checkTermShadowing (s : keys ctx) t &&
     checkTermShadowing (s : keys ctx) e) then do
    liftIO . showErr $ "definition of '" ++ show s
      ++ "' contains shadowed variables"
    return False
  else if not (checkVars ctx t && checkVars ctx e) then do
    liftIO . showErr $ "definition of '" ++ show s
      ++ "' contains undeclared variables"
    return False
  else case lookup s ctx of
    Nothing -> checkSingleToplevel' def
    Just _  -> do
      liftIO . showErr $ "context already contains name '"
        ++ show s ++ "'"
      return False

checkSingleToplevel' :: Toplevel -> StateT ReplState IO Bool
checkSingleToplevel' (Example t) = do
  -- Get the context with the locked names (i.e. erasing definitions
  -- of locked names)
  (unlockedCtx,_,lockedNames) <- get
  let ctx = getLockedCtx lockedNames unlockedCtx
      ty  = inferType ctx emptyDirEnv t
  case ty of
    Left err -> do
       liftIO $ showErr err
       return False
    Right tyVal -> do
      printLnIO $ "\n'" ++ show t ++ "' has (inferred) type '"
        ++ show (readBack (keys ctx) tyVal) ++ "'"
      -- Since `t` typechecks, `t` must have a normal form
      let norm = normalize ctx t
      printLnIO $ "'" ++ show t ++ "' reduces to '" ++ show norm ++ "'"
      -- Update `ans`
      put (ctx,t,lockedNames)
      return True

checkSingleToplevel' (Declaration s t) = do
  -- Get the context with the locked names (i.e. erasing definitions
  -- of locked names)
  (unlockedCtx,_,lockedNames) <- get
  let ctx = getLockedCtx lockedNames unlockedCtx
  printLnIO $ "\nType-checking term '" ++ show s ++ "' of type '"
    ++ show t   ++ "'"
  case addDecl ctx (s,t) of
    Left err -> do
      liftIO . showErr $ err
      return False
```

```haskell
    Right ctx' -> do
      printLnIO "Declaration check OK!"
      -- Update `ans`
      put (ctx',t,lockedNames)
      return True

checkSingleToplevel' (Definition s t e) = do
  -- Get the context with the locked names (i.e. erasing definitions
  -- of locked names)
  (unlockedCtx,_,lockedNames) <- get
  let ctx = getLockedCtx lockedNames unlockedCtx
  printLnIO $ "\nType-checking term '" ++ show s ++ "' of type '"
    ++ show t  ++ "' and body '" ++ show e ++ "'"
  case addDef ctx (s,t,e) of
    Left err -> do
      liftIO . showErr $ err
      return False
    Right ctx' -> do
      printLnIO "Type check OK!"
      -- Update `ans`
      put (ctx',e,lockedNames)
      return True

-- Main REPL (infinite) loop
doRepl :: StateT ReplState IO ()
doRepl = do
  (ctx,ans,lockedNames) <- get
  printIO "\n> "
  s <- liftIO getLine
  let w = words s
  case w of
    -- Quit
    [":q"] -> do
      liftIO exitSuccess
    -- Print last type-checked term
    [":ans"] -> do
      printLnIO $ show ans
    -- Show context (with locked names)
    [":ctx"] -> do
      liftIO . printCtxLn $ getLockedCtx lockedNames ctx
    -- Delete from context the given names (and also the ones
    -- that depend on them)
    ":clear" : idents -> do
      let ctx' = foldl removeFromCtx ctx (map Ident idents)
      put (ctx',ans,lockedNames)
    -- Lock a list of identiers
    ":lock" : idents -> do
      let idents' = map Ident idents
          isInCtx = (`elem` (keys ctx))
```

```
              identsToAdd = filter isInCtx idents'
              identsWrong = filter (not . isInCtx) idents'
          when (length identsWrong > 0) $
            printLnIO $ "identifier(s) " ++ intercalate ", " (map show identsWrong)
                ++ " not found in the current context"
          let lockedNames' = identsToAdd ++ lockedNames
          put (ctx,ans,lockedNames')
        -- Unlock a list of identiers
        ":unlock" : idents -> do
          let lockedNames' = filter (`notElem` map Ident idents) lockedNames
          put (ctx,ans,lockedNames')
        -- Clear the list of locked identiers
        [":unlockall"] ->
          put (ctx,ans,[])
        -- Show the locked identifiers
        [":printlock"] ->
          printLnIO $ "Locked names are: " ++ intercalate ", " (map show lockedNames)
        -- Show help menu
        [":help"] -> do
          liftIO printUsage
        -- Unknown command
        (':' : _ ) : _ ->
          printLnIO "Command not found. Type :help"
        -- Otherwise, check a new declaration
        otherwise -> do
          let ts = myLexer s
          case pToplevel ts of
            Left err -> do
              printLnIO "\nParse failed!"
              liftIO . showErr $ err
            Right toplevel -> do
              _ <- checkSingleToplevel toplevel
              return ()
  doRepl -- Repeat

-- Add a definition to the current context
addDef :: Ctx -> (Ident,Term,Term) -> Either ErrorString Ctx
addDef ctx (s,t,e) = do
  checkType ctx emptyDirEnv t Universe -- Check that `t` is a type
  let tVal = eval ctx t
  checkType ctx emptyDirEnv e tVal -- Check that `e` has type `t`
  Right $ extend ctx s (Def t e)

-- Add a definition to the current context
addDecl :: Ctx -> (Ident,Term) -> Either ErrorString Ctx
addDecl ctx (s,t) = do
  -- Check that `t` is a type or the interval
  unless (t == I) $ checkType ctx emptyDirEnv t Universe
  Right $ extend ctx s (Decl t)
```

```haskell
-- Lock each given identifier in the context,
-- i.e. erase its eventual definition from the context
getLockedCtx :: [Ident] -> Ctx -> Ctx
getLockedCtx idents ctx0 = foldr getLockedCtx' ctx0 idents
  where
    getLockedCtx' :: Ident -> Ctx -> Ctx
    getLockedCtx' s ((s',Def ty def) : ctx) =
      if s == s' then (s,Decl ty) : ctx
              else (s',Def ty def) : getLockedCtx' s ctx
    getLockedCtx' s ((s',Decl ty) : ctx) =
      (s',Decl ty) : getLockedCtx' s ctx
    getLockedCtx' _ ctx = ctx

-- Print the context, line by line
printCtxLn :: Ctx -> IO ()
printCtxLn ctx = mapM_ (putStrLn . showEntry) (reverse ctx)

-- Print an error
showErr :: String -> IO ()
showErr err = putStrLn $ "\nError: " ++ err

main :: IO ()
main = do
  printUsage
  args <- getArgs
  case args of
    -- No files given: start the REPL loop
    [] -> do
      evalStateT doRepl initReplState
      exitSuccess
    -- Some files given: parse each file, then start the REPL loop
    fs -> evalStateT (
      do
        -- `b` is the result of the type-check of each file
        res <- foldM (\b fp -> (b &&) <$> runFile fp) True fs
        liftIO $ unless res exitSuccess
        (ctx,_,_) <- get
        printLnIO "\nCurrent context is:"
        liftIO . printCtxLn $ ctx
        doRepl
      ) initReplState

-- Print help menu
printUsage :: IO ()
printUsage = do
  putStr $ unlines
    [ " ---------------------------------------------------------------------- "
    , "| Usage: ./CTT <file> .. <file>    load and type-check files           |"
```

```
\item                                    then start a REPL              |"
      , "| Commands:                                                    |"
\item x : <term>            add declaration of 'x' to the context       |"
\item x : <term> = <term>   add definition of 'x' to the context        |"
\item <term>                infer type of t and normalize it            |"
\item :help                 print help                                  |"
\item :q                    quit                                        |"
\item :ans                  print the last term used                    |"
\item :ctx                  print current context                       |"
\item :clear <id> .. <id>   remove <id>'s from context (recursively)    |"
\item :lock <id> .. <id>    lock <id>'s definition                      |"
\item :unlock <id> .. <id>  unlock <id>'s definition                    |"
\item :unlockall            unlock every currently locked definition    |"
\item :printlock            print locked definitions                    |"
      , " -------------------------------------------------------------------- "
      ]
  hFlush stdout

-- Auxiliary printing functions
printIO :: String -> StateT ReplState IO ()
printIO s = liftIO $ do
  putStr s
  hFlush stdout

printLnIO :: String -> StateT ReplState IO ()
printLnIO s = printIO $ s ++ "\n"
```

Appendix A

Yoneda lemma

In this section we briefly state and prove a basic version of the (contravariant) Yoneda lemma, which is used in chapter 2.

Definition A.0.1 (Yoneda embedding)**.** *Let $\mathcal{C}$ be a locally small category, i.e. such that $\mathsf{Hom}_{\mathcal{C}}(I, J)$ is a set for all $I, J \in \mathcal{C}$. The Yoneda embedding is the functor $\mathsf{y} : \mathcal{C} \to \hat{\mathcal{C}}$ given by*

- *$(\mathsf{y}I)(J) = \mathsf{Hom}_{\mathcal{C}}(J, I)$ for each object $J \in \mathcal{C}$;*

- *$(\mathsf{y}I)_f : \mathsf{Hom}_{\mathcal{C}}(J, I) \to \mathsf{Hom}_{\mathcal{C}}(K, I)$ which is defined by $h \mapsto fh$,[1] for each morphism $f : K \to J$ of $\mathcal{C}$.*

Theorem A.0.2 (Yoneda lemma)**.** *Let $\mathcal{C}$ be a locally small category and Γ a presheaf on $\mathcal{C}$, i.e. a functor $\Gamma : \mathcal{C}^{op} \to \mathsf{Set}$. Then for each $I \in \mathcal{C}$ there is bijection*

$$\Gamma(I) \cong \mathsf{Hom}_{\hat{\mathcal{C}}}(\mathsf{y}I, \Gamma).$$

Proof. If $\alpha : \mathsf{y}I \to \Gamma$ is a natural trasformation, with components $\alpha_J : \mathsf{Hom}_{\mathcal{C}}(J, I) \to \Gamma(J)$ for each $J \in \mathcal{C}$, then the naturality condition holds for each $f : J \to K$, i.e.

$$\alpha_J \circ (\mathsf{y}I)_f = \Gamma_f \circ \alpha_K$$

which means that for each $h : K \to I$,

$$\alpha_J(fh) = \Gamma_f(\alpha_K h).$$

Exploiting this, we now show how every element of $\Gamma(I)$ determines a natural transformation between $\mathsf{y}I$ and Γ, and viceversa; furthermore, we show that these correspondences are each the inverse of the other.

---

[1]Remember that we use the applicative order, i.e. $fh = h \circ f$.

$$\begin{array}{ccc}
\mathsf{Hom}_{\mathcal{C}}(K, I) & \xrightarrow{\ (\mathsf{y}I)_f\ } & \mathsf{Hom}_{\mathcal{C}}(J, I) \\
\downarrow{\scriptstyle \alpha_K} & & \downarrow{\scriptstyle \alpha_J} \\
\Gamma(K) & \xrightarrow[\ \Gamma_f\ ]{} & \Gamma(J)
\end{array}$$

Figure A.1: Naturality diagram.

- Given $\alpha : \mathsf{y}I \to \Gamma$, we get $\hat{\alpha} = \alpha_I(\mathbf{1}_I) \in \Gamma(I)$ since $\alpha_I : \mathsf{Hom}_{\mathcal{C}}(I, I) \to \Gamma(I)$.

- Given $x \in \Gamma(I)$ and $f : J \to I$, we get $\Gamma_f : \Gamma(I) \to \Gamma(J)$ so that $\Gamma_f(x) \in \Gamma(J)$. Therefore we set $\tilde{x}_J(f) = \Gamma_f(x)$. We now prove that $\tilde{x} = (\tilde{x}_J)_{J \in \mathcal{C}}$ is a natural trasformation. Indeed, given $h : K \to I$, the upper path of diagram A.1 with $\tilde{x}$ instead of α gives $\tilde{x}_J(fh) = \Gamma_{fh}(x)$, while the lower path yields $\Gamma_f(\tilde{x}_K(h)) = \Gamma_f(\Gamma_h(x)) = \Gamma_{fh}(x)$, where the last step if justified by the functoriality of Γ.

- (Second applied to first) If $x \in \Gamma(I)$, then

$$\hat{\tilde{x}}$$

- (First applied to second) If $\alpha : \mathsf{y}I \to \Gamma$, $\tilde{\hat{\alpha}} = \widetilde{\alpha_I(\mathbf{1}_I)}$, so that

$$\tilde{\hat{\alpha}}(f) = \widetilde{\alpha_I(\mathbf{1}_I)}(f) = \Gamma_f(\alpha_I\mathbf{1}_I).$$

  From diagram A.1, by choosing $K = I$ and $h = \mathbf{1}_I$, we obtain $\alpha_J(f) = \Gamma_f(\alpha_I\mathbf{1}_I)$, which allows us to finally conclude that $\tilde{\hat{\alpha}} = \alpha$.

  $\square$

Remark A.0.3. *Actually we could say more about the Yoneda embedding* y, *as it can be shown that:*

1. *$\Gamma(I) \cong \mathsf{Hom}_{\hat{\mathcal{C}}}(\mathsf{y}I, \Gamma)$ is a natural isomorphism both in $I \in \mathcal{C}$ and $\Gamma \in \hat{\mathcal{C}}$.*

2. *y is a fully faithful functor.*

We refer to [Lei14] for further details, since these additional properties are not used in this thesis.

Bibliography

[AW09]   S. Awodey and M. A. Warren. Homotopy theoretic models of identity types. *Mathematical Proceedings of the Cambridge Philosophical Society*, 146(1):45–55, 2009.

[Ben22]   Bruno Bentzen. Naive cubical type theory. *Mathematical Structures in Computer Science*, page 1–27, 2022.

[CCHM15]   Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. Cubical Type Theory: a constructive interpretation of the univalence axiom. In *21st International Conference on Types for Proofs and Programs*, number 69 in 21st International Conference on Types for Proofs and Programs, page 262, Tallinn, Estonia, May 2015. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[Coq96]   Thierry Coquand. An algorithm for type-checking dependent types. *Science of Computer Programming*, 26(1):167–177, 1996.

[Gol79]   Robert Goldblatt. *Topoi, the categorial analysis of logic / Robert Goldblatt.* Elsevier North-Holland Amsterdam, 1979.

[Hof97]   Martin Hofmann. Syntax and semantics of dependent types. In *Semantics and Logics of Computation*, pages 79–130. Cambridge University Press, 1997.

[How80]   William Alvin Howard. The formulae-as-types notion of construction. In Haskell Curry, Hindley B., Seldin J. Roger, and P. Jonathan, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism.* Academic Press, 1980.

[HS96]   Martin Hofmann and Thomas Streicher. The groupoid interpretation of type theory. In *In Venice Festschrift*, pages 83–111. Oxford University Press, 1996.

[Hub16]   Simon Huber. *Cubical Intepretations of Type Theory*. PhD thesis, University of Gothenburg, 2016.

[KL21]   Krzysztof Kapulkin and Peter LeFanu Lumsdaine. The Simplicial Model of Univalent Foundations (after Voevodsky). *J. Eur. Math. Soc.*, page 2071–2126, 2021.

[Lei14]   T. Leinster. *Basic Category Theory*. Cambridge Studies in Advanced Mathematics. Cambridge University Press, 2014.

[ML75]   Per Martin-Löf. An intuitionistic theory of types: Predicative part. In H.E. Rose and J.C. Shepherdson, editors, *Logic Colloquium '73*, volume 80 of *Studies in Logic and the Foundations of Mathematics*, pages 73–118. Elsevier, 1975.

[MLM92]   Saunders Mac Lane and Ieke Moerdijk. *Sheaves in Geometry and Logic a First Introduction to Topos Theory*. Springer New York, 1992.

[OP18]   Ian Orton and Andrew M. Pitts. Axioms for modelling cubical type theory in a topos. *Log. Methods Comput. Sci.*, 14(4), 2018.

[Uni13]   The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. `https://homotopytypetheory.org/book`, Institute for Advanced Study, 2013.

[vBJ77]   L. van Benthem Jutting. *Checking Landau's "Grundlagen" in the Automath system*. PhD thesis, StichtingMathematisch Centrum, 1977.

[Voe06]   Vladimir Voevodsky. A very short note on homotopy λ-calculus, 2006.