Inductive types in constructive languages

Bruin, Peter Johan de

Chapter 2

The language ADAM

In this chapter we introduce our mathematical language named *ADAM*. It serves to provide clear and precise notations for mathematical constructions based on strong typing, including generalized products and sums.

ADAM is based on a formal system, ATT, that is an extension of Martin-Löf's Intuitionistic Type Theory. This type theory is described in appendix B, but it is not necessary to study it separately if one has some acquaintance with generalized type theories. The primitive notions that are included are generalized products, generalized sums, finite types, a cumulative hierarchy of universes, a universe of propositions, impredicative quantification for propositions, the equality predicate, and existential propositions with strong elimination. Furthermore, there are no inductive types except a type of natural numbers. The system is powerful enough to serve as a mathematical foundation of *ADAM*, for set theory (ZFC) can be encoded in it, using the first universe. Conversely, all primitives can be given a set-theoretical interpretation (section B.10) within ZFC extended with a hierarchy of universes, so we have a set-theoretical semantics for *ADAM* as well.

The theory contains many simpler typed lambda calculi as subsystems, and most of our constructions are valid in some of these too.

While being formal, *ADAM* attempts to come much closer to the natural way of expression of informal mathematical language than most other systems based on Type Theory. We achieve this by making a sharp distinction between the abstract expressions that are manipulated by the formal system (sometimes called the "deep structure" or "kernel language"), and the concrete text as it is written down by the mathematician.

In type theory, propositions and proof objects are represented as a kind of types and objects. We have not developed a really natural notation for proof construction, so we describe proofs mainly in ordinary English, sometimes using an equational style.

2.1 Language definition mechanism

Mathematical languages are often defined by first defining a kernel language and subsequently adding "syntactic sugar". We choose for a more sophisticated correspondence between the concrete language and its basic type theory, for the following reasons.

1. There are new syntactic classes, like 'declarations', that do not correspond to expressions in the primitive type theory.

2. Context information for an expression contains more information about the syntactic forms that are allowed for that expression than contexts in the primitive type theory.

3. We wish to have the possibility to introduce new sublanguages within *ADAM* that may have a structure very different from type theory.

We seek to define the language by a form of two-level grammar in the style of Van Wijngaarden [86], but using abstract trees as parameters rather than strings. The grammar formalism is informally described in this section. The description of *ADAM* in the following sections uses this grammar formalism, but some of the more complicated features will be described merely by example.

2.1.1 Two-level grammar. The first level consists of a number of *abstract* syntax classes A, which are defined by context-free production rules. Members of these classes are abstract trees, rather than strings as in Van Wijngaarden grammar [86].

The second level consists of a number of parametrized *concrete* syntax classes $T(\bar{x})$, where $\bar{x}$ is a sequence of parameters or meta-variables x_i, each ranging over an abstract syntax class A_i. Names of syntax classes of both levels are in *Slanted* font and begin with a capital. The concrete classes are defined by production rules which may contain meta-variables. Members of a concrete class (with actual parameters) are pieces of concrete text. The parameters serve to pass information from the context into the derivation of a piece of concrete text and vice versa.

Thus, our notion of two-level grammar is roughly the same as the notion of *Definite Clause Grammar* (or unification grammar) [2, page 79–80] used in the logic programming community. It has also some similarity with the notion of *attribute grammar*, except that we expect parsing to be done simultaneously with parameter instantiation, and do not allow a parse tree to be recomputed with different parameter values. There is no need even to mention parse trees. See Małuszyński [53] for a comparison between Definite Clause Grammars and attribute grammars.

The distinction between the two levels is not really necessary: a first level class A can be seen as a second-level class $A(x)$ on type-free trees, and its context-free production rules as abbreviations for second-level production rules.

2.1.2 Derived notions. We introduce some (syntactic) *predicates* $p(\bar{x})$ over abstract classes. These are inductively defined by Horn clause logic with equality. Clauses are written using '$\Leftarrow$' as main operator. Each such predicate may be identified with a syntax class that contains the empty string ϵ just when the predicate holds. To effect this, replace each Horn clause $p(\bar{x}) \Leftarrow \bar{q}(\bar{x})$ by a production rule $p(\bar{x}) \longrightarrow \bar{q}(\bar{x})$, and any production rule containing a condition $p(\bar{x})$ will be blocked when $p(\bar{x})$ is not rewritable into ϵ.

We also introduce some (syntactic) *operations* $f(\bar{x})$ on abstract classes. Such an operation may be replaced by a predicate $f'(\bar{x}, y)$ that holds just when $f(\bar{x})$ equals y.

The parameters of a class can often be separated into input (or *inherited*) and output (or *synthesized*) parameters. The actual value of an input parameter is assumed to be fixed by the production rules that invoke the syntax class. The actual value of an output parameter is to be determined by the production rules for the class itself.

We use a *generic* abstract class, A^*, for any class A. This class contains all finite (possibly empty) sequences of members of A.

2.1.3 Grammar notation. Syntactic predicate and operation names are in lower case.

The production rules for abstract classes are given in Backus-Naur Form, using meta-symbols '::=', '|', and '.', and braces '{', '}' for grouping. In particular, '{}' stands for an empty production. All other symbols are terminal symbols.

The production rules for concrete classes are written using '$\longrightarrow$' as main operator, a comma for string concatenation, terminal symbols between double quotes, and using italic or greek symbols as variables.

Rules for predicates are written using $\Leftarrow$ as main operator. Note that string concatenation for classes becomes conjunction for predicates. For any abstract class, we assume syntactic equality and inequality predicates, $(a = a')$ and $(a \neq a')$.

2.2 Basic grammar of ADAM

This section introduces the abstract and concrete classes that we will use. The next section (2.3) gives production rules for general use. The other sections describe specific language features; most of these are formalized by means of definitions to be collected in a *standard environment*. Several features require additional syntax classes and production rules; these are only informally described, either by means of example or by means of definitions that treat only some special cases.

The main context-free classes are (abstract) *terms* and *contexts*, just as in B.1. The rules in appendix B define a predicate '$\Gamma \vdash t{:}\,T$' for contexts Γ and terms t and T, meaning that under the assumptions in Γ, abstract term t has type T.

The main concrete syntax class is $\mathit{Term}_{\Gamma,\gamma}(t, T)$, where context Γ records all assumptions $v{:}\,A$ currently made, γ is an *environment* recording all name bindings and other definitions. Now, if T represents a type (i.e. $\Gamma \vdash T{:}\,\mathbf{Type}_i$, where i is called the level of T), then a concrete text produced by $\mathit{Term}_{\Gamma,\gamma}(t, T)$ denotes the abstract term t, and the production rules shall be such that $\Gamma \vdash t{:}\,T$ holds. (We conceive that this might be checked by an appropriate grammar manipulation tool.)

2.2.1 Abstract classes. We have the following abstract classes, listed together with the meta-variables that range over them.

α	A^*	: For any syntax class A: finite sequences of expressions of class A
v	*Var*	: Abstract variables
c, Q	*Const*	: Abstract primitive constants

i	*Nat*	: Natural numbers at the syntactic level, e.g. to index the hierarchy of universes
t, a, b	*Term*	: Terms, which denote objects, including types
T, A, B	*Type*	: Types, being just terms
Γ, Δ	*Context*	: Contexts, being sequences of assumptions like $v{:}\,T$
x	*Name*	: Names (identifiers) used in concrete text
γ, δ	*Env*	: Environments, being sequences of environment items
	EnvItem	: Environment items, being either name bindings or coercions (This may be extended.)
ϕ	*Subst*	: Substitutions for abstract variables

We distinguish between variables used in abstract terms and names written in concrete text for the following reasons:

- Concrete names may be ambiguous; abstract variables must be unambiguous.
- The concrete name x used in a concrete abstraction, $(x :: b)$, may be different from the name y used in the type of the abstraction, $\Pi(y{:}\,A :: By)$.
- When using pattern matching, for example $((x, y) :: b)$, multiple concrete names refer to components of the value of the same unnamed abstract variable.

An environment γ contains, for any visible name x, an item $x{:}\,T := t$ that gives the type and (abstract) value of x. This value may be either the abstract variable named by x, or the value of x as given by some definition. The environment may furthermore specify other information that influences the parsing of concrete terms, such as coercions (described in 2.3.5), and infix operator symbols (not formally described here).

For *Var* and *Const*, see 2.2.3. The other classes are given by:

$$
\begin{array}{rcl}
\mathit{Nat} & ::= & 0 \mid \mathit{Nat}' \ . \\
A^* & ::= & \{\} \mid A\, A^* \ . \\
\mathit{Term} & ::= & \mathit{Var} \\
 & \mid & \mathit{Const}(\{\mathit{Term},\}^*) \\
 & \mid & (\mathit{Var} :: \mathit{Term}) \ . \\
\mathit{Type} & ::= & \mathit{Term} \ . \\
\mathit{Context} & ::= & \{\mathit{Var}{:}\ \mathit{Type};\}^* \ . \\
\mathit{EnvItem} & ::= & \mathit{Name}{:}\ \mathit{Type} := \mathit{Term} \\
 & \mid & \mathit{Const}(\mathit{Context}){:}\ \mathit{Type} := \mathit{Term} \\
 & \mid & \mathit{Context} \vdash \mathit{Term} : \mathit{Type} \subseteq_{\mathsf{t}} \mathit{Type} \ . \\
\mathit{Env} & ::= & \{\mathit{EnvItem};\}^* \ . \\
\mathit{Subst} & ::= & \{\mathit{Var} := \mathit{Term},\}^* \ .
\end{array}
$$

(*EnvItem* may be extended with other kinds of environment information.)

2.2.2 Concrete classes. We did already introduce the class $Term_{\Gamma,\gamma}(t,T)$ of terms of inherited type T. Besides we have a class $TTerm_{\Gamma,\gamma}(t,T)$ of terms of synthesized type T. Thus, expressions of this class define both the abstract term t and its type T, and we will have both $\Gamma \vdash T\colon \mathbf{Type}_i$ for some i, and $\Gamma \vdash t\colon T$.

$Term_{\Gamma,\gamma}(t,T)$	: A term t of inherited type T
$TTerm_{\Gamma,\gamma}(t,T)$	: A term t of synthesized type T, guaranteed to be correct (if Γ,γ are correct)
$Def_{\Gamma,\gamma}(\delta)$	: A definition yielding the new environment δ
$Defs_{\Gamma,\gamma}(\delta)$	: A (nonempty) sequence of definitions
$Decl_{\Gamma,\gamma}(\Delta,\delta,i)$	: A declaration yielding the new assumptions Δ and environment δ, containing only types up to level i
$Pat_{\Gamma,\gamma}(T,t,\delta)$	: An exhaustive pattern for type T that, when matched against term t, yields name bindings δ

Predicates are the following:

$(a = a')$	: a is structurally equal to a'
$(v \neq v')$	: variable v is different from v'
$(t => t')$	: term t reduces to head normal form t'
$(t == t')$	: term t is convertible to t'
$\Gamma \vdash t\colon T$	: in context Γ, t is a term of type T
$in(a,\alpha)$	: a occurs in list α
$fresh(v,\Gamma)$	: Abstract variable v is fresh with respect to Γ
$valu_\Gamma(\phi,\Delta)$	: ϕ is a valid valuation for Δ in context Γ

Predicates $=>$, $==$, and $\vdash$ are given in appendix B. Predicates *in*, *fresh*, and *valu* are given by:

$$
\begin{aligned}
in(a, a\$\alpha) &\quad . \\
in(a, b\$\alpha) &\Leftarrow in(a,\alpha)\ . \\
fresh(v, \{\}) &\quad . \\
fresh(v, \{v'\colon T;\ \Gamma\}) &\Leftarrow (v' \neq v), fresh(v,\Gamma)\ . \\
valu_\Gamma(\{\},\{\}) &\quad . \\
valu_\Gamma(\{\phi, x := t\}, \{\Delta;\ x\colon T\}) &\Leftarrow valu_\Gamma(\phi,\Delta),\ \Gamma \vdash t[\phi]\colon T[\phi]\ .
\end{aligned}
$$

Syntactic operations:

$\alpha\alpha'$	: List concatenation
$t[\phi]$	: Term t under substitution ϕ
$repq(Q,\Delta,t)$	: Repeatedly apply quantifier Q for all typings in Δ to term t

List concatenation and term substitution are defined as usual, taking care for variable renaming. Substitution of a single variable is needed for term reduction in appendix B; substitution of multiple variables is currently only used in our description of coercion, subsection 2.3.5. Repeated quantifier application is defined in 2.5.5.

2.2.3 Identifiers. We consider availability of several styles of identifiers indispensable for writing serious mathematical texts (and hope that automatic proof checkers will soon provide them).

We use single-character names in italic, greek, or calligraphic font, and multiple-character names in sans-serif and boldface font. So the expression 'fax' is the juxtaposition of three variables, while 'fax' is a single constant. In patterns, the underscore '_' will act as an anonymous variable.

Adding a prime '$'$' builds a new name. Other decorations such as subscripts usually denote some operation applied to the undecorated symbol, but may also be used to build new identifiers if this is explicitly stated.

Furthermore, we sometimes use other mathematical symbols, sometimes written in infix (or postfix, outfix, mixfix, ...) position.

Capitalized identifiers are normally used for types, boldface identifiers for types of types, sets of sets, categories etc. When introducing new notation, characters from the end of the alphabet usually stand for arbitrary identifiers, while other symbols stand for expressions.

2.3 Production rules

2.3.1 Terms

As said, we distinguish between terms with inherited and with synthesized type. A typical example of a term with synthesized type is a variable occurence. A term with synthesized type may be used where an inherited type is already given, provided both types are convertible. Terms of both classes may be surrounded by parentheses, and may make use of local definitions.

$$
\begin{aligned}
TTerm_{\Gamma,\gamma}(t,T) &\longrightarrow \mathit{Name}\, x,\ \mathit{in}(\{x{:}\,T := t\},\gamma).\\
Term_{\Gamma,\gamma}(t,T) &\longrightarrow TTerm_{\Gamma,\gamma}(t,T'),(T == T').\\
TTerm_{\Gamma,\gamma}(t,T) &\longrightarrow \text{“(”}, TTerm_{\Gamma,\gamma}(t,T), \text{“)”}.\\
Term_{\Gamma,\gamma}(t,T) &\longrightarrow \text{“(”}, Term_{\Gamma,\gamma}(t,T), \text{“)”}.\\
TTerm_{\Gamma,\gamma}(t,T) &\longrightarrow \text{“}\underline{\mathsf{let}}\text{ ”}, Defs_{\Gamma,\gamma}(\delta), \text{“ }\underline{\mathsf{in}}\text{ ”}, TTerm_{\Gamma,\gamma\delta}(t,T).\\
Term_{\Gamma,\gamma}(t,T) &\longrightarrow \text{“}\underline{\mathsf{let}}\text{ ”}, Defs_{\Gamma,\gamma}(\delta), \text{“ }\underline{\mathsf{in}}\text{ ”}, Term_{\Gamma,\gamma\delta}(t,T).\\
Term_{\Gamma,\gamma}((v :: b),\Pi(A;B)) &\longrightarrow \mathit{fresh}(v,\Gamma), \text{“(”}, Pat_{\Gamma,\gamma}(A,v,\delta),\\
&\qquad \text{“::”}, Term_{\{\Gamma;v:A\},\gamma\delta}(b,B(v)), \text{“)”}.\\
TTerm_{\Gamma,\gamma}((fa),(Ba)) &\longrightarrow TTerm_{\Gamma,\gamma}(f,\Pi(A;B)), Term_{\Gamma,\gamma}(a,A).
\end{aligned}
$$

As one sees, the context and environment parameters Γ,γ are always passed on to subexpressions. Production rules would be quite easier to read if we could omit them from our rule notation. For now, we will accept the load.

2.3.2 Patterns

The class $Pat_{\Gamma,\gamma}(T,t,\delta)$ produces exhaustive patterns for type T that, when matched against term t, yield name bindings δ. The simplest form of patterns are single named

variables, resulting in a single binding, and the anonymous variable '_', resulting in no binding at all.

For composite patterns, we give only a rule for patterns that match dependent pairs (section 2.6). See subsection 2.12.6 for a more general idea of patterns.

$$\begin{aligned} Pat_{\Gamma,\gamma}(T,t,\{x{:}\,T := t\,,\}) &\longrightarrow Name\,x. \\ Pat_{\Gamma,\gamma}(T,t,\{\}) &\longrightarrow \text{“\_”}. \\ Pat_{\Gamma,\gamma}(\Sigma(A;B),t,\alpha\beta) &\longrightarrow \text{“(”}, Pat_{\Gamma,\gamma}(A,\mathsf{fst}\,t,\alpha), \\ &\qquad \text{“;”}, Pat_{\Gamma,\gamma}(B(\mathsf{snd}\,t),\mathsf{snd}\,t,\beta),\text{“)”}. \end{aligned}$$

2.3.3 Definitions

A definition introduces some typed identifiers, and assigns a value to them. We have several forms of definition; not all of these formally described. The class $Def_{\Gamma,\gamma}(\delta)$ produces a single definition, the class $Defs_{\Gamma,\gamma}(\delta)$ a sequence of them.

A (concrete) *simple definition* may have the form $\xi{:}\,T := t$, where ξ is a pattern and T and t are expressions denoting a type and a term of that type. A simplified form of definition is $\xi := t$, which is possible only if t has a synthesized type T. As the rules indicate, a definition yields the bindings obtained by handing the value of the term t over to the pattern ξ.

$$\begin{aligned} Def_{\Gamma,\gamma}(\delta) &\longrightarrow Pat_{\Gamma,\gamma}(T,t,\delta), \text{“:”}, \\ &\qquad Term_{\Gamma,\gamma}(T,\mathbf{Type}_i), \text{“:=”}, Term_{\Gamma,\gamma}(t,T). \\ Def_{\Gamma,\gamma}(\delta) &\longrightarrow Pat_{\Gamma,\gamma}(T,t,\delta), \text{“:=”}, TTerm_{\Gamma,\gamma}(t,T). \\ Defs_{\Gamma,\gamma}(\delta) &\longrightarrow Def_{\Gamma,\gamma}(\delta). \\ Defs_{\Gamma,\gamma}(\delta\delta') &\longrightarrow Def_{\Gamma,\gamma}(\delta), \text{“;”}, Defs_{\Gamma,\gamma\delta}(\delta'). \end{aligned}$$

Secondly we have *parametrized definitions* in various forms, which we do not formally define. Some of these are:

1. Primarily, a parametrized definition consists of some new constant c followed by a type declaration Δ of its parameters, its result type T and defining expression t :

$$\mathsf{c}(\Delta){:}\,T := t$$

2. Instead of declaring the parameters after the constant, they may be declared in front of the definition, separated by the symbol '$\vdash$' in which case only the sequence of variable names $\bar{x}$ declared in Δ appear after the constant:

$$\Delta_{\bar{x}} \vdash \quad \mathsf{c}(\bar{x}){:}\,T := t$$

(Do not confuse this use of '$\vdash$' *inside* the language with the syntactic (meta-) predicate $\vdash$!)

3. Parameters can be made implicit by omitting them from the sequence $\bar{x}$. This is often done for type parameters, e.g.

$$A, B{:}\,\mathbf{Type};\ (x,y){:}\,A\times B \vdash \quad \mathsf{swap}(x,y){:}\,B\times A := (y,x)$$

4. A parameter typing itself can be made implicit by preceding the definition with a separate variable declaration 'Variables $x{:}\,T;\ldots$', like:

$$\begin{array}{l}\underline{\mathsf{Variables}}\ x{:}\,\mathbb{R}_+;\ n{:}\,\mathbb{N}_+;\\ \sqrt[n]{x} := x^{1/n};\\ \sqrt{x} := \sqrt[2]{x}\end{array}$$

   Thus, whenever a variable, that is a declared in a Variables declaration, appears untyped in a subsequent definition, it is implicitly typed by the declaration. We have as yet no clear scope rules for variable declarations.

Later on we will allow several other forms of definitions, e.g.:

- Coercions ($\ldots \subseteq_{\mathsf{t}} \ldots$) in 2.3.5
- Several forms of definition by giving a characteristic property (Define ... by ...) in 2.6.3 and 2.8.5
- Sum types given in BNF form ($S ::= \ldots \mid \ldots$) in 2.8.4

We have yet no formal scheme to define (or declare) new constructs that bind local variables. But we will use a *pseudo definition* that is suggestive of the intended notation, like:

$$(x{:}\,A \vdash b_x{:}\,B) \vdash \quad (x \mapsto b_x) \ := \ \ldots$$

2.3.4 Declarations

A declaration in $Decl_{\Gamma,\gamma}(\Delta,\delta,i)$ is a sequence that may contain typings like $\xi{:}\,T$, where ξ is a pattern, as well as definitions as above. A special form of typing consists of only a type T, and represents an anonymous assumption $\_{:}\,T$. This is intended to be used only when T is a proposition.

For each typing $\xi{:}\,T$, the synthesized parameter Δ contains an abstract assumption $x{:}\,T$, and δ contains the name bindings that are yielded by matching pattern ξ against x.

$$\begin{array}{rcl}
Decl_{\Gamma,\gamma}(\{\},\{\},i) & \longrightarrow & .\\
Decl_{\Gamma,\gamma}(\{v{:}\,T;\ \Delta\},\delta\delta',i) & \longrightarrow & fresh(v,\Gamma), Pat_{\Gamma,\gamma}(T,v,\delta), \text{“:”}, Term_{\Gamma,\gamma}(T,\mathbf{Type}_i),\\
 & & \text{“;”}, Decl_{\{\Gamma;\,v:T\},\gamma\delta}(\Delta,\delta',i).\\
Decl_{\Gamma,\gamma}(\{v{:}\,T;\ \Delta\},\delta',i) & \longrightarrow & fresh(v,\Gamma), Term_{\Gamma,\gamma}(T,\mathbf{Type}_i),\\
 & & \text{“;”}, Decl_{\{\Gamma;\,v:T\},\gamma}(\Delta,\delta',i).\\
Decl_{\Gamma,\gamma}(\Delta,\delta\delta',i) & \longrightarrow & Def_{\Gamma,\gamma}(\delta), \text{“;”}, Decl_{\Gamma,\gamma\delta}(\Delta,\delta',i).
\end{array}$$

2.3.5 Coercion

We sometimes wish to allow objects of one type to be "coerced" into objects of another type, by implicitly applying some conversion function. These coercions may be user-defined, and are recorded in the environment by an item of the form $\Delta \vdash f : T \subseteq_{\mathsf{t}}$

T'. Here, f is the coercion function from type T to T', and Δ declares substitution parameters for f, T, and T'. The rules below describe how coercions are defined and implicitly applied; functions are described in 2.5.3.

$$\begin{aligned} \mathit{Def}_{\Gamma,\gamma}(\{\Delta \vdash f : T \subseteq_{\mathsf{t}} T' , \}) \quad &\longrightarrow \quad \mathit{Decl}_{\Gamma,\gamma}(\Delta, \delta), \text{“}\vdash\text{”}, (\Gamma' = \Gamma\Delta), (\gamma' = \gamma\delta), \\ &\qquad \mathit{Term}_{\Gamma',\gamma'}(f, (T \to T')), \text{“:”}, \\ &\qquad \mathit{Term}_{\Gamma',\gamma'}(T, \mathbf{Type}_i), \text{“}\subseteq_{\mathsf{t}}\text{”}, \mathit{Term}_{\Gamma',\gamma'}(T', \mathbf{Type}_i). \\ \mathit{Term}_{\Gamma,\gamma}(f[\phi].t, U) \quad &\longrightarrow \quad \mathit{in}(\{\Delta \vdash f : T \subseteq_{\mathsf{t}} T'\}, \gamma), \\ &\qquad \mathit{valu}_{\Gamma}(\phi, \Delta), (U == T'[\phi]), \mathit{Term}_{\Gamma,\gamma}(t, T[\phi]). \end{aligned}$$

Note that there are as yet no restrictions on the coercion function, so that one can define very misleading coercions. It would be desirable to require, but difficult to enforce, that if the transitive closure of the set of all coercions in effect contains any circularity $f\colon T \subseteq_{\mathsf{t}} T$, then the coercion function f should be the identity.

This formalization of $\subseteq_{\mathsf{t}}$ is not wholly adequate to cover the use we make of it. For example, many type constructors preserve coercion, like the following. If

$$\begin{gathered} \varphi : A' \subseteq_{\mathsf{t}} A \ ; \\ \psi(x\colon A') : Bx \subseteq_{\mathsf{t}} B'x \end{gathered}$$

then we would like to have a coercion $\Pi(\varphi;\psi) : \Pi(A;B) \subseteq_{\mathsf{t}} \Pi(A';B')$, where $\Pi(\varphi;\psi)$ is the appropriate function ($p \mapsto (x :: \psi x.p(\varphi.x))$).

We will speak a bit loosely about coercions, like stating $T \subseteq_{\mathsf{t}} T'$ without giving the coercion function. We write $T =_{\mathsf{t}} T'$ if we have a bijective pair of coercions $T \subseteq_{\mathsf{t}} T'$ and $T' \subseteq_{\mathsf{t}} T$.

2.4 Types and universes

Types in context Γ are those terms T for which $\Gamma \vdash T\colon \mathbf{Type}_i$ is derivable, for some $\mathit{Nat}\, i$. This i is called the *level* of T. The constants $\mathbf{Type}_i$ are called *universes*. These are types themselves and form a cumulative hierarchy, for we have:

$$\begin{gathered} \mathbf{Type}_i\colon \mathbf{Type}_{i+1} \ , \\ \mathbf{Type}_i \subseteq_{\mathsf{t}} \mathbf{Type}_{i+1} \ . \end{gathered}$$

The latter rule means that any type in $\mathbf{Type}_i$ is in $\mathbf{Type}_{i+1}$ too; see paragraph 2.3.5 for $\subseteq_{\mathsf{t}}$.

These universes are called *predicative*, because the rules for introducing types in $\mathbf{Type}_i$ do not assume that $\mathbf{Type}_i$ itself is completely given. There is also an impredicative universe of propositions **Prop**, which we introduce in section 2.11, that is itself an element of $\mathbf{Type}_0$, and all propositions $P\colon \mathbf{Prop}$ are types in $\mathbf{Type}_0$ as well.

Many definitions can be given at any level, and the subscript i is often left implicit. Definitions that involve a universe, like Fam in 2.7, do actually define a hierarchy of type-constructors

$$T\colon \mathbf{Type}_i \vdash \mathsf{Fam}_i\, T\colon \mathbf{Type}_{i+1} \ .$$

2.5 Products and function spaces

The first paragraph of this and following sections describes a constant that is a primitive type constructor, constants for introducing and eliminating objects of such types, and some derived notation. In most cases, the types of these constants can be given in the standard declaration, but in some cases (including this section) special derivation rules will be needed.

The next paragraphs introduce notions that are derived from the primitive type constructor.

2.5.1 Products. Informally, if A is a type, and for any $x\colon A$ is B_x a type, then the *(generalized) product* $\Pi(x\colon A :: B_x)$ is a type containing all (infinitary) tuples $(x :: b_x)$, where $b_x\colon B_x$ for any $x\colon A$. This is a derived notation for $\Pi(A; (x :: B_x))$.

Selecting a component from a tuple is denoted by juxtaposition, so a tuple can act as a prefix *operator*. Alternative notations are subscripting and reverse application, using an infix '$\backslash$' which we took from DEVA [85].

$$\begin{array}{rcl}
A\colon \mathbf{Type};\ B\colon \Pi(A; (x :: \mathbf{Type})) & \vdash & \Pi(A; B)\colon \mathbf{Type}\ ; \\
p\colon \Pi(A; B);\ a\colon A & \vdash & pa\colon Ba\ ; \\
p_a & := & pa\ , \\
a\backslash p & := & pa \\
(x\colon A \vdash b_x\colon Bx) & \vdash & (x :: b_x)\colon \Pi(A; B) \\
(x\colon A :: B_x) & := & (A;\ (x :: B_x)) \\
p, p'\colon \Pi(A; B) \vdash \quad p =_{\Pi(A;B)} p' & \Leftrightarrow & \forall x\colon A :: px =_{Bx} p'x
\end{array}$$

The last line, which uses the equality predicate of section 2.10, is the *extensionality* rule stating that tuples are equal (but not necessarily convertible) just when their components are equal.

Finite products, like $B_0 \times B_1$, are defined as a generalized product over a finite type in paragraph 2.8.2.

2.5.2 Exponential types. Given types A and B, we write B^A for the type $\Pi(\_\colon A :: B)$ of (possibly infinitary) tuples.

$$A, B\colon \mathbf{Type} \vdash \quad B^A \ :=\ \Pi(\_\colon A :: B)$$

Single objects are identified with one-tuples via the obvious coercions: $B =_{\mathsf{t}} B^1$.

2.5.3 Function spaces. The type $A \to B$ of (total) functions from A to B is isomorphic to the type B^A of tuples, but we prefer to make the conceptual distinguish between these two types explicit. This allows us to define different coercions and other notations, like:

- Composition of tuples of functions, say $f, g\colon (A \to A)^N$, will be defined componentwise, $(f \mathbin{\bar{\circ}} g)_i = f_i \mathbin{\bar{\circ}} g_i$ (as arrows in the category $\mathbf{TYPE}^N$), which would be difficult if the typing were $f, g\colon N \to (A \to A)$.

- Single objects are identified with one-tuples: $B =_{\mathsf{t}} B^1$.
- Functions will occasionally be regarded as (single-valued) binary relations between A and B, by defining a coercion $(A \to B) \subseteq_{\mathsf{t}} \mathcal{P}(A \times B)$. In particular, we have $(A \to A) \subseteq_{\mathsf{t}} \mathcal{P}(A^2)$. We would not do this for tuples.

We define function space $A \to B$ by means of a Backus-Naur notation, that is described in paragraph 2.8.4, as the type containing an object λp for any tuple $p\colon B^A$. Thus, function space is like an Abstract Data Type, that is implemented by the tuple type.

The normal notation for function abstraction will be '$x \mapsto b_x$' where variable x is locally bound. Function application is denoted by an infix low dot, and defined by a case analysis (paragraph 2.8.5) on the only case for $f\colon A \to B$.

$$\begin{array}{rrcl} A, B\colon \mathbf{Type} \vdash & A \to B & ::= & \lambda(p\colon B^A)\ . \\ (x\colon A \vdash b_x\colon B) \vdash & x \mapsto b_x & := & \lambda(x :: b_x)\colon\ A \to B \\ f\colon A \to B;\ a\colon A \vdash & \underline{\mathsf{Define}}\ f.a\colon B\ \underline{\mathsf{by}} & & \\ & (\lambda p).a & := & pa \end{array}$$

Note that the definition of '$\mapsto$' is a pseudo definition because new forms of variable binding cannot be formally defined from within *ADAM*.

The coercion to binary relations is given by

$$A, B\colon \mathbf{Type} \vdash \quad f \mapsto \{x\colon A :: (x, f.x)\}\colon\ (A \to B) \subseteq_{\mathsf{t}} \mathcal{P}(A \times B)\ .$$

Identity functions, constant functions, backward and forward composition of functions are defined by:

$$\begin{array}{rrcl} & \mathsf{I}\colon A \to A & := & x \mapsto x \\ & \mathsf{K}(c\colon C)\colon A \to C & := & x \mapsto c \\ f\colon A \to B;\ g\colon B \to C \vdash & g \circ f\colon A \to C & := & x \mapsto g.(f.x)\ ; \\ & f \mathbin{\bar{\circ}} g\colon A \to C & := & g \circ f \end{array}$$

(Forward composition '$f \mathbin{\bar{\circ}} g$' is Hoare's composition operator '$f\,;g$'.)

Tuples and functions are combined in the following definitions:

$$\begin{array}{rrcl} & \pi_{a:A}\colon \Pi(A; B) \to Ba & := & p \mapsto pa \\ f_{x:A}\colon C \to Bx \vdash & \langle f\rangle\colon C \to \Pi(A; B) & := & z \mapsto (x :: f_x.z) \end{array}$$

2.5.4 Infix operators. We may introduce infix operators $\otimes$, which form a new syntactic class, so that

$$x\colon A;\ y\colon B \vdash x \otimes y\colon C$$

for some types A, B and C. Note that infix abstractors like $\mapsto$ do not follow this pattern. For such an operator $\otimes$ we introduce the following notations. The first two show how to

turn an infix operator into either a prefix operator or a function on binary tuples. The other two notations are called *sections*[1].

$$\begin{array}{rrcl} & (\otimes)\colon (A \times B \triangleright C) & := & ((x,y) :: x \otimes y) \\ & (\otimes)\colon A \times B \to C & := & \lambda(\otimes) \\ x\colon A \vdash & (x\otimes)\colon (B \triangleright C) & := & (y :: x \otimes y) \\ y\colon B \vdash & (\otimes y)\colon (A \triangleright C) & := & (x :: x \otimes y) \end{array}$$

2.5.5 Repeated abstraction. To introduce a sequence of assumptions and definitions using a single declaration, we introduce a triangle '$\triangleright$' as an infix notation. Informally, if Δ is a (concrete) declaration, and T a type in which the identifiers introduced by Δ may occur, then $(\Delta \triangleright T)$ stands for taking the product of T over all assumptions in Δ. For example, $(x\colon A;\ y\colon B \triangleright C) = \Pi(x\colon A :: \Pi(y\colon B :: C))$. The formal rule is:

$$\begin{array}{rcl} \mathit{Term}_{\Gamma,\gamma}(\mathit{repq}(\Pi,\Delta,T),\mathbf{Type}_i & \longrightarrow & \text{“(”}, \mathit{Decl}_{\Gamma,\gamma}(\Delta,\delta,i), \text{“}\triangleright\text{”}, \\ & & \mathit{Term}_{\Gamma\Delta,\gamma\delta}(T,\mathbf{Type}_i), \text{“)”}. \end{array}$$

The syntactic operation *repq* (repeated quantifier application) is given by the syntactic equations

$$\begin{array}{rcl} \mathit{repq}(Q,\{\},t) & = & t \\ \mathit{repq}(Q,\{v\colon A;\ \Delta\},t) & = & Q(A;\ (v :: \mathit{repq}(Q,\Delta,t))) \end{array}$$

When the declaration consists only of a type, we have $(A \triangleright B) = B^A$.

2.6 Sums and declaration types

2.6.1 Sum types. If A is a type and, for any $x\colon A$, Bx is a type, then the *(generalized) sum* $\Sigma(A;B)$ or $\Sigma(x\colon A :: Bx)$ is a type consisting of all pairs $(a;b)$ where $a\colon A$ and $b\colon Ba$. We introduce the typing rules, and define some auxiliary operations.

$$\begin{array}{rrcl} & A\colon \mathbf{Type};\ B\colon \mathbf{Type}^A & \vdash & \Sigma(A;B)\colon \mathbf{Type} \\ & a\colon A;\ b\colon Ba & \vdash & (a;b)\colon \Sigma(A;B) \\ & T\colon \mathbf{Type}^{\Sigma(A;B)};\ t\colon (x\colon A;\ y\colon B \triangleright T(x;y)) & \vdash & \Sigma\_\mathsf{elim}\, t\colon \Pi(\Sigma(A;B);T) \\ (x\colon A;\ y\colon Bx \vdash t_{xy}\colon T(x;y)\ ) \vdash & ((x;y) :: t_{xy}) & := & \Sigma\_\mathsf{elim}(x;y :: t_{xy}) \\ & \mathsf{fst}\colon (z\colon \Sigma(A;B) \triangleright A) & := & ((x;y) :: x) \\ & \mathsf{snd}\colon (z\colon \Sigma(A;B) \triangleright B(\mathsf{fst}\, z)) & := & ((x;y) :: y) \\ & \sigma_{a:A}\colon Ba \to \Sigma(A;B) & := & y \mapsto (a;y) \\ f_{x:A}\colon Bx \to C \vdash & [f]\colon \Sigma(A;B) \to C & := & (x;y) \mapsto f_x.y \end{array}$$

Note that we have $\lambda\mathsf{fst}\colon \Sigma(A;B) \to A$. For finite sums, like $B_0 + B_1$, see paragraph 2.8.2.

[1] after an idea of Richard Bird

Taking sums preserves coercion, for if

$$\varphi : A \subseteq_{\mathsf{t}} A' \ ;$$
$$\psi(x\colon A) : Bx \subseteq_{\mathsf{t}} B'x$$

then we have a coercion $\Sigma(\varphi;\psi) : \Sigma(A;B) \subseteq_{\mathsf{t}} \Sigma(A';B')$, where:

$$\Sigma(\varphi;\psi) \ := \ (x;y) \mapsto (\varphi.x;\psi x.y) \ .$$

2.6.2 Declaration types. From a declaration Δ we can obtain the type of its instances by means of Σ. An instance of a declaration is a sequence of (correctly typed) values for the typed identifiers in the list. For example, declaration

$$(x\colon A;\ y\colon B_x;\ z\colon C_{xy})$$

has triples $(a;b;c)$ as instances, where $a\colon A$, $b\colon B_a$ and $c\colon C_{ab}$. So the class of all instances forms a type, denoted by writing parentheses and braces around the declaration, thus: $\{\,(\Delta)\,\}$. In this case:

$$\{\,(x\colon A;\ y\colon B_x;\ z\colon C_{xy})\,\} \ = \ \Sigma(x\colon A :: \Sigma(y\colon B_x :: C_{xy}))$$

The following grammar rule describes how a declaration type is to be translated into repeated use of quantifier Σ:

$$\mathit{Term}_{\Gamma,\gamma}(\mathit{repq}(\Sigma,\Delta,1),\mathbf{Type}_i) \quad\longrightarrow\quad \text{“}\{\,(\text{”},\mathit{Decl}_{\Gamma,\gamma}(\Delta,\delta,i),\text{“})\,\}\text{”}.$$

A more elaborate notation for the instances $(a;b;c)$ of a declaration is to write them like definitions, $(x := 1;\ y := b;\ z := c)$. This is very much like Pebble [16] or DEVA [85]. Unfortunately, we cannot formalize this, because our abstract encoding of declaration types (via Σ) disregards the concrete identifier names.

2.6.3 Structure definitions. Elimination on a sum type can be done by pattern matching, as in the definition of fst and snd above. But definitions of classes of structures in mathematics are often given in combination with special elimination constructs. Consider for example the common definition of posets:

> A *poset* X is a structure $(X;\leq_X)$ where X is a set and $\leq_X$ a partial order on X.

This definition indicates (1) that there is a type of posets, (2) that for any set X and partial order $\leq$ on X, $(X;\leq)$ is a poset, and (3) that the underlying set of a poset X is noted as X as well, and that its corresponding partial order is noted as $\leq_X$.

We write such a combined structure definition as follows. (Posets appear again in section 9.1.)

$$\underline{\text{Define}}\ \mathbf{Poset}\colon \mathbf{Type}_1\ \underline{\text{by}}$$
$$X\colon \mathbf{Poset} \quad :=: \quad (X\colon \mathbf{Set};\ (\leq_X)\colon \mathsf{PO}\,X) \ .$$

For another example, see the definition of ‘Fam’ below.

2.7 Families and quantifiers

For T a type, we define a *family of* T to be a tuple $(D; t)$ where D is a type, called the *domain* of the family, and t associates any $d\colon D$ with an element t_d of T. Formally:

$$\underline{\text{Define}}\ \mathsf{Fam}_i(T\colon \mathbf{Type}_{i+1})\colon \mathbf{Type}_{i+1}\ \underline{\text{by}}$$
$$t\colon \mathsf{Fam}_i\, T \quad :=: \quad (\mathsf{Dom}\, t\colon \mathbf{Type}_i;\ t\colon T^{\mathsf{Dom}\, t})$$

The arguments of Π or Σ now turn out to be families of types. I.e., we may write:

$$\Pi, \Sigma\colon (\mathsf{Fam}_i\, \mathbf{Type}_i \triangleright \mathbf{Type}_i)$$

Such constants Q, which are typed by $Q\colon (\mathsf{Fam}_i\, T \triangleright T)$ for some type T and $Nat\, i$, are called *quantifiers*.

In 2.5.1, we introduced a double-colon notation $(x\colon A :: B_x)$ for families. We shall now generalize this to allow an arbitrary declaration Δ instead of the single typing $x\colon A$. Informally, a quantification

$$Q(\Delta :: t)$$

shall stand for the repeated quantifier application $repq(Q, \Delta, t)$. For example, we can write, using the universal quantifier for propositions, and finite and infinite types:

$$\forall(n\colon \mathbb{N};\ k := n^n;\ p\colon \mathbb{N}^k;\ i\colon k :: P(n, p, i)\ )$$

The formal rule looks like

$$TTerm_{\Gamma,\gamma}(repq(Q, \Delta, t), T) \quad \longrightarrow \quad TTerm_{\Gamma,\gamma}(Q, T^{\mathsf{Fam}_i\, T}),$$
$$\text{“(”}, Decl_{\Gamma,\gamma}(\Delta, \delta, i), \text{“::”}, Term_{\Gamma\Delta,\gamma\delta}(t, T), \text{“)”}.$$

except that Q must be a single identifier, and that the concrete type expression $T^{\mathsf{Fam}_i\, T}$ should be replaced by the abstract tree expression that it denotes.

The notation $(\Delta :: t)$ may be used too for families that are not argument of a quantifier Q, using the pseudo-definition

$$(\Delta_{\bar{x}} :: t_{\bar{x}}\ ) \ :=\ (\ \{\ (\Delta_{\bar{x}})\ \};\ (\bar{x} :: t_{\bar{x}})\ )\colon\ \mathsf{Fam}\, T$$

A derived notation for finite families is given in paragraph 2.8.2.

2.8 Finite types

2.8.1 Naturals as types. For $Nat\, n$, a finite type with n elements is noted just ‘n’ (in decimal notation), and its elements are named 0 till $n-1$. And if T is an n-tuple of types, $T\colon \mathbf{Type}^n$, so $\Pi(n; T)$ is the finite product of all Ti, then we note elements of this product as ‘$(t_0, \ldots, t_{n-1})'$, where $t_i\colon Ti$, and the parentheses are optional.

$$\begin{aligned} &\vdash\ n\colon \mathbf{Type} \quad \text{for } Nat\, n \\ &\vdash\ k\colon n \quad \text{for } Nat\, k,\ k < n \\ T\colon \mathbf{Type}^n;\ t_0\colon T(0);\ \ldots;\ t_{n-1}\colon T(n-1) \ &\vdash\ (t_0, \ldots, t_{n-1})\colon \Pi(n; T) \end{aligned}$$

2.8.2 Finite families, products, and sums. We note a family whose domain is a finite type n by using (overloaded) angle brackets. This gives an elegant notation for finite products and sums as special cases of generalized products and sums:

$$\begin{aligned} \langle t_0, \ldots, t_{n-1} \rangle &:= (n; (t_0, \ldots, t_{n-1})) \\ B_0 \times \cdots \times B_{n-1} &:= \Pi \langle B_0, \ldots, B_{n-1} \rangle \\ B_0 + \cdots + B_{n-1} &:= \Sigma \langle B_0, \ldots, B_{n-1} \rangle \end{aligned}$$

This 'overloaded' use of '$\langle \ldots \rangle$' bears no relationship to the notation '$\langle f \rangle$' in paragraph 2.5.3. The latter notation says that if $f_i \colon A \to B_i$ $(i = 0, 1)$ then $\langle f_0, f_1 \rangle \colon A \to B_0 \times B_1$. We have furthermore $\pi_i \colon B_0 \times B_1 \to B_i$ and $\sigma_i \colon B_i \to B_0 + B_1$.

We find our definition $B_0 \times B_1 := \Pi \langle B_0, B_1 \rangle$ far more elegant than the more common one in type theory, $B_0 \times' B_1 := \Sigma(x \colon B_0 :: B_1)$. Ours has the advantage that notations and operations on generalized products apply directly to finite products, so that, e.g., B^2 is synonymous with $B \times B$.

2.8.3 Enumerations. For any enumerated list of distinct *labels* ℓ_i, we introduce a finite type $\{\ell_0, \ldots, \ell_{n-1}\}$ whose elements are ℓ_i. Let L abbreviate this type, then L is isomorphic with type n. We define a finite type of booleans as an example.

$$\begin{aligned} &\vdash && \ell_i \colon L \colon \mathbf{Type} \\ T \colon \mathbf{Type}^L;\ t_0 \colon T(\ell_0);\ \ldots;\ t_{n-1} \colon T(\ell_{n-1}) &\vdash && (\ell_0 :: t_0 \mid \cdots \mid \ell_{n-1} :: t_{n-1}) \colon \Pi(L; T) \\ \mathsf{Bool} &:= && \{\mathsf{true}, \mathsf{false}\} \\ \underline{\mathsf{if}}\ b\ \underline{\mathsf{then}}\ p\ \underline{\mathsf{else}}\ q &:= && b^{\backslash}(\mathsf{true} :: p \mid \mathsf{false} :: q) \end{aligned}$$

(We generally use the symbol '|' to join alternatives. Its shape gives a good separation.)

2.8.4 Labeled sums. Sum types whose domain is an enumeration may be defined by grammar rules in Backus-Naur form (BNF), using '::=', which is not to be confused with the BNF definitions used to define the abstract classes of *ADAM*. So the following two definitions are equivalent, where L is $\{\ell_0, \ldots, \ell_{n-1}\}$ and the Δ_i are declarations:

$$\begin{aligned} S &::= \ell_0(\Delta_0) \mid \cdots \mid \ell_{n-1}(\Delta_{n-1}) \\ S &:= \Sigma(L;\ (\ell_0 :: \{\,(\Delta_0)\,\} \mid \cdots \mid \ell_{n-1} :: \{\,(\Delta_{n-1})\,\})\,) \end{aligned}$$

The elements $(\ell_i; \delta)$ of S, where $\delta \colon \{\,(\Delta_i)\,\}$, may be noted $\ell_i(\delta)$, and we can use a case analysis notation like

$$(\ell_0(x_0) :: t_{0x_0} \mid \cdots \mid \ell_{n-1}(x_{n-1}) :: t_{n-1x_{n-1}}) \colon \Pi(S; T)$$

2.8.5 Case analysis in definitions. The notation for finite tuples in 2.8.1 and 2.8.3 is a form of case analysis. We allow a more general form of mapping that uses arbitrary (exhaustive and exclusive) patterns, as yet informally. Definitions may use case analysis using a notation like

$$\underline{\mathsf{Define}}\ \mathsf{c}(z \colon A + B) \colon C_z\ \underline{\mathsf{by}}\ \mathsf{c}(0; x) := c_x \mid \mathsf{c}(1; y) := c'_y\ .$$

This corresponds to $\mathsf{c} \colon (z \colon A + B \triangleright C_z) := \Sigma\_\mathsf{elim}((x :: c_x), (y :: c'_y))$.

2.9 Infinite types

Infinite types are to be characterized by induction principles, and these form the subject matter of this thesis. To prove the existence of inductive types, we need to assume the existence of one infinite type:

2.9.1 Naturals. The syntax class *Nat* of naturals is not itself a type. We assume a type named $\mathbb{N}$, with a dependent recursion rule as is usual in type theory. Its elements may be given in decimal notation, so for $Nat\, n$ we have $n\colon \mathbb{N}$. Conversely, we identify any $n\colon \mathbb{N}$ with a finite type, so $\mathbb{N} \subseteq_{\mathsf{t}} \mathbf{Type}$, where we skip details. We use ω as a synonym for $\mathbb{N}$, especially to index infinite tuples. So $A^\omega := (\mathbb{N} \triangleright A)$.

$$\begin{aligned} & \vdash && \mathbb{N}\colon \mathbf{Type} && (2.1)\\ & \vdash && 0\colon \mathbb{N};\ \ \mathsf{s}\colon (\mathbb{N} \triangleright \mathbb{N}) && (2.2)\\ T\colon \mathbf{Type}^{\mathbb{N}};\ b\colon T(0);\ t(n\colon \mathbb{N};\ h\colon Tn)\colon T(\mathsf{s}\, n) & \vdash && \mathbb{N}\_\mathsf{rec}(b,t)\colon \Pi(\mathbb{N}; T) && (2.3)\\ & \vdash && \mathbb{N} \subseteq_{\mathsf{t}} \mathbf{Type} && (2.4)\\ \omega & := && \mathbb{N} && (2.5) \end{aligned}$$

2.9.2 Finite sequences. Now we can form, for any type A, the type $A^* = \Sigma(n\colon \mathbb{N} :: A^n)$ of finite sequences of A. The length of a sequence s is noted $\#s$. Our definition is similar to the one for families in 2.7:

$$A\colon \mathbf{Type} \vdash \quad \underline{\mathsf{Define}}\ A^*\colon \mathbf{Type}\ \underline{\mathsf{by}}\ s\colon A^* :=: (\#s\colon \mathbb{N};\ s\colon A^{\#s})$$

The angle-bracket notation for finite families of paragraph 2.8.2 can be used for finite sequences in A^* as well, i.e. $\langle t_0, \ldots, t_{n-1}\rangle\colon A^*$, and we have $A^* \subseteq_{\mathsf{t}} \mathsf{Fam}\, A$.

2.10 Equality predicate

We use a primitive equality predicate on objects of any type. It yields for any type A and objects $u, v\colon A$, a type $(u =_A v)$ that is inhabited just when u equals v. This type is actually a proposition in **Prop**, which we introduce in section 2.11. The inhabitant for trivial equalities, including reflexivity, is denoted by 'eq'.

$$\begin{aligned} A\colon \mathbf{Type};\ u, v\colon A & \vdash && (u =_A v)\colon \mathbf{Prop}\\ a\colon A & \vdash && \mathsf{eq}\colon (a =_A a)\\ p\colon (A =_{\mathbf{Type}} B) & \vdash && A =_{\mathsf{t}} B \end{aligned}$$

This last rule implies that, if in some context there is an expression $p\colon (A =_{\mathbf{Type}} B)$, and $a\colon A$, then one has $a\colon B$ as well. This is called *type conversion*. Existence of such an expression p is not effectively decidable, so a correctness checker may have to reject expressions $a\colon B$ when it is not evident how to find p.

The equality predicate (or equality type) might also be defined by the Leibniz equality, setting $(u =_T v)$ equal to

$$\forall(P\colon \mathbf{Prop}^T :: Pu \Rightarrow Pv)\ ,$$

Replacing '$\Rightarrow$' by '$\Leftrightarrow$' would be equivalent. One still needs a rule for type conversion, and one for extensionality of (infinite) tuples.

2.10.1 Uniqueness. For any type A, we define $!A$ to be the type containing the unique element of A, if one exists; otherwise $!A$ will be empty:

$$\underline{\text{Define}}\ !(A\colon \mathbf{Type})\ \underline{\text{by}}$$
$$a\colon !A \quad :=: \quad (a\colon A;\ \mathsf{uniq}\, a\colon (x\colon A \triangleright a = x))\ .$$

That is to say, any element a of $!A$ consists of an element of A that is noted a as well, together with a proof, noted $\mathsf{uniq}\, a$, that any $x\colon A$ equals a. We have $!A \subseteq_{\mathsf{t}} A$.

2.11 The type of propositions

ITT fully identifies prositions with types, while CC contains a special type **Prop** to represent propositions. This is not only useful for making a conceptual distinction between types and propositions, but also necessary for constructing types that represent the class of all subsets of some type, while staying in the same universe, by defining $\mathcal{P}(A)\colon \mathbf{Type}_i$ as $\mathbf{Prop}^A$, for any $A\colon \mathbf{Type}_i$.

Thus, **Prop** must be a member of any universe of the hierarchy. It is construed *a priori* as being a type of types whose members have at most one element, and are members of any other universe. As the product of any number of propositions still has at most one element, it is again a proposition, the universal quantification noted '$\forall(A; P)$'. We introduce **Prop** here in *ADAM*.

$$\begin{aligned}
&\vdash && \mathbf{Prop}\colon \mathbf{Type}_i \\
&\vdash && \mathbf{Prop} \subseteq_{\mathsf{t}} \mathbf{Type}_i \\
P\colon \mathbf{Prop};\ p, q\colon P &\vdash && \mathsf{eq}\colon p =_P q \\
A\colon \mathbf{Type}_i;\ P\colon \mathbf{Prop}^A &\vdash && \forall(A; P)\colon \mathbf{Prop} \\
p\colon \forall(A; P);\ a\colon A &\vdash && pa\colon Pa \\
(x\colon A \vdash p_x\colon Px) &\vdash && (x :: p_x)\colon \forall(A; P)
\end{aligned}$$

From the universal quantifier we derive all other propositional operators and quantifiers. The existential quantifier is defined here as an operator '$\exists A$', meaning "type A is inhabited". We give it a subscript $_{\mathsf{w}}$ now, as it is soon to be replaced.

The product $\forall(A; P)$ is usually written as $\forall(x\colon A :: P_x)$. This and other quantifiers and connectives are defined below. Note that $\exists_{\mathsf{w}}$ is defined as an operator that turns a type into a proposition; existential quantification over a family of propositions is derived from this.

$$\begin{aligned}
P \Rightarrow Q &:= \forall_{_}\colon P :: Q \\
P_0 \wedge \cdots \wedge P_{n-1} &:= \forall\langle P_0, \ldots, P_{n-1}\rangle \\
\mathsf{True} &:= \forall\langle\,\rangle \\
P \Leftrightarrow Q &:= (P \Rightarrow Q) \wedge (Q \Rightarrow P)
\end{aligned}$$

$$\begin{array}{rcl}
\exists_{\mathsf{w}}(A\colon \mathbf{Type}) & := & \forall(X\colon \mathbf{Prop};\, \forall(x\colon A :: X) :: X) \\
\exists_{\mathsf{w}}(P\colon \mathsf{Fam}\,\mathbf{Prop}) & := & \exists_{\mathsf{w}}\{\,(x\colon \mathsf{Dom}\,P;\; Px)\,\} \\
P_0 \vee \cdots \vee P_{n-1} & := & \exists_{\mathsf{w}}\langle P_0, \ldots, P_{n-1}\rangle \\
\mathsf{False} & := & \exists_{\mathsf{w}}\langle\,\rangle \\
\neg P & := & P \Rightarrow \mathsf{False} \\
P, Q\colon \mathbf{Prop} & \vdash & \mathsf{eq}\colon (P =_{\mathbf{Prop}} Q) \;=\; (P \Leftrightarrow Q)
\end{array}$$

In addition, one may need the *axiom of choice*. This axiom states that, given a family of nonempty types, there exists a tuple picking an inhabitant of each type from the family.

$$B\colon \mathbf{Type}^A;\; p\colon \forall(x\colon A :: \exists_{\mathsf{w}} Bx) \;\vdash\; \mathsf{ac}\,p\colon \exists_{\mathsf{w}}\,\Pi(A; B)$$

An existential assumption $p\colon \exists_{\mathsf{w}} A$ may be called weak because it can only be used for proving other propositions. As explained in appendix C, for some purposes we need a stronger notion of existential quantification. On other occasions we want to do classical reasoning. In a constructive calculus, these should not be combined in a single type, for this strong quantifier destroys the constructivity of terms of all types as soon as proofs of propositions need not be constructive. For this reason, we introduce two alternative versions of **Prop**, which we name $\mathbf{Prop}_{\mathsf{c}}$ and $\mathbf{Prop}_{\mathsf{i}}$ for classical and constructive (though not really intuitionistic, for intuitionistic philosophy does not admit impredicative quantification) propositions. Thus, we have two variants of *ADAM*. Combining them in a single language might be desirable, but requires a more careful distinction between classical and constructive propositional operators.

Except when stated otherwise, we use constructive logic and omit the subscript i.

2.11.1 Constructive propositions. As expressions should be equivalent to constructive object definitions, we should have an operator ι (iota) that, given a constructive proof that some type has a unique element, denotes that element. Consequently, the information contained in a proof becomes relevant for computing the object denoted by an expression, and one should not introduce representations in the abstract (kernel) language in which this proof information is removed.

Rather than adding primitive rules for iota, appendix C proposes to introduce a new existential quantifier with a stronger elimination rule. The idea is that the proposition $\exists T$ should be equivalent to the quotient type of T modulo the equivalence relation that identifies everything. Thus $\exists T$ contains at most one equivalence class.

The type $\mathbf{Prop}_{\mathsf{i}}$ of constructive propositions has rules as stated above for **Prop** and this new existential quantifier.

$$\begin{array}{rcl}
A\colon \mathbf{Type} & \vdash & \exists A\colon \mathbf{Prop}_{\mathsf{i}} \\
 & \vdash & \exists_\mathsf{in}\colon A \to \exists A \\
T\colon \mathbf{Type}^{\exists A}; & & \\
t\colon \Pi(x\colon A :: T(\exists_\mathsf{in}\,x)); & & \\
d\colon \forall x, y\colon A :: tx = ty & \vdash & \exists_\mathsf{elim}\,t\colon \Pi(\exists A;\; T)
\end{array}$$

Here we let the types A and T and proof d be hidden in the concrete notation $\exists\_\mathsf{elim}\,t$ because we are not so much concerned with proof objects. (In the appendix, rule C.3,

we chose to show d explicitly.) Using $\exists\_\mathsf{elim}$ we can define iota, as follows:

$$\begin{aligned}\iota_T\colon (\exists!T \triangleright T) \quad &:= \quad \exists\_\mathsf{elim}((u;p)\colon !T :: u)\\ &\qquad\quad \text{noting } (u;p),(v;q)\colon !T \vdash pv\colon (u=v)\end{aligned}$$

When **Prop** in the definition of $\exists_{\mathsf{w}}A$ is read as $\mathbf{Prop}_{\mathsf{i}}$, then $\exists_{\mathsf{w}}A$ and $\exists A$ become equivalent. I.e., one has

$$\exists A \;\Leftrightarrow\; \forall(X\colon \mathbf{Prop}_{\mathsf{i}};\, \forall(x\colon A :: X) :: X)$$

2.11.2 Classical propositions. For classical logic, one just adds a *reductio ad absurdum* rule:

$$P\colon \mathbf{Prop}_{\mathsf{c}} \quad\vdash\quad \mathsf{raa}\colon (\neg\neg P \Rightarrow P)$$

(Adding $\exists\_\mathsf{elim}$ for $\mathbf{Prop}_{\mathsf{c}}$ is possible but destroys the constructive nature of object terms.)

2.12 More derived notions

2.12.1 Predicates

A predicate P on a type T is obviously an operator $P\colon (T \triangleright \mathbf{Prop})$, and if $\prec$ is an infix relation symbol, say

$$x\colon A;\; y\colon B \vdash x \prec y\colon \mathbf{Prop}\ ,$$

then we have $(\prec)\colon (A \times B \triangleright \mathbf{Prop})$. By "sectioning" (paragraph 2.5.4), $(\prec b)$ stands for the predicate $(x :: x \prec b)$.

In a declaration, we may write $(x\colon \prec b)$ for $(x\colon A;\; x \prec b)$.

2.12.2 Subtypes

If A is a type and P a predicate over A, then $\{\, x\colon A \mid\colon Px\}$ is a *subtype* of A which is isomorphic to $\Sigma(A;P)$, but for which we allow a more convenient notation for its elements. Symbol '|:' may be read as 'such that'. We can pseudo define it by:

$$\begin{aligned}&\underline{\mathsf{Define}}\ \{\, x\colon A \mid\colon P_x\}\colon \mathbf{Type}\ \underline{\mathsf{by}}\\ &\qquad\qquad x\colon \{\, x\colon A \mid\colon P_x\} \quad :=: \quad (x\colon A;\ \mathsf{prop}\, x\colon P_x)\ .\end{aligned}$$

One may use a pattern instead of the single variable x. So we have, if $a\colon \{\, x\colon A \mid\colon P_x\}$, then $a\colon A$ and $\mathsf{prop}\, a\colon P_a$. Furthermore, if for some $a\colon A$ there is an evident $p\colon P_a$, one may write just $a\colon \{\, x\colon A \mid\colon P_x\}$.

Here a problem appears: how should a subtype element a be noted when the corresponding proof p is not evident? Writing '$(a;p)$' is rather confusing. We have thought about '$(a|;p)$', where '|;' should be read as "because of", but leave it to informal notation, for now.

Note that subtypes admit only the declaration of variables of that type. To quantify over all subsets of a type one must use the subset type $\mathcal{P}T$ below, which is derived from **Prop**. A membership test for subtypes does not make much sense, because $a \in \{\, x\colon A \mid\colon P_x\}$ can always be replaced by P_a.

2.12.3 Subsets

The type of *subsets* $S{:}\,\mathcal{P}(A)$ is isomorphic to the type of predicates on A, but one writes $a \in S$ for the membership test rather than $S(a)$. We shall write $|P|$ for the subset that corresponds to predicate $P{:}\,\mathbf{Prop}^A$, by making the following definitions.

$$\begin{array}{rrcl}
 & \mathcal{P}(A{:}\,\mathbf{Type}_i){:}\,\mathbf{Type}_i & ::= & |\,(P{:}\,\mathbf{Prop}^A)\,|\ . \\
 & S{:}\,\mathcal{P}A;\ a{:}\,A & \vdash & \underline{\mathsf{Define}}\ a \in S:\ \ \mathbf{Prop}\ \underline{\mathsf{by}} \\
 & & & a \in |P| \ :=\ Pa \\
(x{:}\,A \vdash P_x{:}\,\mathbf{Prop}) \vdash & \{\,x \mid{:}\, P_x\}{:}\,\mathcal{P}A & := & |x :: P_x| \\
 & t{:}\,\mathsf{Fam}\,A \vdash \quad \{t\}{:}\,\mathcal{P}A & := & \{\,x \mid{:}\, \exists d{:}\,\mathsf{Dom}\,t :: x = t_d\} \\
 & R, S{:}\,\mathcal{P}A \vdash \quad R \subseteq S & := & \forall(x{:}\,A :: x \in R \Rightarrow x \in S)
\end{array}$$

This may be completed with all usual set operations, like the empty set $\emptyset$, binary operators $\cup$ and $\cap$, quantifiers $\bigcup$ and $\bigcap$ (intersection within A), and $\overline{S}$ (complement in A).

The third definition above introduces the usual suggestive notation for set comprehension, except that we write '$|{:}$' for "such that", just as with subtypes. The fourth definition introduces a notation similar to the replacement axiom of set theory. For example, $\{x{:}\,D :: t_x\,\}$ stands for the subset that contains t_x for any $x{:}\,D$.

The next definition tells how to interpret subsets as types themselves, that is, we have $\mathcal{P}T \subseteq_{\mathsf{t}} \mathbf{Type}$. Furthermore, a type may stand for its full subset. Finally, we shall sometimes omit the bars around a predicate P.

$$\begin{array}{rrcl}
T{:}\,\mathbf{Type} \vdash \quad S \mapsto \{\,x{:}\,T \mid{:}\, x \in S\} & : & \mathcal{P}T \subseteq_{\mathsf{t}} \mathbf{Type} \\
T{:}\,\mathbf{Type} \vdash \quad T{:}\,\mathcal{P}T & := & \{\,x \mid{:}\, \mathsf{True}\} \\
T{:}\,\mathbf{Type} \vdash \quad P \mapsto |P| & : & \mathbf{Prop}^T \subseteq_{\mathsf{t}} \mathcal{P}T
\end{array}$$

The declaration $X{:}\subseteq T$ now stands for $X{:}\,\mathcal{P}T;\ \forall(x{:}\,T :: x \in X \Rightarrow \mathsf{True})$, which we read as just $X{:}\,\mathcal{P}T$.

2.12.4 Relational notations

A (binary) relation between two types A and B, notation $R{:}\,A \sim B$, should obviously be a subset $R{:}\,\mathcal{P}(A \times B)$. One easily defines converse, left and right domain, and (forward) composition of relations:

$$\begin{array}{rrcl}
A, B{:}\,\mathbf{Type}_i \vdash \quad A \sim B{:}\,\mathbf{Type}_i & := & \mathcal{P}(A \times B) \\
R{:}\,A \sim B \vdash \quad R^{\cup}{:}\,B \sim A & := & \{\,(y,x) \mid{:}\, (x,y) \in R\}\ , \\
 & R^{<}{:}\,\mathcal{P}A & := & \{\,x \mid{:}\, \exists y :: (x,y) \in R\}\ , \\
 & R^{>}{:}\,\mathcal{P}B & := & \{\,y \mid{:}\, \exists x :: (x,y) \in R\} \\
R{:}\,A \sim B,\ S{:}\,B \sim C \vdash \quad R \cdot S{:}\,A \sim C & := & \{\,(x,z) \mid{:}\, \exists y{:}\,B :: (x,y) \in R \wedge (y,z) \in S\}
\end{array}$$

This composition is associative and has identity relations $|{=}_A|$.

Functions $f\colon A \to B$ are sometimes identified with their relational graph, and we define the relational image of a set:

$$\begin{array}{rcl} f \mapsto \{x\colon A :: (x, f.x)\} & : & (A \to B) \subseteq_{\mathsf{t}} (A \sim B) \\ R\colon A \sim B;\ X\colon \mathcal{P}A \vdash \quad R[X] & := & \{\, y\colon B \mid\colon \exists x\colon \in X :: (x,y) \in R\} \\ R\colon A \sim B;\ x\colon A \vdash \quad R[x] & := & R[\{x\}] \end{array}$$

So one has, e.g., for $X\colon \mathcal{P}A$, $Y\colon \mathcal{P}B$; $f\colon A \to B$, $g\colon B \to C$, $h\colon C \to D$; $R\colon B \sim C$, the following equalities:

$$\begin{array}{rcl} f[X] & = & \{x\colon \in X :: f.x\}\colon\ \mathcal{P}B \\ f^{\cup}[Y] & = & \{\, x \mid\colon f.x \in Y\}\colon\ \mathcal{P}A \\ f \cdot g & = & f \mathbin{\bar{\circ}} g\colon\ A \sim C \\ f \cdot R & = & \{\, (x,z) \mid\colon (f.x, z) \in R\}\colon\ A \sim C \\ R \cdot h & = & \{(y,z)\colon \in R :: (y, h.z)\}\colon\ B \sim D \end{array}$$

In appendix D we will see how type constructors extend to relation constructors. The most important of these we give here for general use. For $R\colon A \sim A'$, $S\colon B \sim B'$, we define $R \to S$ to be the set of all pairs of functions that map related arguments to related results:

$$R \to S\colon (A \to A') \sim (B \to B') \ :=\ \{\, (f, f') \mid\colon \forall (x, x')\colon \in R :: (fx, f'x') \in S\}$$

To obtain a function $f\colon A \to B$ given its graph $R\colon A \sim B$ with a proof that R is single-valued,

$$p\colon \forall x\colon A :: \exists! R[x]\ ,$$

one has to use the iota operator (subsection 2.11.1):

$$f.x \ :=\ \iota_{R[x]}(px)\ . \tag{2.6}$$

2.12.5 Currying

We will sometimes consider $(z\colon \{\, (x\colon A; y\colon Bx)\,\} \triangleright Tz)$ to be equivalent to

$$(x\colon A;\ y\colon Bx \triangleright T(x;y))\ .$$

So tab is equivalent to $t(a;b)$ and we can write ta for $(y :: t(a;y))$.

Taking $A := 2$, this convention amounts to $(z\colon B + B' \triangleright Tz)$ being equivalent to

$$(y\colon B \triangleright T(0;y)) \times (y\colon B' \triangleright T(1;y))\ .$$

Thus, if we have $t\colon C^B$ and $t'\colon C^{B'}$, we can write $(t,t')\colon C^{B+B'}$ instead of $((x;y) :: (t,t')xy)$.

(*Currying* is the term used in functional programming for using functions that yield functions again, after an idea of H.B. Curry.)

2.12.6 Pattern matching

The notation of paragraph 2.8.3 for case analysis can be generalized to other patterns. We will be less restrictive than in subsection 2.3.2, for we admit any 'suitable' terms with some free variables to be used as a pattern.

Let meta-expressions $\xi\bar{u}\colon A$ stand for patterns containing a sequence of variables $\bar{u}$. Suppose we have a sequence of n patterns $\xi_i u$, where for simplicity we assume that each pattern has only one and the same variable $u\colon U_i$. If one knows that

$$\forall x\colon A :: \exists i\colon n;\ u\colon U_i :: x = \xi_i u \tag{2.7}$$

and a sequence of expressions $q_i u$ typed by

$$u\colon U_i \vdash q_i u\colon B(\xi_i u)$$

such that

$$\forall i, j\colon n;\ u\colon U_i;\ v\colon U_j :: (\xi_i u = \xi_j v \Rightarrow q_i u = q_j v) \tag{2.8}$$

then there is a unique tuple $p\colon \Pi(A; B)$ such that $p(\xi_i u) = q_i u$, which we note as

$$(\xi_0 u :: q_0 u \mid \cdots \mid \xi_{n-1} u :: q_{n-1} u)\ .$$

If the i and u in (2.7) are known to be unique, then (2.8) holds trivially.

A similar notation may be used for other infix abstractors, e.g.

$$\begin{aligned}(\xi_0(u\colon U_i) \triangleright B_0 u \mid \cdots) &:= (x\colon A \triangleright x^{\backslash}(\xi_0 u :: B_0 u \mid \cdots))\\ (\xi_0 u \mapsto q_0 u \mid \cdots) &:= (x \mapsto x^{\backslash}(\xi_0 u :: q_0 u \mid \cdots))\end{aligned}$$

A problem with these notations is that it might not be clear which identifiers are being bound, in case ξ_i contains free variables itself.

2.12.7 Linear proof notation

Statements $s\ R\ t$ that some transitive relation R, such as $=$ or $\Leftrightarrow$, holds between two objects or propositions are often derived through a linear sequence of steps. We may present these proofs in a three-column format:

$$\begin{array}{lll} & s & \\ R & s' & \{\,\langle\text{reason why } s\ R\ s'\rangle\,\}\\ R & t & \{\,\langle\text{reason why } s'\ R\ t\rangle\,\}\end{array}$$

During such a linear proof, we may sometimes give a definition that extends to the following proof lines, and even beyond.

Of course, any proposition P may be derived by a linear proof of $P \Leftarrow \mathsf{True}$.

2.13 Conclusion

We defined an extensive notation system based on Constructive Type Theory, partly using a two-level Van Wijngaarden grammar. Several of the more advanced features, that we consider valuable to obtain natural notational flexibility, appeared to be too complicated to be formally defined within the scope of this thesis. We shall apply the notations in the rest of the thesis to express our definitions and rules.