Appendix A

Set theory

This appendix contains some basic notions of set theory, and two models of set theory within type theory. First we list the axioms of Zermelo-Fraenkel set theory with choice (ZFC). Then we present two well-known classes of infinite numbers, namely ordinals and cardinals. (Here we used the outline by Manes [54, pp. 71–72], who cites Monk [63].)

In type theory one can define a big type in $\mathbf{Type}_1$ that gives a model of ZFC. We present two models in A.5 and A.6; the latter one uses an inductive type with equations.

Finally, non-wellfounded sets are described in A.7.

A.1 ZFC axioms

We introduce **Set** as a primitive sort, together with a binary predicate *membership*, $(\in)\colon\subseteq \mathbf{Set}^2$, and abbreviate

$$x \subseteq y \;:=\; \forall(z\colon\in x :: z \in y)\ .$$

Please do not get confused by the overloaded use of '$\in$', '$\subseteq$' and other set operations: they are defined for subset types, for families, and for models of ZFC as well. The first interpretation will not be used in this appendix.

The axiom of *extensionality* is about equality of sets:

$$s,t\colon\mathbf{Set} \;\vdash\; \forall(x :: x \in s \Leftrightarrow x \in t) \Rightarrow s = t \tag{A.1}$$

There are five axioms stating the existence of primitive sets, each accompanied with axioms describing the members of these sets.

Separation: If $P(x)$ is a propositional formula with parameter x:

$$\begin{aligned} P\colon\mathbf{Prop}^{\mathbf{Set}}, s\colon\mathbf{Set} \;\vdash\;& \{\,x\colon\in s \mid\colon P(x)\}\colon\mathbf{Set};\\ & x \in \{\,x\colon\in s \mid\colon P(x)\} \;\Leftrightarrow\; x \in s \wedge Px \end{aligned} \tag{A.2}$$

Just as in *ADAM*, we use the symbol '$\mid\colon$', read "such that", instead of the more conventional '$\mid$' or '$\colon$' because we find the latter two too symmetric and want to use them for other purposes. Note also that '$\colon\in$' is used for introducing a variable that ranges over a set.

Union, *Power set*, and *Infinity*:

$$\begin{aligned} s{:}\,\mathbf{Set} \;\vdash\; & \textstyle\bigcup s{:}\,\mathbf{Set}; \\ & x \in \textstyle\bigcup s \;\Leftrightarrow\; \exists(y{:}\in s :: x \in y) && \text{(A.3)} \\ s{:}\,\mathbf{Set} \;\vdash\; & \mathcal{P}(s){:}\,\mathbf{Set}; \\ & x \in \mathcal{P}(s) \;\Leftrightarrow\; x \subseteq s && \text{(A.4)} \\ \vdash\; & \omega{:}\,\mathbf{Set}; \\ & \exists(y{:}\in \omega :: \forall z :: z \notin y) \wedge \forall(y{:}\in \omega :: y \cup \{y\} \in \omega) && \text{(A.5)} \end{aligned}$$

where $y \cup \{y\} \in \omega$ abbreviates the formula $\exists z{:}\in \omega :: \forall u :: u \in z \Leftrightarrow u \in y \vee u = y$.

Replacement. If F is a unary operation (that may be given as a predicate):

$$\begin{aligned} s{:}\,\mathbf{Set}, F{:}\,\mathbf{Set}^{\mathbf{Set}} \;\vdash\; & \{x{:}\in s :: Fx\}{:}\,\mathbf{Set}; \\ & y \in \{x{:}\in s :: Fx\} \;\Leftrightarrow\; \exists(x{:}\in s :: y = Fx) && \text{(A.6)} \end{aligned}$$

Many more operations on sets may be derived, for example:

$$\begin{aligned} \emptyset{:}\,\mathbf{Set} &:= \{x{:}\in \omega \mid{:}\, \mathsf{False}\} \\ x{:}\,\mathbf{Set} \vdash \{x\} &:= \{z{:}\in \omega :: x\} \\ x, y{:}\,\mathbf{Set} \vdash \{x, y\} &:= \{z{:}\in \omega :: Fz\} \\ & \quad \underline{\mathsf{where}}\ Fz := \begin{cases} x & \text{if } z = \emptyset \\ y & \text{if } z \neq \emptyset \end{cases} \\ x \cup y &:= \textstyle\bigcup\{x, y\} \\ x \cap y &:= \{z{:}\in x \mid{:}\, z \in y\} \end{aligned}$$

The axiom of *foundation* or *regularity* says that the membership relation $\in$ is well-founded. We formulate it in a constructive form:

$$P{:}\,\mathbf{Prop}^{\mathbf{Set}} \;\vdash\; \forall(x :: \forall(y{:}\in x :: Py) \Rightarrow Px) \Rightarrow \forall(x :: Px) \qquad \text{(A.7)}$$

The *axiom of choice* says that, given a mapping to nonempty sets, there exists a function picking one element of each set. (We use the encodings for functions given below.)

$$s{:}\,\mathbf{Set} \;\vdash\; (\forall x{:}\in \mathsf{dom}\, s :: s(x) \neq \emptyset) \;\Rightarrow\; (\exists f :: \forall x{:}\in \mathsf{dom}\, s :: f(x) \in s(x)) \qquad \text{(A.8)}$$

A.2 Set encodings

We may use the following standard encodings of pairs, functions, and naturals, using only the above axioms and operations.

$$\begin{aligned} \langle x, y\rangle &:= \{\{x\}, \{x, y\}\} \\ \mathsf{fst}\, p &:= \textstyle\bigcup\{x{:}\in \bigcup p \mid{:}\, \exists y{:}\,\mathbf{Set} :: p = \langle x, y\rangle\} \\ \mathsf{snd}\, p &:= \textstyle\bigcup\{y{:}\in \bigcup p \mid{:}\, \exists x{:}\,\mathbf{Set} :: p = \langle x, y\rangle\} \end{aligned}$$

$$
\begin{array}{rcl}
X \times Y & := & \bigcup \{ x : \in X :: \{ y : \in Y :: \langle x, y \rangle \} \} \\
\mathsf{dom}\, f & := & \{ p : \in f :: \mathsf{fst}\, p \} \\
\mathsf{cod}\, f & := & \{ p : \in f :: \mathsf{snd}\, p \} \\
Y^X & := & \{ f : \mathcal{P}(X \times Y) \mid : \forall x : \in X :: \exists ! y : \mathbf{Set} :: \langle x, y \rangle \in f \} \\
f(x) & := & \bigcup \{ y : \in \mathsf{cod}\, f \mid : \langle x, y \rangle \in f \} \\
0 & := & \emptyset \\
n+1 & := & n \cup \{n\}
\end{array}
$$

A.3 Ordinals

Ordinals are special sets, but the class **Ord** of all ordinals is too big to be a set itself. We give two definitions of this class.

A.3.1 Inductive definition of ordinals. The class **Ord** is the least class (i.e. the intersection of all classes) $X : \subseteq \mathbf{Set}$ such that:

1. For any x in X, its successor $x \cup \{x\}$ is in X;
2. The union of any *set* of X-members is in X.

Note that, as the empty set $\emptyset$ is the union of the empty set of ordinals, it is an ordinal by clause 2. It is named 0 as well.

This definition gives us a principle of transfinite induction: any predicate on sets that is closed under the clauses above holds for all ordinals. Unfortunately, it is a second order definition that cannot be given in first order logic. However, the following one is equivalent [63]:

A.3.2 First order definition of ordinals. A set (of sets) x is $\in$-*transitive* iff whenever $y \in x$ and $z \in y$ then $z \in x$. An *ordinal* is an $\in$-transitive set x such that all $y \in x$ are also $\in$-transitive.

If x, y are ordinals then (using the axiom of foundation) exactly one of $x \in y$, $x = y$, $y \in x$ occurs (classically), so that **Ord** is linearly ordered via:

$$x \leq y \quad := \quad x = y \vee x \in y$$

If X is a nonempty set (or class) of ordinals then $\bigcap X$ is an ordinal and is in X; in particular, X has a least element. Further, for ordinals x, y, $x \leq y$ holds iff $x \subseteq y$.

A.4 Cardinals

A *cardinal* is an ordinal which is not equipotent (*equipotent* means "in bijective correspondence") with a smaller ordinal. The class of cardinals is noted '**Card**'.

Given any set A there exists (classically) a unique cardinal $\mathsf{card}(A)$ that is equipotent with A. So cardinals are useful for measuring the size of sets.

If x is a cardinal, x^+ denotes the next largest cardinal. There is no largest cardinal, that is, x^+ always exists. A cardinal x is *regular* iff x is infinite and for every family $y{:}\,\mathsf{Fam}\,\mathbf{Card}$ with each $y_i < x$ and $\mathsf{card}(\mathsf{Dom}\,y) < x$, it is the case that $\mathsf{card}\,\Sigma y < x$. The first infinite cardinal, ω, is regular, and for any infinite cardinal x, x^+ is regular.

A.5 A model of ZFC

Within type theory, one may represent a set together with all its element sets by a directed graph $(N{:}\,\mathbf{Type}_0;\ S{:}\subseteq N^2)$ together with a designated root $n{:}\,N$.

Given graph $(N;S)$, a node $x{:}\,N$ corresponds to the set of those sets that correspond to the nodes in $S[x] = \{\, y{:}\,N \mid{:}\ (x,y) \in S\}$. A partial interpretation as sets of such triples $(N;S;n)$ is recursively specified by:

$$[\![N;S;n]\!] \;=\; \{m{:}\in S[n] :: [\![N;S;m]\!]\}\ .$$

We define the type T of directed rooted graphs, followed by an inductive definition of a partial equivalence relation $\equiv$ on T. Only triples where the root starts a wellfounded tree appear in $\equiv$.

$$\begin{array}{rcl}
T{:}\,\mathbf{Type}_1 & := & \{\,(N{:}\,\mathbf{Type}_0;\ S{:}\subseteq N^2;\ n{:}\,N)\,\} \\
R{:}\,\mathcal{P}T^2 & \vdash & \underline{\mathsf{Define}}\ (\preceq_R){:}\,\mathcal{P}T^2\ \underline{\mathsf{by}} \\
(N;S;n) \preceq_R (N';S';n') & := & \forall x{:}\in S[n] :: \exists x'{:}\in S'[n'] :: ((N;S;x),(N';S';x')) \in R \\
(\equiv){:}\,\mathcal{P}T^2 & := & \bigcap (R{:}\,\mathcal{P}T^2 \mid{:}\ (\preceq_R) \cap (\succeq_R) \subseteq R)
\end{array}$$

Now $\mathsf{ZFC}{:}\subseteq \mathcal{P}T$ will be the subtype of all $\equiv$-equivalence classes. (For non-wellfounded sets, see A.7.)

$$\begin{array}{rcl}
\mathsf{ZFC}{:}\subseteq \mathcal{P}T & := & \{t{:}\,T;\ t \equiv t :: |t \equiv|\,\} \\
P,Q{:}\,\mathsf{ZFC} \vdash \quad P \in_{\mathsf{ZFC}} Q & := & \exists (N;S;n){:}\in Q;\ x{:}\in S[n] :: (N;S;x) \in P
\end{array}$$

We now define the ZFC set constructions on the type T of triples. So, let $(N;S;n){:}\,T$ be a triple, $P{:}\,\mathcal{P}(T)$ a predicate and $F{:}\,T^T$ an operation on T.

$$\begin{array}{l}
\{_T\, t \in (N;S;n) \mid{:}\ P(t)\} \ := \\
\quad (\ 1+N; \\
\quad\ \{\ x{:}\in S[n];\ P(N;S;x) :: ((0;0),(1;x)) \\
\quad\ \mid\ (x,y){:}\in S :: ((1;x),(1;y)) \\
\quad\ \};(0;0)\ )
\end{array}$$

$$\begin{array}{l}
\bigcup_T (N;S;n) \ := \\
\quad (\ 1+N; \\
\quad\ \{\ x{:}\in S[n];\ y{:}\in Sx :: ((0;0),(1;y)) \\
\quad\ \mid\ (x,y){:}\in S :: ((1;x),(1;y)) \\
\quad\ \};(0;0)\ )
\end{array}$$

$$\mathcal{P}_T(N;S;n) \ :=$$

$$
\begin{array}{l}
(\ \ 1 + \mathcal{P}(S[n]) + N; \\
\quad \{\ \ P{:}\mathcal{P}(S[n]) :: ((0;0),(1;P)) \\
\quad |\ \ P{:}\mathcal{P}(S[n]); x{:}\in P :: ((1;P),(2;x)) \\
\quad |\ \ (x,y){:}\in S :: ((2;x),(2;y)) \\
\quad \};(0;0)\ )
\end{array}
$$

$$
\begin{array}{l}
\{_T\, x \in (N;S;n) :: Fx\} \ := \\
\quad \underline{\mathsf{let}}\ (M_y;Q_y;m_y) := F(N;S;y)\ \underline{\mathsf{in}} \\
\quad (\ \ 1 + \Sigma(y{:}\in S[n] :: M_y); \\
\quad\quad \{\ \ y{:}\in S[n] :: ((0;0),(1;y;m_y)) \\
\quad\quad |\ \ y{:}\in S[n]; (u,v){:}\in Q_y :: ((1;y;u),(1;y;v)) \\
\quad\quad \};(0;0)\ )
\end{array}
$$

$$
\begin{array}{l}
\omega_T \ := \\
\quad (\ \ 1 + \mathbb{N}; \\
\quad\quad \{\ \ i{:}\mathbb{N} :: ((0;0),(1;i)) \\
\quad\quad |\ \ i{:}\mathbb{N}; j{:}< i :: ((1;i),(1;j)) \\
\quad\quad \};(0;0)\ )
\end{array}
$$

It is straightforward to extend these constructions to ZFC, e.g. $\bigcup_{\mathsf{ZFC}} Q := \bigcup(t{:}\in Q :: \bigcup_T |t \equiv|\ )$. We leave it to the reader to check that they satisfy the axioms, and that the axioms of extensionality, foundation, and choice are satisfied.

A.6 An inductive model of ZFC

Yet a simpler model uses an inductive type. Note that $\mathsf{Fam}_0{:}\mathbf{TYPE}_1 \to \mathbf{TYPE}_1$ is a polynomial functor, with

$$
\begin{array}{rcl}
\mathsf{Fam}_0(X{:}\mathbf{TYPE}_1) & := & \Sigma(D{:}\mathbf{Type}_0 :: X^D)\ , \\
\mathsf{Fam}_0(h{:}A \to B) & := & (D;a) \mapsto (D;h^D.a)\ .
\end{array}
$$

We define a mebership relation $(\in_{\mathsf{f}}){:}\subseteq X \times \mathsf{Fam}_0\, X$ and a subfamily relation $(\subseteq_{\mathsf{f}}){:}\subseteq \mathsf{Fam}_0^2\, X$:

$$
\begin{array}{rcl}
x{:}T; t{:}\mathsf{Fam}\,T \vdash \quad x \in_{\mathsf{f}} t & := & \exists d{:}\mathsf{Dom}\,t :: x = t_d \\
t,t'{:}\mathsf{Fam}\,T \vdash \quad t \subseteq_{\mathsf{f}} t' & := & \forall x{:}\in_{\mathsf{f}} t :: x \in_{\mathsf{f}} t'
\end{array}
$$

An initial Fam_0-algebra contains families of families of families *ad infinitum*. Modulo the appropriate equation these families model sets.

$$
\begin{array}{rcl}
(\mathsf{ZFC};\tau) & := & \mu(\mathsf{Fam}_0;E)\ \underline{\mathsf{where}} \\
E(X;\phi) & := & \{f{:}\mathsf{Fam}_0^2\, X;\ f_0 \subseteq_{\mathsf{f}} f_1 \wedge f_1 \subseteq_{\mathsf{f}} f_0 :: (\phi.f_0, \phi.f_1)\,\} \\
x \in_{\mathsf{ZFC}} y & := & \exists(f{:}\mathsf{Fam}_0\,\mathsf{ZFC};\ i{:}\mathsf{Dom}\,f :: y = \tau.f \wedge x = f_i)
\end{array}
$$

The (total) interpretation $[\![\cdot]\!]{:}(\mathsf{ZFC} \triangleright \mathbf{Set})$ is recursively defined by:

$$
[\![\tau.f]\!] \ = \ \{x{:}\in_{\mathsf{f}} f :: [\![x]\!]\,\}
$$

The extensionality axiom (A.1) follows easily from $E(\mathsf{ZFC};\tau) \subseteq (=_{\mathsf{ZFC}})$ and the following lemma.

Lemma A.1 *For $x\colon \mathsf{ZFC}$, $f\colon \mathsf{Fam}\,\mathsf{ZFC}$, one has:*

$$x \in_{\mathsf{ZFC}} \tau.f \;\Leftrightarrow\; x \in_{\mathsf{f}} f$$

Proof. $\Leftarrow$ is trivial.

$\Rightarrow$: we must prove that if $\tau.f = \tau.f'$ and $x \in_{\mathsf{f}} f'$, then $x \in_{\mathsf{f}} f$. Note that, as $E(X;\phi)$ is an equivalence, one has $(=_{\mathsf{ZFC}}) = E(\mathsf{ZFC};\tau)$, so if $\tau.f = \tau.f'$ then $f' \subseteq_{\mathsf{f}} f$. ■

We define the operations required by the axioms (A.3) till (A.6), writing '$\in$' for $\in_{\mathsf{ZFC}}$, by:

$$\begin{array}{rcl}
\{\, x\colon\!\in s \mid\colon P(x)\} & := & \tau.(x\colon\!\in s;\; Px :: x\,) \\
\bigcup s & := & \tau.(y\colon\!\in s;\; x\colon\!\in y :: x\,) \\
\mathcal{P}(s) & := & \tau.(P\colon\mathcal{P}(\mathsf{ZFC}) :: \tau.(x\colon\!\in s;\; Px :: x)\,) \\
\omega & := & \tau.(i\colon\mathbb{N} :: \tau.(h.i)\,) \;\underline{\mathsf{where}} \\
 & & h.0 := \langle\rangle, \\
 & & h.(k+1) := h.k +\!\!+ \langle\tau.(h.k)\rangle \\
\{x\colon\!\in s :: Fx\,\} & := & \tau.(x\colon\!\in s :: Fx\,)
\end{array}$$

To check their properties, one applies lemma A.1 over and again.

Finally, the foundation axiom (A.7) holds by induction over the definition of ZFC, and the lemma.

A.7 Anti-foundation

Peter Aczel [4] proposed an alternative view on sets, in which non-wellfounded sets are permitted. He removed the foundation axiom (A.7) from ZFC, and replaced it by an *anti-foundation axiom* (AFA), stating that, given a directed graph, each node x in it corresponds to a set such that the elements of the set are the sets that correspond to the subnodes of x in the graph. Thus:

$$s\colon \mathbf{Set};\; R\colon \mathcal{P}(s\times s) \vdash \quad \exists!f\colon \mathbf{Set}^s :: \forall x\colon\!\in s :: fx = f[R[x]] \tag{A.9}$$

We may call this system ZFA. A noninductive model of ZFA can be obtained from our T as defined in section A.5, by using the dual equivalence relation:

$$\begin{array}{rcl}
(\equiv') & := & \bigcup(R\colon \mathcal{P}T^2 \mid\colon R \subseteq (\preceq_R) \cap (\succeq_R)) \\
\mathsf{ZFA} & := & \{t\colon T :: |t \equiv'|\,\}
\end{array}$$

Remark that $\equiv'$ is total, that is, $(=_T) \subseteq (\equiv')$.

Appendix B

ADAM's Type Theory

In this appendix we define the type theory ATT that forms the foundation of the language *ADAM* described in chapter 2. It includes impredicative propositions, a hierarchy of universes, strong sums for non-propositions, naturals, finite types, and equality types *à la* Martin-Löf. Without the naturals, finite types, and equality types, it would be Luo's Extended Calculus of Constructions [48].

As this calculus serves merely as a logical foundation, notational convenience is not of primary importance. However, we include a few derived notations to make direct employment of the calculus, as in the examples of appendix C, more comfortable.

We also outline a set-theoretical semantics of the calculus in B.10, assuming a sufficiently strong set theory.

B.1 Abstract syntax

We have the following abstract syntax for terms and contexts, where '::=', '|', '{', '}', and '.' are metasymbols, and T^* stands for a possibly empty list of expressions from class T. We assume a syntax class *Var* of variables, a class *Const* of constants, and a class *Nat* of naturals, with addition '+' and comparison '<', to be used for indexing constants.

$$\begin{array}{rcl} \mathit{Term} & ::= & \mathit{Var} \\ & | & \mathit{Const}(\mathit{Term},^*) \\ & | & (\mathit{Var} :: \mathit{Term})\ . \\ \mathit{Context} & ::= & \{\mathit{Var}\colon \mathit{Term};\}^*\ . \\ \mathit{Statement} & ::= & \mathit{Term}\colon \mathit{Term}\ . \end{array}$$

So (abstract) *terms* are built from variables, constants with a list of arguments, and abstractions $(v :: t)$, which are like $\lambda v.t$ in lambda calculus. As in typed lambda calculus *à la* Curry, abstractions do not carry the type of the bound variable with them, so terms will not have unique types.

A *context* Γ consists of a sequence of assumptions of the form $v\colon T$, where v is a variable and T a term (representing the type of v). We will define a derivability relation

$\Gamma \vdash t{:}T$. The intended meaning is that, for any correct assignment of values to the typed variables in Γ, term t represents a value of type T.

We write '_' for an anonymous variable. Among the constants in *Const* we use the following primitive ones, listed with their arity, notational sugar, and intended meaning.

Prop	0		Type of all propositions
$\mathbf{Type}_i$	0	for any $Nat\,i$	Type of all types of level i
Π	1		Cartesian product of a family of types
@	2	$f(a) := @(f,a)$	Selecting a component from a tuple or function
Σ	1		Disjoint sum of a family of types
$(;)$	2	$(a;b) := (;)(a,b)$	Element of a disjoint sum
Σ_elim	1		Elimination on a disjoint sum
n	0	for any $Nat\,n$	Finite type with elements $0,\ldots,n-1$
$(,_n)$	n	for any $Nat\,n$; $(a_0,\ldots,a_{n-1}) := (,_n)(a_0,\ldots,a_{n-1})$	Element of a cartesian product over finite type n
$\mathbb{N}$	0		Type of natural numbers
s	1		Successor of a natural
$\mathbb{N}$_rec	2		Recursion over the naturals
$\forall$	1		Universal quantification of a family of propositions
hyp	1		Proof of a universal proposition by hypothesis
app	1		Application of a proof of a universal proposition
$\exists$	1		Prop. stating existence of an inhabitant of a type
$\exists$_in	1		Proof of an existential proposition
$\exists$_elim	1		Elimination on an existential proposition
$=$	1	$(a =_A a') := @(=(A),(,_2)(a,a'))$	Equality predicate on any type
eq	0		Trivial proof of equality
ac	1		Proof by the axiom of choice

Parentheses may be omitted when no ambiguity arises.

B.2 Meta-predicates

The calculus defines a substitution operation and two predicates on terms and contexts.

B.2.1 Substitution. $s[v := t]$ for terms s, t and variable v, or more generally $s[\phi]$ where ϕ is a list of single substitutions, $\{Var := Term,\}^*\,\phi$, is defined as usual:

$$
\begin{array}{rcll}
v[\phi] & := & t & \text{if } in(\{v := t\},\phi), \text{ for some } t \\
v[\phi] & := & v & \text{otherwise} \\
c(t_0,\ldots,t_{n-1})[\phi] & := & c(t_0[\phi],\ldots,t_{n-1}[\phi]) & \\
(w :: s)[\phi] & := & (w' :: s[w := w'][\phi]) & (w' \text{ free in neither } (w :: s) \text{ nor } \phi)
\end{array}
$$

The use of named variables may of course be replaced by some numbering scheme.

B.2.2 Reduction. Reduction is a reflexive and transitive binary predicate $t \Rightarrow t'$ on terms. It is inductively defined by structural rules

$$\frac{}{t \Rightarrow t}$$

$$\frac{t \Rightarrow t' \quad t' \Rightarrow t''}{t \Rightarrow t''}$$

$$\frac{t_i \Rightarrow t_i' \quad (i = 0, \ldots, n-1)}{c(t_0, \ldots, t_{n-1}) \Rightarrow c(t_0', \ldots, t_{n-1}')}$$

$$\frac{t \Rightarrow t'}{(v :: t) \Rightarrow (v' :: t'[v := v'])} \qquad (v' \text{ not free in } (v :: t)\ )$$

and rules related to specific constants:

$$\begin{aligned}
(v :: b)(a) &\Rightarrow b[v := a] \\
\Sigma_\mathsf{elim}(t)(a; b) &\Rightarrow t(a)(b) \\
(t_0, \ldots, t_{n-1})(k) &\Rightarrow t_k \\
\mathbb{N}_\mathsf{rec}(b, t)(0) &\Rightarrow b \\
\mathbb{N}_\mathsf{rec}(b, t)(\mathsf{s}\, x) &\Rightarrow t(x)(\mathbb{N}_\mathsf{rec}(b, t)(x))
\end{aligned}$$

Two terms are called *convertible* when they reduce to the same thing: $t == t'$ when there is some term t'' such that $t \Rightarrow t''$ and $t' \Rightarrow t''$.

Reduction has the Church-Rosser property: if $t \Rightarrow t'$ and $t \Rightarrow t''$, then $t' \Rightarrow t'''$ and $t'' \Rightarrow t'''$ for some term t'''. Hence we have that convertibility is transitive.

B.2.3 Derivable judgements. A *judgement* consists of a context Γ and two terms t, T. *Derivability* of judgements is denoted by an infix turnstyle and colon, '$\Gamma \vdash t\colon T$', and is defined by a set of *rules*. Intuitively, such a judgement represents the assertion that for any assignment to the variables in Γ of values of the respective type, the term t denotes a value of the type denoted by T. See section B.10 for a formal semantics.

The structural rules for variable occurrences are:

$$\frac{\Gamma \vdash A\colon \mathbf{Type}_i}{\Gamma;\, v\colon A \vdash v\colon A} \tag{B.1}$$

$$\frac{\Gamma \vdash t\colon T \quad \Gamma \vdash A\colon \mathbf{Type}_i}{\Gamma;\, v\colon A \vdash t\colon T} \tag{B.2}$$

and the type T of a judgement may be replaced by a type that is convertible to T:

$$\frac{\Gamma \vdash t\colon T \quad T == T'}{\Gamma \vdash t\colon T'} \tag{B.3}$$

The rule for introducing a bound variable is

$$\frac{\Gamma;\, x\colon A \vdash b\colon Bx}{\Gamma \vdash (x :: b)\colon \Pi(A; B)} \tag{B.4}$$

All other rules are of the form

$$\frac{\begin{array}{l}\Gamma \vdash t_0 : T_0 \\ \quad\vdots \\ \Gamma \vdash t_{n-1} : T_{n-1}\end{array}}{\Gamma \vdash t' : T'}$$

for arbitrary Γ, and we will write them on a single line, as

$$t_0 : T_0;\ \ldots;\ t_{n-1} : T_{n-1} \ \vdash\ t' : T'$$

As a derived rule, derivability is closed under substitution:

$$\frac{\begin{array}{l}\Gamma;\ x : A;\ \Gamma' \vdash t : T \\ \Gamma \vdash a : A\end{array}}{\Gamma;\ \Gamma'[x := a] \vdash t[x := a] : T[x := a]}$$

B.3 Universes

A universe is a type whose elements are types themselves. There is a universe **Prop** of propositions and a cumulative hierarchy of universes $\mathbf{Type}_i$, each being an inhabitant of the next one.

$$\vdash\ \mathbf{Prop} : \mathbf{Type}_0 \tag{B.5}$$

$$\vdash\ \mathbf{Type}_i : \mathbf{Type}_{i+1} \tag{B.6}$$

$$P : \mathbf{Prop}\ \vdash\ P : \mathbf{Type}_i \tag{B.7}$$

$$T : \mathbf{Type}_i\ \vdash\ T : \mathbf{Type}_j \qquad \text{when } i < j \tag{B.8}$$

If the subscript i of **Type** is irrelevant, it will not be shown.

B.4 Products

The type $\Pi(A; B)$ is thought of as the product of all Bx for $x : A$; see rule (B.4) for introducing its elements.

In the following rules exponentiation of types is used, T^A, which stands itself for a product type $\Pi(A; (\_ :: T))$. So we have $\vdash (x :: t) : T^A$ when $x : A \vdash t : T$ and x doesn't occur free in T. Rule (B.14) (extensionality) refers to the equality predicate described in section B.8.

$$A : \mathbf{Type}_i;\ B : \mathbf{Type}_i^A\ \vdash\ \Pi(A; B) : \mathbf{Type}_i \tag{B.9}$$

$$A : \mathbf{Type}_i;\ P : \mathbf{Prop}^A\ \vdash\ \forall(A; P) : \mathbf{Prop} \tag{B.10}$$

$$f : \Pi(A; P)\ \vdash\ \mathsf{hyp}\, f : \forall(A; P) \tag{B.11}$$

$$f : \Pi(A; B);\ a : A\ \vdash\ fa : Ba \tag{B.12}$$

$$p : \forall(A; P);\ a : A\ \vdash\ \mathsf{app}(p, a) : Pa \tag{B.13}$$

$$f, g : \Pi(A; B);\ h : \forall(A; (x :: fx =_{Bx} gx))\ \vdash\ h : (f =_{\Pi(A;B)} g) \tag{B.14}$$

A pair $(A;B)$ as in (B.9) is called a *family of types*. We define the following alternative notations. In the first one, variable v may occur in B.

$$\begin{aligned}(v{:}A :: B) &:= (A;(v :: B))\\ B^A &:= \Pi(\_{:}A :: B)\\ P \Rightarrow Q &:= \forall(\_{:}P :: Q)\end{aligned}$$

B.5 Sums

In the rules for strong Σ, we use a primitive constant Σ_elim rather than operations fst and snd.

$$A{:}\mathbf{Type}_i;\ B{:}\mathbf{Type}_i^A \ \vdash\ \Sigma(A;B){:}\mathbf{Type}_i \qquad \text{(B.15)}$$

$$B{:}\mathbf{Type}^A;\ a{:}A;\ b{:}Ba \ \vdash\ (a;b){:}\Sigma(A;B) \qquad \text{(B.16)}$$

$$T{:}\mathbf{Type}^{\Sigma(A;B)};\ t{:}\Pi(x{:}A :: \Pi(y{:}Bx :: T(x;y))) \ \vdash\ \Sigma\_\mathsf{elim}\,t{:}\Pi(\Sigma(A;B);T) \qquad \text{(B.17)}$$

Thus, the expression 'Σ_elim$(x :: (y :: t_{xy}))$' denotes the function that maps $(x;y)$ to t_{xy}. A pattern-matching notation suggestive of this is given in section B.11.

B.6 Finite types

Any $Nat\,n$ denotes a type with n elements, named by the Nat's 0 till $n-1$.

$$\vdash\ n{:}\mathbf{Type}_i \qquad \text{(B.18)}$$

$$\vdash\ k{:}n \qquad \text{where } k < n \qquad \text{(B.19)}$$

$$T{:}\mathbf{Type}^n;\ t_i{:}T(i) \text{ for } i < n \ \vdash\ (t_0,\ldots,t_{n-1}){:}\Pi(n;T) \qquad \text{(B.20)}$$

Sequences of arbitrary length are denoted using angle brackets. This allows elegant definitions of finite products and sums:

$$\begin{aligned}\langle t_0,\ldots,t_{n-1}\rangle &:= (n;(t_0,\ldots,t_{n-1}))\\ B_0 \times B_1 &:= \Pi\langle B_0,B_1\rangle\\ B_0 + B_1 &:= \Sigma\langle B_0,B_1\rangle\\ Q_0 \wedge Q_1 &:= \forall\langle Q_0,Q_1\rangle\end{aligned}$$

B.7 Naturals

The rules for naturals are exactly as in *ADAM*, paragraph 2.9.1.

$$\vdash\ \mathbb{N}{:}\mathbf{Type}_i \qquad \text{(B.21)}$$

$$\vdash\ 0{:}\mathbb{N} \qquad \text{(B.22)}$$

$$x{:}\mathbb{N} \ \vdash\ \mathsf{s}\,x{:}\mathbb{N} \qquad \text{(B.23)}$$

$$T{:}\mathbf{Type}^{\mathbb{N}};\ b{:}T(0);\ t{:}\Pi(x{:}\mathbb{N} :: \Pi(h{:}Tx :: T(\mathsf{s}\,x))) \ \vdash\ \mathbb{N}\_\mathsf{rec}(b,t){:}\Pi(\mathbb{N};T) \qquad \text{(B.24)}$$

B.8 Equality

Rules for the equality predicate are the following.

$$a{:}A;\ b{:}A \ \vdash\ (a =_A b){:}\mathbf{Prop} \tag{B.25}$$
$$a{:}A \ \vdash\ \mathsf{eq}{:}(a =_A a) \tag{B.26}$$
$$\_{:}(A =_{\mathbf{Type}} B);\ a{:}A \ \vdash\ a{:}B \tag{B.27}$$
$$P,Q{:}\mathbf{Prop};\ h{:}(P \Rightarrow Q) \wedge (Q \Rightarrow P) \ \vdash\ h{:}(P =_{\mathbf{Prop}} Q) \tag{B.28}$$
$$P{:}\mathbf{Prop};\ p,q{:}P \ \vdash\ \mathsf{eq}{:}(p =_P q) \tag{B.29}$$

This simple version of type conversion (B.27) has a drawback: correct terms need not normalize, because in an inconsistent context any two types can be proven equal. One has, for example,

$$A{:}\mathbf{Type};\ h{:}(A =_{\mathbf{Type}} (A \to A)) \vdash (x :: xx)(x :: xx){:}A\ .$$

An explicit conversion construct, as suggested in C.3.2, third point, would prevent this. In any case, terms that are correct in the empty context reduce to head normal form.

There must be an equality rule for all language constructs, stating that two terms constructed from equal subterms are equal. We do not list all these, but one simple and two more complicated cases are:

$$a =_A a';\ b =_B b' \ \vdash\ (a,b) =_{A\times B} (a',b') \tag{B.30}$$
$$A =_{\mathbf{Type}} A';\ B =_{\mathbf{Type}^A} B' \ \vdash\ \Pi(A;B) =_{\mathbf{Type}} \Pi(A';B') \tag{B.31}$$
$$A =_{\mathbf{Type}} A';\ (a,b) =_{A^2} (a',b') \ \vdash\ (a =_A b) =_{\mathbf{Prop}} (a' =_{A'} b') \tag{B.32}$$

This suffices to derive symmetry and transitivity of equality. But rules like (B.31) have a snag: the type of $B'{:}\mathbf{Type}^{A'}$ has to be converted via (B.27) to get $B'{:}\mathbf{Type}^{A}$. An alternative would be to use an equality predicate indexed by *two* type expressions, which have to denote equal types, thus:

$$\_{:}A = B;\ a{:}A;\ b{:}B \ \vdash\ (a\ {}_A{=}_B\ b){:}\mathbf{Prop}$$

B.9 Existential propositions

As discussed in appendix C, we add strong existential propositions and the axiom of choice. For A a type, $\exists A$ means 'A is inhabited'.

$$A{:}\mathbf{Type}_i \ \vdash\ \exists A{:}\mathbf{Prop} \tag{B.33}$$
$$a{:}A \ \vdash\ \exists\_\mathsf{in}\, a{:}\exists A \tag{B.34}$$
$$\begin{array}{r} T{:}\mathbf{Type}^{\exists A};\\ t{:}\Pi(x{:}A :: T(\exists\_\mathsf{in}\, x));\\ d{:}\forall(x,y{:}A :: tx = ty) \end{array} \ \vdash\ \exists\_\mathsf{elim}(t){:}\Pi(\exists A;T) \tag{B.35}$$
$$B{:}\mathbf{Type}^A;\ p{:}\forall(x{:}A :: \exists(Bx)) \ \vdash\ \mathsf{ac}\, p{:}\exists\,\Pi(A;B) \tag{B.36}$$

Note: in appendix C we write $\exists\_\mathsf{elim}(t|;d)$, rather than $\exists\_\mathsf{elim}\,t$, to make the proof obligation d explicit.

As discussed in C.3.2, we cannot use a reduction rule $\exists\_\mathsf{elim}(t)(\exists\_\mathsf{in}\,a) => ta$. Rather we add an equation:

$$\vdash\ \mathsf{eq}\colon(\exists\_\mathsf{elim}(t)(\exists\_\mathsf{in}\,a) = ta) \tag{B.37}$$

B.10 Semantics

We wish to assign a simple set-theoretical semantics $[\![t]\!]\sigma$ to terms t (under valuation σ), such that any function $f\colon A \to B$ simply denotes the set of pairs $\{x\colon\in [\![A]\!]\sigma :: \langle x, [\![f]\!]\sigma.x\rangle\}$. Unfortunately, an abstraction $(v :: t)$ doesn't show up the type A of its bound variable v. Therefore we introduce *annotated terms*, where abstractions $(v ::_A t)$ are annotated with this type A. Annotated terms are given by:

$$\begin{array}{rcl} \mathit{ATerm} & ::= & \mathit{Var} \\ & | & \mathit{Const}(\mathit{ATerm},^*) \\ & | & (\mathit{Var} ::_{\mathit{ATerm}} \mathit{ATerm})\ . \end{array}$$

For any $\mathit{Term}\,t$, we define its class of *annotations*, a subclass of Aterm.

- The only annotation of a variable v is v;
- An annotation of $\mathsf{c}(t_0,\ldots,t_{n-1})$ is $\mathsf{c}(t'_0,\ldots,t'_{n-1})$ where each t'_i is an annotation of t_i.
- An annotation of an abstraction $(v :: t)$ is $(v ::_{A'} t')$ where A' is any ATerm and t' is an annotation of t.

All propositions denote subsets of $\{\emptyset\}$. In particular, we will have $[\![\mathsf{False}]\!]\sigma = \emptyset$ and $[\![\mathsf{True}]\!]\sigma = \{\emptyset\}$, and all terms that denote proofs are mapped onto the empty set $\emptyset$. Thus, our semantics pays no respect to the computational contents of proofs.

For any $\mathit{Const}\,\mathsf{c}$ and any list of sets $\bar{s}$ of length arity of c, we define a set $[\![]\!]c(\bar{s})$. Empty argument lists are omitted. Note that we use the set encoding of section A.2.

$$\begin{array}{rcl} [\![\mathbf{Prop}]\!] & := & \mathcal{P}\{\emptyset\} \\ [\![\forall]\!](\langle A, P\rangle) & := & \{\,\emptyset \mid\colon \forall x\colon\in A :: \emptyset \in P(x)\} \\ [\![\exists]\!](A) & := & \{\,\emptyset \mid\colon \exists x\colon\in A :: \mathsf{True}\} \\ [\![\mathsf{hyp}]\!](x), [\![\mathsf{app}]\!](x,y), [\![\exists\_\mathsf{in}]\!](x), [\![\mathsf{ac}]\!](x), [\![\mathsf{eq}]\!] & & \\ & := & \emptyset \\ [\![\exists\_\mathsf{elim}]\!](t) & := & \{p\colon\in t :: \langle\emptyset, \mathsf{snd}\,p\rangle\} \\ [\![\Pi]\!](\langle A, B\rangle) & := & \{\,f\colon\in (\bigcup \mathsf{cod}\,B)^A \mid\colon \forall x\colon\in A :: f(x) \in B(x)\} \\ [\![@]\!](f,a) & := & f(a) \\ [\![\Sigma]\!](\langle A, B\rangle) & := & \{x\colon\in A;\ y\colon\in B(x) :: \langle x, y\rangle\} \\ [\![\,;]\!](a,b) & := & \langle a, b\rangle \end{array}$$

$$
\begin{array}{rcl}
[\![\Sigma\_\mathsf{elim}]\!](t) & := & \{x{:}\in \mathsf{dom}\, t;\ y{:}\in \mathsf{dom}\, t(x) :: \langle\langle x, y\rangle, t(x)(y)\rangle\} \\
[\![n]\!] & := & n \\
[\![\ ,n\ ]\!](t_0,\ldots,t_{n-1}) & := & \{\langle 0, t_0\rangle, \ldots, \langle n-1, t_{n-1}\rangle\} \\
[\![\mathbb{N}]\!] & := & \omega \\
[\![\mathsf{s}]\!] & := & \{x{:}\in\omega :: \langle x, x+1\rangle\} \\
[\![\mathbb{N}\_\mathsf{rec}]\!](b,t) & := & \bigcap(f{:}\subseteq \omega\times \mathsf{dom}\, t(0) \mid: \\
 & & \quad \langle 0,b\rangle\in f \wedge \forall\langle x,y\rangle{:}\in f :: \langle x+1, t(x)(y)\rangle\in f) \\
[\![=]\!](A) & := & \{x, x'{:}\in A :: \langle [\![\ ,2\ ]\!](x,x'), \{\emptyset \mid: x = x'\}\rangle\} \\
[\![\mathbf{Type}_i]\!] & := & \bigcap(U \mid: \quad \omega\subseteq U \ \wedge\ [\![\mathbb{N}]\!], [\![\mathbf{Prop}]\!]\in U \\
 & & \quad \wedge \forall A{:}\in U;\ B{:}\in U^A :: [\![\Pi]\!]\langle A,B\rangle, [\![\Sigma]\!]\langle A,B\rangle\in U \\
 & & \quad \wedge \forall j{:}< i :: [\![\mathbf{Type}_j]\!]\in U)
\end{array}
$$

The last line gives an iterated inductive definition of $[\![\mathbf{Type}_i]\!]$ of the form $\bigcap(U \mid: F.U \subseteq U)$ for a monotonic functor F that is not bounded in the sense of section 8.1. Existence of such an inductive set cannot be shown in ZFC, for $[\![\mathbf{Type}_0]\!]$ yields already a model of ZFC (see section A.5). So this requires a strengthening of ZFC, that allows one to give inductive set definitions with clauses of the form

$$\forall\, A{:}\in U;\ B{:}\in U^A :: \phi(A;B)\in U\ .$$

We think a suitable large-cardinal axiom will do.

Substituting a special constant ω for i with $j<\omega$ for all $Nat\, j$, the resulting set $D := [\![\mathbf{Type}_\omega]\!]$ may serve to model types, and $E := \bigcup D$ to model values.

A *valuation* is a partial function $\sigma\colon Var \to E$. The semantics of an annotated term t assigns to any valuation σ a set $[\![t]\!]\sigma\in E$:

$$
\begin{array}{rcl}
[\![v]\!]\sigma & := & \sigma(v) \qquad \text{for } Var\, v \\
[\![\mathsf{c}(t_0,\ldots,t_{n-1})]\!]\sigma & := & [\![]\!]c([\![t_0]\!]\sigma,\ldots,[\![t_{n-1}]\!]\sigma) \\
[\![(v ::_A b)]\!]\sigma & := & \{x{:}\in [\![A]\!]\sigma :: \langle x, [\![b]\!](\sigma \mid v\mapsto x)\rangle\}
\end{array}
$$

The semantics of an annotated context Γ' is a set $[\![\Gamma']\!]$ of valuations:

$$
\begin{array}{rcl}
[\![\{\}]\!] & := & \{\emptyset\} \\
[\![\Gamma';\ v{:}T']\!] & := & \{\sigma{:}\in [\![\Gamma']\!];\ x{:}\in [\![T']\!]\sigma :: (\sigma \mid v\mapsto x)\}
\end{array}
$$

We define an annotated term to be *correct* in an annotated context Γ', as follows.

- Any variable v is correct in Γ'
- A term $\mathsf{c}(t_0,\ldots,t_{n-1})$ is correct in Γ' if each t_i is correct in Γ' and moreover, if c is @ and n is 2, then
$$\forall\sigma{:}\in [\![\Gamma']\!] :: [\![t_1]\!]\sigma\in \mathsf{dom}[\![t_0]\!]\sigma$$
and if c is $\forall$, Π, or Σ, and n is 1, then
$$\forall\sigma{:}\in [\![\Gamma']\!] :: \mathsf{fst}[\![t_0]\!]\sigma = \mathsf{dom}\,\mathsf{snd}[\![t_0]\!]\sigma$$

- An abstraction $(v ::_A b)$ is correct in Γ' if b is correct in $\Gamma'; v{:}A$

Validity is defined by:

$$\begin{array}{rcl} \Gamma' \models t'{:}T' & := & \forall \sigma{:} \in [\![\Gamma']\!] :: [\![t']\!]\sigma \in [\![T']\!]\sigma \\ \Gamma' \models s' = t' & := & \forall \sigma{:} \in [\![\Gamma']\!] :: [\![s']\!]\sigma = [\![t']\!]\sigma \end{array}$$

Now, we hope to have a theorem like the following, but the details of assigning annotations are tricky:

Conjecture (Soundness).

1. If $\Gamma \vdash T{:}\mathbf{Type}$, and Γ', T' are correct annotations of Γ, T, and if $s == t$ and s', t' are correct annotations of s, t in Γ' and $\Gamma' \models s', t'{:}T'$, then $\Gamma' \models s' = t'$
2. If $\Gamma \vdash t{:}T$, then for any correct annotations Γ', T' of Γ, T, there is a correct annotation t' of t such that $\Gamma' \models t'{:}T'$

B.11 More derived notations

B.11.1 Σ-elimination. One may use the following notations for elimination on a Σ-type.

$$\begin{array}{rcl} ((x;y) :: t_{xy}) & := & \Sigma\_\mathsf{elim}(x :: (y :: t_{xy})) \\ \mathsf{fst} & := & ((x;y) :: x) \\ \mathsf{snd} & := & ((x;y) :: y) \end{array}$$

B.11.2 Subtypes. When A is a type, and P a predicate on A, then $\Sigma(A;P)$ represents the type of all $a{:}A$ that come with a proof $p{:}Pa$. As all proofs of a proposition are equal, we have that $(a;p) = (a';p')$ just when $a = a'$. This type gets a special notation.

$$\begin{array}{rcl} \{x{:}A \mid{:} P_x\} & := & \Sigma(x{:}A :: P_x) \\ (a|;p) & := & (a;p) \end{array}$$

One may read '|:' as "such that" and '|;' as "because of". Both bind weaker than '::'. As the proof component p of $(a|;p)$ is irrelevant, we sometimes write just a.

Appendix C

Proof elimination in Type Theory

When Type Theory is to be used as a fully fledged foundation of mathematics, presence of powersets, or equivalently impredicative propositions, is indispensable. We remark that, e.g., the 'iota' or Frege's description operator denoting the element of a one-element set is not representable in current type theories. We propose an existential quantifier with a new elimination rule, and show how the iota operator and quotient types are then representable. We use a version of type theory that unifies finite and infinite products and sums in a particularly elegant way.

This material was distributed earlier, together with appendix B, as report [15].

C.1 Introduction

The basic thought of Brouwer's intuitionistic logic was, a proposition should only be acknowledged as true if we have a construction validating its truth. Martin-Löf's Intuitionistic Type Theory (ITT) [56] was developed to clarify this: a proposition was identified with its set of constructions, called a type, and proofs were identified with constructions. From a construction for the existential statement $\exists(x{:}A :: B_x)$, which is identified in ITT with the generalized sum type $\Sigma(x{:}A :: B_x)$, one can construct the witnessing element of A by the function $\mathsf{fst}{:}\,\Sigma(x{:}A :: B_x) \to A$.[1]

ITT does not allow impredicative quantification; it makes no sense to the orthodox intuitionist to quantify over the class of all propositions before this class is completed. If $\mathbf{Type}_0$ is a universe of types, then types involving $\mathbf{Type}_0$ cannot reside in $\mathbf{Type}_0$ itself.

In traditional set theory, on the other hand, even in a constructive version, one *can* construct the powerset $\mathcal{P}T$ of any set T. This is a set whose elements are definable by arbitrary predicates on T, even those involving quantification over the powerset $\mathcal{P}T$ itself. Thus, the class of propositions is considered to be understood *a priori*.

Coquand and Huet introduced a type theory, the Calculus of Constructions (CC) [21], that has a type of propositions (**Prop**, residing inside $\mathbf{Type}_0$) that allows impredicative

[1] Subscripts stand for variables that may occur in an expression. The symbol '::' separates typed bound variables from the body of a quantification. We have $(x :: b_x){:}\,\Pi(x{:}A :: B_x)$.

quantification. The system has no Σ-constructor. While Luo [48] added (strong) Σ for the higher type universes, it cannot be consistently added for **Prop** [43]. (Propositions in **Prop** are normally interpreted as sets that have at most one element, but Σ builds types with more elements.) One can only define weak existential quantification:

$$\exists_{\mathsf{W}}(x{:}A :: P_x) \;:=\; \forall(X{:}\mathbf{Prop} :: \forall(x{:}A :: P_x \Rightarrow X) \Rightarrow X)$$

From a proof of such an existential proposition one cannot construct the witnessing $x{:}A$ inside the system, even if this witness is provably unique. Formally, there is no function $f{:}\exists_{\mathsf{W}}(x{:}A :: P_x) \to A$, and not even a function $f{:}\exists_{\mathsf{W}}!(x{:}A :: P_x) \to A$, as CC does not allow object construction using proof information. Addition of the latter f would be perfectly valid in the standard set interpretation.

Now, our purpose is to develop type theory into a complete alternative to traditional set theory as a foundation of mathematics. It is not our purpose to extract programs from constructions by omitting redundant proof information. Any kind of reasoning representable in set theory (but not specific to sets) should be representable in our type theory. This involves:

- An impredicative universe of propositions (**Prop**) should be present, so that powertypes are definable: $\mathcal{P}T := (T \to \mathbf{Prop})$. This makes a type theory into a *topos*. Two propositions are equal if they are equivalent. Two proofs of the same proposition are always equal. We will use some appropriate set notations, particularly '$\in$' for subset membership.

- Extra rules for equality types should be present, including type conversion and extensionality. This is standard in ITT, not in CC. A readable notation for fully formal equality proofs is missing. We will use a semi-formal notation, which should guarantee the existence of a proof object.

- For any ordinarily definable object, there should be an expression in type theory denoting it. Equivalently, *function comprehension* should be possible: from a proof that a relation $R{:}\mathcal{P}(A \times B)$ is single valued, the corresponding function $f{:}A \to B$ should be constructible.

In this paper we take a type theory that satisfies the first two requirements, and study the last point. Traditionally, a (constructive) object definition may consist of a *description*, being a predicate together with a (constructive) proof that the predicate is satisfied by one and only one object. Gottlob Frege [31, S 11] introduced a indexdescription operator *description operator* 'ι' (iota) into predicate calculus to denote this object. In a type theory where propositions are distinguished from types, like CC, one cannot obtain the object from the proof.

Let T be the subtype of objects satisfying the predicate. Assume we have

$$p{:}\exists x{:}T :: \forall y{:}T :: x =_T y \ .$$

We need an expression that extracts the object, say $\iota_T(p){:}T$. Rather than adding primitive rules for ι, we propose an equivalent principle in section C.3 for making use of proof information in object expressions.

An essentially equivalent principle is proposed by Pavlović in [71, par. 32], but not worked out. The 'new set type' $\{x : A||P_x\}$ proposed by Constable [19, section 3.1] for Nuprl is based on the same idea too, but doesn't really increase the strength of the system: as Nuprl has no impredicative propositions, one can use a Σ-type modulo the appropriate equivalence.

C.2 The basic system

In this paper, we use the variant of type theory described in appendix B, which includes impredicative propositions, a hierarchy of universes, strong sums for non-propositions, finite types and equality types *à la* Martin-Löf.

Although proofs of propositions always have a proof expression, we won't often show this expression. A really practical formal language for proofs would be much more elaborate. Rather, we use the usual informal language to describe proofs.

In our expressions we will also use *goal variables*, starting with a question mark like '?1', and usually followed by a typing. These represent subexpressions to be defined later on. All names that are visible in the context of a goal variable may be used in its definition as well.

C.3 Strong existence

C.3.1 New rules

We introduce a new existential quantifier with a stronger elimination rule than one has for $\exists_{\mathsf{W}}$ as defined in section C.1. Actually, we define $\exists$ not as a quantifier, but as a constructor operating on types, see rule (C.1). The quantifier is then recovered via the subset type by

$$\exists(A;P) \;:=\; \exists\{x{:}A \mid{:} Px\}\ .$$

The rules are suggested by viewing the proposition $\exists T$ as the quotient type of T modulo the equivalence relation that identifies everything in T. This quotient type contains at most one equivalence class indeed.

$$A{:}\mathbf{Type} \;\vdash\; \exists A{:}\mathbf{Prop} \tag{C.1}$$

$$a{:}A \;\vdash\; \exists\_\mathsf{in}\, a{:}\exists A \tag{C.2}$$

$$\begin{array}{l} T{:}\mathbf{Type}^{\exists A};\\ t{:}\Pi(x{:}A :: T(\exists\_\mathsf{in}\, x));\\ d{:}\forall x, y{:}A :: tx = ty \end{array} \;\vdash\; \exists\_\mathsf{elim}(t|;d){:}\Pi(\exists A;T) \tag{C.3}$$

Note that d is not always shown. The expected reduction rule is

$$\exists\_\mathsf{elim}(t|;d)(\exists\_\mathsf{in}\, a) \;=\!>\; (ta)\ , \tag{C.4}$$

but see the next subsection.

C.3.2 Difficulties with reduction to canonical form

Looking at (C.4), we see that in order to reduce an *object* expression '$(\exists\_\mathsf{elim}\, t)(p)$' one must obtain the canonical form of the *proof* expression p. Therefore, *if* we wish to preserve the attractive property of constructive type theory that any closed expression of some type is reducible to head canonical form for that type, we must take care of the following points.

- Proof information is no longer irrelevant and cannot be removed from some language constructs. For example, objects of a subtype $\{x{:}A \mid{:}\, P_x\}$ cannot be just single $a{:}A$ for which there exists a proof $p{:}P_a$, but must really be pairs $(a|;p)$.

- There should be reduction rules for all noncanonical constructs for proof objects. For example, if we have the *axiom of choice* (which holds trivially in pure ITT, without **Prop**, by the identification of proofs and constructions),

$$B{:}\mathbf{Type}^A;\ p{:}\forall(x{:}A :: \exists(Bx)) \ \vdash\ \mathsf{ac}\, p{:}\exists\,\Pi(A;B)\ , \tag{C.5}$$

  then we must also add a reduction rule to the effect that

$$\mathsf{ac}(x :: \exists\_\mathsf{in}\, b_x) \ =>\ \exists\_\mathsf{in}(x :: b_x)\ . \tag{C.6}$$

  Here we encounter a serious problem: not all $p{:}\forall(x{:}A :: \exists\, B_x)$ reduce to the form $(x :: \exists\_\mathsf{in}\, b_x)$. An ad-hoc solution is to make both $\exists\_\mathsf{in}$ and ac implicit, so that reduction rule (C.6) becomes void. An alternative is to use an *untyped* $\exists\_\mathsf{out}$, and reduction rules:

$$\begin{aligned} \mathsf{ac}\, p &\ =>\ \exists\_\mathsf{in}(x :: \exists\_\mathsf{out}(px)) \\ \exists\_\mathsf{out}(\exists\_\mathsf{in}\, a) &\ =>\ a \end{aligned}$$

- Rule (B.28) says that equivalent propositions are equal. However, a canonical proof term of a proposition need not be a canonical term of an equivalent proposition. Therefore, the type-conversion rule (B.27) has to specify a conversion on term a too. The rule might look like

$$e{:}(A =_{\mathbf{Type}} B);\ a{:}A \ \vdash\ (e \cdot\!> a){:}B$$

  together with a bunch of reduction rules for all constructs that prove equality between types.

In short, the system appears to become rather ugly.

We aim for elegance and therefore choose to part with this property of reduction to canonical form. However, as the calculus is still constructive, one may devise a procedure to extract from a given proof a separate term containing its computational content. Several implemented type theories, including Nuprl and the Calculus of Constructions, do already use such a procedure.

C.4 Applications

C.4.1 Iota

C.4.1 From 'exists' to 'iota'. Now, using $\exists$_elim we can define ι, the construct that, from a proof that a type has a unique element, constructs that element. First, $!A$ is the subtype that, if A has a unique element, contains that single element, and is empty otherwise. Next ι is defined by a $\exists$_elim on $\mathsf{fst}\colon !A \to A$, with a very simple proof that fst does always yield the same result on $!A$.

$$\begin{array}{rcl} !(A\colon \mathbf{Type}) & := & \{x\colon A \mid\colon \forall y\colon A :: x =_A y\} \\ \iota_A\colon \exists !A \to A & := & \exists\_\mathsf{elim}(\,\mathsf{fst}\mid;(x|;p),(y|;q)\colon !A :: py(\colon x = y)) \end{array}$$

C.4.2 From 'iota' to 'exists'. The converse is also possible: we can derive (C.1–C.3) with $\exists$ defined as $\exists_{\mathsf{W}}$ when we assume $\iota_A\colon \exists !A \to A$.

$$\begin{array}{rcl} \exists(A\colon \mathbf{Type}) & := & \forall(X\colon \mathbf{Prop} :: \forall(x\colon A :: X) \Rightarrow X) \\ \exists\_\mathsf{in}(x\colon A) & := & (X :: (h :: hx)) \\ \exists\_\mathsf{elim}(t|;d) & := & (p\colon \exists A :: ?1\colon Tp) \end{array}$$

The context of the goal $?1$ is

$t\colon \Pi(x\colon A :: T(\exists\_\mathsf{in}\, x))$;
$d\colon \forall x, y\colon A :: tx = ty$;
$p\colon \exists A$.

To solve $?1\colon Tp$, we define S to be the subtype containing all tx for $x\colon A$,

$$S \;:=\; \{u\colon Tp \mid\colon \exists x\colon A :: u = tx\} \;.$$

Such a type might be noted as $\{x\colon A :: tx\}$. For $x\colon A$ let $\mathsf{si}\, x\colon S := (tx|;\exists\_\mathsf{in}(x|;\mathsf{eq}))$ be the corresponding S-element.

Using p and d we can prove that S has a unique element:

$$s\colon \exists !S \;:=\; p(\exists !S)(x\colon A :: \exists\_\mathsf{in}(\mathsf{si}\, x \mid;(y|;e)\colon S :: ?2\colon tx = y)) \;.$$

The definition of $?2$, using d and $e\colon \exists(x\colon A :: y = tx)$, is left to the reader. Then we take

$$?1 \;:=\; \mathsf{fst}(\iota_S s) \;.$$

C.4.2 Quotient types

Another application arises with quotient types. These are sometimes added to type theory as primitives [18], but with the strong-existence construct we can *define* them in much the same way as they are defined in set theory. Furthermore, there is a construction dealing with quotient types that should follow from the rules, but which is not derivable in ordinary type theory.

C.4.3 Specification. We wish quotient types to satisfy the following rules:

$$A{:}\mathbf{Type};\ R{:}\mathcal{P}A^2 \quad\vdash\quad A//R{:}\mathbf{Type} \tag{C.7}$$

$$a{:}A \quad\vdash\quad //\_\mathsf{in}_R\, a{:}A//R \tag{C.8}$$

$$x,y{:}A;\ r{:}(x,y)\in R \quad\vdash\quad //\_\mathsf{in}_R\, x = //\_\mathsf{in}_R\, y \tag{C.9}$$

$$\begin{array}{rl} & T{:}\mathbf{Type}^{A//R};\\ & t{:}\Pi(x{:}A :: T(//\_\mathsf{in}\, x));\\ d{:}\forall(x,y){:}\in R :: tx = ty \quad\vdash & //\_\mathsf{elim}(t|;d){:}\Pi(A//R;T) \end{array} \tag{C.10}$$

$$//\_\mathsf{elim}(t\,|;d)(//\_\mathsf{in}\, a) \quad => \quad ta \tag{C.11}$$

A typing '$(x,y){:}\in R$' abbreviates $x,y{:}A;\ r{:}(x,y)\in R$. Note that it is not necessary to require that R be an equivalence relation. Note also that the rules for $\exists$ are exactly those for $//$ with R instantiated to the total relation $((x,y) :: \mathsf{True})$, except that $\exists A$ is a proposition rather than only a type.

C.4.4 Implementation. We present a definition of quotient types satisfying the specification above. It corresponds to the normal set-theoretic construction: $A//R$ is the subtype of those subsets of A that are equivalence classes of some $x{:}A$, where the equivalence class of x is the least subset of A that contains x and is closed under R.

Let $\mathsf{Equiv}(Q{:}\mathcal{P}A^2)$ be the proposition stating that Q is an equivalence relation. First we define the infix relation $\equiv_R$, read 'equivalent modulo R' as the least equivalence containing R. The so-called 'section' $(x\equiv_R){:}\mathcal{P}A$ stands for the predicate (or subset) $(y :: x\equiv_R y)$, which is the equivalence class of x.

$$\begin{array}{rcl} (\equiv_R) & := & \bigcap(Q{:}\mathcal{P}A^2\,|{:}\, R\subseteq Q \wedge \mathsf{Equiv}\, Q)\\ A//R & := & \{P{:}\mathcal{P}A\,|{:}\,\exists x{:}A :: P = (x\equiv_R)\} \qquad (= \{x{:}A :: (x\equiv_R)\})\\ //\_\mathsf{in}_R(x{:}A) & := & ((x\equiv_R)\,|;\exists\_\mathsf{in}(x|;\mathsf{eq}))\\ //\_\mathsf{elim}(t\,|;d) & := & ((P|;e) :: \exists\_\mathsf{elim}((x|;p) :: (?1)tx\,|;(x|;p),(y|;q) :: ?2{:}tx = ty)\, e\,) \end{array}$$

The goal variables in this last definition still have to be filled in. From the required typing (C.10) of $//\_\mathsf{elim}$ one can deduce that the types of the bound variables are:

$t{:}\Pi(x{:}A :: T(//\_\mathsf{in}\, x))$
$d{:}\forall(x,y){:}\in R :: tx = ty$
$P{:}\mathcal{P}A$
$e{:}\exists x{:}A :: P = (x\equiv_R)$
$x{:}A;\ p{:}(P = (x\equiv_R))$
$y{:}A;\ q{:}(P = (y\equiv_R))$

So the subexpression tx has type $T(//\_\mathsf{in}_R\, x)$, while type $T(P|;e)$ is required. A type conversion can be inserted at ?1, for:

$$\begin{array}{rcll} & & //\_\mathsf{in}_R\, x & \\ = & ((x\equiv_R)\,|;\exists\_\mathsf{in}(x|;\mathsf{eq})) & \{\text{definition } //\_\mathsf{in}\} & \\ = & (P|;e) & \{\text{by } p{:}(P = (x\equiv_R)), \text{ and proof equality}\} & \end{array}$$

Next, a proof for $tx = ty$ has to be inserted at ?2. Remark that we have $(x \equiv_R) = (y \equiv_R)$ by assumptions p and q, hence $x \equiv_R y$ follows from $y \equiv_R y$.

$$\begin{array}{lcll} & tx = ty & \\ \Leftarrow & \forall x, y{:}A :: (x \equiv_R y \Rightarrow tz = ty) & \{\text{because } x \equiv_R y\} \\ \Leftrightarrow & (\equiv_R) \subseteq Q & \{\text{taking } Q := ((x,y) :: tx = ty)\ \} \\ \Leftarrow & R \subseteq Q \wedge \mathsf{Equiv}\, Q & \{\text{definition } \equiv_R\} \\ \Leftrightarrow & \mathsf{True} & \{\text{by } d, \text{ and } Q \text{ being an equivalence}\} \end{array}$$

This completes the definition of //_elim.

Finally, (C.9) is derivable:

$$\begin{array}{lcll} & //\_\mathsf{in}_R\, x = //\_\mathsf{in}_R\, y & \\ \Leftarrow & (x \equiv_R) = (y \equiv_R) & \{\text{definition } //\_\mathsf{in}, \text{ and proof equality}\} \\ \Leftrightarrow & \forall z{:}A :: (x \equiv_R z) = (y \equiv_R z) & \{\text{extensionality}\} \\ \Leftarrow & x \equiv_R y & \{\equiv_R \text{ is an equivalence}\} \\ \Leftarrow & (x,y) \in R & \{R \subseteq (\equiv_R)\} \end{array}$$

C.4.5 A problem with quotients. Let's return to the basic system, without our strong existence rules. Rules for quotient types (C.7–C.11) may be (and have been) added as primitive rules. We present a specification that cannot be solved by these rules, presumably. The rules from section C.3, including the axiom of choice (C.5), do solve it.

Assume we have a quotient type $A//R$, where R is an equivalence relation, and a function f on infinite A-tuples that respects R:

$$\begin{array}{rcl} & & A{:}\mathbf{Type};\ R{:}\mathcal{P}A^2;\ \mathsf{Equiv}\, R \\ & & f{:}A^\omega \to A \\ u, v{:}A^\omega & \vdash & \forall (i{:}\omega :: (u_i, v_i) \in R) \Rightarrow (fu, fv) \in R \end{array}$$

(In relational notation, the latter property may be expressed as $(f,f) \in R^\omega \to R$.)

The problem is to construct a corresponding function on $A//R$:

$$f'{:}(A//R)^\omega \to A//R \text{ such that } \forall u{:}A^\omega :: f'(i :: //\_\mathsf{in}\, u_i) = //\_\mathsf{in}(fu)$$

We would naturally expect this to be possible as follows. Suppose we have a tuple $x{:}(A//R)^\omega$. Then:
1. For any $i{:}\omega$ there exists a $u{:}A$ with $x_i = //\_\mathsf{in}\, u$.
2. Thus there exists, by the axiom of choice, a tuple $y = (i :: y_i)$ with $x_i = //\_\mathsf{in}\, y_i$.
3. For such a y, one has $//\_\mathsf{in}(fy){:}A//R$.
4. The property of f says that the value of $//\_\mathsf{in}(fy){:}A//R$ is independent of the particular choice of the y_i — had we chosen other values z_i with $x_i = //\_\mathsf{in}\, z_i$, then would any z_i be in the same equivalence class as y_i, so $(y_i, z_i) \in R$ for all i, and hence $(fy, fz) \in R$, so $//\_\mathsf{in}(fy) = //\_\mathsf{in}(fz)$.
5. Thus, we can define $f'(x) := //\_\mathsf{in}(fy)$ where y is chosen as in step 2.

When we try to formalize this, we get stuck. The problem is that the quotient elimination rule (C.10) eliminates only a *single* element at a time. Repeated application works for a finite number of elements, but we have to eliminate an *infinite* number.

One might replace (C.10) with a stronger rule, but that would miss the point as the present rules already determine $A//R$ uniquely, up to isomorphism.

C.4.6 Our solution. We show how the $\exists$-rules together with the axiom of choice (C.5) solve the problem.

Assume $x\!:\!(A//R)^\omega$. We make the following steps, mirroring the proof above.

$$\begin{aligned}
&s1\!:\!\forall x\!:\!A//R :: \exists u\!:\!A :: x = //_\mathsf{in}\, u\\
&\qquad := \quad //_\mathsf{elim}(u :: \exists_\mathsf{in}(u|;\mathsf{eq}))\\
&s2\!:\!\exists\, \Pi(i\!:\!\omega :: \{u\!:\!A\,|\!:\, x_i = //_\mathsf{in}\, u\})\\
&\qquad := \quad \mathsf{ac}(i :: s1(x_i))\\
&s3\!:\!\Pi(i\!:\!\omega :: \{u\!:\!A\,|\!:\, x_i = //_\mathsf{in}\, u\}) \to A//R\\
&\qquad := \quad (y :: //_\mathsf{in}(f(i :: \mathsf{fst}\, y_i)))\\
&s4\!:\!\forall(y,z\!:\!\Pi(i\!:\!\omega :: \{u\!:\!A\,|\!:\, x_i = //_\mathsf{in}\, u\}) :: s3(y) = s3(z))\\
&\qquad := \quad ?1\\
&s5\!:\!A//R \;:= \quad s2^{\setminus}\, \exists_\mathsf{elim}(s3\,|;s4)
\end{aligned}$$

The skipped proof ?1 runs as follows, for given y, z:

$$\begin{array}{lcll}
 & s3(y) = s3(z) & \\
\Leftarrow & (f(i :: \mathsf{fst}\, y_i), f(i :: \mathsf{fst}\, z_i)) \in R & \{\text{by (C.9)}\}\\
\Leftarrow & \forall i\!:\!\omega :: (\mathsf{fst}\, y_i, \mathsf{fst}\, z_i) \in R & \{\text{for } f \text{ respects } R\}\\
\Leftarrow & \forall i :: \forall(u|;e),(v|;e')\!:\!\{u\!:\!A\,|\!:\, x_i = //_\mathsf{in}\, u\} :: (u,v) \in R & \{\text{Type of } y_i,\, z_i\}\\
\Leftarrow & \forall u,v\!:\!A;\, //_\mathsf{in}\, u = //_\mathsf{in}\, v :: (u,v) \in R &
\end{array}$$

To prove this last proposition, assuming $//\_\mathsf{in}\, u = //\_\mathsf{in}\, v$:

$$\begin{array}{lcll}
 & (u,v) \in R & \\
\Leftrightarrow & //_\mathsf{in}\, u^{\setminus}\, //_\mathsf{elim}(u :: (u,v) \in R\,|;?2) & \{\text{by } //\text{-reduction and ?2 below}\}\\
\Leftrightarrow & //_\mathsf{in}\, v^{\setminus}\, //_\mathsf{elim}(u :: (u,v) \in R\,|;?2) & \{\text{assumption}\}\\
\Leftrightarrow & (v,v) \in R & \{\text{by } //\text{-reduction}\}\\
\Leftrightarrow & \mathsf{True} & \{R \text{ is reflexive}\}
\end{array}$$

Finally, $?2\!:\!\forall(u,u')\!:\!\in R :: ((u,v) \in R) = ((u',v) \in R)$ is solvable by symmetry and transitivity of R, and propositions being equal when they are equivalent.

C.4.3 Inductive types

In set theory, the existence of a single infinite set ω suffices to construct all inductive sets, for example by the construction of Kerkhoff [45]. The new rules allow us to mirror this construction in our strong type theory, as is shown in section 8.2 of this thesis.

C.5 Conclusion

Set theory is embeddable in type theory, by defining a big type $\mathbf{Set}:\mathbf{Type}_1$ and a relation $(\in):\mathcal{P}\mathbf{Set}^2$ such that all ZF axioms are formally derivable. Such a model is defined in section A.5. This implies that type theory is stronger than ZF set theory (because of the extra universes), but only at the level of first order propositions about sets.

However, as to construct objects of other types, or to construct new types (within the universe $\mathbf{Type}_0$), there are constructions possible in set theory that cannot be done in current type theories. We have shown how the addition of a rule for strong proof elimination fills this deficiency. Some type constructors (quotients, inductive types) that have been proposed as primitives become derivable.

We have seen that, if one wishes to use the logic of type theory as a reduction system, our new principle resists the idea of proof irrelevance (section C.3.2). To avoid complications in the proof system, we suggested to distinguish terms as they occur in the proof system from reducible terms as they may be extracted from proofs.

Appendix D

Naturality of Polymorphism

From the type of a polymorphic object we derive a general uniformity theorem that the object must satisfy. We use a scheme for arbitrary inductive type constructors. Applications include natural and dinatural transformations, promotion and induction theorems. We employed the theorem in section 6.3 to prove equivalence between different recursion operators.

The issue was suggested to us by J.G. Hughes of Glasgow University who mentioned property (D.1) during a lecture in Groningen in October 1988. It was discussed in Backhouse's research club which led to our theorem. After sending Hughes an earlier version of our notes we received the draft of a paper [83] from Philip Wadler who derives the same main theorem as we do but in a more formal setting, and gives many promotion-like applications. His paper gave us some entries to the literature.

This is a revision of our report [14]. The notation has been partly adapted to this thesis, and the proofs of theorems D.2 and D.4 have been simplified through replacing inductive arguments by additional applications of the naturality theorem.

D.1 Introduction

Objects of a parametric polymorphic type in a polymorphic functional language like Miranda or Martin-Löf-like systems enjoy the property that instantiations to different types must have a similar behavior. To state this specifically, we use greek letters as type variables, and $T[\alpha]$ stands for a type expression possibly containing free occurrences of α. Stating $t{:}\,T[\alpha]$ means that term t has polymorphic type $T[\alpha]$ (in some implicit context), and hence $t{:}\,T[A]$ for any particular type A. Such instances are sometimes written with a subscript for clarity: $t_A{:}\,T[A]$. Thus the quantification $\forall\alpha$ is implicit. Some type expressions $T[\alpha]$ are *functorial* in α. I.e., there is a polymorphic expression $T[p]{:}\,T[\alpha] \to T[\beta]$ when $p{:}\,\alpha \to \beta$, such that $T[\mathsf{I}_A] = \mathsf{I}_{T[A]}$ and $T[p \mathbin{\bar{\circ}} q] = T[p] \mathbin{\bar{\circ}} T[q]$, where I is the identity function and $(\bar{\circ})$ denotes forward function composition. It has often been observed (e.g. [76]) that polymorphic functions $f{:}\,U[\alpha] \to V[\alpha]$, where U, V are functorial, must be natural transformations:

$$\text{For any } p{:}\,A \to A', \text{ one has } f_A \mathbin{\bar{\circ}} V[p] = U[p] \mathbin{\bar{\circ}} f_{A'} :\ U[A] \to V[A'] \qquad \text{(D.1)}$$

Illustration:

$$\begin{array}{ccc} U[A] & \xrightarrow{f_A} & V[A] \\ \downarrow{\scriptstyle U[p]} & & \downarrow{\scriptstyle V[p]} \\ U[A'] & \xrightarrow{f_{A'}} & V[A'] \end{array}$$

For example, any function $\mathsf{rev}\colon \alpha^* \to \alpha^*$, where α^* is the type of lists over α, must satisfy for $p\colon A \to A'$:

$$\mathsf{rev}_A \mathbin{\bar{\circ}} p^* = p^* \mathbin{\bar{\circ}} \mathsf{rev}_{A'}$$

Unfortunately, type expression $(U[\alpha] \to V[\alpha])$ is generally not functorial itself, because $(\to)$ is *contravariant* in its first argument. One can extend statement (D.1) to *dinatural transformations* in the sense of Mac Lane [51, pp. 214-218], but such a statement is still not provable by induction on the derivation of type correctness of f. In this appendix we develop a generalization to arbitrary types that is provable for all lambda-definable objects of some type. The generalization allows one to derive many properties of polymorphic objects from their type alone, properties which are conventionally proven by induction using the definition of the object, for example "promotion" theorems on functions like:

$$\mathsf{foldr}\colon (\alpha \times \beta \to \beta) \times \beta \to (\alpha^* \to \beta)$$

This promotion theorem says for $p\colon A \to A'$, $q\colon B \to B'$, $c\colon A \times B \to B$ and $c'\colon A' \times B' \to B'$, if

$$c \mathbin{\bar{\circ}} q = (p \times q) \mathbin{\bar{\circ}} c' : \;\; A \times B \to B'$$

then:

$$\mathsf{foldr}(c, b) \mathbin{\bar{\circ}} q = p^* \mathbin{\bar{\circ}} \mathsf{foldr}(c', qb) : \;\; A^* \to B'$$

If one identifies natural numbers with objects of polymorphic type $(\alpha \to \alpha) \to (\alpha \to \alpha)$, one can even derive Peano's induction axiom.

The essential theorem is in fact Reynolds' abstraction theorem [75]. (The inconsistency in his modelling of polymorphic objects as set-theoretic functions on the class of all types is rather irrelevant.) The basic idea had already been given by Plotkin [74], and a more complicated variant is formed by the logical relations of Mitchell and Meyer [62]. It was generally regarded as merely a representation independence theorem for datatype implementations, while its implications for deriving properties of functional programs seem to have been unrecognized at that time. Quite different approaches are used in Bainbridge, Freyd et al. [9, 32], based on dinatural transformations in a category called **LIN** of certain coherent spaces and linear maps, and in Carboni et al. [17] where the so-called Realizability Universe is constructed. Both approaches appear to be conceptually far more complicated than Reynolds'. There is also a paper by John Gray [36] which deals only with naturality of some particular operations like currying.

Our contribution consists of the inclusion of arbitrary initial types, some applications of a different kind than Wadler's applications, a very attractive proof of the dinaturality property, and a proof of equivalence between two different recursion operators. We expect the theorem to hold for generalized typed lambda calculus too, but leave this to further research.

Survey. In section D.2 we shall define a typed lambda calculus, and in D.3 we show how type constructors correspond to relation constructors. In D.4 we derive the main naturality theorem; in D.5 we give some simple applications; in D.6 we derive a dinaturality result which could not be proven directly. In D.7 we add second-order quantification and derive mathematical induction for the encoded natural numbers. In D.8 we see how polymorphism over types that support certain operations can be treated. For a final application, proving the equivalence between categorical recursion and Mendler recursion, we refer to theorem 6.4 in this thesis.

D.2 Polymorphic typed lambda calculus

The proof of the naturality theorem is by induction on the derivation of the type of an object. So we need to specify the derivation rules of the polymorphic functional language that we shall use. This language may be either a programming language with fixpoints, where the semantic domain of a type is a cpo, or a purely constructive language where types are flat sets and all functions are total.

We will use typed lambda calculus. The syntax for types and for terms is:

$$\begin{aligned} T &::= \alpha \mid (T \to T) \mid \Theta\, T^* \ . \\ t &::= x \mid (t\, t) \mid \boldsymbol{\lambda} x.t \mid \theta_j \mid \Theta\_\mathrm{elim}(t^*) \ . \end{aligned}$$

Besides type variables α, we have $(\to)$ as the function type constructor, and an unspecified number of other type constructors Θ, each one constructing from a sequence of types U of some fixed length the least datatype that is closed under a number of object constructors θ_j. Each constructor θ_j has a sequence of arguments of types $F_{jl}[U, \Theta U]$. The types $F_{jl}[\alpha, \beta]$ must be functorial in β, so for $q\colon B \to B'$ we have $F_{jl}[A, q]\colon F_{jl}[A, B] \to F_{jl}[A, B']$. The functions $F_{jl}[A, q]$ are required to preserve relations too (D.3), but this is guaranteed by naturality. We use a categorical elimination construct Θ_elim (compare Hagino [37]), from which other eliminators may be defined.

Note that we often write a single meta-variable, like 'U', to stand for a sequence, '$U_0, \ldots, U_{n-1}$'. Furthermore, a type expression '$U \to V$', where U is a sequence, stands for '$U_0 \to \cdots (U_{n-1} \to V)$'.

The derivability judgement 't has type T under the assumptions $x_j\colon S_j$' is noted $x\colon S \vdash t\colon T$, and is generated by the following rules:

$$\begin{array}{rcll} & & x\colon S \vdash x_j\colon S_j & \{\text{Var-intro}\} \\ x\colon S \vdash f\colon U \to V;\ x\colon S \vdash u\colon U & \Rightarrow & x\colon S \vdash (f\, u)\colon V & \{(\to)\text{-elim}\} \\ x\colon S, y\colon U \vdash v\colon V & \Rightarrow & x\colon S \vdash \boldsymbol{\lambda} y.v\colon U \to V & \{(\to)\text{-intro}\} \\ & & x\colon S \vdash \theta_j\colon F_j[U, \Theta U] \to \Theta U & \{\Theta\text{-intro}\} \\ x\colon S \vdash v_j\colon F_j[U, V] \to V \ (\text{each } j) & \Rightarrow & x\colon S \vdash \Theta\_\mathrm{elim}(v)\colon \Theta U \to V & \{\Theta\text{-elim}\} \end{array}$$

There is an untyped congruence relation $(==)$ on terms, called *conversion*, which is generated by:

$$\begin{aligned} (\boldsymbol{\lambda} y.v\, u) &== v[y := u] \\ (\Theta\_\mathrm{elim}(v)\,(\theta_j d)) &== (v_j\,(F_j[\alpha, \Theta\_\mathrm{elim}(v)]\, d)) \end{aligned}$$

We will define a typed extensional equivalence in section D.3. In section D.7 we will add second-order quantification (in terms of which initial and final datatypes can be defined).

Note that we derive the theorem using only lambda terms, without any reference to models. One can derive the same results as we do in an arbitrary model, see Wadler [83].

D.3 Turning type constructors into relation constructors

Observe that, although we cannot extend a function p from A to A' to a function from $(U[A] \to V[A])$ to $(U[A'] \to V[A'])$, property (D.1) suggests us to consider the binary relation between $f\colon U[A] \to V[A]$ and $f'\colon U[A'] \to V[A']$ that is given by $f \mathbin{\bar{\circ}} V[p] = U[p] \mathbin{\bar{\circ}} f'$. We will use this observation to extend a relation (instead of a function) between A and A' to one between $T[A]$ and $T[A']$.

Definition. A *relation* $R\colon \subseteq A \times A'$ is a set of pairs of terms of types A and A', taken modulo conversion. (Had we used a programming language permitting the construction of non-terminating programs then we would use elements of the corresponding cpo's and R would be required to be closed under directed limits: if $V \subseteq R$ is (pairwise) directed, then $\bigsqcup V \in R$. In particular, $(\bot_A, \bot_{A'}) \in R$ as $\bot$ is the limit of the empty set.)
While we use a colon $(:)$ for typing, $(\in)$ denotes relation-membership.

Definition. We will lift any type-constructor Θ to a relation-constructor such that for any relation sequence $R\colon \subseteq A \times A'$ (i.e. $R_i\colon \subseteq A_i \times A'_i$) one has:

$$\Theta R\colon \subseteq \Theta A \times \Theta A'$$

First, as $(A \to B)$ is the greatest type such that for $f\colon A \to B$ one has $\forall x\colon A \,.\, fx\colon B$, we define for $Q\colon \subseteq A \times A'$, $R\colon \subseteq B \times B'$:

$$Q \to R \quad := \quad \left\{(f, f')\colon (A \to B) \times (A' \to B') \mid \colon \forall (x, x')\colon \in Q \,.\, (fx, f'x') \in R\right\} \tag{D.2}$$

That is to say, the pair of functions should map related arguments to related results, as illustrated by:

$$\begin{array}{ccc} A & \xrightarrow{\quad f \quad} & B \\ \wr\, Q & & \wr\, R \\ A' & \xrightarrow{\quad f' \quad} & B' \end{array}$$

If $\Theta\alpha$ is the initial type that is closed under object-constructors

$$\theta_j\colon F_j[\alpha, \Theta\alpha] \to \Theta\alpha$$

where each F_{jl} may be interpreted as a relation constructor that satisfies for relations $P, Q\colon \subseteq \Theta A \times \Theta A'$, $R\colon \subseteq A \times A'$

$$\forall (g, g')\colon \in P \to Q \,.\, (F_{jl}[A, g], F_{jl}[A', g']) \in F_{jl}[R, P] \to F_{jl}[R, Q]\,, \tag{D.3}$$

and preserves identity, then we define ΘR to be the least relation that is closed under:

$$(\theta_j, \theta_j) \in F_j[R, \Theta R] \to \Theta R \tag{D.4}$$

This makes sense since for $P \subseteq Q$ we have $F_j[R, P] \subseteq F_j[R, Q]$ by (D.3), taking g and g' to be the identity I.

Thus, the induction principle for ΘR is: if we wish to prove a predicate P for all pairs in ΘR, then, considering P as a relation $P{:}\subseteq \Theta A \times \Theta A'$, we must show for each j:

$$\forall (d,d'){:}\in F_j[R,P]\,.\,(\theta_j d, \theta_j d') \in P \tag{D.5}$$

Check for example that all pairs in ΘR are convertible to $(\theta_j d, \theta_j d')$ for some j and $(d,d'){:}\in F_j[R,\Theta R]$

Example. Some relation-constructors corresponding to common type-constructors are

$$\begin{array}{rcl} Q+R & := & \{(x,x'){:}\in Q :: (\mathsf{inl}(x),\mathsf{inl}(x'))\,\} \\ & & \cup\,\{(y,y'){:}\in R :: (\mathsf{inr}(y),\mathsf{inr}(y'))\,\} \\ Q\times R & := & \{(x,x'){:}\in Q,(y,y'){:}\in R :: ((x,y),(x',y'))\,\} \\ \mathsf{Bool} & := & \{(\mathsf{true},\mathsf{true}),(\mathsf{false},\mathsf{false})\} \end{array}$$

and $\mathbb{N}$ and R^* are the least relations such that:

$$\begin{array}{rcl} \mathbb{N} & = & \{(0,0)\}\cup\{(z,z'){:}\in \mathbb{N} :: (\mathsf{s}(z),\mathsf{s}(z'))\,\} \\ R^* & = & \{(\mathsf{nil},\mathsf{nil})\}\cup\{(x,x'){:}\in R,(z,z'){:}\in R^* :: (x +\!\!< z, x' +\!\!< z')\,\} \end{array}$$

Now, for any type-expression $T[\alpha]$ containing only type-variables from the sequence α, we have extended a relation-sequence $R{:}\subseteq A\times A'$ to a relation $T[R]{:}\subseteq T[A]\times T[A']$. However, T is not necessarily a functor on relations, for it need not preserve composition.

Notice that the same schemes (D.2) and (D.4) describe extensional equality on $A\to B$ and ΘA in terms of the equality on the A and B. Thus we can use the relational interpretation of a type to define extensional equality:

Definition. Extensional equality on a closed type T is given by $T[]$ as a relation:

$$(\models t = t'{:}T) \quad := \quad (t,t')\in T[]$$

We will often write just $t=t'$ rather than $\models t=t'{:}T$. We shall consider only relations that are closed under this extensional equality. Equality for terms of types containing variables will be defined in the next section.

D.4 Naturality of expressions

If we can derive $t{:}T[\alpha]$ then not only $t_A{:}T[A]$ for any type-sequence A, by an appropriate substitution theorem, but also $(t_A,t_{A'})\in T[R]$ for any relation-sequence $R{:}\subseteq A\times A'$. (It is to be understood that *overloaded* operators, like the effective equality-test $(==_A){:}A\times A\to\mathsf{Bool}$ in Miranda, may not be used as if they where polymorphic.) Taking the context into account, we have the following main theorem, similar to the abstraction theorem in [75], the fundamental theorem of Logical Relations in [62], and the parametricity result in [83]:

Theorem D.1 (Naturality) *If $x{:}S[\alpha]\vdash t[x]{:}T[\alpha]$ then for any sequences A,A',R,s,s', where $R{:}\subseteq A\times A'$ respects extensional equality, and $(s_j,s'_j)\in S_j[R]$, one has:*

$$(t_A[s],t_{A'}[s'])\in T[R]$$

Remark: we say that $t[x]$ is *natural*. The theorem may be generalized to relations of arbitrary arity.

Proof. By a straightforward induction on the derivation of $x{:}\,S[\alpha] \vdash t[x]{:}\,T[\alpha]$. We check all rules:

Var-intro. The judgement is $x{:}\,S[\alpha] \vdash x_j{:}\,S_j[\alpha]$. By assumption we have $(s_j, s'_j) \in S_j[R]$ indeed.

$(\rightarrow)$-elim. The hypotheses say $(f[s], f[s']) \in U[R] \rightarrow V[R]$ and $(u[s], u[s']) \in U[R]$. Then by definition of $(\rightarrow)$ on relations we obtain:

$$(f[s]u[s], f[s']u[s']) \in V[R]$$

$(\rightarrow)$-intro. The hypothesis for the premise $x{:}\,S[\alpha], y{:}\,U[\alpha] \vdash v[x,y]{:}\,V[\alpha]$ says that for any $(s,s'){:}\in S[R]$ and $(u,u'){:}\in U[R]$ one has $(v[s,u], v[s',u']) \in V[R]$.
So $((\boldsymbol{\lambda} y.v[s,y])u, (\boldsymbol{\lambda} y.v[s',y])u') \in V[R]$ as relations are closed under conversion. Hence:

$$(\boldsymbol{\lambda} y.v[s,y], \boldsymbol{\lambda} y.v[s',y]) \in U[R] \rightarrow V[R]$$

Θ-intro. We must show:

$$(\theta_j, \theta_j) \in F_j[U[R], \Theta U[R]] \rightarrow \Theta U[R]$$

This is an instance of (D.4).

Θ-elim. The (global) hypothesis is: $(v_j[s], v_j[s']) \in F_j[U[R], V[R]] \rightarrow V[R]$ for each j. We must show:

$$(\Theta\_\mathrm{elim}(v[s]), \Theta\_\mathrm{elim}(v[s'])) \in \Theta U[R] \rightarrow V[R]$$

We use a local induction on the generation of $\Theta U[R]$. Thus we will prove $\Theta U[R] \subseteq P$ where:

$$P \;:=\; \{(t,t'){:}\,\Theta U[A] \times \Theta U[A'] \mid{:}\, (\Theta\_\mathrm{elim}(v[s])t, \Theta\_\mathrm{elim}(v[s'])t') \in V[R]\}$$

Note that:

$$(\Theta\_\mathrm{elim}(v[s]), \Theta\_\mathrm{elim}(v[s'])) \in P \rightarrow V[R] \qquad \text{(D.6)}$$

We check (D.5) for each j:

$$\begin{array}{lcll}
 & (d,d') \in F_j[U[R], P] & \\
\Rightarrow & (F_j[\alpha, \Theta_\mathrm{elim}(v[s])]d, F_j[\alpha, \Theta_\mathrm{elim}(v[s'])]d') \in F_j[U[R], V[R]] & \{\text{(D.3) on (D.6)}\} \\
\Rightarrow & (v_j[s](F_j[\alpha, \Theta_\mathrm{elim}(v[s])]d), v_j[s'](F_j[\alpha, \Theta_\mathrm{elim}(v[s'])]d')) \in V[R] & \{\text{global hyp.}\} \\
\Leftrightarrow & (\Theta_\mathrm{elim}(v[s])(\theta_j d), \Theta_\mathrm{elim}(v[s'])(\theta_j d')) \in V[R] & \{\text{conversion}\} \\
\Leftrightarrow & (\theta_j d, \theta_j d') \in P & \{\text{def. } P\}
\end{array}$$

■

The theorem suggests the following definition:

Definition. Extensional equality under a sequence of assumptions $x_j : S_j$, noted

$$x : S[\alpha] \;\models\; t[x] = t'[x] : T[\alpha] \;,$$

holds iff, for all sequences A, A'; $R :\subseteq A \times A'$; $(s, s') :\in S[R]$ one has $(t_A[s], t'_{A'}[s']) \in T[R]$.

Convention. All of the following definitions and theorems may be taken in the context of a set of assumptions, and all terms should be taken modulo extensional equality under these assumptions.

Definition. A type expression $T[\alpha]$ is called *functorial (in α)* if there is an expression $T[p]$ that satisfies

$$p : \alpha \to \beta \vdash T[p] : T[\alpha] \to T[\beta] \tag{D.7}$$

and that preserves identity, $\models T[\mathsf{I}] = \mathsf{I} : \alpha \to \alpha$. We shall shortly proof that because of naturality, T preserves composition too.

Many applications arise by using a sequence of function-like relations:

Definition. For $p : A \to A'$ let the *graph* of p be:

$$(p) \;:=\; \{ x : A :: (x, px) \}$$

A relation $R :\subseteq A \times A'$ is called *function-like* if $R = (p)$ for some $p : A \to A'$. (Note that, in a cpo, (p) is closed under directed limits iff p is continuous and strict, i.e. $p\bot_A = \bot_{A'}$.)

Fact. For function-like relations we have, if $p : A \to A'$, $q : B \to B'$:

$$(p) \to (q) \;=\; \{ (f, f') \mid : f \mathbin{\bar{\circ}} q = p \mathbin{\bar{\circ}} f' : A \to B' \} \tag{D.8}$$

$$(p)^{\cup} \to (q) \;=\; \{ f : A \to B :: (f,\; p \mathbin{\bar{\circ}} f \mathbin{\bar{\circ}} q) \} \tag{D.9}$$

$$\text{using } R^{\cup} \;:=\; \{ (x, y) :\in R :: (y, x) \}$$

Also useful might be, for $f : A' \to B$:

$$(p \mathbin{\bar{\circ}} f,\; f \mathbin{\bar{\circ}} q) \in (p) \to (q) \tag{D.10}$$

For functorial type expressions, the functorial interpretation must coincide with the relational interpretation:

Theorem D.2 *If $T[\alpha]$ is functorial, then for any $p : A \to B$,*

$$(T[p]) = T[(p)] \;.$$

Proof. We derive:

$$\begin{array}{lcll}
 & (\mathsf{I}, p) \in (\mathsf{I}_A) \to (p) & \{\text{(D.8)}\} \\
\Rightarrow & (T[\mathsf{I}], T[p]) \in T[(\mathsf{I}_A)] \to T[(p)] & \{\text{naturality } T\} \\
\Leftrightarrow & (\mathsf{I}, T[p]) \in (\mathsf{I}_{T[A]}) \to T[(p)] & \{\text{preservation of identity}\} \\
\Leftrightarrow & (T[p]) \subseteq T[(p)] & \{\text{definition } (T[p]), \to\}
\end{array}$$

and:

$$\begin{array}{lcll} & (p, \mathsf{I}) \in (p) \to (I_B) & \{\text{(D.8)}\} \\ \Rightarrow & (T[p], T[\mathsf{I}]) \in T[(p)] \to T[(\mathsf{I}_B)] & \{\text{naturality } T\} \\ \Leftrightarrow & (T[p], \mathsf{I}) \in T[(p)] \to (\mathsf{I}_{T[B]}) & \{\text{preservation of identity}\} \\ \Leftrightarrow & T[(p)] \subseteq (T[p]) & \{\text{definition } (T[p]), \to\} \end{array}$$

■

So we have, for example, $(p \times q) = (p) \times (q)$ and can safely omit the parentheses. Notice that $p \to q$ can only be read as $(p) \to (q)$, for $\alpha \to \beta$ is not functorial.

Theorem D.3 *Any functorial $T[\alpha]$ must preserve composition, i.e., if $p\colon A \to B$ and $q\colon B \to C$, in some context, then $\models T[p \mathbin{\bar{\circ}} q] = T[p] \mathbin{\bar{\circ}} T[q]\colon A \to C$.*

Proof.

$$\begin{array}{lcll} & (p \mathbin{\bar{\circ}} q, q) \in (p) \to (\mathsf{I}_C) & \{\text{(D.8)}\} \\ \Rightarrow & (T[p \mathbin{\bar{\circ}} q], T[q]) \in T[(p)] \to T[(\mathsf{I}_C)] & \{\text{naturality } T\} \\ \Leftrightarrow & (T[p \mathbin{\bar{\circ}} q], T[q]) \in (T[p]) \to (\mathsf{I}_{T[C]}) & \{\text{theorem D.2}\} \\ \Leftrightarrow & T[p \mathbin{\bar{\circ}} q] = T[p] \mathbin{\bar{\circ}} T[q] & \{\text{(D.8)}\} \end{array}$$

■

The naturality theorem specializes for polymorphic $t\colon T[\alpha]$ and $p\colon A \to A'$ to:

$$(t_A, t_{A'}) \in T[(p)] \tag{D.11}$$

So if $f\colon U[\alpha] \to V[\alpha]$, and hence $(f, f) \in U[p] \to V[p]$ by (D.11), then f is a natural transformation indeed.

D.5 Applications

Example D.1 A simple application is to prove that $f = \boldsymbol{\lambda} x.x$ is the only polymorphic function of type $f\colon \alpha \to \alpha$. Naturality of f says: for any $R\colon \subseteq A \times A'$ we have $(f, f) \in R \to R$.
Now, fix type A and $a\colon A$. Taking $R := \{(a, a)\}$ yields $(fa, fa) \in R$, as $(a, a) \in R$. So $fa = a$ for all a, hence $f = \boldsymbol{\lambda} x.x$ by extensionality of functions.

(In an alternative language where types are cpo's there are two solutions. If $a \neq \bot$, we must take $R := \{(\bot, \bot), (a, a)\}$ and get $fa \in \{\bot, a\}$. Furthermore, for any $b\colon B$ we can get $(fa, fb) \in \{(\bot, \bot), (a, b)\}$. So in case $fa = \bot$ we have $fb = \bot$ for all b, hence $f = \boldsymbol{\lambda} x.\bot$; and in case $fa = a$ we obtain $fb = b$ and hence $f = \boldsymbol{\lambda} x.x$.)

Example D.2 Let * be a (postfix) functor, say of lists, so for $p\colon A \to A'$ we have $p^*\colon A^* \to A'^*$ and $(p^*) = (p)^*$. Let f be a function with the type of foldr, i.e.:

$$f\colon (\alpha \times \beta \to \beta) \times \beta \to (\alpha^* \to \beta)$$

Naturality of f on function-like relations says: if $p\colon A \to A'$, $q\colon B \to B'$ then

$$(f,f) \in (p \times q \to q) \times q \to (p^* \to q)$$

i.e. if $(c,c')\colon\in (p \times q \to q)$ and $(b,b')\colon\in (q)$ then $(f(c,b), f(c',b')) \in (p^* \to q)$.
Using (D.8) this equivales: if

$$c \mathbin{\bar{\circ}} q = (p \times q) \mathbin{\bar{\circ}} c' : \;\; A \times B \to B'$$

then:

$$f(c,b) \mathbin{\bar{\circ}} q = p^* \mathbin{\bar{\circ}} f(c', qb) : \;\; A^* \to B'$$

(This is a generalization of the promotion-theorem for forward lists [52].) Notice that the result is independent of the definition of f.

In particular, we have for $\oplus\colon A' \times B' \to B$, as by (D.10) $((p \times q) \bar{\circ} \oplus,\ \oplus \bar{\circ} q) \in (p \times q \to q)$:

$$f((p \times q) \mathbin{\bar{\circ}} \oplus,\ b) \mathbin{\bar{\circ}} q = p^* \mathbin{\bar{\circ}} f(\oplus \mathbin{\bar{\circ}} q,\ qb) \tag{D.12}$$

Another instance yields, as $+\!\!< \mathbin{\bar{\circ}} \mathsf{foldr}(c',b') = (\mathsf{I} \times \mathsf{foldr}(c',b')) \mathbin{\bar{\circ}} c'$ and $\mathsf{foldr}(c',b')\,\mathsf{nil} = b'$:

$$f(+\!\!<, \mathsf{nil}) \mathbin{\bar{\circ}} \mathsf{foldr}(c',b') = f(c',b')$$

Example D.3 [1] Finally, we give an application using *ternary* relations. Let $(^+)$ be a functor, say of non-empty lists, and $(/_B)$ a (polymorphic) mapping of operators $\oplus\colon B \times B \to B$ into $\oplus/\colon B^+ \to B$.

We will prove: if $f, g\colon A \to B$, and $\oplus\colon B \times B \to B$ is commutative and associative, then for $l\colon A^+$,

$$\oplus/(f^+ l) \;\oplus\; \oplus/(g^+ l) \;=\; \oplus/((f \oplus^A g)^+ l)$$

where $\oplus^A\colon (A \to B) \times (A \to B) \to (A \to B)$ is the lifted version of $\oplus$. We will regard $\oplus$ as a ternary relation $\oplus\colon \subseteq B \times B \times B$ so that:

$$(x,y,z) \in \oplus \quad := \quad x \oplus y = z$$

We derive:

$$\begin{array}{ll}
 & \forall l\colon A^+ \,.\, (\oplus/(f^+ l),\ \oplus/(g^+ l),\ \oplus/((f \oplus^A g)^+ l)) \in \oplus \\
\Leftarrow & \quad (f^+, g^+, (f \oplus^A g)^+) \in A^+ \to \oplus^+ \\
 & \quad \wedge\ (\oplus/, \oplus/, \oplus/) \in \oplus^+ \to \oplus \\
\Leftarrow & \quad (f, g, (f \oplus^A g)) \in A \to \oplus \qquad\qquad \{\text{naturality of } (^+), (/)\ \} \\
 & \quad \wedge\ (\oplus, \oplus, \oplus) \in \oplus \times \oplus \to \oplus \\
\Leftrightarrow & \forall x\colon A \,.\, fx \oplus gx = (f \oplus^A g)x \\
 & \wedge\ \forall (x_j, y_j, z_j)\colon\in \oplus \,.\, (x_0 \oplus x_1, y_0 \oplus y_1, z_0 \oplus z_1) \in \oplus \\
\Leftrightarrow & \mathsf{true} \\
 & \wedge\ \forall x_j, y_j \,.\, (x_0 \oplus x_1) \oplus (y_0 \oplus y_1) = (x_0 \oplus y_0) \oplus (x_1 \oplus y_1) \\
\Leftarrow & \quad \oplus \text{ is commutative and associative}
\end{array}$$

[1] Suggested by Roland Backhouse

D.6 Dinatural transformations

Mac Lane [51] defines "dinatural transformations". A result by Backhouse [6, section 6] on a dinaturality property in relational calculus inspired us to the following derivation of dinaturality for polymorphic objects from general naturality. (Backhouse' theorem, second half, bears a relationship to our property (D.15), written as $T[p \parallel p] \cdot T[p \parallel \mathsf{I}] \subseteq T[\mathsf{I} \parallel p)$.)

Definition. A *difunctor* $T[\alpha \parallel \beta]$ is given by a type $T[\alpha \parallel \beta]$ and a term $T(x \parallel y)$ typed by

$$x{:}\,\alpha \to \alpha';\ y{:}\,\beta \to \beta' \vdash T(x \parallel y){:}\,T[\alpha' \parallel \beta] \to T[\alpha \parallel \beta'] \tag{D.13}$$

(note the contravariance in α) such that identity is respected.

Take care not to confuse, for $p{:}\,A \to A'$, the term $T(p \parallel p){:}\,T[A' \parallel A] \to T[A \parallel A']$ with the relation $T[p \parallel p]{:}\,\subseteq T[A \parallel A] \times T[A' \parallel A']$.

Normally, any type expression $T[\alpha]$ can be written as a difunctor such that $T[\alpha] = T[\alpha \parallel \alpha]$, by separating all covariant and contravariant type variable occurrences, as follows.

If $T[\alpha] = \alpha_i$, take $T[\alpha \parallel \beta] := \beta_i$ and $T(x \parallel y) := y_i$.

If $T[\alpha] = U[\alpha] \to V[\alpha]$, a difunctor is given by:

$$\begin{array}{rcl} T[\alpha \parallel \beta] & := & U[\beta \parallel \alpha] \to V[\alpha \parallel \beta] \\ T(x \parallel y) & := & U(y \parallel x) \circ\!\!\to V(x \parallel y) \\ \text{using } u \circ\!\!\to v & := & \boldsymbol{\lambda} f.(u \mathbin{\bar{\circ}} f \mathbin{\bar{\circ}} v) \end{array}$$

Remark that, as a relation, $(u \circ\!\!\to v) = (u^{\cup}) \to (v)$ by (D.9).

If $T[\alpha] = \Theta U[\alpha]$, one needs a difunctorial type constructor $\Theta'(\alpha \parallel \beta)$ such that $\Theta U = \Theta'(U \parallel U)$. Normally, if Θ is already functorial one can take just $\Theta'(\alpha \parallel \beta) := \Theta\beta$, otherwise the language is required to contain such a constructor. Then one takes:

$$\begin{array}{rcl} T[\alpha \parallel \beta] & := & \Theta'(U[\beta \parallel \alpha] \parallel U[\alpha \parallel \beta]) \\ T(x \parallel y) & := & \Theta'(U[y \parallel x] \parallel U(x \parallel y)) \end{array}$$

Theorem D.4 (dinaturality) *When $\vdash t{:}\,T[\alpha]$ where T can be written as a difunctor, $T[\alpha] = T[\alpha \parallel \alpha]$, then for any $p{:}\,A \to A'$ one has:*

$$\vdash T(\mathsf{I} \parallel p)\, t_A = T(p \parallel \mathsf{I})\, t_{A'} \tag{D.14}$$

Proof. Naturality of $T(x \parallel y)$ as typed by (D.13) gives, using $(\mathsf{I}, p) \in \mathsf{I}_A \to p$ and $(p, \mathsf{I}) \in p \to \mathsf{I}_{A'}$:

$$(T(\mathsf{I} \parallel p),\, T(p \parallel \mathsf{I})) \ \in\ T[p \parallel p] \to T[\mathsf{I}_A \parallel \mathsf{I}_{A'}]\ . \tag{D.15}$$

Naturality of $t{:}\,T[\alpha \parallel \alpha]$ gives:

$$(t_A, t_{A'}) \in T[p \parallel p]\ .$$

Together this gives $(T(\mathsf{I} \parallel p)\, t_A,\, T(p \parallel \mathsf{I})\, t_{A'}) \in T[\mathsf{I}_A \parallel \mathsf{I}_{A'}]$. As T preserves identity relations, we obtain (D.14). ∎

Corollary D.5 *All polymorphic functions $f: U[\alpha] \to V[\alpha]$, where U and V can be written as difunctors, are* dinatural transformations. *That is to say, they satisfy for $p: A \to A'$:*

$$U(p \parallel \mathsf{I}) \mathbin{\bar{\circ}} f_A \mathbin{\bar{\circ}} V(\mathsf{I} \parallel p) = U(\mathsf{I} \parallel p) \mathbin{\bar{\circ}} f_{A'} \mathbin{\bar{\circ}} V(p \parallel \mathsf{I}) : \;\; U[A' \parallel A] \to V[A \parallel A']$$

Example D.4 Any function

$$f: (\alpha \times \beta \to \beta) \times \beta \to (\alpha^* \to \beta)$$

will satisfy for $p: A \to A'$, $q: B \to B'$:

$$((p \times q \circ\!\!\to \mathsf{I}) \times \mathsf{I}) \mathbin{\bar{\circ}} f_{AB} \mathbin{\bar{\circ}} (\mathsf{I}^* \circ\!\!\to q) = ((\mathsf{I} \times \mathsf{I} \circ\!\!\to q) \times q) \mathbin{\bar{\circ}} f_{A'B'} \mathbin{\bar{\circ}} (p^* \circ\!\!\to \mathsf{I})$$

Applied to some $(\oplus, b): (A' \times B' \to B) \times B$, this is our result (D.12) in section D.5.

D.7 Second-order languages

We may use a second-order language, where, say, $\forall\alpha.T[\alpha]$ is a type with:

$$\begin{array}{rcl} x: S \vdash t: T[\alpha], \text{ where } \alpha \text{ does not occur free in } S & \Rightarrow & x: S \vdash t: \forall\alpha.T[\alpha] \\ x: S \vdash t: \forall\alpha.T[\alpha] & \Rightarrow & x: S \vdash t: T[U] \end{array}$$

The appropriate extension of relations is, if $R[Q]: \subseteq T[A] \times T[A']$ for any $Q: \subseteq A \times A'$ (that is closed under extensional equality):

$$\forall\rho.R[\rho] \;:=\; \{\, (t,t') \mid: \forall A, A': \mathbf{Type}. \forall Q: \subseteq A \times A' \,.\, (t,t') \in R[Q] \}$$

As an application, we define type $\mathbb{N}$, $\mathbf{z}: \mathbb{N}$ and $\mathbf{s}: \mathbb{N} \to \mathbb{N}$ by:

$$\begin{array}{rcl} \mathbb{N} & := & \forall\alpha.((\alpha \to \alpha) \to (\alpha \to \alpha)) \\ \mathbf{z} & := & \boldsymbol{\lambda} f.\boldsymbol{\lambda} a.a \\ \mathbf{s} & := & \boldsymbol{\lambda} m.\boldsymbol{\lambda} f.\boldsymbol{\lambda} a.f(mfa) \end{array}$$

We will prove Peano's induction-axiom.

Theorem D.6 *When for $P: \subseteq \mathbb{N}$ one has*

$$\mathbf{z} \in P \;\wedge\; \forall m: \in P :: \mathbf{s}m \in P$$

then $n \in P$ for all $n: \mathbb{N}$.

Proof. Remember that naturality of n says that for any types A, A', and relation $Q: \subseteq A \times A'$, we have $(n,n) \in (Q \to Q) \to (Q \to Q)$.
The proof is in two steps.

1. Take a *predicate-like* relation $Q := \{n: \in P :: (n,n)\}$. The assumptions say $(\mathbf{s},\mathbf{s}) \in (Q \to Q)$ and $(\mathbf{z},\mathbf{z}) \in Q$, hence by naturality of n we get $(n\mathbf{s}, n\mathbf{s}) \in (Q \to Q)$ and $(n\mathbf{sz}, n\mathbf{sz}) \in Q$, i.e. $n\mathbf{sz} \in P$.
2. What remains to prove is $n\mathbf{sz} = n$, i.e. for any type A, $f: A \to A$, $a: A$ we must prove $n\mathbf{sz}fa = nfa$.
   Taking $Q := \{(m,x): \mathbb{N} \times A \mid: mfa = x\}$, naturality guarantees $(n\mathbf{sz}, nfa) \in Q$ provided $(\mathbf{s}, f) \in (Q \to Q)$ and $(\mathbf{z}, a) \in Q$. But these properties hold by definition of $\mathbf{s}$ and $\mathbf{z}$. ■

D.8 Overloaded operators

We remarked that polymorphic functions may not use overloaded operators, like an effective equality-test $(==_A): A \times A \to$ `bool` that is defined only for some types A. However, if we require types instantiated for type-variables to support certain operations, we can give similar requirements on relations. Such restrictions may be provided explicitly by a "type class" in the language Haskell [38].

Definition. Let z be the class of types α with associated operations $v_i: T_i[\alpha]$, and let A and A' be two "instances" of z with operations $t_i: T_i[A]$ and $t'_i: T_i[A']$. The same written in Haskell:

$$\begin{array}{l}\texttt{class}\ z\ \alpha\ \texttt{where}\ \{\ v_1\ ::\ \ T_1[\alpha]\ ;;\ \ldots ;;\ v_n\ ::\ \ T_i[\alpha]\ \}\\ \texttt{instance}\ z\ A\ \texttt{where}\ \{\ v_1\ \texttt{=}\ t_1\ ;;\ \ldots ;;\ v_n\ \texttt{=}\ t_n\ \}\\ \texttt{instance}\ z\ A'\ \texttt{where}\ \{\ v_1\ \texttt{=}\ t'_1\ ;;\ \ldots ;;\ v_n\ \texttt{=}\ t'_n\ \}\end{array}$$

A relation $R: \subseteq A \times A'$ is said to *respect* class z, iff for each i, one has $(t_i, t'_i) \in T_i[R]$.

For example, consider the class `Eq` of types `a` with equality-test:

```
class Eq a where (==) ::  a -> a -> Bool
```

Relation R respects `Eq` iff for all $(x, x'): \in R$, $(y, y'): \in R$ one has $(x == y) = (x' == y')$: `Bool`. Note that not all relations have this property, hence $(==)$ is not natural. But one can prove the following variant:

Restricted naturality. If expression s has type $S[\alpha]$ for any instance α of class z as above, which is expressed in Haskell by

$$s\ ::\ \ z\ \alpha\ \texttt{=>}\ S[\alpha]$$

then for any relation $R: \subseteq A \times A'$ that respects z we have that $(s, s) \in S[R]$.

Acknowledgement. Thanks are due to Roland Backhouse and Wim Hesselink for many comments that greatly improved upon our presentation.

Index