Isabelle/CTT — Constructive Type Theory with extensional equality and without universes

Larry Paulson

March 13, 2025

Contents

theory *CTT*
imports *Pure*
begin

1 Constructive Type Theory: axiomatic basis

⟨*ML*⟩

typedecl *i*
typedecl *t*
typedecl *o*

consts

```
  — Judgments
  Type     :: t ⇒ prop          (‹(‹notation=‹postfix Type››- type)› [10] 5)
  Eqtype   :: [t,t]⇒prop        (‹(‹notation=‹infix Eqtype››- =/ -)› [10,10] 5)
  Elem     :: [i, t]⇒prop       (‹(‹notation=‹infix Elem››- /: -)› [10,10] 5)
  Eqelem   :: [i,i,t]⇒prop      (‹(‹notation=‹mixfix Eqelem››- =/ - :/ -)› [10,10,10]
5)
  Reduce   :: [i,i]⇒prop        (‹Reduce[-,-]›)
  — Types for truth values
  F        :: t
  T        :: t          — F is empty, T contains one element
  contr    :: i⇒i
  tt       :: i
  — Natural numbers
  N        :: t
  Zero     :: i                 (‹0›)
  succ     :: i⇒i
  rec      :: [i, i, [i,i]⇒i] ⇒ i
  — Binary sum
  Plus     :: [t,t]⇒t           (infixr ‹+› 40)
  inl      :: i⇒i
  inr      :: i⇒i
  when     :: [i, i⇒i, i⇒i]⇒i
  — General sum and binary product
  Sum      :: [t, i⇒t]⇒t
  pair     :: [i,i]⇒i           (‹(‹indent=1 notation=‹mixfix pair››<-,/->)›)
  fst      :: i⇒i
  snd      :: i⇒i
  split    :: [i, [i,i]⇒i] ⇒i
  — General product and function space
  Prod     :: [t, i⇒t]⇒t
  lambda   :: (i ⇒ i) ⇒ i       (binder ‹λ› 10)
  app      :: [i,i]⇒i           (infixl ‹`› 60)
  — Equality type
```

Eq :: $[t,i,i] \Rightarrow t$
eq :: i

Some inexplicable syntactic dependencies; in particular, "0" must be introduced after the judgment forms.

syntax
-PROD :: $[idt,t,t] \Rightarrow t$ (‹(‹*indent=3 notation=*‹*binder* ∏ ››∏ -:-./ -)› *10*)
-SUM :: $[idt,t,t] \Rightarrow t$ (‹(‹*indent=3 notation=*‹*binder* ∑ ››∑ -:-./ -)› *10*)
syntax-consts
-PROD ⇌ *Prod* **and**
-SUM ⇌ *Sum*
translations
$\prod x{:}A.\ B \rightleftharpoons CONST\ Prod(A,\ \lambda x.\ B)$
$\sum x{:}A.\ B \rightleftharpoons CONST\ Sum(A,\ \lambda x.\ B)$

abbreviation *Arrow* :: $[t,t] \Rightarrow t$ (**infixr** ‹⟶› *30*)
where $A \longrightarrow B \equiv \prod \text{-}{:}A.\ B$

abbreviation *Times* :: $[t,t] \Rightarrow t$ (**infixr** ‹×› *50*)
where $A \times B \equiv \sum \text{-}{:}A.\ B$

Reduction: a weaker notion than equality; a hack for simplification. *Reduce*$[a,b]$ means either that $a = b : A$ for some A or else that a and b are textually identical.

Does not verify $a{:}A$! Sound because only *trans-red* uses a *Reduce* premise. No new theorems can be proved about the standard judgments.

axiomatization
where
refl-red: $\bigwedge a.\ Reduce[a,a]$ **and**
red-if-equal: $\bigwedge a\ b\ A.\ a = b : A \Longrightarrow Reduce[a,b]$ **and**
trans-red: $\bigwedge a\ b\ c\ A.\ [\![a = b : A;\ Reduce[b,c]]\!] \Longrightarrow a = c : A$ **and**

— Reflexivity

refl-type: $\bigwedge A.\ A\ type \Longrightarrow A = A$ **and**
refl-elem: $\bigwedge a\ A.\ a : A \Longrightarrow a = a : A$ **and**

— Symmetry

sym-type: $\bigwedge A\ B.\ A = B \Longrightarrow B = A$ **and**
sym-elem: $\bigwedge a\ b\ A.\ a = b : A \Longrightarrow b = a : A$ **and**

— Transitivity

trans-type: $\bigwedge A\ B\ C.\ [\![A = B;\ B = C]\!] \Longrightarrow A = C$ **and**
trans-elem: $\bigwedge a\ b\ c\ A.\ [\![a = b : A;\ b = c : A]\!] \Longrightarrow a = c : A$ **and**

equal-types: $\bigwedge a\ A\ B.\ [\![a : A;\ A = B]\!] \Longrightarrow a : B$ **and** *equal-typesL*: $\bigwedge a\ b\ A\ B.\ [\![a = b : A;\ A = B]\!] \Longrightarrow a = b : B$ **and**

— Substitution

subst-type: $\bigwedge a\ A\ B.\ [\![a : A;\ \bigwedge z.\ z{:}A \Longrightarrow B(z)\ \mathit{type}]\!] \Longrightarrow B(a)\ \mathit{type}$ **and**
subst-typeL: $\bigwedge a\ c\ A\ B\ D.\ [\![a = c : A;\ \bigwedge z.\ z{:}A \Longrightarrow B(z) = D(z)]\!] \Longrightarrow B(a) = D(c)$ **and**

subst-elem: $\bigwedge a\ b\ A\ B.\ [\![a : A;\ \bigwedge z.\ z{:}A \Longrightarrow b(z){:}B(z)]\!] \Longrightarrow b(a){:}B(a)$ **and**
subst-elemL:
$\bigwedge a\ b\ c\ d\ A\ B.\ [\![a = c : A;\ \bigwedge z.\ z{:}A \Longrightarrow b(z){=}d(z) : B(z)]\!] \Longrightarrow b(a){=}d(c) : B(a)$ **and**

— The type N – natural numbers

NF: $N\ \mathit{type}$ **and**
NI0: $\mathit{0} : N$ **and**
NI-succ: $\bigwedge a.\ a : N \Longrightarrow \mathit{succ}(a) : N$ **and**
NI-succL: $\bigwedge a\ b.\ a = b : N \Longrightarrow \mathit{succ}(a) = \mathit{succ}(b) : N$ **and**

NE:
$\bigwedge p\ a\ b\ C.\ [\![p{:}\ N;\ a{:}\ C(\mathit{0});\ \bigwedge u\ v.\ [\![u{:}\ N;\ v{:}\ C(u)]\!] \Longrightarrow b(u,v){:}\ C(\mathit{succ}(u))]\!]$
$\Longrightarrow \mathit{rec}(p,\ a,\ \lambda u\ v.\ b(u,v)) : C(p)$ **and**

NEL:
$\bigwedge p\ q\ a\ b\ c\ d\ C.\ [\![p = q : N;\ a = c : C(\mathit{0});$
$\bigwedge u\ v.\ [\![u{:}\ N;\ v{:}\ C(u)]\!] \Longrightarrow b(u,v) = d(u,v){:}\ C(\mathit{succ}(u))]\!]$
$\Longrightarrow \mathit{rec}(p,\ a,\ \lambda u\ v.\ b(u,v)) = \mathit{rec}(q,c,d) : C(p)$ **and**

NC0:
$\bigwedge a\ b\ C.\ [\![a{:}\ C(\mathit{0});\ \bigwedge u\ v.\ [\![u{:}\ N;\ v{:}\ C(u)]\!] \Longrightarrow b(u,v){:}\ C(\mathit{succ}(u))]\!]$
$\Longrightarrow \mathit{rec}(\mathit{0},\ a,\ \lambda u\ v.\ b(u,v)) = a : C(\mathit{0})$ **and**

NC-succ:
$\bigwedge p\ a\ b\ C.\ [\![p{:}\ N;\ a{:}\ C(\mathit{0});\ \bigwedge u\ v.\ [\![u{:}\ N;\ v{:}\ C(u)]\!] \Longrightarrow b(u,v){:}\ C(\mathit{succ}(u))]\!] \Longrightarrow$
$\mathit{rec}(\mathit{succ}(p),\ a,\ \lambda u\ v.\ b(u,v)) = b(p,\ \mathit{rec}(p,\ a,\ \lambda u\ v.\ b(u,v))) : C(\mathit{succ}(p))$ **and**

— The fourth Peano axiom. See page 91 of Martin-Löf's book.
zero-ne-succ: $\bigwedge a.\ [\![a{:}\ N;\ \mathit{0} = \mathit{succ}(a) : N]\!] \Longrightarrow \mathit{0}{:}\ F$ **and**

— The Product of a family of types

ProdF: $\bigwedge A\ B.\ [\![A\ \mathit{type};\ \bigwedge x.\ x{:}A \Longrightarrow B(x)\ \mathit{type}]\!] \Longrightarrow \prod x{:}A.\ B(x)\ \mathit{type}$ **and**

ProdFL:
$\bigwedge A\ B\ C\ D.\ [\![A = C;\ \bigwedge x.\ x{:}A \Longrightarrow B(x) = D(x)]\!] \Longrightarrow \prod x{:}A.\ B(x) = \prod x{:}C.\ D(x)$ **and**

ProdI:
$\bigwedge b\ A\ B.\ [\![A\ type;\ \bigwedge x.\ x{:}A \Longrightarrow b(x){:}B(x)]\!] \Longrightarrow \boldsymbol{\lambda} x.\ b(x) : \prod x{:}A.\ B(x)$ **and**

ProdIL: $\bigwedge b\ c\ A\ B.\ [\![A\ type;\ \bigwedge x.\ x{:}A \Longrightarrow b(x) = c(x) : B(x)]\!] \Longrightarrow$
$\boldsymbol{\lambda} x.\ b(x) = \boldsymbol{\lambda} x.\ c(x) : \prod x{:}A.\ B(x)$ **and**

ProdE: $\bigwedge p\ a\ A\ B.\ [\![p : \prod x{:}A.\ B(x);\ a : A]\!] \Longrightarrow p`a : B(a)$ **and**
ProdEL: $\bigwedge p\ q\ a\ b\ A\ B.\ [\![p = q{:}\ \prod x{:}A.\ B(x);\ a = b : A]\!] \Longrightarrow p`a = q`b : B(a)$ **and**

ProdC: $\bigwedge a\ b\ A\ B.\ [\![a : A;\ \bigwedge x.\ x{:}A \Longrightarrow b(x) : B(x)]\!] \Longrightarrow (\boldsymbol{\lambda} x.\ b(x))\ `\ a = b(a) : B(a)$ **and**

ProdC2: $\bigwedge p\ A\ B.\ p : \prod x{:}A.\ B(x) \Longrightarrow (\boldsymbol{\lambda} x.\ p`x) = p : \prod x{:}A.\ B(x)$ **and**

— The Sum of a family of types

SumF: $\bigwedge A\ B.\ [\![A\ type;\ \bigwedge x.\ x{:}A \Longrightarrow B(x)\ type]\!] \Longrightarrow \sum x{:}A.\ B(x)\ type$ **and**
SumFL: $\bigwedge A\ B\ C\ D.\ [\![A = C;\ \bigwedge x.\ x{:}A \Longrightarrow B(x) = D(x)]\!] \Longrightarrow \sum x{:}A.\ B(x) = \sum x{:}C.\ D(x)$ **and**

SumI: $\bigwedge a\ b\ A\ B.\ [\![a : A;\ b : B(a)]\!] \Longrightarrow <a,b> : \sum x{:}A.\ B(x)$ **and**
SumIL: $\bigwedge a\ b\ c\ d\ A\ B.\ [\![\ a = c : A;\ b = d : B(a)]\!] \Longrightarrow <a,b> = <c,d> : \sum x{:}A.\ B(x)$ **and**

SumE: $\bigwedge p\ c\ A\ B\ C.\ [\![p{:}\ \sum x{:}A.\ B(x);\ \bigwedge x\ y.\ [\![x{:}A;\ y{:}B(x)]\!] \Longrightarrow c(x,y){:}\ C(<x,y>)]\!]$
$\Longrightarrow split(p,\ \lambda x\ y.\ c(x,y)) : C(p)$ **and**

SumEL: $\bigwedge p\ q\ c\ d\ A\ B\ C.\ [\![p = q : \sum x{:}A.\ B(x);$
$\bigwedge x\ y.\ [\![x{:}A;\ y{:}B(x)]\!] \Longrightarrow c(x,y){=}d(x,y){:}\ C(<x,y>)]\!]$
$\Longrightarrow split(p,\ \lambda x\ y.\ c(x,y)) = split(q,\ \lambda x\ y.\ d(x,y)) : C(p)$ **and**

SumC: $\bigwedge a\ b\ c\ A\ B\ C.\ [\![a{:}\ A;\ \ b{:}\ B(a);\ \bigwedge x\ y.\ [\![x{:}A;\ y{:}B(x)]\!] \Longrightarrow c(x,y){:}\ C(<x,y>)]\!]$
$\Longrightarrow split(<a,b>,\ \lambda x\ y.\ c(x,y)) = c(a,b) : C(<a,b>)$ **and**

fst-def: $\bigwedge a.\ fst(a) \equiv split(a,\ \lambda x\ y.\ x)$ **and**
snd-def: $\bigwedge a.\ snd(a) \equiv split(a,\ \lambda x\ y.\ y)$ **and**

— The sum of two types

PlusF: $\bigwedge A\ B.\ [\![A\ type;\ B\ type]\!] \Longrightarrow A{+}B\ type$ **and**
PlusFL: $\bigwedge A\ B\ C\ D.\ [\![A = C;\ B = D]\!] \Longrightarrow A{+}B = C{+}D$ **and**

PlusI-inl: $\bigwedge a\ A\ B.\ [\![a : A;\ B\ type]\!] \Longrightarrow inl(a) : A{+}B$ **and**
PlusI-inlL: $\bigwedge a\ c\ A\ B.\ [\![a = c : A;\ B\ type]\!] \Longrightarrow inl(a) = inl(c) : A{+}B$ **and**

PlusI-inr: $\bigwedge b\ A\ B.\ [\![A\ type;\ b : B]\!] \Longrightarrow inr(b) : A{+}B$ **and**
PlusI-inrL: $\bigwedge b\ d\ A\ B.\ [\![A\ type;\ b = d : B]\!] \Longrightarrow inr(b) = inr(d) : A{+}B$ **and**

PlusE:
$\bigwedge p\ c\ d\ A\ B\ C.\ [\![p\!: A{+}B;$
$\bigwedge x.\ x{:}A \Longrightarrow c(x)\!: C(inl(x));$
$\bigwedge y.\ y{:}B \Longrightarrow d(y)\!: C(inr(y))\ ]\!] \Longrightarrow when(p,\ \lambda x.\ c(x),\ \lambda y.\ d(y)) : C(p)$ **and**

PlusEL:
$\bigwedge p\ q\ c\ d\ e\ f\ A\ B\ C.\ [\![p = q : A{+}B;$
$\bigwedge x.\ x\!: A \Longrightarrow c(x) = e(x) : C(inl(x));$
$\bigwedge y.\ y\!: B \Longrightarrow d(y) = f(y) : C(inr(y))]\!]$
$\Longrightarrow when(p,\ \lambda x.\ c(x),\ \lambda y.\ d(y)) = when(q,\ \lambda x.\ e(x),\ \lambda y.\ f(y)) : C(p)$ **and**

PlusC-inl:
$\bigwedge a\ c\ d\ A\ B\ C.\ [\![a\!: A;$
$\bigwedge x.\ x{:}A \Longrightarrow c(x)\!: C(inl(x));$
$\bigwedge y.\ y{:}B \Longrightarrow d(y)\!: C(inr(y))\ ]\!]$
$\Longrightarrow when(inl(a),\ \lambda x.\ c(x),\ \lambda y.\ d(y)) = c(a) : C(inl(a))$ **and**

PlusC-inr:
$\bigwedge b\ c\ d\ A\ B\ C.\ [\![b\!: B;$
$\bigwedge x.\ x{:}A \Longrightarrow c(x)\!: C(inl(x));$
$\bigwedge y.\ y{:}B \Longrightarrow d(y)\!: C(inr(y))]\!]$
$\Longrightarrow when(inr(b),\ \lambda x.\ c(x),\ \lambda y.\ d(y)) = d(b) : C(inr(b))$ **and**

— The type *Eq*

EqF: $\bigwedge a\ b\ A.\ [\![A\ type;\ a : A;\ b : A]\!] \Longrightarrow Eq(A,a,b)\ type$ **and**
EqFL: $\bigwedge a\ b\ c\ d\ A\ B.\ [\![A = B;\ a = c : A;\ b = d : A]\!] \Longrightarrow Eq(A,a,b) = Eq(B,c,d)$
and
EqI: $\bigwedge a\ b\ A.\ a = b : A \Longrightarrow eq : Eq(A,a,b)$ **and**
EqE: $\bigwedge p\ a\ b\ A.\ p : Eq(A,a,b) \Longrightarrow a = b : A$ **and**

— By equality of types, can prove $C(p)$ from $C(eq)$, an elimination rule
EqC: $\bigwedge p\ a\ b\ A.\ p : Eq(A,a,b) \Longrightarrow p = eq : Eq(A,a,b)$ **and**

— The type *F*

FF: *F type* **and**
FE: $\bigwedge p\ C.\ [\![p\!: F;\ C\ type]\!] \Longrightarrow contr(p) : C$ **and**
FEL: $\bigwedge p\ q\ C.\ [\![p = q : F;\ C\ type]\!] \Longrightarrow contr(p) = contr(q) : C$ **and**

— The type T
— Martin-Löf's book (page 68) discusses elimination and computation. Elimination can be derived by computation and equality of types, but with an extra

premise $C(x)$ type $x{:}T$. Also computation can be derived from elimination.

TF: T *type* **and**
TI: $tt : T$ **and**
TE: $\bigwedge p\ c\ C.\ [\![p : T;\ c : C(tt)]\!] \Longrightarrow c : C(p)$ **and**
TEL: $\bigwedge p\ q\ c\ d\ C.\ [\![p = q : T;\ c = d : C(tt)]\!] \Longrightarrow c = d : C(p)$ **and**
TC: $\bigwedge p.\ p : T \Longrightarrow p = tt : T$

1.1 Tactics and derived rules for Constructive Type Theory

Formation rules.

lemmas *form-rls* = *NF ProdF SumF PlusF EqF FF TF*
and *formL-rls* = *ProdFL SumFL PlusFL EqFL*

Introduction rules. OMITTED:

- *EqI*, because its premise is an *eqelem*, not an *elem*.

lemmas *intr-rls* = *NI0 NI-succ ProdI SumI PlusI-inl PlusI-inr TI*
and *intrL-rls* = *NI-succL ProdIL SumIL PlusI-inlL PlusI-inrL*

Elimination rules. OMITTED:

- *EqE*, because its conclusion is an *eqelem*, not an *elem*
- *TE*, because it does not involve a constructor.

lemmas *elim-rls* = *NE ProdE SumE PlusE FE*
and *elimL-rls* = *NEL ProdEL SumEL PlusEL FEL*

OMITTED: *eqC* are *TC* because they make rewriting loop: $p = un = un = \ldots$

lemmas *comp-rls* = *NC0 NC-succ ProdC SumC PlusC-inl PlusC-inr*

Rules with conclusion $a{:}A$, an elem judgment.

lemmas *element-rls* = *intr-rls elim-rls*

Definitions are (meta)equality axioms.

lemmas *basic-defs* = *fst-def snd-def*

Compare with standard version: B is applied to UNSIMPLIFIED expression!

lemma *SumIL2*: $[\![c = a : A;\ d = b : B(a)]\!] \Longrightarrow \langle c,d\rangle = \langle a,b\rangle : Sum(A,B)$
⟨*proof*⟩

lemmas *intrL2-rls* = *NI-succL ProdIL SumIL2 PlusI-inlL PlusI-inrL*

Exploit p:$Prod(A,B)$ to create the assumption z:$B(a)$. A more natural form of product elimination.

lemma *subst-prodE*:
  assumes p: $Prod(A,B)$
    and a: A
    and $\bigwedge z.\ z$: $B(a) \Longrightarrow c(z)$: $C(z)$
  shows $c(p\text{‘}a)$: $C(p\text{‘}a)$
  ⟨*proof*⟩

1.2 Tactics for type checking

⟨*ML*⟩

For simplification: type formation and checking, but no equalities between terms.

lemmas *routine-rls* = *form-rls formL-rls refl-type element-rls*

⟨*ML*⟩

1.3 Simplification

To simplify the type in a goal.

lemma *replace-type*: $[\![B = A;\ a : A]\!] \Longrightarrow a : B$
  ⟨*proof*⟩

Simplify the parameter of a unary type operator.

lemma *subst-eqtyparg*:
  assumes *1*: a=c : A
    and *2*: $\bigwedge z.\ z$:$A \Longrightarrow B(z)$ *type*
  shows $B(a) = B(c)$
  ⟨*proof*⟩

Simplification rules for Constructive Type Theory.

lemmas *reduction-rls* = *comp-rls* [*THEN trans-elem*]

⟨*ML*⟩

1.4 The elimination rules for fst/snd

lemma *SumE-fst*: $p : Sum(A,B) \Longrightarrow fst(p) : A$
  ⟨*proof*⟩

The first premise must be p:$Sum(A,B)$!!.

lemma *SumE-snd*:
  assumes *major*: p: $Sum(A,B)$
    and A *type*
    and $\bigwedge x.\ x$:$A \Longrightarrow B(x)$ *type*
  shows $snd(p) : B(fst(p))$
  ⟨*proof*⟩

2 The two-element type (booleans and conditionals)

definition *Bool* :: *t*
where *Bool* ≡ *T*+*T*

definition *true* :: *i*
where *true* ≡ *inl*(*tt*)

definition *false* :: *i*
where *false* ≡ *inr*(*tt*)

definition *cond* :: [*i*,*i*,*i*]⇒*i*
where *cond*(*a*,*b*,*c*) ≡ *when*(*a*, λ-. *b*, λ-. *c*)

lemmas *bool-defs* = *Bool-def true-def false-def cond-def*

2.1 Derivation of rules for the type *Bool*

Formation rule.

lemma *boolF*: *Bool type*
⟨*proof*⟩

Introduction rules for *true*, *false*.

lemma *boolI-true*: *true* : *Bool*
⟨*proof*⟩

lemma *boolI-false*: *false* : *Bool*
⟨*proof*⟩

Elimination rule: typing of *cond*.

lemma *boolE*: ⟦*p*:*Bool*; *a* : *C*(*true*); *b* : *C*(*false*)⟧ ⟹ *cond*(*p*,*a*,*b*) : *C*(*p*)
⟨*proof*⟩

lemma *boolEL*: ⟦*p* = *q* : *Bool*; *a* = *c* : *C*(*true*); *b* = *d* : *C*(*false*)⟧
⟹ *cond*(*p*,*a*,*b*) = *cond*(*q*,*c*,*d*) : *C*(*p*)
⟨*proof*⟩

Computation rules for *true*, *false*.

lemma *boolC-true*: ⟦*a* : *C*(*true*); *b* : *C*(*false*)⟧ ⟹ *cond*(*true*,*a*,*b*) = *a* : *C*(*true*)
⟨*proof*⟩

lemma *boolC-false*: ⟦*a* : *C*(*true*); *b* : *C*(*false*)⟧ ⟹ *cond*(*false*,*a*,*b*) = *b* : *C*(*false*)
⟨*proof*⟩

3 Elementary arithmetic

3.1 Arithmetic operators and their definitions

definition *add* :: $[i,i]\Rightarrow i$ (**infixr** ‹#+› *65*)
  where $a\#{+}b \equiv rec(a,\ b,\ \lambda u\ v.\ succ(v))$

definition *diff* :: $[i,i]\Rightarrow i$ (**infixr** ‹−› *65*)
  where $a{-}b \equiv rec(b,\ a,\ \lambda u\ v.\ rec(v,\ 0,\ \lambda x\ y.\ x))$

definition *absdiff* :: $[i,i]\Rightarrow i$ (**infixr** ‹\|−\|› *65*)
  where $a|{-}|b \equiv (a{-}b)\ \#{+}\ (b{-}a)$

definition *mult* :: $[i,i]\Rightarrow i$ (**infixr** ‹#∗› *70*)
  where $a\#{*}b \equiv rec(a,\ 0,\ \lambda u\ v.\ b\ \#{+}\ v)$

definition *mod* :: $[i,i]\Rightarrow i$ (**infixr** ‹*mod*› *70*)
  where $a\ mod\ b \equiv rec(a,\ 0,\ \lambda u\ v.\ rec(succ(v)\ |{-}|\ b,\ 0,\ \lambda x\ y.\ succ(v)))$

definition *div* :: $[i,i]\Rightarrow i$ (**infixr** ‹*div*› *70*)
  where $a\ div\ b \equiv rec(a,\ 0,\ \lambda u\ v.\ rec(succ(u)\ mod\ b,\ succ(v),\ \lambda x\ y.\ v))$

lemmas *arith-defs = add-def diff-def absdiff-def mult-def mod-def div-def*

3.2 Proofs about elementary arithmetic: addition, multiplication, etc.

3.2.1 Addition

Typing of *add*: short and long versions.

lemma *add-typing*: $[\![a{:}N;\ b{:}N]\!] \Longrightarrow a\ \#{+}\ b : N$
  ⟨*proof*⟩

lemma *add-typingL*: $[\![a = c{:}N;\ b = d{:}N]\!] \Longrightarrow a\ \#{+}\ b = c\ \#{+}\ d : N$
  ⟨*proof*⟩

Computation for *add*: 0 and successor cases.

lemma *addC0*: $b{:}N \Longrightarrow 0\ \#{+}\ b = b : N$
  ⟨*proof*⟩

lemma *addC-succ*: $[\![a{:}N;\ b{:}N]\!] \Longrightarrow succ(a)\ \#{+}\ b = succ(a\ \#{+}\ b) : N$
  ⟨*proof*⟩

3.2.2 Multiplication

Typing of *mult*: short and long versions.

lemma *mult-typing*: $[\![a{:}N;\ b{:}N]\!] \Longrightarrow a\ \#{*}\ b : N$
  ⟨*proof*⟩

lemma *mult-typingL*: $[\![a = c{:}N;\ b = d{:}N]\!] \Longrightarrow a \mathbin{\#*} b = c \mathbin{\#*} d : N$
⟨*proof*⟩

Computation for *mult*: 0 and successor cases.

lemma *multC0*: $b{:}N \Longrightarrow 0 \mathbin{\#*} b = 0 : N$
⟨*proof*⟩

lemma *multC-succ*: $[\![a{:}N;\ b{:}N]\!] \Longrightarrow succ(a) \mathbin{\#*} b = b \mathbin{\#+} (a \mathbin{\#*} b) : N$
⟨*proof*⟩

3.2.3 Difference

Typing of difference.

lemma *diff-typing*: $[\![a{:}N;\ b{:}N]\!] \Longrightarrow a - b : N$
⟨*proof*⟩

lemma *diff-typingL*: $[\![a = c{:}N;\ b = d{:}N]\!] \Longrightarrow a - b = c - d : N$
⟨*proof*⟩

Computation for difference: 0 and successor cases.

lemma *diffC0*: $a{:}N \Longrightarrow a - 0 = a : N$
⟨*proof*⟩

Note: $rec(a,\ 0,\ \lambda z\ w.z)$ is $pred(a)$.

lemma *diff-0-eq-0*: $b{:}N \Longrightarrow 0 - b = 0 : N$
⟨*proof*⟩

Essential to simplify FIRST!! (Else we get a critical pair) $succ(a) - succ(b)$ rewrites to $pred(succ(a) - b)$.

lemma *diff-succ-succ*: $[\![a{:}N;\ b{:}N]\!] \Longrightarrow succ(a) - succ(b) = a - b : N$
⟨*proof*⟩

3.3 Simplification

lemmas *arith-typing-rls* = *add-typing mult-typing diff-typing*
and *arith-congr-rls* = *add-typingL mult-typingL diff-typingL*

lemmas *congr-rls* = *arith-congr-rls intrL2-rls elimL-rls*

lemmas *arithC-rls* =
addC0 addC-succ
multC0 multC-succ
diffC0 diff-0-eq-0 diff-succ-succ

⟨*ML*⟩

3.4 Addition

Associative law for addition.

lemma *add-assoc*: $[\![a{:}N;\ b{:}N;\ c{:}N]\!] \Longrightarrow (a\ \#{+}\ b)\ \#{+}\ c = a\ \#{+}\ (b\ \#{+}\ c) : N$
 ⟨*proof*⟩

Commutative law for addition. Can be proved using three inductions. Must simplify after first induction! Orientation of rewrites is delicate.

lemma *add-commute*: $[\![a{:}N;\ b{:}N]\!] \Longrightarrow a\ \#{+}\ b = b\ \#{+}\ a : N$
 ⟨*proof*⟩

3.5 Multiplication

Right annihilation in product.

lemma *mult-0-right*: $a{:}N \Longrightarrow a\ \#{*}\ 0 = 0 : N$
 ⟨*proof*⟩

Right successor law for multiplication.

lemma *mult-succ-right*: $[\![a{:}N;\ b{:}N]\!] \Longrightarrow a\ \#{*}\ succ(b) = a\ \#{+}\ (a\ \#{*}\ b) : N$
 ⟨*proof*⟩

Commutative law for multiplication.

lemma *mult-commute*: $[\![a{:}N;\ b{:}N]\!] \Longrightarrow a\ \#{*}\ b = b\ \#{*}\ a : N$
 ⟨*proof*⟩

Addition distributes over multiplication.

lemma *add-mult-distrib*: $[\![a{:}N;\ b{:}N;\ c{:}N]\!] \Longrightarrow (a\ \#{+}\ b)\ \#{*}\ c = (a\ \#{*}\ c)\ \#{+}\ (b\ \#{*}\ c) : N$
 ⟨*proof*⟩

Associative law for multiplication.

lemma *mult-assoc*: $[\![a{:}N;\ b{:}N;\ c{:}N]\!] \Longrightarrow (a\ \#{*}\ b)\ \#{*}\ c = a\ \#{*}\ (b\ \#{*}\ c) : N$
 ⟨*proof*⟩

3.6 Difference

Difference on natural numbers, without negative numbers

- $a - b = 0$ iff $a \leq b$
- $a - b = succ(c)$ iff $a > b$

lemma *diff-self-eq-0*: $a{:}N \Longrightarrow a - a = 0 : N$
 ⟨*proof*⟩

lemma *add-0-right*: $[\![c : N;\ 0 : N;\ c : N]\!] \Longrightarrow c \ \#+\ 0 = c : N$
⟨*proof*⟩

Addition is the inverse of subtraction: if $b \leq x$ then $b\ \#+\ (x - b) = x$. An example of induction over a quantified formula (a product). Uses rewriting with a quantified, implicative inductive hypothesis.

schematic-goal *add-diff-inverse-lemma*:
$b{:}N \Longrightarrow\ ?a : \prod x{:}N.\ Eq(N,\ b{-}x,\ 0) \longrightarrow Eq(N,\ b\ \#+\ (x{-}b),\ x)$
⟨*proof*⟩

Version of above with premise $b - a = 0$ i.e. $a \geq b$. Using *ProdE* does not work – for $?B(?a)$ is ambiguous. Instead, *add-diff-inverse-lemma* states the desired induction scheme; the use of *THEN* below instantiates Vars in *ProdE* automatically.

lemma *add-diff-inverse*: $[\![a{:}N;\ b{:}N;\ b - a = 0 : N]\!] \Longrightarrow b\ \#+\ (a{-}b) = a : N$
⟨*proof*⟩

3.7 Absolute difference

Typing of absolute difference: short and long versions.

lemma *absdiff-typing*: $[\![a{:}N;\ b{:}N]\!] \Longrightarrow a\ |{-}|\ b : N$
⟨*proof*⟩

lemma *absdiff-typingL*: $[\![a = c{:}N;\ b = d{:}N]\!] \Longrightarrow a\ |{-}|\ b = c\ |{-}|\ d : N$
⟨*proof*⟩

lemma *absdiff-self-eq-0*: $a{:}N \Longrightarrow a\ |{-}|\ a = 0 : N$
⟨*proof*⟩

lemma *absdiffC0*: $a{:}N \Longrightarrow 0\ |{-}|\ a = a : N$
⟨*proof*⟩

lemma *absdiff-succ-succ*: $[\![a{:}N;\ b{:}N]\!] \Longrightarrow succ(a)\ |{-}|\ succ(b) \ = \ a\ |{-}|\ b : N$
⟨*proof*⟩

Note how easy using commutative laws can be? ...not always...

lemma *absdiff-commute*: $[\![a{:}N;\ b{:}N]\!] \Longrightarrow a\ |{-}|\ b = b\ |{-}|\ a : N$
⟨*proof*⟩

If $a + b = 0$ then $a = 0$. Surprisingly tedious.

schematic-goal *add-eq0-lemma*: $[\![a{:}N;\ b{:}N]\!] \Longrightarrow\ ?c : Eq(N,a\#{+}b,0) \longrightarrow Eq(N,a,0)$
⟨*proof*⟩

Version of above with the premise $a + b = 0$. Again, resolution instantiates variables in *ProdE*.

lemma *add-eq0*: $[\![a{:}N;\ b{:}N;\ a\ \#+\ b = 0 : N]\!] \Longrightarrow a = 0 : N$
⟨*proof*⟩

Here is a lemma to infer $a - b = 0$ and $b - a = 0$ from $a \mid - \mid b = 0$, below.

schematic-goal *absdiff-eq0-lem*:
$[\![a{:}N;\ b{:}N;\ a\ |{-}|\ b = 0 : N]\!] \Longrightarrow ?a : Eq(N,\ a{-}b,\ 0) \times Eq(N,\ b{-}a,\ 0)$
⟨*proof*⟩

If $a \mid - \mid b = 0$ then $a = b$ proof: $a - b = 0$ and $b - a = 0$, so $b = a + (b - a) = a + 0 = a$.

lemma *absdiff-eq0*: $[\![a\ |{-}|\ b = 0 : N;\ a{:}N;\ b{:}N]\!] \Longrightarrow a = b : N$
⟨*proof*⟩

3.8 Remainder and Quotient

Typing of remainder: short and long versions.

lemma *mod-typing*: $[\![a{:}N;\ b{:}N]\!] \Longrightarrow a\ mod\ b : N$
⟨*proof*⟩

lemma *mod-typingL*: $[\![a = c{:}N;\ b = d{:}N]\!] \Longrightarrow a\ mod\ b = c\ mod\ d : N$
⟨*proof*⟩

Computation for *mod*: 0 and successor cases.

lemma *modC0*: $b{:}N \Longrightarrow 0\ mod\ b = 0 : N$
⟨*proof*⟩

lemma *modC-succ*: $[\![a{:}N;\ b{:}N]\!] \Longrightarrow$
$succ(a)\ mod\ b = rec(succ(a\ mod\ b)\ |{-}|\ b,\ 0,\ \lambda x\ y.\ succ(a\ mod\ b)) : N$
⟨*proof*⟩

Typing of quotient: short and long versions.

lemma *div-typing*: $[\![a{:}N;\ b{:}N]\!] \Longrightarrow a\ div\ b : N$
⟨*proof*⟩

lemma *div-typingL*: $[\![a = c{:}N;\ b = d{:}N]\!] \Longrightarrow a\ div\ b = c\ div\ d : N$
⟨*proof*⟩

lemmas *div-typing-rls = mod-typing div-typing absdiff-typing*

Computation for quotient: 0 and successor cases.

lemma *divC0*: $b{:}N \Longrightarrow 0\ div\ b = 0 : N$
⟨*proof*⟩

lemma *divC-succ*: $[\![a{:}N;\ b{:}N]\!] \Longrightarrow$
$succ(a)\ div\ b = rec(succ(a)\ mod\ b,\ succ(a\ div\ b),\ \lambda x\ y.\ a\ div\ b) : N$
⟨*proof*⟩

Version of above with same condition as the *mod* one.

lemma *divC-succ2*: $[\![a{:}N;\ b{:}N]\!] \Longrightarrow$
$succ(a)\ div\ b = rec(succ(a\ mod\ b)\ |{-}|\ b,\ succ(a\ div\ b),\ \lambda x\ y.\ a\ div\ b) : N$

⟨*proof*⟩

For case analysis on whether a number is 0 or a successor.

lemma *iszero-decidable*: $a{:}N \Longrightarrow rec(a,\ inl(eq),\ \lambda ka\ kb.\ inr(\langle ka,\ eq\rangle))$:
$Eq(N,a,0) + (\sum x{:}N.\ Eq(N,a,\ succ(x)))$
⟨*proof*⟩

Main Result. Holds when b is 0 since $a\ mod\ 0 = a$ and $a\ div\ 0 = 0$.

lemma *mod-div-equality*: $[\![a{:}N;\ b{:}N]\!] \Longrightarrow a\ mod\ b\ \#{+}\ (a\ div\ b)\ \#{*}\ b = a : N$
⟨*proof*⟩

end

4 Easy examples: type checking and type deduction

theory *Typechecking*
imports *../CTT*
begin

4.1 Single-step proofs: verifying that a type is well-formed

schematic-goal *?A type*
⟨*proof*⟩

schematic-goal *?A type*
⟨*proof*⟩

schematic-goal $\prod z{:}?A\ .\ N + ?B(z)\ type$
⟨*proof*⟩

4.2 Multi-step proofs: Type inference

lemma $\prod w{:}N.\ N + N\ type$
⟨*proof*⟩

schematic-goal $\langle 0,\ succ(0)\rangle : ?A$
⟨*proof*⟩

schematic-goal $\prod w{:}N\ .\ Eq(?A,w,w)\ type$
⟨*proof*⟩

schematic-goal $\prod x{:}N\ .\ \prod y{:}N\ .\ Eq(?A,x,y)\ type$
⟨*proof*⟩

typechecking an application of fst

schematic-goal $(\boldsymbol{\lambda} u.\ split(u,\ \lambda v\ w.\ v))\ `\ \langle 0,\ succ(0)\rangle : ?A$
⟨*proof*⟩

typechecking the predecessor function

schematic-goal $\boldsymbol{\lambda}n.\ rec(n,\ 0,\ \lambda x\ y.\ x) : ?A$
⟨*proof*⟩

typechecking the addition function

schematic-goal $\boldsymbol{\lambda}n.\ \boldsymbol{\lambda}m.\ rec(n,\ m,\ \lambda x\ y.\ succ(y)) : ?A$
⟨*proof*⟩

Proofs involving arbitrary types. For concreteness, every type variable left over is forced to be N

⟨*ML*⟩

schematic-goal $\boldsymbol{\lambda}w.\ \lt w,w\gt : ?A$
⟨*proof*⟩

schematic-goal $\boldsymbol{\lambda}x.\ \boldsymbol{\lambda}y.\ x : ?A$
⟨*proof*⟩

typechecking fst (as a function object)

schematic-goal $\boldsymbol{\lambda}i.\ split(i,\ \lambda j\ k.\ j) : ?A$
⟨*proof*⟩

end

5 Examples with elimination rules

theory *Elimination*
imports *../CTT*
begin

This finds the functions fst and snd!

schematic-goal [*folded basic-defs*]: $A\ type \Longrightarrow ?a : (A \times A) \longrightarrow A$
⟨*proof*⟩

schematic-goal [*folded basic-defs*]: $A\ type \Longrightarrow ?a : (A \times A) \longrightarrow A$
⟨*proof*⟩

Double negation of the Excluded Middle

schematic-goal $A\ type \Longrightarrow ?a : ((A + (A \longrightarrow F)) \longrightarrow F) \longrightarrow F$
⟨*proof*⟩

Experiment: the proof above in Isar

lemma
assumes $A\ type$ **shows** $(\boldsymbol{\lambda}f.\ f\ \text{`}\ inr(\boldsymbol{\lambda}y.\ f\ \text{`}\ inl(y))) : ((A + (A \longrightarrow F)) \longrightarrow F) \longrightarrow F$
⟨*proof*⟩

schematic-goal $[\![A\ \mathit{type};\ B\ \mathit{type}]\!] \Longrightarrow ?a : (A \times B) \longrightarrow (B \times A)$
⟨*proof*⟩

Binary sums and products

schematic-goal $[\![A\ \mathit{type};\ B\ \mathit{type};\ C\ \mathit{type}]\!] \Longrightarrow ?a : (A + B \longrightarrow C) \longrightarrow (A \longrightarrow C) \times (B \longrightarrow C)$
⟨*proof*⟩

schematic-goal $[\![A\ \mathit{type};\ B\ \mathit{type};\ C\ \mathit{type}]\!] \Longrightarrow ?a : A \times (B + C) \longrightarrow (A \times B + A \times C)$
⟨*proof*⟩

schematic-goal
assumes $A\ \mathit{type}$
and $\bigwedge x.\ x{:}A \Longrightarrow B(x)\ \mathit{type}$
and $\bigwedge x.\ x{:}A \Longrightarrow C(x)\ \mathit{type}$
shows $?a : (\sum x{:}A.\ B(x) + C(x)) \longrightarrow (\sum x{:}A.\ B(x)) + (\sum x{:}A.\ C(x))$
⟨*proof*⟩

Construction of the currying functional

schematic-goal $[\![A\ \mathit{type};\ B\ \mathit{type};\ C\ \mathit{type}]\!] \Longrightarrow ?a : (A \times B \longrightarrow C) \longrightarrow (A \longrightarrow (B \longrightarrow C))$
⟨*proof*⟩

schematic-goal
assumes $A\ \mathit{type}$
and $\bigwedge x.\ x{:}A \Longrightarrow B(x)\ \mathit{type}$
and $\bigwedge z.\ z{:}\ (\sum x{:}A.\ B(x)) \Longrightarrow C(z)\ \mathit{type}$
shows $?a : \prod f{:}\ (\prod z : (\sum x{:}A\ .\ B(x))\ .\ C(z)).$
$(\prod x{:}A\ .\ \prod y{:}B(x)\ .\ C(<x,y>))$
⟨*proof*⟩

Martin-Löf (1984), page 48: axiom of sum-elimination (uncurry)

schematic-goal $[\![A\ \mathit{type};\ B\ \mathit{type};\ C\ \mathit{type}]\!] \Longrightarrow ?a : (A \longrightarrow (B \longrightarrow C)) \longrightarrow (A \times B \longrightarrow C)$
⟨*proof*⟩

schematic-goal
assumes $A\ \mathit{type}$
and $\bigwedge x.\ x{:}A \Longrightarrow B(x)\ \mathit{type}$
and $\bigwedge z.\ z{:}\ (\sum x{:}A\ .\ B(x)) \Longrightarrow C(z)\ \mathit{type}$
shows $?a : (\prod x{:}A\ .\ \prod y{:}B(x)\ .\ C(<x,y>))$
$\longrightarrow (\prod z : (\sum x{:}A\ .\ B(x))\ .\ C(z))$
⟨*proof*⟩

Function application

schematic-goal $[\![A\ type;\ B\ type]\!] \Longrightarrow\ ?a : ((A \longrightarrow B) \times A) \longrightarrow B$
⟨*proof*⟩

Basic test of quantifier reasoning

schematic-goal
assumes $A\ type$
and $B\ type$
and $\bigwedge x\ y.\ [\![x{:}A;\ y{:}B]\!] \Longrightarrow C(x,y)\ type$
shows
$?a :\ \ (\sum y{:}B\ .\ \prod x{:}A\ .\ C(x,y))$
$\longrightarrow (\prod x{:}A\ .\ \sum y{:}B\ .\ C(x,y))$
⟨*proof*⟩

Martin-Löf (1984) pages 36-7: the combinator S

schematic-goal
assumes $A\ type$
and $\bigwedge x.\ x{:}A \Longrightarrow B(x)\ type$
and $\bigwedge x\ y.\ [\![x{:}A;\ y{:}B(x)]\!] \Longrightarrow C(x,y)\ type$
shows $?a :\ \ (\prod x{:}A.\ \prod y{:}B(x).\ C(x,y))$
$\longrightarrow (\prod f{:}\ (\prod x{:}A.\ B(x)).\ \prod x{:}A.\ C(x,\ f`x))$
⟨*proof*⟩

Martin-Löf (1984) page 58: the axiom of disjunction elimination

schematic-goal
assumes $A\ type$
and $B\ type$
and $\bigwedge z.\ z{:}\ A{+}B \Longrightarrow C(z)\ type$
shows $?a : (\prod x{:}A.\ C(inl(x))) \longrightarrow (\prod y{:}B.\ C(inr(y)))$
$\longrightarrow (\prod z{:}\ A{+}B.\ C(z))$
⟨*proof*⟩

schematic-goal [*folded basic-defs*]:
$[\![A\ type;\ B\ type;\ C\ type]\!] \Longrightarrow\ ?a : (A \longrightarrow B \times C) \longrightarrow (A \longrightarrow B) \times (A \longrightarrow C)$
⟨*proof*⟩

AXIOM OF CHOICE! Delicate use of elimination rules

schematic-goal
assumes $A\ type$
and $\bigwedge x.\ x{:}A \Longrightarrow B(x)\ type$
and $\bigwedge x\ y.\ [\![x{:}A;\ y{:}B(x)]\!] \Longrightarrow C(x,y)\ type$
shows $?a : (\prod x{:}A.\ \sum y{:}B(x).\ C(x,y)) \longrightarrow (\sum f{:}\ (\prod x{:}A.\ B(x)).\ \prod x{:}A.\ C(x,\ f`x))$
⟨*proof*⟩

A structured proof of AC

lemma *Axiom-of-Choice*:

assumes A *type*
and $\bigwedge x.\ x{:}A \Longrightarrow B(x)$ *type*
and $\bigwedge x\ y.\ [\![x{:}A;\ y{:}B(x)]\!] \Longrightarrow C(x,y)$ *type*
shows $(\boldsymbol{\lambda} f.\ {<}\boldsymbol{\lambda} x.\ fst(f`x),\ \boldsymbol{\lambda} x.\ snd(f`x){>})$
$: (\prod x{:}A.\ \sum y{:}B(x).\ C(x,y)) \longrightarrow (\sum f{:}\ (\prod x{:}A.\ B(x)).\ \prod x{:}A.\ C(x,\ f`x))$
⟨*proof*⟩

Axiom of choice. Proof without fst, snd. Harder still!

schematic-goal [*folded basic-defs*]:
assumes A *type*
and $\bigwedge x.\ x{:}A \Longrightarrow B(x)$ *type*
and $\bigwedge x\ y.\ [\![x{:}A;\ y{:}B(x)]\!] \Longrightarrow C(x,y)$ *type*
shows $?a : (\prod x{:}A.\ \sum y{:}B(x).\ C(x,y)) \longrightarrow (\sum f{:}\ (\prod x{:}A.\ B(x)).\ \prod x{:}A.\ C(x,$
$f`x))$
⟨*proof*⟩

Example of sequent-style deduction

schematic-goal
assumes A *type*
and B *type*
and $\bigwedge z.\ z{:}A \times B \Longrightarrow C(z)$ *type*
shows $?a : (\sum z{:}A \times B.\ C(z)) \longrightarrow (\sum u{:}A.\ \sum v{:}B.\ C({<}u,v{>}))$
⟨*proof*⟩

end

6 Equality reasoning by rewriting

theory *Equality*
imports *../CTT*
begin

lemma *split-eq*: $p : Sum(A,B) \Longrightarrow split(p,pair) = p : Sum(A,B)$
⟨*proof*⟩

lemma *when-eq*: $[\![A\ type;\ B\ type;\ p : A{+}B]\!] \Longrightarrow when(p,inl,inr) = p : A + B$
⟨*proof*⟩

in the "rec" formulation of addition, $0 + n = n$

lemma $p{:}N \Longrightarrow rec(p,0,\ \lambda y\ z.\ succ(y)) = p : N$
⟨*proof*⟩

the harder version, $n + 0 = n$: recursive, uses induction hypothesis

lemma $p{:}N \Longrightarrow rec(p,0,\ \lambda y\ z.\ succ(z)) = p : N$
⟨*proof*⟩

Associativity of addition

lemma $[\![a{:}N;\ b{:}N;\ c{:}N]\!]$

$\Longrightarrow$ *rec*(*rec*(*a*, *b*, λ*x y*. *succ*(*y*)), *c*, λ*x y*. *succ*(*y*)) =
rec(*a*, *rec*(*b*, *c*, λ*x y*. *succ*(*y*)), λ*x y*. *succ*(*y*)) : *N*
⟨*proof*⟩

Martin-Löf (1984) page 62: pairing is surjective

lemma *p* : *Sum*(*A*,*B*) $\Longrightarrow$ <*split*(*p*,λ*x y*. *x*), *split*(*p*,λ*x y*. *y*)> = *p* : *Sum*(*A*,*B*)
⟨*proof*⟩

lemma ⟦*a* : *A*; *b* : *B*⟧ $\Longrightarrow$ (**λ***u*. *split*(*u*, λ*v w*.<*w*,*v*>)) ‘ <*a*,*b*> = <*b*,*a*> : $\sum$ *x*:*B*. *A*
⟨*proof*⟩

a contrived, complicated simplication, requires sum-elimination also

lemma (**λ***f*. **λ***x*. *f*‘(*f*‘*x*)) ‘ (**λ***u*. *split*(*u*, λ*v w*.<*w*,*v*>)) =
λ*x*. *x* : $\prod$ *x*:($\sum$ *y*:*N*. *N*). ($\sum$ *y*:*N*. *N*)
⟨*proof*⟩

end

7 Synthesis examples, using a crude form of narrowing

theory *Synthesis*
imports ../*CTT*
begin

discovery of predecessor function

schematic-goal *?a* : $\sum$ *pred*:*?A* . *Eq*(*N*, *pred*‘*0*, *0*) × ($\prod$ *n*:*N*. *Eq*(*N*, *pred* ‘ *succ*(*n*), *n*))
⟨*proof*⟩

the function fst as an element of a function type

schematic-goal [*folded basic-defs*]:
A type $\Longrightarrow$ *?a*: $\sum$ *f*:*?B* . $\prod$ *i*:*A*. $\prod$ *j*:*A*. *Eq*(*A*, *f* ‘ <*i*,*j*>, *i*)
⟨*proof*⟩

An interesting use of the eliminator, when

schematic-goal *?a* : $\prod$ *i*:*N*. *Eq*(*?A*, *?b*(*inl*(*i*)), <*0* , *i*>)
× *Eq*(*?A*, *?b*(*inr*(*i*)), <*succ*(*0*), *i*>)
⟨*proof*⟩

schematic-goal *?a* : $\prod$ *i*:*N*. *Eq*(*?A*(*i*), *?b*(*inl*(*i*)), <*0* , *i*>)
× *Eq*(*?A*(*i*), *?b*(*inr*(*i*)), <*succ*(*0*),*i*>)
⟨*proof*⟩

A tricky combination of when and split

schematic-goal [*folded basic-defs*]:

$$?a : \prod i{:}N.\ \prod j{:}N.\ Eq(?A,\ ?b(inl(<i,j>)),\ i) \\ \times\ \ Eq(?A,\ ?b(inr(<i,j>)),\ j)$$

⟨*proof*⟩

schematic-goal $?a : \prod i{:}N.\ \prod j{:}N.\ Eq(?A(i,j),\ ?b(inl(<i,j>)),\ i)$
$\times\ \ Eq(?A(i,j),\ ?b(inr(<i,j>)),\ j)$

⟨*proof*⟩

schematic-goal $?a : \prod i{:}N.\ \prod j{:}N.\ Eq(N,\ ?b(inl(<i,j>)),\ i)$
$\times\ \ Eq(N,\ ?b(inr(<i,j>)),\ j)$

⟨*proof*⟩

Deriving the addition operator

schematic-goal [*folded arith-defs*]:

$$?c : \prod n{:}N.\ Eq(N,\ ?f(0,n),\ n) \\ \times\ \ (\prod m{:}N.\ Eq(N,\ ?f(succ(m),\ n),\ succ(?f(m,n))))$$

⟨*proof*⟩

The addition function – using explicit lambdas

schematic-goal [*folded arith-defs*]:

$$?c : \sum plus : ?A\ . \\ \prod x{:}N.\ Eq(N,\ plus`0`x,\ x) \\ \times\ \ (\prod y{:}N.\ Eq(N,\ plus`succ(y)`x,\ succ(plus`y`x)))$$

⟨*proof*⟩

end