

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

Larry Paulson

September 11, 2023

Contents

1	Constructive Type Theory: axiomatic basis	1
1.1	Tactics and derived rules for Constructive Type Theory . . .	6
1.2	Tactics for type checking	7
1.3	Simplification	7
1.4	The elimination rules for <code>fst/snd</code>	8
2	The two-element type (booleans and conditionals)	8
2.1	Derivation of rules for the type <code>Bool</code>	8
3	Elementary arithmetic	9
3.1	Arithmetic operators and their definitions	9
3.2	Proofs about elementary arithmetic: addition, multiplication, etc.	9
3.2.1	Addition	9
3.2.2	Multiplication	10
3.2.3	Difference	10
3.3	Simplification	10
3.4	Addition	11
3.5	Multiplication	11
3.6	Difference	12
3.7	Absolute difference	12
3.8	Remainder and Quotient	13
4	Easy examples: type checking and type deduction	14
4.1	Single-step proofs: verifying that a type is well-formed	14
4.2	Multi-step proofs: Type inference	14
5	Examples with elimination rules	15
6	Equality reasoning by rewriting	18

```

theory CTT
imports Pure
begin

```

1 Constructive Type Theory: axiomatic basis

$\langle ML \rangle$

```

typedecl i
typedecl t
typedecl o

```

consts

— Judgments

```

Type    :: t ⇒ prop      ((- type) [10] 5)
Eqtype  :: [t,t]⇒prop    ((- =/ -) [10,10] 5)
Elem     :: [i, t]⇒prop   ((- /: -) [10,10] 5)
Eqelem  :: [i,i,t]⇒prop  ((- =/ - :/ -) [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      ((1<-,-/>-))
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]⇒t

```

eq :: *i*

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

syntax

-*PROD* :: $[idt, t, t] \Rightarrow t$ (($\exists \prod$ -:/ -) 10)
 -*SUM* :: $[idt, t, t] \Rightarrow t$ (($\exists \sum$ -:/ -) 10)

translations

$\prod x:A. B \Leftrightarrow \text{CONST Prod}(A, \lambda x. B)$
 $\sum x:A. B \Leftrightarrow \text{CONST Sum}(A, \lambda x. B)$

abbreviation *Arrow* :: $[t, t] \Rightarrow t$ (**infixr** \longrightarrow 30)
where $A \longrightarrow B \equiv \prod -:A. B$

abbreviation *Times* :: $[t, t] \Rightarrow t$ (**infixr** \times 50)
where $A \times B \equiv \sum -:A. B$

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: $\bigwedge a. \text{Reduce}[a, a]$ **and**
red-if-equal: $\bigwedge a b A. a = b : A \Longrightarrow \text{Reduce}[a, b]$ **and**
trans-red: $\bigwedge a b c A. \llbracket a = b : A; \text{Reduce}[b, c] \rrbracket \Longrightarrow a = c : A$ **and**

— Reflexivity

refl-type: $\bigwedge A. A \text{ type} \Longrightarrow A = A$ **and**
refl-lem: $\bigwedge a A. a : A \Longrightarrow a = a : A$ **and**

— Symmetry

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

— Transitivity

trans-type: $\bigwedge A B C. \llbracket A = B; B = C \rrbracket \Longrightarrow A = C$ **and**
trans-lem: $\bigwedge a b c A. \llbracket a = b : A; b = c : A \rrbracket \Longrightarrow a = c : A$ **and**

equal-types: $\bigwedge a A B. \llbracket a : A; A = B \rrbracket \Longrightarrow a : B$ **and**
equal-typesL: $\bigwedge a b A B. \llbracket a = b : A; A = B \rrbracket \Longrightarrow a = b : B$ **and**

— Substitution

subst-type: $\bigwedge a A B. \llbracket a : A; \bigwedge z. z:A \implies B(z) \text{ type} \rrbracket \implies B(a) \text{ type}$ **and**
subst-typeL: $\bigwedge a c A B D. \llbracket a = c : A; \bigwedge z. z:A \implies B(z) = D(z) \rrbracket \implies B(a) = D(c)$ **and**

subst-elim: $\bigwedge a b A B. \llbracket a : A; \bigwedge z. z:A \implies b(z):B(z) \rrbracket \implies b(a):B(a)$ **and**
subst-elimL:
 $\bigwedge a b c d A B. \llbracket a = c : A; \bigwedge z. z:A \implies b(z)=d(z) : B(z) \rrbracket \implies b(a)=d(c) : B(a)$
and

— The type N – natural numbers

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

NE:
 $\bigwedge p a b C. \llbracket p : N; a : C(0); \bigwedge u v. \llbracket u : N; v : C(u) \rrbracket \implies b(u,v) : C(\text{succ}(u)) \rrbracket$
 $\implies \text{rec}(p, a, \lambda u v. b(u,v)) : C(p)$ **and**

NEL:
 $\bigwedge p q a b c d C. \llbracket p = q : N; a = c : C(0);$
 $\bigwedge u v. \llbracket u : N; v : C(u) \rrbracket \implies b(u,v) = d(u,v) : C(\text{succ}(u)) \rrbracket$
 $\implies \text{rec}(p, a, \lambda u v. b(u,v)) = \text{rec}(q,c,d) : C(p)$ **and**

NC0:
 $\bigwedge a b C. \llbracket a : C(0); \bigwedge u v. \llbracket u : N; v : C(u) \rrbracket \implies b(u,v) : C(\text{succ}(u)) \rrbracket$
 $\implies \text{rec}(0, a, \lambda u v. b(u,v)) = a : C(0)$ **and**

NC-succ:
 $\bigwedge p a b C. \llbracket p : N; a : C(0); \bigwedge u v. \llbracket u : N; v : C(u) \rrbracket \implies b(u,v) : C(\text{succ}(u)) \rrbracket \implies$
 $\text{rec}(\text{succ}(p), a, \lambda u v. b(u,v)) = b(p, \text{rec}(p, a, \lambda u v. b(u,v))) : C(\text{succ}(p))$ **and**

— The fourth Peano axiom. See page 91 of Martin-Löf's book.

zero-ne-succ: $\bigwedge a. \llbracket a : N; 0 = \text{succ}(a) : N \rrbracket \implies 0 : F$ **and**

— The Product of a family of types

ProdF: $\bigwedge A B. \llbracket A \text{ type}; \bigwedge x. x:A \implies B(x) \text{ type} \rrbracket \implies \prod x:A. B(x) \text{ type}$ **and**

ProdFL:
 $\bigwedge A B C D. \llbracket A = C; \bigwedge x. x:A \implies B(x) = D(x) \rrbracket \implies \prod x:A. B(x) = \prod x:C. D(x)$ **and**

ProdI:
 $\bigwedge b A B. \llbracket A \text{ type}; \bigwedge x. x:A \implies b(x):B(x) \rrbracket \implies \lambda x. b(x) : \prod x:A. B(x)$ **and**

ProdIL: $\bigwedge b c A B. \llbracket A \text{ type}; \bigwedge x. x:A \implies b(x) = c(x) : B(x) \rrbracket \implies$
 $\lambda x. b(x) = \lambda x. c(x) : \prod x:A. B(x)$ **and**

ProdE: $\bigwedge p a A B. \llbracket p : \prod x:A. B(x); a : A \rrbracket \implies p'a : B(a)$ **and**

ProdEL: $\bigwedge p q a b A B. \llbracket p = q; \prod x:A. B(x); a = b : A \rrbracket \implies p'a = q'b : B(a)$
and

ProdC: $\bigwedge a b A B. \llbracket a : A; \bigwedge x. x:A \implies b(x) : B(x) \rrbracket \implies (\lambda x. b(x)) ' a = b(a) :$
 $B(a)$ **and**

ProdC2: $\bigwedge p A B. p : \prod x:A. B(x) \implies (\lambda x. p'x) = p : \prod x:A. B(x)$ **and**

— The Sum of a family of types

SumF: $\bigwedge A B. \llbracket A \text{ type}; \bigwedge x. x:A \implies B(x) \text{ type} \rrbracket \implies \sum x:A. B(x) \text{ type}$ **and**

SumFL: $\bigwedge A B C D. \llbracket A = C; \bigwedge x. x:A \implies B(x) = D(x) \rrbracket \implies \sum x:A. B(x) =$
 $\sum x:C. D(x)$ **and**

SumI: $\bigwedge a b A B. \llbracket a : A; b : B(a) \rrbracket \implies \langle a, b \rangle : \sum x:A. B(x)$ **and**

SumIL: $\bigwedge a b c d A B. \llbracket a = c : A; b = d : B(a) \rrbracket \implies \langle a, b \rangle = \langle c, d \rangle : \sum x:A.$
 $B(x)$ **and**

SumE: $\bigwedge p c A B C. \llbracket p : \sum x:A. B(x); \bigwedge x y. \llbracket x:A; y:B(x) \rrbracket \implies c(x,y) : C(\langle x, y \rangle) \rrbracket$
 $\implies \text{split}(p, \lambda x y. c(x,y)) : C(p)$ **and**

SumEL: $\bigwedge p q c d A B C. \llbracket p = q : \sum x:A. B(x);$

$\bigwedge x y. \llbracket x:A; y:B(x) \rrbracket \implies c(x,y) = d(x,y) : C(\langle x, y \rangle) \rrbracket$

$\implies \text{split}(p, \lambda x y. c(x,y)) = \text{split}(q, \lambda x y. d(x,y)) : C(p)$ **and**

SumC: $\bigwedge a b c A B C. \llbracket a : A; b : B(a); \bigwedge x y. \llbracket x:A; y:B(x) \rrbracket \implies c(x,y) : C(\langle x, y \rangle) \rrbracket$

$\implies \text{split}(\langle a, b \rangle, \lambda x y. c(x,y)) = c(a,b) : C(\langle a, b \rangle)$ **and**

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

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

— The sum of two types

PlusF: $\bigwedge A B. \llbracket A \text{ type}; B \text{ type} \rrbracket \implies A+B \text{ type}$ **and**

PlusFL: $\bigwedge A B C D. \llbracket A = C; B = D \rrbracket \implies A+B = C+D$ **and**

PlusI-inl: $\bigwedge a A B. \llbracket a : A; B \text{ type} \rrbracket \implies \text{inl}(a) : A+B$ **and**

PlusI-inlL: $\bigwedge a c A B. \llbracket a = c : A; B \text{ type} \rrbracket \implies \text{inl}(a) = \text{inl}(c) : A+B$ **and**

PlusI-inr: $\bigwedge b A B. \llbracket A \text{ type}; b : B \rrbracket \implies \text{inr}(b) : A+B$ **and**

PlusI-inrL: $\bigwedge b d A B. \llbracket A \text{ type}; b = d : B \rrbracket \implies \text{inr}(b) = \text{inr}(d) : A+B$ **and**

PlusE:

$\bigwedge p\ c\ d\ A\ B\ C. \llbracket p : A+B;$
 $\bigwedge x. x:A \implies c(x) : C(\text{inl}(x));$
 $\bigwedge y. y:B \implies d(y) : C(\text{inr}(y)) \rrbracket \implies \text{when}(p, \lambda x. c(x), \lambda y. d(y)) : C(p)$ **and**

PlusEL:

$\bigwedge p\ q\ c\ d\ e\ f\ A\ B\ C. \llbracket p = q : A+B;$
 $\bigwedge x. x : A \implies c(x) = e(x) : C(\text{inl}(x));$
 $\bigwedge y. y : B \implies d(y) = f(y) : C(\text{inr}(y)) \rrbracket$
 $\implies \text{when}(p, \lambda x. c(x), \lambda y. d(y)) = \text{when}(q, \lambda x. e(x), \lambda y. f(y)) : C(p)$ **and**

PlusC-inl:

$\bigwedge a\ c\ d\ A\ B\ C. \llbracket a : A;$
 $\bigwedge x. x:A \implies c(x) : C(\text{inl}(x));$
 $\bigwedge y. y:B \implies d(y) : C(\text{inr}(y)) \rrbracket$
 $\implies \text{when}(\text{inl}(a), \lambda x. c(x), \lambda y. d(y)) = c(a) : C(\text{inl}(a))$ **and**

PlusC-inr:

$\bigwedge b\ c\ d\ A\ B\ C. \llbracket b : B;$
 $\bigwedge x. x:A \implies c(x) : C(\text{inl}(x));$
 $\bigwedge y. y:B \implies d(y) : C(\text{inr}(y)) \rrbracket$
 $\implies \text{when}(\text{inr}(b), \lambda x. c(x), \lambda y. d(y)) = d(b) : C(\text{inr}(b))$ **and**

— The type *Eq*

$EqF: \bigwedge a\ b\ A. \llbracket A\ \text{type}; a : A; b : A \rrbracket \implies Eq(A,a,b)$ *type* **and**
 $EqFL: \bigwedge a\ b\ c\ d\ A\ B. \llbracket A = B; a = c : A; b = d : A \rrbracket \implies Eq(A,a,b) = Eq(B,c,d)$
and

$EqI: \bigwedge a\ b\ A. a = b : A \implies eq : Eq(A,a,b)$ **and**
 $EqE: \bigwedge p\ a\ b\ A. p : Eq(A,a,b) \implies a = b : A$ **and**

— By equality of types, can prove $C(p)$ from $C(eq)$, an elimination rule

$EqC: \bigwedge p\ a\ b\ A. p : Eq(A,a,b) \implies p = eq : Eq(A,a,b)$ **and**

— The type *F*

$FF: F$ *type* **and**

$FE: \bigwedge p\ C. \llbracket p : F; C\ \text{type} \rrbracket \implies \text{contr}(p) : C$ **and**

$FEL: \bigwedge p\ q\ C. \llbracket p = q : F; C\ \text{type} \rrbracket \implies \text{contr}(p) = \text{contr}(q) : C$ **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**

$TE: \bigwedge p \ c \ C. \llbracket p : T; c : C(tt) \rrbracket \implies c : C(p)$ **and**
 $TEL: \bigwedge p \ q \ c \ d \ C. \llbracket p = q : T; c = d : C(tt) \rrbracket \implies c = d : C(p)$ **and**
 $TC: \bigwedge p. p : T \implies p = tt : T$

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 *equelem*, 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 *equelem*, 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: $p = un = un = \dots$

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*: $\llbracket c = a : A; d = b : B(a) \rrbracket \implies \langle c, d \rangle = \langle a, b \rangle : \text{Sum}(A, B)$
<proof>

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

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

lemma *subst-prodE*:

assumes $p: \text{Prod}(A, B)$
and $a: A$
and $\bigwedge z. z: B(a) \implies c(z): C(z)$
shows $c(p'a): C(p'a)$
<proof>

1.2 Tactics for type checking

$\langle ML \rangle$

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

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

$\langle ML \rangle$

1.3 Simplification

To simplify the type in a goal.

lemma *replace-type*: $\llbracket B = A; a : A \rrbracket \implies a : B$
 $\langle proof \rangle$

Simplify the parameter of a unary type operator.

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

Simplification rules for Constructive Type Theory.

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

$\langle ML \rangle$

1.4 The elimination rules for fst/snd

lemma *SumE-fst*: $p : \text{Sum}(A,B) \implies \text{fst}(p) : A$
 $\langle proof \rangle$

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

lemma *SumE-snd*:
 assumes *major*: $p : \text{Sum}(A,B)$
 and A *type*
 and $\bigwedge x. x:A \implies B(x)$ *type*
 shows $\text{snd}(p) : B(\text{fst}(p))$
 $\langle proof \rangle$

2 The two-element type (booleans and conditionals)

definition *Bool* :: t
 where $\text{Bool} \equiv T+T$

definition $true :: i$
where $true \equiv \text{inl}(tt)$

definition $false :: i$
where $false \equiv \text{inr}(tt)$

definition $cond :: [i,i,i] \Rightarrow i$
where $cond(a,b,c) \equiv \text{when}(a, \lambda-. b, \lambda-. c)$

lemmas $\text{bool-defs} = \text{Bool-def true-def false-def cond-def}$

2.1 Derivation of rules for the type *Bool*

Formation rule.

lemma boolF : *Bool* type
<proof>

Introduction rules for *true*, *false*.

lemma boolI-true : $true : \text{Bool}$
<proof>

lemma boolI-false : $false : \text{Bool}$
<proof>

Elimination rule: typing of *cond*.

lemma boolE : $\llbracket p : \text{Bool}; a : C(\text{true}); b : C(\text{false}) \rrbracket \Longrightarrow \text{cond}(p,a,b) : C(p)$
<proof>

lemma boolEL : $\llbracket p = q : \text{Bool}; a = c : C(\text{true}); b = d : C(\text{false}) \rrbracket$
 $\Longrightarrow \text{cond}(p,a,b) = \text{cond}(q,c,d) : C(p)$
<proof>

Computation rules for *true*, *false*.

lemma boolC-true : $\llbracket a : C(\text{true}); b : C(\text{false}) \rrbracket \Longrightarrow \text{cond}(\text{true},a,b) = a : C(\text{true})$
<proof>

lemma boolC-false : $\llbracket a : C(\text{true}); b : C(\text{false}) \rrbracket \Longrightarrow \text{cond}(\text{false},a,b) = b : C(\text{false})$
<proof>

3 Elementary arithmetic

3.1 Arithmetic operators and their definitions

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

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

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 $a\#*b \equiv \text{rec}(a, 0, \lambda u v. b \#+ v)$

definition *mod* :: $[i,i] \Rightarrow i$ (**infixr** *mod* 70)

where $a \text{ mod } b \equiv \text{rec}(a, 0, \lambda u v. \text{rec}(\text{succ}(v) \text{ } |-| \ b, 0, \lambda x y. \text{succ}(v)))$

definition *div* :: $[i,i] \Rightarrow i$ (**infixr** *div* 70)

where $a \text{ div } b \equiv \text{rec}(a, 0, \lambda u v. \text{rec}(\text{succ}(u) \text{ mod } \ b, \text{succ}(v), \lambda x y. v))$

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*: $\llbracket a:N; b:N \rrbracket \Longrightarrow a \#+ b : N$

<proof>

lemma *add-typingL*: $\llbracket a = c:N; b = d:N \rrbracket \Longrightarrow a \#+ b = c \#+ d : N$

<proof>

Computation for *add*: 0 and successor cases.

lemma *addC0*: $b:N \Longrightarrow 0 \#+ b = b : N$

<proof>

lemma *addC-succ*: $\llbracket a:N; b:N \rrbracket \Longrightarrow \text{succ}(a) \#+ b = \text{succ}(a \#+ b) : N$

<proof>

3.2.2 Multiplication

Typing of *mult*: short and long versions.

lemma *mult-typing*: $\llbracket a:N; b:N \rrbracket \Longrightarrow a \#* b : N$

<proof>

lemma *mult-typingL*: $\llbracket a = c:N; b = d:N \rrbracket \Longrightarrow a \#* b = c \#* d : N$

<proof>

Computation for *mult*: 0 and successor cases.

lemma *multC0*: $b:N \Longrightarrow 0 \#* b = 0 : N$

<proof>

lemma *multC-succ*: $\llbracket a:N; b:N \rrbracket \Longrightarrow \text{succ}(a) \#* b = b \#+ (a \#* b) : N$

<proof>

3.2.3 Difference

Typing of difference.

lemma *diff-typing*: $\llbracket a:N; b:N \rrbracket \Longrightarrow a - b : N$
<proof>

lemma *diff-typingL*: $\llbracket a = c:N; b = d:N \rrbracket \Longrightarrow a - b = c - d : N$
<proof>

Computation for difference: 0 and successor cases.

lemma *diffC0*: $a:N \Longrightarrow a - 0 = a : N$
<proof>

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

lemma *diff-0-eq-0*: $b:N \Longrightarrow 0 - b = 0 : N$
<proof>

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

lemma *diff-succ-succ*: $\llbracket a:N; b:N \rrbracket \Longrightarrow \text{succ}(a) - \text{succ}(b) = a - b : N$
<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*: $\llbracket a:N; b:N; c:N \rrbracket \Longrightarrow (a \#+ b) \#+ c = a \#+ (b \#+ c) : N$
<proof>

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

lemma *add-commute*: $\llbracket a:N; b:N \rrbracket \Longrightarrow a \#+ b = b \#+ a : N$
<proof>

3.5 Multiplication

Right annihilation in product.

lemma *mult-0-right*: $a:N \implies a \#* 0 = 0 : N$
 $\langle proof \rangle$

Right successor law for multiplication.

lemma *mult-succ-right*: $\llbracket a:N; b:N \rrbracket \implies a \#* succ(b) = a \#+ (a \#* b) : N$
 $\langle proof \rangle$

Commutative law for multiplication.

lemma *mult-commute*: $\llbracket a:N; b:N \rrbracket \implies a \#* b = b \#* a : N$
 $\langle proof \rangle$

Addition distributes over multiplication.

lemma *add-mult-distrib*: $\llbracket a:N; b:N; c:N \rrbracket \implies (a \#+ b) \#* c = (a \#* c) \#+ (b \#* c) : N$
 $\langle proof \rangle$

Associative law for multiplication.

lemma *mult-assoc*: $\llbracket a:N; b:N; c:N \rrbracket \implies (a \#* b) \#* c = a \#* (b \#* c) : N$
 $\langle proof \rangle$

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*: $a:N \implies a - a = 0 : N$
 $\langle proof \rangle$

lemma *add-0-right*: $\llbracket c : N; 0 : N; c : N \rrbracket \implies c \#+ 0 = c : N$
 $\langle proof \rangle$

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*:
 $b:N \implies ?a : \prod x:N. Eq(N, b-x, 0) \longrightarrow Eq(N, b \#+ (x-b), x)$
 $\langle proof \rangle$

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*: $\llbracket a:N; b:N; b - a = 0 : N \rrbracket \Longrightarrow b \# + (a - b) = a : N$
 $\langle \text{proof} \rangle$

3.7 Absolute difference

Typing of absolute difference: short and long versions.

lemma *absdiff-typing*: $\llbracket a:N; b:N \rrbracket \Longrightarrow a \text{ |-| } b : N$
 $\langle \text{proof} \rangle$

lemma *absdiff-typingL*: $\llbracket a = c:N; b = d:N \rrbracket \Longrightarrow a \text{ |-| } b = c \text{ |-| } d : N$
 $\langle \text{proof} \rangle$

lemma *absdiff-self-eq-0*: $a:N \Longrightarrow a \text{ |-| } a = 0 : N$
 $\langle \text{proof} \rangle$

lemma *absdiffC0*: $a:N \Longrightarrow 0 \text{ |-| } a = a : N$
 $\langle \text{proof} \rangle$

lemma *absdiff-succ-succ*: $\llbracket a:N; b:N \rrbracket \Longrightarrow \text{succ}(a) \text{ |-| } \text{succ}(b) = a \text{ |-| } b : N$
 $\langle \text{proof} \rangle$

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

lemma *absdiff-commute*: $\llbracket a:N; b:N \rrbracket \Longrightarrow a \text{ |-| } b = b \text{ |-| } a : N$
 $\langle \text{proof} \rangle$

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

schematic-goal *add-eq0-lemma*: $\llbracket a:N; b:N \rrbracket \Longrightarrow ?c : \text{Eq}(N, a \# + b, 0) \longrightarrow \text{Eq}(N, a, 0)$
 $\langle \text{proof} \rangle$

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

lemma *add-eq0*: $\llbracket a:N; b:N; a \# + b = 0 : N \rrbracket \Longrightarrow a = 0 : N$
 $\langle \text{proof} \rangle$

Here is a lemma to infer $a - b = 0$ and $b - a = 0$ from $a \text{ |-| } b = 0$, below.

schematic-goal *absdiff-eq0-lem*:
 $\llbracket a:N; b:N; a \text{ |-| } b = 0 : N \rrbracket \Longrightarrow ?a : \text{Eq}(N, a - b, 0) \times \text{Eq}(N, b - a, 0)$
 $\langle \text{proof} \rangle$

If $a \text{ |-| } b = 0$ then $a = b$ proof: $a - b = 0$ and $b - a = 0$, so $b = a + (b - a) = a + 0 = a$.

lemma *absdiff-eq0*: $\llbracket a \text{ |-| } b = 0 : N; a:N; b:N \rrbracket \Longrightarrow a = b : N$
 $\langle \text{proof} \rangle$

3.8 Remainder and Quotient

Typing of remainder: short and long versions.

lemma *mod-typing*: $\llbracket a:N; b:N \rrbracket \Longrightarrow a \text{ mod } b : N$
<proof>

lemma *mod-typingL*: $\llbracket a = c:N; b = d:N \rrbracket \Longrightarrow a \text{ mod } b = c \text{ mod } d : N$
<proof>

Computation for *mod*: 0 and successor cases.

lemma *modC0*: $b:N \Longrightarrow 0 \text{ mod } b = 0 : N$
<proof>

lemma *modC-succ*: $\llbracket a:N; b:N \rrbracket \Longrightarrow$
 $\text{succ}(a) \text{ mod } b = \text{rec}(\text{succ}(a \text{ mod } b) \mid - \mid b, 0, \lambda x y. \text{succ}(a \text{ mod } b)) : N$
<proof>

Typing of quotient: short and long versions.

lemma *div-typing*: $\llbracket a:N; b:N \rrbracket \Longrightarrow a \text{ div } b : N$
<proof>

lemma *div-typingL*: $\llbracket a = c:N; b = d:N \rrbracket \Longrightarrow a \text{ div } b = c \text{ div } d : N$
<proof>

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

Computation for quotient: 0 and successor cases.

lemma *divC0*: $b:N \Longrightarrow 0 \text{ div } b = 0 : N$
<proof>

lemma *divC-succ*: $\llbracket a:N; b:N \rrbracket \Longrightarrow$
 $\text{succ}(a) \text{ div } b = \text{rec}(\text{succ}(a \text{ mod } b), \text{succ}(a \text{ div } b), \lambda x y. a \text{ div } b) : N$
<proof>

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

lemma *divC-succ2*: $\llbracket a:N; b:N \rrbracket \Longrightarrow$
 $\text{succ}(a) \text{ div } b = \text{rec}(\text{succ}(a \text{ mod } b) \mid - \mid b, \text{succ}(a \text{ div } b), \lambda x y. a \text{ div } b) : N$
<proof>

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

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

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

lemma *mod-div-equality*: $\llbracket a:N; b:N \rrbracket \Longrightarrow a \text{ mod } b \# + (a \text{ div } b) \# * b = a : N$
<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  $(\lambda u. split(u, \lambda v w. v)) \text{ ' } \langle 0, succ(0) \rangle : ?A$ 
  <proof>
```

typechecking the predecessor function

```
schematic-goal  $\lambda n. rec(n, 0, \lambda x y. x) : ?A$ 
  <proof>
```

typechecking the addition function

```
schematic-goal  $\lambda n. \lambda m. rec(n, m, \lambda x y. succ(y)) : ?A$ 
  <proof>
```

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

<ML>

```

schematic-goal  $\lambda w. \langle w, w \rangle : ?A$ 
   $\langle proof \rangle$ 

schematic-goal  $\lambda x. \lambda y. x : ?A$ 
   $\langle proof \rangle$ 

typechecking fst (as a function object)
schematic-goal  $\lambda i. split(i, \lambda j k. j) : ?A$ 
   $\langle proof \rangle$ 

end

```

5 Examples with elimination rules

```

theory Elimination
imports ../CTT
begin

```

This finds the functions fst and snd!

```

schematic-goal [folded basic-defs]:  $A \text{ type} \implies ?a : (A \times A) \longrightarrow A$ 
   $\langle proof \rangle$ 

```

```

schematic-goal [folded basic-defs]:  $A \text{ type} \implies ?a : (A \times A) \longrightarrow A$ 
   $\langle proof \rangle$ 

```

Double negation of the Excluded Middle

```

schematic-goal  $A \text{ type} \implies ?a : ((A + (A \longrightarrow F)) \longrightarrow F) \longrightarrow F$ 
   $\langle proof \rangle$ 

```

Experiment: the proof above in Isar

lemma

```

  assumes  $A \text{ type}$  shows  $(\lambda f. f \text{ ' } inr(\lambda y. f \text{ ' } inl(y))) : ((A + (A \longrightarrow F)) \longrightarrow F) \longrightarrow F$ 
   $\langle proof \rangle$ 

```

```

schematic-goal [ $A \text{ type}; B \text{ type}$ ]:  $?a : (A \times B) \longrightarrow (B \times A)$ 
   $\langle proof \rangle$ 

```

Binary sums and products

```

schematic-goal [ $A \text{ type}; B \text{ type}; C \text{ type}$ ]:  $?a : (A + B \longrightarrow C) \longrightarrow (A \longrightarrow C) \times (B \longrightarrow C)$ 
   $\langle proof \rangle$ 

```

```

schematic-goal [ $A \text{ type}; B \text{ type}; C \text{ type}$ ]:  $?a : A \times (B + C) \longrightarrow (A \times B + A \times C)$ 
   $\langle proof \rangle$ 

```


schematic-goalassumes A typeand $\bigwedge x. x:A \implies B(x)$ typeand $\bigwedge x. x:A \implies C(x)$ typeshows $?a : (\sum x:A. B(x) + C(x)) \longrightarrow (\sum x:A. B(x)) + (\sum x:A. C(x))$ \langle proof \rangle

Construction of the currying functional

schematic-goal $\llbracket A$ type; B type; C type $\rrbracket \implies ?a : (A \times B \longrightarrow C) \longrightarrow (A \longrightarrow (B \longrightarrow C))$ \langle proof \rangle **schematic-goal**assumes A typeand $\bigwedge x. x:A \implies B(x)$ typeand $\bigwedge z. z: (\sum x:A. B(x)) \implies C(z)$ typeshows $?a : \prod f: (\prod z: (\sum x:A . B(x)) . C(z)).$
 $(\prod x:A . \prod y:B(x) . C(\langle x,y \rangle))$ \langle proof \rangle

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

schematic-goal $\llbracket A$ type; B type; C type $\rrbracket \implies ?a : (A \longrightarrow (B \longrightarrow C)) \longrightarrow (A \times B \longrightarrow C)$ \langle proof \rangle **schematic-goal**assumes A typeand $\bigwedge x. x:A \implies B(x)$ typeand $\bigwedge z. z: (\sum x:A . B(x)) \implies C(z)$ typeshows $?a : (\prod x:A . \prod y:B(x) . C(\langle x,y \rangle))$
 $\longrightarrow (\prod z: (\sum x:A . B(x)) . C(z))$ \langle proof \rangle

Function application

schematic-goal $\llbracket A$ type; B type $\rrbracket \implies ?a : ((A \longrightarrow B) \times A) \longrightarrow B$ \langle proof \rangle

Basic test of quantifier reasoning

schematic-goalassumes A typeand B typeand $\bigwedge x y. \llbracket x:A; y:B \rrbracket \implies C(x,y)$ 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 $\bigwedge x. x:A \implies B(x)$ type
and $\bigwedge x y. \llbracket x:A; y:B(x) \rrbracket \implies C(x,y)$ 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 $\bigwedge z. z: A+B \implies C(z)$ type
shows $?a : (\prod x:A. C(\text{inl}(x))) \longrightarrow (\prod y:B. C(\text{inr}(y)))$
 $\longrightarrow (\prod z: A+B. C(z))$
 ⟨proof⟩

schematic-goal [folded basic-defs]:

$\llbracket A \text{ type}; B \text{ type}; C \text{ type} \rrbracket \implies ?a : (A \longrightarrow B \times C) \longrightarrow (A \longrightarrow B) \times (A \longrightarrow C)$
 ⟨proof⟩

AXIOM OF CHOICE! Delicate use of elimination rules

schematic-goal

assumes A type
and $\bigwedge x. x:A \implies B(x)$ type
and $\bigwedge x y. \llbracket x:A; y:B(x) \rrbracket \implies C(x,y)$ 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 $\bigwedge x. x:A \implies B(x)$ type
and $\bigwedge x y. \llbracket x:A; y:B(x) \rrbracket \implies C(x,y)$ type
shows $(\lambda f. \langle \lambda x. \text{fst}(f'x), \lambda x. \text{snd}(f'x) \rangle)$
 $: (\prod x:A. \sum y:B(x). C(x,y)) \longrightarrow (\sum f: (\prod x:A. B(x)). \prod x:A. C(x, f'x))$
 ⟨proof⟩

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

schematic-goal [folded basic-defs]:

assumes A type
and $\bigwedge x. x:A \implies B(x)$ type

and $\bigwedge x y. \llbracket x:A; y:B(x) \rrbracket \implies C(x,y)$ *type*
shows $?a : (\prod x:A. \sum y:B(x). C(x,y)) \longrightarrow (\sum f: (\prod x:A. B(x)). \prod x:A. C(x, f(x)))$
 $\langle proof \rangle$

Example of sequent-style deduction

schematic-goal

assumes A *type*

and B *type*

and $\bigwedge z. z:A \times B \implies C(z)$ *type*

shows $?a : (\sum z:A \times B. C(z)) \longrightarrow (\sum u:A. \sum v:B. C(\langle u,v \rangle))$

$\langle proof \rangle$

end

6 Equality reasoning by rewriting

theory *Equality*

imports $../CTT$

begin

lemma *split-eq*: $p : Sum(A,B) \implies split(p,pair) = p : Sum(A,B)$

$\langle proof \rangle$

lemma *when-eq*: $\llbracket A \text{ type}; B \text{ type}; p : A+B \rrbracket \implies when(p,inl,inr) = p : A + B$

$\langle proof \rangle$

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

lemma $p:N \implies rec(p,0, \lambda y z. succ(y)) = p : N$

$\langle proof \rangle$

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

lemma $p:N \implies rec(p,0, \lambda y z. succ(z)) = p : N$

$\langle proof \rangle$

Associativity of addition

lemma $\llbracket a:N; b:N; c:N \rrbracket$

$\implies rec(rec(a, b, \lambda x y. succ(y)), c, \lambda x y. succ(y)) =$

$rec(a, rec(b, c, \lambda x y. succ(y)), \lambda x y. succ(y)) : N$

$\langle proof \rangle$

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

lemma $p : Sum(A,B) \implies \langle split(p,\lambda x y. x), split(p,\lambda x y. y) \rangle = p : Sum(A,B)$

$\langle proof \rangle$

lemma $\llbracket a : A; b : B \rrbracket \implies (\lambda u. split(u, \lambda v w. \langle w,v \rangle)) \text{ ' } \langle a,b \rangle = \langle b,a \rangle : \sum x:B.$

A

<proof>

a contrived, complicated simplication, requires sum-elimination also

lemma $(\lambda f. \lambda x. f'(f'x)) \text{ ' } (\lambda u. \text{split}(u, \lambda v w. \langle w, v \rangle)) =$
 $\lambda x. x : \prod x: (\sum y:N. N). (\sum y:N. N)$
<proof>

end

7 Synthesis examples, using a crude form of narrowing

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

discovery of predecessor function

schematic-goal $?a : \sum \text{pred}: ?A . \text{Eq}(N, \text{pred}'0, 0) \times (\prod n:N. \text{Eq}(N, \text{pred}' \text{succ}(n), n))$
<proof>

the function fst as an element of a function type

schematic-goal [*folded basic-defs*]:
 $A \text{ type} \implies ?a : \sum f: ?B . \prod i:A. \prod j:A. \text{Eq}(A, f \text{ ' } \langle i, j \rangle, i)$
<proof>

An interesting use of the eliminator, when

schematic-goal $?a : \prod i:N. \text{Eq}(?A, ?b(\text{inl}(i)), \langle 0, i \rangle)$
 $\times \text{Eq}(?A, ?b(\text{inr}(i)), \langle \text{succ}(0), i \rangle)$
<proof>

schematic-goal $?a : \prod i:N. \text{Eq}(?A(i), ?b(\text{inl}(i)), \langle 0, i \rangle)$
 $\times \text{Eq}(?A(i), ?b(\text{inr}(i)), \langle \text{succ}(0), i \rangle)$
<proof>

A tricky combination of when and split

schematic-goal [*folded basic-defs*]:
 $?a : \prod i:N. \prod j:N. \text{Eq}(?A, ?b(\text{inl}(\langle i, j \rangle)), i)$
 $\times \text{Eq}(?A, ?b(\text{inr}(\langle i, j \rangle)), j)$
<proof>

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

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

Deriving the addition operator

schematic-goal [*folded arith-defs*]:
 $?c : \prod n:N. Eq(N, ?f(0, n), n)$
 $\times (\prod m:N. Eq(N, ?f(succ(m), n), succ(?f(m, n))))$
 $\langle proof \rangle$

The addition function – using explicit lambdas

schematic-goal [*folded arith-defs*]:
 $?c : \sum plus : ?A .$
 $\prod x:N. Eq(N, plus'0'x, x)$
 $\times (\prod y:N. Eq(N, plus'succ(y)'x, succ(plus'y'x)))$
 $\langle proof \rangle$

end