

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	8
1.4	The elimination rules for <code>fst/snd</code>	10
2	The two-element type (booleans and conditionals)	11
2.1	Derivation of rules for the type <code>Bool</code>	11
3	Elementary arithmetic	12
3.1	Arithmetic operators and their definitions	12
3.2	Proofs about elementary arithmetic: addition, multiplication, etc.	13
3.2.1	Addition	13
3.2.2	Multiplication	13
3.2.3	Difference	13
3.3	Simplification	14
3.4	Addition	15
3.5	Multiplication	15
3.6	Difference	16
3.7	Absolute difference	17
3.8	Remainder and Quotient	18
4	Easy examples: type checking and type deduction	20
4.1	Single-step proofs: verifying that a type is well-formed	20
4.2	Multi-step proofs: Type inference	20
5	Examples with elimination rules	21
6	Equality reasoning by rewriting	26

```

theory CTT
imports Pure
begin

```

1 Constructive Type Theory: axiomatic basis

ML-file $\langle \sim\sim / \text{src} / \text{Provers} / \text{typedsimp} . \text{ML} \rangle$

setup *Pure-Thy.old-appl-syntax-setup*

```

typedecl i
typedecl t
typedecl o

```

consts

— Judgments

```

Type   ::  $t \Rightarrow prop$            ((- type) [10] 5)
Eqtype  ::  $[t, t] \Rightarrow prop$        ((- =/ -) [10,10] 5)
Elem    ::  $[i, t] \Rightarrow prop$        ((- /: -) [10,10] 5)
Eqelem  ::  $[i, i, t] \Rightarrow prop$     ((- =/ - :/ -) [10,10,10] 5)
Reduce  ::  $[i, i] \Rightarrow prop$        (Reduce[-,-])

```

— Types for truth values

```

F       ::  $t$ 
T       ::  $t$            — F is empty, T contains one element
contr   ::  $i \Rightarrow i$ 
tt      ::  $i$ 

```

— Natural numbers

```

N       ::  $t$ 
Zero   ::  $i$            (0)
succ   ::  $i \Rightarrow i$ 
rec    ::  $[i, i, [i, i] \Rightarrow i] \Rightarrow i$ 

```

— Binary sum

```

Plus   ::  $[t, t] \Rightarrow t$            (infixr + 40)
inl    ::  $i \Rightarrow i$ 
inr    ::  $i \Rightarrow i$ 
when   ::  $[i, i \Rightarrow i, i \Rightarrow i] \Rightarrow i$ 

```

— General sum and binary product

```

Sum    ::  $[t, i \Rightarrow t] \Rightarrow t$ 
pair   ::  $[i, i] \Rightarrow i$            ((1<-,-/>-))
fst    ::  $i \Rightarrow i$ 
snd    ::  $i \Rightarrow i$ 
split  ::  $[i, [i, i] \Rightarrow i] \Rightarrow i$ 

```

— General product and function space

```

Prod   ::  $[t, i \Rightarrow t] \Rightarrow t$ 
lambda ::  $(i \Rightarrow i) \Rightarrow i$       (binder  $\lambda$  10)
app   ::  $[i, i] \Rightarrow i$           (infixl ‘ 60)

```

— Equality type

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

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

syntax

$-PROD \quad :: [idt, t, t] \Rightarrow t \quad ((\exists \prod \text{-:./ -}) 10)$
 $-SUM \quad :: [idt, t, t] \Rightarrow t \quad ((\exists \sum \text{-:./ -}) 10)$

translations

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

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

abbreviation *Times* $:: [t, t] \Rightarrow t$ (**infixr** \times 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: $\bigwedge a b c A. \llbracket a = b : A; Reduce[b, c] \rrbracket \Longrightarrow a = c : A$ **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: $\bigwedge a\ b\ A. \llbracket A\ \text{type}; a : A; b : A \rrbracket \implies \text{Eq}(A, a, b)$ *type* **and**

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

EqI: $\bigwedge a\ b\ A. a = b : A \implies \text{eq} : \text{Eq}(A, a, b)$ **and**

EqE: $\bigwedge p\ a\ b\ A. p : \text{Eq}(A, a, b) \implies a = b : A$ **and**

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

EqC: $\bigwedge p\ a\ b\ A. p : \text{Eq}(A, a, b) \implies p = \text{eq} : \text{Eq}(A, a, b)$ **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 $form\text{-}rls = NF\ ProdF\ SumF\ PlusF\ EqF\ FF\ TF$
and $formL\text{-}rls = ProdFL\ SumFL\ PlusFL\ EqFL$

Introduction rules. OMITTED:

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

lemmas $intr\text{-}rls = NI0\ NI\text{-}succ\ ProdI\ SumI\ PlusI\text{-}inl\ PlusI\text{-}inr\ TI$
and $intrL\text{-}rls = NI\text{-}succL\ ProdIL\ SumIL\ PlusI\text{-}inlL\ PlusI\text{-}inrL$

Elimination rules. OMITTED:

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

lemmas $elim\text{-}rls = NE\ ProdE\ SumE\ PlusE\ FE$
and $elimL\text{-}rls = NEL\ ProdEL\ SumEL\ PlusEL\ FEL$

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

lemmas $comp\text{-}rls = NC0\ NC\text{-}succ\ ProdC\ SumC\ PlusC\text{-}inl\ PlusC\text{-}inr$

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

lemmas $element\text{-}rls = intr\text{-}rls\ elim\text{-}rls$

Definitions are (meta)equality axioms.

lemmas $basic\text{-}defs = fst\text{-}def\ snd\text{-}def$

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

lemma $SumIL2: \llbracket c = a : A; d = b : B(a) \rrbracket \implies \langle c, d \rangle = \langle a, b \rangle : Sum(A, B)$
by (*rule sym-elem*) (*rule SumIL*; *rule sym-elem*)

lemmas $intrL2\text{-}rls = NI\text{-}succL\ ProdIL\ SumIL2\ PlusI\text{-}inlL\ PlusI\text{-}inrL$

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

lemma $subst\text{-}prodE$:

```

assumes p: Prod(A,B)
  and a: A
  and  $\bigwedge z. z: B(a) \implies c(z): C(z)$ 
shows c(p'a): C(p'a)
by (rule assms ProdE)+

```

1.2 Tactics for type checking

```

ML <
  local

```

```

  fun is-rigid-elem Const-<Elem for a -> = not(is-Var (head-of a))
    | is-rigid-elem Const-<Eelem for a -> = not(is-Var (head-of a))
    | is-rigid-elem Const-<Type for a> = not(is-Var (head-of a))
    | is-rigid-elem - = false

```

```

  in

```

```

  (*Try solving a:A or a=b:A by assumption provided a is rigid!*)
  fun test-assume-tac ctxt = SUBGOAL (fn (prem, i) =>
    if is-rigid-elem (Logic.strip-assums-concl prem)
    then assume-tac ctxt i else no-tac)

```

```

  fun ASSUME ctxt tf i = test-assume-tac ctxt i ORELSE tf i

```

```

  end
  >

```

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

```

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

```

```

ML <

```

```

  fun routine-tac rls ctxt prems =
    ASSUME ctxt (filt-resolve-from-net-tac ctxt 4 (Tactic.build-net (prems @ rls)));

```

```

  (*Solve all subgoals A type using formation rules. *)
  val form-net = Tactic.build-net @ {thms form-rls};
  fun form-tac ctxt =
    REPEAT-FIRST (ASSUME ctxt (filt-resolve-from-net-tac ctxt 1 form-net));

```

```

  (*Type checking: solve a:A (a rigid, A flexible) by intro and elim rules. *)
  fun typechk-tac ctxt thms =
    let val tac =
      filt-resolve-from-net-tac ctxt 3
      (Tactic.build-net (thms @ @ {thms form-rls} @ @ {thms element-rls}))
    in REPEAT-FIRST (ASSUME ctxt tac) end

```

```

  (*Solve a:A (a flexible, A rigid) by introduction rules.

```



```

    Cannot use stringtrees (filt-resolve-tac) since
    goals like ?a:SUM(A,B) have a trivial head-string *)
fun intr-tac ctxt thms =
  let val tac =
    filt-resolve-from-net-tac ctxt 1
    (Tactic.build-net (thms @ @{thms form-rls} @ @{thms intr-rls}))
  in REPEAT-FIRST (ASSUME ctxt tac) end

(*Equality proving: solve a=b:A (where a is rigid) by long rules. *)
fun equal-tac ctxt thms =
  REPEAT-FIRST
  (ASSUME ctxt
   (filt-resolve-from-net-tac ctxt 3
    (Tactic.build-net (thms @ @{thms form-rls element-rls intrL-rls elimL-rls
    refl-elem}))))
  >

```

```

method-setup form = <Scan.succeed (fn ctxt => SIMPLE-METHOD (form-tac
  ctxt))>
method-setup typechk = <Attrib.thms >> (fn ths => fn ctxt => SIMPLE-METHOD
  (typechk-tac ctxt ths))>
method-setup intr = <Attrib.thms >> (fn ths => fn ctxt => SIMPLE-METHOD
  (intr-tac ctxt ths))>
method-setup equal = <Attrib.thms >> (fn ths => fn ctxt => SIMPLE-METHOD
  (equal-tac ctxt ths))>

```

1.3 Simplification

To simplify the type in a goal.

```

lemma replace-type:  $\llbracket B = A; a : A \rrbracket \implies a : B$ 
  apply (rule equal-types)
  apply (rule-tac [2] sym-type)
  apply assumption+
  done

```

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  $B(a) = B(c)$ 
  apply (rule subst-typeL)
  apply (rule-tac [2] refl-type)
  apply (rule 1)
  apply (erule 2)
  done

```

Simplification rules for Constructive Type Theory.

```

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

```

```

ML <
(*Converts each goal  $e : Eq(A,a,b)$  into  $a=b:A$  for simplification.
  Uses other intro rules to avoid changing flexible goals.*)
val eqintr-net = Tactic.build-net @{thms EqI intr-rls}
fun eqintr-tac ctxt =
  REPEAT-FIRST (ASSUME ctxt (filt-resolve-from-net-tac ctxt 1 eqintr-net))

(** Tactics that instantiate CTT-rules.
  Vars in the given terms will be incremented!
  The (rtac EqE i) lets them apply to equality judgments. **)

fun NE-tac ctxt sp i =
  TRY (resolve-tac ctxt @{thms EqE} i) THEN
  Rule-Insts.res-inst-tac ctxt [((p, 0), Position.none), sp] [] @{thm NE} i

fun SumE-tac ctxt sp i =
  TRY (resolve-tac ctxt @{thms EqE} i) THEN
  Rule-Insts.res-inst-tac ctxt [((p, 0), Position.none), sp] [] @{thm SumE} i

fun PlusE-tac ctxt sp i =
  TRY (resolve-tac ctxt @{thms EqE} i) THEN
  Rule-Insts.res-inst-tac ctxt [((p, 0), Position.none), sp] [] @{thm PlusE} i

(** Predicate logic reasoning, WITH THINNING!! Procedures adapted from NJ.
**)

(*Finds  $f:Prod(A,B)$  and  $a:A$  in the assumptions, concludes there is  $z:B(a)$  *)
fun add-mp-tac ctxt i =
  resolve-tac ctxt @{thms subst-prodE} i THEN assume-tac ctxt i THEN as-
  sume-tac ctxt i

(*Finds  $P \rightarrow Q$  and  $P$  in the assumptions, replaces implication by  $Q$  *)
fun mp-tac ctxt i = eresolve-tac ctxt @{thms subst-prodE} i THEN assume-tac
  ctxt i

(*safe when regarded as predicate calculus rules*)
val safe-brls = sort (make-ord lessb)
  [ (true, @{thm FE}), (true, asm-rl),
    (false, @{thm ProdI}), (true, @{thm SumE}), (true, @{thm PlusE}) ]

val unsafe-brls =
  [ (false, @{thm PlusI-inl}), (false, @{thm PlusI-inr}), (false, @{thm SumI}),
    (true, @{thm subst-prodE}) ]

(*0 subgoals vs 1 or more*)
val (safe0-brls, safep-brls) =
  List.partition (curry (op =) 0 o subgoals-of-brl) safe-brls

```

```

fun safestep-tac ctxt thms i =
  form-tac ctxt ORELSE
  resolve-tac ctxt thms i ORELSE
  biresolve-tac ctxt safe0-brls i ORELSE mp-tac ctxt i ORELSE
  DETERM (biresolve-tac ctxt safestep-brls i)

fun safe-tac ctxt thms i = DEPTH-SOLVE-1 (safestep-tac ctxt thms i)

fun step-tac ctxt thms = safestep-tac ctxt thms ORELSE' biresolve-tac ctxt un-
safe-brls

(*Fails unless it solves the goal!*)
fun pc-tac ctxt thms = DEPTH-SOLVE-1 o (step-tac ctxt thms)
>

method-setup eqintr = <Scan.succeed (SIMPLE-METHOD o eqintr-tac)>
method-setup NE = <
  Scan.lift Parse.embedded-inner-syntax >> (fn s => fn ctxt => SIMPLE-METHOD'
  (NE-tac ctxt s))
>
method-setup pc = <Attrib.thms >> (fn ths => fn ctxt => SIMPLE-METHOD'
  (pc-tac ctxt ths))
method-setup add-mp = <Scan.succeed (SIMPLE-METHOD' o add-mp-tac)>

ML-file <rew.ML>
method-setup rew = <Attrib.thms >> (fn ths => fn ctxt => SIMPLE-METHOD
  (rew-tac ctxt ths))
method-setup hyp-rew = <Attrib.thms >> (fn ths => fn ctxt => SIMPLE-METHOD
  (hyp-rew-tac ctxt ths))

```

1.4 The elimination rules for fst/snd

```

lemma SumE-fst: p : Sum(A,B) ==> fst(p) : A
  unfolding basic-defs
  apply (erule SumE)
  apply assumption
  done

```

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

```

lemma SumE-snd:
  assumes major: p : Sum(A,B)
  and A type
  and  $\bigwedge x. x:A \implies B(x)$  type
  shows snd(p) : B(fst(p))
  unfolding basic-defs
  apply (rule major [THEN SumE])
  apply (rule SumC [THEN subst-eqtyparg, THEN replace-type])
  apply (typechk assms)
  done

```

2 The two-element type (booleans and conditionals)

definition *Bool* :: *t*
 where *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 *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
 unfolding *bool-defs* **by** *typechk*

Introduction rules for *true*, *false*.

lemma *boolI-true*: *true* : *Bool*
 unfolding *bool-defs* **by** *typechk*

lemma *boolI-false*: *false* : *Bool*
 unfolding *bool-defs* **by** *typechk*

Elimination rule: typing of *cond*.

lemma *boolE*: $\llbracket p:\text{Bool}; a : C(\text{true}); b : C(\text{false}) \rrbracket \Longrightarrow \text{cond}(p,a,b) : C(p)$
 unfolding *bool-defs*
 apply (*typechk*; *erule TE*)
 apply *typechk*
 done

lemma *boolEL*: $\llbracket p = q : \text{Bool}; a = c : C(\text{true}); b = d : C(\text{false}) \rrbracket$
 $\Longrightarrow \text{cond}(p,a,b) = \text{cond}(q,c,d) : C(p)$
 unfolding *bool-defs*
 apply (*rule PlusEL*)
 apply (*erule asm-rl refl-elem [THEN TEL]*)
 done

Computation rules for *true*, *false*.

lemma *boolC-true*: $\llbracket a : C(\text{true}); b : C(\text{false}) \rrbracket \Longrightarrow \text{cond}(\text{true},a,b) = a : C(\text{true})$
 unfolding *bool-defs*
 apply (*rule comp-rls*)

```

  apply typechk
  apply (erule-tac [!] TE)
  apply typechk
done

```

```

lemma boolC-false:  $\llbracket a : C(true); b : C(false) \rrbracket \implies cond(false,a,b) = b : C(false)$ 
  unfolding bool-defs
  apply (rule comp-rls)
  apply typechk
  apply (erule-tac [!] TE)
  apply typechk
done

```

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

```

```

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

```

```

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:  $\llbracket a:N; b:N \rrbracket \implies a \# + b : N$ 
  unfolding arith-defs by typechk

```

```

lemma add-typingL:  $\llbracket a = c:N; b = d:N \rrbracket \implies a \# + b = c \# + d : N$ 
  unfolding arith-defs by equal

```

Computation for *add*: 0 and successor cases.

lemma *addC0*: $b:N \implies 0 \#+ b = b : N$
unfolding *arith-defs* **by** *rew*

lemma *addC-succ*: $\llbracket a:N; b:N \rrbracket \implies \text{succ}(a) \#+ b = \text{succ}(a \#+ b) : N$
unfolding *arith-defs* **by** *rew*

3.2.2 Multiplication

Typing of *mult*: short and long versions.

lemma *mult-typing*: $\llbracket a:N; b:N \rrbracket \implies a \#* b : N$
unfolding *arith-defs* **by** (*typechk add-typing*)

lemma *mult-typingL*: $\llbracket a = c:N; b = d:N \rrbracket \implies a \#* b = c \#* d : N$
unfolding *arith-defs* **by** (*equal add-typingL*)

Computation for *mult*: 0 and successor cases.

lemma *multC0*: $b:N \implies 0 \#* b = 0 : N$
unfolding *arith-defs* **by** *rew*

lemma *multC-succ*: $\llbracket a:N; b:N \rrbracket \implies \text{succ}(a) \#* b = b \#+ (a \#* b) : N$
unfolding *arith-defs* **by** *rew*

3.2.3 Difference

Typing of difference.

lemma *diff-typing*: $\llbracket a:N; b:N \rrbracket \implies a - b : N$
unfolding *arith-defs* **by** *typechk*

lemma *diff-typingL*: $\llbracket a = c:N; b = d:N \rrbracket \implies a - b = c - d : N$
unfolding *arith-defs* **by** *equal*

Computation for difference: 0 and successor cases.

lemma *diffC0*: $a:N \implies a - 0 = a : N$
unfolding *arith-defs* **by** *rew*

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

lemma *diff-0-eq-0*: $b:N \implies 0 - b = 0 : N$
unfolding *arith-defs*
by (*NE b*) *hyp-rew*

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*: $\llbracket a:N; b:N \rrbracket \implies \text{succ}(a) - \text{succ}(b) = a - b : N$
unfolding *arith-defs*
apply *hyp-rew*

```

apply (NE b)
  apply hyp-rew
done

```

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 ‹

```

structure Arith-simp = TSimpFun(
  val refl = @{thm refl-elem}
  val sym = @{thm sym-elem}
  val trans = @{thm trans-elem}
  val refl-red = @{thm refl-red}
  val trans-red = @{thm trans-red}
  val red-if-equal = @{thm red-if-equal}
  val default-rls = @{thms arithC-rls comp-rls}
  val routine-tac = routine-tac @{thms arith-typing-rls routine-rls}
)

```

```

fun arith-rew-tac ctxt prems =
  make-rew-tac ctxt (Arith-simp.norm-tac ctxt (@{thms congr-rls}, prems))

```

```

fun hyp-arith-rew-tac ctxt prems =
  make-rew-tac ctxt
    (Arith-simp.cond-norm-tac ctxt (prove-cond-tac ctxt, @{thms congr-rls},
prems))
›

```

method-setup arith-rew = ‹

```

  Attrib.thms >> (fn ths => fn ctxt => SIMPLE-METHOD (arith-rew-tac ctxt
ths))
›

```

method-setup hyp-arith-rew = ‹

```

  Attrib.thms >> (fn ths => fn ctxt => SIMPLE-METHOD (hyp-arith-rew-tac
ctxt ths))
›

```

3.4 Addition

Associative law for addition.

lemma *add-assoc*: $\llbracket a:N; b:N; c:N \rrbracket \Longrightarrow (a \# + b) \# + c = a \# + (b \# + c) : N$
by (*NE a*) *hyp-arith-rew*

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

lemma *add-commute*: $\llbracket a:N; b:N \rrbracket \Longrightarrow a \# + b = b \# + a : N$
apply (*NE a*)
apply *hyp-arith-rew*
apply (*rule sym-elem*)
prefer 2
apply (*NE b*)
prefer 4
apply (*NE b*)
apply *hyp-arith-rew*
done

3.5 Multiplication

Right annihilation in product.

lemma *mult-0-right*: $a:N \Longrightarrow a \# * 0 = 0 : N$
apply (*NE a*)
apply *hyp-arith-rew*
done

Right successor law for multiplication.

lemma *mult-succ-right*: $\llbracket a:N; b:N \rrbracket \Longrightarrow a \# * \text{succ}(b) = a \# + (a \# * b) : N$
apply (*NE a*)
apply (*hyp-arith-rew add-assoc [THEN sym-elem]*)
apply (*assumption | rule add-commute mult-typingL add-typingL intrL-rls refl-elem*)
done

Commutative law for multiplication.

lemma *mult-commute*: $\llbracket a:N; b:N \rrbracket \Longrightarrow a \# * b = b \# * a : N$
apply (*NE a*)
apply (*hyp-arith-rew mult-0-right mult-succ-right*)
done

Addition distributes over multiplication.

lemma *add-mult-distrib*: $\llbracket a:N; b:N; c:N \rrbracket \Longrightarrow (a \# + b) \# * c = (a \# * c) \# + (b \# * c) : N$
apply (*NE a*)
apply (*hyp-arith-rew add-assoc [THEN sym-elem]*)
done

Associative law for multiplication.


```

lemma mult-assoc:  $\llbracket a:N; b:N; c:N \rrbracket \implies (a \#* b) \#* c = a \#* (b \#* c) : N$ 
apply (NE a)
apply (hyp-arith-rew add-mult-distrib)
done

```

3.6 Difference

Difference on natural numbers, without negative numbers

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

```

lemma diff-self-eq-0:  $a:N \implies a - a = 0 : N$ 
apply (NE a)
apply hyp-arith-rew
done

```

```

lemma add-0-right:  $\llbracket c : N; 0 : N; c : N \rrbracket \implies c \#+ 0 = c : N$ 
by (rule addC0 [THEN [3] add-commute [THEN trans-elem]])

```

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*:

```

 $b:N \implies ?a : \prod x:N. Eq(N, b-x, 0) \longrightarrow Eq(N, b \#+ (x-b), x)$ 
apply (NE b)
  — strip one "universal quantifier" but not the "implication"
apply (rule-tac [3] intr-rls)
  — case analysis on  $x$  in  $\text{succ}(u) \leq x \longrightarrow \text{succ}(u) \#+ (x - \text{succ}(u)) = x$ 
prefer 4
apply (NE x)
apply assumption
  — Prepare for simplification of types – the antecedent  $\text{succ}(u) \leq x$ 
apply (rule-tac [2] replace-type)
apply (rule-tac [1] replace-type)
apply arith-rew
  — Solves first 0 goal, simplifies others. Two subgoals remain. Both follow by
  rewriting, (2) using quantified induction hyp.
apply intr — strips remaining  $\prod$ s
apply (hyp-arith-rew add-0-right)
apply assumption
done

```

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:  $\llbracket a:N; b:N; b - a = 0 : N \rrbracket \Longrightarrow b \# + (a - b) = a : N$ 
  apply (rule EqE)
  apply (rule add-diff-inverse-lemma [THEN ProdE, THEN ProdE])
  apply (assumption | rule EqI) +
  done

```

3.7 Absolute difference

Typing of absolute difference: short and long versions.

```

lemma absdiff-typing:  $\llbracket a:N; b:N \rrbracket \Longrightarrow a \mid - \mid b : N$ 
  unfolding arith-defs by typechk

```

```

lemma absdiff-typingL:  $\llbracket a = c:N; b = d:N \rrbracket \Longrightarrow a \mid - \mid b = c \mid - \mid d : N$ 
  unfolding arith-defs by equal

```

```

lemma absdiff-self-eq-0:  $a:N \Longrightarrow a \mid - \mid a = 0 : N$ 
  unfolding absdiff-def by (arith-rew diff-self-eq-0)

```

```

lemma absdiffC0:  $a:N \Longrightarrow 0 \mid - \mid a = a : N$ 
  unfolding absdiff-def by hyp-arith-rew

```

```

lemma absdiff-succ-succ:  $\llbracket a:N; b:N \rrbracket \Longrightarrow \text{succ}(a) \mid - \mid \text{succ}(b) = a \mid - \mid b : N$ 
  unfolding absdiff-def by hyp-arith-rew

```

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

```

lemma absdiff-commute:  $\llbracket a:N; b:N \rrbracket \Longrightarrow a \mid - \mid b = b \mid - \mid a : N$ 
  unfolding absdiff-def
  by (rule add-commute) (typechk diff-typing)

```

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

```

schematic-goal add-eq0-lemma:  $\llbracket a:N; b:N \rrbracket \Longrightarrow ?c : Eq(N, a \# + b, 0) \longrightarrow Eq(N, a, 0)$ 
  apply (NE a)
  apply (rule-tac [3] replace-type)
  apply arith-rew
  apply intr — strips remaining  $\prod$ s
  apply (rule-tac [2] zero-ne-succ [THEN FE])
  apply (erule-tac [3] EqE [THEN sym-elem])
  apply (typechk add-typing)
  done

```

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

```

lemma add-eq0:  $\llbracket a:N; b:N; a \# + b = 0 : N \rrbracket \Longrightarrow a = 0 : N$ 
  apply (rule EqE)
  apply (rule add-eq0-lemma [THEN ProdE])
  apply (rule-tac [3] EqI)
  apply typechk
  done

```

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 \mid - \mid b = 0 : N]] ==> ?a : Eq(N, a-b, 0) × Eq(N, b-a, 0)
unfolding absdiff-def
apply intr
apply eqintr
apply (rule-tac [2] add-eq0)
apply (rule add-eq0)
apply (rule-tac [6] add-commute [THEN trans-elem])
apply (typechk diff-typing)
done

```

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*: $[[a \mid - \mid b = 0 : N; a:N; b:N]] \implies a = b : N$

```

apply (rule EqE)
apply (rule absdiff-eq0-lem [THEN SumE])
apply eqintr
apply (rule add-diff-inverse [THEN sym-elem, THEN trans-elem])
apply (erule-tac [3] EqE)
apply (hyp-arith-rew add-0-right)
done

```

3.8 Remainder and Quotient

Typing of remainder: short and long versions.

lemma *mod-typing*: $[[a:N; b:N]] \implies a \text{ mod } b : N$
unfolding *mod-def* **by** (*typechk absdiff-typing*)

lemma *mod-typingL*: $[[a = c:N; b = d:N]] \implies a \text{ mod } b = c \text{ mod } d : N$
unfolding *mod-def* **by** (*equal absdiff-typingL*)

Computation for *mod*: 0 and successor cases.

lemma *modC0*: $b:N \implies 0 \text{ mod } b = 0 : N$
unfolding *mod-def* **by** (*rew absdiff-typing*)

lemma *modC-succ*: $[[a:N; b:N]] \implies$
 $\text{succ}(a) \text{ mod } b = \text{rec}(\text{succ}(a \text{ mod } b) \mid - \mid b, 0, \lambda x y. \text{succ}(a \text{ mod } b)) : N$
unfolding *mod-def* **by** (*rew absdiff-typing*)

Typing of quotient: short and long versions.

lemma *div-typing*: $[[a:N; b:N]] \implies a \text{ div } b : N$
unfolding *div-def* **by** (*typechk absdiff-typing mod-typing*)

lemma *div-typingL*: $[[a = c:N; b = d:N]] \implies a \text{ div } b = c \text{ div } d : N$
unfolding *div-def* **by** (*equal absdiff-typingL mod-typingL*)

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

Computation for quotient: 0 and successor cases.

lemma *divC0*: $b:N \implies 0 \text{ div } b = 0 : N$
unfolding *div-def* **by** (*rew mod-typing absdiff-typing*)

lemma *divC-succ*: $\llbracket a:N; b:N \rrbracket \implies$
 $\text{succ}(a) \text{ div } b = \text{rec}(\text{succ}(a) \text{ mod } b, \text{succ}(a \text{ div } b), \lambda x y. a \text{ div } b) : N$
unfolding *div-def* **by** (*rew mod-typing*)

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

lemma *divC-succ2*: $\llbracket a:N; b:N \rrbracket \implies$
 $\text{succ}(a) \text{ div } b = \text{rec}(\text{succ}(a \text{ mod } b) \text{ |-| } b, \text{succ}(a \text{ div } b), \lambda x y. a \text{ div } b) : N$
apply (*rule divC-succ [THEN trans-elem]*)
apply (*rew div-typing-rls modC-succ*)
apply (*NE succ (a mod b) |-| b*)
apply (*rew mod-typing div-typing absdiff-typing*)
done

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

lemma *iszero-decidable*: $a:N \implies \text{rec}(a, \text{inl}(eq), \lambda ka kb. \text{inr}(\langle ka, eq \rangle)) :$
 $Eq(N, a, 0) + (\sum x:N. Eq(N, a, \text{succ}(x)))$
apply (*NE a*)
apply (*rule-tac [3] PlusI-inr*)
apply (*rule-tac [2] PlusI-inl*)
apply *eqintr*
apply *equal*
done

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

lemma *mod-div-equality*: $\llbracket a:N; b:N \rrbracket \implies a \text{ mod } b \# + (a \text{ div } b) \# * b = a : N$
apply (*NE a*)
apply (*arith-rew div-typing-rls modC0 modC-succ divC0 divC-succ2*)
apply (*rule EqE*)
— case analysis on $\text{succ}(u \text{ mod } b) \text{ |-| } b$
apply (*rule-tac a1 = succ (u mod b) |-| b in iszero-decidable [THEN PlusE]*)
apply (*erule-tac [3] SumE*)
apply (*hyp-arith-rew div-typing-rls modC0 modC-succ divC0 divC-succ2*)
— Replace one occurrence of b by $\text{succ}(u \text{ mod } b)$. Clumsy!
apply (*rule add-typingL [THEN trans-elem]*)
apply (*erule EqE [THEN absdiff-eq0, THEN sym-elem]*)
apply (*rule-tac [3] refl-elem*)
apply (*hyp-arith-rew div-typing-rls*)
done

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
  by (rule form-rls)
```

```
schematic-goal ?A type
  apply (rule form-rls)
  back
  apply (rule form-rls)
  apply (rule form-rls)
  done
```

```
schematic-goal  $\prod z: ?A . N + ?B(z)$  type
  apply (rule form-rls)
  apply (rule form-rls)
  apply (rule form-rls)
  apply (rule form-rls)
  apply (rule form-rls)
  done
```

4.2 Multi-step proofs: Type inference

```
lemma  $\prod w: N . N + N$  type
  by form
```

```
schematic-goal  $\langle 0, succ(0) \rangle : ?A$ 
  apply intr done
```

```
schematic-goal  $\prod w: N . Eq(?A, w, w)$  type
  apply typechk done
```

```
schematic-goal  $\prod x: N . \prod y: N . Eq(?A, x, y)$  type
  apply typechk done
```

typechecking an application of fst

```
schematic-goal  $(\lambda u. split(u, \lambda v w. v)) \langle 0, succ(0) \rangle : ?A$ 
  apply typechk done
```

typechecking the predecessor function

```
schematic-goal  $\lambda n. rec(n, 0, \lambda x y. x) : ?A$ 
  apply typechk done
```

typechecking the addition function

```
schematic-goal  $\lambda n. \lambda m. \text{rec}(n, m, \lambda x y. \text{succ}(y)) : ?A$   
  apply typechk done
```

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

```
method-setup  $N =$   
   $\langle \text{Scan.succeed } (fn \text{ ctxt} \Rightarrow \text{SIMPLE-METHOD } (\text{TRYALL } (\text{resolve-tac } \text{ctxt } @\{\text{thms } NF\}))) \rangle$ 
```

```
schematic-goal  $\lambda w. \langle w, w \rangle : ?A$   
  apply typechk  
  apply  $N$   
  done
```

```
schematic-goal  $\lambda x. \lambda y. x : ?A$   
  apply typechk  
  apply  $N$   
  done
```

typechecking fst (as a function object)

```
schematic-goal  $\lambda i. \text{split}(i, \lambda j k. j) : ?A$   
  apply typechk  
  apply  $N$   
  done
```

end

5 Examples with elimination rules

```
theory Elimination  
imports ../CTT  
begin
```

This finds the functions fst and snd!

```
schematic-goal [folded basic-defs]:  $A \text{ type} \Longrightarrow ?a : (A \times A) \longrightarrow A$   
  apply pc  
  done
```

```
schematic-goal [folded basic-defs]:  $A \text{ type} \Longrightarrow ?a : (A \times A) \longrightarrow A$   
  apply pc  
  back  
  done
```

Double negation of the Excluded Middle

```
schematic-goal  $A \text{ type} \Longrightarrow ?a : ((A + (A \longrightarrow F)) \longrightarrow F) \longrightarrow F$   
  apply intr
```

```

apply (rule ProdE)
apply assumption
apply pc
done

```

Experiment: the proof above in Isar

```

lemma
  assumes A type shows  $(\lambda f. f \text{ ' } \text{inr}(\lambda y. f \text{ ' } \text{inl}(y))) : ((A + (A \longrightarrow F)) \longrightarrow F)$ 
   $\longrightarrow F$ 
proof intr
  fix f
  assume  $f: f : A + (A \longrightarrow F) \longrightarrow F$ 
  with assms have  $\text{inr}(\lambda y. f \text{ ' } \text{inl}(y)) : A + (A \longrightarrow F)$ 
  by pc
  then show  $f \text{ ' } \text{inr}(\lambda y. f \text{ ' } \text{inl}(y)) : F$ 
  by (rule ProdE [OF f])
qed (rule assms)+

```

```

schematic-goal  $\llbracket A \text{ type}; B \text{ type} \rrbracket \Longrightarrow ?a : (A \times B) \longrightarrow (B \times A)$ 
apply pc
done

```

Binary sums and products

```

schematic-goal  $\llbracket A \text{ type}; B \text{ type}; C \text{ type} \rrbracket \Longrightarrow ?a : (A + B \longrightarrow C) \longrightarrow (A \longrightarrow C)$ 
 $\times (B \longrightarrow C)$ 
apply pc
done

```

```

schematic-goal  $\llbracket A \text{ type}; B \text{ type}; C \text{ type} \rrbracket \Longrightarrow ?a : A \times (B + C) \longrightarrow (A \times B +$ 
 $A \times C)$ 
by pc

```

```

schematic-goal
  assumes A type
  and  $\bigwedge x. x:A \Longrightarrow B(x) \text{ type}$ 
  and  $\bigwedge x. x:A \Longrightarrow C(x) \text{ type}$ 
  shows  $?a : (\sum x:A. B(x) + C(x)) \longrightarrow (\sum x:A. B(x)) + (\sum x:A. C(x))$ 
apply (pc assms)
done

```

Construction of the currying functional

```

schematic-goal  $\llbracket A \text{ type}; B \text{ type}; C \text{ type} \rrbracket \Longrightarrow ?a : (A \times B \longrightarrow C) \longrightarrow (A \longrightarrow (B$ 
 $\longrightarrow C))$ 
apply pc
done

```

schematic-goal
 assumes A type
 and $\bigwedge x. x:A \implies B(x)$ type
 and $\bigwedge z. z: (\sum x:A. B(x)) \implies C(z)$ type
 shows $?a : \prod f: (\prod z: (\sum x:A. B(x)) . C(z)).$
 $(\prod x:A . \prod y:B(x) . C(\langle x,y \rangle))$
 apply (*pc assms*)
 done

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

schematic-goal $\llbracket A \text{ type}; B \text{ type}; C \text{ type} \rrbracket \implies ?a : (A \longrightarrow (B \longrightarrow C)) \longrightarrow (A \times B \longrightarrow C)$
 apply *pc*
 done

schematic-goal
 assumes A type
 and $\bigwedge x. x:A \implies B(x)$ type
 and $\bigwedge z. z: (\sum x:A . B(x)) \implies C(z)$ type
 shows $?a : (\prod x:A . \prod y:B(x) . C(\langle x,y \rangle))$
 $\longrightarrow (\prod z: (\sum x:A . B(x)) . C(z))$
 apply (*pc assms*)
 done

Function application

schematic-goal $\llbracket A \text{ type}; B \text{ type} \rrbracket \implies ?a : ((A \longrightarrow B) \times A) \longrightarrow B$
 apply *pc*
 done

Basic test of quantifier reasoning

schematic-goal
 assumes A type
 and B type
 and $\bigwedge x y. \llbracket x:A; y:B \rrbracket \implies C(x,y)$ type
 shows
 $?a : (\sum y:B . \prod x:A . C(x,y))$
 $\longrightarrow (\prod x:A . \sum y:B . C(x,y))$
 apply (*pc assms*)
 done

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 $?a : (\prod x:A . \prod y:B(x) . C(x,y))$
 $\longrightarrow (\prod f: (\prod x:A . B(x)) . \prod x:A . C(x, f'x))$

apply (*pc assms*)
done

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 $?a : (\prod x:A. C(\text{inl}(x))) \longrightarrow (\prod y:B. C(\text{inr}(y)))$
 $\longrightarrow (\prod z: A+B. C(z))$
apply (*pc assms*)
done

schematic-goal [*folded basic-defs*]:
 $\llbracket A \text{ type}; B \text{ type}; C \text{ type} \rrbracket \implies ?a : (A \longrightarrow B \times C) \longrightarrow (A \longrightarrow B) \times (A \longrightarrow C)$
apply *pc*
done

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 $?a : (\prod x:A. \sum y:B(x). C(x,y)) \longrightarrow (\sum f: (\prod x:A. B(x)). \prod x:A. C(x, f'x))$
apply (*intr assms*)
prefer 2 **apply** *add-mp*
prefer 2 **apply** *add-mp*
apply (*erule SumE-fst*)
apply (*rule replace-type*)
apply (*rule subst-eqtyparg*)
apply (*rule comp-rls*)
apply (*rule-tac* [4] *SumE-snd*)
apply (*typechk SumE-fst assms*)
done

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 (*intr assms*)
fix $f a$
assume $f : f : \prod x:A. \text{Sum}(B(x), C(x))$ **and** $a : A$
then have $fa : f'a : \text{Sum}(B(a), C(a))$
by (*rule ProdE*)

```

then show  $\text{fst}(f \text{ ' } a) : B(a)$ 
  by (rule SumE-fst)
have  $\text{snd}(f \text{ ' } a) : C(a, \text{fst}(f \text{ ' } a))$ 
  by (rule SumE-snd [OF fa]) (typechk SumE-fst assms <a : A>)
moreover have  $(\lambda x. \text{fst}(f \text{ ' } x)) \text{ ' } a = \text{fst}(f \text{ ' } a) : B(a)$ 
  by (rule ProdC [OF <a : A>]) (typechk SumE-fst f)
ultimately show  $\text{snd}(f \text{ ' } a) : C(a, (\lambda x. \text{fst}(f \text{ ' } x)) \text{ ' } a)$ 
  by (intro replace-type [OF subst-eqtyparg]) (typechk SumE-fst assms <a : A>)
qed

```

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  $?a : (\prod x:A. \sum y:B(x). C(x,y)) \longrightarrow (\sum f: (\prod x:A. B(x)). \prod x:A. C(x, f \text{ ' } x))$ 
  apply (intr assms)

  apply (rule ProdE [THEN SumE])
    apply assumption
    apply assumption
    apply assumption
  apply (rule replace-type)
  apply (rule subst-eqtyparg)
  apply (rule comp-rls)
  apply (erule-tac [4] ProdE [THEN SumE])
  apply (typechk assms)
  apply (rule replace-type)
  apply (rule subst-eqtyparg)
  apply (rule comp-rls)
  apply (typechk assms)
  apply assumption
done

```

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  $?a : (\sum z:A \times B. C(z)) \longrightarrow (\sum u:A. \sum v:B. C(\langle u,v \rangle))$ 
  apply (rule intr-rls)
  apply (tactic <biresolve-tac context safe-brls 2>)

  apply (rule-tac [2] a = y in ProdE)
  apply (typechk assms)
  apply (rule SumE, assumption)
  apply intr
  defer 1

```

```

    apply assumption+
    apply (typechk assms)
done

```

end

6 Equality reasoning by rewriting

```

theory Equality
  imports ../CTT
begin

```

```

lemma split-eq:  $p : \text{Sum}(A,B) \implies \text{split}(p,\text{pair}) = p : \text{Sum}(A,B)$ 
  apply (rule EqE)
  apply (rule elim-rls, assumption)
  apply rew
done

```

```

lemma when-eq:  $\llbracket A \text{ type}; B \text{ type}; p : A+B \rrbracket \implies \text{when}(p,\text{inl},\text{inr}) = p : A + B$ 
  apply (rule EqE)
  apply (rule elim-rls, assumption)
  apply rew
done

```

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

```

lemma p:N  $\implies \text{rec}(p,0, \lambda y z. \text{succ}(y)) = p : N$ 
  apply (rule EqE)
  apply (rule elim-rls, assumption)
  apply rew
done

```

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

```

lemma p:N  $\implies \text{rec}(p,0, \lambda y z. \text{succ}(z)) = p : N$ 
  apply (rule EqE)
  apply (rule elim-rls, assumption)
  apply hyp-rew
done

```

Associativity of addition

```

lemma  $\llbracket a:N; b:N; c:N \rrbracket$ 
 $\implies \text{rec}(\text{rec}(a, b, \lambda x y. \text{succ}(y)), c, \lambda x y. \text{succ}(y)) =$ 
 $\text{rec}(a, \text{rec}(b, c, \lambda x y. \text{succ}(y)), \lambda x y. \text{succ}(y)) : N$ 
  apply (NE a)
  apply hyp-rew
done

```

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

```

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

```

```

apply (rule EqE)
apply (rule elim-rls, assumption)
apply (tactic ‹DEPTH-SOLVE-1 (rew-tac context [])›)
done

```

```

lemma [[a : A; b : B]] => (λu. split(u, λv w.<w,v>)) ‹<a,b> = <b,a> : ∑ x:B.
A
by rew

```

a contrived, complicated simplication, requires sum-elimination also

```

lemma (λf. λx. f'(f'x)) ‹(λu. split(u, λv w.<w,v>)) =
  λx. x : ∏ x:(∑ y:N. N). (∑ y:N. N)
apply (rule reduction-rls)
apply (rule-tac [3] intrL-rls)
apply (rule-tac [4] EqE)
apply (erule-tac [4] SumE)

apply rew
done

```

end

7 Synthesis examples, using a crude form of narrowing

```

theory Synthesis
imports ../CTT
begin

```

discovery of predecessor function

```

schematic-goal ?a : ∑ pred:?A . Eq(N, pred'0, 0) × (∏ n:N. Eq(N, pred ‹
succ(n), n)›)
apply intr
apply eqintr
apply (rule-tac [3] reduction-rls)
apply (rule-tac [5] comp-rls)
apply rew
done

```

the function fst as an element of a function type

```

schematic-goal [folded basic-defs]:
A type => ?a: ∑ f:?B . ∏ i:A. ∏ j:A. Eq(A, f ‹<i,j>, i)
apply intr
apply eqintr
apply (rule-tac [2] reduction-rls)
apply (rule-tac [4] comp-rls)
apply typechk

```

now put in A everywhere

apply *assumption+*
done

An interesting use of the eliminator, when

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

apply *intr*
apply *eqintr*
apply (*rule comp-rls*)
apply *rew*
done

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

oops

A tricky combination of when and split

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

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

apply *intr*
apply *eqintr*
apply (*rule PlusC-inl [THEN trans-elem]*)
apply (*rule-tac [4] comp-rls*)
apply (*rule-tac [7] reduction-rls*)
apply (*rule-tac [10] comp-rls*)
apply *typechk*

done

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

oops

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

oops

Deriving the addition operator

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

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

apply *intr*
apply *eqintr*
apply (*rule comp-rls*)

```

apply rew
done

```

The addition function – using explicit lambdas

```

schematic-goal [folded arith-defs]:

```

$$\begin{aligned}
& ?c : \sum plus : ?A . \\
& \quad \prod x:N. Eq(N, plus'0'x, x) \\
& \quad \quad \times (\prod y:N. Eq(N, plus'succ(y)'x, succ(plus'y'x)))
\end{aligned}$$

```

apply intr
  apply eqintr
  apply (tactic resolve-tac context [TSimp.split-eqn] 3)
  apply (tactic SELECT-GOAL (rew-tac context []) 4)
  apply (tactic resolve-tac context [TSimp.split-eqn] 3)
  apply (tactic SELECT-GOAL (rew-tac context []) 4)
  apply (rule-tac [3] p = y in NC-succ)

```

```

  apply rew
done

```

```

end

```