The HOL-Z Reference Manual
Version 3.0

Achim D. Brucker Burkhart Wolff

February 7, 2007

Copyright (C) 2000–2006 Achim D. Brucker and Burkhart Wolff
Permission is granted to make and distribute verbatim copies of this manual provided the copyright notice and this permission notice are preserved on all copies.
Permission is granted to copy and distribute modified versions of this manual under the conditions for verbatim copying, provided that the entire resulting derived work is distributed under the terms of a permission notice identical to this one.
Permission is granted to copy and distribute translations of this manual into another language, under the above conditions for modified versions, except that this permission notice may be stated in a translation approved by the Free Software Foundation.

Important note:
This manual describes

Contents

1 Abstract **5**

2 Introduction **7**

3 Starting HOL-Z **9**

4 The Z Syntax of HOL-Z **11**

5 HOL-Z Commands in the ISAR Language **13**
- 5.1 HOL-Z Toplevel and Prooflevel Commands . . . 13
- 5.2 Proof Methods . . . 15
- 5.3 Modifiers (Attributes) . . . 15
- 5.4 Miscellaneous . . . 16

6 Isabelle Theories **17**
- 6.1 Core of the Mathematical Toolkit . . . 17
- 6.2 Integers in the Mathematical Toolkit . . . 50
- 6.3 Functions in the Mathematical Toolkit . . . 60
  - 6.3.1 Relation Apply . . . 60
  - 6.3.2 Relations . . . 62
- 6.4 Finite Sets for the Mathematical Toolkit . . . 76
  - 6.4.1 Finite Sets . . . 79
  - 6.4.2 Finite Functions . . . 83
  - 6.4.3 Finite Number Range . . . 86
- 6.5 Sequences in the Mathematical Toolkit . . . 89
- 6.6 Support for a rudimentary lifter . . . 104
- 6.7 A rudimentery Theory-Morphism from List to Sequences . . . 110
- 6.8 Bags from the Mathematical Toolkit . . . 133
  - 6.8.1 Basic Theorems . . . 135
  - 6.8.2 Bag Union . . . 137
- 6.9 Definitions of Syntax and Core-Combinators of Z . . . 140
  - 6.9.1 Initialize Z-Environment, provide Access . . . 140
  - 6.9.2 Syntax . . . 141

6.9.3 Semantic Representation 145
6.9.4 Syntax: The parse-translation setup 147
6.9.5 Derived Rules 150
6.9.6 Proof Support: Z-tactics 153
6.10 Method Package for Z Representation and Z Refinement 158
6.10.1 Configuring the Generic PO Manager 158
6.10.2 The Z Method Package 159
6.11 Main Theory for the HOL-Z Environment 168
6.11.1 Proof Support: The .holz-loader (generated by ZETA) . 168
6.11.2 Proof Support: Toplevel-Command **zlemma** 170
6.11.3 Baustelle: Code von DARMA zum integrieren 172

Chapter 1

Abstract

HOL-Z is a proof environment for Z built as plug-in of the generic theorem prover Isabelle/HOL (Version 2005). It allows for importing Z specifications written in LaTeX and type-checked by the Javadfgdfg-based ZeTa System. HOL-Z then allows for the formal analysis of such specifications, i.e. by proving the conjectures stated in the specfication, generating proof obligations concerning the consistency of state and operation schemas as well as their proof, and by generating proof obligations resulting from refinement statements for functional and data refinement.

Proof documents containing lemmas, analytical statements, proofs and global checks were written in an extension of the Isabelle/ISAR proof language and can be edited with the emacs-based ProofGeneral interface.

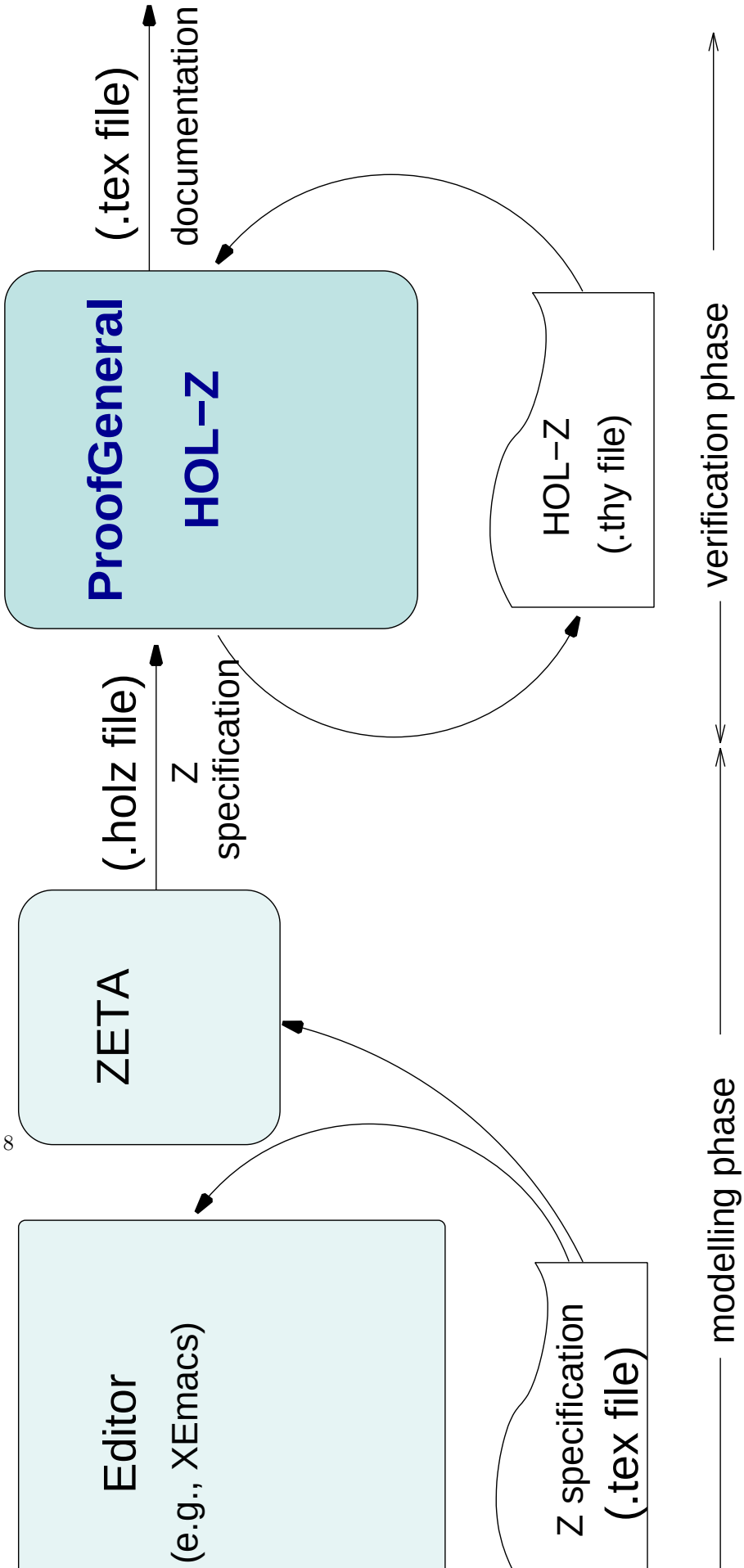
Chapter 2

Introduction

We present a new proof environment for the specification language Z. The basis is a semantic representation of Z in a structure-preserving, shallow em- bedding in Isabelle/HOL. On top of the embedding, new proof support for the Z schema calculus and for proof structuring are developed. Thus, we integrate Z into a well-known and trusted theorem prover with advanced deduction technology such as higher-order rewriting, tableaux-based provers and arithmetic decision procedures. A further achievement of this work is the integration of our embedding into a new tool- chain providing a Z-oriented type checker, documentation facilities and macro support for refinement proofs; as a result, the gap has been closed between a logical embedding proven correct and a tool suited for applications of non-trivial size.

As a proof environment allowing interactive proof (although comprising highly sophisticated automated proof procedures), HOL-Z assumes two different stakeholders during the development process: the *designer* and the *proof engineer*. Both will have different interests, may have different educational background and should have a different prespective to a given modeling problem. The particular toolchain, into which HOL-Z 3.0 is embedded, allows for the two parties to work together and make a question/answer game during the development phase. The designers question might be: ("does conjecture X hold"? "Is A a refinement of B"?, while answers of the proof engineer might be: "Yes, provided that you change definition Y as follows: ...". This toolchain is also reflected by a particular workflow depicted below:

The designers side (the *modeling phase*) is essentially an editor plus a lightweight type-checker ZeTa, which is just a JAVA app which releases the designer from the quite heavy burden to install full Isabelle. Via a plug-in into ZeTa, the designer can export his/her models to the proof engineers side (the *verification phase*) which consists of an enhanced editor (ProofGeneral based on Emacs) editing proof documents which were incrementally checked by Isabelle/HOL-Z.


Editor
(e.g., XEmacs)
ZETA
(.holz file)
Z
specification
ProofGeneral
HOL−Z
(.tex file)
documentation
Z specification
(.tex file)
HOL−Z
(.thy file)
modelling phase
verification phase

Chapter 3

Starting HOL-Z

Having installed Isabelle/HOL-Z correctly, the system can be started for the interactive mode by the standard launcher:

```
$ISABELLE/bin/Isabelle -L HOL-Z
```

where the shell variable `$ISABELLE` denotes the home of the Isabelle package in your file system. Do not forget the option `-L HOL-Z` which lets ProofGeneral, when started via the launcher, load a a keyword-file containing information on the HOL-Z specific keywords.

For the batch mode, Isabelle can be started via:

```
$ISABELLE/bin/isatool make
```

provided the usual Isamakefile has been set up. (See Isabelle system documentation).

MORE TO COME.

Chapter 4

The Z Syntax of HOL-Z

MORE TO COME.

Chapter 5

HOL-Z Commands in the ISAR Language

We assume that the reader is familiar with the Isabelle/Isar Reference Manual for Isabelle 2005. Given the fact that HOL-Z is just a plug-in into the Isabelle framework, its command-language is just a superset of the Isabelle/HOL ISAR command language.

The ISAR command language assumes a general theory file with a header containing thory dependencies, *toplevel commands* and *prooflevel commands* (which can only be entered after certain toplevel commands let the system enter the proof mode). Furthermore, there are *proof methods* which are arguments of the preconceived prooflevel command `apply`.

5.1 HOL-Z Toplevel and Prooflevel Commands

There are three general toplevel commands:

```
load_holz "<string>"

zlemma <name> : "<HOL-Z>"

po "<po-name>"
```

The first command just includes .holz files generated by ZeTa containing the Z model provided by the designer. Note that in current HOL-Z, the string must be the same as the name of the surrounding theory (otherwise problems with namespaces will arise). The second command is an equivalent to the standard `lemma`-command, but uses an own parser for the mixture of e-mail-format or latex-format Z expressions. The last comment takes the name of a generated obligation; in the sequel, the system is in proof mode (of this obligation).

Most commands are part of the package that implements support for the Z Refinement method.

There are a number of generic commands controlling the proof-obligation management:

```
list_po       [except <po-class-name>]
show_po       <po-name>+
check_po      [except <po-class-name>]
discharge_po  <po-name>
```

The first command causes a listing of the proof-obligations which are currently not resolved (not "discharged"). Certain pre-conceived classes of proof obligations (for the Z language, there are: "tcc", "axdefConservative", "ccState", "ccOp", "fwRefinementFunctional", "fwRefinementInit", "fwRefinementOp") can be excluded from listing or checking. The second allow for the display of a given proof obligation, the third succeeds only if none of the proof obligations remained undischarged (except those explicitly excluded from the check). The last command can only be executed in a proof-state when no logical subgoals remained unproven. It corresponds roughly to the standard command `done` but will only work for proofs started with the `po` command (see below).

Then there is a number of Z Method specific commands related to analysis by proof obligation management:

```
gen_state_cc <name>
gen_op_cc    <name>
gen_thm_cc   <name>

set_abs <name> ["[" functional "]" ]
refine_init <name> <name>
refine_op   <name> <name>
```

The first group of commands generates proof-obligations related to the consistency of Z condtructs: The first the existence of a state satisfying a state invariant captured by a state schema, the second captures the functional implementability of an operation schema, while the third captures the side-conditions related to the Z application operator, which requires that each function symbol is only applied to an argument within its domain (Not implemented at present).

The latter three commands are related o the refinement package. `set_abs` initializes the package with the abstraction predicate (which also comprises syntactic checks). If the option `[functional]` is set, the proof obligations for functional refinement were generated. `refine_init` generates the proof obligations with resp. to the initialization schemas, `refine_ops` the the obligations for operations (depending if the refinement is done in a relational or a functional setting).

5.2 Proof Methods

The following proof methods are available:

```
zstrip  ["(" <goal-nr> ")" ]
zunfold ["(" <goal-nr> ")" ] <thmname>+
zrule   ["(" <goal-nr> ")" ] <thmname>+
zfrule  ["(" <goal-nr> ")" ] <thmname>+
zdrule  ["(" <goal-nr> ")" ] <thmname>+
zerule  ["(" <goal-nr> ")" ] <thmname>+
```

`zstrip` erases the topmost schema-logical operations, in particular schema universal quantification, schema existential quantification, or schema implications. Standard HOL universal quantifications were also eliminted. The method does its best to rename variables such that the resulting proof state is still readable. The other methods are the Z-logical equivalents to the standard methods `unfold`,`rule`, `erule`,`frule`, `drule`, `erule`, but they perform an internal `zstrip` on the resolvend and a cleanup-phase for type-checking related constraints afterwards.

5.3 Modifiers (Attributes)

The following modifiers (called attributes in isabelle-terminology) are available:

```
<thm>  "["zstrip"]"
<thm>  "["zdecl "["<nr>"]""]"
<thm>  "["zpred "["<nr>"]""]"
```

They correspond to forward reasoning. `zstrip` erases the topmost schema-logical operations, in particular schema universal quantification, schema existential quantification, or schema implications. Standard HOL universal quantifications were also eliminted. The method does its best to rename variables such that the resulting proof state is still readable. (see also the proof method zstrip above which does the same in backward reasoning). `zdecl` constructs from a schema a projection lemma (in HOL-Z syntax) into the n‘th conjoint in the declaration part of schema; `zpred` constructs from a schema a projection lemma (in HOL-Z syntax) into the n‘th conjoint in the predicate part this schema.

5.4 Miscellaneous

There is a collection of definitions for Z, called `Z2HOL`, which allows for the conversion of a number of schema or binding related operators to standard HOL constructs. This rule set can be used by the simplifier or the standard unfold commands.

Chapter 6

Isabelle Theories

The main dependencies are shown in Fig. 6.1 on the following page. is build on top of Isabelle/HOL.[1]

6.1 Core of the Mathematical Toolkit

theory *ZMathTool* **imports** *Finite-Set Sum-Type Main* **begin**

types *('a,'b) <=> = ('a*'b) set* (**infixr** *20*)

consts

idZ ::*'a set => ('a <=> 'a)*
forw-comp ::*['a <=> 'b, 'b <=> 'c] => ('a <=> 'c)* (- %; - [*60*,*61*] *60*)
dom-restr ::*['a set , 'a <=> 'b] => ('a <=> 'b)* (- <: - [*71*,*70*] *70*)
ran-restr ::*['a <=> 'b, ('b)set] => 'a <=> 'b* (- :> - [*65*,*66*] *65*)
dom-substr ::*[('a)set , 'a <=> 'b] => 'a <=> 'b* (- <−: - [*71*,*70*] *70*)
ran-substr ::*['a <=> 'b, ('b) set] => ('a <=> 'b)* (- :−> - [*65*,*66*] *65*)

trans-clos ::*('a <=> 'a) => ('a <=> 'a)* (- %+ [*1000*] *999*)
ref-trans-clos::*('a <=> 'a) => ('a <=> 'a)* (- %* [*1000*] *999*)

partial-func ::*['a set,'b set] => ('a <=> 'b) set* (- −|−> - [*54*,*53*] *53*)

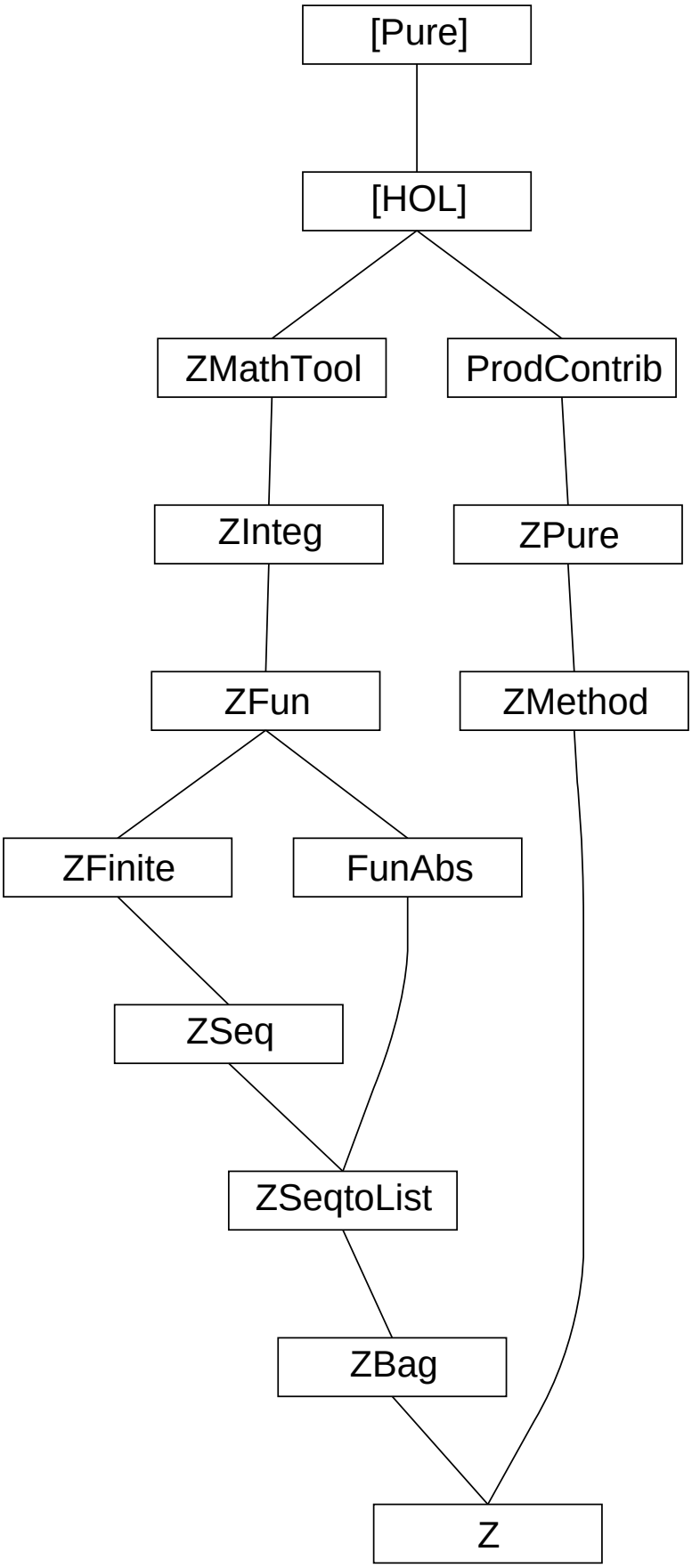
total-func ::*['a set,'b set] => ('a <=> 'b) set* (- −−−> - [*54*,*53*] *53*)

partial-inj ::*['a set,'b set] => ('a <=> 'b) set* (- >−|−> - [*54*,*53*] *53*)

total-inj ::*['a set,'b set] => ('a <=> 'b) set* (- >−−> - [*54*,*53*] *53*)

partial-surj ::*['a set,'b set] => ('a <=> 'b) set* (- −|−>> - [*54*,*53*] *53*)

[1] More precise, is build on top of Isabelle/HOL-Complex but only to support real numbers. If you dont't need them, should be build just fine on top of a plain Isabelle/HOL.


[Pure]
[HOL]
ZMathTool
ProdContrib
ZInteg
ZPure
ZFun
ZMethod
ZFinite
FunAbs
ZSeq
ZSeqtoList
ZBag
Z

total-surj ::['a set,'b set] => ('a <=> 'b) set (- -->> - [54,53] 53)

bijject ::['a set,'b set] => ('a <=> 'b) set (- >-->> - [54,53] 53)

fin-part-func::['a set,'b set] => ('a <=> 'b) set (- -||-> - [54,53] 53)

fin-part-inj ::['a set,'b set] => ('a <=> 'b) set (- >-||-> - [54,53] 53)

rel ::['a set, 'b set] => ('a <=> 'b) set (- <--> - [54,53] 53)
rel-appl ::['a<=>'b,'a] => 'b (- %^ - [90,91] 90)
override ::['a <=> 'b, 'a <=> 'b] => ('a <=> 'b) (- '(+') - [55,56] 55)
iter ::['a <=> 'a, int] => ('a <=> 'a)
numb-range ::[int,int] => (int) set (- .. - [50,51] 50)

syntax (*xsymbols*)
idZ ::'a set => ('a <=> 'a) (id)
forw-comp ::['a <=> 'b, 'b <=> 'c] => ('a <=> 'c) (- ⨾ - [60,61] 60)
dom-restr ::['a set , 'a <=> 'b] => ('a <=> 'b) (- ◁ - [71,70] 70)
ran-restr ::['a <=> 'b, ('b)set] => 'a <=> 'b (- ▷ - [65,66] 65)
dom-substr ::[('a)set , 'a <=> 'b] => 'a <=> 'b (- ⩤ - [71,70] 70)
ran-substr ::['a <=> 'b, ('b) set] => ('a <=> 'b) (- ⩥ - [65,66] 65)

trans-clos ::('a <=> 'a) => ('a <=> 'a) (- ⁺ [1000] 999)
ref-trans-clos::('a <=> 'a) => ('a <=> 'a) (- ⁺ [1000] 999)

partial-func ::['a set,'b set] => ('a <=> 'b) set (- ⇸ - [54,53] 53)
total-func ::['a set,'b set] => ('a <=> 'b) set (- → - [54,53] 53)
partial-inj ::['a set,'b set] => ('a <=> 'b) set (- ⤔ - [54,53] 53)
total-inj ::['a set,'b set] => ('a <=> 'b) set (- ↣ - [54,53] 53)
partial-surj ::['a set,'b set] => ('a <=> 'b) set (- ⤀ - [54,53] 53)
total-surj ::['a set,'b set] => ('a <=> 'b) set (- ↠ - [54,53] 53)
bijject ::['a set,'b set] => ('a <=> 'b) set (- ⤖ - [54,53] 53)
override::['a <=> 'b, 'a <=> 'b] => ('a <=> 'b) (- ⊕ - [55,56] 55)
numb-range ::[int,int] => (int) set (- .. - [50,51] 50)
fin-part-func::['a set,'b set] => ('a <=> 'b) set (- ⇸ - [54,53] 53)
fin-part-inj ::['a set,'b set] => ('a <=> 'b) set (- ⤔ - [54,53] 53)
rel ::['a set, 'b set] => ('a <=> 'b) set (- ↔ - [54,53] 53)
rel-appl ::['a<=>'b,'a] => 'b (-(.-.) [90,91] 90)

syntax
dom ::('a <=> 'b) => 'a set

```
ran        ::('a <=> 'b) => 'b set
zinverse   ::('a <=> 'b) => ('b <=> 'a)          (- %~    [1000]999)
rel-image  ::('a <=> 'b) => ('a set) => 'b set   (-'(|-|')[1000,0]999)
back-comp  ::['b <=> 'c, 'a <=> 'b] => ('a <=> 'c)  (- %o -  [60,61] 60)
prodZ      ::['a set,'b set] => ('a <=> 'b)      (- %x -  [81,80] 80)
sumZ       ::['a set,'b set] => ('a + 'b) set    (- %+ -  [66,65] 65)
gen-un     ::'a set set => 'a set
gen-int    ::'a set set => 'a set
zint       :: int => nat                         ($i)
```

syntax (*xsymbols*)

```
dom        ::('a <=> 'b) => 'a set               (dom)
ran        ::('a <=> 'b) => 'b set               (ran)
zinverse   ::('a <=> 'b) => ('b <=> 'a)          (-~    [1000]999)
rel-image  ::('a <=> 'b) => ('a set) => 'b set   (-(|-|)[1000,0]999)
back-comp  ::['b <=> 'c, 'a <=> 'b] => ('a <=> 'c)  (- ∘ -  [60,61] 60)
prodZ      ::['a set,'b set] => ('a <=> 'b)      (- × - [81,80] 80)
sumZ       ::['a set,'b set] => ('a + 'b) set    (- + -  [66,65] 65)
gen-un     ::'a set set => 'a set                (⋃ -)
gen-int    ::'a set set => 'a set                (⋂ -)
```

translations

```
dom r          == Domain r
ran r          == Range r
r %~           == converse r
rel-image r s  == r``s
r %o s         == r O s
a %x b         == a <*> b
a %+ b         == a Plus b
gen-un s       == Union s
gen-int s      == Inter s
zint           == nat
```

consts

```
zsize      ::('a set) => int                     (# )
String     ::string set
Integers   ::int set                             (%Z)
Naturals   ::int set                             (%N)
Naturals-1 ::int set
<->        ::[bool, bool] => bool                (infixr 25)
maplet     ::['a, 'b] => ('a * 'b)               ((2- |-->/ -) [60,61] 60)
disjoint   ::('a <=> ('b set)) => bool
```

```
  partition    ::[('a <=> ('b set)), ('b set)] => bool (infixl 60)


syntax(xsymbols)
  zsize        ::('a set) => int                    (# )
  String       ::string set                         (String)
  Integers     ::int set                            (ℤ)
  Naturals     ::int set                            (ℕ)
  Naturals-1   ::int set                            (ℕ₁)
  maplet       ::['a, 'b] => ('a * 'b)              ((2- ↦/ -) [60,61] 60)
  disjoint     ::('a <=> ('b set)) => bool          (disjoint)
  partition    ::[('a <=> ('b set)), ('b set)] => bool (- partition - [60,61]60)


consts Fin :: 'a set => 'a set set                  (%F)
syntax (xsymbols)
  Fin :: 'a set => 'a set set                       (𝔽)

inductive Fin(A)
  intros
    emptyI : {} : Fin(A)
    insertI: [| a: A;   b: Fin(A) |] ==> insert a b : Fin(A)


defs

idZ-def:          idZ X       == {p. ? x:X . p = (x,x)}
rel-def:          A <--> B    == Pow (A %x B)
pfun-def:   S -|-> R    == {f. f:(S <--> R)
                              & (! x y1 y2. (x,y1):f & (x,y2):f  --> (y1=y2))}

forw-comp-def:     r %;  s      == {(x,z). ? y. (x,y):r & (y,z):s }
dom-restr-def:     s <: r       == {(x,y). (x,y) : r & x : s}
ran-restr-def:     r :> s       == {(x,y). (x,y) : r & y : s}
dom-substr-def:    s <-: r      == {(x,y). (x,y) : r & x ~: s}
ran-substr-def:    r :-> s      == {(x,y). (x,y) : r & y ~: s}


defs

ref-trans-clos-def: r%*         == lfp (% s. idZ(dom r Un ran r) Un (r %o s))
trans-clos-def:     r%+         == r %o r%*
```

```
tfun-def:        S ---> R     == {s. s : S -|-> R & dom s = S}
partial-inj-def:  S >-|-> R   == {s. s : S -|-> R &
                                 (ALL x1 x2 y. (x1,y) : s & (x2,y) : s
                                  --> x1 = x2)}
total-inj-def:    S >--> R    == (S >-|-> R) Int (S ---> R)
partial-surj-def: S -|->> R   == {s. s : S -|-> R & ran s = R}
total-surj-def:   S -->> R    == (S -|->> R) Int (S ---> R)
biject-def:       S >-->> R   == (S -->> R) Int (S >--> R)
override-def:     S (+) R     == (dom R <-: S) Un R

String-def:       String      == UNIV
Integers-def:     Integers    == {a::int. True}
Naturals-def:     Naturals    == {a::int. 0 <= a }
Naturals-1-def:   Naturals-1  == {a::int. 1 <= a }


iff-def:          A <-> B     == A = B

zsize-def:        # S         == int (card S)

iter-def:         iter R n    == if (n:Naturals)
                                 then (nat-rec (idZ(dom R))
                                          (%n it. R %o it)
                                          ($i n))
                                 else (nat-rec (idZ(dom (R %~)))
                                          (%n it. (R %~) %o it))
                                          ($i ( n))

numb-range-def:     (a .. b)    == { k. a <= k & k<= b}
fin-part-func-def:  S -||-> R   == {s. s : S -|-> R & dom(s) : Fin(S)}
fin-part-inj-def:   S >-||-> R  == (S >-|-> R) Int (S -||-> R)
rel-apply-def:      R %^ x      == ( • y. (x,y) : R)
maplet-def:         maplet a b  == (a,b)
disjoint-def:       disjoint f  == (! p:f. ! q:f. p ~= q -->
                                          (snd p) Int (snd q) = {})
partition-def:      f partition a == (disjoint f & gen-un(ran f) = a)
```

constdefs

Lambda :: ['a set, 'a => 'b] => 'a <=> 'b
Lambda A f == {(x,y). x : A & y = f x}
Mu :: ['a set, 'a => bool] => 'a
Mu A P == • x. x : A & P x

syntax
*Lambda :: [pttrn,'a set,'a => 'b] => 'a<=>'b ((3lambda (-):(-)./ -) [0,0,10]10)
*Mu :: [pttrn,'a set,'a => bool] => 'a ((3mu (-):(-)./ -) [0,0,10]10)

translations
lambda x:A. E == *Lambda A* (% *x. E*)
mu x:A. P == *Mu A* (% *x. P*)

syntax
- *powset* :: ('a set) => ('a set)set (%P)
- *emptyset* :: ('a set) => ('a set) (emptyset- 10)

syntax (*xsymbols*)
- *powset* :: ('a set) => ('a set)set (ℙ)
- *emptyset* :: ('a set) => ('a set) (∅- 10)

translations
%*P A* == *Pow A*
emptyset a => {}

Lambda Lemmas

lemma *Lambda-total*: *Lambda A f* : (*A* ---> (*range f*))
apply (*unfold Lambda-def tfun-def pfun-def rel-def single-valued-def*)
apply *auto*
done

lemma *Lambda-dom*: *dom*(*Lambda A f*) = *A*
apply (*unfold Lambda-def tfun-def pfun-def rel-def*)
apply *auto*

done

lemma *Lambda-ran*: *ran*(*Lambda A f*) <= *range f*
apply (*unfold Lambda-def*)
apply *auto*
done

lemma *Lambda-beta*: !! *A*. *a* : *A* ==> ((*Lambda A f*) %^ *a*) = *f a*
apply (*unfold Lambda-def rel-apply-def*)
apply *auto*
done

lemma *Mu-equality*:
[|*a* : *A* ; *P a*; ! *x*:*A*. *P x* --> *x* = *a*|] ==> (*Mu A P*) = *a*
apply (*unfold Mu-def*)
apply *blast*
done

declare *Lambda-total*[*simp*] *Lambda-beta*[*simp*] *Lambda-ran*[*simp*]
Mu-equality[*simp*] *Lambda-dom*[*simp*]

Set Lemmas

lemma *Int-empty*[*simp*]: *A Int B* = {} ==> *A* − *B* = *A*
apply *auto*
done

lemma *subset-union-mono*[*simp*]: *A* <= *B* ==> *A* <= *B Un C*
apply *fast*
done

lemma *union-subset-distr*[*simp*]: (*A Un B* <= *C*) = (*B* <= *C* & *A* <= *C*)
apply *fast*
done

Simpler Definitions

lemma *idZ-simp*: *idZ X* = {(*x*,*x*) | *x*. *x*:*X*}
apply (*unfold idZ-def*)
apply *auto*
done

lemma *inverse-simp*: *R* %~ = {(*y*,*x*) | *x y*. (*x*,*y*):*R*}

apply *auto*
done

lemma *dom-simp*: *dom R* = {*x*. ? *y*. (*x*,*y*):*R*}
apply *auto*
done

lemma *ran-simp*: *ran R* = {*y*. ? *x*. (*x*,*y*):*R*}
apply *auto*
done

lemma *idZ-eqI*: !! *A*.[| *a*=*b*; *a*:*A* |] ==> (*a*,*b*) : *idZ*(*A*)
apply (*unfold idZ-def*)
apply *blast*
done

lemmas *idZI* = *refl* [*THEN idZ-eqI*]

lemma *idZE*:
assumes *p1*: *c* : *idZ*(*A*)
assumes *p2*: !!*x y*. ⟦ *x*:*A*; *c* = (*x*,*x*)⟧ ⟹ *P*
shows *P*
apply (*cut-tac p1*)
apply (*cut-tac CollectD*)
apply (*unfold idZ-def*)
apply (*erule bexE*)
apply (*erule p2*)
apply *assumption*
apply *auto*
done

declare *idZI* [*intro*!]
declare *idZE* [*elim*!]

lemma *idZ-iff*: ((*x*,*y*) : *idZ A*) = (*x*=*y* & *x* : *A*)
apply (*unfold idZ-def*)
apply *blast*
done

lemma *idZ-subset-Times*: *idZ*(*A*) <= *A* <*> *A*

apply (*unfold idZ-def*)
apply *blast*
done

Domain and Range Properties

lemma *Dom-In*: *x*:*dom R* = (*? y*. *y*:*ran*(*R*) & (*x*,*y*):*R*)
apply *auto*
done

lemma *Ran-In*: *y*:*ran R* = (*? x*. *x*:*dom R* & (*x*,*y*):*R*)
apply *auto*
done

lemma *Dom-Union*[*simp*]: *dom*(*Q Un R*) = *dom*(*Q*) *Un dom*(*R*)
apply *auto*
done

lemma *Ran-Union*[*simp*]: *ran*(*Q Un R*) = *ran*(*Q*) *Un ran*(*R*)
apply *auto*
done

lemma *Dom-Inter*[*simp*]: *dom*(*Q Int R*) <= *dom*(*Q*) *Int dom*(*R*)
apply *auto*
done

lemma *Ran-Inter*[*simp*]: *ran*(*Q Int R*) <= *ran*(*Q*) *Int ran*(*R*)
apply *auto*
done

lemma *Dom-Empty*[*simp*]: *dom* {} = {}
apply *auto*
done

lemma *Ran-Empty* [*simp*]: *ran* {} = {}
apply *auto*
done

lemma *Dom-Insert0*: (*dom* (*insert* (*x*,*y*) *S*)) = *insert x* (*dom S*)
apply *auto*
done

lemma *Dom-Insert* [*simp*]: *dom* (*insert p A*) = (*insert* (*fst p*) (*dom A*))
apply (*subst surjective-pairing*)

apply (*simp only*: *Dom-Insert0*)
done

lemma *Ran-Insert0*: (*ran* (*insert* (*x*,*y*) *S*)) = *insert y* (*ran S*)
apply *auto*
done

lemma *Ran-Insert*[*simp*]: (*ran* (*insert p S*)) = *insert* (*snd p*) (*ran S*)
apply (*subst surjective-pairing*)
apply (*simp only*: *Ran-Insert0*)
done

lemma *pair-rel-dom* [*simp*]: (*a*,*b*):*r* ==> (*a* : (*dom r*))
apply *auto*
done

lemma *pair-rel-dom-fst* [*simp*]: *a*:*r* ==> ((*fst a*) : (*dom r*))
apply (*rule pair-rel-dom*)
apply (*subst surjective-pairing* [*symmetric*])
apply *assumption*
done

lemma *pair-rel-ran* [*simp*]: (*a*,*b*):*r* ==> *b* : *ran r*
apply *auto*
done

lemma *pair-rel-dom-snd* [*simp*]: *a*:*r* ==> *snd a* : *ran r*
apply (*rule pair-rel-ran*)
apply (*subst surjective-pairing* [*symmetric*])
apply *assumption*
done

lemma *dom-implies-ran* : *a* : *dom r* ==> (*EX b*. (*a*,*b*) : *r*)
apply (*simp add*: *dom-simp*)
done

lemma *Dom-Rel-Empty* [*simp*]:(*dom f* = {}) = (*f* = {})
apply *auto*

done

lemma *Ran-Rel-Empty* [*simp*]: (*ran f* = {}) = (*f* = {})
apply *auto*
done

Domain and Range Restriction

lemma *dom-restrI* [*rule-format*]: *p* : *r* −−> *fst p* : *s* −−> *p* : (*s* <: *r*)
apply (*unfold dom-restr-def*)
apply *auto*
done
declare *dom-restrI* [*intro*!]

lemma *dom-restrD1*:*p* : (*s* <: *r*) ==> *p* : *r*
apply (*unfold dom-restr-def*)
apply *auto*
done

lemma *dom-restrD2*: *p* : (*s* <: *r*) ==> *fst p* :*s*
apply (*unfold dom-restr-def*)
apply *auto*
done

lemma *dom-restrE2*:
assumes *p1*: *p* : (*s* <: *r*)
assumes *p2*: [| *p* : *r*; *fst p* :*s* |] ==> *P*
shows *P*
apply (*rule p2*)
apply (*rule dom-restrD1*)
apply (*rule p1*)
apply (*rule dom-restrD2*)
apply (*rule p1*)
done

lemma *dom-restrE* :
assumes *p1*: *p* : (*s* <: *r*)
assumes *p2*: !! *a b*.[| *p* = (*a*, *b*); (*a*,*b*) : *r*; *a*:*s* |] ==> *P*
shows *P*
apply (*rule surjective-pairing* [*THEN p2*])
apply (*rule surjective-pairing* [*THEN subst*])
apply (*rule dom-restrD1*)
apply (*rule p1*)

apply (*rule dom-restrD2*)
apply (*rule p1*)
done
declare *dom-restrE* [*elim*!]

lemma *ran-restrI* [*rule-format*]: *p* : *r* −−> *snd p* : *s* −−> *p* : (*r* :> *s*)
apply (*unfold ran-restr-def*)
apply *auto*
done
declare *ran-restrI* [*intro*!]

lemma *ran-restrD1*:*p* : (*r* :> *s*) ==> *p* : *r*
apply (*unfold ran-restr-def*)
apply *auto*
done

lemma *ran-restrD2*:!!*r*. *p* : (*r* :> *s*) ==> *snd p* :*s*
apply (*unfold ran-restr-def*)
apply *auto*
done

lemma *ran-restrE2*:
assumes *p1*: *p* : (*r* :> *s*)
assumes *p2*: [| *p* : *r*; *snd p* :*s* |] ==> *P*
shows *P*
apply (*rule p2*)
apply (*rule ran-restrD1*)
apply (*rule p1*)
apply (*rule ran-restrD2*)
apply (*rule p1*)
done

lemma *ran-restrE*:
assumes *p1*: *p* : (*r* :> *s*)
assumes *p2*: !! *a b*.[| *p* = (*a*, *b*); (*a*,*b*) : *r*; *b*:*s* |] ==> *P*
shows *P*
apply (*rule surjective-pairing* [*THEN p2*])
apply (*rule surjective-pairing* [*THEN subst*])
apply (*rule ran-restrD1*)
apply (*rule p1*)
apply (*rule ran-restrD2*)
apply (*rule p1*)

done
declare *ran-restrE* [*elim*!]

lemma *Dom-Restrict* [*simp*]: *dom* (S<:R) = S *Int* (*dom* R)
apply *auto*
done

lemma *Ran-Restrict* [*simp*]: *ran* (S:>R) = *ran*(S) *Int* R
apply *auto*
done

lemma *Dom-Rest-Subset* [*simp*]: (S<:R) <= R
apply *auto*
done

lemma *Ran-Rest-Subset* [*simp*]: (S:>R) <= S
apply *auto*
done

lemma *Dom-Restr-mt*[*simp*]: X <: {} = {}
by(*simp add*:*dom-restr-def*)

lemma *Dom-Ran-Restr-Comp* : ((S <: R) :> T) = (S<: (R :> T))
apply *auto*
done

lemma *Dom-Dom-Restr-Comp* [*simp*]: (S <: (R <: T)) = ((S *Int* R) <: T)
apply *auto*
done

lemma *Ran-Ran-Restr-Comp* [*simp*]: ((S :> R) :> T) = (S :> (R *Int* T))
apply *auto*
done

lemma *No-Dom-Restr* [*simp*]: (*dom* R <= S) = (S <: R = R)

apply *auto*
done

lemma *Empty-Dom-Restr* [*simp*]: {} <: R = {}
apply *auto*

done

lemma *No-Ran-Restr* [*simp*]: (*ran R* <= *S*) = (*R* :> *S* = *R*)

apply *auto*
done

lemma *Empty-Ran-Restr* [*simp*]: *R* :> {} = {}
apply *auto*
done

lemma *UNIV-Ran-Restr* [*simp*]: *x* :> *UNIV* = *x*
apply *auto*
done

lemma *dom-restr-single* [*simp*]: !!*R*. *fst p* : *R* ==> *p* : *R* <: *f* = (*p*:*f*)
apply *auto*
done

Domain and Range Anti-Restriction

lemma *dom-substrI* [*rule-format* (*no-asm*)]: *p* : *r* --> *fst p* ~: *s* --> *p* : (*s* <−: *r*)
apply (*unfold dom-substr-def*)
apply *auto*
done
declare *dom-substrI* [*intro*!]

lemma *dom-substrD1*: !!*r*. *p* : (*s* <−: *r*) ==> *p* : *r*
apply (*unfold dom-substr-def*)
apply *auto*
done

lemma *dom-substrD2*:
!!*r*. *p* : (*s* <−: *r*) ==> *fst p* ~:*s*
apply (*unfold dom-substr-def*)
apply *auto*
done

lemma *dom-substrE2*:

assumes *p1*: p : (s <−: r)
assumes *p2*: [| p : r; *fst* p ~:s |] ==> P
shows P
apply (*rule p2*)
apply (*rule dom-substrD1*)
apply (*rule p1*)
apply (*rule dom-substrD2*)
apply (*rule p1*)
done

lemma *dom-substrE*:
assumes *p1*: p : (s<−:r)
assumes *p2*: !! a b.[| p = (a, b); (a,b) : r; a ~:s |] ==> P
shows P
apply (*rule surjective-pairing* [*THEN p2*])
apply (*rule surjective-pairing* [*THEN subst*])
apply (*rule dom-substrD1*)
apply (*rule p1*)
apply (*rule dom-substrD2*)
apply (*rule p1*)
done
declare *dom-substrE* [*elim*!]

lemma *ran-substrI* [*rule-format* (*no-asm*)]: p : r −−> *snd* p ~: s −−> p : (r :−> s)
apply (*unfold ran-substr-def*)
apply *auto*
done
declare *ran-substrI* [*intro*!]

lemma *ran-substrD1*: !!r. p : (r :−> s) ==> p : r
apply (*unfold ran-substr-def*)
apply *auto*
done

lemma *ran-substrD2*:!!r. p : (r :−> s) ==> *snd* p ~:s
apply (*unfold ran-substr-def*)
apply *auto*
done

lemma *ran-substrE2*:

assumes *p1*: $p : (r :-> s)$
assumes *p2*: $[|\ p : r;\ \ snd\ p\ ^{\sim}{:}s\ |] ==> P$
shows P
apply (*rule p2*)
apply (*rule ran-substrD1*)
apply (*rule p1*)
apply (*rule ran-substrD2*)
apply (*rule p1*)
done

lemma *ran-substrE*:
assumes *p1*: $p : (r :-> s)$
assumes *p2*: $!!\ a\ b.[|\ p = (a,\ b);\ \ (a,b) : r;\ \ b\ ^{\sim}{:}s\ |] ==> P$
shows P
apply (*rule surjective-pairing* [*THEN p2*])
apply (*rule surjective-pairing* [*THEN subst*])
apply (*rule ran-substrD1*)
apply (*rule p1*)
apply (*rule ran-substrD2*)
apply (*rule p1*)
done
declare *ran-substrE* [*elim*!]

lemma *Dom-Anti-Restr* : $(S <-: R) = (dom(R) - S) <: R$
apply *auto*
done

lemma *dom-substr-Un-distribL* [*simp*]: $(S <-: (A\ Un\ B)) = ((S <-: A)\ Un\ (S <-: B))$
apply *auto*
done

lemma *dom-substr-Un-asso* [*simp*]: $(A <-: (B <-: C)) = ((A\ Un\ B) <-: C)$
apply *auto*
done

lemma *Ran-Anti-Restr* [*simp*]: $(R :-> T) = R :> (ran(R) - T)$
apply *auto*
done

lemma *DomRestr-Un-DomSubstr* [*simp*]:$(S <: R)\ Un\ (S <-: R) = R$
apply *auto*

done

lemma *RanRestr-Un-RanSubstr* [*simp*]: (*S* :> *R*) *Un* (*S* :–> *R*) = *S*
apply *auto*
done

lemma *dom-anti-restr-single* [*simp*]: !! *R*. *fst p* ~: *R* ==> *p* : *R* <–: *f* = (*p* : *f*)
apply *auto*
done

lemma *dom-substr-empty* [*simp*]: (*dom r* <–: *r*) = {}
apply *auto*
done

lemma *equals0E*:
assumes *p1*: *Collect P* = {}
assumes *p2*: ~ *P x* ==> *Q*
shows *Q*
apply (*rule p2*)
apply (*rule notI*)
apply (*rule-tac Q* = *Collect P* = {} **in** *impE*)
prefer *2*
apply (*assumption*)
apply (*rule impI*)
apply (*rule p1*)
apply *auto*
done

lemma *dom-substr-subset* : ((*m* <–: *r*) = {}) = ((*dom r*) <= *m*)
apply *auto*
done

lemma *dom-substr-subset-imp* [*simp*]: !! *r*. ((*dom r*) <= *m*) ==> ((*m* <–: *r*) = {})
apply *fast*
done

Relational Distributivity

lemma *Rel-Union-Distr* [*simp*]: (*r* : (*X* <–> *Y*) & *s* : (*X* <–> *Y*)) = (*r Un s* : (*X* <–> *Y*))
apply (*unfold rel-def Pow-def*)

apply *fast*
done

Relational Composition

lemma *forw-comp-simp* [*simp*]: $(r\ \%;\ \ s) = (s\ \%o\ r)$
apply (*unfold forw-comp-def*)
apply *auto*
done

lemma *Compose-Rel* [*simp*]: $(x,z):(R\ \%;\ \ Q) = (?\ y.\ (x,y):R\ \&\ (y,z):Q)$
apply *auto*
done

lemma *Compose-Transitive* : $(P\ \%;\ \ (Q\ \%;\ \ R)) = ((P\ \%;\ \ Q)\ \%;\ \ R)$
apply *auto*
done

lemma *Compose-Id-Left* : $((idZ\ X)\ \%;\ \ \ P) = (X <: P)$
apply (*unfold idZ-def*)
apply *auto*
done

lemma *Compose-Id-Right* [*simp*]: $(P\ \%;\ \ idZ\ X) = (P :> X)$
apply (*unfold idZ-def*)
apply *auto*
done

lemma *Compose-Refl-Trans* [*simp*]: $!!\ R.\ (R\ \%;\ \ R) <= R ==> ((R\ Un\ idZ(S))$
$\%;\ \ (R\ Un\ idZ(S))) <= (R\ Un\ idZ(S))$
apply *auto*
done

Relational Inversion

lemma *Inversion* [*simp*]: $(y,x):(R\ \%\sim) = ((x,y):R)$
apply *auto*
done

lemma *Inverse-Comp* [*simp*]: $((Q\ \%;\ \ R)\ \%\sim) = ((R\ \%\sim)\ \%;\ \ (Q\ \%\sim))$
apply *auto*

done

lemma *Inverse-Idem* [*simp*]: ((*R* %~) %~) = *R*
apply *auto*
done

lemma *Inverse-Id* [*simp*]: ((*idZ X*) %~) = *idZ X*
apply (*unfold idZ-def*)
apply *auto*
done

lemma *Inverse-Dom* [*simp*]: *dom*(*R* %~) = *ran* (*R*)
apply *auto*
done

lemma *Inverse-Ran* [*simp*]: *ran*(*R* %~) = *dom* (*R*)
apply *auto*
done

lemma *Dom-Anti-Restr-Int* [*simp*]: (((*dom g*) <−: *f*) *Int g*) = {}
apply *auto*
done

lemma *Dom-Dom-Anti-Restr-Int* [*simp*]: ((*dom* ((*dom g*) <−: *f*)) *Int* (*dom g*)) = {}
apply *auto*
done

lemma *Rel-Dom-Restr* [*simp*]: !!*f*. [|*f*:(*A*<−−>*B*); *s*: (*Pow A*) |] ==> ((*s* <: *f*) : (*A* <−−> *B*))
apply (*unfold rel-def*)
apply *fast*
done

Relational Image

lemma *Image-Rel* : *y*:(*R*(|*S*|)) = (? *x*. {*x*} <= *S* & (*x*,*y*):*R*)
apply *auto*
done

lemma *Image-ran* : *R*(|*S*|) = *ran* (*S* <: *R*)
apply *auto*
done

lemma *Image-dom-comp* : $dom(Q\ \%;\ R) = (Q\ \%\sim)\ (|dom\ (R)|)$
apply *auto*
done

lemma *Image-ran-comp* : $ran(Q\ \%;\ R) = R(|ran\ (Q)|)$
apply *auto*
done

lemma *Image-union* [*simp*]: $R(|(S\ Un\ T)\ |) = (R(|S|))\ Un\ (R(|T|))$
apply *auto*
done
declare *Image-union* [*simp*]

lemma *Image-inter* [*simp*]: $R(|(S\ Int\ T)|) <= (R(|S|))\ Int\ (R(|T|))$
apply *auto*
done

lemma *Image-dom* [*simp*]: $R(|dom\ (R)|) = ran(R)$
apply *auto*
done

Overriding

lemma *overrideI1*: $!!p.\ p : dom\ s <-: r ==> p : (r\ (+)\ s)$
apply (*unfold override-def*)
apply *auto*
done

lemma *overrideI2*: $!!p.\ p : s ==> p : (r\ (+)\ s)$
apply (*unfold override-def*)
apply *auto*
done

lemma *overrideCI*[*intro*!]:
assumes *prem*: $(\ p\ \sim: s ==>\ \ p : (dom\ s) <-: r)$
shows $p : (r\ (+)\ s)$
apply (*unfold override-def*)
apply (*fast intro*!: *prem*)
done

lemma *overrideE*[*elim*!]: $[\![\ p{:}(r\ (+)\ s);\ \ p : (dom\ s) <-: r \Longrightarrow Q;\ \ p{:}\ s \Longrightarrow Q]\!] \Longrightarrow Q$
apply (*unfold override-def*)
apply (*rule UnE*)
apply (*auto*)
done

lemma *override-mt-left* [*simp*]: $(\{\}\ (+)\ R) = R$
apply *auto*
done

lemma *override-mt-right* [*simp*]: $(R\ (+)\ \{\}) = R$
apply *auto*
done

lemma *override-idem* [*simp*]: $(R(+)R) = R$
apply *auto*
done

lemma *override-Domain* [*simp*]: $dom(R(+)Q) = (dom\ (Q))\ Un\ (dom\ (R))$
apply (*unfold override-def*)
apply *auto*
done

lemma *override-assoc* : $(P(+)(Q(+)R)) = ((P(+)Q)(+)R)$
apply *auto*
done

lemma *override-left-idem* [*simp*]: $(A\ (+)\ (A\ (+)\ B)) = (A\ (+)\ B)$
apply *auto*
done

lemma *override-inter* [*simp*]: $((dom(Q)\ Int\ dom(R)) = \{\}) ==> ((Q(+)R) = (Q\ Un\ R))$
apply *auto*
done

lemma *override-res-left* [*simp*]: $(V <: (Q(+)R)) = ((V <: Q)\ (+)\ (V <: R))$
apply *auto*
done

lemma *override-res-right* [*simp*]: $((Q(+)R):> W) <= ((Q:>W)\ (+)\ (R:>W))$
apply *auto*
done

lemma *override-single* [*simp*]: $\{(x,y)\}\ (+)\ \{(x,z)\} = \{(x,z)\}$
apply *auto*
done

Closure

lemmas *zrtrancl-def* = *ref-trans-clos-def*

lemma *zrtrancl-fun-mono* : *mono*(%*s*. *idZ*(*dom r Un ran r*) *Un* (*r* %*o s*))
apply (*rule monoI*)
apply *auto*
done

lemmas *zrtrancl-unfold* = *zrtrancl-fun-mono* [*THEN zrtrancl-def* [*THEN def-lfp-unfold*]]

lemma *zrtrancl-refl0* : !!*x*. [| *x* : *dom r Un ran r* |] ==> (*x*,*x*) : (*r* %∗)
apply (*subst zrtrancl-unfold*)
apply *auto*
done

lemmas *zrtrancl-refl* = *UnCI* [*THEN zrtrancl-refl0*]

declare *zrtrancl-refl* [*simp*]
declare *zrtrancl-refl* [*intro*!]

lemma *zrtrancl-into-zrtrancl* : !!*r*. [| (*a*,*b*) : *r*%∗; (*b*,*c*) : *r* |] ==> (*a*,*c*) : *r*%∗
apply (*subst zrtrancl-unfold*)
apply *auto*
done

lemma *r-into-zrtrancl* [*intro*]: !!*p*. *p* : *r* ==> *p* : *r*%∗
apply (*simp* (*no-asm-simp*) *only*: *split-tupled-all*)
apply (*rule zrtrancl-refl* [*THEN zrtrancl-into-zrtrancl*])
apply *auto*
done

lemma *zrtrancl-mono*: !!r. r <= s ==> r%* <= s%*
apply (*unfold zrtrancl-def*)
apply (*rule lfp-mono Un-mono rel-comp-mono subset-refl*)+
apply (*auto*)
done

lemma *zrtrancl-full-induct*:
⟦(a,b) : r%*; !!x. x:*dom* r ⟹ P(x,x); !!y. y:*ran* r ⟹ P(y,y); !!x y z.⟦ P(x,y); (x,y): r%*; (y,z): r ⟧ ⟹ P(x,z)⟧ ⟹ P(a,b)
apply (*rule def-lfp-induct* [*OF zrtrancl-def zrtrancl-fun-mono*])
apply (*assumption*)
apply (*blast*)
done

lemma *zrtrancl-full-induct2*:
assumes *p1*: (a,b) : r%*
assumes *p2*: !!x. x:*dom* r ==> P x x
assumes *p3*: !!y. y:*ran* r ==> P y y
assumes *p4*: !!x y z.[| P x y; (x,y): r%*; (y,z): r |] ==> P x z
shows P a b
apply (*subgoal-tac* P (*fst* (a,b)) (*snd* (a,b)))
apply (*rule-tac* [*2*] *p1* [*THEN zrtrancl-full-induct*])
apply *simp-all*
apply (*erule p2*)
apply (*erule p3*)
apply (*erule p4*)
apply (*assumption*)
apply (*assumption*)
done

lemma *zrtrancl-induct*:
[| (a::'a,b) : r%*;
a: *dom* r ==> P(a);
a: *ran* r ==> P(a);
!!y z.[| (a,y) : r%*; (y,z) : r; P(y) |] ==> P(z) |]
==> P(b)

apply (*subgoal-tac* ! *y*. (*a*::'*a*,*b*) = (*a*,*y*) −−> *P* (*y*))

apply *blast*

apply (*rule zrtrancl-full-induct*)
apply (*blast*)+
done

lemma *trans-zrtrancl*: *trans*(*r*%∗)
apply (*unfold trans-def*)
apply *safe*
apply (*erule-tac b* = *z* **in** *zrtrancl-induct*)
apply (*blast intro*: *zrtrancl-into-zrtrancl*)+
done

lemmas *zrtrancl-trans* = *trans-zrtrancl* [*THEN transD*, *standard*]

lemma *zrtrancl-dom1* [*simp*]: !!*r* .[| (*a*,*b*) : (*r*%∗); *a*~:*ran r* |] ==> *a* : *dom r*
apply (*erule zrtrancl-induct*)
apply *auto*
done

lemma *zrtrancl-ran0* : !!*r* .(*a*,*b*) : (*r*%∗) ==> *b* : *dom r Un ran r*
apply (*erule zrtrancl-induct*)
apply *auto*
done

lemma *zrtrancl-ran1* [*simp*]: !!*r* .[| (*a*,*b*) : (*r*%∗); *b*~:*ran r* |] ==> *b* : *dom r*
apply (*drule zrtrancl-ran0*)
apply *auto*
done

lemma *zrtranclE0*: [| (*a*,*b*) : *r*%∗;
  [| (*a*,*b*): (*idZ* (*dom r Un ran r*))|] ==> *P*;
  !!*y*.[| (*a*,*y*) : *r*%∗; (*y*,*b*) : *r* |] ==> *P*
 |] ==> *P*
apply (*subgoal-tac* (*a*,*b*) : *idZ* (*dom r Un ran r*) | (? *y*. (*a*,*y*) : (*r*%∗) & (*y*,*b*) : *r*))
apply (*erule asm-rl exE disjE conjE*)+

apply *blast*
apply (*erule exE*)
apply *blast*
apply (*rule-tac a = a* **in** *zrtrancl-induct*)
apply (*auto*)
done

lemma *zrtranclE*:
assumes *p1*: (*a*,*b*) : *r*%∗
assumes *p2*: [| *a*:(*dom r*); *a* = *b*|] ==> *P*
assumes *p3*: [| *a*:(*ran r*); *a* = *b*|] ==> *P*
assumes *p4*: !!*y*.[| (*a*,*y*) : *r*%∗; (*y*,*b*) : *r* |] ==> *P*
shows *P*
apply (*rule p1* [*THEN zrtranclE0*])
apply (*erule idZE*)
apply (*erule UnE*)
apply (*rule p2*)
apply (*rule-tac* [*3*] *p3*)
apply (*rule-tac* [*5*] *p4*)
apply *auto*
done

lemmas *zrtrancl-into-zrtrancl2* = *r-into-zrtrancl* [*THEN zrtrancl-trans*, *standard*]

lemma *reach-induct*:
assumes *p1*: *q* : (*r*%∗) (| *Q* |)
assumes *p2*: !! *x*. *x* : *Q* ==> *P x*
assumes *p3*: !! *y z*. [| *y* : (*r*%∗) (| *Q* |); (*y*, *z*) : *r*; *P y* |] ==> *P z*
shows *P q*
apply (*rule p1* [*THEN ImageE*])
apply (*erule zrtrancl-induct*)
apply (*erule p2*)
apply (*erule p2*)
apply (*drule ImageI*)
apply (*assumption*)
apply (*erule p3*)
apply (*assumption*)
apply (*assumption*)
done

lemma *dom-zrtrancl* [*simp*]: *dom* (*r*%∗) = (*dom r Un ran r*)
apply (*auto elim*: *zrtrancl-induct*)
done

lemma *zrtrancl-ran-eq-dom* [*simp*]: !!*r* . *ran*(*r*%∗) = *dom*(*r*%∗)
apply (*rule iffI* [*THEN set-ext*])
apply (*erule RangeE*)
apply (*erule zrtrancl-induct*)
apply *auto*
done

lemma *dom-zrtrancl* : *ran* (*r*%∗) = (*dom r Un ran r*)
apply *auto*
done

lemma *zrtrancl-idemp* [*simp*]: (*r*%∗)%∗ = *r*%∗
apply *auto*
apply (*erule zrtrancl-induct*)
apply (*auto intro*: *zrtrancl-trans*)
done

lemma *zrtrancl-idemp-self-comp* [*simp*]: *R*%∗ *O R*%∗ = *R*%∗
apply (*auto intro*: *zrtrancl-trans*)
done

lemma *zrtrancl-subset-zrtrancl* : !!*r*. *r* <= *s*%∗ ==> *r*%∗ <= *s*%∗
apply (*drule zrtrancl-mono*)
apply *simp*
done

lemma *zrtrancl-subset* : !! *R*. [| *R* <= *S*; *S* <= *R*%∗ |] ==> *S*%∗ = *R*%∗
apply (*drule zrtrancl-mono*)
apply (*drule zrtrancl-mono*)
apply *simp*
done

lemma *zrtrancl-Un-zrtrancl* : (*R*%∗ *Un S*%∗)%∗ = (*R Un S*)%∗
apply (*blast intro*!: *zrtrancl-subset intro*: *r-into-zrtrancl zrtrancl-mono* [*THEN subsetD*])
done

lemma *zrtrancl-converseD* : !!r.(x,y) : (r%~)%* ==> (y,x) : r%*
apply (*erule zrtrancl-induct*)
apply (*auto intro*: *zrtrancl-trans*)
done

lemma *zrtrancl-converseI* : !!r.(y,x) : r%* ==> (x,y) : (r^−1)%*
apply (*erule zrtrancl-induct*)
apply (*auto intro*: *zrtrancl-trans*)
done

lemma *zrtrancl-converse* : ((r%~)%*) = ((r%*)%~)

apply (*rule set-ext*)
apply (*simp* (*no-asm-simp*) *only*: *split-tupled-all*)
apply (*safe dest*!: *zrtrancl-converseD intro*!: *zrtrancl-converseI*)
done

lemma *converse-zrtrancl-induct*:
assumes *major*: (a,b) : r%*
assumes *prem1*: b : *dom* r ==> P(b)
assumes *prem2*: b : *ran* r ==> P(b)
assumes *prem3*: !!y z.⟦ (y,z) : r; (z,b) : r%*; P(z) ⟧ ⟹ P(y)
shows P(a)
apply (*rule major* [*THEN zrtrancl-converseI* [*THEN zrtrancl-induct*]])
apply (*auto intro*: *prem1 prem2 prem3 dest*!: *zrtrancl-converseD*)
done

lemmas *converse-zrtrancl-induct2* = *converse-zrtrancl-induct* [**where** a = (ax,ay) **and** b = (bx,by)]

lemma *converse-zrtranclE0*:⟦(a,b) : r%*; (a,b): (*idZ* (*dom r Un ran r*)) ==>P; !!y.[| (a,y) : r; (y,b) : r%* |] ==> P⟧ ⟹ P
apply (*subgoal-tac* (a,b) : *idZ* (*dom r Un ran r*) | (? y. (a,y) : r & (y,b) : r%*))
apply (*erule disjE*)
apply *blast*+
apply (*erule converse-zrtrancl-induct*)
apply (*auto*)
done

lemma *converse-zrtranclE*:
assumes *p1*: (x,z):r%*
assumes *p2*: [| z:(*dom r*); z = x|] ==> P
assumes *p3*: [| z:(*ran r*); z = x|] ==> P
assumes *p4*: !!y. [| (x,y):r; (y,z):r%* |] ==> P

shows *P*
apply (*rule p1* [*THEN converse-zrtranclE0*])
apply (*erule idZE*)
apply (*erule UnE*)
apply (*rule p2*)
apply (*rule-tac* [*3*] *p3*)
apply (*rule-tac* [*5*] *p4*)
apply *auto*
done

lemmas *converse-zrtranclE2* = *converse-zrtranclE* [**where** *x* = (*xa*,*xb*) **and** *z* = (*za*,*zb*)]

lemma *r-comp-zrtrancl-eq* : *r O r%∗ = r%∗ O r*
apply (*blast elim*: *zrtranclE converse-zrtranclE intro*: *zrtrancl-into-zrtrancl zrtrancl-into-zrtrancl2*)
done

+

lemmas *ztrancl-def* = *trans-clos-def*

lemma *ztrancl-mono*:!!*r*.[| *p* : *r%+*; *r* <= *s* |] ==> *p* : *s%+*
apply (*unfold ztrancl-def*)
apply (*blast intro*: *zrtrancl-mono* [*THEN subsetD*])
done

lemma *ztrancl-into-zrtrancl*:!!*p*. *p* : *r%+* ==> *p* : *r%∗*
apply (*unfold ztrancl-def*)
apply (*simp* (*no-asm-simp*) *only*: *split-tupled-all*)
apply (*erule rel-compEpair*)
apply (*rule zrtrancl-into-zrtrancl*)
apply (*assumption+*)
done

lemma *r-into-ztrancl*[*intro*]: !!*p*. *p* : *r* ==> *p* : *r%+*
apply (*unfold ztrancl-def*)
apply (*simp* (*no-asm-simp*) *only*: *split-tupled-all*)
apply *auto*
done

lemma *zrtrancl-into-ztrancl1*: !!r. [| (a,b) : r%*; (b,c) : r |] ==> (a,c) : r%+
apply (*unfold ztrancl-def*)
apply *auto*
done

lemma *zrtrancl-into-ztrancl2* : !!r. [| (a,b) : r; (b,c) : r%* |] ==> (a,c) : r%+
apply (*erule zrtranclE*)
apply *blast*
apply *blast*
apply (*rule zrtrancl-trans* [*THEN zrtrancl-into-ztrancl1*])
apply (*rule r-into-zrtrancl*)+
apply (*assumption*)+
done

lemma *ztrancl-induct*:
assumes *major*: (a,b) : r%+
assumes *prem1*: !!y. [| (a,y) : r |] ==> P(y)
assumes *prem2*: !!y z.[| (a,y) : r%+; (y,z) : r; P(y) |] ==> P(z)
shows P(b)
apply (*insert major*)
apply (*rule rel-compEpair*)
apply (*unfold ztrancl-def*)
apply *assumption*

apply (*subgoal-tac ALL z*. (y,z) : r --> P (z))

apply *blast*
apply (*erule zrtrancl-induct*)
apply (*insert prem1 prem2*)
apply (*blast intro*: *zrtrancl-into-ztrancl1*)+
done

lemma *ztrancl-trans-induct*:
[| (x,y) : r%+;
!!x y. (x,y) : r ==> P x y;
!!x y z. [| (x,y) : r%+; P x y; (y,z) : r%+; P y z |] ==> P x z
|] ==> P x y
apply (*blast intro*: *ztrancl-induct*)
done

lemma *ztranclE*:
[| (a::'a,b) : r%+;
(a,b) : r ==> P;
!!y.[| (a,y) : r%+; (y,b) : r |] ==> P
|] ==> P
apply (*subgoal-tac* (a::'a,b) : r | (? y. (a,y) : r%+ & (y,b) : r))
apply (*erule disjE*)
apply *blast*
apply (*erule exE*)
apply *blast*
apply (*unfold ztrancl-def*)
apply (*erule rel-compEpair*)
apply (*erule zrtranclE*)
apply *blast*+
done

lemma *trans-ztrancl*: *trans*(r%+)
apply (*unfold ztrancl-def*)
apply (*rule transI*)
apply (*erule rel-compEpair*)+
apply (*rule rel-compI*)
apply (*rule zrtrancl-trans*)
apply (*rule zrtrancl-into-zrtrancl*)
apply (*assumption*)+
done

lemmas *ztrancl-trans* = *trans-ztrancl* [*THEN transD*, *standard*]

lemma *zrtrancl-ztrancl-ztrancl*: !!r. [| (x,y):r%*; (y,z):r%+ |] ==> (x,z):r%+
apply (*unfold ztrancl-def*)
apply (*blast intro*: *zrtrancl-trans*)
done

lemmas *ztrancl-into-ztrancl2* = *transD* [*OF trans-ztrancl r-into-ztrancl*]

lemma *ztrancl-converse*: (*r*^−*1*)%+ = (*r*%+)^−*1*
apply (*unfold ztrancl-def*)
apply (*simp* (*no-asm*) *add*: *zrtrancl-converse converse-rel-comp*)
apply (*simp* (*no-asm*) *add*: *zrtrancl-converse* [*symmetric*] *r-comp-zrtrancl-eq*)
done

lemma *ztrancl-converseI* : !! *r*. (*x*,*y*) : (*r*%+)^−*1* ==> (*x*,*y*) : (*r*^−*1*)%+
apply (*simp add*: *ztrancl-converse*)
done

lemma *ztrancl-converseD* : !!*r*. (*x*,*y*) : (*r*^−*1*)%+ ==> (*x*,*y*) : (*r*%+)^−*1*
apply (*simp add*: *ztrancl-converse*)
done

lemma *converse-ztrancl-induct*:
assumes *major*: (*a*,*b*) : *r*%+
assumes *prem1*: !!*y*. (*y*,*b*) : *r* ==> *P*(*y*)
assumes *prem2*: !!*y z*.[| (*y*,*z*) : *r*; (*z*,*b*) : *r*%+; *P*(*z*) |] ==> *P*(*y*)
shows *P*(*a*)
apply (*rule-tac P* = *P* **in** *ztrancl-induct*)
apply (*rule ztrancl-converseI*)
apply (*rule converseI*)
apply (*rule major*)
apply (*rule prem1*)
apply (*erule converseD*)
apply (*blast intro*: *prem1 prem2 dest*!: *ztrancl-converseD*)
done

lemma *ztranclD* : !!*R*. (*x*,*y*):*R*%+ ==> ? *z*. (*x*,*z*):*R* & (*z*,*y*):*R*%∗
apply (*erule converse-ztrancl-induct*)
apply *auto*
apply (*blast intro*: *zrtrancl-trans*)
done

lemma *lemma1*: !!*r*. [| (*a*,*b*) : *r*%∗; *r* <= *A* %*x A* |] ==> *a*=*b* | *a*:*A*
apply (*erule zrtrancl-induct*)
apply *auto*
done

lemma *ztrancl-subset-Sigma*: !!*r*. *r* <= *A* %*x A* ==> *r*%+ <= *A* %*x A*
apply (*unfold ztrancl-def*)
apply (*blast dest*!: *lemma1*)

done

lemma *Closure-Trans-Inclusion* [*simp*]: $R <= (R\%+)$
apply *fast*
done

lemma *Closure-Trans-Compose* [*simp*]:
$((R\%+)\ \%o\ (R\%+)) <= (R\%+)$
apply (*blast intro*: *ztrancl-trans*)
done

lemma *Closure-Transitive* :
$!!R.\ [|\ R<=Q\ \&\ Q\ \%o\ Q <= Q\ |] ==> R\%+ <= Q$
apply *auto*
apply (*erule ztrancl-induct*)
apply *auto*
done

lemma *Closure-Ref-Inclusion* [*simp*]: $idZ\ (dom\ R) <= R\%*$
apply *auto*
done

lemma *Closure-RefTrans-Inclus* [*simp*]: $R <= R\%*$
apply *auto*
done

lemma *Closure-Trans-RefTrans-Inclus* [*simp*]: $R\%+ <= R\%*$
apply (*rule subsetI*)
apply (*erule ztrancl-into-zrtrancl*)
done

lemma *Closure-RefTransitive* :
$!!R.\ [|\ (idZ\ (dom\ R\ Un\ ran\ R) <= Q)\ ;\ (R<=Q)\ ;\ ((Q\ \%o\ Q) <= Q)\ |] ==>$
$((R\%*\ ) <= Q)$
apply *auto*
apply (*erule zrtrancl-induct*)

apply *auto*
done

lemma *Closure-RefTrans-Id* : (*R*%∗) = ((*R*%+) *Un* (*idZ* (*dom R Un ran R*)))
apply (*rule set-ext*)
apply (*simp* (*no-asm-simp*) *only*: *split-tupled-all*)
apply (*blast elim*: *zrtranclE intro*: *zrtrancl-into-ztrancl1 ztrancl-into-zrtrancl*)
done

lemma *Closure-RefTrans-Id-Union* : *R*%∗ = (*R Un idZ* (*dom R Un ran R*))%+
apply (*rule equalityI*)
apply (*rule Closure-RefTransitive*)
apply (*rule-tac* [*4*] *Closure-Transitive*)
apply *auto*
done

lemma *isString* [*simp*]: *x*:*String*
apply (*unfold String-def*)
apply *auto*
done

end

6.2 Integers in the Mathematical Toolkit

theory *ZInteg* **imports** *ZMathTool* **begin**

constdefs
zpred :: *int*=>*int*
zpred(*Z*) == *Z* − *1*

zsuc :: *int*=>*int*
zsuc(*Z*) == *Z* + *1*

znat and zint

lemma *znat-bin-0*: (*int 0*) = *0*
apply *auto*

done

lemma *zint-inverse*[*simp*]: $\$i(int\ i) = i$
apply *auto*
done

lemma *zless-def-Suc*: $(n < m) = (?\ x.\ m = n + int\ (Suc\ x))$
apply (*rule iffI*)
apply (*simp add*: *zless-iff-Suc-zadd*)
apply (*erule exE*)
apply (*erule ssubst*)
apply *arith*
done

lemma *zle-imp-zless-or-eq*: $z <= w \Longrightarrow z < w | z = (w::int)$
apply *auto*
done

lemma *znat-inverse*: $(b \in \%N) ==> (b = int(\$i\ b))$
apply (*unfold Naturals-def*)
apply *auto*
done

lemma *iter-0*[*simp*]: $(iter\ R\ (int\ 0)) = (idZ\ (dom\ R))$
apply (*unfold iter-def Naturals-def*)
apply (*simp* (*no-asm*) *del*: *znat-bin-0*)
done

lemma *znat-pred*[*simp*]: $!!n.\ n\ \sim= 0 ==> (int\ (n - 1)) = zpred\ (int\ n)$
apply *auto*
apply (*unfold zpred-def*)
apply *arith*
done

lemma *znat-Suc-not-zint-conc-Zero*[*simp*]: $Suc\ n\ \sim= \$i\ 0$
apply (*subst znat-bin-0* [*symmetric*])
apply (*subst zint-inverse*)
apply (*simp* (*no-asm*))
done

lemma *inj-zint*[*simp*]: $[\![\ n: \%N;\ m: \%N\ ]\!] \Longrightarrow ((\$i\ n) = (\$i\ m)) = (n = m)$
apply *auto*

apply (*subst znat-inverse*)
prefer *2*
apply (*subst znat-inverse*)
back back
apply *auto*
done

lemma *in-naturals*: (*0* <= *n*) = (*n* : %*N*)
apply (*unfold Naturals-def*)
apply *auto*
done

lemma *zero-is-natural*: *0* : %*N*
apply (*unfold Naturals-def*)
apply *auto*
done

lemma *one-is-natural*: *1* : %*N*
apply (*unfold Naturals-def*)
apply *auto*
done

lemma *two-is-natural*: *2* : %*N*
apply (*unfold Naturals-def*)
apply *auto*
done

declare *zero-is-natural* [*simp*] *one-is-natural* [*simp*] *two-is-natural* [*simp*]

lemma *in-naturals2*[*simp*]: !!*n*. *0* < *n* ==> (*n* : %*N*)
apply (*unfold Naturals-def*)
apply (*simp add*: *symmetric*)
done

lemma *zpred-zint*: !!*n*. *0* < *n* ==> (($*i* *n*) − *1*) = ($*i* (*zpred n*))
apply (*rule-tac t* = ($*i* *n*) − *1* **in** *zint-inverse* [*THEN subst*])
apply (*rule znat-pred* [*THEN ssubst*])
apply *auto*
done

inequalities

lemma *zless-not-refl3*: $((z::int) < w) ==> z \sim= w$
apply (*subst eq-sym-conv*)
apply (*simp add*: *less-imp-neq*)
done

lemma *less-zsuc-eq-le* [*simp*]: $(a < zsuc\ m) = (a <= m)$
apply (*unfold zsuc-def*)
apply *simp*
done

lemma *less-zpred-eq-le* [*simp*]: $(a <= zpred\ m) = (a < m)$
apply (*unfold zpred-def*)
apply *simp*
done

Number range

lemma *num-range-empty*: $(\{\} = (a\ ..\ b)) = (b < a)$
apply (*unfold numb-range-def*)
apply (*rule iffI*)
apply *auto*
done

lemma *num-range-empty2*: $((a\ ..\ b) = \{\}) = (b < a)$
apply (*unfold numb-range-def*)
apply (*rule iffI*)
apply *auto*
done

declare *num-range-empty* [*simp*] *num-range-empty2* [*simp*]

lemma *num-range-empty-imp*[*simp*]: $!!a.\ b < a ==> (a\ ..\ b)=\{\}$
apply (*unfold numb-range-def*)
apply *auto*
done

lemma *zadd-numb-range-collect*[*simp*]: $\{d.\ EX\ n: (q..m).\ d = ((n::int)+s)\} = ((q+s)..(m+s))$
apply (*unfold numb-range-def*)
apply *auto*
apply (*rule-tac* $x = x - s$ **in** *exI*)
apply (*simp*)
done

lemma *num-range-zless-mem*[*simp*]: !!a::*int*. $n < a$ ==> (n ~: (a..b))
apply (*unfold numb-range-def*)
apply *auto*
done

lemma *num-range-zless-mem2*[*simp*]: !!b::*int*. $b < n$ ==> (n ~: (a..b))
apply (*unfold numb-range-def*)
apply *auto*
done

lemma *num-range-Naturals-mem*:
!!a. [| a:%N; n : (a..b) |] ==> n: %N
apply (*unfold numb-range-def Naturals-def*)
apply *auto*
done

lemma *numb-range-single*[*simp*]: (a .. a) = {a}
apply (*unfold numb-range-def*)
apply *auto*
done

lemma *numb-range-insert-first*: !!a. a <= b ==> ((*zpred* a) .. b) = *insert* (*zpred* a) (a .. b)
apply (*unfold numb-range-def zpred-def*)
apply *auto*
done

lemma *numb-range-insert-last*: !!a. a <= b ==> (a .. (*zsuc* b)) = *insert* (*zsuc* b) (a .. b)
apply (*unfold numb-range-def zsuc-def*)
apply *auto*
done

lemma *zadd-to-zdiff-zle*: ((a::*int*) <= b + c) = (a − c <= b)
apply *auto*
done

lemma *numb-range-mem-zadd*[*simp*]: ((a + b) : (m .. n)) = (a : (m−b .. n−b))

apply (*unfold numb-range-def*)
apply (*simp* (*no-asm*) *add*: *zadd-to-zdiff-zle*)
apply *auto*
done

lemma *numb-range-mem-neq-zsuc*: !!a. a ~= m ==> (a : (*zsuc* m .. n)) = (a : (m .. n))
apply (*unfold numb-range-def zsuc-def*)
apply *auto*
done

declare *numb-range-mem-neq-zsuc* [*symmetric*, *simp*]

lemma *numb-range-mem-neq-zpred*:
!!a. a ~= m ==> (a : (n .. m)) = (a : (n .. *zpred* m))
apply (*unfold numb-range-def*)
apply *auto*
done
declare *numb-range-mem-neq-zsuc* [*symmetric*, *simp*] *numb-range-mem-neq-zpred* [*simp*]

lemma *numb-range-subset*[*simp*]: !!k. k <= m ==> (m .. n) <= (k .. n)
apply (*unfold numb-range-def*)
apply *auto*
done

lemma *numb-range-mem-subset*[*simp*]: !!k. [| k < m ; a : (m .. n) |] ==> (a : (k .. n))
apply (*drule order-less-imp-le*)
apply (*rule subsetD*)
apply *auto*
done

lemma *numb-range-mem-subset2*[*simp*]:
!!n. [| n <= k; a : (m .. n) |] ==> a : (m .. k)
apply (*unfold numb-range-def*)
apply *auto*
done

lemma *numb-range-mem-zsuc-zpred*[*simp*]: !!*a*. [| *a* : (*m* .. *zpred n*) |] ==> (*zsuc a* : (*m* .. *n*))
apply (*unfold numb-range-def zsuc-def*)
apply *auto*
done

lemma *empty-numb-range*[*simp*]: !!*b*. *b* < *c* ==> (((*a* .. *b*) *Int* (*c* .. *d*))) = {}
apply (*unfold numb-range-def*)
apply *auto*
done

lemma *numb-range-Un* [*simp*]: !!*a*. [| *a* <= (*b* + *1*); *b* <= *c* |] ==>
((*a* .. *b*) *Un* (*b* + *1* .. *c*)) = (*a* .. *c*)
apply (*unfold numb-range-def zsuc-def*)
apply *auto*
done

Integers and Naturals

lemma *Integers-Subset*[*simp*]: {(*n*::*int*). (*P n*)}<= %*Z*
apply (*unfold Integers-def*)
apply *auto*
done

lemma *Naturals-1-Mem*[*simp*]: ((*n*::*int*):%*Z*)
apply (*unfold Integers-def*)
apply (*simp* (*no-asm*))
done

lemma *Naturals-Mem*[*simp*]: ~ *neg n* --> *n*:%*N*
apply (*unfold Naturals-def neg-def*)
apply *auto*
done

lemma *Nat-zless-zadd*[*simp*]: !!*b*. [| *b*:%*N*; *a* < *c* |] ==> (*a* < *c* + *b*)
apply (*unfold numb-range-def Naturals-def*)
apply *auto*
done

lemma *Nat-zle-zadd*[*simp*]: !!*b*.[| *b*:%*N* ; *a* <= *c* |] ==> *a* <= *c* + *b*
apply (*unfold numb-range-def Naturals-def*)
apply *auto*
done

lemma *Nat-zsuc*[*simp*]: !!*x*. *x*:%*N* ==> *zsuc x*:%*N*
apply (*unfold zsuc-def Naturals-def*)
apply *auto*
done

lemma *Nat-zadd* [*simp*]: !!*n*. [| *n*:%*N* ; *m*:%*N* |] ==> *n*+*m*:%*N*
apply (*unfold Naturals-def*)
apply *auto*
done

lemma *zsuc-simp*: (*zsuc x*) = (*x* + *1*)
apply (*unfold zsuc-def*)
apply *auto*
done

lemma *Naturals-1-zadd-zle* [*simp*]: !!*n*. (*1*::*nat*) <= *n* ==> ~ (*c* + *n* <= *c*)
apply *auto*
done

lemma *Naturals-1-Mem*[*simp*]: !!*a*. [| *a* : %*N* ; *a*~=*0* |] ==> (*a* : *Naturals-1*)
apply (*unfold Naturals-1-def Naturals-def*)
apply *auto*
done

lemma *zless-eq-zadd-Suc*: *w* < *z* ⟹? *n*. *z* = *w* + *int* (*Suc n*)
apply (*simp only*: *zless-iff-Suc-zadd*)
done

lemma *zless-eq-zadd*: *x* <= *y* = (? *z*::*nat*. *y* = *x* + *int z*)
apply (*rule iffI*)
apply (*drule order-le-less* [*THEN iffD1*])
apply (*erule disjE*)
apply (*drule zless-eq-zadd-Suc*)
apply *fast*
apply (*rule-tac x* = *0* **in** *exI*)
apply *simp*

apply (*erule exE*)
apply *simp*
done

lemma *znatprop1*: !!x. x = *int xnat* ==> P(*xnat*::*nat*) = P($*i* x)
apply *simp*
done

lemma *znatprop2*:
!!x. [|x = *int xnat*; y = *int ynat*|] ==> P(*xnat*::*nat*)(*ynat*::*nat*) = P($*i* x)($*i* y)
apply *simp*
done

lemma *znat-Suc*[*simp*]: *int* (*Suc n*) = *zsuc* (*int n*)
apply (*unfold zsuc-def*)
apply (*arith*)
done

lemma *naturalsI*: !!x. x = *int xnat* ==> (x : %N)
apply *simp*
done

lemma *nat2naturals*: (? *xnat*. x = *int xnat*) = (x : %N)
apply (*rule iffI*)
apply (*erule exE*)
apply *simp*
apply (*unfold Naturals-def*)
apply *simp*
apply (*drule zless-eq-zadd* [*THEN iffD1*])
apply *simp*
done

lemma *naturals-induct*:
assumes *prem1*: x : %N
assumes *prem2*: !! x. P(x) ==> P(x + *int 1*)
assumes *prem3*: P(*int 0*)
shows P(x)
apply (*insert prem1*)
apply (*drule nat2naturals* [*THEN iffD2*])
apply (*erule exE*)

apply (*rule-tac s = P* (*int xnat*) **in** *subst*)
prefer *2*
apply (*induct-tac xnat*)
apply (*rule prem3*)
apply (*rule-tac P = P* **in** *subst*)
prefer *2*
apply (*rule prem2*)
apply *assumption*
apply (*rule-tac s = int* (*xnat + 1*) **and** *t = int xnat + int 1* **in** *subst*)
apply *simp*
apply *auto*
done

lemma *naturals-induct*:
assumes *prem1*: *x* : *%N*
assumes *prem2*: !! *x*.[| *x*:*%N*; *P*(*x*) |] ==> *P*(*x + int 1*)
assumes *prem3*: *P*(*int 0*)
shows *P*(*x*)
apply (*insert prem1*)
apply (*drule nat2naturals* [*THEN iffD2*])
apply (*erule exE*)
apply (*tactic* ⟪*hyp-subst-tac 1*⟫)
apply (*induct-tac xnat*)
apply (*rule prem3*)
apply (*rule-tac P = P* **in** *subst*)
prefer *2*
apply (*rule prem2*)
prefer *2*
apply *assumption*
apply (*simp add*: *ring-eq-simps*)+
done

lemma *z-add-induct*:
assumes *prem1*: *P*(*k*::*int*)
assumes *prem2*: *k* <= *x*
assumes *prem3*: !! *x*. *P*(*x*) ==> *P*(*x + int 1*)
shows *P*(*x*)
apply (*insert prem1 prem2 prem3*)
apply (*drule zless-eq-zadd* [*THEN iffD1*])
apply (*erule exE*)
apply (*simp*)

apply (*induct-tac z*)
apply *simp*
apply (*rule-tac s* = (*k* + *int n*) + *int 1* **and** *t* = *k* + *int* (*Suc n*) **in** *subst*)
apply (*simp add*: *ring-eq-simps*)
apply *auto*
done

lemma *naturals-shift*:
$[\![n \in \%N \ ; \ n \neq 0 \ ]\!] \Longrightarrow \exists \ m. \ (m \in \%N \wedge n = m + 1)$
apply(*rule-tac P* = *n* ≠ *0* **in** *mp*)
by(*erule naturals-induct*,*simp-all*)

lemma *naturals-exhaust* :
assumes *typing* : $n \in \%N$
assumes *H1* : $[\![n \in \%N \ ; \ n = 0 \ ]\!] \Longrightarrow P \ 0$
assumes *H2* : $[\![n \in \%N \ ; \ \exists \ m. \ (m \in \%N \wedge n = m + 1) \ ]\!] \Longrightarrow P \ n$
shows *P n*
apply (*insert typing*)
apply (*cases n* = *0*)
apply (*simp add*: *H1*)
apply (*rule H2*,*simp*)
apply (*frule naturals-shift*,*simp-all*)
done

end

6.3 Functions in the Mathematical Toolkit

theory *ZFun* **imports** *ZMathTool ZInteg* **begin**

declare *dom-implies-ran* [*simp*]

6.3.1 Relation Apply

lemma *apply-singleton-rel*[*simp*]: ({(*x*,*y*)}% ^*x*) = *y*
apply (*unfold rel-apply-def*)
apply (*simp* (*no-asm*))
done

lemma *mem-apply*[*rule-format*]: *x*:*dom R* & (*R*% ^*x*) = *y* −−> (*x*,*y*):*R*

apply (*unfold rel-apply-def dom-def Ex-def*)
apply (*auto intro*: *someI*)
done

lemma *override-apply* [*simp*]:
(*R* (+) {(*x*,*y*)}) %ˆ *x* = *y*
apply (*simp* (*no-asm*) *add*: *override-def dom-substr-def*
rel-apply-def some-equality)
done

lemma *override-by-pair-apply2* [*simp*]:
!!*x*. *x* ~= *z* ==> (*R* (+) {(*z*,*y*)}) %ˆ *x* = *R* %ˆ *x*
apply (*simp* (*no-asm*) *add*: *override-def dom-substr-def*
rel-apply-def some-equality)
apply *auto*
done

lemma *override-by-pair-apply1*[*simp*]:
x = *z* ==> (*R* (+) {(*z*, *y*)}) %ˆ *x* = *y*
by(*auto simp*:*override-def dom-substr-def rel-apply-def*)

lemma *apply-single*: ({(*x*,*y*). *y* = *F x*}%ˆ*z*) = (*F z*)
apply (*unfold rel-apply-def*)
apply (*simp* (*no-asm*))
done

lemma *apply-single-tuple*:
({((*x1*,*x2*),*y*). (*y* = (*F x1 x2*))}%ˆ(*z1*,*z2*)) = (*F z1 z2*)
apply (*unfold rel-apply-def*)
apply (*simp* (*no-asm*))
done

declare *dom-implies-ran* [*simp del*]

lemma *dom-un-apply*[*rule-format*]: (*z* ~: (*dom Q*)) --> (((*R Un Q*)%ˆ*z*) = (*R*%ˆ*z*))
apply (*unfold rel-apply-def*)
apply *auto*
apply (*simp add*: *dom-simp*)
done
declare *dom-un-apply* [*simp*]
declare *dom-implies-ran* [*simp*]

lemma *insert-apply*: !!*a*. *z* ~= *fst a* ==> (*insert a R*) %^ *z* = *R* %^ *z*
apply (*subst insert-is-Un*)
apply (*subst Un-commute*)
apply (*rule dom-un-apply*)
apply *auto*
done

lemma *insert-apply2*: !!*a*. *z* ~: *dom R* ==> (*insert a R*) %^ *z* = {*a*} %^ *z*
apply (*subst insert-is-Un*)

apply (*rule dom-un-apply*)
apply *assumption*
done

declare *insert-apply* [*simp*] *insert-apply2* [*simp*]

lemma *dom-substr-apply* [*simp*]: *z* ~: *S* ==> (*S* <-: *R*) %^ *z* = *R* %^ *z*
apply (*unfold dom-substr-def rel-apply-def*)
apply *auto*
done

lemma *override-apply2* [*simp*]:
(*z* ~: (*dom S*)) ==> (((*R* (+) *S*)%^*z*) = (*R*%^*z*))
apply (*unfold override-def*)
apply *auto*
done

lemma *override-apply1* :
z : *dom S* ==> (*R* (+) *S*) %^ *z* = *S* %^ *z*
by(*auto simp*: *override-def dom-substr-def rel-apply-def*)
declare *override-apply override-apply2*[*simp*]

6.3.2 Relations

lemma *Rel-Dom-subset*: *f* : (*A* <--> *B*) ==> *dom f* <= *A*
apply (*unfold dom-def rel-def*)
apply *auto*
done

lemma *Rel-Dom-Elem*: [| *f* : (*A* <--> *B*); *x* : *dom f* |] ==> *x* : *A*
apply (*unfold dom-def rel-def*)

apply *auto*
done

lemma *empty-rel* [*simp*]: {} : (A <--> B)
apply (*unfold rel-def*)
apply *auto*
done

lemma *select-rel* [*rule-format*,*simp*]: R ~= {} --> {(x,y). x : S & y = (• y. y:R)}
: (S <--> R)
apply (*unfold rel-def*)
apply *auto*
apply (*rule someI*)
apply *assumption*
done

lemma *rel-dom-member*: [| f : (X <--> Y); (x,y) : f |] ==> x : X
apply (*unfold rel-def*)
apply *fast*
done

lemma *rel-pair-dom*[*rule-format*]:
r : (X <--> Y) --> (a,b) : r --> a : X
apply (*unfold rel-def*)
apply *auto*
done

lemma *rel-pair-ran*[*rule-format*]:
r : (X <--> Y) --> (a,b) : r --> b : Y
apply (*unfold rel-def*)
apply *auto*
done

lemma *rel-pair-dom-fst*[*rule-format*]:
r :(X<-->Y) --> (x:r) --> fst x: X
apply (*unfold rel-def*)
apply *auto*
done

lemma *Dom-Collect-Rel-fst* [*simp*]:
dom {p. EX x:A. p = (f x) } = {d. EX x:A. d = fst (f x)}

apply (*simp* (*no-asm*) *add*: *dom-simp*)
apply *auto*
apply (*rule-tac x* = *xa* **in** *bexI*)
apply (*simp add*: *Pair-fst-snd-eq*)+
apply (*rule-tac x* = *snd* (*f xa*) **in** *exI*)
apply (*rule-tac x* = *xa* **in** *bexI*)
apply (*simp add*: *surjective-pairing* [*symmetric*])
apply *assumption*
done

lemma *Rel-Dom-Restr-Ident* [*rule-format*]:
x: (*X* <--> *Y*) --> (*X* <: *x* = *x*)
apply (*unfold rel-def*)
apply *auto*
done

lemma *Rel-Dom-Pow* [*rule-format*]: *x*: (*X* <--> *Y*) --> *dom x* : (*Pow X*)
apply (*unfold Pow-def*)
apply (*simp add*: *Rel-Dom-Restr-Ident*)
done
declare *Rel-Dom-Pow* [*simp*]

lemma *x-in-S-implies-S-nonempty*: *x* : *S* ==> *S* ~= {}

apply *auto*
done

lemma *nonempty-Pow* [*simp*]: *Pow S* ~= {}
apply (*unfold Pow-def*)
apply (*rule x-in-S-implies-S-nonempty*)
apply *auto*
done

lemma *nonempty-Rel* [*rule-format*,*simp*]: *R* ~= {} --> (*D* <--> *R*) ~= {}
apply (*unfold rel-def*)
apply (*simp* (*no-asm*))
done

Partial Functions

lemma *pfunI*:
assumes *p1*: *f* : *A* <--> *B*
assumes *p2*: !! *x y1 y2*. [| (*x*,*y1*) : *f*; (*x*,*y2*) : *f* |] ==> *y1* = *y2*
shows *f* : *A* -|-> *B*

apply (*unfold pfun-def*)
apply (*auto intro*: *p1 simp add*: *p2*)
done

lemma *pfunE*:
[| *f* : *A* −|−> *B*;
  [| *f* : *A* <−−> *B*; ! *x y1 y2*. (*x*,*y1*) : *f* & (*x*,*y2*) : *f* −−> *y1* = *y2* |] ==> *R*
 |] ==> *R*
apply (*unfold pfun-def*)
apply *auto*
done

lemma *Pfun-Single*[*simp*]: {*p*} : (*A* −|−> *B*) = (*fst p* : *A* & *snd p* : *B*)
apply (*unfold pfun-def rel-def Pow-def*)
apply (*rule-tac p* = *p* **in** *PairE*)
apply (*rule iffI*)
apply *simp*+
done

lemma *Partial-Union-Distr*[*rule-format*]:
(*f* : (*X* −|−> *Y*) & *g* : (*X* −|−> *Y*) & (*dom f Int dom g*) = {})
 −−> (*f Un g* : (*X* −|−> *Y*))
apply (*unfold pfun-def dom-simp* [*THEN eq-reflection*])
apply (*simp cong add*: *conj-cong*)
apply (*rule Rel-Union-Distr* [*THEN subst*])
apply *simp*
apply (*rule impI*)
apply (*rule allI*)+
apply (*rule impI*)
apply (*fast elim*: *equalityCE*)
done

lemma *Partial-Union-Distr-rule* [*simp*]:
  [| *f* : (*X* −|−> *Y*); *g* : (*X* −|−> *Y*); (*dom f Int dom g*) = {}|]
  ==> (*f Un g* : (*X* −|−> *Y*))
apply (*simp* (*no-asm-simp*) *add*: *Partial-Union-Distr*)
done

lemma *Partial-Dom-Restr* [*simp*]:
[|*f*:(*A* −|−> *B*); *s*: (*Pow A*) |] ==> ((*s* <: *f*) : (*A* −|−> *B*))
apply (*unfold pfun-def dom-simp* [*THEN eq-reflection*])
apply (*simp cong add*: *conj-cong*)
apply *safe*

apply *fast*
done

lemma *override-pfunI*[*simp*]:
[|*f*:(*A* −|−> *B*); *g*:(*A* −|−> *B*)|] ==> ((*f* (+) *g*):(*A* −|−> *B*))
apply (*unfold override-def*)
apply (*rule Partial-Union-Distr-rule*)
apply (*simp add*: *Dom-Anti-Restr*)
apply (*rule Partial-Dom-Restr*)
apply *simp*
apply (*unfold pfun-def rel-def dom-simp* [*THEN eq-reflection*])
apply (*simp add*: *Pow-def*)
apply *fast+*
done

lemma *pfun-subset*[*elim*!]: [| *f*: (*a* −|−> *c*); *a* <= *b* |] ==> (*f*: (*b* −|−> *c*))
apply (*unfold pfun-def rel-def Pow-def*)
apply *auto*
done

lemma *tfun-implies-pfun*: *f* : *A* −−−> *B* ==> *f* : *A* −|−> *B*
apply (*unfold tfun-def*)
apply *auto*
done

lemma *Pfun-Rel*: *f* : (*A* −|−> *B*) ==> *f* : (*A* <−−> *B*)
apply (*unfold pfun-def*)
apply *auto*
done
declare *tfun-implies-pfun* [*simp*] *Pfun-Rel* [*simp*]

lemma *empty-is-pfun* [*simp*]: {} : (*S* −|−> *R*)
apply (*unfold pfun-def*)
apply *auto*
done

lemma *ex-elim4* [*simp*]: (*EX x. x : S*) = (*S* ~= {})
apply *auto*
done

lemma *select-partial* [*rule-format*,*simp*]:
R ~= {} --> {(*x*,*y*). *x* : *S* & *y* = (*SOME y. y*:*R*)} : (*S* -|-> *R*)
apply (*unfold pfun-def rel-def*)
apply (*simp* (*no-asm*))
apply (*subst ex-elim4* [*symmetric*])
apply *auto*
apply (*erule someI*)
done

lemma *beta-apply-pfun*[*rule-format*,*simp*]:
r : (*X* -|-> *Y*) --> (*a*,*b*) : *r* --> (*r* %^ *a*) = *b*
apply (*unfold pfun-def rel-apply-def*)
apply *auto*
done

lemmas *beta-apply-tfun* = *tfun-implies-pfun* [*THEN beta-apply-pfun*]

lemma *rel-apply-in-rel*[*rule-format*,*simp*]:
r : (*X* -|-> *Y*) -->(*a* : *dom r*) --> (*a*,*r* %^ *a*) : *r*
apply (*unfold pfun-def rel-apply-def*)
apply (*auto*)
apply (*simplesubst some-eq-ex*)
apply (*simp* (*no-asm-simp*))
done

lemma *collect-apply-rel*[*rule-format*]:
r : (*X* -|-> *Y*) --> {(*x*,*y*). *x* : *dom r* & (*r* %^ *x*) = *y*} = *r*
apply *auto*
done

lemma *Insert-Partial*[*rule-format*]:
((*insert x F*) : (*X* -|-> *Y*)) --> (*F* : (*X* -|-> *Y*))
apply (*unfold pfun-def insert-def rel-def*)
apply *simp*
apply *safe*
apply *fast+*
done

lemma *Set-Diff-Partial* [*rule-format*,*simp*]:
$(F : (X -|-> Y)) --> ((F - A) : (X -|-> Y))$
apply (*unfold pfun-def insert-def rel-def*)
apply *simp*
apply *safe*
apply *fast*+
done

declare *dom-implies-ran* [*simp del*]

lemma *Set-Diff-Dom-Partial* [*rule-format*,*simp*]:
$(F : (X -|-> Y)) --> (G :(X -|-> Y)) --> (G <= F) -->$
$((dom\ (F - G)) = ((dom\ F) - (dom\ G)))$
apply (*unfold pfun-def*)
apply *blast*
done

lemma *Rel-Apply-in-Partial-Ran*[*rule-format*]:
$(F : (X -|-> Y)) --> (x: dom\ F) --> (x,F\%\hat{}x):F --> ((F\%\hat{}x) : Y)$
apply (*unfold rel-apply-def pfun-def rel-def*)
apply (*simp add*: *dom-simp*)
apply *auto*
done

declare *Rel-Apply-in-Partial-Ran* [*simp*]

lemma *pfun-apply*[*simp*]:
$[|F : (X -|-> Y);\ x: dom\ F\ |] ==> (F\%\hat{}x) : Y$
apply (*simp add*: *dom-simp*)
done

lemma *Partial-Dom-Insert*[*rule-format*]:
$t : (X -|-> Y)-->a\ ^{\sim}: dom\ t-->a:X --> b:Y--> insert\ (a,b)\ t : (X -|-> Y)$
apply (*unfold insert-def*)
apply *auto*
apply (*unfold pfun-def rel-def*)
apply (*simp add*: *dom-simp*)
apply (*rule allI*)+
apply *auto*
apply (*erule allE*)
apply (*erule-tac* $x = x$ **in** *allE*)

apply (*erule-tac x = y1* **in** *allE*)
apply (*erule-tac x = y2* **in** *allE*)
apply *auto*
done

lemma *Dom-Nat-Part-Func-Add*[*rule-format*,*simp*]:
t : (*Naturals* −|−> *X*) −−> *m*: *Naturals* −−> {*p*. *EX n*:*dom t*. *p* = (*n* + *m*, *t*% ^*n*)} : (*Naturals* −|−> *X*)
apply (*unfold pfun-def rel-def*)
apply *auto*
apply (*subst beta-apply-pfun*)
apply (*auto simp add*: *rel-def pfun-def*)
done

lemma *apply-ident*: *f* : (*X* −|−> *Y*) ⟹ (*f*={*p*. *EX n*: *dom f*. *p* = (*n*,*f*% ^*n*)})
apply *auto*
done

lemma *Partial-Func-Ran-Subset*[*rule-format*]:
Y <= *Z* −−> *f*: (*X* −|−> *Y*) −−> *f*: (*X* −|−> *Z*)
apply (*unfold pfun-def rel-def*)
apply *auto*
done

declare *dom-implies-ran* [*simp*]

lemma *part-func-select*[*rule-format*,*simp*]:
R : (*X* −|−> *Y*) −−> (*x*, *y*) : *R* −−> *y* = (• *z*. (*x*, *z*) : *R*)
apply (*fold rel-apply-def*)
apply (*rule impI*)+
apply (*rule sym*)
apply (*rule beta-apply-pfun*)
apply *assumption*+
done

lemma *nonempty-PFun* [*rule-format*,*simp*]: *R* ~= {} −−> (*D* −|−> *R*) ~= {}
apply (*unfold pfun-def*)
apply (*rule impI*)
apply (*rule x-in-S-implies-S-nonempty*)
apply *auto*
done

Total Functions

lemma *empty-fun*: {} : ({} −−−> *R*)
apply (*unfold tfun-def*)
apply *auto*
done
declare *empty-fun* [*simp*]

lemma *select-total* [*rule-format*,*simp*]:
R ~= {} −−> {(*x*,*y*). *x* : *S* & *y* = (• *y*. *y*:*R*)} : (*S* −−−> *R*)
apply (*unfold tfun-def pfun-def rel-def*
dom-simp [*THEN eq-reflection*])
apply *auto*
apply (*rule someI*)
apply *assumption*
done

lemma *empty-is-pfun*[*simp*]: {} ∈ *A* −|−> *B*
by(*auto simp*: *tfun-def pfun-def rel-def*)

lemma *empty-biject*[*simp*]: {} : ({} >−−>> {})
apply (*auto simp*: *biject-def total-surj-def partial-surj-def*
total-inj-def partial-inj-def)
done

lemma *total-func-dom*[*rule-format*]: *r* : (*X* −−−> *Y*) −−> *x* : *X* −−> *x* : *dom r*
apply (*unfold tfun-def*)
apply *auto*
done
declare *total-func-dom*[*simp*]

lemma *total-is-partial*[*rule-format*,*simp*]: *f* : (*X* −−−> *Y*) −−> *f* : (*X* −|−> *Y*)
apply (*unfold tfun-def*)
apply *auto*
done

lemma *nonempty-Fun*[*rule-format*,*simp*]: *R* ~= {} −−> (*D* −−−> *R*) ~= {}
apply (*unfold tfun-def pfun-def rel-def*)
apply (*rule impI*)
apply (*rule-tac x* = { (*x*,*y*) | *x y*. *x* : *D* & *y* = (• *x*. *x*:*R*) }
in *x-in-S-implies-S-nonempty*)

apply *auto*
apply (*rule someI*)
apply *assumption*
done

Partial Injections

lemma *empty-pinj-fun*[*simp*,*intro*!]: {} : (*S* >−|−> *R*)
apply (*unfold partial-inj-def*)
apply *auto*
done
declare *empty-pinj-fun* [*simp*]

Total Injections

lemma *empty-total-inj*: {} : ({} >−−> *R*)
apply (*unfold total-inj-def*)
apply *auto*
done
declare *empty-total-inj* [*simp*]

Partial Surjections

lemma *empty-psurj-fun*[*simp*]: {} : (*S* −|−>> {})
apply (*unfold partial-surj-def*)
apply *auto*
done

Total Surjections

lemma *empty-total-surj*[*simp*]: {} : ({} −−>> {})
apply (*unfold total-surj-def*)
apply *auto*
done
declare *empty-total-surj* [*simp*]

declare *dom-implies-ran* [*simp del*]

Simplified Definitions

declare *Pfun-Rel* [*simp del*]

lemma *partial-func-simp* [*simp*]:
(*f* : (*A* −|−> *B*)) = (*f* : (*A* <−−> *B*) &
(! *x y1 y2*. (*x*,*y1*):*f* & (*x*,*y2*):*f* −−> *y1*=*y2*))

apply (*unfold pfun-def*)
apply *simp*
done

lemma *total-func-simp*[*simp*]:
(*f* : (*A* −−−> *B*)) = (*f* : (*A* −|−> *B*) & *dom f* = *A*)
apply (*unfold tfun-def*)
apply *simp*
done

lemma *partial-inj-simp* [*simp*]:
(*f* : (*A* >−|−> *B*)) = (*f* : (*A* −|−> *B*) & (! *x1 x2*. (? *y*. (*x1*,*y*):*f* & (*x2*,*y*):*f*) −−> *x1*=*x2*))
apply (*unfold partial-inj-def*)
apply *simp*
done

lemma *total-inj-simp* [*simp*]:
(*f* : (*A* >−−> *B*)) = (*f* : (*A* >−|−> *B*) & *dom f* = *A*)
apply (*unfold total-inj-def Int-def*)
apply *simp*
apply *fast*
done

lemma *partial-surj-simp* [*simp*]:
(*f* : (*A* −|−>> *B*)) = (*f* : (*A* −|−> *B*) & *ran f* = *B*)
apply (*unfold partial-surj-def*)
apply *simp*
done

lemma *total-surj-simp* [*simp*]:
(*f* : (*A* −−>> *B*)) = (*f* : (*A* −|−> *B*) & *dom f* = *A* & *ran f* = *B*)
apply (*unfold total-surj-def*)
apply *simp*
apply *fast*
done

declare *partial-func-simp* [*simp del*]
declare *total-func-simp* [*simp del*]
declare *partial-inj-simp* [*simp del*]

declare *total-inj-simp* [*simp del*]
declare *partial-surj-simp* [*simp del*]
declare *total-surj-simp* [*simp del*]
declare *Pfun-Rel* [*simp*]

lemma *fin-part-funcI*:
!! f. [| f : A −|−> B; *dom* f : *Fin* A |] ==> f : A −||−> B
apply (*unfold fin-part-func-def*)
apply *auto*
done
declare *fin-part-funcI* [*intro*!]

lemma *fin-part-funcE*:
⟦f : A −||−> B; [| f : A −|−> B; *dom* f : %F A |] ==> P ⟧ ⟹ P
apply (*unfold fin-part-func-def*)
apply *auto*
done
declare *fin-part-funcE* [*elim*!]

lemma *insert-is-pfun*[*simp*]:
[| x:A; y:B; x∉*dom* S; S : A −|−> B |] ==> (*insert* (x,y) S) : A −|−> B
by(*auto simp*: *tfun-def pfun-def rel-def*,*fast*)

lemma *pair-pfunI*[*simp*]:
[| x:A; y:B |] ==> {(x,y)} : A −|−> B
by *simp*

lemma *tfun-apply*[*simp*]:
[| F : X −−−> Y; x : X |] ==> F %^ x : Y
by(*simp only*: *total-func-simp*, *erule conjE*,
rule pfun-apply, *auto*)

lemma *pair-exhaust*:
[| x: B %x C |] ==> *EX* y z. x = (y,z)
by *auto*

lemma *tfun-apply-fst*:
[| f : A −−−> B %x C; x : A |] ==> *fst*(f %^ x) : B

apply(*drule tfun-apply*, *assumption*)
by (*frule pair-exhaust*, (*erule exE*)+, *simp*)

lemma *tfun-apply-snd*:
[| *f* : *A* −−−> *B* %*x C*; *x* : *A* |] ==> *snd*(*f* %^ *x*) : *C*
apply(*drule tfun-apply*, *assumption*)
by (*frule pair-exhaust*, (*erule exE*)+, *simp*)

lemma *pfun-apply-fst*:
[| *f* : *A* −|−> *B* %*x C*; *x* : *dom f* |] ==> *fst*(*f* %^ *x*) : *B*
apply(*drule pfun-apply*, *assumption*)
by (*frule pair-exhaust*, (*erule exE*)+, *simp*)

lemma *pfun-apply-snd*:
[| *f* : *A* −|−> *B* %*x C*; *x* : *dom f* |] ==> *snd*(*f* %^ *x*) : *C*
apply(*drule pfun-apply*, *assumption*)
by (*frule pair-exhaust*, (*erule exE*)+, *simp*)

lemma *Pow-right-subset*[*simp*,*elim*!]:
[| *f* : %*P* (*a* %*x b*); *b* <= *c* |] ==> *f* : %*P* (*a* %*x c*)
by(*auto simp*: *Pow-def*)

lemma *Pow-left-subset*[*simp*,*elim*!]:
[| *f* : %*P* (*a* %*x b*); *a* <= *c* |] ==> *f* : %*P* (*c* %*x b*)
by(*auto simp*: *Pow-def*)

lemma *rel-ran-subset*[*elim*!]:
[| *f* : *a* <−−> *b*; *b* <= *c* |] ==> *f* : *a* <−−> *c*
by(*auto simp*: *rel-def*)

lemma *rel-dom-subset*[*elim*!]:
[| *f* : *a* <−−> *c*; *a* <= *b* |] ==> *f* : *b* <−−> *c*
by(*auto simp*: *rel-def*)

lemma *pfun-ran-subset*[*elim*!]:
[| *f* : *a* −|−> *b*; *b* <= *c* |] ==> *f* : *a* −|−> *c*

apply(*simp only*: *pfun-def*)
apply(*auto*)
done

lemma *dom-override-I*:
$[|x{:}dom\ X;\ x{:}dom\ Y|] ==> x : dom\ (X\ (+)\ Y)$
by(*simp add*: *override-Domain*)

lemma *dom-Un-I*:
$[|x{:}dom\ X;\ x{:}dom\ Y|] ==> x : dom\ (X\ Un\ Y)$
by(*simp add*: *Dom-Union*)

lemma *dom-insert-I1*:
$[|x = fst\ a|] ==> x : dom\ (insert\ a\ Y)$
by(*simp add*: *Dom-Insert*)

lemma *dom-insert-I2*:
$[|x \neq fst\ a;\ x{:}dom\ Y|] ==> x : dom\ (insert\ a\ Y)$
by(*simp add*: *Dom-Insert*)

lemma *dom-dres-I*:
$[|x{:}X;\ x{:}dom\ Y|] ==> x : dom\ (X <: Y)$
by(*simp add*: *Dom-Restrict*)

lemma *dom-LambdaI*:
$[|x{:}A|] ==> x : dom\ (Lambda\ A\ f)$
by(*simp add*: *Lambda-dom*)

lemma *pfun-implies-dom-bounded*: $f : A -|-> B \Longrightarrow dom\ f \subseteq A$
by(*simp only*: *pfun-def rel-def*, *auto*)

lemma *override-tfunI1*[*simp*]:
$[\![\ f : A ---> B;\ g : A -|-> B\ ]\!] \Longrightarrow f(+)g : A ---> B$
by(*simp only*: *tfun-def*, *auto dest*!: *pfun-implies-dom-bounded*)

lemma *override-tfunI2*[*simp*]:
$[\![\ f : A -|-> B;\ g : A ---> B\ ]\!] \Longrightarrow f(+)g : A ---> B$
by(*simp only*: *tfun-def*, *auto dest*!: *pfun-implies-dom-bounded*)

lemma *pfun-ext*:
assumes *f-pfun*: $f : A -|-> B$
assumes *g-pfun*: $g : A -|-> B$
assumes *dom-eq*: $dom\ f = dom\ g$
assumes *cong*: $!!\ i.\ [\![\ i : dom\ f;\ dom\ f = dom\ g\ ]\!] \Longrightarrow f\ \%\hat{}\ i = g\ \%\hat{}\ i$
shows $f = g$
apply(*rule set-ext*, *insert f-pfun g-pfun dom-eq*, *auto*)
apply(*rule-tac t=b* **in** *subst*) **defer** *1*
apply(*erule rel-apply-in-rel*,*auto*)
apply(*rule-tac t=b* **in** *subst*) **defer** *1*
apply(*erule rel-apply-in-rel*,*auto*)
apply(*subst beta-apply-pfun* [*of f - - a b*,*symmetric*],*assumption*+)
apply(*rule cong*[*symmetric*], *auto*)
apply(*subst beta-apply-pfun* [*of g - - a b*,*symmetric*]) **prefer** *3*
apply(*rule cong*)
apply *auto*
done

Due to internal translation restrictions, certain expressions were represented by *rel_applf*‘(*j*..*k*). The following property removes this artifact:

lemma *rel-appl-norm* :
rel-appl f ‘ (*j* .. *k*) = {*a*. *EX i*:*j*..*k*. *f* %ˆ *i* = *a*}
by(*auto*)

end

6.4 Finite Sets for the Mathematical Toolkit

theory *ZFinite* **imports** *ZFun* **begin**

axioms
Fin-subset: [| *A* <= *B*; *B*: %*F M* |] ==> *A*: %*F M*

lemma *Fin-mono*: !!*A B*. *A* <= *B* ==> %*F A* <= %*F B*
apply (*unfold Fin.defs*)
apply (*rule lfp-mono*)
apply *auto*
done

lemma *Fin-mono2*: [| $X <= Y$; $a : \%F\ X$ |] ==> $a : \%F\ Y$
apply (*cut-tac Fin-mono*)
prefer *2*
apply *assumption*
apply *fast*
done

lemma *Fin-subset-Pow*: $\%F\ A <= \%P\ A$
apply (*unfold Fin.defs*)
apply (*rule lfp-lowerbound*)
apply *fast*
done

lemmas *FinD* = *Fin-subset-Pow* [*THEN subsetD* [*THEN PowD*]]

lemma *Fin-induct*:
assumes *major1*: $F : \%F\ A$
assumes *major2*: $P\ (\{\})$
assumes *prems*: $!!F\ x.$ [| $x : A$; $F : \%F\ A$; $x\ \sim: F$; $P(F)$ |] ==> $P(insert\ x\ F)$
shows $P(F)$
apply (*rule major1* [*THEN Fin.induct*])
defer *1*
apply (*case-tac a : b*)
apply (*erule insert-absorb* [*THEN ssubst*], *assumption*)
apply (*assumption*, *rule prems*)+
done

declare *Fin.insertI* [*simp*]
declare *Fin.emptyI* [*simp*]

declare *Fin.intros* [*simp*]

lemma *Fin-UnI*:
assumes *major*: $F : \%F\ A$
assumes *prems*: $G : \%F\ A$
shows $F\ Un\ G : \%F\ A$
apply (*rule prems* [*THEN Fin-induct*])
apply (*simp add*: *prems Un-insert-left*)+
done

lemma *subset-Fin* [*simp*]: (*F Un G* : %*F A*) = (*F* : %*F A* & *G* : %*F A*)
apply (*fast intro*: *Fin-UnI dest*: *Un-upper1* [*THEN Fin-subset*] *Un-upper2* [*THEN Fin-subset*])
done

lemma *insert-Fin* [*simp*]: (*insert a A* : %*F M*) = (*a* : *M* & *A* : %*F M*)
apply (*auto intro*!: *Fin.emptyI Fin.insertI dest*: *FinD*)
apply (*rule mp*)
prefer *2*
apply (*assumption*)
apply (*subst insert-is-Un*)
apply (*subst subset-Fin*)
apply *auto*
done

lemma *Fin-imageI*: *F* : %*F A* ==> *h*‘*F* : %*F* (*h*‘*A*)
apply (*erule Fin.induct*)
apply (*simp* (*no-asm*))
apply (*simp* (*no-asm-simp*) *add*: *image-eqI* [*THEN Fin.insertI*] *image-insert del*: *insert-Fin*)
done

lemma *aux*:
assumes *major*: *c* : %*F A*
assumes *prem1*: *b*: %*F A*
assumes *prem2*: *P*(*b*)
assumes *prem3*: !!(*x*::'*a*) *y*. ⟦*x* : *A*; *y* : %*F A*; *x* : *y*; *P*(*y*) ⟧ ⟹ *P*(*y*−{*x*})
shows *c*<=*b* ⟶ *P*(*b*−*c*)
apply (*rule major* [*THEN Fin-induct*])
prefer *2*
apply (*subst Diff-insert*)
prefer *2*
apply (*simp* (*no-asm-simp*) *add* : *prem1 prem2 prem3 Diff-subset* [*THEN Fin-subset*])+
done

lemma *Fin-empty-induct*: [| *b* : %*F A*; *P*(*b*); !!*x y*. [| *x* : *A*; *y* : %*F A*; *x* : *y*; *P*(*y*) |] ==> *P*(*y* − {*x*})|] ==> *P*({})

apply (*rule Diff-cancel* [*THEN subst*])
apply (*rule aux* [*THEN mp*])
apply *assumption+*
apply *auto*
done

6.4.1 Finite Sets

lemma *Fin-Diff-Set*: [|(*A* − *B*) : %*F X*; *B* : %*F X*|] ==> *A* : %*F X*
apply (*drule Fin-UnI*)
apply *fast*
apply (*rule-tac B* = *A* − *B Un B* **in** *Fin-subset*)
apply *fast*
apply *fast*
done

lemma *size-empty*: (# {}) = *int 0*
apply (*unfold zsize-def*)
apply (*simp* (*no-asm*))
done
declare *size-empty* [*simp*]

lemma *card-empty-set*: (*t* = {}) ==> (*card t* = *0*)
apply *auto*
done

lemma *Fin-finite*: *A*: %*F X* ==> *finite A*
apply (*unfold lfp-def insert-def Un-def Inter-def Finites.defs Fin.defs*)
apply *simp*
apply (*rule allI*)
apply (*rule impI*)
apply (*erule allE*)
apply (*erule impE*)
apply *auto*
done

lemma *finite-Fin*: [|*finite A*; *A* <= *X* |] ==> *A* : %*F X*
apply (*rule-tac F* = *A* **in** *finite-subset-induct*)
apply *assumption+*
apply (*rule Fin.emptyI*)
apply (*rule Fin.insertI*)
apply *fast+*
done

lemma *finite-vs-Fin*: (*finite A* & *A* <= *X*) = (*A* : %*F X*)
apply (*rule iffI*)
apply (*blast intro*: *Fin-finite finite-Fin*)
apply (*rule conjI*)
apply (*blast intro*!: *Fin-finite finite-Fin FinD*)
apply (*blast intro*!: *Fin-finite finite-Fin FinD*)
done

lemma *Collect-Dom-Insert*: {*d*. *EX n* : *insert x F*. *d* = *xa n*} = {*d*. *EX n* : *F*. *d* = *xa n*} *Un* {*xa x*}
apply *auto*
done

lemma *Collect-Fin-Func*:
assumes *prem1*: *A* : %*F X*
assumes *prem2*: {*d*. *EX x*:*X*. *d* = *f x*} <= *X*
shows {*d*. *EX n* : *A*. *d* = (*f n*) } : %*F X*
apply (*cut-tac prem2*)
apply (*rule mp*)
prefer *2*
apply (*assumption*)
apply (*cut-tac prem1*)
apply (*rule mp*)
prefer *2*
apply (*assumption*)
apply (*rule-tac x* = *f* **in** *spec*)
apply (*rule-tac F* = *A* **and** *A* = *X* **in** *Fin-induct*)
apply *safe*
apply *simp*
apply (*erule-tac x* = *xa* **in** *allE*)
apply (*subst Collect-Dom-Insert*)
apply *auto*
done

lemma *Collect-Fin-Func-Subset*: [|*A* : %*F X* ; {*d*. *EX x*:*X*. *d* = *f x*} <= *X*|] ==> {*d*. *EX n* : *A*. *d* = (*f n*) } : %*F X*
apply (*rule Collect-Fin-Func*)
apply *assumption*
apply *assumption*
done

lemma *insert-card-eq*:
[|*x* ~: *F*; *y* ~: *G*; *finite F*; *finite G*; *card F* = *card G*|] ==>

card (*insert x F*) = *card* (*insert y G*)
apply *auto*
done

lemma *Nat-Fin-Add-Collect*: [|*A* : %*F* %*N* ; *m* : %*N*|] ==> {*d*. *EX n* : *A*. *d* = (*n* + *m*)}: %*F* %*N*
apply (*rule Collect-Fin-Func*)
apply *assumption*
apply *auto*
done

lemma *Collect-Dom-Insert*:
{*d*. *EX n* : *insert x F*. *d* = *xa n*} = *insert* (*xa x*) {*d*. *EX n* : *F*. *d* = *xa n*}
apply *auto*
done

lemma *size-image*:
!!*A*. [|*A* : %*F X* ; *inj-on f A* |] ==> #(*f* ‘ *A*) = #*A*
apply (*unfold zsize-def*)
apply *auto*
apply (*rule card-image*)
apply (*drule Fin-finite*)
apply *auto*
done

lemma *fin-finite*: !!*A*. *A* : %*F X* ==> *finite A*
apply (*erule Fin-induct*)
apply *auto*
done

lemma *size-insert-disjoint* [*simp*]: !!*A*. [|*A* : %*F X* ; *x* ~: *A*|] ==> #(*insert x A*) = *zsuc* (# *A*)
apply (*unfold zsize-def*)
apply (*drule fin-finite*)
apply (*subst znat-Suc* [*THEN sym*])
apply *auto*
done

lemma *card-size-eq*: (*card m*) = ($*i* (# *m*))
apply (*unfold zsize-def*)
apply *simp*
done

lemma *Size-Naturals* [*simp*]: #*m* : %*N*
apply (*unfold zsize-def*)
apply (*subst card-size-eq*)
apply *auto*
apply (*case-tac #m = 0*)
apply *auto*
done

lemma *size-empty2*: !!*x*. *m* : %*F X* ==> (# *m* = *0*) = (*m* = {})
apply (*erule Fin-induct*)
apply (*simp-all add*: *zsuc-def*)
apply (*auto*)
apply(*subgoal-tac 0* ≤ # *F*,*arith*)
apply(*subgoal-tac #F* : %*N*, *simp add*: *in-naturals*[*symmetric*])
apply(*simp add*: *Size-Naturals*)
done

lemma *card-empty2*: *m* : %*F X* ==> ((*card m* = *0*) = (*m* = {}))
apply (*erule Fin-induct*)
apply (*simp* (*no-asm*))
apply (*subst card-insert-disjoint*)
apply (*erule Fin-finite*)
apply *assumption*
apply *simp*
done

lemma *size-set-notempty* [*rule-format*]: !!*m*. *m* : %*F X* ==> *m* ~= {} --> *0* < #*m*
apply (*erule Fin-induct*)
apply *auto*
done

lemma *size-set-empty*: # {} = *0*
apply (*rule size-empty2* [*THEN iffD2*])

apply *auto*
done

lemma *card-notempty*: !!x. [|x ~= {} ; x : %F X|] ==> *card* x ~= *0*
apply *auto*

apply (*drule card-empty2* [*THEN subst*])
apply *fast+*
done

6.4.2 Finite Functions

lemma *empty-fin-pfun*[*simp*, *intro*!]: {} : (S −||−> R)
apply (*unfold fin-part-func-def*)
apply *auto*
done

lemma *empty-fin-pinj*[*simp*, *intro*!]: {} : (S >−||−> R)
apply (*unfold fin-part-inj-def*)
apply *auto*
done

lemma *override-fpfun*[*simp*]: [|f:(A −||−> B) ; g:(A −||−> B)|] ==> ((f (+) g):(A −||−> B))
apply (*unfold fin-part-func-def*)
apply *simp*
done

lemma *fin-part-fun-single* [*simp*]: [|*fst* p : A ; *snd* p : B |] ==> {p} : (A −||−> B)
apply (*unfold fin-part-func-def*)
apply *auto*
done

lemma *Fin-Dom-Partial*: !!t.[|t : %F (X %x Y) ; t : (X −|−> Y)|] ==> (*dom* t: %F X)
apply (*erule Fin-induct*)
apply *auto*
done

lemma *Dom-Partial-Fin-lemma* [*rule-format* (*no-asm*)]:
!! *dt*. *dt* : %F X ==> *ALL* t . t:(X −|−> Y) −−> *dom* t = *dt* −−> t : *Fin* (X %x Y)

apply (*erule Fin-induct*)
apply *auto*
apply (*erule-tac x = t − { (x,t%^x) }* **in** *allE*)
apply (*auto intro*: *Fin-Diff-Set*)
done

lemma *Dom-Partial-Fin*: !! *t*. [|*t* : (*X* −|−> *Y*); *dom t* : %*F X*|] ==> (*t* : *Fin* (*X* %*x Y*))
apply (*auto intro*: *Dom-Partial-Fin-lemma*)
done

lemma *Fin-Partial-Func*[*simp*]: !!*t*. *t* : (*X* −||−> *Y*) ==> *t* : %*F* (*X* %*x Y*)
apply (*unfold fin-part-func-def*)
apply (*simp add*: *Dom-Partial-Fin*)
done

lemma *Fin-Partial-Dom-Func* [*simp*]: !!*t*. *t* : (*X* −||−> *Y*) ==> *dom t* : %*F X*
apply (*unfold fin-part-func-def*)
apply *auto*
done

lemma *Fin-Partial-Partial-Func* [*simp*]: *t* : (*X* −||−> *Y*) ==> *t*: (*X* −|−> *Y*)
apply (*unfold fin-part-func-def*)
apply *auto*
done

lemma *Dom-Int-Fin-Func-Add*:
[|*dom t* : %*F* %*Z* ; *t* : (%*Z* −|−> *X*); *m* : %*Z*|] ==>
dom {*p*. *EX n*:*dom t*. *p* = (*n* + *m*, *t*%^*n*)} : %*F* %*Z*
apply (*subst Dom-Collect-Rel-fst*)
apply (*rule Collect-Fin-Func-Subset*)
apply *auto*
done

lemma *Dom-Nat-Fin-Func-Add*:
[|*dom t* : %*F* %*N* ; *t* : (%*N* −|−> *X*) ; *m* : %*N*|] ==>
dom {*p*. *EX n*:*dom t*. *p* = (*n* + *m*, *t*%^*n*)} : %*F* %*N*
apply (*subst Dom-Collect-Rel-fst*)
apply (*rule Collect-Fin-Func-Subset*)
apply *auto*
done

lemma *Dom-Nat-Fin-Part-Func-Add* [*simp*]:[|*t* : (%*N* −||−> *X*) ; *m* : %*N*|] ==>
{*p*. *EX n*:*dom t*. *p* = (*n* + *m*, *t*% ^*n*)} : (%*N* −||−> *X*)
apply (*unfold fin-part-func-def*)
apply *safe*
apply (*rule-tac* [*2*] *Dom-Nat-Fin-Func-Add*)
apply *auto*
done

lemma *Fin-Part-Func-Dom-Size-aux*:
assumes *prem1*: (*dom t*) = *dt*
assumes *prem2*: *t*:(*X* −|−> *Y*)
assumes *prem3*: *dt* : %*F X*
shows # *t* = # (*dom t*)
apply (*unfold zsize-def*)
apply *auto*
apply (*insert prem1 prem2 prem3*)
apply (*rule mp*)
prefer *2*
apply (*assumption*)
apply (*rotate-tac 2*)
apply (*rule mp*)
prefer *2*
apply (*assumption*)
apply (*rotate-tac 2*)
apply (*rule mp*)
prefer *2*
apply (*assumption*)
apply (*rule-tac x* = *t* **in** *spec*)
apply (*rule-tac x* = *Y* **in** *spec*)
apply (*rule-tac F* = *dt* **and** *A* = *X* **in** *Fin-induct*)
apply *safe*
apply *simp*
apply (*erule-tac x* = *xa* **in** *allE*)
apply (*erule-tac x* = *xb*−{ (*x*,*xb*% ^*x*) } **in** *allE*)
apply *auto*
apply (*rotate-tac 7*)
apply (*rotate-tac 7*)
apply (*rule-tac s* = *Suc* (*card F*) **and** *t* = *card* (*insert x F*) **in** *subst*)
apply (*rule card-insert-disjoint* [*THEN sym*])
apply (*rotate-tac 5*)
apply (*drule finite-vs-Fin* [*THEN iffD2*])
apply (*drule conjE*)

apply *auto*
apply (*rotate-tac 1*)
apply (*drule sym*)
apply *auto*
apply (*rule card-Suc-Diff1* [*THEN sym*])
apply *auto*
apply (*rule-tac X = X%xxa* **in** *Fin-finite*)
apply (*rule Dom-Partial-Fin*)
apply *auto*
done

lemma *Fin-Part-Func-Dom-Size*: *t* : (*X* −||−> *Y*) ==> # *t* = # (*dom t*)
apply (*rule-tac X = X* **and** *Y = Y* **in** *Fin-Part-Func-Dom-Size-aux*)
apply *auto*
done

lemma *Fin-part-func-union* [*simp*]:
[|*f* : (*X* −||−> *Y*) ; *g* : (*X* −||−> *Y*) ; ((*dom f*) *Int* (*dom g*)) = {}|] ==>
f Un g : (*X* −||−> *Y*)
apply (*unfold fin-part-func-def*)
apply (*rule CollectI*)
apply (*rule conjI*)
apply (*erule CollectE*)+
apply (*erule conjE*)+
apply (*rule Partial-Union-Distr*)
apply *fast*
apply (*simp* (*no-asm*))
apply (*rule conjI*)
apply *auto*
done

6.4.3 Finite Number Range

lemma *fin-x-upto-x*: (*x* .. *x*) : %*F* %*Z*
apply *simp*
done

lemma *numb-range-Fin-add*: *a*:%*N* ⟹(*a* .. (*a* + *int b*)) : %*F* %*N*
apply (*induct-tac b*)
apply (*simp* (*no-asm*))
apply (*simp* (*no-asm*))+

apply (*rule-tac t = a + (1 + int n)* **and** *s = zsuc (a + int n)* **in** *subst*)
prefer *2*
apply (*rule numb-range-insert-last* [*THEN ssubst*])
apply *simp+*
apply (*unfold zsuc-def*)
apply (*simp add*: *ring-eq-simps*)
done

lemma *numb-range-Fin-zadd-aux*: [|*a*:%*N*; *b*:%*N*|] ==> (*a*..*a*+*b*) : %*F* %*N*
apply (*rule-tac t = b* **in** *znat-inverse* [*THEN ssubst*])
apply *assumption*
apply (*rule numb-range-Fin-add*)
apply *assumption*
done

lemma *numb-range-Size-add*: *a*:%*N* ==> #(*a* .. (*a* + *int b*)) = *zsuc* (*int b*)
apply (*induct-tac b*)
apply (*simp add*: *zsuc-def zsize-def*)
apply *auto*
apply (*rotate-tac 1*)
apply (*rule-tac t = a + (1 + int n)* **and** *s = zsuc (a + int n)* **in** *subst*)
prefer *2*
apply (*rule numb-range-insert-last* [*THEN ssubst*])
apply *simp*
apply (*unfold zsize-def*)
apply (*rule-tac t = 1 + int n* **and** *s = zsuc (int n)* **in** *subst*)
prefer *2*
apply (*rule card-insert-disjoint* [*THEN ssubst*])
apply *auto*
apply (*rule-tac X = %N* **in** *Fin-finite*)
apply (*simp add*: *numb-range-Fin-zadd-aux*)
apply (*subst znat-inverse*)
apply *auto*
apply (*unfold zsuc-def*)
apply (*simp add*: *ring-eq-simps*)+
done

lemma *numb-range-Size-zadd1*: !!*b*. [|*b*:%*N* ; *a*:%*N*|] ==> #(*a* .. (*a* + *b*)) = *zsuc b*
apply (*rule-tac t = b* **in** *znat-inverse* [*THEN ssubst*])

apply *assumption*
apply (*rule numb-range-Size-add*)
apply *assumption*
done

lemma *numb-range-Size-zadd* [*simp*]: !!*a*. [|*a* : %*N* ; *a* <= (*b* + *1*)|] ==> #(*a* .. *b*) = *zsuc* (*b* − *a*)
apply (*case-tac a* <= *b*)
apply (*rule-tac a1* = *a* **in** *numb-range-Size-zadd1* [*THEN subst*])
prefer *2*
apply *assumption*
apply (*unfold Naturals-def*)
apply *auto*
apply (*unfold zsuc-def*)
apply *arith*
done

lemma *aux*: !!*z*. ~ *z* <= *w* ==> *w* < (*z*::*int*)
apply *auto*
done

lemma *numb-range-Fin-zadd*: !!*a*. [|*a* : %*N*|] ==> (*a* .. *b*) : %*F* %*N*
apply (*case-tac a* <= *b*)
prefer *2*
apply (*drule aux*)
prefer *2*
apply (*simp add*: *zless-eq-zadd*)
apply (*erule exE*)
apply *auto*
apply (*simp add*: *numb-range-Fin-zadd-aux*)
done
declare *numb-range-Fin-zadd* [*simp*]

lemma *zadd-numb-range-Fin-Nat* [*simp*]: !!*a*. [| *a* : %*N* ; *c* : %*N*|] ==> (*a*+*c* .. *b*+*c*) : %*F* %*N*
apply *auto*
done

lemma *numb-range-set-notempty* [*simp*]: !!*x*. [|*x*: %*F X* ; *x* ~= {} |]==> (*1*: (

1..#x))
apply (*unfold Naturals-def numb-range-def*)
apply *simp*
apply (*drule size-set-notempty*)
apply *simp*
apply *auto*
done

lemma *numb-range-mem* [*simp*]: [|*a* <= *n* ; *n* <= *b*|] ==> *n* : (*a* .. *b*)
apply (*unfold numb-range-def*)
apply *auto*
done

lemma *upto-size-empty*: *y* < *x* ==> #(*x* .. *y*) = *int 0*
apply *auto*
done

lemma *size-x-upto-x*: *card* (*x* .. *x*) = *1*
apply *auto*
done

end

6.5 Sequences in the Mathematical Toolkit

theory *ZSeq* **imports** *ZFinite Wellfounded-Relations* **begin**

types
'a seq = *int* <=> *'a*

constdefs
seq ::*'a set* => (*'a seq*) *set*
seq S == { *f*. (*f*: (*Naturals* −||−> *S*)) & (*dom f* = (*1* .. (# *f*)))}
seq1 ::*'a set* => (*'a seq*) *set*
seq1 S == {*f*. *f*: (*seq S*) & ((*int* (*0*::*nat*)) < (# *f*))}

iseq ::'*a set* => ('*a seq*) *set*
iseq S == (*seq S*) *Int* (*Naturals* >−||−> *S*)

emptyseq ::'*a seq* (%<%>)
emptyseq == {}
insertseq ::['*a*, '*a seq*] => '*a seq*
insertseq x s == *insert* (*1*,*x*) {*p*. *EX n*: *dom s*. *p* = (*zsuc n*,*s*%^*n*)}

insertseq-rel :: '*a set* => ('*a seq* ∗ '*a seq*) *set*
insertseq-rel A == {(*s*,*s*'). *s* : *seq A* & (? *a*. *s*' = *insertseq a s*)}

seq-case :: ['*a set*, '*b*, ('*a* => '*a seq* => '*b*), '*a seq*] => '*b*
seq-case A b f s == • *z*. (*s* = %<%> −−> *z* = *b*) & (! *a*:*A*. ! *t*:*seq A*. *s* = *insertseq a t* −−> *z* = *f a t*)

seq-rec :: ['*a set*, '*b*, ['*a*, '*a seq*, '*b*] => '*b*, '*a seq*] => '*b*
seq-rec A b d == *wfrec* (*insertseq-rel A*) (%*f*. *seq-case A b* (%*a t*. *d a t* (*f t*)))

mhead :: '*a set* => '*a seq* <=> '*a*
mhead A == *lambda s*: *seq A*. *s* %^ *1*

mtail :: '*a set* => '*a seq* <=> '*a seq*
mtail A == *lambda s* : *seq A*. *lambda n* : (*1* .. ((# *s*) − (*1*))) . *s* %^ (*n*+*1*)

syntax
• *Sequence* :: *args* => '*a seq* (%<(-)%>)

translations
%< *x*, *xs* %> == *insertseq x* %<*xs*%>
%< *x* %> == *insertseq x* %<%>

axioms

lem1: !!*x s*. [| *s* : *seq A*; *x* : *s* |] ==> *x* = (*1*, *mhead A* %^ *s*) | (*EX n* : *1* .. # *s* − *1*. *x* = (*zsuc n*, *mtail A* %^ *s* %^ *n*))
insert-mhead-mtail: !!*s*. [| *s* : *seq A*; *s* ~= %<%> |] ==> *s* = *insertseq* (*mhead A* %^ *s*) (*mtail A* %^ *s*)
seq-mem-dom-neq-first: [|*s*: *seq X* ; *a*:*s* ; *a* ~= (*1*,*s*%^*1*)|] ==> ((*fst a*) : (*2*..#*s*))

Dom-insertseq: [|*y*:*X*; *t*:*seq X*|] ==> *dom* (*insertseq y t*) = (*1*..#(*insertseq y t*))
insertseq-eq: [|*s*:*seq X*; *t*:*seq Y*; *insertseq a s* = *insertseq b t*|] ==> (*a*=*b*)

ML
⟪
fun au i = (by (SELECT-GOAL (auto-tac (claset(), simpset())) 1))
⟫

lemma *seqI*: !!*s*. [| *s* : %*N* −||−> *A* ; *dom s* = (*1* .. # *s*) |] ==> *s* : *seq A*
apply (*unfold seq-def*)
apply *auto*
done

lemma *seqE*: [| *s*:*seq A*; [| *s* : %*N* −||−> *A* ; *dom s* = (*1* .. # *s*) |] ==> *R* |] ==> *R*
apply (*unfold seq-def*)
apply *auto*
done

lemma *emptyseqI* [*simp*]: !!*s*. *x*:*s* ==> *s* ~= %<%>
apply (*unfold emptyseq-def*)
apply *auto*
done

lemma *mhead-eq*[*simp*]: !!*s*. [| *s* : *seq A*; *s* ~= %<%> |] ==> *mhead A* %^ *s* = *s*%^*1*
apply (*unfold mhead-def*)
apply *auto*
done

lemma *mtail-eq* [*simp*]: !!*s*. [| *s* : *seq A*; *s* ~= %<%> |] ==> *mtail A* %^ *s* = (*lambda n* : (*1* .. ((# *s*) − (*1*))) . *s* %^ (*n* + *1*))
apply (*unfold mtail-def*)
apply (*simp*)
done

lemma *aux*: *zpred 2* = *1*
apply (*unfold zpred-def*)
apply *auto*
done

lemma *aux1*: *zsuc* (*x* − *int 1*) = *x*
apply (*unfold zsuc-def*)

apply *auto*
done

sequences-general

lemma *seq-ran-subset* [*simp*]: (*s* : *seq X*) ==> (*ran s* <= *X*)
apply (*unfold seq-def fin-part-func-def pfun-def rel-def*)
apply (*simp* (*no-asm*) *add*: *ran-simp*)
apply *auto*
done

lemma *size-emptyseq* [*simp*]: # %<%> = *0*
apply (*unfold emptyseq-def zsize-def*)
apply (*simp* (*no-asm*))
done

lemma *ran-emptyseq* [*simp*]: *ran* (%<%>) = {}
apply (*unfold emptyseq-def*)
apply (*simp* (*no-asm*))
done

lemma *dom-emptyseq* [*simp*]: *dom* (%<%>) = {}
apply (*unfold emptyseq-def*)
apply (*simp* (*no-asm*))
done

lemma *seq-imp-Fin-func*: *t*: *seq X* −−> *t* : (%*N* −||−> *X*)
apply (*unfold seq-def*)
apply *auto*
done

lemma *seq-imp-numb-range*: *t*: *seq X* ==> (*dom t*) = (*1* ..#*t*)
apply (*unfold seq-def*)
apply *auto*
done

lemma *empty-seq* [*simp*]: %<%> : *seq UNIV*
apply (*unfold emptyseq-def seq-def fin-part-func-def pfun-def rel-def zsize-def Pow-def Naturals-def*)
apply *simp*
done

lemma *empty-seq2* [*simp*]: %<%> : *seq G*
apply (*unfold emptyseq-def seq-def fin-part-func-def pfun-def rel-def zsize-def Pow-def Naturals-def*)
apply *simp*
done

lemma *Seq-Subset* [*simp*]: [|*X* <= *Y* ; *s*:*seq X*|] ==> *s*: *seq Y*
apply (*unfold seq-def fin-part-func-def*)
apply (*simp add*: *Partial-Func-Ran-Subset*)
done

lemma *first-elem-notemptyseq* [*simp*]: [|*x*: *seq X* ; *x* ~= %<%>|] ==> (*1*, *x*%^*1*) : *x*
apply (*unfold emptyseq-def seq-def*)
apply (*rule-tac X* = %*N* **and** *Y* = *X*
  in *rel-apply-in-rel*)
apply *simp*
apply *simp*
apply (*rule-tac X* = %*N* %*x X* **in** *numb-range-set-notempty*)
apply *auto*
done

lemma *zsuc-zpred*: *zsuc*(*zpred z*) = *z*
apply (*unfold zsuc-def zpred-def*)
apply *auto*
done

declare *seq-mem-dom-neq-first* [*simp*]

lemma *seq-Collect-zpred-zsuc-Fin-Func*: *x* : *seq X* ==>
  {*p*. *EX n*: *1*..*zpred* (#*x*). *p* = (*n*, *x*%^(*zsuc n*))} : (%*N* −||−> *X*)
apply (*unfold seq-def fin-part-func-def*)
apply *auto*
apply (*unfold pfun-def rel-def*)
apply *auto*
apply (*rule num-range-Naturals-mem*)
prefer *2*
apply *fast*
apply *simp*
apply (*rule-tac X* = %*N* **in**

```
    Rel-Apply-in-Partial-Ran)
apply (unfold pfun-def rel-def)
apply fast
apply (simp (no-asm-simp))
apply (rule-tac X = %N and Y = X in rel-apply-in-rel)
apply (unfold pfun-def rel-def)
apply fast
apply (simp (no-asm-simp))
done


lemma unknown:
!!x. [|x : seq X ; x ~= %<%>|] ==> 0 < #x
apply (unfold seq-def emptyseq-def)
apply auto
done




lemma size-ge-zero [simp]: 0 <= # s
apply (subst znat-bin-0 [symmetric])
apply (unfold zsize-def)
apply (simp (no-asm))
done


lemma size-is-Nat [simp]:  int x : %N
apply (simp add: Naturals-def size-ge-zero)
done


lemma numb-range-size-front [simp]: #(1 .. #s) = #s
apply (rule numb-range-Size-zadd [THEN ssubst])
apply simp
apply (rule-tac t = 1 in zadd-0 [THEN subst])
back
back
apply (subst add-le-cancel-right)
apply (simp (no-asm))
apply (unfold zsuc-def)
apply arith
```

done

insertseq

lemma *Insertseq-Partial-Func* [*simp*]:
[|*y* : *X* ; *t* : (%*N* −|−> *X*) ; ((*dom t*) = (*1*..#*t*))|]
==> *insertseq y t* : (%*N* −|−> *X*)
apply (*unfold insertseq-def zsuc-def*)
apply (*rule Partial-Dom-Insert*, *auto*)
apply(*drule sym*,*simp*)
done

lemma *Insertseq-Dom-Fin-Nat* [*simp*]: !!*t*. *dom t* : *Fin* %*N* ==> *dom* (*insertseq y t*) : %*F* %*N*
apply (*unfold insertseq-def zsuc-def*)
apply *auto*
apply (*rule Nat-Fin-Add-Collect*)
apply (*rule Fin-subset*)
apply *auto*
done

lemma *Insertseq-Fin-Part-Func-Nat*: [|*y* : *X* ; *t* : (%*N* −||−> *X*) ; ((*dom t*) = (*1*..#*t*))|] ==>
insertseq y t : (%*N* −||−> *X*)
apply (*unfold fin-part-func-def*)
apply *auto*
done

declare *size-insert-disjoint* [*simp del*]
declare *size-ge-zero* [*simp del*] *numb-range-size-front* [*simp del*]

lemma *Dom-size-insert-seq* [*simp*]: *t* : *seq X* ==> #(*dom* (*insertseq y t*))= *1* + (#(*dom t*))
apply (*unfold seq-def insertseq-def fin-part-func-def zsuc-def*)
apply *auto*
apply (*rule size-insert-disjoint* [*THEN ssubst*])
apply *auto*
apply (*rule numb-range-Fin-zadd*)
apply (*simp add*: *zsuc-def ring-eq-simps*)+
done

lemma *size-insert-seq*:
!!*t*. [|*t* : *seq X* ; *y*: *X*|] ==> #(*insertseq y t*) = *zsuc* (# *t*)
apply (*unfold seq-def*)
apply (*auto del*: *fin-part-funcE*)
apply (*simplesubst Fin-Part-Func-Dom-Size*)
apply *assumption*
apply (*subst Fin-Part-Func-Dom-Size*)
apply (*rule Insertseq-Fin-Part-Func-Nat*)
apply *assumption+*
apply (*subst Dom-size-insert-seq*)
apply (*simp-all* (*no-asm-simp*) *add*: *seq-def*)
apply (*simp add*: *Ring-and-Field.ring-eq-simps zsuc-simp*)
done
declare *size-insert-seq* [*simp*]

declare *size-ge-zero* [*simp del*] *numb-range-size-front* [*simp del*]

declare *Dom-insertseq* [*simp*]
declare *size-ge-zero* [*simp*] *numb-range-size-front* [*simp*]

lemma *insertseq-seq-pred* [*simp*]: [|*t*:(*seq X*) ; *y*:*X*|] ==> (*insertseq y t*):(*seq X*)
apply (*unfold seq-def*)
apply *simp*
apply (*rule conjI*)
prefer *2*
apply (*rule Dom-insertseq*)
apply (*unfold seq-def*)
apply *auto*
done
declare *insertseq-seq-pred* [*simp*]

lemma *insertseq-not-empty* [*simp*]: *insertseq a b* ~= %<%>
apply (*unfold emptyseq-def insertseq-def*)
apply *auto*
done

lemma *insertseq-not-empty2* [*simp*]: %<%> ~= *insertseq a b*
apply (*unfold emptyseq-def insertseq-def*)
apply *auto*
done

lemma *insert-not-absorb2* [*simp*]: ~(*b* <= *c*) --> *insert a b* ~= *c*
apply *auto*
done

lemma *insertseq-not-absorb* [*simp*]: *b*: *seq X* ==> *insertseq a b* ~= *b*
apply (*unfold insertseq-def seq-def zsuc-def*)
apply (*case-tac b* = {})
apply (*simp* (*no-asm-simp*))
apply (*rule insert-not-absorb2* [*THEN mp*])
apply *simp*

apply (*rule notI*)
apply (*erule-tac c* = (#*b* + *1*,*b*%^#*b*) **in** *subsetCE*)
apply (*rotate-tac* −*1*)
apply (*erule notE*)
apply *simp*

apply (*rule numb-range-mem*)
apply *auto*
apply (*drule pair-rel-dom*)
apply *simp*
done

axioms *insertseq-eq-lemma*: !!*s*. ⟦ *s*: *seq X* ; *t* : *seq Y*⟧ ⟹
{*p*. *EX x*: *dom s* . *p*=(*x*+(*n*::*int*),*s*%^*x*)} =
{*p*. *EX x*: *dom t* . *p*=(*x*+*n*,*t*%^*x*)} -->
{*p*. *EX x*: *dom s* . *p*=(*x*,*s*%^*x*)} = {*p*. *EX x*: *dom t* . *p*=(*x*,*t*%^*x*)}

lemma *insert-eq*: *insert a m* = *insert a n* --> *a* ~:*m* --> *a*~: *n* --> *m* = *n*

apply *auto*
done

lemma *insertseq-eq-seq-help*:
assumes *prem1*: *s*: *seq X*
assumes *prem2*: *t* : *seq Y*
assumes *prem3*: *insertseq a s* = *insertseq a t*
shows (*s*=*t*)
apply (*insert prem1 prem2 prem3*)
apply (*unfold insertseq-def zsuc-def*)
apply (*rotate-tac* −*1*)
apply (*drule insert-eq* [*THEN mp* [*THEN mp* [*THEN mp*]]])
apply (*simp add*: *seq-def*)
apply (*simp add*: *seq-def*)
apply (*drule insertseq-eq-lemma* [*THEN mp*])
apply *fast*
apply (*simp* (*no-asm*))
apply (*insert prem1 prem2 prem3*)
apply (*rule-tac X1* = %*N* **and** *Y1* = *Y* **in** *apply-ident* [*THEN ssubst*])
apply (*unfold seq-def fin-part-func-def*)
apply *simp*
apply (*rule sym*)
apply (*rule-tac X1* = %*N* **and** *Y1* = *X* **in** *apply-ident* [*THEN ssubst*])
apply *simp*+
done

lemma *insertseq-eq-seq*:
[| *s*: *seq X*; *t* : *seq Y*; *insertseq a s* = *insertseq b t*|] ==> (*s*=*t*)
apply (*frule-tac s* = *s* **and** *t* = *t* **and** *a* = *a* **and** *b* = *b* **in** *insertseq-eq*)
apply *fast*
apply *simp*
apply (*rotate-tac* −*1*)
apply *simp*
apply (*drule-tac s* = *s* **and** *t* = *t* **in** *insertseq-eq-seq-help*)
apply *auto*
done

lemma *insertseq-relI*:
!!*s*. [| *s* : *seq A*; *s*′ = *insertseq a s*|] ==> (*s*,*s*′) : *insertseq-rel A*
apply (*unfold insertseq-rel-def*)
apply *auto*
done

lemma *insertseq-relE*: [| (*s*,*s*′) : *insertseq-rel A*; !!*a*. [| *s* : *seq A*; *s*′ = *insertseq a s*|] ==> *R* |] ==> *R*
apply (*unfold insertseq-rel-def*)
apply (*simp* (*no-asm-use*))
apply (*erule conjE*)
apply (*erule exE*)
apply *auto*
done

declare *insertseq-relI* [*intro*!]
declare *insertseq-relE* [*elim*!]

lemma *insertseq-wf-lem*: *insertseq-rel A* <= *measure* (% *x*. *zint*(*zsize*(*dom x*)))
apply (*unfold measure-def inv-image-def*)
apply *auto*

done

lemmas *insertseq-wf* = *insertseq-wf-lem* [*THEN wf-measure* [*THEN wf-subset*]]

lemma *mhead-closed*: !!*s*. [| *s* : *seq A*; *s* ~= %<%> |] ==> *mhead A* %^ *s* : *A*
apply (*frule first-elem-notemptyseq*)
apply (*assumption*)
apply (*simp* (*no-asm-simp*))
apply (*drule seq-ran-subset*)
apply (*erule subsetD*)
apply (*erule RangeI*)
done

lemma *empty2-lem*: {(*x*,*y*). *False*} = {}
apply *auto*
done

lemma *insert-lem*:
{(*xa*,*y*). *xa* : *insert x F* & *y* = *f xa*} =
insert (*x*, *f x*) {(*xa*, *y*). *xa* : *F* & *y* = *f xa*}
apply *auto*
done

lemma *finite-lem*: !! *f*. *finite A* ==> *finite* {(*x*,*y*). *x*:*A* & *y* = *f x*}

apply (*erule finite-induct*)
prefer *2*
apply (*rule insert-lem* [*THEN ssubst*])
apply (*auto simp add*: *empty2-lem*)
done

lemma *zsize-lambda*: !!*f*. *A* : %*F X* ==> (# (*Lambda A f*)) = (# *A*)
apply (*unfold Lambda-def zsize-def*)
apply (*erule Fin-induct*)
prefer *2*
apply (*rule insert-lem* [*THEN ssubst*])
apply (*drule fin-finite*)
apply (*auto simp add*: *empty2-lem finite-lem*)
done

lemma *numb-range-empty*: !! *x*. *y* < *x* ==> *x* .. *y* = {}
apply (*unfold numb-range-def*)
apply *auto*
done

ML ⟪
fun int-case-tac x = *res-inst-tac* [(*z*,*x*)] *int-cases*;
⟫

lemma *numb-range-lemma*: ((*1*::*int*) .. # (*1* .. *n*)) = (*1* .. *n*)
apply (*tactic* ⟪*int-case-tac n 1*⟫)
apply (*simp add*: *zsuc-def*)
apply (*simp add*: *zpred-def*)
done

lemma *lambda-total1*:
(!! *x*. *x*:*A* ==> *f x* : *B*) ==> (*lambda x* : *A*. *f x*) : *A* ---> *B*
apply (*unfold Lambda-def tfun-def pfun-def rel-def*)
apply *auto*
done

lemma *lem2*: !!*n*. *1* <= *n* ==>*int 0* <= *n*
apply (*rule zle-trans*)
apply *auto*
done

lemma *mtail-closed*: !!*s*. [| *s* : *seq A*; *s* ~= %<%> |] ==> *mtail A* %^ *s* : *seq A*
apply (*simp* (*no-asm-simp*))

apply (*erule seqE*)
apply (*rule seqI*)
prefer *2*
apply (*simp add*: *numb-range-Fin-zadd* [*THEN zsize-lambda*] *numb-range-lemma*)
apply (*rule fin-part-funcI*)
prefer *2*
apply *simp*
apply (*rule-tac a = 1 .. (#s − 1)* **in** *pfun-subset*)
apply (*rule total-is-partial*)
apply (*rule lambda-total1*)
apply (*erule fin-part-funcE*)
apply (*erule pfun-apply*)
apply (*auto simp add*: *Naturals-def numb-range-def lem2*)
done

lemma *seqE-lem*:
!! *s*. *s*: *seq A*
  ==> (*EX s′* : *seq A*. *EX a*:*A*. *s* = *insertseq a s′*) | *s* = %<%>
apply (*rule disjCI*)
apply (*rule-tac x = mtail A* %ˆ *s* **in** *bexI*)
apply (*rule-tac x = mhead A* %ˆ *s* **in** *bexI*)
apply (*erule insert-mhead-mtail*)
apply *assumption*
apply (*erule mhead-closed*)
apply *assumption*
apply (*erule mtail-closed*)
apply *assumption*
done

lemma *seq-cases*:
assumes *p1*: *s*:*seq A*
assumes *p2*: *s* = %<%> ==> *P s*
assumes *p3*: !!*a s′*. [| *s′* : *seq A*; *a*:*A*; *s* = (*insertseq a s′*) |] ==> *P s*
shows *P s*
apply (*rule p1* [*THEN seqE-lem*, *THEN disjE*])
apply (*erule-tac* [*2*] *p2*)
apply (*erule bexE*)
apply (*erule bexE*)
apply (*erule p3*)
apply *assumption*+
done

lemma *seq-case-emptyseq*: *seq-case A a f* %<%> = *a*

apply (*unfold seq-case-def*)
apply (*rule some-equality*)
apply *auto*
done

lemma *seq-case-insertseq*:
!!*s*. [| *s*: *seq A*; *x* : *A* |] ==> *seq-case A a f* (*insertseq x s*) = *f x s*

apply (*unfold seq-case-def*)
apply (*rule some-equality*)
apply (*auto dest*: *insertseq-eq-seq insertseq-eq*)
done

declare *seq-case-emptyseq* [*simp*] *seq-case-insertseq* [*simp*]

lemma *seq-rec-unfold-help*:
(%*s*. *seq-rec A c d s*) =
wfrec (*insertseq-rel A*) (%*f*. *seq-case A c* (%*a t*. *d a t* (*f t*)))
apply (*unfold seq-rec-def*)
apply (*simp* (*no-asm*))
done

lemmas *seq-rec-unfold* = *insertseq-wf* [*THEN seq-rec-unfold-help* [*THEN eq-reflection* [*THEN def-wfrec*]]]

lemma *seq-rec-emptyseq*: *seq-rec A c h* %<%> = *c*
apply (*rule seq-rec-unfold* [*THEN trans*])
apply *simp*
done

lemma *seq-rec-insertseq*: !!*s*. [| *s*: *seq A*; *a*:*A*|] ==>
seq-rec A c h (*insertseq a s*) = *h a s* (*seq-rec A c h s*)
apply (*rule seq-rec-unfold* [*THEN trans*])
apply (*simp* (*no-asm-simp*) *add*: *insertseq-relI cut-apply*)
done

lemma *def-seq-rec-emptyseq*:
assumes *rew*: !!*s*. *f s* == *seq-rec A c h s*
shows *f*(%<%>) = *c*
apply (*unfold rew*)
apply (*rule seq-rec-emptyseq*)

done

lemma *def-seq-rec-insertseq*:
assumes *rew*: !!*s*. *f s* == *seq-rec A c h s*
assumes *p1*: *s* : *seq A*
assumes *p2*: *a* : *A*
shows *f*(*insertseq a s*) = *h a s* (*f s*)
apply (*unfold rew*)
apply (*rule p2* [*THEN p1* [*THEN seq-rec-insertseq*]])
done

lemma *seq-induct*:
assumes *p1*: *s*:*seq A*
assumes *p2*: *P* %<%>
assumes *p3*: !! *a s*. [| *s* : *seq A*; *a*:*A*; *P s*|] ==> *P*(*insertseq a s*)
shows *P s*
apply (*rule p1* [*THEN rev-mp*])
apply (*rule insertseq-wf* [*THEN wf-induct*])
apply (*rule impI*)
apply (*erule seq-cases*)
apply (*auto intro*!: *p2 p3 simp add*: *insertseq-rel-def*)
done

lemma *seq-imp-fin*: !! *X*. *list* : *seq X* ==> *list* : %*F* (%*N* %*x X*)
apply (*unfold seq-def*)
apply *simp-all*
done

lemma *card-list-zero-imp-list-empty*:
!! *X*. *list* : *seq X* ==> (# *list* = *0*) = (*list* = %<%>)
apply (*unfold emptyseq-def*)
apply (*rule iffI*)
apply *simp-all*
apply (*rule size-empty2* [*THEN iffD1*])
apply *simp-all*
apply (*rule seq-imp-fin*)
apply (*assumption*)
done

lemma *card-list-zero-imp-list-empty2*: !! *X*. *list* : *seq X* ==> (# *list* <= *0*) = (*list*

= %<%>)
apply (*rule iffI*, *simp-all*)
apply (*rule card-list-zero-imp-list-empty* [*THEN iffD1*], *assumption*)
apply (*drule zle-imp-zless-or-eq*)
apply (*erule disjE*)
apply (*subgoal-tac # list* : *%N*)
apply (*simp only*: *ZInteg.in-naturals*[*symmetric*])
apply (*simp-all add*: *Size-Naturals*)
done

lemma *card-list-zero-imp-list-empty3*: !! *X*. *list* : *seq X* ==> (# *list* < *1*) = (*list* = %<%>)
apply (*rule trans*)
apply (*rule-tac* [*2*] *card-list-zero-imp-list-empty2*)
apply *auto*
done

declare *card-list-zero-imp-list-empty* [*simp*] *card-list-zero-imp-list-empty2* [*simp*]
card-list-zero-imp-list-empty3[*simp*]

declare *card-list-zero-imp-list-empty* [*simp del*] *card-list-zero-imp-list-empty2*[*simp del*]
card-list-zero-imp-list-empty3[*simp del*]

lemma *zsize-image*: !!*f*. [| *inj f*; *x* :*%N* |] ==> #(*f* ‘ (*x*..*y*)) = #(*x*..*y*)

apply (*unfold zsize-def*)
apply (*subst card-image*)
apply (*simp-all*)
apply (*unfold inj-on-def*)
apply *auto*
done

end

6.6 Support for a rudimentary lifter

theory *FunAbs* **imports** *ZMathTool ZFun* **begin**

types

```
('a, 'b) typeabs = ('b set) * ('a => 'b) * ('b => 'a)
```

constdefs

```
TASet :: ('a, 'b) typeabs => ('b set)
TASet == fst

TARep :: ('a, 'b) typeabs => ('a => 'b)
TARep == fst o snd

TAAbs :: ('a, 'b) typeabs => ('b => 'a)
TAAbs == snd o snd

Typeabs :: ('a, 'b) typeabs => bool
Typeabs TA == (let T = TASet TA;
                   R = TARep TA;
                   A = TAAbs TA
               in T ~= {}
                & (ALL (x::'a). (R x) : (T::'b set))
                & (ALL (x::'a). (A (R x)) = x)
                & (ALL (y::'b). y : T --> (R (A y)) = y))



FRep :: [('a, 'b) typeabs, 'a => 'c] => ('b <=> 'c)
FRep TA f == {z. EX x. z = ((TARep TA x), (f x))}

FAbs :: [('a, 'b) typeabs, ('b <=> 'c), 'a] => 'c
FAbs TA g == (% x. g %^ (TARep TA x))

PRep :: ['a => 'b] => ('a <=> 'b)
PRep f == {p. EX x. p = (x, f x)}

PAbs :: [('a <=> 'b), 'a] => 'b
PAbs R == (% x. R%^x)
```

lemma *TASet*: *TASet* (*T*,*R*,*A*) = *T*
apply (*unfold TASet-def*)

apply *auto*
done

lemma *TARep*: *TARep* (*T*,*R*,*A*) = *R*
apply (*unfold TARep-def*)
apply *auto*
done

lemma *TAAbs*: *TAAbs* (*T*,*R*,*A*) = *A*
apply (*unfold TAAbs-def*)
apply *auto*
done

declare *TASet* [*simp*] *TARep* [*simp*] *TAAbs* [*simp*]

lemma *TASet-nonempty*[*rule-format*] : *Typeabs TA* −−> (*TASet TA*) ~= {}
apply (*unfold Typeabs-def Let-def*)
apply *auto*
done

lemma *TARep-in-TASet*[*rule-format*] : *Typeabs TA* −−> (*TARep TA x*) : (*TASet TA*)
apply (*unfold Typeabs-def Let-def*)
apply *auto*
done

lemma *TAAbs-inverse*[*rule-format*] : *Typeabs TA* −−> (*TAAbs TA* (*TARep TA x*)) = *x*
apply (*unfold Typeabs-def Let-def*)
apply *auto*
done

lemma *TARep-inverse*[*rule-format*] : *Typeabs TA* −−> *y* : (*TASet TA*) −−> (*TARep TA* (*TAAbs TA y*)) = *y*
apply (*unfold Typeabs-def Let-def*)
apply *auto*
done

declare *TASet-nonempty* [*simp*] *TARep-in-TASet* [*simp*] *TAAbs-inverse* [*simp*] *TARep-inverse* [*simp*]

lemma *TARep-inj*[*rule-format*] : *Typeabs TA* −−> (*inj* (*TARep TA*))
apply (*unfold Typeabs-def Let-def inj-on-def*)
apply (*rule impI*)

apply (*rule ballI*)+
apply (*erule conjE*)+
apply (*rule impI*)
apply (*erule-tac x = x* **in** *allE*)
apply (*subgoal-tac TAAbs TA* (*TARep TA y*) = *y*)
apply (*erule-tac x = x* **in** *allE*)
apply *simp*
apply *fast*
done

lemma *TARep-iff-inj*[*rule-format*] : *Typeabs TA* −−> ((*TARep TA x*) = (*TARep TA y*)) = (*x* = *y*)
apply (*rule impI*)
apply (*rule iffI*)
apply (*cut-tac TA* = *TA* **in** *TARep-inj*)
apply (*unfold inj-on-def*)
apply *fast*+
done

declare *TARep-iff-inj* [*simp*]

lemma *FRep-apply*[*rule-format*] : *Typeabs TA* −−> ((*FRep TA f*) %ˆ (*TARep TA x*)) = (*f x*)
apply (*unfold FRep-def rel-apply-def*)
apply (*simp* (*no-asm*) *cong add*: *conj-cong*)
done

lemma *FRep-rel*[*rule-format*] : *Typeabs TA* −−> (*FRep TA f*) : ((*TASet TA*) <−−> *UNIV*)
apply (*unfold FRep-def rel-def Typeabs-def Let-def*)
apply *fast*
done

declare *FRep-rel* [*simp*]

lemma *FRep-pfun*[*rule-format*] : *Typeabs TA* −−> (*FRep TA f*) : ((*TASet TA*) −|−> *UNIV*)
apply (*unfold pfun-def*)
apply (*simp* (*no-asm*))
apply (*unfold FRep-def*)
apply *simp*
apply *auto*
done

declare *FRep-pfun* [*simp*]

lemma *FRep-dom*[*rule-format*] : *Typeabs TA* −−> *dom* (*FRep TA f*) = *TASet TA*
apply (*unfold dom-simp* [*THEN eq-reflection*])
apply *auto*
apply (*rule-tac Y* = *UNIV* **and** *f* = *FRep TA f* **in** *rel-dom-member*)
apply (*simp* (*no-asm-simp*))
apply *assumption*
apply (*unfold FRep-def Typeabs-def Let-def*)

apply (*rule eq-sym-conv* [*THEN ext*, *THEN ext*, *THEN ssubst*])
apply (*simp* (*no-asm*))
apply *fast*
done

declare *FRep-dom* [*simp*]

lemma *FRep-total-fun*[*rule-format*] : *Typeabs TA* −−> (*FRep TA f*) : ((*TASet TA*) −−−> *UNIV*)
apply (*unfold tfun-def*)
apply (*simp* (*no-asm*))
done

declare *FRep-total-fun* [*simp*]

lemma *FAbs-inverse*[*rule-format*] : *Typeabs TA* −−> (*FAbs TA* (*FRep TA f*)) = *f*
apply (*unfold FAbs-def FRep-def*)
apply (*simp* (*no-asm*) *add*: *rel-apply-def some-equality*)
done

declare *FAbs-inverse* [*simp*]

lemma *FRep-inverse* [*simp*]: *Typeabs TA* −−> *r* : ((*TASet TA*) −−−> *UNIV*) −−> (*FRep TA* (*FAbs TA r*)) = *r*
apply (*unfold FAbs-def FRep-def*)
apply *auto*
apply (*rule rel-apply-in-rel*)
apply (*rule total-is-partial*)
apply *assumption*
apply (*unfold Typeabs-def Let-def*)
apply (*fast intro*:*total-func-dom*)
apply (*subgoal-tac a*: *TASet TA*)
apply (*rule-tac x* = *TAAbs TA a* **in** *exI*)
apply (*drule total-is-partial*)

apply *simp*
apply (*drule total-is-partial*)
apply (*drule Pfun-Rel*)
apply (*drule rel-pair-dom*)
apply *assumption*+
done

lemma *Typeabs-Fun*:
assumes *prems*: *Typeabs TA*
shows *Typeabs* (*TASet TA* −−−> *UNIV*, *FRep TA*, *FAbs TA*)
apply (*unfold Typeabs-def Let-def*)
apply (*simp* (*no-asm*) *add*: *prems*)
done

lemma *apply-FAb*[*rule-format*] :
Typeabs TA −−> *x* : *TASet TA* −−> (*g* %ˆ *x*) = (*FAbs TA g*) (*TAAbs TA x*)
apply (*unfold FAbs-def*)
apply (*simp* (*no-asm*))
done

lemma *TAAbs-inj-onto*[*rule-format*]:
Typeabs TA −−> *inj-on* (*TAAbs TA*) (*TASet TA*)
apply (*unfold Typeabs-def Let-def*)
apply (*rule impI*)+
apply (*erule conjE*)+
apply (*rule inj-on-inverseI*)
apply (*rule TARep-inverse*)
prefer *2*
apply (*assumption*)
apply (*unfold Typeabs-def Let-def*)
apply *auto*
done

lemma *TAAbs-iff-inj-onto*[*rule-format*]:
Typeabs TA −−> *x* : *TASet TA* −−> *y* : *TASet TA* −−>
((*TAAbs TA x*) = (*TAAbs TA y*)) = (*x* = *y*)
apply (*rule impI*)+
apply (*rule iffI*)
apply (*cut-tac TA* = *TA* **in** *TAAbs-inj-onto*)
apply (*unfold inj-on-def*)
apply *fast*+
done

lemma *PRep-inverse*[*rule-format*]: *R* : (*UNIV* −−−> *UNIV*) −−> ((*PRep* (*PAbs*

R)) = *R*)
apply (*unfold PRep-def PAbs-def*)
apply *auto*
apply (*rule rel-apply-in-rel*)
apply (*rule total-is-partial*)
apply *assumption*
apply (*fast intro*: *total-func-dom*)
apply (*drule total-is-partial*)
apply (*simp* (*no-asm-simp*))
done

lemma *PAbs-inverse*:(*PAbs* (*PRep f*)) = *f*
apply (*unfold PRep-def PAbs-def rel-apply-def*)
apply (*simp* (*no-asm*) *add*: *rel-apply-def some-equality*)
done

lemma *PRep-apply*: (*f x*) = ((*PRep f*) %^ *x*)
apply (*unfold PRep-def rel-apply-def*)
apply (*simp* (*no-asm*) *cong add*: *conj-cong*)
done

lemma *PAbs-apply*: (*R* %^ *x*) = ((*PAbs R*) *x*)
apply (*unfold rel-apply-def PAbs-def*)
apply (*rule refl*)
done
end

6.7 A rudimentery Theory-Morphism from List to Sequences

theory *ZSeqtoList* **imports** *ZSeq FunAbs* **begin**

consts
Rep-seqtype :: *'a list* => (*'a seq*)

primrec
seqOfList-Nil: *Rep-seqtype* [] = %<%>
seqOfList-Cons: *Rep-seqtype* (*x*#*xs*) = *insertseq x* (*Rep-seqtype xs*)

constdefs

```
seqtype      :: ('a seq) set
seqtype      == seq UNIV
Abs-seqtype  :: ('a seq) => 'a list
Abs-seqtype == (% y. • x . Rep-seqtype(x)=y)
ListSeqAbs   :: ('a list, 'a seq) typeabs
ListSeqAbs  == (seqtype, Rep-seqtype, Abs-seqtype)
```

```
concatseq    :: ('a seq * 'a seq) <=> 'a seq
concatseq   == {((s,t),y). y = TARep ListSeqAbs
                    ((TAAbs ListSeqAbs s) • (TAAbs ListSeqAbs t))}

snoc-rel     :: 'a set => ('a seq * 'a seq) set
snoc-rel A  == {(s,s'). s : seq A &(? a:A. s'=concatseq%^(s,%<a%>))}

snoc-case    :: ['a set, 'b, ('a => 'a seq => 'b), 'a seq] => 'b
snoc-case A b f s == • z. (s = %<%> --> z = b) &
                    (! a:A. ! t:seq A. s =concatseq%^(t,%<a%>)-->
                                z = f a t)
```

```
foldrseq     :: [(('a * 'b) set <=> 'b),'b] => ('a seq <=> 'b)
foldrseq f g== UNIV

foldlseq     :: ['b,(('b * 'a) set <=> 'b)] => ('a seq <=> 'b)
foldlseq f g== UNIV

foldseq      :: [(('a * 'a) set <=> 'a)] => ('a seq <=> 'a)
foldseq f   == UNIV
```

```
headseq      :: 'a seq <=> 'a
headseq     == {(s,h). h = (hd(TAAbs ListSeqAbs s))}

lastseq      :: 'a seq <=> 'a
lastseq     == {(s,l). l = (last(TAAbs ListSeqAbs s))}

frontseq     :: 'a seq <=> 'a seq
frontseq    == {(s,f). f = TARep ListSeqAbs(butlast(TAAbs ListSeqAbs s))}
```

```
tailseq      :: 'a seq <=> 'a seq
tailseq     == {(s,t). t = TARep ListSeqAbs(tl(TAAbs ListSeqAbs s))}

revseqseq    :: 'a seq <=> 'a seq
revseqseq   == {(s,r). r = TARep ListSeqAbs (rev (TAAbs ListSeqAbs s))}

filtering    :: ('a seq * 'a set) <=> 'a seq
filtering   == {((s,V),F). F = TARep ListSeqAbs (filter
                                        (% a. a : V) (TAAbs ListSeqAbs s))}

extraction   ::(int set * 'a seq) <=> 'a seq
extraction  == UNIV

prefixseq    :: 'a seq <=> 'a seq
prefixseq   == {(s,t). ? v:seq UNIV. (concatseq%^(s,v)) = t}
suffixseq    :: 'a seq <=> 'a seq
suffixseq   == {(s,t). ? u:seq UNIV. (concatseq%^(u,s)) = t}
inseq        :: 'a seq <=> 'a seq
inseq       == {(s,t). ? v:seq UNIV. (concatseq%^(v,s)) = t}
```

syntax

```
*concatseq ::['a seq, 'a seq] => 'a seq     ((2 - %&^/ -) [70,71] 70)
*headseq   ::('a seq) => 'a                 ((1 head -) 70)
*lastseq   ::('a seq) => 'a                 ((1 last -) 70)
*frontseq  ::('a seq) => ('a seq)           ((1 front -) 70)
*tailseq   ::('a seq) => ('a seq)           ((1 tail -) 70)
*filtering ::['a seq, 'a set] => 'a seq     ((2 - %|'/ -) [70,71] 70)
*revseqseq ::('a seq) => ('a seq)           ((1 revseq -) 70)
*prefix    ::['a seq, 'a set] => 'a seq     ((2 - prefix/ -) [70,71] 70)
*suffix    ::['a seq, 'a set] => 'a seq     ((2 - suffix/ -) [70,71] 70)
*in        ::['a seq, 'a set] => 'a seq     ((2 - in/ -) [70,71] 70)
```

translations

```
s %&^ t     == concatseq%^(s,t)
head s      == headseq%^s
last s      == lastseq%^s
front s     == frontseq%^s
tail s      == tailseq%^s
revseq s    == revseqseq%^s
```

s %|‘ *m* == *filtering*%^(*s*,*m*)
s prefix t == (*s*,*t*):*prefixseq*
s suffix t == (*s*,*t*):*suffixseq*
s in t == (*s*,*t*):*inseq*

axioms
snoc-neq-empty: *t* %&^ %< *aa* %> ~= %<%>
snoc-inj1: [| *t*:*seq A*; *t′*:*seq A*; *aa*: *A*; *aa′*:*A* |] ==>
t %&^ %< *aa* %> = *t′* %&^ %< *aa′* %> ==> *t* = *t′*
snoc-inj2: [| *t*:*seq A*; *t′*:*seq A*; *aa*: *A*; *aa′*:*A* |] ==>
t %&^ %< *aa* %> = *t′* %&^ %< *aa′* %> ==> *aa* = *aa′*

rev-insert-eq-snoc:
[| *s*:*seq A*; *a*:*A* |] ==>
revseq (*insertseq a s*) = *s* %&^ %< *a* %>

seq-card-dom: *s*:*seq A* ==> #(*dom s*) = # *s*

mem-concatseq: *s* : *seq X* −−> *t* : *seq Y* −−> *x* : (*ran*(*s* %&^ *t*)) = (*x*: (*ran s*) | *x* : (*ran t*))
mem-filtering: *s* : *seq X* −−> *x* : (*ran*(*s* %|‘ *V*)) = (*x* : (*ran s*) & (*x* : *V*))
snoc-wf-lem: *snoc-rel A* <= *measure* (% *x*. *zint*(*zsize*(*dom x*)))

lemma *Rep-seqtype* [*simp*]: *Rep-seqtype*(*x*): *seqtype*
apply (*unfold seqtype-def*)
apply (*induct-tac x*)
apply (*simp*)
apply *auto*
done

lemma *insertseq-not-absorb-seqtype* [*simp*]: *s* : *seqtype* ==> (*insertseq a s* ~= *s*)
apply (*unfold seqtype-def*)
apply *auto*
done

lemma *Rep-seqtype-inj* [*simp*]: *inj Rep-seqtype*

apply (*unfold inj-on-def*)
apply (*rule ballI*)
apply (*induct-tac x*)
apply (*rule ballI*)
apply (*induct-tac y*)
apply (*simp* (*asm-lr*))
apply (*simp* (*asm-lr*))
apply (*rule ballI*)
apply (*induct-tac y*)
apply (*simp* (*asm-lr*))
apply (*erule-tac x = lista* **in** *ballE*)
apply (*simp* (*asm-lr*))
apply (*rule impI*)
apply (*rule conjI*)
apply (*rule-tac s = Rep-seqtype list* **and** *X = UNIV::'a set* **and** *t =* (*Rep-seqtype lista*) **and** *Y = UNIV::'a set* **in** *insertseq-eq*)
apply (*fold seqtype-def*)
apply (*simp* (*asm-lr*))
apply (*simp* (*asm-lr*))
apply (*simp* (*asm-lr*))
apply (*rotate-tac 1*)
apply (*erule impE*)
prefer *2*
apply (*simp* (*asm-lr*))
apply (*rule-tac a = a* **and** *X = UNIV::'a set* **and** *b = aa* **and** *Y = UNIV::'a set* **in** *insertseq-eq-seq*)
apply (*fold seqtype-def*)
apply (*simp* (*asm-lr*))
apply (*simp* (*asm-lr*))
apply (*simp* (*asm-lr*))
apply (*simp* (*asm-lr*))
done

lemma *Inv-f-f*: *inj*(*f*) ==> (• *xa*. *f xa* = *f x*) = *x*
apply (*rule injD*)
apply *assumption*
apply (*rule someI*)
apply (*rule refl*)
done

lemma *Rep-seqtype-inverse* [*simp*]: *Abs-seqtype*(*Rep-seqtype*(*x*)) = *x*
apply (*unfold Abs-seqtype-def*)
apply (*rule Inv-f-f*)

apply (*simp* (*asm-lr*))
done

lemma *List-of-seq-emptyseq*: ((*Abs-seqtype* %<%>) = [])
apply (*unfold Abs-seqtype-def*)
apply (*rule some-equality*)
apply *auto*
apply (*rule mp*)
prefer *2*
apply (*assumption*)
apply (*induct-tac x*)
apply *auto*
done

lemma *seq-Collect-Fin-Func*:
!!*x*. [|*x* : (*Naturals* −||−> *UNIV*) ;
(*ka* + *1*) = (*card x*) ; ((*dom x*) = (*1*.. *zsuc* (*int ka*)))|] ==>
{*p*. *EX n*: *1*..(*int ka*). *p* = (*n*, *x*% ^*n* + *1*)} : (*Naturals* −||−> *UNIV*)
apply (*unfold fin-part-func-def*)
apply *auto*
apply (*unfold pfun-def rel-def*)
apply *auto*
apply (*rule num-range-Naturals-mem*)
apply *auto*
done

lemma *seq-Collect-Size*: [|*x* : (*Naturals* −||−> *UNIV*) ;
((*Suc ka*) = (*card x*)) ; ((*dom x*) = (*1*.. *zsuc* (*int ka*)))|] ==>
(*ka* = *card* {*p*. *EX n*: *1*..*int ka*. *p* = (*n*, *x*% ^*n* + *1*::*int*)})
apply (*simp add*: *card-size-eq*)
apply (*rule-tac X1* = *Naturals* **and** *Y1* = *UNIV* **in** *Fin-Part-Func-Dom-Size* [*THEN ssubst*])
apply (*erule seq-Collect-Fin-Func*)
apply (*simp add*: *card-size-eq*)
apply (*unfold zsuc-def*)
apply *simp*
apply *simp*
apply (*case-tac 1*<=(*int ka*))
apply (*unfold zsuc-def*)
apply *simp*
apply *simp*
done

lemma *inj-Abs-seqtype2*:
  !!*x*. *x* : *seqtype* ==> (*EX y*. *x* = *Rep-seqtype y*)
apply (*unfold seqtype-def*)
apply (*erule seq-induct*)
apply (*rule-tac x* = [] **in** *exI*)
apply (*erule-tac* [*2*] *exE*)
apply (*rule-tac* [*2*] *x* = *op*# *a y* **in** *exI*)
apply *auto*
done

lemma *f-Inv-f*: *y* : *range*(*f*) ==> *f* (• *x*. *f x* = *y*) = *y*
apply (*rule rangeE*)
apply *assumption*
apply (*rule someI*)
apply (*erule sym*)
done

lemma *Abs-seqtype-inverse* [*simp*]:*y* : *seqtype* ==> (*Rep-seqtype* (*Abs-seqtype y*)) = *y*
apply (*unfold Abs-seqtype-def*)
apply (*rule f-Inv-f*)
apply (*unfold image-def Bex-def*)
apply (*simp* (*no-asm*))
apply (*erule inj-Abs-seqtype2*)
done

lemma *seqtype-notempty* [*simp*]: *seqtype* ~= {}
apply (*unfold seqtype-def*)
apply (*rule x-in-S-implies-S-nonempty*)
apply (*rule empty-seq*)
done

lemma *ListSeqAbsIsTypeabs* [*simp*]: *Typeabs ListSeqAbs*
apply (*unfold Typeabs-def Let-def ListSeqAbs-def*)
apply *auto*
done

lemma *Rep-seqtype-lemma*: *Rep-seqtype* = *TARep ListSeqAbs*
apply (*unfold TARep-def ListSeqAbs-def*)
apply (*simp* (*no-asm*))
done

lemma *Abs-seqtype-lemma*: *Abs-seqtype* = *TAAbs ListSeqAbs*
apply (*unfold TARep-def ListSeqAbs-def*)
apply (*simp* (*no-asm*))
done

lemma *seqtype-lemma*: *seqtype* = *TASet ListSeqAbs*
apply (*unfold TARep-def ListSeqAbs-def*)
apply (*simp* (*no-asm*))
done

lemma *TARep-LSAbs-Nil*: *TARep ListSeqAbs* [] = %<%>
apply (*simp* (*no-asm*) *add*: *Rep-seqtype-lemma* [*symmetric*])
done

lemma *TARep-LSAbs-Cons*: *TARep ListSeqAbs* (*x*#*xs*) = *insertseq x* (*TARep List-SeqAbs xs*)
apply (*simp* (*no-asm*) *add*: *Rep-seqtype-lemma* [*symmetric*])
done

lemma *TAAbs-LSAbs-emptyseq*: (*TAAbs ListSeqAbs* %<%>) = []
apply (*rule-tac TA1* = *ListSeqAbs* **in** *TARep-iff-inj* [*THEN subst*])
apply (*simp* (*no-asm*))
apply (*subst TARep-inverse*)
apply (*simp* (*no-asm*))
apply (*simp* (*no-asm*) *add*: *TARep-LSAbs-Nil* [*symmetric*])
apply (*simp* (*no-asm*) *add*: *TARep-LSAbs-Nil* [*symmetric*])
done

lemma *seqtype-emptyseq* [*simp*]: %<%> : *TASet ListSeqAbs*
apply (*simp* (*no-asm*) *add*: *seqtype-lemma* [*symmetric*] *seqtype-def*)
done

lemma *seq-seqtype*: *s*: *seq X* ==> *s*: *seqtype*
apply (*unfold seqtype-def*)
apply (*simp* (*no-asm-simp*))
done

lemma *dom-insertseq*: (*dom* (*insertseq a s*)) =

```
                    (insert 1 {d . EX x: dom s. d = (x+1)})
apply (unfold insertseq-def zsuc-def)
apply auto
done

ML
⟪
fun au i = by (SELECT-GOAL (auto-tac (claset(), simpset()) )i)
⟫

lemma insert-ran [simp]: ran (insert (a, b) R) = insert b (ran R)
apply auto
done


lemma ran-lem:
!!s. s: seq A ==>
  Range {p. ? n:Domain s. p = (zsuc n, s %^ n)} = Range s
apply (unfold seq-def)
apply auto
apply (rule-tac a = n in RangeI)
apply (rule-tac [2] a = zsuc xa in RangeI)
apply (drule-tac [2] sym)
apply auto
done

lemma ran-insertseq [simp]:
!!s. s:seq X ==> ran (insertseq a s) = (insert a (ran s))
apply (unfold insertseq-def)
apply (simp add: ran-lem)
done

lemma ran-insertseq-TAAbs [simp]: s: TASet ListSeqAbs ==> ran (insertseq a s)
= (insert a (ran s))
apply (rule mp)
prefer 2
apply (assumption)
apply (simp (no-asm) add: seqtype-lemma [symmetric] seqtype-def)
done
```

lemma *concatseq-TypeAbs*: (*concatseq* % ^ (*s*,*t*)) =
TARep ListSeqAbs ((*TAAbs ListSeqAbs s*) • (*TAAbs ListSeqAbs t*))
apply (*unfold concatseq-def*)
apply (*subst apply-single-tuple*)
apply (*rule refl*)
done

lemma *revseqseq-TypeAbs*: (*revseq s*) = (*TARep ListSeqAbs* (*rev* (*TAAbs ListSeqAbs s*)))
apply (*unfold revseqseq-def*)
apply (*subst apply-single*)
apply (*rule refl*)
done

lemma *last-TypeAbs*: (*last s*) = (*List.last* (*TAAbs ListSeqAbs s*))
apply (*unfold lastseq-def*)
apply (*subst apply-single*)
apply (*rule refl*)
done

lemma *head-TypeAbs*: (*head s*) = (*hd*(*TAAbs ListSeqAbs s*))
apply (*unfold headseq-def*)
apply (*subst apply-single*)
apply (*rule refl*)
done

lemma *front-TypeAbs*: (*front s*) = (*TARep ListSeqAbs* (*butlast*(*TAAbs ListSeqAbs s*)))
apply (*unfold frontseq-def*)
apply (*subst apply-single*)
apply (*rule refl*)
done

lemma *tail-TypeAbs*: (*tail s*) = (*TARep ListSeqAbs* (*tl*(*TAAbs ListSeqAbs s*)))
apply (*unfold tailseq-def*)
apply (*subst apply-single*)
apply (*rule refl*)
done

lemma *filtering-TypeAbs*: (*filtering*% ^(*s*, *V*)) = *TARep ListSeqAbs* (*filter* (% *a*. *a* : *V*) (*TAAbs ListSeqAbs s*))

apply (*unfold filtering-def*)
apply (*subst apply-single-tuple*)
apply (*rule refl*)
done

lemma *sizeseq-TypeAbs-seq*: *s*:*seq X* ==> (# *s*) = *int* (*length* (*TAAbs ListSeqAbs s*))
apply (*drule seq-seqtype*)
apply (*subgoal-tac ALL l. l* = (*TAAbs ListSeqAbs s*) --> (# *s*) = *int* (*length l*))
apply *simp*
apply (*rule allI*)
apply (*rule mp*)
prefer *2*
apply (*assumption*)
apply (*rule-tac x* = *s* **in** *spec*)
apply (*induct-tac l*)
apply (*rule allI*)
apply (*rule impI*)
apply (*simp add*: *TAAbs-LSAbs-emptyseq* [*THEN sym*] *TARep-LSAbs-Nil TAAbs-iff-inj-onto seqtype-lemma*)
apply (*rule allI*)
apply (*rule impI*)
apply (*rule-tac x1* = *a* # *list* **in** *TARep-iff-inj* [*THEN subst*])
apply (*rule ListSeqAbsIsTypeabs*)
apply (*subst TARep-inverse*)
apply (*simp* (*no-asm*))
apply (*simp add*: *seqtype-lemma*)
apply (*simp add*: *TARep-LSAbs-Cons*)
apply (*rule impI*)
apply (*rotate-tac* −*1*)
apply (*drule sym*)
apply (*simp add*: *seqtype-lemma*)
apply (*rule-tac X1* = *UNIV* **in** *size-insert-seq* [*THEN ssubst*])
apply (*fold seqtype-def*)
apply (*simp add*: *seqtype-lemma*)
apply (*simp* (*no-asm*))
apply (*erule-tac x* = *TARep ListSeqAbs list* **in** *allE*)
apply *auto*
apply (*rotate-tac 2*)
apply (*drule sym*)
apply *auto*
apply (*rotate-tac 2*)
apply (*unfold zsuc-def*)
apply (*simp add*: *ring-eq-simps*)

done

lemma *sizeseq-TypeAbs*: !!*s*. *s*: *TASet ListSeqAbs* ==> (# *s*) = *int* (*length* (*TAAbs ListSeqAbs s*))
apply (*simp add*: *seqtype-lemma* [*symmetric*] *seqtype-def*)
apply (*subst sizeseq-TypeAbs-seq*)
apply *auto*
done

lemma *insertseq-TypeAbs*: !!*s*. *t*:*TASet ListSeqAbs* ==> *insertseq x t* = *TARep ListSeqAbs* (*x* # (*TAAbs ListSeqAbs t*))
apply (*subst TARep-LSAbs-Cons*)
apply (*simp* (*no-asm-simp*))
done

lemma *ran-mem-TypeAbs-seq*: *s*:*seq X* ==> *a* : (*ran s*) = (*a mem* (*TAAbs ListSeqAbs s*))
apply (*drule seq-seqtype*)
apply (*subgoal-tac ALL l*. *l* = (*TAAbs ListSeqAbs s*) --> *a* : (*ran s*) = (*a mem l*)
)
apply *simp*
apply (*rule allI*)
apply (*rule mp*)
prefer *2*
apply (*assumption*)
apply (*rule-tac x* = *s* **in** *spec*)
apply (*induct-tac l*)
apply (*rule allI*)
apply (*rule impI*)
apply (*rule impI*)
apply (*simp* (*no-asm*))
apply (*simp add*: *TAAbs-LSAbs-emptyseq* [*symmetric*] *TARep-LSAbs-Nil TAAbs-iff-inj-onto seqtype-lemma*)
apply (*rotate-tac* −*1*)
apply (*rule allI*)
apply (*rule impI*)
apply (*rule-tac x1* = *aa* # *list* **in** *TARep-iff-inj* [*THEN subst*])
apply (*rule ListSeqAbsIsTypeabs*)
apply (*subst TARep-inverse*)
apply (*simp add*: *seqtype-lemma*)
apply (*simp add*: *seqtype-lemma*)
apply (*rule impI*)
apply (*rotate-tac* −*1*)

apply (*drule sym*)
apply (*simp* (*no-asm-simp*))
apply (*simp add*: *TARep-LSAbs-Cons*)
apply (*erule-tac x* = *TARep ListSeqAbs list* **in** *allE*)
apply (*case-tac aa* = *a*)
apply (*simp* (*no-asm-simp*))
apply *auto*
apply (*simp add*: *seqtype-lemma*)
apply (*simp add*: *seqtype-lemma*)
done

lemma *ran-mem-TypeAbs*:*s*: *TASet ListSeqAbs* ==> *a* : (*ran s*) = (*a mem* (*TAAbs ListSeqAbs s*))
apply (*simp add*: *seqtype-lemma* [*symmetric*] *seqtype-def*)
apply (*erule ran-mem-TypeAbs-seq*)
done

ML
⟪
(∗*from contrib*∗)
val concatseq-TypeAbs = *thm concatseq-TypeAbs*
val revseqseq-TypeAbs = *thm revseqseq-TypeAbs*
val sizeseq-TypeAbs = *thm sizeseq-TypeAbs*
val TARep-in-TASet = *thm TARep-in-TASet*

val holempty-ss = *HOL-basic-ss*

val TypeAbs-Op-ss = *holempty-ss addsimps* [*concatseq-TypeAbs*,
revseqseq-TypeAbs,*sizeseq-TypeAbs*, *thm ListSeqAbsIsTypeabs*,
TARep-in-TASet,*thm last-TypeAbs*,*thm head-TypeAbs*,
thm front-TypeAbs, *thm tail-TypeAbs*,*thm filtering-TypeAbs*,
thm insertseq-TypeAbs, *thm ran-mem-TypeAbs*]

val TypeAbs-Const-ss = *holempty-ss addsimps* [*thm TARep-LSAbs-Nil RS sym*]

val TypeAbs-norm-ss = *holempty-ss addsimps* [*thm seqtype-lemma*]
⟫

ML

```
⟪
fun seq-type t =
    let val typ = (snd t)
    fun typ-test (Type(set,[(Type(∗,[(Type(int,[])),-]))])) = true
     | typ-test (-) = false
    in
        typ-test (typ)
    end
 ⟫
```

ML

```
⟪
fun rep-abs-op t =
    let val operator = (fst t)
    fun op-test ((TAAbs,-),(ListSeqAbs,-)) = false
      | op-test (-) = true
    in
        (op-test(operator))
    end
⟫
```

ML

```
⟪
val TARep-inverse = thm TARep-inverse
val ListSeqAbsIsTypeabs = thm ListSeqAbsIsTypeabs
val TARep-iff-inj = thm TARep-iff-inj
val seq-seqtype = thm seq-seqtype


fun find-var-tac t =
  let fun const-var (Abs -) = []
        | const-var ((Const a)$(Const b)$(Free f)) = [((a,b),f)]
        | const-var (Const -) = []
        | const-var (Var -) = []
        | const-var (Bound -) = []
        | const-var (Free -) = []
        | const-var (t1$t2) = (const-var t1) union (const-var t2)
      fun single-var (Abs -) = []
        | single-var ((Const a)$(Const b)$(Free f)) = []
        | single-var (Const -) = []
        | single-var (Var -) = []
        | single-var (Bound -) = []
        | single-var (Free f) = [f]
        | single-var (t1$t2) = (single-var t1) union (single-var t2)
```

```
    val v = (const-var t)
    val f = (single-var t)
    val f-seq = (filter seq-type f)
    val v-abs = (map snd (filter rep-abs-op v))
  in
    f-seq union v-abs
  end

fun make-tac i t =
        (TRY (EVERY [(res-inst-tac [(t,fst t)]
                    (TARep-inverse RS sym RS ssubst)i),
                    (rtac ListSeqAbsIsTypeabs i),
                    (Asm-full-simp-tac i)]))

⟫
```

ML

```
⟪
fun STATE tacfun st = tacfun st st;

fun getgoal-new top i =
    (case  Library.drop (i−1, prems-of top) of
        [] => errorgetgoal: Goal number out of range
      | Q::- => Q);

fun goal-params-new top i =
  let val gi = getgoal-new top i
    val frees = map Free (Logic.strip-params gi)
  in (gi,frees) end;

fun concl-of-goal-new top i =
  let val (gi,frees) = goal-params-new top i
    val B = Logic.strip-assums-concl gi
  in subst-bounds(frees,B) end;
⟫
```

ML

```
⟪
fun var-replace-tac i = (STATE (fn top =>
        let val list = (find-var-tac (concl-of-goal-new (topthm()) i))
            val tactic-list = (map (make-tac i) list)
        in
        if (list = []) then no-tac
```

```
        else
           EVERY tactic-list
        end))
⟫
```

ML

```
⟪
fun TARep-inj-tac i = EVERY [(stac TARep-iff-inj i),
                             (rtac ListSeqAbsIsTypeabs i)]

fun seq-to-list-tac  i = (EVERY [(asm-full-simp-tac TypeAbs-Op-ss i),
                                 (asm-full-simp-tac TypeAbs-Const-ss i),
                                 TRY (var-replace-tac i),
                                 TRY (TARep-inj-tac i)])

fun seq-norm i = EVERY [ (rtac impI i),
                         (rotate-tac ~1 1),
                         (dmatch-tac [seq-seqtype] i)]


fun seq-to-list-auto-tac i =
                EVERY [REPEAT (seq-norm i),
                       (asm-full-simp-tac TypeAbs-norm-ss i),
                       (seq-to-list-tac i),
                       (Simp-tac i)]


⟫
```

lemma *concatseq-assoc*: s %&^ t %&^ u = s %&^ (t %&^ u)
apply (*tactic* ⟪*seq-to-list-auto-tac 1*⟫)
apply *auto*
done

lemma *concatseq-emptyseq*[*rule-format*] : s :*seq* X −−> %<%> %&^ s = s
apply (*tactic* ⟪*seq-to-list-auto-tac 1*⟫)
apply *auto*

done

lemma *concatseq-emptyseq2*[*rule-format*]: *s* :*seq X* −−> *s* %&ˆ %<%> = *s*
apply (*tactic* ⟪*seq-to-list-auto-tac 1*⟫)
apply *auto*
done

lemma *concatseq-is-empty*[*rule-format*]:
s : *seq X* −−> *t* : *seq Y* −−>
(*s* %&ˆ *t* = %<%>) = (*s* = %<%> & *t* = %<%>)
apply (*tactic* ⟪*seq-to-list-auto-tac 1*⟫)
apply (*rule iffI*)
apply (*erule conjE*)
apply (*rule-tac t* = [] **in** *subst*)
back
apply *assumption*
back
apply (*rule-tac t* = [] **in** *subst*)
apply *assumption*
apply (*subst TARep-inverse*)
prefer *3*
apply (*subst TARep-inverse*)
apply *simp-all*
done

lemma *same-concatseq-eq*[*rule-format*] :
s : *seq X* −−> *t* : *seq Y* −−> *u*: *seq Z* −−> (*s* %&ˆ *t* = *s* %&ˆ *u*) = (*t* = *u*)
apply (*tactic* ⟪*seq-to-list-auto-tac 1*⟫)
apply (*rule iffI*)
apply (*tactic* ⟪*asm-full-simp-tac* (*HOL-ss addsimps* [*thm Abs-seqtype-lemma RS sym*, *thm Abs-seqtype-def*]) *1*⟫)
apply (*subgoal-tac Rep-seqtype* (*Abs-seqtype u*) = *u*)
apply (*tactic* ⟪*asm-full-simp-tac* (*HOL-ss addsimps* [*thm Abs-seqtype-lemma RS sym*, *thm Abs-seqtype-def*]) *1*⟫)
apply (*subgoal-tac Rep-seqtype* (*Abs-seqtype t*) = *t*)
apply (*tactic* ⟪*asm-full-simp-tac* (*HOL-ss addsimps* [*thm Abs-seqtype-lemma RS sym*, *thm Abs-seqtype-def*]) *1*⟫)
apply (*rule Abs-seqtype-inverse*)
apply (*subst seqtype-lemma*)
apply *assumption*
apply (*rule Abs-seqtype-inverse*)
apply (*subst seqtype-lemma*)
apply *assumption*

apply *simp*
done

declare *concatseq-emptyseq* [*simp*] *concatseq-emptyseq2* [*simp*] *concatseq-is-empty* [*simp*]

lemma *revseq-revseq-ident*[*rule-format*] : *s* : *seq X* −−> (*revseq* (*revseq s*)) = *s*
apply (*tactic* ⟪*seq-to-list-auto-tac 1*⟫)
apply *auto*
done

lemma *revseq-concatseq*[*rule-format*]: *s*:*seq X* −−> *t*:*seq Y* −−>
(*revseq* (*s* %&ˆ *t*)) = ((*revseq t*) %&ˆ (*revseq s*))
apply (*tactic* ⟪*seq-to-list-auto-tac 1*⟫)
apply *auto*
done

lemma *size-revseq*[*rule-format*]: *s* : *seq X* −−> #(*revseq s*) = #*s*
apply (*tactic* ⟪*seq-to-list-auto-tac 1*⟫)
apply *auto*
done

lemma *last-cons*[*rule-format*] :
t : *seq X* −−>
(*last* (*insertseq x* (*insertseq y t*))) = (*last* (*insertseq y t*))
apply (*tactic* ⟪*seq-to-list-auto-tac 1*⟫)
apply *auto*

done

lemma *last-concatseq*[*rule-format*] :
t : *seq X* −−> *s* : *seq Y* −−>
(*last* (*s* %&ˆ (*insertseq y t*))) = (*last* (*insertseq y t*))
apply (*tactic* ⟪*seq-to-list-auto-tac 1*⟫)
apply *auto*
done

lemma *laSt-single*: (*last* (*insertseq x* %<%>)) = *x*
apply (*tactic* ⟪*seq-to-list-auto-tac 1*⟫)
apply (*tactic* ⟪*seq-to-list-auto-tac 1*⟫)
apply *auto*
done

lemma *front-mt*: *front*(%<%>) = %<%>
apply (*tactic* ⟪*seq-to-list-auto-tac 1*⟫)
apply *auto*
done

lemma *aux*:
(*x* : *seq Z* & *y* : *Z*) −−> *front*(*x* %&ˆ %< *y* %>) = *x*
apply (*tactic* ⟪*seq-to-list-auto-tac 1*⟫)
apply (*subst insertseq-TypeAbs*)
apply (*tactic* ⟪*seq-to-list-auto-tac 1*⟫)
apply (*rule impI*)
apply (*simp add*: *TARep-inverse*)
apply (*simp* (*no-asm-simp*) *add*: *ListSeqAbs-def seqtype-def*)
apply (*rule Abs-seqtype-inverse*)
apply (*unfold seqtype-def*)
apply *auto*
done

lemma *front-snoc*:
!!*x* . [| *x* : *seq Z*; *y* : *Z* |] ==> *front*(*x* %&ˆ %< *y* %>) = *x*
apply (*cut-tac aux*)
apply *auto*

done

declare *front-mt* [*simp*] *front-snoc* [*simp*]

lemma *TAAbs-LSA*: *TAAbs ListSeqAbs* = (%*u*. • *x*. (*TARep ListSeqAbs*) *x* = *u*)
apply (*subst Rep-seqtype-lemma* [*symmetric*])
apply (*subst Abs-seqtype-lemma* [*symmetric*])
apply (*unfold Abs-seqtype-def*)
apply (*rule refl*)
done

lemma *TASet-LSA*: *TASet ListSeqAbs* = *seq UNIV*
apply (*subst seqtype-lemma* [*symmetric*])
apply (*unfold seqtype-def*)
apply (*rule refl*)
done

lemma *TARep-in-seqUNIV*: *TARep ListSeqAbs l* : *seq UNIV*
apply (*subst TASet-LSA* [*symmetric*])
apply (*rule TARep-in-TASet*)
apply *auto*
done

lemma *TAAbs-LSA-cons*: (*x*#*xs*) = *TAAbs ListSeqAbs* (*insertseq x* (*TARep ListSeqAbs xs*))
apply (*rule-tac x1* = *x* # *xs* **in** *TARep-iff-inj* [*THEN subst*])

apply (*rule ListSeqAbsIsTypeabs*)
apply (*subst TARep-inverse*)
apply (*rule ListSeqAbsIsTypeabs*)
apply (*subst TASet-LSA*)
apply (*rule insertseq-seq-pred*)
apply (*subst TASet-LSA* [*symmetric*])
apply (*simp* (*no-asm*))
apply (*simp* (*no-asm*))
apply (*subst TARep-LSAbs-Cons*)
apply (*rule refl*)
done

lemma *insertseq-concat*: *a*:*X* −−> *s* :*seq X* −−> *t* :*seq X* −−>
(*insertseq a s*) %&ˆ *t* = *insertseq a* (*s* %&ˆ *t*)
apply (*tactic* ⟪*seq-to-list-auto-tac 1*⟫)
apply *auto*
apply (*subst insertseq-TypeAbs*)
apply *auto*
apply (*subst TASet-LSA*)
apply *auto*
done
declare *insertseq-concat* [*simp*]

lemma *concatseq-closed*:
!!*a*. [|*a* : *seq G*; *b* : *seq G*|] ==> (*a* %&ˆ *b*) : *seq G*
apply (*erule-tac s* = *a* **in** *seq-induct*)
apply *simp*
apply *auto*
done
declare *concatseq-closed* [*simp*]

lemma *snoc-closed*:
!!*a*. [|*s* : *seq G*; *b* : *G*|] ==> (*s* %&ˆ %<*b*%>) : *seq G*
apply (*rule concatseq-closed*)
apply *auto*
done
declare *snoc-closed* [*simp*]

declare *seq-card-dom* [*simp*]

lemma *aux*: (*int m* < *int n*) = (*m* < *n*)
apply (*simp add*: *ring-eq-simps*)
done

lemmas *snoc-wf* = *snoc-wf-lem* [*THEN wf-measure* [*THEN wf-subset*]]

lemma *snocE-lem*:
!! *s*. *s*: *seq A*
==> (? *t* : *seq A*. ? *a*:*A*. *s* = *t* %& ^ %<*a*%>) | *s* = %<%>
apply (*erule-tac s* = *s* **in** *seq-induct*)
apply *auto*
apply (*rule-tac x* = *insertseq a t* **in** *bexI*)
apply (*rule-tac x* = *aa* **in** *bexI*)
apply (*rule-tac* [*4*] *x* = %<%> **in** *bexI*)
apply (*rule-tac* [*4*] *x* = *a* **in** *bexI*)
prefer *4*
apply (*subst concatseq-emptyseq*)
apply (*rule insertseq-seq-pred*)
apply *auto*
done

lemma *seqsnoc-cases*:
assumes *p1*: *s*:*seq A*
assumes *p2*: *s* = %<%> ==> *P s*
assumes *p3*: !!*a s'*. ⟦ *s'* : *seq A*; *a*:*A*; *s* = *s'* %& ^ %< *a* %> ⟧ ⟹ *P s*
shows *P s*
apply (*rule p1* [*THEN snocE-lem* [*THEN disjE*]])
prefer *2*
apply (*erule p2*)
apply (*erule bexE*)
apply (*erule bexE*)
apply (*erule p3*)
apply *assumption*+
done

lemma *seq-snocinduct*:
assumes *p1*: *s*:*seq A*
assumes *p2*: *P* %<%>
assumes *p3*: !! *a s*. [| *s* : *seq A*; *a*:*A*; *P s*|] ==> *P*(*s* %&ˆ %< *a* %>)
shows *P s*
apply (*rule p1* [*THEN rev-mp*])
apply (*rule snoc-wf* [*THEN wf-induct*])
apply (*rule impI*)
apply (*erule seqsnoc-cases*)
apply (*auto intro*!: *p2 p3 simp add*: *snoc-rel-def*)
done

declare *snoc-neq-empty* [*simp*] *snoc-neq-empty* [*THEN not-sym*, *simp*]

lemma *snoc-case-mt*: *snoc-case A a f* %<%> = *a*
apply (*unfold snoc-case-def*)
apply (*rule some-equality*)
apply (*simp* (*no-asm-simp*))
apply *auto*
done

lemma *snoc-case-conc*:
!!*s*. [| *s*: *seq A*; *x* : *A* |] ==> *snoc-case A a f* (*s* %&ˆ %< *x* %>) = *f x s*
apply (*unfold snoc-case-def*)
apply (*rule some-equality*)
apply (*auto dest*: *snoc-inj1 snoc-inj2*)
done

declare *snoc-case-mt* [*simp*] *snoc-case-conc* [*simp*]

lemma *front-closed*:
!!*a*. [|*a* : *seq G*|] ==> *front*(*a*) : *seq G*
apply (*erule-tac s* = *a* **in** *seq-snocinduct*)
apply *auto*
done
declare *front-closed* [*simp*]

end

6.8 Bags from the Mathematical Toolkit

theory *ZBag* **imports** *ZSeqtoList* **begin**

types

```
('a) bag = 'a <=> int
```

constdefs

```
bag :: 'a set => ('a bag) set
bag S == S -|-> Naturals-1

emptybag :: 'a bag                          (%[%])
%[%] == {}

insertbag :: ['a, 'a bag] => 'a bag
insertbag x B == (if x : dom B
                  then (B (+) {(x, 1 + (B %^ x))})
                  else B Un {(x,1)})

count :: 'a bag <=> ('a <=> int)
count == {(B,c). c = ((lambda x:UNIV . 0) (+) B)}

infixcount :: ['a bag, 'a] => int           ((- [#]/ -) [70,71]70)
B [#] x == count %^ B %^ x

bagelem :: 'a <=> ('a bag)
bagelem == {(x,B). x : dom B}

bagelem-fun :: ['a,'a bag] => bool          ((- [:/ -)  [60,61]60)
x [: B == (x,B) : bagelem

bagunion :: ('a bag * 'a bag) <=> 'a bag
bagunion == {((B,C),D). dom D = dom B Un dom C &
                        (ALL x. D [#] x = B [#] x + C [#] x)}

bagunion-fun
  :: ['a bag, 'a bag] => 'a bag             ((- BUn/ -)  [70,71]70)
B BUn C == bagunion %^ (B,C)

itemsbag :: ('a seq) <=> ('a bag)
itemsbag == {(S,B). ALL x. (B : (bag UNIV)) &
```

(B [#] x = # {i. (i: dom S) &
((S %^ i) = x)})}

items :: ('a *seq*) => ('a *bag*)
items S == (*itemsbag* %^ S)

syntax
• *Bag* :: *args* => 'a *bag* (%[(-) %])

translations
%[x, xs %] == *insertbag* x %[xs%]
%[x %] == *insertbag* x %[%]

axioms
dom-ran-item: t:*seq* X --> ((*dom* (*items* t)) = (*ran* t))
mem-bagunion: (A : (*bag* X)) --> (B : (*bag* Y)) -->
(x [: (A *BUn* B)) = ((x [: A) | (x [: B))
bagunion-not-emptybag: (B ~= %[%]) --> ((A *BUn* B) ~= %[%])
bagunion-is-bag: (A : (*bag* X)) --> (B : (*bag* X)) --> ((A *BUn* B) : (*bag* X))
items-concatseq: (S : (*seq* X)) --> (T : (*seq* X)) -->
((*items* (S %&^ T)) = ((*items* S) *BUn* (*items* T)))
bagunion-emptybag: (A : (*bag* X)) --> ((%[%] *BUn* A) = A)
bagunion-emptybag2: (A : (*bag* X)) --> ((A *BUn* %[%]) = A)
count-bagunion: (A : (*bag* X)) --> (B : (*bag* X)) -->
(((A *BUn* B) [#] x) = (A [#] x + B [#] x))
items-emptybag: *items* %<%> = %[%]
items-singletonbag: *items* %< x %> = %[x %]
items-mem-ran: (s : (*seq* X)) --> ((x [: (*items* s)) = (x : (*ran* s)))
dom-bagunion: A : *bag* X --> B : *bag* X -->
(*dom* (A *BUn* B)) = ((*dom* A) *Un* (*dom* B))
domsubstr-bagunion: A : *bag* X --> B : *bag* X --> *dom* B <= M -->
((M <-: (A *BUn* B)) = (M <-: A))

declare *bagunion-not-emptybag* [*simp*] *bagunion-is-bag* [*simp*]
items-singletonbag [*simp*]
items-concatseq bagunion-emptybag bagunion-emptybag2
dom-bagunion domsubstr-bagunion

6.8.1 Basic Theorems

lemma *BagDomPow* [*simp*]: *x*: *bag X* −−> *dom x* : (*Pow X*)
apply (*unfold bag-def pfun-def*)
apply (*rule impI*)
apply (*rule Rel-Dom-Pow*)
apply *fast*
done

lemma *BagDomMem* [*simp*]: (*a* [: *B*) = (*a* : (*dom B*))
apply (*unfold bagelem-fun-def bagelem-def*)
apply (*simp* (*no-asm*))
done

lemma *BagSingleton*: %[*x* %] = {(*x*,*1*)}
apply (*unfold insertbag-def emptybag-def*)
apply *simp*
done

lemma *BagEmpty*[*simp*]: %[%] : (*bag X*)
apply (*unfold emptybag-def bag-def*)
apply *auto*
done

lemma *SizeBagEmpty*[*simp*]: # %[%] = *0*
apply (*unfold emptybag-def*)
apply *auto*
done
declare *SizeBagEmpty* [*simp*]

lemma *SizeBagSingleton*[*simp*]: # %[*x* %] = *1*
apply (*unfold emptybag-def insertbag-def zsize-def*)
apply *auto*
done

lemma *NotBagEmptyEqualSingleton* [*simp*]: %[*x* %] ~= %[%]
apply (*unfold emptybag-def insertbag-def*)
apply *auto*
done

lemma *NotBagEmptyEqualSingleton2* [*simp*]: %[%] ~= (%[*x* %])
apply (*unfold emptybag-def insertbag-def*)
apply *auto*
done

lemma *BagSingletonEqual* [*simp*]: (%[*x* %] = %[*y* %]) = (*x* = *y*)
apply (*unfold emptybag-def insertbag-def*)
apply (*rule iffI*)
apply *auto*
done

lemma *DomBagEmpty* [*simp*]: *dom* %[%] = {}
apply (*unfold emptybag-def*)
apply *auto*
done

lemma *DomBagSingleton* [*simp*]: *dom* %[*x* %] = {*x*}
apply (*unfold emptybag-def insertbag-def*)
apply *auto*
done

lemma *NotMemBagEmpty* [*simp*]: ~ *x* [: %[%]
apply (*unfold emptybag-def insertbag-def bagelem-fun-def bagelem-def*)
apply *auto*
done

lemma *MemBagSingleton* [*simp*]: (*x* [: %[*y*%]) = (*x* = *y*)
apply (*unfold emptybag-def insertbag-def bagelem-fun-def bagelem-def*)
apply *auto*
done

lemma *CountBagEmpty* [*simp*]: %[%] [#] *x* = *0*
apply (*unfold infixcount-def count-def emptybag-def Lambda-def*)
apply *auto*
apply (*subst apply-single*)
apply (*simp* (*no-asm*))
apply (*subst apply-single*)
apply (*simp* (*no-asm*))
done

lemma *count-bagsingleton*:
%[*x*%] [#] *y* = (*if x=y then 1 else 0*)
apply (*unfold infixcount-def count-def*
emptybag-def insertbag-def Lambda-def)
apply (*subst apply-single*)
apply (*unfold override-def rel-apply-def*)
apply (*case-tac x=y*)

apply *auto*
done

lemma *single-is-bag* [*rule-format*,*simp*]:
((*fst a*) : *X*) −−> ((*snd a*) : *Naturals-1*) −−> ({*a*} : (*bag X*))
apply (*unfold Naturals-1-def bag-def*)
apply *auto*
done

lemma *single-is-bag2* [*rule-format*,*simp*]:
(*a*: *G*) −−> (%[*a* %] : (*bag G*))
apply (*unfold insertbag-def emptybag-def*)
apply (*simp* (*no-asm*))
done

lemma *single-bagset*[*rule-format*,*simp*]:
(*a* = (*fst b*)) −−> ((*snd b*) = *1*) −−> (%[*a* %] = {*b*})
apply (*unfold insertbag-def emptybag-def*)
apply *auto*
apply (*rule sym*)
apply (*rule-tac t* = *1* **in** *subst*)
apply *assumption*
apply (*rule surjective-pairing*)
done

lemma *not-mem-emptybag* [*simp*]: *a* ~: %[%]
apply (*unfold emptybag-def*)
apply (*simp* (*no-asm*))
done

6.8.2 Bag Union

lemma *bagunion-emptybag-singlebag* [*simp*]: (%[%] *BUn* %[*a* %]) = %[*a* %]
apply (*rule-tac X1* = *UNIV* **in** *bagunion-emptybag* [*THEN mp*])
apply (*simp* (*no-asm*))
done

lemma *bagunion-emptybag-singlebag2* [*simp*]: (%[*a* %] *BUn* %[%]) = %[*a* %]
apply (*rule-tac X1* = *UNIV* **in** *bagunion-emptybag2* [*THEN mp*])
apply (*simp* (*no-asm*))
done

lemma *domsubstr-bagunion-singlebag* [*rule-format*,*simp*]:
(*i* : *G*) −−> *A* : *bag G* −−> ({*i*} <−: (*A BUn* %[*i* %])) = ({*i*} <−: *A*)
apply (*rule impI*)+
apply (*rule domsubstr-bagunion* [*THEN mp*, *THEN mp*, *THEN mp*])
apply *auto*
done

lemma *dom-BagSingleton-I*:
[|*x* = *a*|] ==> *x* : *dom* (%[*a* %])
by(*simp add*: *DomBagSingleton*)

end

theory *ProdContrib*
imports *Main*
begin

lemma *split-paired-Lam*: *P* = (λ (*a*,*b*). *P* (*a*,*b*))
apply (*rule ext*)
apply (*simp* (*no-asm-simp*) *only*: *split-tupled-all*)
apply (*simp* (*no-asm*))
done

lemma *split-paired-split*: (% *x*. *split f x*) = (% (*a*,*b*). *split f* (*a*,*b*))
apply (*rule ext*)
apply (*simp* (*no-asm-simp*) *only*: *split-tupled-all*)
apply (*simp* (*no-asm*))
done

lemma *split-paired-CollectL*: {(*a*,*b*). *P a b*} = {((*a*,*c*),*b*). *P* (*a*,*c*) *b*}

apply *auto*
apply (*tactic* ⟪ *TRYALL split-all-tac* ⟫)
done

lemma *split-pattern*: **assumes** *a*: *z*=(*x*,*y*) **shows** (% (*x*,*y*). *F x y*) *z* = *F x y*
apply (*simp* (*no-asm*) *add*: *a*)
done

lemma *split-paired-Ex*: (*EX x*. *P x*) = (*EX a b*. *P* (*a*,*b*))
apply *auto*
done

ML
⟪
val split-pattern = *thm split-pattern*
⟫

declare *split-paired-Ex* [*simp*]

ML ⟪
val norm-split-apply-tac =
  (*SELECT-GOAL* (*CHANGED*
   (*EVERY1* [(*match-tac* [*split-pattern RS ssubst*]),
         (*rtac refl*)])
   THEN flexflex-tac))

⟫

lemma *METACOND x* ==> ((*x*::*unit*),*y*) = ((),*y*)
apply (*subst unit-eq*)
apply (*rule refl*)
done

lemma *METACOND-pair-snd-unit*: *METACOND y* ==> (*x*,(*y*::*unit*)) = (*x*,())
apply (*subst unit-eq*)
apply (*rule refl*)
done

lemma *unit-eq-true*: ((*x*::*unit*) = *y*) = *True*

apply ((*subst unit-eq*))
apply (*simp* (*no-asm*))
done

declare *unit-eq-true* [*simp*]

end

6.9 Definitions of Syntax and Core-Combinators of Z

theory *ZPure*
imports *contrib/ProdContrib*
uses *ZPrelude.sml*
ZAbsy.sml
ZEnv.sml
ZTheory.sml
ZEncoder.sml
ZConversion.sml

begin

6.9.1 Initialize Z-Environment, provide Access

setup *ZTheory.ZEnv-setup*

ML-setup ⟪

```
fun change-global f (thy, th) = (f [th] thy, th);

fun Add-TC tc thy =
    let val zenv = ZTheory.get-zenv thy
        val tc' = (ZEnv.tc-simps-of zenv) addsimps tc
    in ZTheory.put-zenv (ZEnv.update-Zenv-by-tc-simps zenv tc') thy
    end

fun Del-TC tc thy =
    let val zenv = ZTheory.get-zenv thy
        val tc' = (ZEnv.tc-simps-of zenv) delsimps tc
```

```
  in ZTheory.put-zenv (ZEnv.update-Zenv-by-tc-simps zenv tc') thy
  end;

local
val tc-simp-attr = (Attrib.add-del-args (change-global Add-TC)
                                        (change-global Del-TC),
                    Attrib.add-del-args (fn - => error (no local tc's!))
                                        (fn - => error (no local tc's!)));
in
val - = Context.add-setup
          [Attrib.add-attributes
                [(tc-simp,
                  tc-simp-attr,
                  declaration of type checking simplification rule)]]

end;
⟫
```

setup *Context.setup* ()

6.9.2 Syntax

types *Zschema* = *bool*

syntax

```
SSTROKE   :: idt => idt        (-'')
-IN       :: idt => idt        (-?)
-OUT      :: idt => idt        (-!)
SSTROKE   :: idt => logic      (-'')
-IN       :: idt => logic      (-?)
-OUT      :: idt => logic      (-!)
```

```
%Delta  :: idt => Zschema      ((%Delta -) [100]5)
%Xi     :: idt => Zschema      ((%Xi -)    [100]5)
%theta  :: 'a  => Zschema      ((%theta -) [100]5)

syntax (xsymbols)
%Delta  :: idt => Zschema      ((Δ -) [100]5)
%Xi     :: idt => Zschema      ((Ξ -)    [100]5)
%theta  :: 'a  => Zschema      ((ϑ -) [100]5)
```

```
nonterminals
  rnm
  rnms
```

```
syntax
  -rnm    :: [idt , idt] => rnm              ((- '/ -) )
          :: rnm => rnms                     (-)
  -rnms   :: [rnm, rnms] => rnms             (-,/ -)
 %not     :: Zschema => Zschema              (%not (-)// [40]40)
  \'/     :: [Zschema, Zschema]    => Zschema (infixl 30)
  '/\     :: [Zschema, Zschema]    => Zschema (infixl 35)
  =+=>    :: [Zschema, Zschema]    => Zschema (infixl 25)
  <=+=>   :: [Zschema, Zschema]    => Zschema (infixr 25)
  pre     :: Zschema               => Zschema
  %E      :: [Zschema, Zschema]    => Zschema (%E (2-) • (-)    [30,31]30)
  %E1     :: [Zschema, Zschema]    => Zschema (%E1 (3-) • (-)   [30,31]30)
  %A      :: [Zschema, Zschema]    => Zschema (%A (2-) • (-)    [30,31]30)
  -hiding :: [Zschema, args]       => Zschema ((-)\[(-)]    [80,81]80)
  -geninst:: [idt, args]           => Zschema ((-)[(-)]     [80,81]80)
  -rename :: [idt, rnms]           => Zschema ((-)\[(-)]    [80,81]80)
  -schset :: [idt, 'a]             => 'b set  ((1{.(-) • (-).}))
  -select :: ['a,idt]              => 'c      (-.- [200,201]200)
  |\      :: [Zschema, Zschema]    => Zschema (infixl 80)
  %%;     :: [Zschema, Zschema]    => Zschema (infixl 80)
  >>      :: [Zschema, Zschema]    => Zschema (infixl 80)
```

syntax (*xsymbols*)
%E :: [*Zschema*, *Zschema*] => *Zschema* (∃ (*2*-) • (-) [*30*,*31*]*30*)
%E1 :: [*Zschema*, *Zschema*] => *Zschema* (∃! (*3*-) • (-) [*30*,*31*]*30*)
%A :: [*Zschema*, *Zschema*] => *Zschema* (∀ (*2*-) • (-) [*30*,*31*]*30*)

translations

%*not P* => ~ *P*
A \/ *B* => *A* | *B*
A /\ *B* => *A* & *B*
A =+=> *B* => *A* −−> *B*
A <=+=> *B* => *A* = *B*

types *Predicate* = *bool*
DeclPart = *bool*

nonterminals
DeclParts
SchemaName

syntax
:: *DeclPart* => *DeclParts* (-)
-combdecls :: [*DeclPart*, *DeclParts*] => *DeclParts* (-; / -)

consts
SCHEMA :: [*bool*, *bool*] => *Zschema*

translations
-combdecls D D1 => *D* & *D1*

syntax
-AXIOM0 :: *Zschema*
(+.. //−..)

```
-AXIOM1     :: DeclParts => Zschema
            (+..//(2 -)//−..)
-AXIOM2     :: Predicate => Zschema
            (+.. //|−− //(2 -)//−..)
-AXIOM3     :: [DeclParts, Predicate] => Zschema
            (+..//(2 -)//|−−//(2 -)//−..)
```

translations

```
-AXIOM0     == SCHEMA True True
-AXIOM1 D   == SCHEMA D True
-AXIOM2 P   == SCHEMA True P
-AXIOM3 D P == SCHEMA D P
```

syntax

```
-GEN0 :: args => Zschema
                    (+== [(-)] ===//−==)
-GEN1 :: [args, DeclParts]=> Zschema
                    (+== [(-)] ===//(2 -) //−==)
-GEN2 :: [args, Predicate]=> Zschema
                    (+== [(-)] ===//|−−//(2 -)//−==)
-GEN3 :: [args, DeclParts, Predicate] => Zschema
                    (+== [(-)] ===//(2 -)//|−−//(2 -)//−==)
```

translations

```
-GEN0 M == %M. (-AXIOM0)
-GEN1 M D == %M. (-AXIOM1 D)
-GEN2 M P == %M. (-AXIOM2 P)
-GEN3 M D P == %M. (-AXIOM3 D P)
```

syntax

```
-SCHEME0:: SchemaName => Zschema                (+−− - −−− //−−− )
-SCHEME1::[SchemaName, args] => Zschema         (+−− - [-] −−−//−−−)
-SCHEME2::[SchemaName, DeclParts] => Zschema    (+−− - −−−//(2
-)//−−−)
-SCHEME3::[SchemaName, Predicate] => Zschema    (+−− - −−−//|−−//(2
-)//−−− )
-SCHEME4::[SchemaName, args, DeclPart] => Zschema  (+−− - [-] −−−//(2
-)//−−−)
-SCHEME5::[SchemaName, args, Predicate]=> Zschema  (+−− - [-] −−−//|−−//(2
-)//−−−)
```

-SCHEME6::[*SchemaName*, *DeclParts*,*Predicate*]=>*Zschema*(+−− - −−−//(*2* -)//|−−//(*2* -)//−−−)
-SCHEME7::[*SchemaName*, *args*, *DeclParts*, *Predicate*]=>*Zschema* (+−− - [-] −−−//(*2* -)//|−−//(*2* -)//−−−)

translations
-SCHEME0 N == (*prop*) *N* == *-AXIOM0*
-SCHEME1 N M == (*prop*) *N* == (*-GEN0 M*)
-SCHEME2 N D == (*prop*) *N* == (*-AXIOM1 D*)
-SCHEME3 N P == (*prop*) *N* == (*-AXIOM2 P*)
-SCHEME4 N M D == (*prop*) *N* == (*-GEN1 M D*)
-SCHEME5 N M P == (*prop*) *N* == (*-GEN2 M P*)
-SCHEME6 N D P == (*prop*) *N* == (*-AXIOM3 D P*)
-SCHEME7 N M D P == (*prop*) *N* == (*-GEN3 M D P*)

6.9.3 Semantic Representation

consts

SNAME0 :: *'a* => *'a*
SNAME :: [*'a*=>*'b*,*'a*] => *'b*
SBinder0 :: [*'a*, *'b* => *'d*] => (*'b* => *'d*)
SBinder :: [*'a*,[*'b*,*'c*] => *'d*] => ((*'b*∗*'c*)=>*'d*)

asSet :: [*'a* => *bool*] => *'a set*
asPred :: [*'a set*] => *'a* => *bool*
turnstyle :: (*'a* => *bool*) => *bool* (|− (*3*-))
theta :: [*string*,*'a*] => *'a*
DELTA :: [*bool*,*bool*] => *bool*
XI :: [*bool*,*'a*,*'a*] => *bool*
PROJ :: [*'a*,*'a*=>*'b*,*'c*]=>*'b*
DECL :: [*bool*,*bool*] => *bool* ((*0*-/ |−−−−/ -) [*15*,*14*]*14*)
PRE :: [*'a* => *bool*] => *bool*
SBall :: [*'a set*, *'a* => *bool*] => *bool*
SBex :: [*'a set*, *'a* => *bool*] => *bool*
SBex1 :: [*'a set*, *'a* => *bool*] => *bool*
SSet :: [*'a set*, *'a* => *'b*] => *'b set*

defs
SNAME0-def: *SNAME0 x* == *x*
SNAME-def: *SNAME f args* == *f args*

```
SBinder0-def: SBinder0 An P == P
SBinder-def:  SBinder An P  == (% (x,y). (P x y))
asSet-def:    asSet         == Collect
asPred-def:   asPred        == (%X. %x. x : X)
turnstyle-def:|- P          == All P
PROJ-def:     PROJ X F ide  == F X
DELTA-def:    DELTA s t     == s & t
XI-def:       XI s t t'     == s & t=t'
theta-def:    theta s t     == t
DECL-def:     DECL P Q      == P & Q
PRE-def:      PRE           == Ex
SBall-def:    SBall         == Ball
SBex-def:     SBex          == Bex
SBex1-def:    SBex1 A P     == ?! x. x : A & P x
SSet-def:     SSet A f      == f ` A
```

nonterminals

```
sbind
sbinds
```

syntax

```
-sbind  :: ['a , 'b] => sbind          ((2- ~~> -) 10)
        :: sbind => sbinds             (-)
-sbinds :: [sbind, sbinds] => sbinds   (-,/ -)
-SB     :: [sbinds,'a] => 'a           ((4SB (-). (-)) 10)
```

translations

```
%Delta S  => DELTA S (SSTROKE S)
%Delta S  <= DELTA S T
%Xi    S  => XI (%Delta S) (%theta S) (%theta (SSTROKE S))
%Xi    S  <= XI (%Delta S) (T) (U)


-SB (-sbinds (-sbind x y) sbds) e == SBinder x (%y. -SB sbds e)
-SB (-sbind x y) e                == SBinder0 x (%y. e)


pre S     <= PRE S
%E S • T  <= SBex  S T
%E1 S • T <= SBex1 S T
```

```
%A S • T <= SBall S T
{.S • T.} <= SSet S T
```

6.9.4 Syntax: The parse-translation setup

ML⟪

```
val show-full-sem = ref false;
val SAVE = ref([]:term list)

(* switch to shut off the pretty printing to email format *)

local

local open Ast ZPrelude ZEnv in

(* ************************************************************ *)
(* absolute pre−parsing ... runs before ZEncoder.schema-conv !!! *)
(* ************************************************************ *)
fun parse-SSTROKE [Free(x,t)] = Free(stroke (x,t))
  |parse-SSTROKE [Const(x,t)]= Const(stroke(x,t))
  |parse-SSTROKE - = error (Illegal STROKE−Term: must be identifier.);

fun parse-IN [Free(x,t)] = Free(x^ZPrelude.in-sym,t)
  |parse-IN - = error (Illegal IN−Term: must be identifier.);

fun parse-OUT [Free(x,t)] = Free(x^ZPrelude.out-sym,t)
  |parse-OUT - = error (Illegal OUT−Term: must be identifier.);

fun parse-SCHEMA [decl,pred] = Const(DECL,dummyT) $ decl $ pred
  |parse-SCHEMA - = error (Illegal Schema Term.);
   (* why as parse translation ??? *)


(* *************** *)
(* pretty−printing *)
(* *************** *)


fun print-decls (Const (op &,t) $ T $ T') =
                    Const (-combdecls,t) $ T $ (print-decls T')
  |print-decls T = T;

fun print-Proto [decl, preds] =
(* Introduces the root−symbol for pretty−printing schemata in Zproto *)
```

```
      if !show-full-sem then raise Match
      else Const(SCHEMA,dummyT) $ (print-decls decl) $ preds
  |print-Proto - = raise Match;

fun TR s h = (writeln s; h)

fun print-SB0 [anno, Abs(-,t,body)] =
      if !show-full-sem then (raise Match)
      else (subst-bounds([Free(dest-string anno,t)],body))
  |print-SB0 - = raise Match;



fun print-SB (anno:: Abs(-,t,body)::R) =
      if !show-full-sem then raise Match
      else (list-comb(subst-bounds
                    ([Free(dest-string anno,t)],body),R))
  |print-SB - = raise Match

fun print-SCH-name0 [x] =
      if !show-full-sem then raise Match else x
  |print-SCH-name0 - = raise Match


fun print-SCH-name [f as Const(s,-),arg] =
(* delifting all applications marked as a schema−names applied to argument
   tuple. It is checked that 1. arg forms a tuple, 2. components of arg
   are just Free's, 3. all names in the free's match to the signature sign
   of s modulo a unique number d of strokes '.

   USES ZENV FROM ENCODER (parser . . .)
  *)
      if !show-full-sem then raise Match
      else (let val - = (SAVE:=[f,arg] • (!SAVE))
              fun stripFree (Free(s,t) :: R) = s :: (stripFree R)
                | stripFree((Const(-bound,-) $ Free(s,t))::R) =
                                          s :: (stripFree R)
                  (* this case captures variables bound by other
                     quantifiers than just schema quantifiers ...*)
                | stripFree (- :: R) = raise Match
                | stripFree []     = [];
              val x::R = stripFree (strip-tuple arg);
              val sign = (case lookup (schemas-of(!ZEncoder.ZENV),s) of
                          NONE => raise Match
```

```
                    |SOME n => map fst (schemasig-of n));
          val y    = case sign of a::- => a | - => raise Match;
          val d    = (size x) - (size y);
          fun cm [] [] = true
            |cm (a::r)(a'::r') = if (size a) + d = size a' andalso
                                   a = (substring(a',0,size a' - d))
                                 then cm r r'
                                 else raise Match
            |cm - - =  raise Match
        in  if d >= 0 andalso cm sign (x::R)
          then genstrokeN d f
          else raise Match
        end)
  |print-SCH-name - = (raise Match);

fun print-theta [s,e] =
      if !show-full-sem then raise Match
      else let val t = dest-string s;
              val n = strokes t;
          in   Const(%theta,dummyT)$
                 (genstrokeN n (Free(destroke t,dummyT)))
          end
  |print-theta - = raise Match

fun print-asSet [a] = if !show-full-sem then raise Match else a
  |print-asSet - = raise Match;

fun print-asPred [a] =  if !show-full-sem then raise Match else a
  |print-asPred - = raise Match

fun print-PROJ [X,F,str] =
      if !show-full-sem then raise Match
      else Const(-select,dummyT)$ X $
          Const(dest-string str,dummyT)
  |print-PROJ - = raise Match

end (* local open *)

in

val parse-translation = [(SSTROKE, parse-SSTROKE),
                         (-IN, parse-IN),
                         (-OUT, parse-OUT),
                         (SCHEMA, parse-SCHEMA)];
```

```
val print-translation = [(SBinder0, print-SB0),
                         (SBinder, print-SB),
                         (DECL, print-Proto),
                         (SNAME0,print-SCH-name0),
                         (SNAME,print-SCH-name),
                         (theta,print-theta),
                         (asSet,print-asSet),
                         (asPred,print-asPred),
                         (PROJ,print-PROJ)
                        ];

end
⟫
```

parse-translation ⟪ *parse-translation* ⟫
print-translation ⟪ *print-translation* ⟫

6.9.5 Derived Rules

Schema Binders

lemma *SB0-ext*: $(\bigwedge x.\ f\ x = g\ x) \Longrightarrow$ *SBinder0 a f = SBinder0 b g*
by (*rule ext*, *simp add*: *SBinder0-def*)

lemma *SB-ext*: $(\bigwedge x.\ f\ x = g\ x) \Longrightarrow$ *SBinder a f = SBinder b g*
by(*simp add*: *SBinder-def*)

lemma *conv-sname0* : $f \equiv g \Longrightarrow$ *SNAME0 f = g*
by(*simp add*: *SNAME0-def*)

lemma *conv-sname* : $f \equiv g \Longrightarrow$ *SNAME f args = g args*
by(*simp add*: *SNAME-def*)

lemma *turnstyle-SB0*: $(\bigwedge x.\ f\ x) \Longrightarrow$ $|-$ *SBinder0 a f*
by(*auto simp*: *SBinder0-def turnstyle-def*)

lemma *turnstyle-SB*: $(\bigwedge x.\ |-\ f\ x) \Longrightarrow$ $|-$ *SBinder a f*
by(*simp add*: *SBinder-def turnstyle-def*)

lemma *SB0-ext-id*: $(\bigwedge x.\ f\ x = g\ x) \Longrightarrow$ *SBinder0 a f = SBinder0 a g*
by(*rule ext*, *simp add*: *SBinder0-def*)

lemma *SB-ext-id*: $(\bigwedge x.\ f\ x = g\ x) \Longrightarrow$ *SBinder a f = SBinder a g*
by(*simp add*: *SBinder-def*)

lemma *SB0-extL*: $(\bigwedge x.\ f\ x = g\ x) \Longrightarrow SBinder0\ a\ f = g$
by(*rule ext*, *simp add*: *SBinder0-def*)

lemma *SB-extL*: $(\bigwedge x\ y.\ f\ x\ y = g\ (x,\ y)) \Longrightarrow SBinder\ a\ f = g$
by(*simp add*: *SBinder-def split-eta*)

lemmas *Zexts* = *ext SB0-ext SB-ext*

lemma *SB0-eq*: *SBinder0 a P x* = *P x*
by(*simp add*: *SBinder0-def*)

lemma *SB-eq*: *SBinder a P z* = (*P* (*fst z*) (*snd z*))
apply(*simp add*: *SBinder-def*)
apply (*rule surjective-pairing* [*THEN ssubst*])
apply (*safe+*, *simp*)
done

lemma *SB-pair-eq*: *SBinder a P* (*x*,*y*) = (*P x y*)
by (*simp add*: *SBinder-def*)

lemma *EX-SBinder0*: (∃ *x*. (*SBinder0 a P*) *x*) = (∃ *x*. *P x*)
by (*simp add*: *SBinder0-def*)

lemma *EX-SBinder*: (∃ *x*. (*SBinder a P*) *x*) = (∃ *x y*. *P x y*)
by (*simp add*: *SBinder-def*)

declare *SB0-eq SB-eq EX-SBinder0 EX-SBinder*[*simp*]

DECL

lemma *DECL-I* : $[\![A;\ A \Longrightarrow B]\!] \Longrightarrow A$ |---- B
by (*auto simp*: *DECL-def*)

lemma *DECL-D1* : A |---- $B \Longrightarrow A$
by (*auto simp*: *DECL-def*)

lemma *DECL-D2* : A |---- $B \Longrightarrow B$
by (*auto simp*: *DECL-def*)

lemma *DECL-E*: $[\![P$ |---- $Q;\ [\![P;\ Q]\!] \Longrightarrow R]\!] \Longrightarrow R$
by (*auto simp*: *DECL-def*)

lemma *DECL-cong*:
$[\![P = P'; \; P' \Longrightarrow Q = Q']\!] \Longrightarrow (P \; |---- \; Q) = (P' \; |---- \; Q')$
by (*auto simp*: *DECL-def*)

declare *DECL-I* [*intro*!]
declare *DECL-E* [*elim*!]
declare *DECL-cong* [*cong*]

Tests:

ML⟪
(∗ *Syntax−Checks*... ∗)
goalw (*the-context*()) [] (*SB x* ~~> *y*, *a* ~~> *b*. *DECL A B*) = *?X*;
goalw (*the-context*()) [] *fgh''* = *?X*
⟫

lemma *SBall-I* : $(\bigwedge x.\ x \in A \Longrightarrow P\ x) \Longrightarrow$ *SBall A P*
by(*auto simp*: *SBall-def*)

lemma *SBall-I2* : $(\bigwedge x.\ A\ x$ ==> $P\ x)$ ==> *SBall* (*asSet A*) *P*
by(*auto simp*: *asSet-def SBall-def*)

lemma *bla* : $(\bigwedge z\ y\ .\ x{=}(y{,}z) \Longrightarrow f\ y\ z) \Longrightarrow$ (*SBinder a f*) *x*
by(*auto simp*: *split-def SBinder-def*)

lemma *bla0* : $(\bigwedge y.\ x{=}y \Longrightarrow f\ y) \Longrightarrow$ (*SBinder0 a f*) *x*
by(*auto simp*: *split-def SBinder0-def*)

lemma *bla'* : (*SBinder a f*) *x* $\Longrightarrow$ $(\exists\ y\ z.\ f\ y\ z)$
by(*auto simp*: *split-def SBinder-def*)

lemma *bla0'*: (*SBinder0 a f*) *x* $\Longrightarrow f\ x$
by(*auto simp*: *split-def SBinder0-def*)

lemma *blaE* : $(\exists\ y\ z.\ f\ y\ z \wedge x{=}(y{,}z)) \Longrightarrow$ (*SBinder a f*) *x*
by(*auto simp*: *split-def SBinder-def*)

lemma *bla0E*: $(f\ x) \Longrightarrow$ (*SBinder0 a f*) *x*
by(*auto simp*: *split-def SBinder0-def*)

lemma *PRE-I* : $[\![\ P\ x\ ]\!] \Longrightarrow$ *PRE P*
by(*auto simp*: *PRE-def*)

lemma *PRE-E* : $[\![$ *PRE P*; $\bigwedge x.\ \ P\ x \Longrightarrow Q\ ]\!] \Longrightarrow Q$
by(*auto simp*: *PRE-def*)

lemma *SBex-E1* : $[\![$ *SBex A P*; $\bigwedge x.\ [\![\ x : A;\ \ P\ x\ ]\!] \Longrightarrow Q\ ]\!] \Longrightarrow Q$
by(*auto simp*: *SBex-def asSet-def*)

lemma *SBex-E2* : $[\![$ *SBex* (*asSet A*) *P*; $\bigwedge x.\ [\![\ A\ x;\ \ P\ x\ ]\!] \Longrightarrow Q\ ]\!] \Longrightarrow Q$
by(*auto simp*: *SBex-def asSet-def*)

lemma *SBex-I1* : $[\![\ P\ x;\ \ x : A\ ]\!] \Longrightarrow$ *SBex A P*
by(*auto simp*: *SBex-def asSet-def*)

lemma *SBex-I2* : $[\![\ P\ x;\ \ A\ x\ ]\!] \Longrightarrow$ *SBex* (*asSet A*) *P*
by(*auto simp*: *SBex-def asSet-def*)

lemma *turnstyle-E*: $|\!-\ P \Longrightarrow P\ x$
by(*auto simp*: *turnstyle-def*)

lemma *SBall-E* : $[\![$ *SBall A P*; $x : A\ ]\!] \Longrightarrow P\ x$
by(*auto simp*: *SBall-def*)

6.9.6 Proof Support: Z-tactics

lemmas *Z2HOL* = *SNAME0-def SNAME-def SBinder0-def SBinder-def turnstyle-def*

asSet-def asPred-def DECL-def theta-def PROJ-def
DELTA-def XI-def PRE-def SBall-def SBex-def SBex1-def

lemmas *set-simps* = *insert-subset*
insert-not-empty empty-not-insert
Int-absorb Int-empty-left Int-empty-right
Un-absorb Un-empty-left Un-empty-right Un-empty
UN-empty UN-insert image-empty
Compl-disjoint double-complement
Union-empty Union-insert empty-subsetI subset-refl
Diff-cancel empty-Diff Diff-empty Diff-disjoint

lemmas *prod-simps* = *fst-conv snd-conv split Pair-eq*

ML Code

```
ML⟪
(* compatibility: *)
structure ZPure =
struct
  val thy = the-context()
end


val SB0-def      = thmSBinder0-def  val SB-def      = thmSBinder-def
val SNAME0-def   = thmSNAME0-def    val SNAME-def   = thmSNAME-def
val SBall-def    = thmSBall-def     val SBex-def    = thmSBex-def
val SBex1-def    = thmSBex1-def     val asSet-def   = thmasSet-def
val PRE-def      = thmPRE-def       val DECL-def    = thmDECL-def
val theta-def    = thmtheta-def     val asPred-def  = thmasPred-def
val turnstyle-def= thmturnstyle-def val PROJ-def    = thmPROJ-def
val DELTA-def    = thmDELTA-def     val XI-def      = thmXI-def

val empty-not-insert = thm empty-not-insert;

val SB0-ext      = thmSB0-ext       val SB-ext       = thmSB-ext
val turnstyle-SB0= thmturnstyle-SB0val turnstyle-SB = thmturnstyle-SB
val SB0-ext-id   = thmSB0-ext-id    val SB-ext-id    = thmSB-ext-id
val SB0-extL     = thmSB0-extL      val SB-extL      = thmSB-extL
val SB0-eq       = thmSB0-eq        val SB-eq        = thmSB-eq
val SB-pair-eq   = thmSB-pair-eq    val EX-SBinder0  = thmEX-SBinder0
val EX-SBinder   = thmEX-SBinder    val DECL-I       = thmDECL-I
val DECL-D1      = thmDECL-D1       val DECL-D2      = thmDECL-D2
val DECL-E       = thmDECL-E        val DECL-cong    = thmDECL-cong

val conv-sname0  = thmconv-sname0   val conv-sname   = thmconv-sname
val SBall-I      = thmSBall-I       val SBall-I2 = thmSBall-I2
val SBallE       = thmSBall-E

val sbexE1       = thmSBex-E1       val sbexE2   =thmSBex-E2
val sbexI1       = thmSBex-I1       val sbexI2   =thmSBex-I2
val preI         = thmPRE-I         val preE = thmPRE-E
val bla          = thmbla           val bla0 = thmbla0
val bla'         = thmbla'          val bla0'= thmbla0'
val blaE         = thmblaE          val bla0E=thmbla0E
val turnstyleE   = thmturnstyle-E

val set-ss =
  HOL-ss addsimps mem-simps
```

```
        addcongs [ball-cong,bex-cong]
        setmksimps (mksimps mksimps-pairs);

val set-ss = set-ss addsimps
  [insert-subset,
   insert-not-empty,empty-not-insert,
   Int-absorb,Int-empty-left,Int-empty-right,
   Un-absorb,Un-empty-left,Un-empty-right,Un-empty,
   UN-empty, UN-insert,image-empty,
   Compl-disjoint,double-complement,
   Union-empty,Union-insert,empty-subsetI,subset-refl,
   Diff-cancel,empty-Diff,Diff-empty,Diff-disjoint];

val prod-ss = set-ss addsimps [fst-conv, snd-conv, split, Pair-eq];

fun prover s = prove-goal HOL.thy s (fn - => [fast-tac HOL-cs 1]);

⟫
```

ML⟪

```
use ZProofUtil.sml;
open ZProofUtil;

(* Test *)
ZProofUtil.lift conj-commute 5;
(*val it =  (SB ?A ~~> x, ?A1 x ~~> xa, ?A2 x xa ~~> xb,
                ?A3 x xa xb ~~> xc, ?A4 x xa xb xc ~~> xd.
                     ?P5 x xa xb xc xd /\\\\ ?Q5 x xa xb xc xd) =
            (SB ?A ~~> x, ?A1 x ~~> xa, ?A2 x xa ~~> xb,
                ?A3 x xa xb ~~> xc,  ?A4 x xa xb xc ~~> xd.
                     ?Q5 x xa xb xc xd /\\\\ ?P5 x xa xb xc xd) *)
⟫
```

ML-setup⟪

```
fun nat-arg f = (f) oo
 (#2 oo Method.syntax(Scan.lift (Scan.optional (Args.parens Args.nat) 1)));

fun nat-thms-args f = uncurry f oo
   (#2 oo Method.syntax (Scan.lift (Scan.optional (Args.parens Args.nat) 1) --
Attrib.local-thmss));
```

```
fun tac2meth-unary tac n = Method.SIMPLE-METHOD (tac n);
fun tac2meth-binary tac thmS n = Method.SIMPLE-METHOD (tac thmS n);
fun tac2meth tac n ths = let fun multitac tac thms n = FIRST(map (fn thm => tac
thm n) thms)
                    in  Method.SIMPLE-METHOD
                            (CHANGED-PROP (multitac tac ths n))
                    end;

val zrule-meth   = tac2meth zrtac;
val zdrule-meth  = tac2meth zdtac;
val zerule-meth  = tac2meth zetac;
val zfrule-meth  = tac2meth zftac;
val zstrip-meth  = tac2meth-unary stripS-tac;
val zunfold-meth = tac2meth-binary (fn x => fn y => if x = 0
                                        then expand-schema-tac y 1
                                        else full-expand-schema-tac y 1)
              (* ^^^^^^^^^^^^^^^^^^^ HACK *)

val zintro-schema-meth  = tac2meth-unary  intro-schema-tac;
val zintro-sch-all-meth = tac2meth-unary  intro-sch-all-tac
val zintro-sch-ex-meth  = tac2meth-unary  intro-sch-ex-tac
val zintro-pre-meth     = tac2meth-binary (fn x => fn y => intro-pre-tac y x)
val zintro-sch-ex1-meth = tac2meth-unary  intro-sch-ex1-tac
val zelim-sch-ex-meth   = tac2meth-unary  elim-sch-ex-tac
val zelim-pre-meth      = tac2meth-unary  elim-pre-tac

⟫
```

ML⟪

```
val zmethods = [
  (zrule, nat-thms-args zrule-meth,
          apply Z−rule in resolution manner (improper)),
  (zerule, nat-thms-args zerule-meth,
          apply Z−rule in elimination manner (improper)),
  (zdrule, nat-thms-args zdrule-meth,
          apply Z−rule in destruction resolution manner (improper)),
  (zfrule, nat-thms-args zfrule-meth,
          apply Z−rule in forward resolution manner (improper)),
  (zintro-pre, nat-thms-args zintro-pre-meth (*zfrule-meth should be:  zintro-pre-meth
*),
          apply Z−rule in resolution manner (improper)),
```

```
  (zstrip, nat-arg zstrip-meth,
           strip schema calculus operators (improper)),
  (zunfold, nat-thms-args zunfold-meth,
           unfold schema definition (improper)),
  (zintro-schema, nat-arg zintro-schema-meth,intro schema expression),
  (zintro-sch-all,nat-arg zintro-sch-all-meth,intro schema all quantifier),
  (zintro-sch-ex,nat-arg zintro-sch-ex-meth,intro schema ex quantifier),
  (zintro-sch-ex1,nat-arg (tac2meth-unary intro-sch-ex1-tac),
                  intro schema ex1 quantifier),
  (zelim-sch-ex,nat-arg zelim-sch-ex-meth,eliminate schema ex quantifier),
  (zelim-pre,nat-arg zelim-pre-meth,eliminate schema pre)

 ]

val - = Context.add-setup [Method.add-methods zmethods];

  ⟫
```

ML⟪

```
local
fun gen-GET thmP comb =
    let fun apply f B (x, A) = (x, f(A,B))
    in Attrib.syntax ( Scan.lift (Scan.optional (Args.bracks Args.nat) 1) >>
                     (apply (uncurry comb)))
    end;
val GET-global = gen-GET Attrib.global-thm (get-decl2);
val GET-local  = gen-GET Attrib.local-thm;
fun zstrip x = Attrib.no-args (Drule.rule-attribute (K (stripS))) x;

in

val - =
let val z-attributes =
    [(zstrip,  (zstrip,zstrip),strip schema operators),
     (zdecl,(gen-GET Attrib.global-thm get-decl2,
              gen-GET Attrib.local-thm get-decl2),get conjunct from declaration part),
     (zpred,(gen-GET Attrib.global-thm get-conj2,
              gen-GET Attrib.local-thm get-conj2),get conjunct from predicative part)]
in Context.add-setup [Attrib.add-attributes z-attributes] end;

end;
⟫
```

setup *Context.setup* ()

end

6.10 Method Package for Z Representation and Z Refinement

theory *ZMethod*
imports *ZPure*
uses *ProofObligationMgr.sml*
begin

6.10.1 Configuring the Generic PO Manager

ML⟪

```
structure FMethod (*: FMETHOD*) =
  struct
    open POKind

    val PO-classifier = Symtab.make[(tcc,                    repr),
                                    (axdefConservative,      lethal),
                                    (ccState,                wff),
                                    (ccOp,                   wff),
                                    (fwRefinementFunctional,meth),
                                    (fwRefinementInit,       meth),
                                    (fwRefinementOp,         meth)]

    datatype content = Mk of {abs : (string*string*string) option,
                              functional-refinement : bool}

    val  init    = Mk{abs = NONE,functional-refinement=false}
    val  merge   = fn (x,y) => init
    fun  boolToString x    = if x then true else false
    fun  print (Mk{abs,functional-refinement}) =
            (writeln (Method Package State:);
             writeln (====================);
             writeln (abstraction relation: ^
                        (case abs of NONE => --- | SOME (x,y,z) => x));
             writeln (functional refinemt : ^
                        (boolToString functional-refinement)))
  end
```

structure ZPO-Mgr = PO-Manager (FMethod);

⟫

Initialize Environment

setup *ZPO-Mgr.PO-state-setup*

6.10.2 The Z Method Package

On top of the generic PO Manager, which has now been imported and instantiated, we define a number of Method- specific PO generation methods. In our case, we implement support for classical forward data refinement (with an optimizing option for functional abstraction relations) following the lines described in the Spivey-Book also described in the Woodcock/Davis Book "Using Z".

There is a general configuration operation `set_abs` that sets the underlying abstration relation and concepts such as abstract or concrete state.

PO generators are in particular:

1. `gen_state_cc` for checking that state schemas (representing system invariants) are in fact satisfyable.
2. `gen_op_cc` for checking that operation schemas are "implementable" or "non-blocking", i.e. there exists a function that maps inputs and state to outputs and successor state. (Note that this function is **not** necessarily computable).
3. `gen_thm_tcc` extracts from an arbitrary definition or axiom the side-conditions for type-constraint-consistency. (a representational side-condition of HOL-Z).
4. `declare_abs` allows for setting the abstraction relation (per schema definition). An additional option `[functional]` specifies the relation to be functional, which results in different (usually simpler) proof obligations.
5. `refine_init` for generating a PO assuring compatibility of Init schemas (in forward simulation)
6. `refine_op` for generating two PO's establishing refinement of operation schemas.

ML⟪

```
signature ZMETHOD-PACKAGE =
  sig
    type schema-name = string

    val gen-tcc-from : string -> theory -> theory
     (* PRE string denotes thm which is schema-def or axdef.
        POST add PO for each C(F%^E): CONT(C, F : Domain E)
      *)

    val gen-cc-state : schema-name -> theory -> theory
     (* PRE stateschema is schema without x?, x!, x' variables.
        POST add PO  : ∃ x. x : stateschema *)

    val gen-cc-op    : schema-name -> theory -> theory
     (* PRE opschema is schema with x and x' variables.
        POST add PO  : ∀ state.∀ in.∃ state.
                                                ∃ out. op
      *)

    val declare-abs  : bool -> schema-name -> theory -> theory
     (* PRE  absname must denote Z schema,
             which imports exactly two schemas.
             The first is used as abstract state,
             the second as concrete state.
        POST if functional,
                add PO: ∀ state-conc. ∃1 abs-state. abs
             else SKIP
      *)

    val refine-init  : schema-name * schema-name -> theory -> theory
     (* PRE  abs relation must be set.
             init-abs init-conc must be schema names.
        POST if functional:
                  add PO:∀ state-abs state-conc.
                                  conc-init & abs \implies abs-init
             else add PO:∀ state-conc. init-conc
                                      \implies (∃ state-abs.
                                              init-abs & abs)
      *)
    val refine-ops   : schema-name * schema-name -> theory -> theory
     (* PRE  abs relation must be set.
             op-abs op-conc must be schema names.
        POST if functional:
                add PO: ∀ state-abs state-conc. ∀ in.
```

```
                    pre op-abs ∧ abs ⟶ pre op-conc
        add PO: ∀ state-abs state-conc. ∀ in out.
                    pre op-abs ∧ abs ∧ op-conc ∧ abs′
                        ⟶ op-abs
      else
        add PO: ∀ state-abs state-conc. ∀ in. pre op-abs
                                    & abs ⟶ pre op-conc
        add PO: ∀ abs-state conc-state. ∀ in out.
                    pre op-abs ∧ abs ∧ op-conc ⟶
                        ∃ state-abs′. abs′ ∧ op-abs
    *)

  end

⟫
```

ML⟪

```
structure ZMethod-Package : ZMETHOD-PACKAGE =
struct
  val DEBUG = ref([]: string list)

  type schema-name = string

  fun gen-tcc-from name thy          = error(NOT YET IMPLEMENTED)

  fun is-schema name thy =
      ZEnv.is-def(ZEnv.schemas-of (ZTheory.get-zenv thy),name)


  fun gen-cc-state stateschema thy      =
  let val - = if is-schema stateschema thy then ()
            else error(Must be schema name: ^stateschema)
      val zsig-opt= ZEnv.lookup (schemas-of(ZTheory.get-zenv thy),stateschema)
      val nn= map fst (filter(fn (x,-) => ZPrelude.is-in-name x orelse
                                  ZPrelude.is-out-name x orelse
                                  ZPrelude.is-stroke-name x)
                      (schemasig-of(the zsig-opt)))
      val - = if null nn then ()
            else error(state schema should contain only \
                    \undecorated variables: ^(String.concat nn))
      val -    = (ZEncoder.ZENV:=ZTheory.get-zenv thy)

      val agoal= %E ^ stateschema^ • True;
```

```
    val cf   =  cterm-of(sign-of thy)
                  (ZEncoder.schema-read thy agoal)
    val thy' = ZPO-Mgr.declare-PO
                    (SOME(Context.theory-name thy),
                     ccState,stateschema)
                     cf thy
    val -    = writeln(PO declared: ^(the
                        (ZPO-Mgr.get-name-of-PO cf thy')))

  in thy'
  end


 fun gen-cc-op opschema thy              =
 let val - = if is-schema opschema thy then ()
            else error(Must be schema name: ^opschema)
    val zsig-opt= ZEnv.lookup (schemas-of(ZTheory.get-zenv thy),opschema)
    val nn   = map fst (filter(fn (x,-) => ZPrelude.is-stroke-name x)
                         (schemasig-of(the zsig-opt)))
    val -    = if not(null nn) then ()
              else error(op schema should contain  \
                    \post state (i.e. 'stroked') components. )
    val -    = (ZEncoder.ZENV:=ZTheory.get-zenv thy)

    val agoal= pre ^ opschema;
    val cf   = cterm-of(sign-of thy)
                  (ZEncoder.schema-read thy agoal)
    val thy' = ZPO-Mgr.declare-PO
                    (SOME(Context.theory-name thy),
                     ccOp,opschema)
                     cf thy
    val -    = writeln(PO declared: ^(the
                        (ZPO-Mgr.get-name-of-PO cf thy')))

  in thy'
  end


 fun declare-abs functional abs thy =
 let
    val - = if is-schema abs thy then ()
            else error(Must be schema name: ^abs)
    val abs-term = concl-of (get-thm thy (Name(abs^-def)))
```

```
val schema = ZEncoder.strip-Binder
            (case abs-term of
               Const(==,-) $ - $ t => t
             | - => error(name must denote schema!))
val decl   = (case schema of
             Const(ZPure.DECL,-) $ H $ - => H
            |- => error(name must denote schema!))
fun strip-sname (Const(ZPure.SNAME,-)$(Const(s,-))$-) = s
   |strip-sname  - = error(declaration part must consist of \
                     \two schema names only!)
val (state-abs,state-conc) = (case decl of
                  Const (op &, -)$H$H' => (strip-sname H,
                                           strip-sname H')
                | - => error(declaration part must consist of \
                            \two schema names only!))
val cont = FMethod.Mk{abs=SOME(abs,
                           (NameSpace.base state-abs),
                           (NameSpace.base state-conc)),
                 functional-refinement=functional}
val thy' = ZPO-Mgr.set-content cont thy

val -    = (ZEncoder.ZENV:=ZTheory.get-zenv thy');

val agoal= %A ^(NameSpace.base state-conc)^ • (%E1 ^
           (NameSpace.base state-abs)^ • ^(NameSpace.base abs)^);
val cf   =  cterm-of(sign-of thy')
                 (ZEncoder.schema-read thy' agoal)

val thy''=  if functional
           then ZPO-Mgr.declare-PO
                  (SOME(Context.theory-name thy'),
                   fwRefinementFunctional,abs)
                  cf thy'
           else thy'
val - =     if functional
           then writeln(PO declared: ^(the
                     (ZPO-Mgr.get-name-of-PO cf thy''))))
           else ()

in  thy'' end

fun refine-init(init-abs, init-conc)(thy) =
   let val - = if is-schema init-abs thy then ()
           else error(Must be schema name: ^init-abs)
```

```
val - = if is-schema init-conc thy then ()
        else error(Must be schema name: ^init-conc)
val FMethod.Mk{abs,functional-refinement} = ZPO-Mgr.get-content thy
val (abs, state-abs, state-conc)= case abs of
                                  NONE => errorno abstraction relation set!
                                | SOME x => x
val term-abs  = concl-of (get-thm thy (Name(init-abs^-def)))
val term-conc = concl-of (get-thm thy (Name(init-conc^-def)))
val term-abs  = ZEncoder.strip-Binder
              (case term-abs of
                 Const(==,-) $ - $ t => t
               | - => error(init-abs^must denote init schema!))
val term-conc = ZEncoder.strip-Binder
              (case term-conc of
                 Const(==,-) $ - $ t => t
               | - => error(init-conc^must denote init schema!))
fun test m (Const(ZPure.DECL,-)
             $ (Const(ZPure.SNAME,-) $ Const(s,-) $-)
             $ -) =
               if (NameSpace.base s) = m then ()
               else error(imported schema must be :^m)
   |test m - = errorillegal pattern of import schema
val - = test state-abs  term-abs
val - = test state-conc term-conc
val goal = if functional-refinement
           then %A ^state-abs^ • (%A ^state-conc^ • (
                ^ init-conc ^ & ^abs^ --> ^init-abs^))
           else %A ^state-conc^ • (^ init-conc ^ --> ^
                (%E ^state-abs^ • (^abs^ & ^init-abs^)))
val cf  =  cterm-of(sign-of thy)
             (ZEncoder.schema-read thy goal)
val thy' =  ZPO-Mgr.declare-PO
              (SOME(Context.theory-name thy),
               fwRefinementInit,state-abs)
              cf thy
val -    = writeln(PO declared: ^(the
                 (ZPO-Mgr.get-name-of-PO cf thy')))
in  thy' end


fun refine-ops (op-abs, op-conc)     thy =
   let (* checking syntactic side-conditions ... *)
      val - = if is-schema op-abs thy then ()
```

```
        else error(Must be schema name: ^op-abs)
val - = if is-schema op-conc thy then ()
        else error(Must be schema name: ^op-conc)
val FMethod.Mk{abs,functional-refinement} = ZPO-Mgr.get-content thy
val SOME(abs, state-abs, state-conc)=abs
val term-abs  = concl-of (get-thm thy (Name(op-abs^-def)))
val term-conc = concl-of (get-thm thy (Name(op-conc^-def)))
val term-abs  = ZEncoder.strip-Binder
              (case term-abs of
                 Const(==,-) $ - $ t => t
               | - => error(op-abs^must denote op schema!))
val term-conc = ZEncoder.strip-Binder
              (case term-conc of
                 Const(==,-) $ - $ t => t
               | - => error(op-conc^must denote op schema!))
fun test m (Const(ZPure.DECL,-) $
           (Const (op &,-)
             $ (Const (ZPure.DELTA,-)
                $ (Const(ZPure.SNAME,-)$(Const(s,-))$-)
                $ -)
             $ -) $ -) =
             (if (NameSpace.base s) = m then ()
              else error(imported schema must be :^m))
  |test m (Const(ZPure.DECL,-) $
           (Const (op &,-)
             $ (Const (ZPure.DELTA,-)
                $ (Const (ZPure.DELTA,-)
                   $ (Const(ZPure.SNAME,-)$(Const(s,-))$-)
                   $ -)
                $ - $ -)
             $ -) $ -) =
             (if (NameSpace.base s) = m then ()
              else error(imported schema must be :^m))
  |test m -  = error(illegal pattern of schema:^m)
val - = test state-abs  term-abs
val - = test state-conc term-conc

fun zsig-of name =  schemasig-of(the(ZEnv.lookup
                    (schemas-of(ZTheory.get-zenv thy),name)))
fun get-params test name = filter test
                           (map fst (zsig-of name))
val in-parms  = get-params ZPrelude.is-in-name op-abs
val in-parms' = get-params ZPrelude.is-in-name op-conc
val -         = if in-parms = in-parms' then ()
```

```
                else error(Input parameters must agree !)
val out-parms = get-params ZPrelude.is-out-name op-abs
val out-parms'= get-params ZPrelude.is-out-name op-conc
val -         = if out-parms = out-parms' then ()
                else error(Output parameters must agree !)

(* generating proof obligations ... *)
fun ALL [] b  = (^b^)
  |ALL S  b  = (ALL ^(String.concat
                    (map(fn x => x^ )S))^. ^b^)

val goal1 = %A ^state-abs^ • (%A ^state-conc^ • ^
          (ALL in-parms
              (pre ^op-abs^ ∧ ^abs^
                ⟶ pre ^op-conc))^)
val goal2c= if functional-refinement
          then ∧ ^abs^' ⟶ ^op-abs
          else ⟶ (%E ^state-abs^' •
              ^abs^' ∧ ^op-abs^)
val goal2a= %A ^state-abs^ • (%A ^state-conc^ • ^
          (%A ^state-conc^' • ^
          (ALL in-parms
              (ALL out-parms (pre ^op-abs^ ∧ ^abs^
                              ∧ ^op-conc^
                              ⟶ (%E
                              ^state-abs^' • ^abs^
                              ' ∧ ^op-abs^)))) ^))
val goal2b= %A ^state-abs^ • (%A ^state-abs^
          ' • (%A ^state-conc^ • ^ (%A ^state-conc^' • ^
          (ALL in-parms
              (ALL out-parms (pre ^op-abs^ ∧ ^abs^
                              ∧ ^op-conc^
                              ∧ ^abs^
                              ' ⟶ ^op-abs
                              ))) ^)))
val goal2 = if functional-refinement then goal2b else goal2a
val cf1   = cterm-of(sign-of thy)
              (ZEncoder.schema-read thy goal1)
val cf2   = cterm-of(sign-of thy)
              (ZEncoder.schema-read thy goal2)
val thy'  = ZPO-Mgr.declare-PO
              (SOME(Context.theory-name thy),
               fwRefinementOp,op-abs)
              cf1 thy
```

```
        val thy'' = ZPO-Mgr.declare-PO
                      (SOME(Context.theory-name thy),
                       fwRefinementOp,op-abs)
                      cf2 thy'
        val -     = writeln(PO declared: ^(the
                          (ZPO-Mgr.get-name-of-PO cf1 thy''))
        val -     = writeln(PO declared: ^(the
                          (ZPO-Mgr.get-name-of-PO cf2 thy''))
    in  thy'' end



structure P = OuterParse and K = OuterKeyword;

val declare-absP =
   let fun declare-abs'(name,X) = declare-abs (isSome X) name
   in OuterSyntax.command set-abs set abstraction relation
                    OuterKeyword.thy-script
                    ((P.name) -- (Scan.option (P.$$$ [ |-- P.$$$ functional --|
P.$$$ ]))
                    >> (Toplevel.theory o declare-abs'))
   end;
val - =  OuterSyntax.add-parsers [declare-absP];
val - = OuterSyntax.add-keywords [functional];

val gen-tcc-fromP =
   OuterSyntax.command gen-thm-tcc generate type constraint side conditions
                    OuterKeyword.thy-script
                    ((P.name)  >> (Toplevel.theory o gen-tcc-from));
val - =  OuterSyntax.add-parsers [gen-tcc-fromP];


val gen-cc-stateP =
   OuterSyntax.command gen-state-cc generate state consistency conditions
                    OuterKeyword.thy-script
                    ((P.name)  >> (Toplevel.theory o gen-cc-state));
val - =  OuterSyntax.add-parsers [gen-cc-stateP];


val gen-cc-opP =
   OuterSyntax.command gen-op-cc generate operation consistency relation
                    OuterKeyword.thy-script
                    ((P.name)  >> (Toplevel.theory o gen-cc-op));
```

```
val - =  OuterSyntax.add-parsers [gen-cc-opP];


val refine-opsP =
   OuterSyntax.command refine-op generate forward refinement proof obligations
                    OuterKeyword.thy-script
                    ((P.name)  -- (P.name) >> (Toplevel.theory o refine-ops));
val - =  OuterSyntax.add-parsers [refine-opsP];

val refine-initP =
    OuterSyntax.command refine-init generate forward refinement proof obligations
for init schemas
                    OuterKeyword.thy-script
                    ((P.name)  -- (P.name) >> (Toplevel.theory o refine-init));
val - =  OuterSyntax.add-parsers [refine-initP];




end
```

⟫

end

6.11 Main Theory for the HOL-Z Environment

theory *Z*
imports *ZSeqtoList*
ZBag
ZPure
ZMethod

begin

6.11.1 Proof Support: The .holz-loader (generated by ZETA)

ML⟪

```
  val store-sec = ref(ZAbsy.ZSection(,[],[]));
```

```
fun register-unit (sec as (ZAbsy.ZSection(name,-,-)))
                schemas axdefs conjectures thy-name thy =
   let val zenv   = ZTheory.get-zenv thy
       val imprt'  = (thy-name::ZEnv.imports-of zenv)
       val zenv'   = ZEnv.update-Zenv-by-Imports zenv imprt'
       val zenv''  = ZEnv.update-Zenv-by-Unit zenv' (name,sec)
       val thy'     = fst(PureThy.add-thmss [((SCHEMAS,schemas),[])]
                                        thy)
       val thy''   = fst(PureThy.add-thmss [((AXDEFS, axdefs),[])]
                                       thy')
       val thy'''  = fst(PureThy.add-thmss [((CONJECTURES, conjectures),[])]
                                       thy'')
   in  ZTheory.put-zenv zenv'' thy''' end

fun check-imports (ZAbsy.ZSection(name,importS,itemS)) thy =
   let val thy-name = #Name(ZConversion.unitName name)
       val loaded-imports  =ZEnv.imports-of(ZTheory.get-zenv thy)
     val impnames = map(fn x=> (#Name(ZConversion.unitName x)))importS;
       (*Check if parents have already been loaded*)
   in  case filter(fn x => not(x mem loaded-imports))(impnames) of
         [] => thy-name
       | S  => error(Units: ^(concat S)^ not yet loaded!)
   end


fun load-holz thy-file thy  =
   let val intext  = (File.read (Path.unpack(thy-file^.holz)))
                  handle - => File.read(Path.unpack(
                                     holz/^thy-file^.holz));
       val sec     = (use-text Context.ml-output false
                            (store-sec := ^intext);
                        !store-sec)
       val -       = ZEncoder.init();
       val -       = ZConversion.printSec sec;
       val thy-name= check-imports sec thy;
       val thy'    = let val (ZAbsy.ZSection(-,-,itemS)) = sec
                  in  foldl (ZConversion.convItem thy-name)
                            thy itemS
                  end
       fun thm x   = get-thm thy' (Name x)
       val thy''   = register-unit
                           sec
                           (map thm (!ZEncoder.SCHEMANAMES))
                           (map thm (!ZEncoder.AXDEFS))
```

```
                                (map thm (!ZEncoder.CONJECTURES))
                                thy-name
                                thy'
          val -        = (ZEncoder.ZENV := (ZTheory.get-zenv thy''))
      in  thy''  end;
⟫
```

```
ML⟪
val load-holzP =
    OuterSyntax.command load-holz load ZETA−generated .holz file
                        OuterKeyword.thy-script
                        ((OuterParse.name)  >> (Toplevel.theory o load-holz));

val - =  OuterSyntax.add-parsers [load-holzP];

⟫
```

6.11.2 Proof Support: Toplevel-Command zlemma

```
ML⟪
val zlemmaK = zlemma

val statement = OuterParse.and-list1
              (OuterParse.opt-thm-name :
               −− Scan.repeat1 OuterParse.propp);
val general-statement = statement >> pair [] ||
                        Scan.repeat OuterParse.locale-elem-or-expr
                        −− (OuterParse.$$$ shows |−− statement);

⟫
```

```
ML⟪
(∗ Old:  (does not work but reports parameter correctly . . . )

   (∗ For Debugging ∗)

val X = ref (NONE : xstring option) ;
val Y = ref((,[]):string ∗ Attrib.src list);
val Z = ref([]:Locale.element Locale.elem-expr list);
val W = ref([]:((string ∗ Attrib.src list)
            ∗ (string ∗ (string list ∗ string list)) list) list);
```

```
fun smart-thm x y z w thy =
    let val - = (X:=x)
        val - = (Y:=y)
        val - = (Z:=z)
        val - = (W:=w)
    in  case w of
        [((lemma-name,attr),[(goal,([],[]))])] =>
            let val H     = !ZEncoder.ZENV
                val -     = (ZEncoder.ZENV:= ZTheory.get-zenv thy)
                val cgoal = (ZEncoder.schema-read thy goal)
                val -     = (ZEncoder.ZENV:= H )
                val w'    = [((lemma-name,attr),[(goal,([],[]))])]
            in  (* Locale.smart-theorem zlemmaK x y z w thy *)
                (Proof.theorem zlemmaK (K (K I)) (SOME ) y w o
                 ProofContext.init) thy
            end
      | - => error zlemma : wrong argument pattern !
    end;


*)

fun smart-thm x y z w thy =
    let
    in  case w of
        [((lemma-name,attr),[(goal,([],[]))])] =>
            let val H     = !ZEncoder.ZENV
                val -     = (ZEncoder.ZENV:= ZTheory.get-zenv thy)
                val cgoal = (ZEncoder.schema-read thy goal)
                val -     = (ZEncoder.ZENV:= H )
                val y'    = (fst y,[])
                val w'    = [((lemma-name,[]),[(cgoal,([],[]))])]
            in  (* Locale.smart-theorem zlemmaK x y z w thy *)
                (Proof.theorem-i zlemmaK (K (K I)) (SOME ) y' w' o
                 ProofContext.init) thy
            end
      | - => error zlemma : wrong argument pattern !
    end;

val gen-zlemmaP =
  OuterSyntax.command zlemmaK (state ^ zlemmaK) OuterKeyword.thy-goal
    (OuterParse.opt-locale-target -- Scan.optional
      (OuterParse.opt-thm-name : --|
```

```
    Scan.ahead (OuterParse.locale-keyword || OuterParse.$$$ shows))
      (, []) --
  general-statement >> (fn ((x, y), (z, w)) =>
  (Toplevel.print o Toplevel.theory-to-proof (smart-thm x y z w))));


val - =   OuterSyntax.add-parsers [gen-zlemmaP];
```

⟫

6.11.3 Baustelle: Code von DARMA zum integrieren

lemmas *dom-infers* = *dom-override-I dom-Un-I dom-insert-I1 dom-insert-I2 dom-dres-I dom-BagSingleton-I*

lemmas *type-infers* = *SigmaI subsetI* [*THEN PowI*] *pair-rel-dom-fst pair-rel-dom*

declare *UNIV-I* [*tc-simp*]
declare *dom-infers* [*tc-simp*]
declare *type-infers*[*tc-simp*]

ML⟪

```
val tfun-apply-fst = thmtfun-apply-fst
val tfun-apply-snd = thmtfun-apply-snd
val tfun-apply = thmtfun-apply
val Dom-Partial-Fin=thmDom-Partial-Fin
val pfun-apply-fst=thmpfun-apply-fst
val pfun-apply-snd=thmpfun-apply-snd
val pfun-apply=thmpfun-apply

val mapfilter = Library.mapfilter


val mapfilter = Library.mapfilter
fun Prf y = (fn x => SOME(x RS y) handle THM - => NONE)

fun fun-type-infers axdefs =
```

```
    (mapfilter (Prf tfun-apply-fst) axdefs) •
    (mapfilter (Prf tfun-apply-snd)  axdefs) •
    (mapfilter (Prf tfun-apply)      axdefs) •
    (mapfilter (Prf Dom-Partial-Fin) axdefs) •
    (mapfilter (Prf pfun-apply-fst)  axdefs) •
    (mapfilter (Prf pfun-apply-snd)  axdefs) •
    (mapfilter (Prf pfun-apply)      axdefs);

fun Add-axdefs-TC axdef =
    let fun flat-conj m = Library.flat (mapfilter (
                            fn x          =>SOME(flat-conj[x RS conjunct1,
                                                           x RS conjunct2])
                              handle THM - =>SOME[x]) m)
        fun closure m   = m • (fun-type-infers m)
    in  Add-TC (Library.flat(mapfilter
                  (fn x => SOME(closure(flat-conj[x RS DECL-D1]))
                           handle THM - => NONE)
                           axdef))

    end;
```

⟫

lemmas *oplus-def* = *override-def*
lemmas *oplus-I2* = *overrideI2*
lemmas *oplus-I1* = *overrideI1*
lemmas *oplus-CI* = *overrideCI*
lemmas *oplus-single* = *override-single*
lemmas *oplus-res-right* = *override-res-right*
lemmas *oplus-res-left* = *override-res-left*
lemmas *oplus-Inter* = *override-inter*
lemmas *oplus-idem* = *override-idem*
lemmas *oplus-mt-right* = *override-mt-right*
lemmas *oplus-mt-left* = *override-mt-left*
lemmas *oplus-Domain* = *override-Domain*
lemmas *oplus-comp* = *override-assoc*
lemmas *oplus-fpfun* = *override-fpfun*
lemmas *oplus-pfunI* = *override-pfunI*

lemmas *oplus-apply2* = *override-apply2*
lemmas *oplus-apply* = *override-apply*

lemmas *oplus-apply1* = *override-apply1*
lemmas *oplus-non-apply*= *override-by-pair-apply2*
lemmas *oplus-by-pair-apply1* = *override-by-pair-apply1*
lemmas *oplus-by-pair-apply2* = *override-by-pair-apply2*

lemmas *Rel-Apply-in-total-range* = *tfun-apply*
lemmas *partial-func-def* = *pfun-def*
lemmas *total-func-def* = *tfun-def*

lemmas *pair-mem-apply* = *beta-apply-pfun*
lemmas *Rel-Apply-in-Partial-Ran2* = *pfun-apply*
lemmas *total-func-implies-Pfun* = *tfun-implies-pfun*

lemmas *partial-fun-ran-subset* = *pfun-ran-subset*

lemmas *partial-fun-dom-subset* = *pfun-subset*

lemmas *dom-oplus-I* = *dom-override-I*

lemmas *empty-pfun* = *empty-is-pfun*

lemmas *dom-insert-apply* = *insert-apply*
lemmas *dom-insert-apply2* = *insert-apply2*

end