# Tables

Wolfram Kahl

August 20, 2003

## Contents


**1 Utilities** **3**

1.1 Functions . . . 3

1.1.1 Combinators . . . 3

1.1.2 Currying . . . 3

1.1.3 Pair Manipulation . . . 4

1.1.4 Extensionality . . . 4

1.2 Function Properties . . . 4

1.2.1 Associativity . . . 4

1.2.2 Idempotence . . . 4

1.2.3 Commutativity . . . 5

1.2.4 Units . . . 5

1.3 Additional Material for Lists . . . 6

1.4 Option Utilities . . . 7

1.4.1 The Option Monad . . . 7

1.4.2 Equality Option Collapse . . . 8

**2 Non-Empty Lists** **9**

2.1 Datatype Definition . . . 9

2.2 Construction . . . 9

2.3 foldr1 . . . 12

2.4 map . . . 13

2.5 Induction . . . 14

2.6 Elements . . . 16

2.7 neFold . . . 17

2.8 ZipWith . . . 17

2.9 Other Properties . . . 19

**3 The Table Datatype Definition** **19**

3.1 Datatype and Primitive Operations . . . 19

3.2 Folding and Induction via hConc . . . 20

3.3 hCons . . . 20

**4 Table Utilities and Properties** **22**
4.1 Additional Properties of *tFold* . . . 22
4.2 tFoldC . . . 23
4.3 hHead . . . 23
4.4 The “Cons View” . . . 24
4.5 Mapping . . . 26
4.6 Headers and Subtables . . . 27
4.7 Regular Skeletons . . . 29
4.8 Two-Dimensional Regular Tables . . . 32
4.9 List Interface . . . 33
4.10 ZipWith . . . 33
4.11 Collapsing . . . 35
4.12 Compression . . . 36
4.13 Elementary Transformations . . . 36
4.13.1 Permuting two $(-1)$-slices with their corresponding header entries . . . 36
4.13.2 Deleting a $(-1)$-slice with “false” in the corresponding header entry . . . 37
4.13.3 Deleting a principal slice with only “false” entries from an inverted table . . . 37
4.13.4 Splitting a principal slice by “splitting a disjunction” in the corresponding header in an inverted table . . . 37
4.13.5 Combining two or more principal slices with the same value header entry into a single slice in an inverted table 38

**5 Functions Interacting Directly with the Second Table Dimension** **38**
5.1 Construction in the Second Dimension . . . 38
5.2 Regular Skeletons and the Second Dimension . . . 42
5.3 Table Transposition . . . 47

**6 Inversion of Normal Tables** **54**
6.1 One-Dimensional Inversion . . . 54
6.2 spread1 . . . 55
6.3 spread2 . . . 56
6.4 delH1 . . . 57
6.5 delH2 . . . 59
6.6 Third Dimension Operators . . . 60
6.7 Slimming Operators . . . 61
6.8 Right-Updating Horizontal Concatenation . . . 63
6.9 Diagonal Table Concatenation . . . 63
6.10 Lifting of Inversion Combinators to the Next Dimension . . . 72
6.11 The Inversion Operator for Two Dimensions . . . 81
6.12 Two-Dimensional Inversion . . . 83

# 1 Utilities

**theory** *Utils* = *Main*:

The following variant of modus ponens is useful as erule, for example for preparing local inductions.

**lemma** *mp1*: $[\![ P; P \longrightarrow Q ]\!] \Longrightarrow Q$
**by** *simp*

## 1.1 Functions

### 1.1.1 Combinators

**consts** *const* :: $'a \Rightarrow 'b \Rightarrow 'a$
**defs** *const-def*: *const* == % *x y* . *x*

**lemma** *const*[*simp*]: *const x y* = *x*
**by** (*simp add*: *const-def*)

**lemma** *const-expand*: *const c* = (% *x* . *c*)
**by** (*rule ext*, *simp*)

**consts** *flip* :: $('a \Rightarrow 'b \Rightarrow 'c) \Rightarrow ('b \Rightarrow 'a \Rightarrow 'c)$
**defs** *flip-def*[*simp*]: *flip f x y* == *f y x*

### 1.1.2 Currying

**constdefs**
  *curry* :: $(('a * 'b) \Rightarrow 'c) \Rightarrow 'a \Rightarrow 'b \Rightarrow 'c$
  *curry f* == % *a b* . *f* (*a*,*b*)
  *uncurry* :: $('a \Rightarrow 'b \Rightarrow 'c) \Rightarrow ('a * 'b) \Rightarrow 'c$
  *uncurry g* == % *p* . *g* (*fst p*) (*snd p*)

**lemma** *curry*[*simp*]: *curry f a b* = *f* (*a*,*b*)
**by** (*simp add*: *curry-def*)

**lemma** *uncurry*[*simp*]: *uncurry g* (*a*,*b*) = *g a b*
**by** (*simp add*: *uncurry-def*)

**lemma** *uncurry-lambda2*[*simp*]: *uncurry* ($\lambda h\ t.\ F\ h\ t$) = ($\lambda p$ . *F* (*fst p*) (*snd p*))
**by** (*simp add*: *uncurry-def*)

**lemma** *uncurry-K2*[*simp*]: *uncurry* ($\lambda h\ t.\ F\ h$) = ($\lambda p$ . *F* (*fst p*))
**by** *simp*

**lemma** *uncurry-K*[*simp*]: *uncurry* ($\lambda h.\ F$) = ($\lambda p$ . *F* (*snd p*))
**by** *simp*

### 1.1.3 Pair Manipulation

**constdefs**
$pupd1 :: ('a \Rightarrow 'b) \Rightarrow ('a * 'c) \Rightarrow ('b * 'c)$
$pupd1\ f\ p == (f\ (fst\ p),\ snd\ p)$
$pupd2 :: ('a \Rightarrow 'b) \Rightarrow ('c * 'a) \Rightarrow ('c * 'b)$
$pupd2\ g\ p == (fst\ p,\ g\ (snd\ p))$

**lemma** *pupd1*[*simp*]: $pupd1\ f\ (x,y) = (f\ x,\ y)$
**by** (*simp add*: *pupd1-def*)

**lemma** *pupd2*[*simp*]: $pupd2\ g\ (x,y) = (x,\ g\ y)$
**by** (*simp add*: *pupd2-def*)

### 1.1.4 Extensionality

I had some problems applying *HOL.ext* directly.

**lemma** *f-ext*:
**assumes** *eq*: $\bigwedge x\ .\ f\ x = g\ x$
**shows** $f = g$
**by** (*rule HOL.ext*)

## 1.2 Function Properties

### 1.2.1 Associativity

**constdefs**
$assoc :: ('a \Rightarrow 'a \Rightarrow 'a) \Rightarrow bool$
$assoc\ f == (ALL\ x\ y\ z\ .\ f\ (f\ x\ y)\ z = f\ x\ (f\ y\ z))$

**lemma** *assoc*[*simp*]:
$assoc\ f \Longrightarrow f\ (f\ x\ y)\ z = f\ x\ (f\ y\ z)$
**by** (*unfold assoc-def*, *auto*)

**lemma** *assoc-intro*[*intro?*]:
$[\![ \bigwedge x\ y\ z\ .\ f\ (f\ x\ y)\ z = f\ x\ (f\ y\ z)\ ]\!] \Longrightarrow assoc\ f$
**by** (*unfold assoc-def*, *auto*)

**lemma** *const-assoc*[*simp*]: *assoc const*
**by** (*rule assoc-intro*, *simp*)

### 1.2.2 Idempotence

**constdefs**
$idempotent :: ('a \Rightarrow 'a \Rightarrow 'a) \Rightarrow bool$
$idempotent\ f == (ALL\ x\ .\ f\ x\ x = x)$

**lemma** *idempotent*[*simp*]:

```
  idempotent f ⟹ f x x = x
by (unfold idempotent-def, auto)

lemma idempotent-intro[intro?]:
  ⟦ ⋀ x . f x x = x ⟧ ⟹ idempotent f
by (unfold idempotent-def, auto)
```

### 1.2.3 Commutativity

```
constdefs
  commutative :: ('a ⇒ 'a ⇒ 'a) ⇒ bool
  commutative f == (ALL x y . f x y = f y x)

lemma commutative:
  commutative f ⟹ f x y = f y x
by (unfold commutative-def, auto)

lemma commutative-intro[intro?]:
  ⟦ ⋀ x y . f x y = f y x ⟧ ⟹ commutative f
by (unfold commutative-def, auto)
```

### 1.2.4 Units

```
consts LRunit :: ('a ⇒ 'a ⇒ 'a) ⇒ 'a ⇒ 'bool

defs LRunit-def: LRunit f u == ALL y . f u y = y & f y u = y

lemma LRunit-left: LRunit f u ⟹ f u x = x
by (unfold LRunit-def, simp)

lemma LRunit-right: LRunit f u ⟹ f x u = x
by (unfold LRunit-def, simp)

lemma LRunit-equal:
  assumes u[intro,simp]: LRunit f u
  assumes v[intro,simp]: LRunit f v
  shows u = v
proof −
  have u = f u v by (rule sym, rule LRunit-right, simp)
  also have f u v = v by (rule LRunit-left, simp)
  finally show ?thesis .
qed

lemma LRunit-const:
 ALL x . LRunit c (F x) ⟹ F x = F arbitrary
apply (subgoal-tac ALL x y . F x = F y, simp)
  apply (intro strip)
  apply (rule LRunit-equal [of c])
apply simp-all
done
```

### 1.3 Additional Material for Lists

**lemma** *foldr-append*:
*foldr f* (*xs* @ *ys*) *z* = *foldr f xs* (*foldr f ys z*)
**by** (*induct-tac xs*, *auto*)

**consts**
*foldr-1* :: ('*a* ⇒ '*a* ⇒ '*a*) ∗ '*a list* ⇒ '*a*

**recdef** *foldr-1 measure* (% (*f*,*xs*) . *length xs*)
*foldr-1-sing*[*simp*]: *foldr-1* (*f*, *x* # []) = *x*
*foldr-1-cons0*[*simp*]: *foldr-1* (*f*, *x* # *y* # *ys*) = *f x* (*foldr-1* (*f*, *y* # *ys*))

**lemma** *foldr-1-cons*[*simp*]:
⟦ *xs* ~= [] ⟧ ⟹ *foldr-1* (*f*, *x* # *xs*) = *f x* (*foldr-1* (*f*, *xs*))
**by** (*cases xs*, *auto*)

**lemma** *foldr-1-append-distr-0*:
*assoc f* ⟹
(*ALL x ys* . *ys* ~= [] ⟶ *foldr-1* (*f*, *x* # *xs* @ *ys*) = *f* (*foldr-1* (*f*, *x* # *xs*))
(*foldr-1* (*f*, *ys*)))
**by** (*induct-tac xs*, *auto*)

**lemma** *foldr-1-append-distr*:
**assumes** *xs*[*simp*]: *xs* ~= []
**assumes** *ys*[*simp*]: *ys* ~= []
**assumes** *f*[*simp*]: *assoc f*
**shows** *foldr-1* (*f*, *xs* @ *ys*) = *f* (*foldr-1* (*f*, *xs*)) (*foldr-1* (*f*, *ys*))
**proof** (*cases xs*)
**case** *Nil*
**thus** *?thesis* **by** *simp*
**next**
**case** *Cons*
**thus** *?thesis* **by** (*insert foldr-1-append-distr-0* [*of f tl xs*], *simp*)
**qed**

**lemma** *mem-append*[*simp*]:
*x mem* (*xs* @ *ys*) = (*x mem xs* | *x mem ys*)
**by** (*induct-tac xs*, *simp-all*)

**constdefs**
*zipWith* :: ('*a* ⇒ '*b* ⇒ '*c*) ⇒ '*a list* ⇒ '*b list* ⇒ '*c list*
*zipWith f xs ys* == *map* (*uncurry f*) (*zip xs ys*)

**lemma** *zipWith-Cons*[*simp*]:
*zipWith f* (*x*#*xs*) (*y*#*ys*) = *f x y* # *zipWith f xs ys*
**by** (*simp add*: *zipWith-def*)

**lemma** *zipWith-Nil-1*[*simp*]:
*zipWith f* [] *ys* = []

**by** (*simp add*: *zipWith-def*)

**lemma** *zipWith-Nil-2*[*simp*]:
*zipWith f xs* [] = []
**by** (*simp add*: *zipWith-def*)

**lemma** *zipWith-assoc-0*:
*assoc f* ⟹ *ALL ys zs . zipWith f* (*zipWith f xs ys*) *zs* = *zipWith f xs* (*zipWith f ys zs*)
**apply** (*induct-tac xs*, *simp*)
**apply** (*intro strip*)
**apply** (*induct-tac ys*, *simp*)
**apply** (*induct-tac zs*, *simp*)
**apply** *simp*
**done**

**lemma** *zipWith-assoc*[*simp*]:
*assoc f* ⟹ *zipWith f* (*zipWith f xs ys*) *zs* = *zipWith f xs* (*zipWith f ys zs*)
**by** (*insert zipWith-assoc-0*, *auto*)

**lemma** *assoc-zipWith*[*simp*]: *assoc f* ⟹ *assoc* (*zipWith f*)
**by** (*rule assoc-intro*, *simp*)

## 1.4 Option Utilities

**consts**
*option* :: *'b* ⇒ (*'a* ⇒ *'b*) ⇒ *'a option* ⇒ *'b*

**primrec**
*option-N*: *option r f None* = *r*
*option-S*: *option r f* (*Some a*) = *f a*

### 1.4.1 The Option Monad

**consts**
*optThen* :: *'a option* ⇒ (*'a* ⇒ *'b option*) ⇒ *'b option*

**primrec**
*optThen-N*: *optThen None* = (% *f* . *None*)
*optThen-S*: *optThen* (*Some x*) = (% *f* . *f x*)

**declare** *optThen-N*[*simp del*]
**declare** *optThen-S*[*simp del*]

**lemma** *optThen-N-f*[*simp*]: *optThen None f* = *None*
**by** (*simp add*: *optThen-N*)

**lemma** *optThen-S-f*[*simp*]: *optThen* (*Some x*) *f* = *f x*
**by** (*simp add*: *optThen-S*)

**lemma** *optThen-result-Some*[*simp*]:
*optThen m f* = *Some x* ⟹ *EX y* . *m* = *Some y* & *f y* = *Some x*
**by** (*cases m*, *simp-all*)

**lemma** *optThen-option-map*[*simp*]:
*optThen* (*option-map f x*) *g* = *optThen x* (*g o f*)
**by** (*cases x*, *simp-all*)

**lemma** *optThen-assoc*[*simp*]:
*optThen* (*optThen x f*) *g* = *optThen x* (% *x* . *optThen* (*f x*) *g*)
**by** (*cases x*, *simp-all*)

### 1.4.2 Equality Option Collapse

**consts**
*optEq* :: *'a option* ⇒ *'a option* ⇒ *'a option*

**primrec**
*optEq-N*: *optEq None* = (% *my* . *None*)
*optEq-S*: *optEq* (*Some x*) = *option None* (% *y* . *if* (*x*=*y*) *then Some x else None*)

**declare** *optEq-N*[*simp del*]
**declare** *optEq-S*[*simp del*]

**lemma** *optEq-S-S*[*simp*]:
*optEq* (*Some x*) (*Some y*) = (*if* (*x*=*y*) *then Some x else None*)
**by** (*simp add*: *optEq-S*)

**lemma** *optEq-S-N*[*simp*]: *optEq* (*Some x*) *None* = *None*
**by** (*simp add*: *optEq-S*)

**lemma** *optEq-N-S*[*simp*]: *optEq None* (*Some x*) = *None*
**by** (*simp add*: *optEq-N*)

**lemma** *optEq-N-N*[*simp*]: *optEq None None* = *None*
**by** (*simp add*: *optEq-N*)

**lemma** *optEq-assoc*[*simp*]: *optEq* (*optEq a b*) *c* = *optEq a* (*optEq b c*)
**apply** (*cases a*, *simp add*: *optEq-N*)
**apply** (*cases b*, *simp add*: *optEq-N*)
**apply** (*cases c*, *simp add*: *optEq-N*)
**apply** *simp*
**done**

**lemma** *assoc-optEq*[*simp*]: *assoc optEq*
**by** (*rule assoc-intro*, *simp*)

**lemma** *optEq-idempotent*[*simp*]: *idempotent optEq*
**by** (*rule idempotent-intro*, *case-tac x*, *simp-all*)

**lemma** *optEq-commutative*[*simp*]: *commutative optEq*
**apply** (*rule commutative-intro*)
**apply** (*case-tac x*, *case-tac y*, *simp-all add*: *optEq-N optEq-S*)
**apply** (*case-tac y*, *simp-all add*: *optEq-N optEq-S*)
**done**

**end**

# 2 Non-Empty Lists

**theory** *NEList* = *Utils*:

## 2.1 Datatype Definition

**typedef** *'a neList* = { *xs* :: *'a list* . *xs* ~= [] }
**by** *auto*

**lemma** *Rep-neList-non-Nil*[*simp*]: *Rep-neList x* ~= []
**by** (*insert Rep-neList* [*of x*], *simp add*: *neList-def*)

**lemma** *neList-append*[*simp*,*intro?*]:
⟦ *xs* : *neList*; *ys* : *neList* ⟧ ⟹ (*xs* @ *ys*) : *neList*
**by** (*simp add*: *neList-def*)

**lemma** *neList-map*[*simp*,*intro?*]:
⟦ *xs* : *neList* ⟧ ⟹ (*map f xs*) : *neList*
**by** (*simp add*: *neList-def*)

**lemma** *neList-zipWith*[*simp*,*intro?*]:
⟦ *xs* : *neList*; *ys* : *neList* ⟧ ⟹ (*zipWith f xs ys*) : *neList*
**apply** (*auto simp add*: *neList-def*)
**apply** (*cases xs*, *simp*)
**apply** (*cases ys*, *simp-all*)
**done**

## 2.2 Construction

**constdefs**
*singleton* :: *'a* ⇒ *'a neList*
*singleton x* == *Abs-neList* [*x*]
*append* :: *'a neList* ⇒ *'a neList* ⇒ *'a neList*
*append xs ys* == *Abs-neList* (*Rep-neList xs* @ *Rep-neList ys*)
*cons1* :: *'a* ⇒ *'a neList* ⇒ *'a neList*
*cons1 x xs* == *Abs-neList* (*x* # *Rep-neList xs*)

**lemma** *singleton-inj*: *singleton x = singleton y* $\Longrightarrow$ *x = y*
**apply** (*simp add*: *singleton-def Abs-neList-inverse*)
**apply** (*subgoal-tac* [*x*] = [*y*])
  **prefer** *2*
  **apply** (*subst Abs-neList-inject* [*THEN sym*])
  **apply** (*simp add*: *neList-def*)
  **apply** (*simp add*: *neList-def*)
  **apply** *assumption*
**apply** *auto*
**done**

**lemma** *cons1*:
 *cons1 x xs = append* (*singleton x*) *xs*
**apply** (*unfold cons1-def*)
**apply** (*unfold append-def*)
**apply** (*unfold singleton-def*)
**apply** (*simp-all add*: *Abs-neList-inverse Rep-neList-inverse neList-def*)
**done**

**lemma** *cons1-inj*: *cons1 x xs = cons1 y ys* $\Longrightarrow$ *x = y* & *xs = ys*
**apply** (*simp add*: *cons1-def Abs-neList-inverse*)
**apply** (*subgoal-tac x # Rep-neList xs = y # Rep-neList ys*)
  **prefer** *2*
  **apply** (*subst Abs-neList-inject* [*THEN sym*])
  **apply** (*simp add*: *neList-def*)
  **apply** (*simp add*: *neList-def*)
  **apply** *assumption*
**apply** *auto*
**apply** (*subst Rep-neList-inject* [*THEN sym*])
**apply** *assumption*
**done**

**lemma** *cons1-neq-tl*[*simp*]: *cons1 x xs ~= xs*
**apply** (*simp add*: *cons1-def singleton-def Abs-neList-inverse*)
**apply** (*cases Rep-neList xs*)
**apply** (*auto simp add*: *neList-def Abs-neList-inverse*)
**done**

**lemma** *cons1-neq-singleton*[*simp*]: *cons1 x xs ~= singleton y*
**apply** (*simp add*: *cons1-def singleton-def Abs-neList-inverse*)
**apply** (*subst Abs-neList-inject*)
**apply** (*auto simp add*: *neList-def*)
**done**

**lemma** *singleton-neq-cons1*[*simp*]: *singleton y ~= cons1 x xs*
**apply** (*simp add*: *cons1-def singleton-def Abs-neList-inverse*)
**apply** (*subst Abs-neList-inject*)
**apply** (*auto simp add*: *neList-def*)
**apply** (*cases Rep-neList xs*)

**apply** (*auto simp add*: *neList-def*)
**done**

**lemma** *append-neq-singleton*[*simp*]: *append xs ys* ~= *singleton x*
**apply** (*simp add*: *append-def singleton-def Abs-neList-inverse*)
**apply** (*subst Abs-neList-inject*)
**apply** (*auto simp add*: *neList-def*)
**apply** (*cases xs*)
**apply** (*cases ys*)
**apply** (*simp add*: *neList-def Abs-neList-inverse*)
**apply** (*case-tac ya*, *simp*)
**apply** (*case-tac y*, *simp-all*)
**done**

**lemma** *singleton-neq-append*[*simp*]: *singleton x* ~= *append xs ys*
**apply** (*simp add*: *append-def singleton-def Abs-neList-inverse*)
**apply** (*subst Abs-neList-inject*)
**apply** (*auto simp add*: *neList-def*)
**apply** (*cases xs*)
**apply** (*cases ys*)
**apply** (*simp add*: *neList-def Abs-neList-inverse*)
**apply** (*case-tac ya*, *simp*)
**apply** (*case-tac y*, *simp-all*)
**done**

**lemma** *append-inj2-0*: (*append xs ys* = *append xs zs*) = (*ys* = *zs*)
**apply** (*simp add*: *append-def Abs-neList-inverse*)
**apply** (*subst Abs-neList-inject*)
**apply** (*auto simp add*: *neList-def Rep-neList-inject* [*THEN iffD1*])
**done**

**lemma** *append-inj2*: *append xs ys* = *append xs zs* ⟹ *ys* = *zs*
**by** (*rule append-inj2-0* [*THEN iffD1*])

**lemma** *append-inj2-neq*: *append xs ys* ~= *append xs zs* ⟹ *ys* ~= *zs*
**by** (*insert append-inj2-0*, *auto*)

**lemma** *append-inj2-conv*: *ys* = *zs* ⟹ *append xs ys* = *append xs zs*
**by** (*insert append-inj2-0*, *auto*)

**lemma** *append-inj2-neq-conv*: *ys* ~= *zs* ⟹ *append xs ys* ~= *append xs zs*
**by** (*insert append-inj2-0*, *auto*)

**lemma** *append-inj1-0*: (*append xs zs* = *append ys zs*) = (*xs* = *ys*)
**apply** (*simp add*: *append-def Abs-neList-inverse*)
**apply** (*subst Abs-neList-inject*)
**apply** (*auto simp add*: *neList-def Rep-neList-inject* [*THEN iffD1*])
**done**

**lemma** *append-inj1*: *append xs zs* = *append ys zs* ⟹ *xs* = *ys*
**by** (*rule append-inj1-0* [*THEN iffD1*])

**lemma** *append-inj1-neq*: *append xs zs* ~= *append ys zs* ⟹ *xs* ~= *ys*
**by** (*insert append-inj1-0*, *auto*)

**lemma** *append-assoc*[*simp*]:
*append* (*append xs ys*) *zs* = *append xs* (*append ys zs*)
**apply** (*unfold append-def*)
**apply** (*cases xs*)
**apply** (*cases ys*)
**apply** (*cases zs*)
**apply** (*subgoal-tac y* @ *ya* : *neList*)
**apply** (*subgoal-tac ya* @ *yb* : *neList*)
**apply** (*simp add*: *Abs-neList-inverse*)
**apply** (*unfold neList-def*, *simp-all*)
**done**

**lemma** *assoc-append*[*simp*]: *assoc append*
**by** (*rule assoc-intro*, *simp*)

## 2.3 foldr1

**constdefs**
*foldr1* :: (′*a* ⇒ ′*a* ⇒ ′*a*) ⇒ ′*a neList* ⇒ ′*a*
*foldr1 f xs* == *foldr-1* (*f*, *Rep-neList xs*)

**lemma** *foldr1-singleton*[*simp*]:
*foldr1 f* (*singleton x*) = *x*
**apply** (*unfold singleton-def*)
**apply** (*unfold foldr1-def*)
**apply** (*simp-all add*: *Abs-neList-inverse Rep-neList-inverse neList-def*)
**done**

**lemma** *foldr1-cons1*[*simp*]:
*foldr1 f* (*cons1 x xs*) = *f x* (*foldr1 f xs*)
**apply** (*unfold cons1-def*)
**apply** (*unfold foldr1-def*)
**apply** (*simp-all add*: *Abs-neList-inverse Rep-neList-inverse neList-def*)
**done**

**lemma** *foldr1-append-distr*[*simp*]:
*assoc f* ⟹ *foldr1 f* (*append xs ys*) = *f* (*foldr1 f xs*) (*foldr1 f ys*)
**apply** (*cases xs*)
**apply** (*cases ys*)
**apply** (*unfold append-def*)
**apply** (*unfold foldr1-def*)
**apply** (*cases Rep-neList xs*)
**prefer** *2*

**apply** (*cases Rep-neList ys*)
**apply** (*simp-all add*: *Abs-neList-inverse Rep-neList-inverse neList-def*
*del*: *foldr-1-cons*)
**apply** (*insert foldr-1-append-distr* [*of Rep-neList xs Rep-neList ys f*, *THEN sym*])
**apply** (*simp-all add*: *Abs-neList-inverse Rep-neList-inverse neList-def*
*del*: *foldr-1-cons*)
**done**

## 2.4 map

**constdefs**
*map1* :: (*'a* ⇒ *'b*) ⇒ *'a neList* ⇒ *'b neList*
*map1 f xs* == *Abs-neList* (*map f* (*Rep-neList xs*))

**lemma** *map1-singleton*[*simp*]:
*map1 f* (*singleton x*) = *singleton* (*f x*)
**apply** (*unfold singleton-def*)
**apply** (*unfold map1-def*)
**apply** (*simp-all add*: *Abs-neList-inverse Rep-neList-inverse neList-def*)
**done**

**lemma** *map1-cons1*[*simp*]:
*map1 f* (*cons1 x xs*) = *cons1* (*f x*) (*map1 f xs*)
**apply** (*unfold cons1-def*)
**apply** (*unfold map1-def*)
**apply** (*simp-all add*: *Abs-neList-inverse Rep-neList-inverse neList-def*)
**done**

**lemma** *map1-append-distr*[*simp*]:
*map1 f* (*append xs ys*) = *append* (*map1 f xs*) (*map1 f ys*)
**apply** (*cases xs*)
**apply** (*cases ys*)
**apply** (*unfold append-def*)
**apply** (*unfold map1-def*)
**apply** (*cases Rep-neList xs*)
**prefer** *2*
**apply** (*cases Rep-neList ys*)
**apply** (*simp-all add*: *Abs-neList-inverse Rep-neList-inverse neList-def*
*del*: *foldr-1-cons*)
**done**

**lemma** *map1-contract-comp*:
*map1 f* (*map1 g xs*) = *map1* (*f o g*) *xs*
**apply** (*unfold map1-def*)
**apply** (*cases xs*)
**apply** (*cases Rep-neList xs*)
**apply** (*simp-all add*: *Abs-neList-inverse Rep-neList-inverse neList-def map-compose*
*del*: *foldr-1-cons*)
**done**

**lemma** *map1-comp*:
(*map1 f*) *o* (*map1 g*) = *map1* (*f o g*)
**by** (*simp add*: *comp-def map1-contract-comp*)

## 2.5 Induction

**lemma** *neList-patterns*:
(*EX x* . *xs* = *singleton x*) | (*EX y ys* . *xs* = *append* (*singleton y*) *ys*)
**apply** (*cases xs*)
**apply** (*cases Rep-neList xs*)
**apply** (*simp-all add*: *Abs-neList-inverse Rep-neList-inverse neList-def*
*del*: *foldr-1-cons*)
**apply** (*cases tl* (*Rep-neList xs*))
**apply** (*unfold singleton-def*)
**apply** (*rule disjI1*)
**apply** (*rule exI*)
**apply** (*simp add*: *Abs-neList-inverse Rep-neList-inverse neList-def*
*del*: *foldr-1-cons*)
**apply** (*rule disjI2*)
**apply** (*rule-tac x=a* **in** *exI*)
**apply** (*rule-tac x=Abs-neList list* **in** *exI*)
**apply** (*unfold append-def*)
**apply** (*simp add*: *Abs-neList-inverse Rep-neList-inverse neList-def*
*del*: *foldr-1-cons*)
**done**

**lemma** *neList-list-induct*:
(∀ *x* . *P* (*singleton x*)) ⟶
(∀ *x y ys* . *P* (*Abs-neList* (*y* # *ys*)) ⟶ *P* (*Abs-neList* (*x* # *y* # *ys*))) ⟶
(∀ *z* . *P* (*Abs-neList* (*z* # *zs*)))
**apply** (*induct-tac zs*)
**apply** (*simp add*: *Abs-neList-inverse Rep-neList-inverse neList-def singleton-def*
*del*: *foldr-1-cons*)
**apply** (*intro strip*)
**apply** *simp*
**done**

**lemma** *neList-cons-induct-0*:
⟦⋀*x* . *P* (*singleton x*); ⋀*x xs*. *P xs* ⟹ *P* (*cons1 x xs*)⟧
⟹ *P zs*
**apply** (*insert neList-patterns* [*of zs*])
**apply** (*simp only*: *cons1*)
**apply** (*erule disjE*)
**apply** (*erule exE*)
**apply** *simp*
**apply** (*erule exE*)
**apply** (*erule exE*)
**apply** (*insert neList-list-induct* [*of P tl* (*Rep-neList zs*)])

**apply** *simp*
**apply** (*subgoal-tac* ∀ *x y ys*. *P* (*Abs-neList* (*y* # *ys*)) ⟶ *P* (*Abs-neList* (*x* # *y* # *ys*)))
  **prefer** *2*
  **apply** (*intro strip*)
  **apply** (*subgoal-tac P* (*singleton x*))
    **prefer** *2*
    **apply** *simp*
  **apply** (*subgoal-tac P* (*append* (*singleton x*) (*Abs-neList* (*ya* # *ysa*))))
    **prefer** *2*
    **apply** *simp*
  **apply** (*subgoal-tac Abs-neList* (*x* # *ya* # *ysa*) = *append* (*singleton x*) (*Abs-neList* (*ya* # *ysa*)))
    **prefer** *2*
    **apply** (*simp* (*no-asm-simp*) *add*: *Abs-neList-inverse Rep-neList-inverse neList-def singleton-def append-def*
                *del*: *foldr-1-cons*)
  **apply** *simp*
**apply** *simp*
**apply** (*drule-tac x=y* **in** *spec*)
**apply** (*simp add*: *Abs-neList-inverse Rep-neList-inverse neList-def singleton-def append-def*
                *del*: *foldr-1-cons*)
**done**

**lemma** *neList-cons-induct*:
⋀ *zs* . ⟦⋀*x* . *P* (*singleton x*); ⋀*x xs*. *P xs* ⟹ *P* (*cons1 x xs*)⟧
  ⟹ *P zs*
**by** (*rule neList-cons-induct-0*, *auto*)

**lemma** *neList-cons-inductA*:
⟦⋀*x* . *P* (*singleton x*); ⋀*x xs*. *P xs* ⟹ *P* (*cons1 x xs*)⟧
  ⟹ *ALL zs* . *P zs*
**by** (*intro strip*, *rule neList-cons-induct*, *simp-all*)

**lemma** *neList-append-induct-0*:
⟦⋀*x* . *P* (*singleton x*); ⋀*xs ys*. *P xs* ⟹ *P ys* ⟹ *P* (*append xs ys*)⟧
  ⟹ *P zs*
**apply** (*insert neList-patterns* [*of zs*])
**apply** (*erule disjE*)
**apply** (*erule exE*)
**apply** *simp*
**apply** (*erule exE*)
**apply** (*erule exE*)
**apply** (*insert neList-list-induct* [*of P tl* (*Rep-neList zs*)])
**apply** *simp*
**apply** (*subgoal-tac* ∀ *x y ys*. *P* (*Abs-neList* (*y* # *ys*)) ⟶ *P* (*Abs-neList* (*x* # *y* # *ys*)))
  **prefer** *2*

**apply** (*intro strip*)
**apply** (*subgoal-tac P* (*singleton x*))
**prefer** *2*
**apply** *simp*
**apply** (*subgoal-tac P* (*append* (*singleton x*) (*Abs-neList* (*ya* # *ysa*))))
**prefer** *2*
**apply** *simp*
**apply** (*subgoal-tac Abs-neList* (*x* # *ya* # *ysa*) = *append* (*singleton x*) (*Abs-neList* (*ya* # *ysa*)))
**prefer** *2*
**apply** (*simp* (*no-asm-simp*) *add*: *Abs-neList-inverse Rep-neList-inverse neList-def singleton-def append-def*
*del*: *foldr-1-cons*)
**apply** *simp*
**apply** *simp*
**apply** (*drule-tac x*=*y* **in** *spec*)
**apply** (*simp add*: *Abs-neList-inverse Rep-neList-inverse neList-def singleton-def append-def*
*del*: *foldr-1-cons*)
**done**

**lemma** *neList-append-induct*:
$\bigwedge$ *zs* . ⟦$\bigwedge$*x* . *P* (*singleton x*); $\bigwedge$*xs ys*. *P xs* ⟹ *P ys* ⟹ *P* (*append xs ys*)⟧
⟹ *P zs*
**by** (*rule neList-append-induct-0*, *auto*)

## 2.6 Elements

**constdefs**
*elem* :: *'a* ⇒ *'a neList* ⇒ *bool*
*elem x xs* == *x mem* (*Rep-neList xs*)

**lemma** *elem-singleton*[*simp*]:
*elem x* (*singleton y*) = (*x* = *y*)
**apply** (*simp add*: *Abs-neList-inverse Rep-neList-inverse neList-def singleton-def elem-def*
*del*: *foldr-1-cons*)
**apply** *fast*
**done**

**lemma** *elem-append*[*simp*]:
*elem x* (*append xs ys*) = (*elem x xs* | *elem x ys*)
**by** (*simp add*: *Abs-neList-inverse Rep-neList-inverse neList-def elem-def append-def*
*del*: *foldr-1-cons*)

**constdefs**
*set1* :: *'a neList* ⇒ *'a set*
*set1 xs* == *set* (*Rep-neList xs*)

**lemma** *set1-singleton*[*simp*]:
*set1* (*singleton x*) = { *x* }
**by** (*simp add*: *Abs-neList-inverse Rep-neList-inverse neList-def singleton-def set1-def*
*del*: *foldr-1-cons*)

**lemma** *set1-append*[*simp*]:
*set1* (*append xs ys*) = *set1 xs Un set1 ys*
**apply** (*subgoal-tac* (*Rep-neList xs* @ *Rep-neList ys*) : *neList*)
**apply** (*simp add*: *Abs-neList-inverse Rep-neList-inverse neList-def append-def set1-def*
*del*: *foldr-1-cons*)
**apply** (*cases xs*)
**apply** (*simp add*: *Abs-neList-inverse Rep-neList-inverse neList-def elem-def append-def*
*del*: *foldr-1-cons*)
**done**

## 2.7 neFold

**constdefs**
*neFold* :: ('*a* ⇒ '*b*) ⇒ ('*b* ⇒ '*b* ⇒ '*b*) ⇒ '*a neList* ⇒ '*b*
*neFold s c xs* == *foldr1 c* (*map1 s xs*)

**lemma** *neFold-singleton*[*simp*]:
*neFold s c* (*singleton x*) = *s x*
**by** (*simp add*: *neFold-def*)

**lemma** *neFold-cons1*[*simp*]:
*neFold s c* (*cons1 x xs*) = *c* (*s x*) (*neFold s c xs*)
**by** (*simp add*: *neFold-def*)

**lemma** *neFold-append*[*simp*]:
*assoc c* ⟹ *neFold s c* (*append xs ys*) = *c* (*neFold s c xs*) (*neFold s c ys*)
**by** (*simp add*: *neFold-def*)

**lemma** *neFold-const-idempotent*[*simp*]:
⟦ *assoc c*; *idempotent c* ⟧ ⟹ *neFold* (% *h* . *a*) *c t* = *a*
**by** (*induct-tac t rule*: *neList-append-induct*, *simp-all*)

## 2.8 ZipWith

**constdefs**
*zipWith1* :: ('*a* ⇒ '*b* ⇒ '*c*) ⇒ '*a neList* ⇒ '*b neList* ⇒ '*c neList*
*zipWith1 f xs ys* == *Abs-neList* (*zipWith f* (*Rep-neList xs*) (*Rep-neList ys*))

**lemma** *zipWith1-S-S*[*simp*]:
*zipWith1 f* (*singleton x*) (*singleton y*) = *singleton* (*f x y*)
**by** (*simp add*: *Abs-neList-inverse Rep-neList-inverse neList-def singleton-def zipWith1-def*)

**lemma** *zipWith1-S-C*[*simp*]:
*zipWith1 f* (*singleton x*) (*cons1 y ys*) = *singleton* (*f x y*)

**by** (*simp add*: *Abs-neList-inverse Rep-neList-inverse neList-def singleton-def cons1-def append-def zipWith1-def*)

**lemma** *zipWith1-C-S*[*simp*]:
*zipWith1 f*(*cons1 x xs*) (*singleton y*) = *singleton* (*f x y*)
**by** (*simp add*: *Abs-neList-inverse Rep-neList-inverse neList-def singleton-def cons1-def append-def zipWith1-def*)

**lemma** *zipWith1-C-C*[*simp*]:
*zipWith1 f*(*cons1 x xs*) (*cons1 y ys*) = *cons1* (*f x y*) (*zipWith1 f xs ys*)
**apply** (*subgoal-tac zipWith f* (*Rep-neList xs*) (*Rep-neList ys*) : *neList*)
**apply** (*simp add*: *Abs-neList-inverse Rep-neList-inverse neList-def singleton-def cons1-def append-def zipWith1-def*)
**apply** (*cases xs*, *simp*)
**apply** (*cases ys*, *simp*)
**apply** (*simp-all add*: *Abs-neList-inverse*)
**done**

**lemma** *zipWith1-assoc*[*simp*]:
*assoc f* ⟹ *zipWith1 f* (*zipWith1 f xs ys*) *zs* = *zipWith1 f xs* (*zipWith1 f ys zs*)
**apply** (*simp add*: *zipWith1-def*)
**apply** (*cases xs*, *simp*)
**apply** (*cases ys*, *simp*)
**apply** (*cases zs*, *simp*)
**apply** (*simp-all add*: *Abs-neList-inverse*)
**done**

**lemma** *assoc-zipWith1*[*simp*]: *assoc f* ⟹ *assoc* (*zipWith1 f*)
**by** (*rule assoc-intro*, *simp*)

Attempting a direct proof of this associativity, not using associativity of zipWith:

**lemma** *zipWith1-assoc-2*[*simp*]:
*assoc f* ⟹
*ALL xs ys zs* . *zipWith1 f* (*zipWith1 f xs ys*) *zs* = *zipWith1 f xs* (*zipWith1 f ys zs*)
**apply** (*rule neList-cons-inductA*)
**apply** (*rule neList-cons-inductA*)
**apply** *simp-all*
**apply** (*rule neList-cons-inductA*)
**apply** *simp-all*
**apply** (*rule neList-cons-inductA*)
**apply** *simp-all*
**done**

**lemma** *zipWith1-assoc-1*[*simp*]:
*assoc f* ⟹
*ALL ys zs* . *zipWith1 f* (*zipWith1 f xs ys*) *zs* = *zipWith1 f xs* (*zipWith1 f ys zs*)
**apply** (*induct-tac xs rule*: *neList-cons-induct*)

**apply** *simp*
**apply** (*rule allI*)
**apply** (*induct-tac ys rule*: *neList-cons-induct*)
**apply** (*simp add*: *cons1*)
**apply** (*rule allI*)
**apply** (*induct-tac zs rule*: *neList-cons-induct*)
**apply** (*simp-all add*: *cons1*)
**done**

**lemma** *zipWith1-assoc-3*[*simp*]:
*assoc f* $\Longrightarrow$
*zipWith1 f* (*zipWith1 f xs ys*) *zs* = *zipWith1 f xs* (*zipWith1 f ys zs*)
**apply** (*induct-tac xs rule*: *neList-cons-induct*)
**apply** *simp*
**apply** (*induct-tac ys rule*: *neList-cons-induct*)
**apply** (*simp add*: *cons1*)
**apply** (*induct-tac zs rule*: *neList-cons-induct*)
**apply** (*simp-all add*: *cons1*)
**done**

### 2.9 Other Properties

**lemma** *foldr1-map1-LRunit*[*simp*]:
$[\![ \bigwedge x$ . *LRunit f* (*u x*); *assoc f* $]\!] \Longrightarrow$
*foldr1 f* (*map1 u xs*) = *u arbitrary*
**apply** (*induct-tac xs rule*: *neList-append-induct*, *simp-all*)
**apply** (*rule LRunit-equal* [*of f*], *simp-all add*: *LRunit-left LRunit-right*)
**done**

**end**

## 3 The Table Datatype Definition

**theory** *Table* = *NEList*:

### 3.1 Datatype and Primitive Operations

**datatype** ('*h*,'*t*) *T* = *T* ('*h* ∗ '*t*) *neList*

**consts**
*unT* :: ('*h*,'*t*) *T* ⇒ ('*h* ∗ '*t*) *neList*

**primrec**
*unT-def*[*simp*]: *unT* (*T ps*) = *ps*

**constdefs**
*addH* :: '*h* ⇒ '*t* ⇒ ('*h*,'*t*) *T*
*addH h t* == *T* (*singleton* (*h*,*t*))

**constdefs**
*hConc* :: $('h,'t)\ T \Rightarrow ('h,'t)\ T \Rightarrow ('h,'t)\ T$
*hConc t1 t2* == *T* (*append* (*unT t1*) (*unT t2*))

**lemma** *hConc-assoc*[*simp*]: *hConc* (*hConc t1 t2*) *t3* = *hConc t1* (*hConc t2 t3*)
**by** (*unfold hConc-def*, *simp*)

**lemma** *assoc-hConc*[*simp*]: *assoc hConc*
**by** (*rule assoc-intro*, *simp*)

**constdefs**
*cell* :: $'c \Rightarrow ('c,\ unit)\ T$
*cell c* == *addH c* ()

## 3.2 Folding and Induction via hConc

**constdefs**
*tFold* :: $('h \Rightarrow 't \Rightarrow 'r) \Rightarrow ('r \Rightarrow 'r \Rightarrow 'r) \Rightarrow ('h,'t)\ T \Rightarrow 'r$
*tFold h c t* == *neFold* (*uncurry h*) *c* (*unT t*)

**lemma** *tFold-addH*[*simp*]:
*tFold ah c* (*addH h t*) = *ah h t*
**by** (*simp add*: *tFold-def addH-def*)

**lemma** *tFold-hConc*[*simp*]:
*assoc c* $\Longrightarrow$ *tFold h c* (*hConc t1 t2*) = *c* (*tFold h c t1*) (*tFold h c t2*)
**by** (*simp add*: *tFold-def hConc-def*)

**lemma** *T-induct-0*:
$[\![\bigwedge h\ t0\ .\ P\ (addH\ h\ t0);\ \bigwedge t1\ t2.\ P\ t1 \Longrightarrow P\ t2 \Longrightarrow P\ (hConc\ t1\ t2)]\!]$
$\Longrightarrow P\ t$
**apply** (*cases t*, *simp*)
**apply** (*rule-tac zs*=*neList* **in** *neList-append-induct*)
**apply** (*subst surjective-pairing*)
**apply** (*simp only*: *addH-def*)
**apply** (*simp add*: *hConc-def*)
**apply** (*subgoal-tac append xs ys* = *append* (*unT* (*T xs*)) (*unT* (*T ys*)))
**apply** (*simp* (*no-asm-simp*) *del*: *unT-def*)
**apply** *simp*
**done**

**lemma** *T-induct*:
$\bigwedge\ t\ .\ [\![\bigwedge h\ t0\ .\ P\ (addH\ h\ t0);\ \bigwedge t1\ t2.\ P\ t1 \Longrightarrow P\ t2 \Longrightarrow P\ (hConc\ t1\ t2)]\!]$
$\Longrightarrow P\ t$
**by** (*rule T-induct-0*, *auto*)

## 3.3 hCons

**constdefs**

*hCons* :: ('*h* ∗ '*t*) ⇒ ('*h*,'*t*) *T* ⇒ ('*h*,'*t*) *T*
*hCons p t1* == *T* (*cons1 p* (*unT t1*))

**lemma** *hCons*:
*hCons p t1* = *hConc* (*uncurry addH p*) *t1*
**proof** −
**have** *hCons p t1* = *T* (*cons1 p* (*unT t1*)) **by** (*simp add*: *hCons-def*)
**also have** ... = *T* (*append* (*singleton p*) (*unT t1*)) **by** (*simp add*: *cons1*)
**also have** ... = *T* (*append* (*unT* (*T* (*singleton* (*fst p*, *snd p*)))) (*unT t1*)) **by** *simp*
**also have** ... = *T* (*append* (*unT* (*addH* (*fst p*) (*snd p*))) (*unT t1*)) **by** (*simp add*: *addH-def*)
**also have** ... = *hConc* (*addH* (*fst p*) (*snd p*)) *t1* **by** (*simp add*: *hConc-def*)
**also have** ... = *hConc* (*uncurry addH* (*fst p*, *snd p*)) *t1* **by** (*simp del*: *surjective-pairing* [*THEN sym*])
**also have** ... = *hConc* (*uncurry addH p*) *t1* **by** *simp*
**finally show** *?thesis* **.**
**qed**

**lemma** *hCons-p*:
*hCons* (*h*,*t*) *t1* = *hConc* (*addH h t*) *t1*
**by** (*simp add*: *hCons*)

**lemma** *tFold-hCons*[*simp*]:
*tFold ah c* (*hCons p t*) = *c* (*uncurry ah p*) (*tFold ah c t*)
**by** (*simp add*: *tFold-def hCons-def*)

**lemma** *T-cons-induct-0*:
⟦⋀*h t0* . *P* (*addH h t0*); ⋀ *p t2*. *P t2* ⟹ *P* (*hCons p t2*)⟧
⟹ *P t*
**apply** (*cases t*, *simp*)
**apply** (*rule-tac zs*=*neList* **in** *neList-cons-induct*)
**apply** (*subst surjective-pairing*)
**apply** (*simp only*: *addH-def*)
**apply** (*subst surjective-pairing*)
**apply** (*simp add*: *hConc-def hCons-def addH-def*)
**apply** (*subgoal-tac cons1 x xs* = *cons1 x* (*unT* (*T xs*)))
**apply** (*simp* (*no-asm-simp*) *del*: *unT-def*)
**apply** *simp*
**done**

**lemma** *T-cons-induct*:
⋀ *t* .( ⟦⋀*h t0* . *P* (*addH h t0*); ⋀ *p t2*. *P t2* ⟹ *P* (*hCons p t2*)⟧ ⟹ *P t* )
**by** (*rule T-cons-induct-0*, *auto*)

**lemma** *T-cons-inductA*:
⟦⋀*h t0* . *P* (*addH h t0*); ⋀ *p t2*. *P t2* ⟹ *P* (*hCons p t2*)⟧ ⟹ *ALL t* . *P t*
**by** (*intro strip*, *rule T-cons-induct*, *simp-all*)

**lemma** *hConc-hCons*[*simp*]: *hConc* (*hCons p t1*) *t2* = *hCons p* (*hConc t1 t2*)
**by** (*induct-tac t1 rule*: *T-cons-induct*, *simp-all add*: *hCons*)

**end**

# 4 Table Utilities and Properties

**theory** *Tables* = *Table*:

## 4.1 Additional Properties of *tFold*

**lemma** *tFold-const2-idempotent*[*simp*]:
⟦ *assoc c*; *idempotent c* ⟧ ⟹ *tFold* (% *h t* . *a*) *c t* = *a*
**by** (*induct-tac t rule*: *T-induct*, *simp-all*)

**lemma** *tFold-tFold-join*:
⟦ *assoc c2* ⟧ ⟹
*tFold a2 c2* (*tFold* (λ*h0 t0*. *addH* (*HH h h0 t0*) (*TT h h0 t0*)) *hConc t*) =
*tFold* (λ *h0 t0* . *a2* (*HH h h0 t0*) (*TT h h0 t0*)) *c2 t*
**by** (*induct-tac t rule*: *T-induct*, *simp-all*)

**lemma** *tFold-tFold-hConc*:
⟦ *assoc c2* ⟧ ⟹
*tFold a2 c2* (*tFold a1 hConc t*) =
*tFold* (λ *h0 t0* . *tFold a2 c2* (*a1 h0 t0*)) *c2 t*
**by** (*induct-tac t rule*: *T-induct*, *simp-all*)

**lemma** *tFold-tFold-const*:
⟦ *assoc c2* ⟧ ⟹
*tFold a2 c2* (*tFold a1 const t*) =
*tFold* (λ *h0 t0* . *tFold a2 c2* (*a1 h0 t0*)) *const t*
**apply** (*induct-tac t rule*: *T-induct*, *simp-all del*: *const*)
**apply** (*subst const*)+
**apply** (*simp del*: *const*)
**done**

**lemma** *f-tFold-const*:
*f* (*tFold a const t*) = *tFold* (% *h0 t0* . *f* (*a h0 t0*)) *const t*
**apply** (*induct-tac t rule*: *T-induct*, *simp-all del*: *const*)
**apply** (*subst const*)+
**apply** (*simp del*: *const*)
**done**

**lemma** *tFold-const-const*[*simp*]: *tFold* (% *h t* . *r*) *const t* = *r*
**by** (*induct-tac t rule*: *T-induct*, *simp-all del*: *const*, *simp*)

**lemma** *tFold-foldr1-hConc*:

*assoc c* ⟹ *tFold a c* (*foldr1 hConc ts*) = *foldr1 c* (*map1* (*tFold a c*) *ts*)
**by** (*induct-tac ts rule*: *neList-append-induct*, *simp-all*)

**lemma** *tFold-LRunit*[*simp*]:
⟦ ⋀ *h t* . *LRunit c* (*a h t*); *assoc c* ⟧ ⟹
*tFold a c t* = *a arbitrary arbitrary*
**apply** (*induct-tac t rule*: *T-induct*, *simp-all*)
**apply** (*rule LRunit-equal* [*of c*], *simp-all add*: *LRunit-left LRunit-right*)
**done**

**consts** *tFold0* :: ('*c* ⇒ '*c* ⇒ '*c*) ⇒ ('*c*,'*u*) *T* ⇒ '*c*
**defs** *tFold0-def*: *tFold0* == *tFold* (% *h t* . *h*)

**lemma** *tFold0-addH*[*simp*]: *tFold0 c* (*addH h t*) = *h*
**by** (*unfold tFold0-def*, *simp*)

**lemma** *tFold0-hConc*[*simp*]:
*assoc c* ⟹ *tFold0 c* (*hConc t1 t2*) = *c* (*tFold0 c t1*) (*tFold0 c t2*)
**by** (*unfold tFold0-def*, *simp*)

## 4.2 tFoldC

**consts** *tFoldC* :: ('*h* ⇒ '*c*) ⇒ ('*c* ⇒ '*c* ⇒ '*c*) ⇒ ('*h*,'*u*) *T* ⇒ '*c*
**defs** *tFoldC-def*: *tFoldC a* == *tFold* (% *h t* . *a h*)

## 4.3 hHead

**constdefs**
*hHead* :: ('*h*,'*t*) *T* ⇒ ('*h* ∗ '*t*)
*hHead* == *tFold Pair const*

**lemma** *hHead-addH*[*simp*]: *hHead* (*addH h t*) = (*h*,*t*)
**by** (*simp add*: *hHead-def*)

**lemma** *hHead-uncurry-addH*[*simp*]: *hHead* (*uncurry addH p*) = *p*
**by** (*simp add*: *hHead-def*)

**lemma** *hHead-hCons*[*simp*]: *hHead* (*hCons p t*) = *p*
**by** (*simp add*: *hHead-def*)

**lemma** *hHead-hConc*[*simp*]: *hHead* (*hConc t1 t2*) = *hHead t1*
**by** (*simp add*: *hHead-def*)

**constdefs**
*hLength* :: ('*h*,'*t*) *T* ⇒ *nat*
*hLength* == *tFold* (% *h t* . *1*) (% *x y* . *x* + *y*)

**lemma** *hLength-addH*[*simp*]: *hLength* (*addH h t*) = *1*
**by** (*simp add*: *hLength-def*)

**lemma** *hLength-hCons*[*simp*]: *hLength* (*hCons p t*) = *Suc* (*hLength t*)
**by** (*simp add*: *hLength-def*)

**lemma** *hLength-hConc*[*simp*]: *hLength* (*hConc t1 t2*) = *hLength t1* + *hLength t2*
**apply** (*subgoal-tac assoc* ((*op* +) :: *nat* ⇒ *nat* ⇒ *nat*))
  **prefer** *2*
  **apply** (*rule assoc-intro*, *simp*)
**apply** (*simp add*: *hLength-def*)
**done**

**lemma** *hLength-pos*[*simp*]: *0* < *hLength t*
**by** (*induct-tac t rule*: *T-induct*, *simp-all*)

## 4.4 The "Cons View"

**constdefs**
  *hUnCons0* :: (('*h* ∗ '*t*) + (('*h* ∗ '*t*) ∗ ('*h*,'*t*) *T*)) ∗ ('*h*,'*t*) *T* ⇒
        (('*h* ∗ '*t*) + (('*h* ∗ '*t*) ∗ ('*h*,'*t*) *T*)) ∗ ('*h*,'*t*) *T* ⇒
        (('*h* ∗ '*t*) + (('*h* ∗ '*t*) ∗ ('*h*,'*t*) *T*)) ∗ ('*h*,'*t*) *T*
  *hUnCons0 p1 p2* == (*case fst p1 of*
            *Inl p* ⇒ *Inr* (*p*,*snd p2*)
           | *Inr* (*p*,*t*) ⇒ *Inr* (*p*,*hConc t* (*snd p2*))
          ,*hConc* (*snd p1*) (*snd p2*)
          )

**lemma** *snd-hUnCons0-p*: *snd* (*hUnCons0 p1 p2*) = *hConc* (*snd p1*) (*snd p2*)
**by** (*simp add*: *hUnCons0-def*)

**lemma** *fst-hUnCons0-p*: *fst* (*hUnCons0 p1 p2*) = (*case fst p1 of*
            *Inl p* ⇒ *Inr* (*p*,*snd p2*)
           | *Inr* (*p*,*t*) ⇒ *Inr* (*p*,*hConc t* (*snd p2*)))
**by** (*simp add*: *hUnCons0-def*)

**lemma** *snd-hUnCons0*[*simp*]: *snd* (*hUnCons0* (*r1*,*t1*) (*r2*,*t2*)) = *hConc t1 t2*
**by** (*simp add*: *snd-hUnCons0-p*)

**lemma** *assoc-hUnCons0*[*simp*]: *assoc hUnCons0*
**apply** (*rule assoc-intro*)
**apply** (*simp add*: *hUnCons0-def*)
**apply** (*split sum.split*, *rule conjI*)
**apply** (*intro strip*, *simp*)
**apply** (*intro strip*, *simp*)
**apply** (*split sum.split*, *rule conjI*)
**apply** (*intro strip*)
**apply** (*case-tac b*, *simp*)
**apply** (*intro strip*, *case-tac b*, *simp*)
**apply** (*rotate-tac* −*2*, *drule sym*, *simp*)
**done**

**constdefs**
*hUnCons1* :: $('h,'t)\ T \Rightarrow (('h * 't) + (('h * 't) * ('h,'t)\ T)) * ('h,'t)\ T$
*hUnCons1* == *tFold* (% *h t* . (*Inl* (*h*,*t*), *addH h t*)) *hUnCons0*

**lemma** *hUnCons1-addH*[*simp*]: *hUnCons1* (*addH h t*) = (*Inl* (*h*,*t*), *addH h t*)
**by** (*simp add*: *hUnCons1-def*)

**lemma** *hUnCons1-hCons*[*simp*]: *hUnCons1* (*hCons p t*) = (*Inr* (*p*,*t*), *hCons p t*)
**apply** (*simp add*: *hUnCons1-def hUnCons0-def*)
**apply** (*induct-tac t rule*: *T-induct*)
**apply** (*simp add*: *hCons*)
**apply** (*simp add*: *snd-hUnCons0-p hCons*)
**done**

**lemma** *hUnCons1-hConc*[*simp*]:
*hUnCons1* (*hConc t1 t2*) = *hUnCons0* (*hUnCons1 t1*) (*hUnCons1 t2*)
**by** (*simp add*: *hUnCons1-def*)

**lemma** *snd-hUnCons1*[*simp*]: *snd* (*hUnCons1 t*) = *t*
**by** (*induct-tac t rule*: *T-induct*, *simp-all add*: *snd-hUnCons0-p*)

**constdefs**
*hUnCons* :: $('h,'t)\ T \Rightarrow (('h * 't) + (('h * 't) * ('h,'t)\ T))$
*hUnCons t* == *fst* (*hUnCons1 t*)

**lemma** *hUnCons-addH*[*simp*]: *hUnCons* (*addH h t*) = *Inl* (*h*,*t*)
**by** (*simp add*: *hUnCons-def*)

**lemma** *hUnCons-hCons*[*simp*]: *hUnCons* (*hCons p t*) = *Inr* (*p*,*t*)
**by** (*simp add*: *hUnCons-def*)

**lemma** *hUnCons-hConc*[*simp*]: *hUnCons* (*hConc t1 t2*) = (*case hUnCons t1 of*
*Inl p* $\Rightarrow$ *Inr* (*p*, *t2*)
| *Inr* (*p*, *t*) $\Rightarrow$ *Inr* (*p*, *hConc t t2*))
**apply** (*simp add*: *hUnCons-def fst-hUnCons0-p*)
**apply** (*subst snd-hUnCons1*, *simp*)
**done**

**consts**
*tFoldr0* :: $('h \Rightarrow 't \Rightarrow 'r) * ('h \Rightarrow 't \Rightarrow 'r \Rightarrow 'r) * ('h,'t)\ T \Rightarrow 'r$

**lemma** *tFoldr0-measure*[*simp*]:
$\forall h\ t\ t'.\ (\exists ta.\ hUnCons\ t = Inr\ ((h,\ ta),\ t')) \longrightarrow hLength\ t' < hLength\ t$
**apply** (*rule allI*)
**apply** (*rule allI*)
**apply** (*rule-tac x*=*h* **in** *spec*)
**apply** (*induct-tac t rule*: *T-induct*, *auto*)
**apply** (*case-tac hUnCons t1*, *simp*)
**apply** (*case-tac b*, *simp*)

**apply** (*erule conjE*, *rotate-tac* $-1$, *drule sym*, *simp*)
**done**

**recdef** *tFoldr0 measure* (% (*a*,*f*,*t*) . *hLength t*)
  *tFoldr0-def*: *tFoldr0* (*a*,*f*,*t*) = (*case hUnCons t of*
        *Inl* (*h*,*t*) ⇒ *a h t*
      | *Inr* ((*h*,*t*),*t*′) ⇒ *f h t* (*tFoldr0* (*a*,*f*,*t*′)))

**constdefs**
  *tFoldr* :: (′*h* ⇒ ′*t* ⇒ ′*r*) ⇒ (′*h* ⇒ ′*t* ⇒ ′*r* ⇒ ′*r*) ⇒ (′*h*,′*t*) *T* ⇒ ′*r*
  *tFoldr a f* == % *t* . *tFoldr0* (*a*,*f*,*t*)

**lemma** *tFoldr-addH*[*simp*]: *tFoldr a f* (*addH h t*) = *a h t*
**by** (*simp add*: *tFoldr-def tFoldr0-def*)

**lemma** *tFoldr-hCons*[*simp*]: *tFoldr a f* (*hCons p t*) = *uncurry f p* (*tFoldr a f t*)
**by** (*case-tac p*, *simp add*: *tFoldr-def tFoldr0-def*)

**lemma** *tFoldr-hConc*:
  *tFoldr a f* (*hConc t1 t2*) = *tFoldr* (% *h t* . *f h t* (*tFoldr a f t2*)) *f t1*
**apply** (*induct-tac t1 rule*: *T-cons-induct*, *simp*)
**apply** (*subst hCons-p* [*THEN sym*], *simp-all*)
**done**

**lemma** *hConc-via-tFoldr*:
  *hConc t1 t2* = *tFoldr* (% *h t* . *hCons* (*h*,*t*) *t2*) (*curry hCons*) *t1*
**by** (*induct-tac t1 rule*: *T-cons-induct*, *simp-all*, *simp add*: *hCons*)

**lemma** *tFold-via-tFoldr*:
  *tFold a c t* = *tFoldr a* (% *h t r* . *c* (*a h t*) *r*) *t*
**by** (*induct-tac t rule*: *T-cons-induct*, *simp-all*)

## 4.5 Mapping

**constdefs**
 *hMap* :: (′*h1* ⇒ ′*h2*) ⇒ (′*h1*,′*t*) *T* ⇒ (′*h2*,′*t*) *T*
 *hMap f* == *tFold* (*addH o f*) (*hConc*)
 *tMap* :: (′*t1* ⇒ ′*t2*) ⇒ (′*h*,′*t1*) *T* ⇒ (′*h*,′*t2*) *T*
 *tMap f* == *tFold* (% *h t* . *addH h* (*f t*)) (*hConc*)

**lemma** *tMap-addH*[*simp*]: *tMap f* (*addH h t*) = *addH h* (*f t*)
**by** (*simp add*: *tMap-def*)

**lemma** *tMap-hConc*[*simp*]: *tMap f* (*hConc t1 t2*) = *hConc* (*tMap f t1*) (*tMap f t2*)
**by** (*simp add*: *tMap-def*)

**lemma** *tMap-hCons*[*simp*]: *tMap f* (*hCons p t*) = *hCons* (*fst p*, *f* (*snd p*)) (*tMap f t*)

**by** (*simp add*: *tMap-def hCons*)

**lemma** *hMap-addH*[*simp*]: *hMap f* (*addH h t*) = *addH* (*f h*) *t*
**by** (*simp add*: *hMap-def*)

**lemma** *hMap-hConc*[*simp*]: *hMap f* (*hConc t1 t2*) = *hConc* (*hMap f t1*) (*hMap f t2*)
**by** (*simp add*: *hMap-def*)

**lemma** *hMap-hCons*[*simp*]: *hMap f* (*hCons p t*) = *hCons* (*f* (*fst p*), *snd p*) (*hMap f t*)
**by** (*simp add*: *hMap-def hCons*)

**lemma** *tFold-tMap*[*simp*]:
*assoc c* $\Longrightarrow$ *tFold a c* (*tMap f t*) = *tFold* (% *h t0* . *a h* (*f t0*)) *c t*
**by** (*induct-tac t rule*: *T-induct*, *simp-all*)

**lemma** *tMap-tFold-hConc*:
*tMap f* (*tFold a hConc t*) = *tFold* ($\lambda$ *h0 t0* . *tMap f* (*a h0 t0*)) *hConc t*
**by** (*induct-tac t rule*: *T-induct*, *simp-all*)

**lemma** *tMap-tMap*: *tMap f* (*tMap g t*) = *tMap* (*f o g*) *t*
**by** (*induct-tac t rule*: *T-induct*, *simp-all*)

**lemma** *tMap-const-tMap*[*simp*]:
*tMap* (*const t1*) (*tMap f t*) = *tMap* (*const t1*) *t*
**by** (*subst tMap-tMap*, *simp add*: *comp-def*)

**lemma** *tMap-CONST-tMap*[*simp*]:
*tMap* ($\lambda$ *x* . *t1*) (*tMap f t*) = *tMap* (*const t1*) *t*
**by** (*subst tMap-tMap*, *simp add*: *comp-def*)

### 4.6 Headers and Subtables

**constdefs**
*headers* :: ('*h*,'*t*) *T* $\Rightarrow$ '*h neList*
*headers* == *tFold* (% *h t* . *singleton h*) *append*
*subtables* :: ('*h*,'*t*) *T* $\Rightarrow$ '*t neList*
*subtables* == *tFold* (% *h t* . *singleton t*) *append*

**lemma** *headers-addH*[*simp*]: *headers* (*addH h t*) = *singleton h*
**by** (*simp add*: *headers-def*)

**lemma** *headers-hCons*[*simp*]: *headers* (*hCons p t*) = *cons1* (*fst p*) (*headers t*)
**by** (*simp add*: *headers-def cons1*)

**lemma** *headers-hConc*[*simp*]: *headers* (*hConc t1 t2*) = *append* (*headers t1*) (*headers t2*)
**by** (*simp add*: *headers-def*)

**lemma** *subtables-addH*[*simp*]: *subtables* (*addH h t*) = *singleton t*
**by** (*simp add*: *subtables-def*)

**lemma** *subtables-hCons*[*simp*]: *subtables* (*hCons p t*) = *cons1* (*snd p*) (*subtables t*)
**by** (*simp add*: *subtables-def cons1*)

**lemma** *subtables-hConc*[*simp*]: *subtables* (*hConc t1 t2*) = *append* (*subtables t1*) (*subtables t2*)
**by** (*simp add*: *subtables-def*)

**lemma** *headers-eq-singleton-0*:
*headers t* = *singleton h* ⟶ (*EX t0* . *t* = *addH h t0*)
**apply** (*induct-tac t rule*: *T-cons-induct*, *auto*)
**apply** (*rule-tac x*=*t0* **in** *exI*, *drule singleton-inj*, *simp*)
**done**

**lemma** *headers-eq-singleton*[*dest?*]:
*headers t* = *singleton h* ⟹ (*EX t0* . *t* = *addH h t0*)
**by** (*rule headers-eq-singleton-0* [*THEN mp*])

**lemma** *headers-eq-cons1-0*:
*headers t1* = *cons1 h hs* ⟶
(*EX t0 t2* . *t1* = *hCons* (*h*,*t0*) *t2* & *headers t2* = *hs*)
**apply** (*induct-tac t1 rule*: *T-cons-induct*, *simp-all*)
**apply** (*intro strip*)
**apply** (*drule cons1-inj*, *erule conjE*, *simp*)
**apply** (*rule-tac x*=*snd p* **in** *exI*)
**apply** (*rule-tac x*=*t2* **in** *exI*)
**apply** (*rotate-tac* −*2*, *drule sym*, *simp*)
**done**

**lemma** *headers-eq-cons1*[*dest?*]:
*headers t1* = *cons1 h hs* ⟹
(*EX t0 t2* . *t1* = *hCons* (*h*,*t0*) *t2* & *headers t2* = *hs*)
**by** (*rule headers-eq-cons1-0* [*THEN mp*])

**lemma** *headers-tMap*[*simp*]: *headers* (*tMap f t*) = *headers t*
**by** (*induct-tac t rule*: *T-induct*, *simp-all*)

**lemma** *headers-tFold-hConc*[*simp*]:
*headers* (*tFold a hConc t*) = *tFold* (% *h t* . *headers* (*a h t*)) *append t*
**by** (*induct-tac t rule*: *T-induct*, *simp-all*)

**lemma** *tMap-CONST--headers*:
*tMap* (λ *x* . *c*) *t* = *foldr1 hConc* (*map1* (λ *h* . *addH h c*) (*headers t*))
**by** (*induct-tac t rule*: *T-induct*, *simp-all*)

**lemma** *tFold-CONST-hConc--headers*:
*tFold* (λ*h* (*t* :: *unit*). *c t*) *hConc t* = *foldr1 hConc* (*map1* (λ *h* . *c* ()) (*headers t*))
**by** (*induct-tac t rule*: *T-induct*, *simp-all*)

## 4.7 Regular Skeletons

**constdefs**
*regSkelStep* :: ('*t* ⇒ '*s option*) ⇒ ('*h*, '*t*) *T* ⇒ ('*h neList* ∗ '*s*) *option*
*regSkelStep rs t* == *option-map* (% *s* . (*headers t*, *s*))
(*tFold* (% *h t* . *rs t*) *optEq t*)

**lemma** *regSkelStep-addH*[*simp*]:
*regSkelStep rs* (*addH h t*) = *option-map* (% *s* . (*singleton h*, *s*)) (*rs t*)
**by** (*simp add*: *regSkelStep-def*)

**lemma** *regSkelStep-hConc*[*simp*]:
*regSkelStep rs* (*hConc t1 t2*) =
*optThen* (*regSkelStep rs t1*) (λ *p1* .
*optThen* (*regSkelStep rs t2*) (λ *p2* .
*if snd p1* = *snd p2*
*then Some* (*append* (*fst p1*) (*fst p2*), *snd p1*)
*else None*))
**apply** (*simp add*: *regSkelStep-def*)
**apply** (*cases* (*tFold* (λ*h*. *rs*) *optEq t1*), *simp add*: *optEq-N*)
**apply** (*cases* (*tFold* (λ*h*. *rs*) *optEq t2*), *simp*)
**apply** *simp*
**done**

**lemma** *regSkelStep-hCons*[*simp*]:
*regSkelStep rs* (*hCons p t*) =
*optThen* (*rs* (*snd p*))
(λ*hs2*. *optThen* (*regSkelStep rs t*)
(λ*p2*. *if hs2* = *snd p2*
*then Some* (*cons1* (*fst p*) (*fst p2*), *hs2*) *else None*))
**apply** (*simp add*: *hCons cons1*)
**apply** (*cases rs* (*snd p*), *simp-all add*: *cons1*)
**apply** (*cases regSkelStep rs t*, *simp-all add*: *cons1*)
**done**

**constdefs**
*regSkelOuter1* :: ('*h*,'*t*) *T* ⇒ '*h neList option*
*regSkelOuter1 t* == *Some* (*headers t*)

**lemma** *regSkelOuter1-addH*[*simp*]:
*regSkelOuter1* (*addH h t*) = *Some* (*singleton h*)
**by** (*simp add*: *regSkelOuter1-def*)

**lemma** *regSkelOuter1-hConc*[*simp*]:

*regSkelOuter1* (*hConc t1 t2*) = *Some* (*append* (*headers t1*) (*headers t2*))
**by** (*simp add*: *regSkelOuter1-def*)

**lemma** *regSkelOuter1-hCons*[*simp*]:
*regSkelOuter1* (*hCons p t*) = *Some* (*cons1* (*fst p*) (*headers t*))
**by** (*simp add*: *hCons cons1*)

**constdefs**
*regSkelOuter2* :: ('*h1*,('*h2*,'*t*) *T*) *T* ⇒ ('*h1 neList* ∗ ('*h2 neList*)) *option*
*regSkelOuter2 t* == *regSkelStep regSkelOuter1 t*

**lemma** *regSkelOuter2-addH*[*simp*]:
*regSkelOuter2* (*addH h t*) = *Some* (*singleton h*, *headers t*)
**by** (*simp add*: *regSkelOuter2-def regSkelOuter1-def*)

**lemma** *regSkelOuter2-hCons*[*simp*]:
*regSkelOuter2* (*hCons p t2*) = *optThen* (*regSkelOuter2 t2*)
(λ*p2*. *if headers* (*snd p*) = *snd p2*
*then Some* (*cons1* (*fst p*) (*fst p2*), *snd p2*) *else None*)
**by** (*case-tac regSkelOuter2 t2*, *simp-all add*: *regSkelOuter2-def regSkelOuter1-def*)

**lemma** *regSkelOuter2-hConc*[*simp*]:
*regSkelOuter2* (*hConc t1 t2*) = *optThen* (*regSkelOuter2 t1*)
(λ*p1*. *optThen* (*regSkelOuter2 t2*)
(λ*p2*. *if snd p1* = *snd p2*
*then Some* (*append* (*fst p1*) (*fst p2*), *snd p1*) *else None*))
**by** (*simp add*: *regSkelOuter2-def*)

**lemma** *regSkelOuter2-addH-eq-Some*:
*regSkelOuter2* (*addH h t*) = *Some* (*hs1*, *hs2*) ⟹
*hs1* = *singleton h* & *hs2* = *headers t*
**by** (*simp add*: *regSkelOuter1-def regSkelOuter2-def*)

**lemma** *regSkelOuter2-hConc-eq-Some*:
*regSkelOuter2* (*hConc t1 t2*) = *Some* (*hs1*, *hs2*) ⟹
*EX hs1a hs1b* . *append hs1a hs1b* = *hs1* &
*regSkelOuter2 t1* = *Some* (*hs1a*, *hs2*) &
*regSkelOuter2 t2* = *Some* (*hs1b*, *hs2*)
**apply** *simp*
**apply** (*drule optThen-result-Some*, *erule exE*, *erule conjE*)
**apply** (*drule optThen-result-Some*, *erule exE*, *erule conjE*)
**apply** (*split split-if-asm*)
**apply** (*rule-tac x*=*fst y* **in** *exI*)
**apply** (*rule-tac x*=*fst ya* **in** *exI*)
**apply** *simp*
**apply** (*erule conjE*, *rule conjI*)
**apply** (*rotate-tac 2*, *drule sym*, *simp*)
**apply** (*rotate-tac −1*, *drule sym*, *simp*)
**apply** *fastsimp*

**done**

**lemma** *regSkelOuter2-hCons-eq-Some*:
  *regSkelOuter2* (*hCons p t2*) = *Some* (*hs1*, *hs2*) $\Longrightarrow$
   *EX h1 t1 hs1b* . *p* = (*h1*,*t1*) &
          *hs1* = *cons1 h1 hs1b* &
          *headers t1* = *hs2* &
          *regSkelOuter2 t2* = *Some* (*hs1b*, *hs2*)
**apply** (*simp only*: *cons1 hCons*)
**apply** (*drule regSkelOuter2-hConc-eq-Some*)
**apply** (*erule exE*, *erule exE*, *erule conjE*, *erule conjE*)
**apply** (*rule-tac x*=*fst p* **in** *exI*)
**apply** (*rule-tac x*=*snd p* **in** *exI*)
**apply** (*rule-tac x*=*hs1b* **in** *exI*)
**apply** *simp*
**done**

**lemma** *regSkelOuter2-eq-Some-0*:
  *ALL hs1 hs2* . *regSkelOuter2 t* = *Some* (*hs1*, *hs2*) $\longrightarrow$ *headers t* = *hs1*
**apply** (*induct-tac t rule*: *T-induct*, *simp-all*)
**apply** (*intro strip*)
**apply** (*erule exE*, *drule optThen-result-Some*, *simp*)
**apply** (*erule exE*, *erule exE*, *erule conjE*, *drule optThen-result-Some*, *simp*)
**apply** (*erule exE*, *erule exE*, *erule conjE*, *simp*)
**apply** (*split split-if-asm*, *simp-all*)
**done**

**lemma** *regSkelOuter2-eq-Some*[*simp*]:
  *regSkelOuter2 t* = *Some* (*hs1*, *hs2*) $\Longrightarrow$ *headers t* = *hs1*
**by** (*insert regSkelOuter2-eq-Some-0*, *auto*)

**lemma** *regSkelOuter2-tMap-const*[*simp*]:
*regSkelOuter2* (*tMap* (*const t2*) *t1*) = *Some* (*headers t1*, *headers t2*)
**by** (*induct-tac t1 rule*: *T-induct*, *simp-all*)

**lemma** *regSkelOuter2-tMap-CONST*[*simp*]:
*regSkelOuter2* (*tMap* (% *x* . *t2*) *t1*) = *Some* (*headers t1*, *headers t2*)
**by** (*induct-tac t1 rule*: *T-induct*, *simp-all*)

**lemma** *regSkelOuter2-tMap-h-0*:
$\llbracket$ $\bigwedge$*t*. *headers* (*h t*) = *headers t*; $\bigwedge$*t*. *h* (*h t*) = *h t*;
  $\bigwedge$*t1 t2*. *h* (*hConc t1 t2*) = *hConc* (*h t1*) (*h t2*)
 $\rrbracket$ $\Longrightarrow$
 *ALL hs1 hs2* .
 *regSkelOuter2 t* = *Some* (*hs1*, *hs2*) $\longrightarrow$
 *regSkelOuter2* (*tMap h t*) = *Some* (*hs1*, *hs2*)
**apply** (*induct-tac t rule*: *T-cons-induct*, *simp-all*)
**apply** (*intro strip*)
**apply** (*drule optThen-result-Some*, *erule exE*, *erule conjE*)

**apply** (*split split-if-asm*) **prefer** *2* **apply** *simp*
**apply** (*case-tac p*, *case-tac y*, *simp*)
**done**

**lemma** *regSkelOuter2-tMap-h*[*simp*]:
⟦ ⋀*t*. *headers* (*h t*) = *headers t*; ⋀*t*. *h* (*h t*) = *h t*;
⋀*t1 t2*. *h* (*hConc t1 t2*) = *hConc* (*h t1*) (*h t2*);
*regSkelOuter2 t* = *Some* (*hs1*, *hs2*)
⟧ ⟹
*regSkelOuter2* (*tMap h t*) = *Some* (*hs1*, *hs2*)
**by** (*insert regSkelOuter2-tMap-h-0* [*of h t*], *auto*)

## 4.8 Two-Dimensional Regular Tables

**constdefs**
*regularOuter2* :: (′*h1*,(′*h2*,′*t*) *T*) *T* ⇒ *bool*
*regularOuter2 t* == *option False* (% *x* . *True*) (*regSkelOuter2 t*)

**lemma** *regularOuter2I*[*simp*,*intro*]:
*regSkelOuter2 t* = *Some* (*hs1*, *hs2*) ⟹ *regularOuter2 t*
**by** (*simp add*: *regularOuter2-def*)

**lemma** *regularOuter2*[*simp*,*dest*]:
*regularOuter2 t* ⟹ *EX hs1 hs2* . *regSkelOuter2 t* = *Some* (*hs1*, *hs2*)
**by** (*case-tac regSkelOuter2 t*, *simp-all add*: *regularOuter2-def*)

**typedef** (*regT2*) (′*h1*,′*h2*,′*t*) *regT2* =
{*t* :: (′*h1*,(′*h2*,′*t*) *T*) *T* . *regularOuter2 t*}
**apply** (*rule-tac x*=*addH arbitrary* (*addH arbitrary arbitrary*) **in** *exI*)
**apply** (*simp add*: *regularOuter2-def*)
**done**

**lemma** *regT2*[*simp*,*intro*!]: *regSkelOuter2 t* = *Some* (*hs1*, *hs2*) ⟹ *t* ∈ *regT2*
**by** (*simp add*: *regT2-def*, *fast*)

**consts**
*regSkel2* :: (′*h1*,′*h2*,′*t*) *regT2* ⇒ ′*h1 neList* ∗ ′*h2 neList*
*reg2dim1* :: (′*h1*,′*h2*,′*t*) *regT2* ⇒ ′*h1 neList*
*reg2dim2* :: (′*h1*,′*h2*,′*t*) *regT2* ⇒ ′*h2 neList*

**defs**
*regSkel2-def*: *regSkel2 t* == *option arbitrary id* (*regSkelOuter2* (*Rep-regT2 t*))

**lemma** *regSkel2*[*simp*]:
*regSkelOuter2 t* = *Some* (*hs1*, *hs2*) ⟹ *regSkel2* (*Abs-regT2 t*) = (*hs1*, *hs2*)
**apply** (*simp add*: *regSkel2-def regT2*)
**apply** (*subst Abs-regT2-inverse*, *fast*)
**apply** *simp*
**done**

**defs**
*reg2dim1-def*: *reg2dim1* == *fst o regSkel2*
*reg2dim2-def*: *reg2dim2* == *snd o regSkel2*

**lemma** *reg2dim1*[*simp*]:
*regSkelOuter2 t* = *Some* (*hs1*, *hs2*) ⟹ *reg2dim1* (*Abs-regT2 t*) = *hs1*
**by** (*auto simp add*: *reg2dim1-def dest*: *regSkel2*)

**lemma** *reg2dim2*[*simp*]:
*regSkelOuter2 t* = *Some* (*hs1*, *hs2*) ⟹ *reg2dim2* (*Abs-regT2 t*) = *hs2*
**by** (*auto simp add*: *reg2dim2-def dest*: *regSkel2*)

## 4.9 List Interface

**constdefs**
*tList* :: (′*h*, ′*t*) *T* ⇒ (′*h* ∗ ′*t*) *neList*
*tList* == *tFold* (% *h t* . *singleton* (*h*,*t*)) *append*
*tOfList* :: (′*h* ∗ ′*t*) *neList* ⇒ (′*h*, ′*t*) *T*
*tOfList* == *foldr1 hConc o map1* (*uncurry addH*)
*zipT* :: ′*h neList* ⇒ ′*t neList* ⇒ (′*h*, ′*t*) *T*
*zipT hs ts* == *foldr1 hConc* (*zipWith1 addH hs ts*)

**lemma** *tList-addH*[*simp*]: *tList* (*addH h t*) = *singleton* (*h*,*t*)
**by** (*simp add*: *tList-def*)

**lemma** *tList-hConc*[*simp*]: *tList* (*hConc t1 t2*) = *append* (*tList t1*) (*tList t2*)
**by** (*simp add*: *tList-def*)

**lemma** *tList-hCons*[*simp*]: *tList* (*hCons p t2*) = *cons1 p* (*tList t2*)
**by** (*simp add*: *hCons cons1*)

**lemma** *tOfList-singleton*[*simp*]: *tOfList* (*singleton* (*h*,*t*)) = *addH h t*
**by** (*simp add*: *tOfList-def*)

**lemma** *tOfList-append*[*simp*]: *tOfList* (*append t1 t2*) = *hConc* (*tOfList t1*) (*tOfList t2*)
**by** (*simp add*: *tOfList-def*)

**lemma** *tOfList-cons1*[*simp*]: *tOfList* (*cons1 p ps*) = *hCons p* (*tOfList ps*)
**by** (*simp add*: *tOfList-def cons1 hCons*)

## 4.10 ZipWith

**constdefs**
*tZipWith* :: ((′*h1* ∗ ′*t1*) ⇒ (′*h2* ∗ ′*t2*) ⇒ (′*h* ∗ ′*t*)) ⇒
(′*h1*, ′*t1*) *T* ⇒ (′*h2*, ′*t2*) *T* ⇒ (′*h*, ′*t*) *T*
*tZipWith f t1 t2* ==
*foldr1 hConc* (*map1* (*uncurry addH*) (*zipWith1 f* (*tList t1*) (*tList t2*)))

**lemma** *tZipWith-A-A*[*simp*]:
*tZipWith f* (*addH h1 t1*) (*addH h2 t2*) = *uncurry addH* (*f* (*h1*,*t1*) (*h2*,*t2*))
**by** (*simp add*: *tZipWith-def*)

**lemma** *tZipWith-A-C*[*simp*]:
*tZipWith f* (*addH h1 t1*) (*hCons p2 t2*) = *uncurry addH* (*f* (*h1*,*t1*) *p2*)
**by** (*simp add*: *tZipWith-def*)

**lemma** *tZipWith-C-A*[*simp*]:
*tZipWith f* (*hCons p1 t1*) (*addH h2 t2*) = *uncurry addH* (*f p1* (*h2*,*t2*))
**by** (*simp add*: *tZipWith-def*)

**lemma** *tZipWith-C-C*[*simp*]:
*tZipWith f* (*hCons p1 t1*) (*hCons p2 t2*) = *hCons* (*f p1 p2*) (*tZipWith f t1 t2*)
**proof** −
**have** *tZipWith f* (*hCons p1 t1*) (*hCons p2 t2*) = *foldr1 hConc*
(*map1* (λ*p*. *addH* (*fst p*) (*snd p*))
(*zipWith1 f* (*append* (*singleton p1*) (*tList t1*))
(*append* (*singleton p2*) (*tList t2*))))
**by** (*simp add*: *tZipWith-def hCons*)
**also have** ... = *hCons* (*f p1 p2*) (*tZipWith f t1 t2*)
**by** (*simp add*: *tZipWith-def hCons cons1* [*THEN sym*])
**finally show** *?thesis* **.**
**qed**

**lemma** *tZipWith-A*[*simp*]:
*tZipWith f* (*addH h1 t1*) *t2* = *uncurry addH* (*f* (*h1*,*t1*) (*hHead t2*))
**by** (*induct-tac t2 rule*: *T-cons-induct*, *simp-all*)

**lemma** *tZipWith-C*[*simp*]:
*hHead* (*tZipWith f* (*hCons p1 t1*) *t2*) = *f p1* (*hHead t2*)
**by** (*induct-tac t2 rule*: *T-cons-induct*, *simp-all*)

**lemma** *tZipWith--A*[*simp*]:
*tZipWith f t1* (*addH h2 t2*) = *uncurry addH* (*f* (*hHead t1*) (*h2*,*t2*))
**by** (*induct-tac t1 rule*: *T-cons-induct*, *simp-all*)

**lemma** *tZipWith--C*[*simp*]:
*hHead* (*tZipWith f t1* (*hCons p2 t2*)) = *f* (*hHead t1*) *p2*
**by** (*induct-tac t1 rule*: *T-cons-induct*, *simp-all*)

**lemma** *tZipWith-assoc-0*[*simp*]:
*assoc f* ⟹
*ALL t1 t2 t3* . *tZipWith f* (*tZipWith f t1 t2*) *t3* = *tZipWith f t1* (*tZipWith f t2 t3*)
**apply** (*rule T-cons-inductA*)
**apply** (*rule T-cons-inductA*, *simp-all*)
**apply** (*rule T-cons-inductA*, *simp-all*)
**apply** (*rule T-cons-inductA*, *simp-all*)

**done**

**lemma** *assoc-tZipWith*[*simp*]: *assoc f* $\Longrightarrow$ *assoc* (*tZipWith f*)
**by** (*rule assoc-intro*, *simp*)

## 4.11 Collapsing

**constdefs**
*collapse* :: ('*h1* ⇒ '*t1* ⇒ ('*h2*, '*t2*) *T*) ⇒ ('*h1*, '*t1*) *T* ⇒ ('*h2*, '*t2*) *T*
*collapse f* == *tFold f hConc*

**lemma** *collapse-addH*[*simp*]: *collapse f* (*addH h t*) = *f h t*
**by** (*simp add*: *collapse-def*)

**lemma** *collapse-hConc*[*simp*]:
*collapse f* (*hConc t1 t2*) = *hConc* (*collapse f t1*) (*collapse f t2*)
**by** (*simp add*: *collapse-def*)

**constdefs**
*collapse2* :: ('*h1* ⇒ '*h2* ⇒ '*h3*) ⇒ ('*h1*, ('*h2*, '*t2*) *T*) *T* ⇒ ('*h3*, '*t2*) *T*
*collapse2 f* == *collapse* (% *h1* . *hMap* (*f h1*))

**lemma** *collapse2-addH*[*simp*]: *collapse2 f* (*addH h t*) = *hMap* (*f h*) *t*
**by** (*simp add*: *collapse2-def*)

**lemma** *collapse2-hConc*[*simp*]:
*collapse2 f* (*hConc t1 t2*) = *hConc* (*collapse2 f t1*) (*collapse2 f t2*)
**by** (*simp add*: *collapse2-def*)

**theorem** *collapse2-tFold2*:
⟦ *assoc c1*; *assoc c2*;
⋀ *h1 h2 x* . *a1 h1* (*a2 h2 x*) = *a3* (*f h1 h2*) *x*;
⋀ *h x y* . *a1 h* (*c2 x y*) = *c1* (*a1 h x*) (*a1 h y*)
⟧ $\Longrightarrow$
*tFold a1 c1* (*tMap* (*tFold a2 c2*) *t*) = *tFold a3 c1* (*collapse2 f t*)
**apply** (*induct-tac t rule*: *T-induct*, *simp*)
**apply** (*induct-tac t0 rule*: *T-induct*, *simp-all*)
**done**

**theorem** *collapse2-tFold2-wrapped*:
⟦ *assoc c1*; *assoc c2*; *assoc c3*;
⋀ *h1 h2 x* . *w1* (*a1 h1* (*a2 h2 x*)) = *w3* (*a3* (*f h1 h2*) *x*);
⋀ *h x y* . *w1* (*a1 h* (*c2 x y*)) = *c4* (*w1* (*a1 h x*)) (*w1* (*a1 h y*));
⋀ *x y* . *w1* (*c1 x y*) = *c5* (*w1 x*) (*w1 y*);
(* *These swapped equations look more intuitive*, *but destroy simplification*:
⋀ *x y* . *w3* (*c3 x y*) = *c4* (*w3 x*) (*w3 y*);
⋀ *x y* . *w3* (*c3 x y*) = *c5* (*w3 x*) (*w3 y*)
*)
⋀ *x y* . *c4* (*w3 x*) (*w3 y*) = *w3* (*c3 x y*);

$\bigwedge x\ y \quad .\ c5\ (w3\ x)\ (w3\ y) = w3\ (c3\ x\ y)$
$]\!] \Longrightarrow$
$w1\ (tFold\ a1\ c1\ (tMap\ (tFold\ a2\ c2)\ t)) = w3\ (tFold\ a3\ c3\ (collapse2\ f\ t))$
**apply** (*induct-tac t rule*: *T-induct*, *simp*)
**apply** (*induct-tac t0 rule*: *T-induct*, *simp-all del*: *tFold-tMap*)
**done**

### 4.12 Compression

**lemma** *hCompress-0*:
$[\![\ assoc\ c;$
$\bigwedge h1\ h2\ t1\ t2\ .\ c\ (a\ h1\ t1)\ (a\ h2\ t2) = a\ (c'\ h1\ h2)\ (c''\ t1\ t2)$
$]\!] \Longrightarrow$
$tFold\ a\ c\ (hCons\ (h1,t1)\ (hCons\ (h2,t2)\ t)) =$
$tFold\ a\ c\ (hCons\ (c'\ h1\ h2,\ c''\ t1\ t2)\ t)$
**by** (*simp del*: *assoc add*: *assoc* [*THEN sym*])

**consts**
*dim1addH* :: $('h \Rightarrow 'c \Rightarrow 'v) \Rightarrow ('h \Rightarrow ('c,'u)\ T \Rightarrow 'v)$

**defs**
*dim1addH-def*: *dim1addH a h* == *tFold* (% *c u* . *a h c*) *arbitrary*

**lemma** *dim1addH*[*simp*]: *dim1addH a h* (*cell c*) = *a h c*
**by** (*simp add*: *dim1addH-def cell-def*)

**lemma** *hCompress-cells*:
$[\![\ assoc\ c;$
$\bigwedge h1\ h2\ g1\ g2\ .\ c\ (a\ h1\ g1)\ (a\ h2\ g2) = a\ (c'\ h1\ h2)\ (c''\ g1\ g2)$
$]\!] \Longrightarrow$
$tFold\ (dim1addH\ a)\ c\ (hCons\ (h1,cell\ g1)\ (hCons\ (h2,cell\ g2)\ t)) =$
$tFold\ (dim1addH\ a)\ c\ (hCons\ (c'\ h1\ h2,\ cell\ (c''\ g1\ g2))\ t)$
**by** (*simp del*: *assoc add*: *assoc* [*THEN sym*])

### 4.13 Elementary Transformations

The titles of the subsections here are the structural elementary transformations as listed in [**?**].

The lemmas show the corresponding general properties for the first dimension; for other dimensions; the corresponding properties can be obtained using theorem *tTranspose-tFold2*.

#### 4.13.1 Permuting two $(-1)$-slices with their corresponding header entries

**lemma** *hCommute*:
$[\![\ assoc\ c;\ commutative\ c\ ]\!] \Longrightarrow$
$tFold\ a\ c\ (hConc\ t1\ t2) = tFold\ a\ c\ (hConc\ t2\ t1)$

**by** (*simp*, *erule commutative*)

#### 4.13.2 Deleting a $(-1)$-slice with "false" in the corresponding header entry

Since Zucker also does not allow empty tables, this deletion is only possible for tables that are in the range of *hConc*.

**lemma** *hDelLeftUnitHeader*:
$\llbracket$ *assoc c*; $\bigwedge$ *t1 b . c* (*a h1 t1*) *b* = *b* $\rrbracket \Longrightarrow$
*tFold a c* (*hConc* (*addH h1 t1*) *t*) = *tFold a c t*
**by** *simp*

**lemma** *hDelLeftUnitHeader-hCons*:
$\llbracket$ *assoc c*; $\bigwedge$ *t1 b . c* (*a h1 t1*) *b* = *b* $\rrbracket \Longrightarrow$
*tFold a c* (*hCons* (*h1*,*t1*) *t*) = *tFold a c t*
**by** *simp*

**lemma** *hDelRightUnitHeader*:
$\llbracket$ *assoc c*; $\bigwedge$ *t1 b . c b* (*a h1 t1*) = *b* $\rrbracket \Longrightarrow$
*tFold a c* (*hConc t* (*addH h1 t1*)) = *tFold a c t*
**by** *simp*

#### 4.13.3 Deleting a principal slice with only "false" entries from an inverted table

**lemma** *hDelLeftUnitSubtable*:
$\llbracket$ *assoc c*; $\bigwedge$ *h1 b . c* (*a h1 t1*) *b* = *b* $\rrbracket \Longrightarrow$
*tFold a c* (*hConc* (*addH h1 t1*) *t*) = *tFold a c t*
**by** *simp*

**lemma** *hDelLeftUnitSubtable-hCons*:
$\llbracket$ *assoc c*; $\bigwedge$ *h1 b . c* (*a h1 t1*) *b* = *b* $\rrbracket \Longrightarrow$
*tFold a c* (*hCons* (*h1*,*t1*) *t*) = *tFold a c t*
**by** *simp*

**lemma** *hDelRightUnitSubtable*:
$\llbracket$ *assoc c*; $\bigwedge$ *h1 b . c b* (*a h1 t1*) = *b* $\rrbracket \Longrightarrow$
*tFold a c* (*hConc t* (*addH h1 t1*)) = *tFold a c t*
**by** *simp*

#### 4.13.4 Splitting a principal slice by "splitting a disjunction" in the corresponding header in an inverted table

This is strange: the corresponding header of a principal slice in an inverted table is a value header!

Another thing to keep in mind here is that splitting a condition header in a normal or inverted table may turn a proper table into an improper table.

**lemma** *hSplitHeader*:
  $\llbracket$ *assoc c*; $\bigwedge$ *h1 h2 t . a (c' h1 h2) t = c (a h1 t) (a h2 t)* $\rrbracket \Longrightarrow$
  *tFold a c (addH (c' h1 h2) t) =*
  *tFold a c (hConc (addH h1 t) (addH h2 t))*
**by** *simp*

### 4.13.5 Combining two or more principal slices with the same value header entry into a single slice in an inverted table

**lemma** *hCombineEqualHeaders*:
  $\llbracket$ *assoc c*; $\bigwedge$ *h t1 t2 . c (a h t1) (a h t2) = a h (c' t1 t2)* $\rrbracket \Longrightarrow$
  *tFold a c (hConc (addH h t1) (addH h t2)) =*
  *tFold a c (addH h (c' t1 t2))*
**by** *simp*

**end**

# 5 Functions Interacting Directly with the Second Table Dimension

**theory** *Tables2 = Tables*:

## 5.1 Construction in the Second Dimension

**constdefs**
  *addH2 :: 'h2 ⇒ ('h1, 't) T ⇒ ('h1, ('h2, 't) T) T*
  *addH2 h2 == tFold (% h1 t . addH h1 (addH h2 t)) hConc*

**lemma** *addH2-addH*[*simp*]: *addH2 h2 (addH h1 t) = addH h1 (addH h2 t)*
**by** (*simp add*: *addH2-def*)

**lemma** *addH2-hConc*[*simp*]: *addH2 h2 (hConc t1 t2) = hConc (addH2 h2 t1) (addH2 h2 t2)*
**by** (*simp add*: *addH2-def*)

**lemma** *addH2-hCons*[*simp*]: *addH2 h2 (hCons p t) = hCons (fst p, addH h2 (snd p)) (addH2 h2 t)*
**by** (*simp add*: *addH2-def hCons*)

**lemma** *hHead-addH2*[*simp*]: *hHead (addH2 h t) = (fst (hHead t), addH h (snd (hHead t)))*
**apply** (*simp add*: *addH2-def*)
**apply** (*induct-tac t rule*: *T-cons-induct*)
**apply** *simp-all*
**done**

**lemma** *addH2-tMap*: *addH2 h = tMap (addH h)*
**by** (*simp add*: *addH2-def tMap-def*)

**lemma** *tFold-addH2*[*simp*]:
*assoc c* $\Longrightarrow$ *tFold a c* (*addH2 h t*) = *tFold* ($\lambda$*ha t0*. *a ha* (*addH h t0*)) *c t*
**by** (*simp add*: *addH2-tMap*)

**lemma** *tFold-tFold-addH2*:
⟦ *assoc c2* ⟧ $\Longrightarrow$
*tFold a2 c2* (*tMap* (*tFold a1 c1*) (*addH2 h t*)) =
*tFold* ($\lambda$ *h0 t0* . *a2 h0* (*a1 h t0*)) *c2 t*
**by** (*induct-tac t rule*: *T-induct*, *simp-all*)

**lemma** *tMap-tFold-addH2*:
*assoc c1* $\Longrightarrow$
*tMap* (*tFold a1 c1*) (*addH2 h t*) = *tFold* ($\lambda$ *h0 t0* . *addH h0* (*a1 h t0*)) *hConc t*
**by** (*induct-tac t rule*: *T-induct*, *simp-all*)

**constdefs**
*vConc0* :: ('*h1* ∗ ('*h2*, '*t*) *T*) ⇒ ('*h1* ∗ ('*h2*, '*t*) *T*) ⇒ ('*h1* ∗ ('*h2*, '*t*) *T*)
*vConc0* == % *p1 p2* . (*fst p1*, *hConc* (*snd p1*) (*snd p2*))

**lemma** *vConc0*[*simp*]: *vConc0* (*h1*,*t1*) (*h2*,*t2*) = (*h1*, *hConc t1 t2*)
**by** (*simp add*: *vConc0-def*)

**lemma** *fst-vConc0*[*simp*]: *fst* (*vConc0 p1 p2*) = *fst p1*
**by** (*simp add*: *vConc0-def*)

**lemma** *snd-vConc0*[*simp*]: *snd* (*vConc0 p1 p2*) = *hConc* (*snd p1*) (*snd p2*)
**by** (*simp add*: *vConc0-def*)

**lemma** *vConc0-assoc*[*simp*]: *vConc0* (*vConc0 p1 p2*) *p3* = *vConc0 p1* (*vConc0 p2 p3*)
**proof** −
**have** *vConc0* (*vConc0 p1 p2*) *p3* = *vConc0* (*vConc0* (*fst p1*, *snd p1*) (*fst p2*, *snd p2*)) (*fst p3*, *snd p3*) **by** *simp*
**also have** ... = *vConc0* (*fst p1*, *snd p1*) (*vConc0* (*fst p2*, *snd p2*) (*fst p3*, *snd p3*)) **by** (*simp only*: *vConc0 hConc-assoc*)
**also have** ... = *vConc0 p1* (*vConc0 p2 p3*) **by** *simp*
**finally show** *?thesis* **.**
**qed**

**lemma** *assoc-vConc0*[*simp*]: *assoc vConc0*
**by** (*rule assoc-intro*, *simp*)

**constdefs**
*vConc* :: ('*h1*, ('*h2*, '*t*) *T*) *T* ⇒ ('*h1*, ('*h2*, '*t*) *T*) *T* ⇒ ('*h1*, ('*h2*, '*t*) *T*) *T*
*vConc* == *tZipWith vConc0*

**lemma** *assoc-vConc*[*simp*]: *assoc vConc*

**by** (*simp add*: *vConc-def*)

**lemma** *vConc-addH*[*simp*]:
*vConc* (*addH h t1*) *t2* = *uncurry addH* (*vConc0* (*h*, *t1*) (*hHead t2*))
**by** (*simp add*: *vConc-def*)

**lemma** *hHead-vConc-hCons*[*simp*]:
*hHead* (*vConc* (*hCons p1 t1*) *t2*) = *vConc0 p1* (*hHead t2*)
**by** (*simp add*: *vConc-def*)

**lemma** *vConc--addH*[*simp*]:
*vConc t1* (*addH h t2*) = *uncurry addH* (*vConc0* (*hHead t1*) (*h*, *t2*))
**by** (*simp add*: *vConc-def*)

**lemma** *hHead-vConc--hCons*[*simp*]:
*hHead* (*vConc t1* (*hCons p2 t2*)) = *vConc0* (*hHead t1*) *p2*
**by** (*simp add*: *vConc-def*)

**lemma** *vConc-hCons-hCons*[*simp*]:
*vConc* (*hCons p1 t1*) (*hCons p2 t2*) = *hCons* (*vConc0 p1 p2*) (*vConc t1 t2*)
**by** (*simp add*: *vConc-def*)

**constdefs**
*vCons* :: (*'h2* ∗ (*'h1*, *'t*) *T*) ⇒ (*'h1*, (*'h2*, *'t*) *T*) *T* ⇒ (*'h1*, (*'h2*, *'t*) *T*) *T*
*vCons p t* == *vConc* (*uncurry addH2 p*) *t*

**lemma** *vCons*: *vCons* (*h*,*t0*) *t* = *vConc* (*addH2 h t0*) *t*
**by** (*simp add*: *vCons-def*)

**lemma** *vCons-addH*[*simp*]:
*vCons p* (*addH h t*) = *addH* (*fst* (*hHead* (*uncurry addH2 p*)))
(*hConc* (*snd* (*hHead* (*uncurry addH2 p*))) *t*)
**by** (*simp add*: *vCons-def*)

**lemma** *hHead-vCons-hCons*[*simp*]:
*hHead* (*vCons p1* (*hCons p2 t2*)) = *vConc0* (*hHead* (*uncurry addH2 p1*)) *p2*
**by** (*simp add*: *vCons-def*)

**lemma** *vCons-hCons-hCons*[*simp*]:
*vCons* (*h*, *hCons p1 t1*) (*hCons p2 t2*) =
*hCons* (*vConc0* (*fst p1*, *addH h* (*snd p1*)) *p2*) (*vCons* (*h*, *t1*) *t2*)
**by** (*simp add*: *vCons-def vConc-def*)

**lemma** *headers-addH2*[*simp*]: *headers* (*addH2 h t*) = *headers t*
**by** (*induct-tac t rule*: *T-induct*, *simp-all*)

**lemma** *headers-vCons-0*:
*ALL t p* . *headers* (*snd p*) = *headers t* ⟶ *headers* (*vCons p t*) = *headers t*
**apply** (*rule T-cons-inductA*, *simp-all*)

**apply** (*intro strip*, *drule headers-eq-singleton*)
**apply** (*erule exE*, *simp*)
**apply** (*intro strip*, *drule headers-eq-cons1*)
**apply** (*erule exE*, *erule exE*, *erule conjE*, *simp*)
**done**

**lemma** *headers-vCons*[*simp*]:
*headers* (*snd p*) = *headers t* ⟹ *headers* (*vCons p t*) = *headers t*
**apply** (*insert headers-vCons-0*)
**apply** (*drule-tac x=t* **in** *spec*)
**apply** (*drule-tac x=p* **in** *spec*)
**apply** *simp*
**done**

**lemma** *tMap-h-vConc-0*:
⟦ ⋀ *t* . *headers* (*h t*) = *headers t*;
⋀ *t* . *h* (*h t*) = *h t*;
⋀ *t1 t2* . *h* (*hConc t1 t2*) = *hConc* (*h t1*) (*h t2*)
⟧ ⟹
*ALL t2 hs1 hs2a hs2b* .
*regSkelOuter2 t1* = *Some* (*hs1*, *hs2a*) ⟶
*regSkelOuter2 t2* = *Some* (*hs1*, *hs2b*) ⟶
*tMap h* (*vConc t1 t2*) = *vConc* (*tMap h t1*) (*tMap h t2*)
**apply** (*induct-tac t1 rule*: *T-cons-induct*, *simp-all*)
**apply** (*rule allI*)
**apply** (*induct-tac t2 rule*: *T-cons-induct*, *simp-all*)
**apply** (*rule allI*)
**apply** (*induct-tac t2a rule*: *T-cons-induct*, *simp-all*)
**apply** (*intro strip*, *case-tac p*, *simp*)
**apply** (*erule exE*, *drule optThen-result-Some*, *erule exE*, *erule conjE*)
**apply** (*split split-if-asm*) **prefer** *2* **apply** *simp*
**apply** (*drule optThen-result-Some*, *erule exE*, *erule conjE*)
**apply** (*split split-if-asm*) **prefer** *2* **apply** *simp*
**apply** (*case-tac y*, *case-tac ya*, *case-tac pa*, *simp*)
**apply** (*erule conjE*)+
**apply** (*rotate-tac −2*, *drule sym*, *simp*, *drule cons1-inj*, *erule conjE*, *simp*)
**done**

**lemma** *tMap-h-vConc*[*simp*]:
⟦ ⋀ *t* . *headers* (*h t*) = *headers t*;
⋀ *t* . *h* (*h t*) = *h t*;
⋀ *t1 t2* . *h* (*hConc t1 t2*) = *hConc* (*h t1*) (*h t2*);
*regSkelOuter2 t1* = *Some* (*hs1*, *hs2a*);
*regSkelOuter2 t2* = *Some* (*hs1*, *hs2b*)
⟧ ⟹
*tMap h* (*vConc t1 t2*) = *vConc* (*tMap h t1*) (*tMap h t2*)
**by** (*insert tMap-h-vConc-0* [*of h t1*], *auto*)

**lemma** *tMap-const-vConc-0*:

*ALL t2 hs1a hs1b hs2a hs2b .*
*regSkelOuter2 t1 = Some (hs1a, hs2a)* ⟶
*regSkelOuter2 t2 = Some (hs1b, hs2b)* ⟶
*hs1a = hs1b* ⟶
*tMap (const t) (vConc t1 t2) = tMap (const t) t1*
**apply** (*induct-tac t1 rule*: *T-cons-induct*, *simp*)
**apply** (*rule allI*)
**apply** (*induct-tac t2a rule*: *T-cons-induct*, *simp-all*)
**apply** (*intro strip*, *erule exE*)
**apply** (*drule optThen-result-Some*, *erule exE*, *erule conjE*)
**apply** (*case-tac p*, *case-tac y*, *simp*)
**apply** (*split split-if-asm*, *simp*, *simp*)
**apply** (*intro strip*, *erule exE*, *erule exE*)
**apply** (*drule optThen-result-Some*, *erule exE*, *erule conjE*)
**apply** (*drule optThen-result-Some*, *erule exE*, *erule conjE*)
**apply** (*case-tac p*, *case-tac y*, *simp*)
**apply** (*case-tac pa*, *case-tac ya*, *simp*)
**apply** (*split split-if-asm*, *simp*)
**apply** (*split split-if-asm*, *simp*)
**apply** (*erule conjE*, *rotate-tac* −*2*, *drule sym*, *simp*)
**apply** (*drule cons1-inj*, *erule conjE*, *simp-all*)
**done**

**lemma** *tMap-const-vConc*[*simp*]:
⟦ *regSkelOuter2 t1 = Some (hs1a, hs2a)*;
*regSkelOuter2 t2 = Some (hs1b, hs2b)*;
*hs1a = hs1b*
⟧ ⟹
*tMap (const t) (vConc t1 t2) = tMap (const t) t1*
**by** (*insert tMap-const-vConc-0 [of t1 t]*, *auto*)

**lemma** *tMap-const-hConc*:
*tMap (const (hConc t1 t2)) t = vConc (tMap (const t1) t) (tMap (const t2) t)*
**apply** (*cut-tac t2.0=t1* **and** *t1.0=t* **in** *regSkelOuter2-tMap-const*)
**apply** (*cut-tac t2.0=t2* **and** *t1.0=t* **in** *regSkelOuter2-tMap-const*)
**apply** (*erule mp1*)
**apply** (*erule mp1*)
**apply** (*induct-tac t rule*: *T-cons-induct*, *simp-all*)
**done**

## 5.2 Regular Skeletons and the Second Dimension

**lemma** *regSkelOuter1-addH2*[*simp*]:
*regSkelOuter1 (addH2 h t) = Some (headers t)*
**by** (*simp add*: *regSkelOuter1-def*)

**lemma** *regSkelStep-addH2*:
*regSkelStep rs (addH2 h t) = option-map (Pair (headers t)) (tFold (λh. rs) optEq (addH2 h t))*

**by** (*simp add*: *regSkelStep-def*)

**lemma** *regSkelStep-Some-0*:
*ALL hs . regSkelStep rs t = Some (hs,r) ⟶ headers t = hs*
**apply** (*induct-tac t rule*: *T-cons-induct*, *simp-all*)
**apply** (*intro strip*)
**apply** (*drule optThen-result-Some*)
**apply** (*erule exE*, *erule conjE*)
**apply** (*drule optThen-result-Some*)
**apply** (*erule exE*, *erule conjE*)
**apply** (*case-tac y = snd ya*, *simp-all*)
**apply** (*erule conjE*)
**apply** (*rotate-tac −1*, *drule sym*, *simp*)
**apply** (*drule-tac x=fst ya* **in** *spec*)
**apply** (*rotate-tac 3*, *drule sym*, *simp*)
**done**

**lemma** *regSkelStep-Some*:
*regSkelStep rs t = Some (hs,r) ⟹ headers t = hs*
**apply** (*insert regSkelStep-Some-0* [*of rs t r*])
**apply** (*drule-tac x=hs* **in** *spec*, *simp*)
**done**

**lemma** *regSkelOuter2-addH2*[*simp*]:
*regSkelOuter2 (addH2 h t) = Some (headers t, singleton h)*
**by** (*induct-tac t rule*: *T-induct*, *simp-all*)

**lemma** *regSkelOuter2-eq-Some*:
*regSkelOuter2 t = Some (hs1,hs2) ⟹ headers t = hs1*
**by** (*simp add*: *regSkelStep-Some regSkelOuter2-def*)

**lemma** *regSkelOuter2-vCons-0*:
*ALL t p . headers (snd p) = headers t ⟶*
*regSkelOuter2 (vCons p t) =*
*optThen (tFold (λh t . Some (headers t)) optEq (uncurry addH2 p)) (λ hs1 .*
*optThen (regSkelOuter2 t) (λ p2 . Some (headers t, append hs1 (snd p2))))*
**apply** (*rule T-cons-inductA*)

Case 1: *t=addH*

**apply** (*simp add*: *regSkelOuter2-def*)
**apply** (*intro strip*)
**apply** (*drule headers-eq-singleton*)
**apply** (*erule exE*)
**apply** (*simp add*: *regSkelOuter1-def regSkelOuter2-def cons1 singleton-def append-def neList-def Abs-neList-inverse*)

Case 2: *t=hCons*

**apply** (*intro strip*)
**apply** (*simp add*: *regSkelOuter2-def*)

**apply** (*drule headers-eq-cons1*)
**apply** (*erule exE*, *erule exE*, *erule conjE*)
**apply** (*subgoal-tac vCons pa* = *vCons* (*fst pa*, *hCons* (*fst p*, *t0*) *t2a*))
 **prefer** *2*
 **apply** (*drule sym*, *simp*)
**apply** (*rotate-tac* −*1*, *drule sym*, *rotate-tac* −*1*, *erule subst*)
**apply** *simp*
**apply** (*case-tac regSkelStep regSkelOuter1 t2*, *simp*)
**apply** (*simp add*: *regSkelOuter1-def*)
**apply** (*case-tac regSkelOuter1* (*snd p*), *simp*)

Case 2.1: *regSkelOuter1* (*snd p*) = *None*

**apply** (*rule append-inj2-neq-conv*)
**apply** (*drule-tac x*=*fst pa* **in** *spec*)
**apply** (*drule-tac x*=*t2a* **in** *spec*)
**apply** (*simp add*: *regSkelOuter1-def*)

Case 2.2: *regSkelOuter1* (*snd p*) = *Some aa*

**apply** *simp*
**apply** (*rule conjI*)

Case 2.2.1: *aa* = *snd a*

**apply** (*intro strip*, *rule append-inj2-conv*)
**apply** (*simp add*: *regSkelOuter1-def*)

Case 2.2.2: *aa* ≠ *snd a*

**apply** (*intro strip*, *rule append-inj2-neq-conv*)
**apply** (*simp add*: *regSkelOuter1-def*)
**done**

**lemma** *regSkelOuter2-vCons*:
 *headers* (*snd p*) = *headers t* ⟹
  *regSkelOuter2* (*vCons p t*) =
  *optThen* (*tFold* (λ*h t* . *Some* (*headers t*)) *optEq* (*uncurry addH2 p*)) (λ *hs1* .
  *optThen* (*regSkelOuter2 t*) (λ *p2* . *Some* (*headers t*, *append hs1* (*snd p2*))))
**apply** (*insert regSkelOuter2-vCons-0*)
**apply** (*drule-tac x*=*t* **in** *spec*)
**apply** (*drule-tac x*=*p* **in** *spec*)
**apply** *simp*
**done**

**lemma** *regSkelOuter2-Some-vCons*:
 ⟦ *regSkelOuter2 t* = *Some* (*hs1*, *hs2*); *headers* (*snd p*) = *headers t* ⟧ ⟹
  *regSkelOuter2* (*vCons p t*) = *Some* (*hs1*, *cons1* (*fst p*) *hs2*)
**by** (*simp add*: *regSkelOuter2-vCons cons1*)

**lemma** *regSkelOuter2-vConc-0*:
 *ALL t1 t2 hs1a hs2a hs1b hs2b* . *headers t1* = *headers t2* ⟶

```
    regSkelOuter2 t1 = Some (hs1a, hs2a) ⟶
    regSkelOuter2 t2 = Some (hs1b, hs2b) ⟶
    hs1a = hs1b ⟶
    regSkelOuter2 (vConc t1 t2) = Some (hs1a, append hs2a hs2b)
apply (rule T-cons-inductA)
apply (rule T-cons-inductA)
apply simp
apply (intro strip)
apply (simp add: regSkelOuter1-def regSkelOuter2-def)
apply (rule T-cons-inductA)
apply simp
apply simp
apply (intro strip, drule cons1-inj, erule conjE)
apply (drule-tac x=t2a in spec)
apply (simp add: regSkelOuter1-def)
apply (case-tac regSkelOuter2 t2, simp)
apply (case-tac regSkelOuter2 t2a, simp-all)
apply (case-tac headers (snd p) = snd a, simp-all)
apply (case-tac headers (snd pa) = snd aa, simp-all)
apply (erule conjE, erule conjE)
apply (drule-tac x=fst a in spec)
apply (drule-tac x=hs2a in spec)
apply (drule mp, fastsimp)
apply (drule-tac x=hs2b in spec)
apply (drule mp)
apply (subgoal-tac cons1 (fst pa) (fst a) = cons1 (fst pa) (fst aa))
  prefer 2
  apply bestsimp
apply (drule cons1-inj, erule conjE, fastsimp)
apply simp
done

lemma regSkelOuter2-vConc[simp]:
 ⟦ regSkelOuter2 t1 = Some (hs1a, hs2a);
    regSkelOuter2 t2 = Some (hs1b, hs2b);
    hs1a = hs1b
  ⟧ ⟹ regSkelOuter2 (vConc t1 t2) = Some (hs1a, append hs2a hs2b)
apply (insert regSkelOuter2-vConc-0)
apply (drule-tac x=t1 in spec)
apply (drule-tac x=t2 in spec)
apply (drule-tac x=hs1a in spec)
apply (drule-tac x=hs2a in spec)
apply (drule-tac x=hs1b in spec)
apply (drule-tac x=hs2b in spec)
apply (frule regSkelOuter2-eq-Some)
apply (rotate-tac 1, frule regSkelOuter2-eq-Some, simp)
done
```

**lemma** *headers-vConc*[*simp*]:
⟦ *regSkelOuter2 t1* = *Some* (*hs1*,*hs2a*);
*regSkelOuter2 t2* = *Some* (*hs1*,*hs2b*)
⟧ ⟹ *headers* (*vConc t1 t2*) = *headers t1*
**apply** (*frule-tac t1.0*=*t1* **and** *t2.0*=*t2* **in** *regSkelOuter2-vConc*, *assumption*)
**apply** *simp*
**apply** (*rotate-tac* −*1*, *drule regSkelOuter2-eq-Some*)
**apply** (*drule regSkelOuter2-eq-Some* [*THEN sym*], *simp*)
**done**

**lemma** *vConc-hConc-0*:
*ALL t3 h1 v1 v2 t2 t4 h2* .
*regSkelOuter2 t1* = *Some* (*h1*,*v1*) ⟶
*regSkelOuter2 t2* = *Some* (*h2*,*v1*) ⟶
*regSkelOuter2 t3* = *Some* (*h1*,*v2*) ⟶
*regSkelOuter2 t4* = *Some* (*h2*,*v2*) ⟶
*vConc* (*hConc t1 t2*) (*hConc t3 t4*) = *hConc* (*vConc t1 t3*) (*vConc t2 t4*)
**apply** (*induct-tac t1 rule*: *T-cons-induct*, *simp add*: *regSkelOuter1-def*)
**apply** (*rule allI*)
**apply** (*induct-tac t3 rule*: *T-cons-induct*, *simp add*: *regSkelOuter1-def*)
**apply** (*intro strip*, *drule singleton-inj*)
**apply** (*simp add*: *hCons-p* [*THEN sym*])

1.2

**apply** (*intro strip*, *simp*)
**apply** (*drule optThen-result-Some*, *erule exE*, *erule conjE*)
**apply** (*simp add*: *regSkelOuter1-def*)
**apply** (*case-tac p*, *simp*)
**apply** (*split split-if-asm*, *simp*, *simp*)

2

**apply** (*rule allI*)
**apply** (*erule mp1*)
**apply** (*induct-tac t3 rule*: *T-cons-induct*, *simp add*: *regSkelOuter1-def*)
**apply** (*intro strip*)
**apply** (*drule optThen-result-Some*, *erule exE*, *erule conjE*)
**apply** (*split split-if-asm*, *simp*)
**apply** (*erule conjE*, *rotate-tac* −*2*, *drule sym*, *simp*, *simp*)

2.2

**apply** (*intro strip*, *simp add*: *regSkelOuter1-def del*: *vConc-hCons-hCons*)
**apply** (*drule optThen-result-Some*, *erule exE*, *erule conjE*)+
**apply** (*case-tac y*)
**apply** (*case-tac ya*)
**apply** (*case-tac p*)
**apply** (*case-tac pa*)
**apply** (*simp del*: *vConc-hCons-hCons*)
**apply** (*split split-if-asm*, *simp*)

**apply** (*split split-if-asm*, *simp*)
**apply** (*erule conjE*, *rotate-tac* −*2*, *drule sym*, *simp*)
**apply** (*drule cons1-inj*, *erule conjE*, *simp*)
**apply** *simp*
**apply** *simp*
**done**

**lemma** *vConc-hConc*:
⟦ *regSkelOuter2 t1* = *Some* (*h1*,*v1*);
*regSkelOuter2 t2* = *Some* (*h2*,*v1*);
*regSkelOuter2 t3* = *Some* (*h1*,*v2*);
*regSkelOuter2 t4* = *Some* (*h2*,*v2*) ⟧ =⇒
*vConc* (*hConc t1 t2*) (*hConc t3 t4*) = *hConc* (*vConc t1 t3*) (*vConc t2 t4*)
**by** (*insert vConc-hConc-0* [*of t1*], *auto*)

## 5.3 Table Transposition

**constdefs**
*tTranspose* :: ('*h1*, ('*h2*, '*t*) *T*) *T* ⇒ ('*h2*, ('*h1*, '*t*) *T*) *T*
*tTranspose* == *tFold addH2 vConc*

**lemma** *tTranspose-addH*[*simp*]: *tTranspose* (*addH h t*) = *addH2 h t*
**by** (*simp add*: *tTranspose-def*)

**lemma** *tTranspose-hConc*[*simp*]: *tTranspose* (*hConc t1 t2*) = *vConc* (*tTranspose t1*) (*tTranspose t2*)
**by** (*simp add*: *tTranspose-def*)

**lemma** *tTranspose-hCons*[*simp*]: *tTranspose* (*hCons p t*) = *vCons p* (*tTranspose t*)
**by** (*simp add*: *tTranspose-def hCons vCons-def*)

**lemma** *regSkelOuter2-tTranspose-via-T-cons-induct*:
*ALL hs1 hs2* . *regSkelOuter2 t* = *Some* (*hs1*,*hs2*) −→
*regSkelOuter2* (*tTranspose t*) = *Some* (*hs2*,*hs1*)
**apply** (*induct-tac t rule*: *T-cons-induct*)
**apply** (*simp add*: *regSkelOuter1-def regSkelOuter2-def*)
**apply** (*induct-tac t0 rule*: *T-cons-induct*, *simp*, *simp*)
**apply** (*subst tTranspose-hCons*)
**apply** (*intro strip*)
**apply** (*simp only*: *regSkelOuter2-hCons*)
**apply** (*case-tac regSkelOuter2 t2*, *simp*, *simp*)
**apply** (*split split-if-asm*,*simp-all*, *erule conjE*)
**apply** (*case-tac a*, *simp*)
**apply** (*frule-tac p*=*p* **in** *regSkelOuter2-Some-vCons*)
**apply** (*simp add*: *regSkelOuter2-def*)
**apply** (*drule regSkelStep-Some*, *simp*, *simp*)
**done**

**lemma** *regSkelOuter2-tTranspose-via-T-induct*:
*ALL hs1 hs2 . regSkelOuter2 t = Some (hs1,hs2)* $\longrightarrow$
*regSkelOuter2 (tTranspose t) = Some (hs2,hs1)*
**apply** (*induct-tac t rule*: *T-induct*)
**apply** (*simp add*: *regSkelOuter1-def regSkelOuter2-def*)
**apply** (*induct-tac t0 rule*: *T-induct*, *simp*, *simp*)
**apply** (*subst tTranspose-hConc*)
**apply** (*intro strip*, *simp only*: *regSkelOuter2-hConc*)
**apply** (*case-tac regSkelOuter2 t1*, *simp*, *simp*)
**apply** (*drule optThen-result-Some*, *erule exE*, *erule conjE*)
**apply** (*case-tac snd a = snd y*, *simp-all*)
**apply** (*erule conjE*)
**apply** (*drule-tac x=fst a* **in** *spec*)
**apply** (*drule-tac x=hs2* **in** *spec*)
**apply** (*drule mp*, *fastsimp*)
**apply** (*drule-tac x=fst y* **in** *spec*)
**apply** (*drule-tac x=hs2* **in** *spec*)
**apply** (*drule mp*, *fastsimp*)
**apply** (*rotate-tac* $-4$, *drule sym*)
**apply** (*simp add*: *regSkelOuter1-def*)
**done**

**theorem** *regSkelOuter2-tTranspose*:
*regSkelOuter2 t = Some (hs1,hs2)* $\Longrightarrow$
*regSkelOuter2 (tTranspose t) = Some (hs2,hs1)*
**by** (*insert regSkelOuter2-tTranspose-via-T-induct* [*of t*], *auto*)

**lemma** *tFold-tFold-vConc-0*:
$[\![$ *assoc c1*; *assoc c2*;
$\bigwedge$ *x y z . a2 x (c1 y z) = c1 (a2 x y) (a2 x z)*;
$\bigwedge$ *x1 x2 y1 y2 . c2 (c1 x1 y1) (c1 x2 y2) = c1 (c2 x1 x2) (c2 y1 y2)*
$]\!] \Longrightarrow$
*ALL t2 hs1a hs2a hs1b hs2b .*
*regSkelOuter2 t1 = Some (hs1a,hs2a)* $\longrightarrow$
*regSkelOuter2 t2 = Some (hs1b,hs2b)* $\longrightarrow$
*hs1a = hs1b* $\longrightarrow$
*tFold a2 c2 (tMap (tFold a1 c1) (vConc t1 t2)) =*
*c1 (tFold a2 c2 (tMap (tFold a1 c1) t1))*
*(tFold a2 c2 (tMap (tFold a1 c1) t2))*
**apply** (*induct-tac t1 rule*: *T-cons-induct*)
**apply** (*rule T-cons-inductA*)

Case 1.1: *t1=addH*, *t2=addH*

**apply** (*intro strip*)
**apply** (*simp add*: *regSkelOuter1-def regSkelOuter2-def*)
**apply** (*erule conjE*, *erule conjE*)
**apply** (*subgoal-tac singleton h = singleton ha*, *drule singleton-inj*, *simp*)
**apply** *fastsimp*

Case 1.2: *t1=addH*, *t2=hCons*

**apply** (*intro strip*)
**apply** (*drule regSkelOuter2-hCons-eq-Some*)
**apply** (*erule exE*, *erule exE*, *erule exE*, *erule conjE*, *erule conjE*, *erule conjE*)
**apply** (*simp add*: *regSkelOuter2-def*)

Case 2: *t1*=*hCons*

**apply** (*rule T-cons-inductA*)

Case 2.1: *t1*=*hCons*, *t2*=*addH*

**apply** (*intro strip*)
**apply** (*drule regSkelOuter2-hCons-eq-Some*)
**apply** (*erule exE*, *erule exE*, *erule exE*, *erule conjE*, *erule conjE*, *erule conjE*)
**apply** (*drule regSkelOuter2-addH-eq-Some*, *erule conjE*)
**apply** *simp*

Case 2.2: *t1*=*hCons*, *t2*=*hCons*

**apply** (*intro strip*)
**apply** (*drule regSkelOuter2-hCons-eq-Some*)
**apply** (*erule exE*, *erule exE*, *erule exE*, *erule conjE*, *erule conjE*, *erule conjE*)
**apply** (*drule regSkelOuter2-hCons-eq-Some*)
**apply** (*erule exE*, *erule exE*, *erule exE*, *erule conjE*, *erule conjE*, *erule conjE*)
**apply** *simp*
**apply** (*drule cons1-inj*, *erule conjE*, *simp*)
**done**

**lemma** *tFold-tFold-vConc*:
⟦ *assoc c1*; *assoc c2*;
  ⋀ *x y z* . *a2 x* (*c1 y z*) = *c1* (*a2 x y*) (*a2 x z*);
  ⋀ *x1 x2 y1 y2* . *c2* (*c1 x1 y1*) (*c1 x2 y2*) = *c1* (*c2 x1 x2*) (*c2 y1 y2*);
  *regSkelOuter2 t1* = *Some* (*hs1a*,*hs2a*);
  *regSkelOuter2 t2* = *Some* (*hs1b*,*hs2b*);
  *hs1a* = *hs1b*
 ⟧ ⟹
 *tFold a2 c2* (*tMap* (*tFold a1 c1*) (*vConc t1 t2*)) =
     *c1* (*tFold a2 c2* (*tMap* (*tFold a1 c1*) *t1*))
      (*tFold a2 c2* (*tMap* (*tFold a1 c1*) *t2*))
**by** (*insert tFold-tFold-vConc-0* [*of c1 c2 a2 t1 a1*], *auto*)

**lemma** *tTranspose-tFold2-0*:
⟦ *assoc c1*; *assoc c2*;
  ⋀ *x y z* . *a1 x* (*a2 y z*) = *a2 y* (*a1 x z*);
  ⋀ *x y z* . *a1 x* (*c2 y z*) = *c2* (*a1 x y*) (*a1 x z*);
  ⋀ *x y z* . *a2 x* (*c1 y z*) = *c1* (*a2 x y*) (*a2 x z*);
  ⋀ *x1 x2 y1 y2* . *c2* (*c1 x1 y1*) (*c1 x2 y2*) = *c1* (*c2 x1 x2*) (*c2 y1 y2*)
 ⟧ ⟹
 *ALL hs1 hs2* . *regSkelOuter2 t* = *Some* (*hs1*,*hs2*) ⟶
  *tFold a2 c2* (*tMap* (*tFold a1 c1*) (*tTranspose t*)) =
  *tFold a1 c1* (*tMap* (*tFold a2 c2*) *t*)
**apply** (*induct-tac t rule*: *T-induct*)

```
apply (induct-tac t0 rule: T-induct)
apply (intro strip, simp)
apply (intro strip, simp add: regSkelOuter1-def regSkelOuter2-def)
apply (intro strip)
apply (drule regSkelOuter2-hConc-eq-Some)
apply (erule exE, erule exE, erule conjE, erule conjE)
apply (simp del: tFold-tMap)
apply (subst tFold-tFold-vConc)
  apply (assumption, assumption, simp, simp)
  apply (erule regSkelOuter2-tTranspose)
  apply (erule regSkelOuter2-tTranspose)
  apply simp
apply simp
done

theorem tTranspose-tFold2:
 ⟦ assoc c1; assoc c2;
    ⋀ x y z . a1 x (a2 y z) = a2 y (a1 x z);
    ⋀ x y z . a1 x (c2 y z) = c2 (a1 x y) (a1 x z);
    ⋀ x y z . a2 x (c1 y z) = c1 (a2 x y) (a2 x z);
    ⋀ x1 x2 y1 y2 . c2 (c1 x1 y1) (c1 x2 y2) = c1 (c2 x1 x2) (c2 y1 y2);
    regSkelOuter2 t = Some (hs1,hs2)
  ⟧ ⟹
   tFold a2 c2 (tMap (tFold a1 c1) (tTranspose t)) =
   tFold a1 c1 (tMap (tFold a2 c2) t)
by (insert tTranspose-tFold2-0 [of c1 c2 a1 a2 t], auto)

lemma tFold-tFold-vConc-gen-0:
 ⟦ assoc c1; assoc c2; assoc c3;
    ⋀ x y z . a3 x (c2 y z) = c1 (a1 x y) (a1 x z);
    ⋀ x1 x2 y1 y2 . c3 (c1 x1 y1) (c1 x2 y2) = c1 (c1 x1 x2) (c1 y1 y2)
  ⟧ ⟹
  ALL t2 hs1a hs2a hs1b hs2b .
  regSkelOuter2 t1 = Some (hs1a,hs2a) ⟶
  regSkelOuter2 t2 = Some (hs1b,hs2b) ⟶
  hs1a = hs1b ⟶
  tFold a3 c3 (tMap (tFold a2 c2) (vConc t1 t2)) =
          c1 (tFold a1 c1 (tMap (tFold a2 c2) t1))
            (tFold a1 c1 (tMap (tFold a2 c2) t2))
apply (induct-tac t1 rule: T-cons-induct)
apply (rule T-cons-inductA)
```

Case 1.1: *t1=addH*, *t2=addH*

```
apply (intro strip)
apply (simp add: regSkelOuter1-def regSkelOuter2-def)
apply (erule conjE, erule conjE)
apply (subgoal-tac singleton h = singleton ha, drule singleton-inj, simp)
apply fastsimp
```

Case 1.2: *t1=addH*, *t2=hCons*

**apply** (*intro strip*)
**apply** (*drule regSkelOuter2-hCons-eq-Some*)
**apply** (*erule exE*, *erule exE*, *erule exE*, *erule conjE*, *erule conjE*, *erule conjE*)
**apply** (*simp add*: *regSkelOuter2-def*)

Case 2: *t1*=*hCons*

**apply** (*rule T-cons-inductA*)

Case 2.1: *t1*=*hCons*, *t2*=*addH*

**apply** (*intro strip*)
**apply** (*drule regSkelOuter2-hCons-eq-Some*)
**apply** (*erule exE*, *erule exE*, *erule exE*, *erule conjE*, *erule conjE*, *erule conjE*)
**apply** (*drule regSkelOuter2-addH-eq-Some*, *erule conjE*)
**apply** *simp*

Case 2.2: *t1*=*hCons*, *t2*=*hCons*

**apply** (*intro strip*)
**apply** (*drule regSkelOuter2-hCons-eq-Some*)
**apply** (*erule exE*, *erule exE*, *erule exE*, *erule conjE*, *erule conjE*, *erule conjE*)
**apply** (*drule regSkelOuter2-hCons-eq-Some*)
**apply** (*erule exE*, *erule exE*, *erule exE*, *erule conjE*, *erule conjE*, *erule conjE*)
**apply** *simp*
**apply** (*drule cons1-inj*, *erule conjE*, *simp*)
**done**

**lemma** *tFold-tFold-vConc-gen*:
⟦ *assoc c1*; *assoc c2*; *assoc c3*;
  ⋀ *x y z* . *a3 x* (*c2 y z*) = *c1* (*a1 x y*) (*a1 x z*);
  ⋀ *x1 x2 y1 y2* . *c3* (*c1 x1 y1*) (*c1 x2 y2*) = *c1* (*c1 x1 x2*) (*c1 y1 y2*);
  *regSkelOuter2 t1* = *Some* (*hs1a*,*hs2a*);
  *regSkelOuter2 t2* = *Some* (*hs1b*,*hs2b*);
  *hs1a* = *hs1b*
 ⟧ ⟹
 *tFold a3 c3* (*tMap* (*tFold a2 c2*) (*vConc t1 t2*)) =
   *c1* (*tFold a1 c1* (*tMap* (*tFold a2 c2*) *t1*))
    (*tFold a1 c1* (*tMap* (*tFold a2 c2*) *t2*))
**by** (*insert tFold-tFold-vConc-gen-0* [*of c1 c2 c3 a3 a1 t1*], *auto*)

**lemma** *tFold-tFold-vConc-wrapped-0*:
⟦ *assoc c1*; *assoc c2*; *assoc c3*;
  ⋀ *h x y* . *w1* (*a3 h* (*c2 x y*)) = *w2* (*c1* (*a1 h x*) (*a1 h y*));
  ⋀ *x y* . *w1* (*c3 x y*) = *c4* (*w1 x*) (*w1 y*);
  ⋀ *x1 x2 y1 y2* . *c4* (*w2* (*c1 x1 x2*)) (*w2* (*c1 y1 y2*)) =
    *w2* (*c1* (*c1 x1 y1*) (*c1 x2 y2*))
 ⟧ ⟹
 *ALL t2 hs1a hs2a hs1b hs2b* .
 *regSkelOuter2 t1* = *Some* (*hs1a*,*hs2a*) ⟶

*regSkelOuter2 t2 = Some (hs1b,hs2b) ⟶*
*hs1a = hs1b ⟶*
*w1 (tFold a3 c3 (tMap (tFold a2 c2) (vConc t1 t2))) =*
*w2 (c1 (tFold a1 c1 (tMap (tFold a2 c2) t1))*
*(tFold a1 c1 (tMap (tFold a2 c2) t2)))*

**apply** (*induct-tac t1 rule*: *T-cons-induct*)
**apply** (*rule T-cons-inductA*)

Case 1.1: *t1=addH*, *t2=addH*

**apply** (*intro strip*)
**apply** (*simp add*: *regSkelOuter1-def regSkelOuter2-def*)
**apply** (*erule conjE*, *erule conjE*)
**apply** (*subgoal-tac singleton h = singleton ha*, *drule singleton-inj*, *simp*)
**apply** *fastsimp*

Case 1.2: *t1=addH*, *t2=hCons*

**apply** (*intro strip*)
**apply** (*drule regSkelOuter2-hCons-eq-Some*)
**apply** (*erule exE*, *erule exE*, *erule exE*, *erule conjE*, *erule conjE*, *erule conjE*)
**apply** (*simp add*: *regSkelOuter2-def*)

Case 2: *t1=hCons*

**apply** (*rule T-cons-inductA*)

Case 2.1: *t1=hCons*, *t2=addH*

**apply** (*intro strip*)
**apply** (*drule regSkelOuter2-hCons-eq-Some*)
**apply** (*erule exE*, *erule exE*, *erule exE*, *erule conjE*, *erule conjE*, *erule conjE*)
**apply** (*drule regSkelOuter2-addH-eq-Some*, *erule conjE*)
**apply** *simp*

Case 2.2: *t1=hCons*, *t2=hCons*

**apply** (*intro strip*)
**apply** (*drule regSkelOuter2-hCons-eq-Some*)
**apply** (*erule exE*, *erule exE*, *erule exE*, *erule conjE*, *erule conjE*, *erule conjE*)
**apply** (*drule regSkelOuter2-hCons-eq-Some*)
**apply** (*erule exE*, *erule exE*, *erule exE*, *erule conjE*, *erule conjE*, *erule conjE*)
**apply** *simp*
**apply** (*drule cons1-inj*, *erule conjE*, *simp*)
**done**

**lemma** *tFold-tFold-vConc-wrapped*:
⟦ *assoc c1*; *assoc c2*; *assoc c3*;
⋀ *h x y . w1 (a3 h (c2 x y)) = w2 (c1 (a1 h x) (a1 h y))*;
⋀ *x y . w1 (c3 x y) = c4 (w1 x) (w1 y)*;
⋀ *x1 x2 y1 y2 . c4 (w2 (c1 x1 x2)) (w2 (c1 y1 y2)) =*
*w2 (c1 (c1 x1 y1) (c1 x2 y2))*;
*regSkelOuter2 t1 = Some (hs1,hs2a)*;
*regSkelOuter2 t2 = Some (hs1,hs2b)*

⟧ ⟹
w1 (tFold a3 c3 (tMap (tFold a2 c2) (vConc t1 t2))) =
w2 (c1 (tFold a1 c1 (tMap (tFold a2 c2) t1))
(tFold a1 c1 (tMap (tFold a2 c2) t2)))
**by** (*insert tFold-tFold-vConc-wrapped-0 [of c1 c2 c3 w1 a3 w2 a1 c4 t1], auto*)

**lemma** *tTranspose-tFold2-gen-0*:
⟦ assoc c1; assoc c2; assoc c3; assoc c4; assoc c5;
⋀ x y z . w1 (a1 x (a2 y z)) = w2 (a3 y (a4 x z));
⋀ x y . w1 (c1 x y) = c5 (w1 x) (w1 y);
⋀ x y . w2 (c3 x y) = c5 (w2 x) (w2 y);
⋀ x y z . w1 (a1 x (c2 y z)) = c5 (w1 (a1 x y)) (w1 (a1 x z));
⋀ h x y . w2 (a3 h (c4 x y)) = c5 (w2 (a3 h x)) (w2 (a3 h y));
⋀ x1 x2 y1 y2 . c5 (c5 x1 x2) (c5 y1 y2) = c5 (c5 x1 y1) (c5 x2 y2)
(*
⋀ x y z . a2 x (c1 y z) = c1 (a2 x y) (a2 x z);
*)
⟧ ⟹
ALL hs1 hs2 . regSkelOuter2 t = Some (hs1,hs2) ⟶
w2 (tFold a3 c3 (tMap (tFold a4 c4) (tTranspose t))) =
w1 (tFold a1 c1 (tMap (tFold a2 c2) t))
**apply** (*induct-tac t rule*: *T-induct*)
**apply** (*induct-tac t0 rule*: *T-induct*)
**apply** (*intro strip*, *simp*)
**apply** (*intro strip*, *simp add*: *regSkelOuter1-def regSkelOuter2-def*)
**apply** (*intro strip*)
**apply** (*drule regSkelOuter2-hConc-eq-Some*)
**apply** (*erule exE*, *erule exE*, *erule conjE*, *erule conjE*)
**apply** (*simp del*: *tFold-tMap*)
**apply** (*subst tFold-tFold-vConc-wrapped [of c3 c4 c3 w2 a3 w2 a3 c5]*)
**apply** (*assumption*, *assumption*, *assumption*)
**apply** *simp*
**apply** *simp*
**apply** *simp*
**apply** (*erule regSkelOuter2-tTranspose*)
**apply** (*erule regSkelOuter2-tTranspose*)
**apply** *simp*
**done**

**lemma** *tTranspose-tFold2-gen*:
⟦ assoc c1; assoc c2; assoc c3; assoc c4; assoc c5;
⋀ x y z . w1 (a1 x (a2 y z)) = w2 (a3 y (a4 x z));
⋀ x y . w1 (c1 x y) = c5 (w1 x) (w1 y);
⋀ x y . w2 (c3 x y) = c5 (w2 x) (w2 y);
⋀ x y z . w1 (a1 x (c2 y z)) = c5 (w1 (a1 x y)) (w1 (a1 x z));
⋀ h x y . w2 (a3 h (c4 x y)) = c5 (w2 (a3 h x)) (w2 (a3 h y));
⋀ x1 x2 y1 y2 . c5 (c5 x1 x2) (c5 y1 y2) = c5 (c5 x1 y1) (c5 x2 y2);
regSkelOuter2 t = Some (hs1,hs2)
⟧ ⟹

*w2* (*tFold a3 c3* (*tMap* (*tFold a4 c4*) (*tTranspose t*))) =
*w1* (*tFold a1 c1* (*tMap* (*tFold a2 c2*) *t*))
**by** (*insert tTranspose-tFold2-gen-0* [*of c1 c2 c3 c4 c5 w1 a1 a2 w2 a3 a4 t*], *auto*)

**end**

# 6 Inversion of Normal Tables

**theory** *Inversion* = *Tables2*:

## 6.1 One-Dimensional Inversion

**consts**
*inverse1* :: ('*a*,('*b*,'*c*) *T*) *T* ⇒ ('*b*,('*a*,'*c*) *T*) *T*

**defs**
*inverse1-def*: *inverse1* == *tFold addH2 hConc*

**lemma** *inverse1-addH*[*simp*]:
*inverse1* (*addH h t*) = *addH2 h t*
**by** (*unfold inverse1-def*, *simp*)

**lemma** *inverse1-hConc*[*simp*]:
*inverse1* (*hConc t1 t2*) = *hConc* (*inverse1 t1*) (*inverse1 t2*)
**by** (*unfold inverse1-def*, *simp*)

**lemma** *tFold-tFold-inverse1*:
⟦ *assoc c2*; *assoc c4*;
⋀ *h2 h1 t* . *a2 h2* (*a1 h1 t*) = *a3 h1* (*a4 h2 t*);
⋀ *h t u* . *a3 h* (*c4 t u*) = *c2* (*a3 h t*) (*a3 h u*)
⟧ ⟹
*tFold a2 c2* (*tMap* (*tFold a1 c1*) (*inverse1 t*)) =
*tFold a3 c2* (*tMap* (*tFold a4 c4*) *t*)
**apply** (*induct-tac t rule*: *T-induct*)
**apply** (*induct-tac t0 rule*: *T-induct*, *simp*)
**apply** (*induct-tac t1 rule*: *T-induct*, *simp*)
**apply** *simp*
**apply** (*subgoal-tac tFold* (*λha t0*. *a3 h* (*a4 ha t0*)) *c2 t1a* = *a3 h* (*tFold a4 c4 t1a*), *simp*)
**apply** (*induct-tac t1a rule*: *T-induct*, *simp*, *simp*)

original induction, second case:

**apply** (*simp del*: *tFold-tMap*)
**done**

**lemma** *tFold-tFold0-inverse1*:
⟦ *assoc c2*; *assoc c4*;
⋀ *h t u* . *a2* (*c4 t u*) *h* = *c2* (*a2 t h*) (*a2 u h*)

]] ⟹
*tFold a2 c2* (*tMap* (*tFold0 c1*) (*inverse1 t*)) =
*tFold* (*flip a2*) *c2* (*tMap* (*tFold0 c4*) *t*)
**by** (*unfold tFold0-def*, *rule tFold-tFold-inverse1*, *simp-all*)

## 6.2 spread1

**consts** *spread1* :: *'a neList* ⇒ *'b* ⇒ (*'a*, *'b*) *T*

**defs** *spread1-def*: *spread1 hs t* == *foldr1 hConc* (*map1* (λ*h*. *addH h t*) *hs*)

**lemma** *regSkelOuter2-spread1*[*simp*]:
*regSkelOuter2* (*spread1 hs t*) = *Some* (*hs*, *headers t*)
**apply** (*subst neList-append-induct* [*of* % *hs* . *regSkelOuter2* (*spread1 hs t*) = *Some* (*hs*, *headers t*)], *simp-all*)

WHy does standard induction not work here?!

**apply** (*unfold spread1-def*, *simp-all*)
**done**

**lemma** *spread1-singleton*[*simp*]: *spread1* (*singleton h*) *t* = *addH h t*
**by** (*simp add*: *spread1-def*)

**lemma** *spread1-append*[*simp*]:
*spread1* (*append hs1 hs2*) *t* = *hConc* (*spread1 hs1 t*) (*spread1 hs2 t*)
**by** (*simp add*: *spread1-def*)

**lemma** *spread1-tFold-append*:
*spread1* (*tFold f append t1*) *t* = *tFold* (% *h0 t0* . *spread1* (*f h0 t0*) *t*) *hConc t1*
**by** (*induct-tac t1 rule*: *T-induct*, *simp-all*)

**lemma** *spread1-hConc-0*:
*ALL t1* . *spread1 hs* (*hConc t1 t2*) = *vConc* (*spread1 hs t1*) (*spread1 hs t2*)
**apply** (*induct-tac hs rule*: *neList-append-induct*, *simp*)
**apply** *simp*
**apply** (*rule allI*)
**apply** (*subst vConc-hConc*)
  **apply** *simp*
  **apply** (*rule conjI*, *simp*, *simp*)
  **apply** *simp*
  **apply** *simp*
  **apply** *simp*
**apply** *simp*
**done**

**lemma** *spread1-hConc*:
*spread1 hs* (*hConc t1 t2*) = *vConc* (*spread1 hs t1*) (*spread1 hs t2*)
**by** (*insert spread1-hConc-0* [*of hs t2*], *auto*)

**lemma** *tMap-f-spread1*:

*tMap f (spread1 hs t) = spread1 hs (f t)*
**by** (*induct-tac hs rule*: *neList-append-induct*, *simp-all*)

**lemma** *tFold-spread1*:
*assoc c* ⟹ *tFold a c (spread1 hs t) = foldr1 c (map1 (% h . a h t) hs)*
**by** (*induct-tac hs rule*: *neList-append-induct*, *simp-all*)

## 6.3 spread2

This was used in the previous definition of *dconc*.

**consts**
*spread2* :: *'a neList* ⇒ *('b,'c) T* ⇒ *('b,('a,'c) T) T*

**defs**
*spread2-def*: *spread2 hs t == foldr1 vConc (map1 (% h . addH2 h t) hs)*

**lemma** *spread2-singleton*[*simp*]:
*spread2 (singleton h) t = addH2 h t*
**by** (*unfold spread2-def*, *simp*)

**lemma** *spread2-append*[*simp*]:
*spread2 (append hs1 hs2) t = vConc (spread2 hs1 t) (spread2 hs2 t)*
**by** (*unfold spread2-def*, *simp*)

**lemma** *spread2-cons1*[*simp*]:
*spread2 (cons1 h hs) t = vCons (h,t) (spread2 hs t)*
**by** (*simp add*: *cons1 vCons*)

**lemma** *headers-spread2*[*simp*]:
*headers (spread2 hs t) = headers t*
**by** (*induct-tac hs rule*: *neList-cons-induct*, *simp-all*)

**lemma** *regSkelOuter2-spread2-0*:
*ALL hs2 . regSkelOuter2 (spread2 hs2 t) = Some (headers t,hs2)*
**apply** (*induct-tac t rule*: *T-cons-induct*, *simp*)
**apply** (*rule allI*, *induct-tac hs2 rule*: *neList-append-induct*)
**apply** *simp*
**apply** *simp*
**apply** (*rule allI*)
**apply** (*erule mp1*)
**apply** (*rule-tac x=t2* **in** *spec*)
**apply** (*induct-tac hs2 rule*: *neList-cons-induct*)
**apply** (*intro strip*, *simp*)

*hs2 = cons1 x xs*

**apply** (*intro strip*, *simp*)
**apply** (*drule-tac x=xa* **in** *spec*)
**apply** *simp*
**apply** (*subst regSkelOuter2-vCons*, *simp*)

**apply** (*simp add*: *cons1*)
**done**

**lemma** *regSkelOuter2-spread2*[*simp*]:
*regSkelOuter2* (*spread2 hs2 t*) = *Some* (*headers t*,*hs2*)
**by** (*rule regSkelOuter2-spread2-0* [*THEN spec*])

**lemma** *spread2-hConc-0*:
*ALL t1 t2 . spread2 hs2* (*hConc t1 t2*) = *hConc* (*spread2 hs2 t1*) (*spread2 hs2 t2*)
**apply** (*induct-tac hs2 rule*: *neList-append-induct*)
**apply** *simp*
**apply** *simp*
**apply** (*intro strip*)
**apply** (*subst vConc-hConc*)
**apply** *simp*
**apply** (*rule conjI*)
**apply** *simp*
**apply** *simp*
**apply** *simp*
**apply** *simp*
**apply** *simp*
**apply** *simp*
**done**

**lemma** *spread2-hConc*[*simp*]:
*spread2 hs2* (*hConc t1 t2*) = *hConc* (*spread2 hs2 t1*) (*spread2 hs2 t2*)
**by** (*insert spread2-hConc-0*, *auto*)

**lemma** *spread2-addH*:
*spread2 hs* (*addH h t*) = *addH h* (*foldr1 hConc* (*map1* (% *h2 . addH h2 t*) *hs*))
**by** (*induct-tac hs rule*: *neList-append-induct*, *simp-all*)

**lemma** *spread2-tFold-hConc*:
*spread2 hs2* (*tFold a hConc t*) = *tFold* (% *h t . spread2 hs2* (*a h t*)) *hConc t*
**by** (*induct-tac t rule*: *T-induct*, *simp-all*)

**lemma** *tMap-tFold-spread2*:
*assoc c* $\Longrightarrow$
*tMap* (*tFold a c*) (*spread2 hs t*) =
*tMap* (% *t0 . foldr1 c* (*map1* ($\lambda$*h2. a h2 t0*) *hs*)) *t*
**by** (*induct-tac t rule*: *T-induct*, *simp-all add*: *spread2-addH map1-contract-comp tFold-foldr1-hConc o-def*)

## 6.4 delH1

**consts** *delH1* :: ('*a*,'*b*) *T* $\Rightarrow$ '*b*
**defs** *delH1-def*: *delH1* == *tFold* (% *h t . t*) *const*

**lemma** *delH1-hConc*[*simp*]:
*delH1* (*hConc t1 t2*) = *delH1 t1*
**by** (*simp add*: *delH1-def*)

**lemma** *delH1-addH*[*simp*]:
*delH1* (*addH h t*) = *t*
**by** (*simp add*: *delH1-def*)

**lemma** *delH1-hCons*[*simp*]:
*delH1* (*hCons* (*h*,*t0*) *t*) = *t0*
**by** (*simp add*: *hCons*)

**lemma** *delH1-tFold-hConc*[*simp*]:
*delH1* (*tFold f hConc t*) = *delH1* (*uncurry f* (*hHead t*))
**by** (*induct-tac t rule*: *T-induct*, *simp-all*)

**lemma** *headers-delH1--reg2-0*:
*ALL hs1 hs2* . *regSkelOuter2 t1* = *Some* (*hs1*, *hs2*) ⟶ *headers* (*delH1 t1*) = *hs2*
**apply** (*induct-tac t1 rule*: *T-induct*, *simp-all*)
**apply** (*intro strip*)
**apply** (*drule optThen-result-Some*, *erule exE*, *erule conjE*)
**apply** (*drule optThen-result-Some*, *erule exE*, *erule conjE*)
**apply** (*split split-if-asm*, *simp-all*)
**apply** (*case-tac y*, *case-tac ya*, *simp*)
**done**

**lemma** *headers-delH1--reg2*[*simp*]:
*regSkelOuter2 t1* = *Some* (*hs1*, *hs2*) ⟹ *headers* (*delH1 t1*) = *hs2*
**by** (*insert headers-delH1--reg2-0*, *auto*)

**lemma** *delH1-addH2*[*simp*]: *delH1* (*addH2 h t*) = *addH h* (*delH1 t*)
**by** (*induct-tac t rule*: *T-induct*, *simp-all*)

**lemma** *delH1-vConc-0*:
*ALL hs1a hs1b hs2a hs2b* .
*regSkelOuter2 t1* = *Some* (*hs1a*, *hs1b*) ⟶
*regSkelOuter2 t2* = *Some* (*hs2a*, *hs2b*) ⟶
*delH1* (*vConc t1 t2*) = *hConc* (*delH1 t1*) (*delH1 t2*)
**apply** (*induct-tac t1 rule*: *T-cons-induct*, *simp-all*)
**apply** (*induct-tac t2 rule*: *T-cons-induct*, *simp-all*)
**apply** (*intro strip*, *erule exE*, *erule exE*)
**apply** (*drule optThen-result-Some*, *erule exE*, *erule conjE*)
**apply** (*split split-if-asm*, *simp-all*)
**apply** (*case-tac p*, *case-tac y*, *simp*)
**apply** (*induct-tac t2 rule*: *T-cons-induct*, *simp-all*)
**apply** (*intro strip*, *erule exE*, *erule exE*)
**apply** (*drule optThen-result-Some*, *erule exE*, *erule conjE*)
**apply** (*split split-if-asm*, *simp-all*)

**apply** (*case-tac p*, *case-tac y*, *simp*)
**apply** (*intro strip*, *erule exE*, *erule exE*, *erule exE*, *erule exE*)
**apply** *simp*
**apply** (*case-tac p*, *case-tac pa*, *simp*)
**done**

**lemma** *delH1-vConc*:
⟦ *regSkelOuter2 t1* = *Some* (*hs1a*, *hs1b*);
*regSkelOuter2 t2* = *Some* (*hs2a*, *hs2b*)
⟧ ⟹
*delH1* (*vConc t1 t2*) = *hConc* (*delH1 t1*) (*delH1 t2*)
**by** (*insert delH1-vConc-0*, *auto*)

**lemma** *delH1-tMap-const*[*simp*]: *delH1* (*tMap* (*const t2*) *t1*) = *t2*
**by** (*induct-tac t1 rule*: *T-induct*, *simp-all*)

**lemma** *delH1-tMap-CONST*[*simp*]: *delH1* (*tMap* (% *x* . *t2*) *t1*) = *t2*
**by** (*induct-tac t1 rule*: *T-induct*, *simp-all*)

## 6.5 delH2

**consts** *delH2* :: ('*a*,('*b*,'*c*) *T*) *T* ⇒ ('*a*,'*c*) *T*

**defs** *delH2-def*: *delH2* == *tMap delH1*

**lemma** *delH2-hConc*[*simp*]:
*delH2* (*hConc t1 t2*) = *hConc* (*delH2 t1*) (*delH2 t2*)
**by** (*simp add*: *delH2-def*)

**lemma** *delH2-addH*[*simp*]:
*delH2* (*addH h t*) = *addH h* (*delH1 t*)
**by** (*simp add*: *delH2-def*)

**lemma** *delH2-hCons*[*simp*]:
*delH2* (*hCons* (*h*,*t0*) *t*) = *hCons* (*h*, *delH1 t0*) (*delH2 t*)
**by** (*simp add*: *hCons*)

**lemma** *delH2-addH2*[*simp*]:
*delH2* (*addH2 h t*) = *t*
**by** (*induct-tac t rule*: *T-induct*, *simp-all*)

**lemma** *headers-delH2*[*simp*]: *headers* (*delH2 t*) = *headers t*
**by** (*induct-tac t rule*: *T-induct*, *simp-all*)

**lemma** *delH2-vConc-0*:
*ALL hs1 hs2a t2 hs2b* .
*regSkelOuter2 t1* = *Some* (*hs1*, *hs2a*) ⟶
*regSkelOuter2 t2* = *Some* (*hs1*, *hs2b*) ⟶
*delH2* (*vConc t1 t2*) = *delH2 t1*

**apply** (*induct-tac t1 rule*: *T-cons-induct*, *simp-all*)
**apply** (*rule allI*)
**apply** (*rule impI*)
**apply** (*rule allI*)
**apply** (*erule exE*)+
**apply** (*drule optThen-result-Some*, *erule exE*, *erule conjE*, *simp*)
**apply** (*split split-if-asm*, *simp-all*)
**apply** (*erule conjE*, *case-tac y*, *simp*)
**apply** (*induct-tac t2a rule*: *T-cons-induct*, *simp-all*)
**apply** (*rule impI*, *rotate-tac −1*, *drule sym*, *simp*)
**apply** (*intro strip*)
**apply** (*erule exE*)+
**apply** (*drule optThen-result-Some*, *erule exE*, *erule conjE*, *simp*)
**apply** (*split split-if-asm*, *simp-all*)
**apply** (*erule conjE*, *case-tac p*, *case-tac pa*, *case-tac ya*, *simp*)
**apply** (*drule-tac x=t2b* **in** *spec*, *simp*)
**apply** (*rotate-tac 2*, *drule sym*, *simp*, *drule cons1-inj*, *simp*)
**done**

**lemma** *delH2-vConc*[*simp*]:
⟦ *regSkelOuter2 t1* = *Some* (*hs1*, *hs2a*);
*regSkelOuter2 t2* = *Some* (*hs1*, *hs2b*) ⟧ ⟹
*delH2* (*vConc t1 t2*) = *delH2 t1*
**by** (*insert delH2-vConc-0* [*of t1*], *auto*)

**lemma** *delH2-spread2*[*simp*]:
*delH2* (*spread2 hs t*) = *t*
**apply** (*induct-tac t rule*: *T-induct*, *simp-all*)
**apply** (*induct-tac hs rule*: *neList-append-induct*, *simp-all*)
**apply** (*subst delH2-vConc*, *simp-all*)
**apply** (*rule conjI*, *simp-all*)
**apply** (*rule conjI*, *simp-all*)
**done**

**lemma** *delH2-tMap-const*:
*delH2* (*tMap* (*const t2*) *t1*) = *tMap* (*const* (*delH1 t2*)) *t1*
**by** (*induct-tac t1 rule*: *T-induct*, *simp-all*)

**lemma** *delH2-tMap-CONST*:
*delH2* (*tMap* (% *x* . *t2*) *t1*) = *tMap* (*const* (*delH1 t2*)) *t1*
**by** (*induct-tac t1 rule*: *T-induct*, *simp-all*)

## 6.6 Third Dimension Operators

**consts** *addH3* :: *'c* ⇒ (*'a*, (*'b*,*'d*) *T*) *T* ⇒ (*'a*, (*'b*, (*'c*, *'d*) *T*) *T*) *T*

**defs** *addH3-def*: *addH3* == *tMap* ∘ *addH2*

**lemma** *delH1-addH3*[*simp*]: *delH1* (*addH3 h t*) = *addH2 h* (*delH1 t*)

**by** (*unfold addH3-def*, *induct-tac t rule*: *T-induct*, *simp-all*)

**consts** *delH3* :: ('*a*, ('*b*, ('*c* , '*d*) *T*) *T*) *T* ⇒ ('*a*, ('*b*,'*d*) *T*) *T*

**defs** *delH3-def*: *delH3* == *tMap delH2*

## 6.7 Slimming Operators

These operators slim down a selected dimension in the regular table skeleton to a singleton containing a supplied entry *u*.

**consts** *slimH1* :: '*a* ⇒ ('*c*,'*b*) *T* ⇒ ('*a*,'*b*) *T*

**defs** *slimH1-def*[*simp*]: *slimH1 h1 t* == *addH h1* (*delH1 t*)

**lemma** *tFold-slimH1*:
  *assoc c* ⟹ *tFold a c* (*slimH1 u t*) = *tFold* (% *h0 t0* . *a u t0*) *const t*
**apply** (*induct-tac t rule*: *T-induct*, *simp-all del*: *const*)
**apply** (*subst const*)
**apply** (*simp* (*no-asm-simp*) *del*: *const*)
**done**

**lemma** *headers-slimH1*[*simp*]: *headers* (*slimH1 u t*) = *singleton u*
**by** *simp*

**consts** *slimH2* :: '*b* ⇒ ('*a*,('*d*,'*c*) *T*) *T* ⇒ ('*a*,('*b*,'*c*) *T*) *T*

**defs** *slimH2-def*[*simp*]: *slimH2 h2 t* == *addH2 h2* (*delH2 t*)

**lemma** *slimH2-tMap*: *slimH2 h2* = *tMap* (*slimH1 h2*)
**by** (*rule ext*, *induct-tac x rule*: *T-induct*, *simp-all*)

**lemma** *slimH2-slimH2*[*simp*]: *slimH2 u* (*slimH2 v t*) = *slimH2 u t*
**by** *simp*

**lemma** *slimH2-addH*[*simp*]: *slimH2 u* (*addH h t*) = *addH h* (*slimH1 u t*)
**by** *simp*

**lemma** *slimH2-hConc*[*simp*]:
  *slimH2 u* (*hConc t1 t2*) = *hConc* (*slimH2 u t1*) (*slimH2 u t2*)
**by** *simp*

**lemma** *slimH2-as-tFold*[*simp*]:
  *tFold* (λ*h t*. *addH h* (*slimH1 u t*)) *hConc t* = *slimH2 u t*
**by** (*induct-tac t rule*: *T-induct*, *simp-all*)

**lemma** *slimH2-addH2*[*simp*]: *slimH2 u* (*addH2 h2 t*) = *addH2 u t*
**by** *simp*

**lemma** *headers-slimH2*[*simp*]: *headers* (*slimH2 u t*) = *headers t*

**by** (*induct-tac t rule*: *T-induct*, *simp-all*)

**lemma** *regSkelOuter2-slimH2*[*simp*]:
*regSkelOuter2* (*slimH2 u t*) = *Some* (*headers t*, *singleton u*)
**by** (*induct-tac t rule*: *T-induct*, *simp-all*)

**lemma** *slimH2-tFold-hConc*[*simp*]:
*slimH2 u* (*tFold f hConc t*) = *tFold* (% *h t* . *slimH2 u* (*f h t*)) *hConc t*
**by** (*induct-tac t rule*: *T-induct*, *simp-all del*: *slimH2-def*)

**lemma** *slimH2-tFold-addH2-hConc*[*simp*]:
*slimH2 u* (*tFold* (% *h* . *addH2 v*) *hConc t*) = *tFold* (% *h* . *addH2 u*) *hConc t*
**by** (*induct-tac t rule*: *T-induct*, *simp-all del*: *slimH2-def*)

**lemma** *del1-slimH2-isConst*[*simp*]:
*delH1* (*slimH2 u* (*t1* :: ('*a*, ('*b*, *unit*) *T*) *T*)) = *delH1* (*slimH2 u t2*)
**by** *simp*

**lemma** *tFold-slimH2*:
*assoc c* ⟹ *tFold a c* (*slimH2 u t*) = *tFold* (% *h0 t0* . *a h0* (*slimH1 u t0*)) *c t*
**by** (*induct-tac t rule*: *T-induct*, *simp-all*)

**consts** *slimH3* :: '*c* ⇒ ('*a*, ('*b*, ('*e* , '*d*) *T*) *T*) *T* ⇒ ('*a*, ('*b*, ('*c*, '*d*) *T*) *T*) *T*

**defs** *slimH3-def*: *slimH3* == *tMap* ∘ *slimH2*

**lemma** *slimH3-expand*: *slimH3 u t* = *addH3 u* (*delH3 t*)
**by** (*induct-tac t rule*: *T-induct*, *simp-all add*: *delH3-def addH3-def slimH3-def*)

**lemma** *slimH3-addH*[*simp*]: *slimH3 u* (*addH h t*) = *addH h* (*slimH2 u t*)
**by** (*simp add*: *slimH3-def del*: *slimH2-def*)

**lemma** *slimH3-hConc*[*simp*]: *slimH3 u* (*hConc t1 t2*) = *hConc* (*slimH3 u t1*) (*slimH3 u t2*)
**by** (*simp add*: *slimH3-def del*: *slimH2-def*)

**lemma** *spread1-slimH2*: *spread1 hs* (*slimH2 u t*) = *slimH3 u* (*spread1 hs t*)
**by** (*induct-tac hs rule*: *neList-append-induct*, *simp-all del*: *slimH2-def slimH3-def*)

**lemma** *tFold-slimH3*:
*assoc c* ⟹ *tFold a c* (*slimH3 u t*) = *tFold* (% *h0 t0* . *a h0* (*slimH2 u t0*)) *c t*
**by** (*induct-tac t rule*: *T-induct*, *simp-all del*: *slimH2-def slimH3-def*)

**lemma** *tMap-f-slimH3*:
*tMap f* (*slimH3 u t*) = *tMap* (% *t0* . *f* (*slimH2 u t0*)) *t*
**apply** (*unfold slimH3-def*)
**apply** (*unfold comp-def*)
**apply** (*subst tMap-tMap*)
**apply** (*unfold comp-def*)

**apply** (*rule refl*)
**done**

## 6.8 Right-Updating Horizontal Concatenation

**consts** *hConcU* :: ('*c* ⇒ ('*a*,'*b*) *T*) ⇒ ('*a*,'*b*) *T* ⇒ '*c* ⇒ ('*a*,'*b*) *T*

**defs** *hConcU-def*[*simp*]: *hConcU u t1 t2* == *hConc t1* (*u t2*)

**lemma** *hConcU-assoc*[*simp*]:
⟦ ⋀ *t* . *u* (*u t*) = *u t*;
⋀ *t1 t2* . *u* (*hConc t1 t2*) = *hConc* (*u t1*) (*u t2*)
⟧ ⟹ *assoc* (*hConcU u*)
**by** (*rule assoc-intro*, *simp*)

**lemma** *U-hConcU*[*simp*]:
⟦ ⋀ *t* . *u* (*u t*) = *u t*;
⋀ *t1 t2* . *u* (*hConc t1 t2*) = *hConc* (*u t1*) (*u t2*)
⟧ ⟹ *u* (*hConcU u t1 t2*) = *hConc* (*u t1*) (*u t2*)
**by** *simp*

**lemma** *U-tFold-hConcU*[*simp*]:
⟦ ⋀ *t* . *u* (*u t*) = *u t*;
⋀ *t1 t2* . *u* (*hConc t1 t2*) = *hConc* (*u t1*) (*u t2*)
⟧ ⟹ *u* (*tFold f* (*hConcU u*) *t*) = *tFold* (% *h0 t0* . *u* (*f h0 t0*)) *hConc t*
**by** (*induct-tac t rule*: *T-induct*, *simp-all del*: *hConcU-def*)

**consts** *hConcSH2* :: '*b* ⇒ ('*a*,('*b*,'*c*) *T*) *T* ⇒ ('*a*,('*d*,'*c*) *T*) *T* ⇒ ('*a*,('*b*,'*c*) *T*) *T*

**defs** *hConcSH2-def*: *hConcSH2 u* == *hConcU* (*slimH2 u*)

**lemma** *hConcSH2-assoc*[*simp*]: *assoc* (*hConcSH2 u*)
**by** (*rule assoc-intro*, *simp add*: *hConcSH2-def*)

**lemma** *slimH2-hConcSH2*[*simp*]:
*slimH2 u* (*hConcSH2 u t1 t2*) = *hConc* (*slimH2 u t1*) (*slimH2 u t2*)
**by** (*unfold hConcSH2-def*, *simp del*: *slimH2-def*)

**lemma** *slimH2-tFold-hConcSH2*[*simp*]:
*slimH2 u* (*tFold f* (*hConcSH2 u*) *t*) = *tFold* (% *h t* . *slimH2 u* (*f h t*)) *hConc t*
**by** (*induct-tac t rule*: *T-induct*, *simp-all del*: *slimH2-def*)

## 6.9 Diagonal Table Concatenation

Diagonal concatenation allocates the two last arguments as blocks on the main diagonal and fills the remainder with "empty" tables created using the first argument.

Since we shall use this as combinator of a table fold, we need its associativity, but that holds only on tables that are regular in their outer two dimensions.

For this reason we define diagonal concatenation on the subtype *regT2*.

**consts** *dConc* :: (('b,'c) *T* ⇒ ('b,'c) *T*) ⇒
('a,'b,'c) *regT2* ⇒
('a,'b,'c) *regT2* ⇒
('a,'b,'c) *regT2*

**defs** *dConc-def*: *dConc h t1 t2* ==
*let t1' = Rep-regT2 t1*;
*t2' = Rep-regT2 t2*
*in Abs-regT2*
(*hConc* (*vConc t1'* (*tMap* (*const* (*h* (*delH1 t2'*))) *t1'*))
(*vConc* (*tMap* (*const* (*h* (*delH1 t1'*))) *t2'*) *t2'* )
)

**lemma** *dConc*[*simp*]:
⟦ *regSkelOuter2 t1* = *Some* (*cs1*, *hs1*);
*regSkelOuter2 t2* = *Some* (*cs2*, *hs2*)
⟧ ⟹
*dConc h* (*Abs-regT2 t1*) (*Abs-regT2 t2*)
= *Abs-regT2*
(*hConc* (*vConc t1* (*tMap* (*const* (*h* (*delH1 t2*))) *t1*))
(*vConc* (*tMap* (*const* (*h* (*delH1 t1*))) *t2*) *t2*)
)
**apply** (*unfold dConc-def*, *simp only*: *Let-def regT2-def*)
**apply** (*subst Abs-regT2-inverse*, *simp add*: *regT2-def*, *fast*)+
**apply** *simp*
**done**

**lemma** *dConc1*:
⟦ *regSkelOuter2* (*Rep-regT2 t1*) = *Some* (*cs1*, *hs1*);
*regSkelOuter2* (*Rep-regT2 t2*) = *Some* (*cs2*, *hs2*)
⟧ ⟹
*dConc h t1 t2*
= *Abs-regT2*
(*hConc* (*vConc* (*Rep-regT2 t1*) (*tMap* (*const* (*h* (*delH1* (*Rep-regT2 t2*))))
(*Rep-regT2 t1*)))
(*vConc* (*tMap* (*const* (*h* (*delH1* (*Rep-regT2 t1*)))) (*Rep-regT2 t2*))
(*Rep-regT2 t2*))
)
**apply** (*frule-tac h*=*h* **and** *t2.0*=*Rep-regT2 t2* **in** *dConc*, *assumption*)
**apply** (*simp only*: *Rep-regT2-inverse*)
**done**

**lemma** *regSkelOuter2-dConcDef*[*simp*]:
⟦ ⋀ *t* . *headers* (*h t*) = *headers t*;
*regSkelOuter2 t1* = *Some* (*cs1*, *hs1*);
*regSkelOuter2 t2* = *Some* (*cs2*, *hs2*)
⟧ ⟹

```
  regSkelOuter2
       (hConc (vConc t1 (tMap (const (h (delH1 t2))) t1))
              (vConc (tMap (const (h (delH1 t1))) t2)  t2))
    = Some (append cs1 cs2, append hs1 hs2)
apply (rotate-tac −1, frule regSkelOuter2-eq-Some)
apply (rotate-tac −2, frule regSkelOuter2-eq-Some)
apply (cut-tac t1.0=t1 and t2.0=tMap (const (h (delH1 t2))) t1 in regSkelOuter2-vConc)
    apply assumption
    apply (simp (no-asm-simp))
    apply (rule conjI)
    apply (simp (no-asm-simp))
    apply (simp (no-asm-simp))
    apply (simp (no-asm-simp))
apply (rotate-tac −1, frule regSkelOuter2-eq-Some)
apply (cut-tac t1.0=tMap (const (h (delH1 t1))) t2 and t2.0=t2 in regSkelOuter2-vConc)
  apply (simp (no-asm-simp))
  apply (rule conjI)
  apply (simp (no-asm-simp))
  apply (simp (no-asm-simp))
  apply assumption
  apply (simp (no-asm-simp))
apply (rotate-tac −1, frule regSkelOuter2-eq-Some)
apply (simp (no-asm-simp))
apply (subst headers-delH1--reg2, assumption)
apply (subst headers-delH1--reg2, assumption)
apply (simp (no-asm-simp))
done

lemma regSkelOuter2-dConc[simp]:
  ⟦ ⋀ t . headers (h t) = headers t;
    regSkelOuter2 t1 = Some (cs1, hs1);
     regSkelOuter2 t2 = Some (cs2, hs2)
  ⟧ ⟹
   regSkelOuter2 (Rep-regT2 (dConc h (Abs-regT2 t1) (Abs-regT2 t2)))
    = Some (append cs1 cs2, append hs1 hs2)
apply (subst dConc)
    apply assumption
    apply assumption
apply (subst Abs-regT2-inverse, rule regT2)
apply (subst regSkelOuter2-dConcDef)
  apply simp
  apply assumption
  apply assumption
  apply simp
  apply (rule conjI, simp, simp)
apply (subst regSkelOuter2-dConcDef)
  apply simp
  apply assumption
  apply assumption
```

**apply** *simp*
**done**

**lemma** *dConc2*[*simp*]:
⟦ ⋀ *t* . *headers* (*h t*) = *headers t*;
*regSkelOuter2* (*Rep-regT2 t1*) = *Some* (*cs1*, *hs1*);
*regSkelOuter2* (*Rep-regT2 t2*) = *Some* (*cs2*, *hs2*)
⟧ ⟹
*Rep-regT2* (*dConc h t1 t2*)
= *hConc* (*vConc* (*Rep-regT2 t1*) (*tMap* (*const* (*h* (*delH1* (*Rep-regT2 t2*)))) (*Rep-regT2 t1*)))
(*vConc* (*tMap* (*const* (*h* (*delH1* (*Rep-regT2 t1*)))) (*Rep-regT2 t2*)) (*Rep-regT2 t2*))
**apply** (*subst dConc1*)
**apply** *assumption+*
**apply** (*subst Abs-regT2-inverse*, *rule regT2*)
**apply** (*subst regSkelOuter2-dConcDef*)
**apply** *simp*
**apply** *assumption*
**apply** *assumption*
**apply** *simp*
**apply** (*rule conjI*, *simp*, *simp*)
**apply** (*simp del*: *const*)
**done**

**lemma** *regSkelOuter2-dConc1*[*simp*]:
⟦ ⋀ *t* . *headers* (*h t*) = *headers t*;
*regSkelOuter2* (*Rep-regT2 t1*) = *Some* (*cs1*, *hs1*);
*regSkelOuter2* (*Rep-regT2 t2*) = *Some* (*cs2*, *hs2*)
⟧ ⟹
*regSkelOuter2* (*Rep-regT2* (*dConc h t1 t2*))
= *Some* (*append cs1 cs2*, *append hs1 hs2*)
**apply** (*subst dConc2*)
**apply** *simp*
**apply** *assumption*
**apply** *assumption*
**apply** (*subst regSkelOuter2-dConcDef*)
**apply** *simp*
**apply** *assumption*
**apply** *assumption*
**apply** *simp*
**done**

**lemma** *headers-dConc*[*simp*]:
⟦ ⋀ *t* . *headers* (*h t*) = *headers t*
⟧ ⟹
*headers* (*Rep-regT2* (*dConc h t1 t2*)) = *append* (*headers* (*Rep-regT2 t1*)) (*headers* (*Rep-regT2 t2*))
**apply** (*cut-tac x=t1* **in** *Rep-regT2*)

**apply** (*cut-tac x=t2* **in** *Rep-regT2*)
**apply** (*simp add*: *regT2-def*)
**apply** (*drule regularOuter2*, *erule exE*, *erule exE*)
**apply** (*drule regularOuter2*, *erule exE*, *erule exE*)
**apply** (*subst dConc2*)
    **apply** *simp*
    **apply** *assumption*
    **apply** *assumption*
**apply** *simp*
**apply** (*subst headers-vConc*)
  **apply** *simp*
  **apply** (*rule conjI*)
  **apply** *simp*
  **apply** *simp*
  **apply** *simp*
  **apply** (*rule conjI*)
  **apply** (*rule sym*, *simp*)
  **apply** *simp*
**apply** (*subst headers-vConc*)
  **apply** *simp*
  **apply** (*rule conjI*)
  **apply** *simp*
  **apply** *simp*
  **apply** (*simp del*: *const*)
**apply** (*simp del*: *const*)
**done**

**lemma** *reg2dim2-dConc*[*simp*]:
  ⟦ ⋀ *t* . *headers* (*h t*) = *headers t*;
    *regSkelOuter2* (*Rep-regT2 t1*) = *Some* (*cs1*, *hs1*);
    *regSkelOuter2* (*Rep-regT2 t2*) = *Some* (*cs2*, *hs2*)
  ⟧ ⟹
   *reg2dim2* (*dConc h t1 t2*) = *append hs1 hs2*
**apply** (*cut-tac h=h* **and** *t1.0=Rep-regT2 t1* **and** *t2.0=Rep-regT2 t2* **in** *regSkelOuter2-dConc*)
  **apply** (*simp* (*no-asm-simp*))
  **apply** *assumption+*
**apply** (*simp only*: *Rep-regT2-inverse*)
**apply** (*subst Rep-regT2-inverse* [*THEN sym*])
**apply** (*rule reg2dim2*)
**apply** *assumption*
**done**

**lemma** *cs-dConc*[*simp*]:
  ⟦ ⋀ *t* . *headers* (*h t*) = *headers t*;
    ⋀ *t* . *h* (*h t*) = *h t*;
    ⋀ *t1 t2* . *h* (*hConc t1 t2*) = *hConc* (*h t1*) (*h t2*);
    *regSkelOuter2* (*Rep-regT2 t1*) = *Some* (*cs1*, *hs1*);
    *regSkelOuter2* (*Rep-regT2 t2*) = *Some* (*cs2*, *hs2*)
  ⟧ ⟹

```
  h (delH1 (Rep-regT2 (dConc h t1 t2))) =
  hConc (h (delH1 (Rep-regT2 t1)))
        (h (delH1 (Rep-regT2 t2)))
apply (cut-tac h=h and t1.0=Rep-regT2 t1 and t2.0=Rep-regT2 t2 in regSkelOuter2-dConc)
  apply (simp (no-asm-simp))
  apply assumption+
apply (subst dConc1)
  apply assumption+
apply (subst Abs-regT2-inverse, rule regT2)
  apply (subst regSkelOuter2-dConcDef)
    apply simp
    apply assumption
    apply assumption
    apply (simp (no-asm-simp))
    apply (rule conjI)
    apply (simp (no-asm-simp))
    apply (simp (no-asm-simp))
apply (simp (no-asm-simp))
apply (subst delH1-vConc)
  apply (simp (no-asm-simp))
  apply (rule conjI)
  apply (simp (no-asm-simp))
  apply (simp (no-asm-simp))
  apply (simp (no-asm-simp))
  apply (rule conjI)
  apply (rule sym, simp (no-asm-simp))
  apply (simp (no-asm-simp))
apply (simp del: const)
done

lemma dConc-assoc[simp]:
 ⟦ ⋀ t . headers (h t) = headers t;
   ⋀ t . h (h t) = h t;
   ⋀ t1 t2 . h (hConc t1 t2) = hConc (h t1) (h t2)
 ⟧ ⟹ assoc (dConc h)
apply (rule assoc-intro)
apply (cut-tac x=x in Rep-regT2)
apply (cut-tac x=y in Rep-regT2)
apply (cut-tac x=z in Rep-regT2)
apply (simp add: regT2-def)
apply (drule regularOuter2, erule exE, erule exE)
apply (drule regularOuter2, erule exE, erule exE)
apply (drule regularOuter2, erule exE, erule exE)
apply (cut-tac h=h and t1.0=Rep-regT2 x and t2.0=Rep-regT2 y in regSkelOuter2-dConc)
  apply (simp (no-asm-simp))
  apply assumption+
apply (cut-tac h=h and t1.0=Rep-regT2 y and t2.0=Rep-regT2 z in regSkelOuter2-dConc)
  apply (simp (no-asm-simp))
  apply assumption+
```

**apply** (*simp only*: *Rep-regT2-inverse*)
**apply** (*cut-tac h=h* **and** *t1.0=Rep-regT2* (*dConc h x y*) **and** *t2.0=Rep-regT2 z* **in** *regSkelOuter2-dConc*)
  **apply** (*simp* (*no-asm-simp*))
  **apply** *assumption+*
**apply** (*cut-tac h=h* **and** *t1.0=Rep-regT2 x* **and** *t2.0=Rep-regT2* (*dConc h y z*) **in** *regSkelOuter2-dConc*)
  **apply** (*simp* (*no-asm-simp*))
  **apply** *assumption+*
**apply** (*simp only*: *Rep-regT2-inverse*)
**apply** (*rotate-tac* −*1*, *frule regSkelOuter2-eq-Some*)
**apply** (*rotate-tac* −*1*, *frule regSkelOuter2-eq-Some*)
**apply** (*rotate-tac* −*1*, *frule regSkelOuter2-eq-Some*)
**apply** (*rotate-tac* −*1*, *frule regSkelOuter2-eq-Some*)
**apply** (*rotate-tac* −*1*, *frule regSkelOuter2-eq-Some*)
**apply** (*rotate-tac* −*1*, *frule regSkelOuter2-eq-Some*)
**apply** (*rotate-tac* −*1*, *frule regSkelOuter2-eq-Some*)
**apply** (*cut-tac h=h* **and** *t1.0=dConc h x y* **and** *t2.0=z* **in** *dConc1*)
  **apply** *assumption*
  **apply** *assumption*
**apply** (*cut-tac h=h* **and** *t1.0=x* **and** *t2.0=dConc h y z* **in** *dConc1*)
  **apply** *assumption*
  **apply** *assumption*
**apply** (*simp* (*no-asm-simp*))
**apply** (*subst cs-dConc* [*of h*])
  **apply** (*simp* (*no-asm-simp*))
  **apply** (*simp* (*no-asm-simp*))
  **apply** (*simp* (*no-asm-simp*))
  **apply** *assumption*
  **apply** *assumption*
**apply** (*subst cs-dConc* [*of h*])
  **apply** (*simp* (*no-asm-simp*))
  **apply** (*simp* (*no-asm-simp*))
  **apply** (*simp* (*no-asm-simp*))
  **apply** *assumption*
  **apply** *assumption*
**apply** (*subst dConc2* [*of h*])
  **apply** (*simp* (*no-asm-simp*))
  **apply** *assumption*
  **apply** *assumption*
**apply** (*subst dConc2* [*of h*])
  **apply** (*simp* (*no-asm-simp*))
  **apply** *assumption*
  **apply** *assumption*

Lots of preparations

**apply** (*frule-tac t=Rep-regT2 x* **in** *reg2dim2*)
**apply** (*frule-tac t=Rep-regT2 y* **in** *reg2dim2*)
**apply** (*frule-tac t=Rep-regT2 z* **in** *reg2dim2*)

**apply** (*simp only*: *Rep-regT2-inverse*)
**apply** (*cut-tac t1.0=Rep-regT2 y* **and** *t2.0=*(*tMap* (*const* (*h* (*delH1* (*Rep-regT2 z*)))) (*Rep-regT2 y*)) **in** *regSkelOuter2-vConc*)
  **apply** *assumption*
  **apply** (*simp* (*no-asm-simp*))
  **apply** (*rule conjI*)
  **apply** (*simp* (*no-asm-simp*))
  **apply** (*simp* (*no-asm-simp*))
  **apply** (*rule sym*, *simp* (*no-asm-simp*))
**apply** (*rotate-tac* −*1*, *frule regSkelOuter2-eq-Some*)
**apply** (*cut-tac t1.0=*(*tMap* (*const* (*h* (*delH1* (*Rep-regT2 y*)))) (*Rep-regT2 z*)) **and** *t2.0=Rep-regT2 z* **in** *regSkelOuter2-vConc*)
 **apply** (*simp* (*no-asm-simp*))
 **apply** (*rule conjI*)
 **apply** (*simp* (*no-asm-simp*))
 **apply** (*simp* (*no-asm-simp*))
 **apply** *assumption*
 **apply** (*simp* (*no-asm-simp*))
**apply** (*rotate-tac* −*1*, *frule regSkelOuter2-eq-Some*)
**apply** (*cut-tac t1.0=Rep-regT2 x* **and** *t2.0=*(*tMap* (*const* (*h* (*delH1* (*Rep-regT2 y*)))) (*Rep-regT2 x*)) **in** *regSkelOuter2-vConc*)
  **apply** *assumption*
  **apply** (*simp* (*no-asm-simp*))
  **apply** (*rule conjI*)
  **apply** (*simp* (*no-asm-simp*))
  **apply** (*simp* (*no-asm-simp*))
  **apply** (*rule sym*, *simp* (*no-asm-simp*))
**apply** (*rotate-tac* −*1*, *frule regSkelOuter2-eq-Some*)
**apply** (*cut-tac t1.0=*(*tMap* (*const* (*h* (*delH1* (*Rep-regT2 x*)))) (*Rep-regT2 y*)) **and** *t2.0=Rep-regT2 y* **in** *regSkelOuter2-vConc*)
 **apply** (*simp* (*no-asm-simp*))
 **apply** (*rule conjI*)
 **apply** (*simp* (*no-asm-simp*))
 **apply** (*simp* (*no-asm-simp*))
 **apply** *assumption*
 **apply** (*simp* (*no-asm-simp*))
**apply** (*rotate-tac* −*1*, *frule regSkelOuter2-eq-Some*)

end of preparations

The remainder is rewriting to normal form with *vConc-hConc*.

**apply** (*simp* (*no-asm-simp*) *del*: *const*)
**apply** (*subst vConc-hConc*)
 **apply** (*simp* (*no-asm-simp*))
 **apply** (*rule conjI*)
 **apply** (*simp* (*no-asm-simp*))
 **apply** (*simp* (*no-asm-simp*))
 **apply** (*simp* (*no-asm-simp*))
 **apply** *assumption*
 **apply** (*simp* (*no-asm-simp*))

```
    apply (subst headers-delH1--reg2, assumption)
    apply (subst headers-delH1--reg2, assumption)
  apply (simp (no-asm-simp))
apply (subst vConc-hConc)
  apply assumption
  apply (simp (no-asm-simp) del: const)
  apply (rule conjI)
  apply (simp (no-asm-simp))
    apply (subst headers-delH1--reg2, assumption)
    apply (subst headers-delH1--reg2, assumption)
  apply (simp (no-asm-simp))
  apply (simp (no-asm-simp) del: const)
  apply (simp (no-asm-simp) del: const)
apply (simp (no-asm-simp) del: const)
apply (subst tMap-const-vConc)
  apply (simp (no-asm-simp) del: const)
  apply (rule conjI)
  apply (simp (no-asm-simp))
    apply (subst headers-delH1--reg2, assumption)
  apply (simp (no-asm-simp))
  apply assumption
  apply (simp (no-asm-simp))
apply (subst tMap-const-vConc)
  apply (simp (no-asm-simp) del: const)
  apply (rule conjI)
  apply (simp (no-asm-simp))
  apply (simp (no-asm-simp))
  apply (simp (no-asm-simp) del: const)
  apply (rule conjI)
  apply (simp (no-asm-simp))
    apply (subst headers-delH1--reg2, assumption)
  apply (simp (no-asm-simp))
  apply (rule sym, simp (no-asm-simp))
apply (subst tMap-const-vConc)
  apply (simp (no-asm-simp) del: const)
  apply (rule conjI)
  apply (simp (no-asm-simp))
    apply (subst headers-delH1--reg2, assumption)
  apply (simp (no-asm-simp))
  apply assumption
  apply (simp (no-asm-simp))
apply (subst tMap-const-vConc)
  apply (simp (no-asm-simp) del: const)
  apply (rule conjI)
  apply (simp (no-asm-simp))
  apply (simp (no-asm-simp))
  apply (simp (no-asm-simp) del: const)
  apply (rule conjI)
  apply (simp (no-asm-simp))
```

```
    apply (subst headers-delH1--reg2, assumption)
  apply (simp (no-asm-simp))
  apply (rule sym, simp (no-asm-simp))
apply (subst tMap-const-hConc)+
apply (simp (no-asm-simp) del: const)
done
```

## 6.10 Lifting of Inversion Combinators to the Next Dimension

The initial inversion operator is *addH2*, as used in the definition of *inverse1*. The first argument $u$ is the contents of "empty" cells created by inversion. The second argument $h$ converts $u$ into an update function for *dConc*, and the third argument $g$ is the inversion operator of the next-lower dimension.

```
consts invLift0 ::
  'a ⇒
  ('a ⇒ ('e,'d) T ⇒ ('e,'d) T) ⇒
  ('a ⇒      'b    ⇒ ('c,'d) T) ⇒
  ('a ⇒ ('e,'b) T ⇒ ('c,'e,'d) regT2)

defs invLift0-def:
 invLift0 u h g h1 ==
  tFold (% h2 t2 . Abs-regT2 (addH2 h2 (g h1 t2))) (dConc (h u))

lemma invLift0-addH[simp]:
 invLift0 u h g h1 (addH h2 t2) = Abs-regT2 (addH2 h2 (g h1 t2))
by (unfold invLift0-def, simp)

lemma invLift0-addH1[simp]:
 Rep-regT2 (invLift0 u h g h1 (addH h2 t2)) = addH2 h2 (g h1 t2)
apply simp
apply (subst Abs-regT2-inverse, rule regT2, simp)
apply (rule conjI, simp-all)
done

lemma reg2dim2-invLift0-addH[simp]:
 reg2dim2 (invLift0 u h g h1 (addH h2 t2)) = singleton h2
by simp

lemma cs-invLift0-addH[simp]:
⟦ ⋀ u v t . h u (g v t) = g u t ⟧ ⟹
  h u (delH2 (Rep-regT2 (invLift0 u h g h1 (addH h2 t2)))) = g u t2
apply simp
apply (subst Abs-regT2-inverse, rule regT2, simp)
apply (rule conjI, simp-all)
done

lemma invLift0-hConc[simp]:
```

⟦ ⋀ *t* . *headers* (*h u t*) = *headers t*;
⋀ *t* . *h u* (*h u t*) = *h u t*;
⋀ *t1 t2* . *h u* (*hConc t1 t2*) = *hConc* (*h u t1*) (*h u t2*)
⟧ ⟹ *invLift0 u h g h1* (*hConc t1 t2*) =
*dConc* (*h u*) (*invLift0 u h g h1 t1*) (*invLift0 u h g h1 t2*)
**by** (*unfold invLift0-def*, *simp*)

**lemma** *reg2dim2-invLift0*:
⟦ ⋀ *t* . *headers* (*h u t*) = *headers t*;
⋀ *t* . *h u* (*h u t*) = *h u t*;
⋀ *t1 t2* . *h u* (*hConc t1 t2*) = *hConc* (*h u t1*) (*h u t2*);
⋀ *x t* . *headers* (*g x t*) = *headers t* ⟧ ⟹
*reg2dim2* (*invLift0 u h g h1 t*) = *headers t*
**apply** (*unfold invLift0-def*, *induct-tac t rule*: *T-induct*, *simp-all*)
**apply** (*fold invLift0-def*)
**apply** (*cut-tac x*=*invLift0 u h g h1 t1* **in** *Rep-regT2*)
**apply** (*cut-tac x*=*invLift0 u h g h1 t2* **in** *Rep-regT2*)
**apply** (*subst reg2dim2-dConc*)
**apply** *simp*
**apply** (*simp add*: *regT2-def*)
**apply** (*drule regularOuter2*, *erule exE*, *erule exE*, *simp*)
**apply** (*rule conjI*)
**apply** (*rule sym*, *erule regSkelOuter2-eq-Some*)
**apply** (*drule reg2dim2*)
**apply** (*rule sym*, *assumption*)
**apply** (*simp add*: *regT2-def*)
**apply** (*rotate-tac* −*1*, *drule regularOuter2*, *erule exE*, *erule exE*, *simp*)
**apply** (*rule conjI*)
**apply** (*rule sym*, *erule regSkelOuter2-eq-Some*)
**apply** (*drule reg2dim2*)
**apply** (*rule sym*, *assumption*)
**apply** (*simp add*: *Rep-regT2-inverse*)
**done**

**lemma** *regSkelOuter2-invLift0-EX*:
⟦ ⋀ *t* . *headers* (*h u t*) = *headers t*;
⋀ *t* . *h u* (*h u t*) = *h u t*;
⋀ *t1 t2* . *h u* (*hConc t1 t2*) = *hConc* (*h u t1*) (*h u t2*);
⋀ *x y t* . *headers* (*g x t*) = *headers* (*g y t*) ⟧ ⟹
*EX hs1* .
*regSkelOuter2* (*Rep-regT2* (*invLift0 u h g h1 t*)) = *Some* (*hs1*, *headers t*)
**apply** (*unfold invLift0-def*, *induct-tac t rule*: *T-induct*, *simp-all*)
**apply** (*subst Abs-regT2-inverse*, *rule regT2*, *simp*)
**apply** (*rule conjI*)
**apply** *simp*
**apply** *simp*
**apply** *simp*
**apply** (*fold invLift0-def*)
**apply** (*erule exE*, *erule exE*)

**apply** (*subst regSkelOuter2-dConc1*)
  **apply** *simp*
  **apply** *assumption*
  **apply** *assumption*
**apply** (*rule-tac x=append hs1 hs1a* **in** *exI*, *simp*)
**done**

**lemma** *cs-invLift0-hConc*[*simp*]:
 ⟦ ⋀ *t* . *headers* (*h u t*) = *headers t*;
   ⋀ *t* . *h u* (*h u t*) = *h u t*;
   ⋀ *t1 t2* . *h u* (*hConc t1 t2*) = *hConc* (*h u t1*) (*h u t2*)
 ⟧ ⟹
 *h u* (*delH1* (*Rep-regT2* (*invLift0 u h g h1* (*hConc t1 t2*)))) =
 *hConc* (*h u* (*delH1* (*Rep-regT2* (*invLift0 u h g h1 t1*))))
   (*h u* (*delH1* (*Rep-regT2* (*invLift0 u h g h1 t2*))))
**apply** (*cut-tac x=invLift0 u h g h1 t1* **in** *Rep-regT2*)
**apply** (*cut-tac x=invLift0 u h g h1 t2* **in** *Rep-regT2*)
**apply** (*simp add*: *regT2-def*)
**apply** (*drule regularOuter2*, *erule exE*, *erule exE*)
**apply** (*drule regularOuter2*, *erule exE*, *erule exE*)
**apply** (*subst cs-dConc* [*of h u*])
  **apply** *simp*
  **apply** *simp*
  **apply** *simp*
  **apply** *simp*
  **apply** (*rule conjI*)
  **apply** *simp*
  **apply** *simp*
  **apply** *simp*
  **apply** (*rule conjI*)
  **apply** *simp*
  **apply** *simp*
**apply** *simp*
**done**

**consts** *invLift* ::
 ′*a* ⇒
  (′*a* ⇒ (′*e*,′*d*) *T* ⇒ (′*e*,′*d*) *T*) ⇒
  (′*a* ⇒ ′*b* ⇒ (′*c*,′*d*) *T*) ⇒
  (′*a* ⇒ (′*e*,′*b*) *T* ⇒ (′*c*,(′*e*,′*d*) *T*) *T*)

**defs** *invLift-def*: *invLift u h g h1 t* == *Rep-regT2* (*invLift0 u h g h1 t*)

**lemma** *invLift-addH*[*simp*]:
 ⟦ ⋀ *x y t* . *headers* (*g x t*) = *headers* (*g y t*) ⟧ ⟹
  *invLift u h g h1* (*addH h2 t2*) = *addH2 h2* (*g h1 t2*)
**apply** (*unfold invLift-def*, *simp*)
**apply** (*subst Abs-regT2-inverse*, *simp-all add*: *regT2-def*)
**apply** (*rule regularOuter2I*, *auto*)

```
done

lemma headers-invLift0-invariant:
 ⟦ ⋀ t . headers (h u t) = headers t;
   ⋀ t . h u (h u t) = h u t;
   ⋀ t1 t2 . h u (hConc t1 t2) = hConc (h u t1) (h u t2);
   ⋀x y t. headers (g x t) = headers (g y t) ⟧ ⟹
  headers (Rep-regT2 (invLift0 u h g x t)) =
  headers (Rep-regT2 (invLift0 u h g y t))
apply (induct-tac t rule: T-induct)
apply (subst invLift0-addH1)
apply (subst invLift0-addH1)
apply simp
apply simp
done
```

The following lemma justifies the build-up for higher-dimensional inversion functions.

```
lemma headers-invLift-invariant:
 ⟦ ⋀ t . headers (h u t) = headers t;
   ⋀ t . h u (h u t) = h u t;
   ⋀ t1 t2 . h u (hConc t1 t2) = hConc (h u t1) (h u t2);
   ⋀ x y t . headers (g x t) = headers (g y t) ⟧ ⟹
   headers (invLift u h g x t) = headers (invLift u h g y t)
apply (unfold invLift-def)
apply (cut-tac x=invLift0 u h g x t in Rep-regT2)
apply (cut-tac x=invLift0 u h g y t in Rep-regT2)
apply (unfold regT2-def, simp)
apply (drule regularOuter2, erule exE, erule exE)
apply (drule regularOuter2, erule exE, erule exE)
apply (drule regSkelOuter2-eq-Some)
apply (drule regSkelOuter2-eq-Some)
apply (rule headers-invLift0-invariant)
apply simp-all
done

lemma invLift-hConc[simp]:
 ⟦ ⋀ t . headers (h u t) = headers t;
   ⋀ t . h u (h u t) = h u t;
   ⋀ t1 t2 . h u (hConc t1 t2) = hConc (h u t1) (h u t2);
   ⋀ x y t . headers (g x t) = headers (g y t);
   regSkelOuter2 (invLift u h g h1 t1) = Some (hs1a, hs2a);
   regSkelOuter2 (invLift u h g h1 t2) = Some (hs1b, hs2b)
  ⟧ ⟹
   invLift u h g h1 (hConc t1 t2) =
   hConc (vConc (invLift u h g h1 t1)
               (tMap (const (h u (delH1 (invLift u h g h1 t2)))) (invLift u h g h1
t1)))
       (vConc (tMap (const (h u (delH1 (invLift u h g h1 t1)))) (invLift u h g h1
```

*t2*))
            (*invLift u h g h1 t2*))
**apply** (*unfold invLift-def*, *simp*)
**apply** (*subst dConc2* [*THEN sym*], *auto*)
**done**

**lemma** *regSkelOuter2-invLift-EX*:
⟦ ⋀ *t* . *headers* (*h u t*) = *headers t*;
  ⋀ *t* . *h u* (*h u t*) = *h u t*;
  ⋀ *t1 t2* . *h u* (*hConc t1 t2*) = *hConc* (*h u t1*) (*h u t2*);
  ⋀ *x y t* . *headers* (*g x t*) = *headers* (*g y t*) ⟧ ⟹
 *EX hs1* . *regSkelOuter2* (*invLift u h g h1 t*) = *Some* (*hs1*, *headers t*)
**apply** (*unfold invLift-def*)
**apply** (*rule regSkelOuter2-invLift0-EX*)
**apply** *simp-all*
**done**

**lemma** *delH1-invLift*:
⟦ ⋀ *t* . *headers* (*h u t*) = *headers t*;
  ⋀ *t* . *h u* (*h u t*) = *h u t*;
  ⋀ *t1 t2* . *h u* (*hConc t1 t2*) = *hConc* (*h u t1*) (*h u t2*);
  ⋀ *x y t* . *headers* (*g x t*) = *headers* (*g y t*)
 ⟧ ⟹
 *delH1* (*invLift u h g v t*) = *tFold* (% *h0 t0* . *addH h0* (*delH1* (*g v t0*))) (*hConcU*
(*h u*)) *t*
**apply** (*induct-tac t rule*: *T-induct*, *simp-all del*: *const*)
**apply** (*cut-tac u*=*u* **and** *h*=*h* **and** *g*=*g* **and** *h1.0*=*v* **and** *t*=*t1* **in** *regSkelOuter2-invLift-EX*)
  **apply** (*simp* (*no-asm-simp*))
  **apply** (*simp* (*no-asm-simp*))
  **apply** (*simp* (*no-asm-simp*))
  **apply** (*simp* (*no-asm-simp*))
  **apply** (*erule exE*)
**apply** (*cut-tac u*=*u* **and** *h*=*h* **and** *g*=*g* **and** *h1.0*=*v* **and** *t*=*t2* **in** *regSkelOuter2-invLift-EX*)
  **apply** (*simp* (*no-asm-simp*))
  **apply** (*simp* (*no-asm-simp*))
  **apply** (*simp* (*no-asm-simp*))
  **apply** (*simp* (*no-asm-simp*))
  **apply** (*erule exE*)
**apply** (*subst invLift-hConc*)
 **apply** (*simp* (*no-asm-simp*))
 **apply** (*simp* (*no-asm-simp*))
 **apply** (*simp* (*no-asm-simp*))
 **apply** (*simp* (*no-asm-simp*))
 **apply** *assumption*
 **apply** *assumption*
**apply** (*simp* (*no-asm-simp*))
**apply** (*subst delH1-vConc*)
 **apply** *assumption*
 **apply** (*simp* (*no-asm-simp*))

**apply** (*rule conjI*, *simp* (*no-asm-simp*), *simp* (*no-asm-simp*))
**apply** (*subst delH1-tMap-const*)
**apply** (*simp* (*no-asm-simp*))
**apply** (*fold hConcU-def*)
**apply** (*cut-tac u=h u* **in** *hConcU-assoc*)
**apply** (*simp* (*no-asm-simp*))
**apply** (*simp* (*no-asm-simp*))
**apply** (*simp del*: *hConcU-def*)
**done**

**lemma** *h-u-delH1-invLift*:
⟦ ⋀ *t* . *headers* (*h u t*) = *headers t*;
⋀ *t x* . *h u* (*h x t*) = *h u t*;
⋀ *t1 t2* . *h u* (*hConc t1 t2*) = *hConc* (*h u t1*) (*h u t2*);
⋀ *x y t* . *headers* (*g x t*) = *headers* (*g y t*);
⋀ *h0 t0* . *h u* (*addH h0* (*delH1* (*g v t0*))) = *addH h0* (*delH1* (*g u t0*))
⟧ ⟹
*h u* (*delH1* (*invLift u h g v t*)) = *tFold* (% *h0 t0* . *addH h0* (*delH1* (*g u t0*)))
*hConc t*
**apply** (*subst delH1-invLift*)
**apply** (*simp* (*no-asm-simp*))
**apply** (*simp* (*no-asm-simp*))
**apply** (*simp* (*no-asm-simp*))
**apply** (*simp* (*no-asm-simp*))
**apply** (*simp* (*no-asm-simp*))
**done**

**lemma** *headers-h-u-delH1-invLift*[*simp*]:
⟦ ⋀ *t* . *headers* (*h u t*) = *headers t*;
⋀ *t x* . *h u* (*h x t*) = *h u t*;
⋀ *t1 t2* . *h u* (*hConc t1 t2*) = *hConc* (*h u t1*) (*h u t2*);
⋀ *x y t* . *headers* (*g x t*) = *headers* (*g y t*);
⋀ *h0 t0* . *h u* (*addH h0* (*delH1* (*g v t0*))) = *addH h0* (*delH1* (*g u t0*))
⟧ ⟹
*headers* (*h u* (*delH1* (*invLift u h g v t*))) = *headers t*
**apply** (*subst h-u-delH1-invLift* [*of h u g v t*])
**apply** (*simp* (*no-asm-simp*))
**apply** (*simp* (*no-asm-simp*))
**apply** (*simp* (*no-asm-simp*))
**apply** (*simp* (*no-asm-simp*))
**apply** (*simp* (*no-asm-simp*))
**apply** (*simp* (*no-asm-simp*))
**apply** (*subst headers-def*)
**apply** (*simp* (*no-asm-simp*))
**done**

**lemma** *headers-delH1-invLift*[*simp*]:
⟦ ⋀ *t* . *headers* (*h u t*) = *headers t*;
⋀ *t x* . *h u* (*h x t*) = *h u t*;

⋀ *t1 t2 . h u (hConc t1 t2) = hConc (h u t1) (h u t2);*
⋀ *x y t . headers (g x t) = headers (g y t);*
⋀ *h0 t0 . h u (addH h0 (delH1 (g v t0))) = addH h0 (delH1 (g u t0))*
⟧ ⟹
*headers (delH1 (invLift u h g v t)) = headers t*
**apply** (*cut-tac headers-h-u-delH1-invLift [of h u g v t]*)
**apply** *simp*
**apply** (*simp (no-asm-simp)*)
**apply** (*simp (no-asm-simp)*)
**apply** (*simp (no-asm-simp)*)
**apply** (*simp (no-asm-simp)*)
**apply** (*simp (no-asm-simp)*)
**done**

**lemma** *regSkelOuter2-dConcFill*:
⟦ ⋀ *t . headers (h u t) = headers t;*
⋀ *t x . h u (h x t) = h u t;*
⋀ *t1 t2 . h u (hConc t1 t2) = hConc (h u t1) (h u t2);*
⋀ *x y t . headers (g x t) = headers (g y t);*
*regSkelOuter2 (invLift u h g h1 t1) = Some (hs1a, hs2a);*
*regSkelOuter2 (invLift u h g h1 t2) = Some (hs1b, hs2b)*
⟧ ⟹
*regSkelOuter2 (tMap (const (h u (delH1 (invLift u h g h1 t2))))*
*(invLift u h g h1 t1)) = Some (hs1a,hs2b)*
**apply** (*cut-tac t2.0=h u (delH1 (invLift u h g h1 t2))* **and** *t1.0=invLift u h g h1 t1* **in** *regSkelOuter2-tMap-const*)
**apply** *simp*
**done**

**lemma** *headers-invLift-hConc[simp]*:
⟦ ⋀ *t . headers (h u t) = headers t;*
⋀ *t x . h u (h x t) = h u t;*
⋀ *t1 t2 . h u (hConc t1 t2) = hConc (h u t1) (h u t2);*
⋀ *x y t . headers (g x t) = headers (g y t);*
*regSkelOuter2 (invLift u h g h1 t1) = Some (hs1a, hs2a);*
*regSkelOuter2 (invLift u h g h1 t2) = Some (hs1b, hs2b)*
⟧ ⟹
*headers (invLift u h g h1 (hConc t1 t2)) =*
*append (headers (invLift u h g h1 t1))*
*(headers (invLift u h g h1 t2))*
**apply** (*subst invLift-hConc*)
**apply** (*simp (no-asm-simp)*)
**apply** (*simp (no-asm-simp)*)
**apply** (*simp (no-asm-simp)*)
**apply** (*simp (no-asm-simp)*)
**apply** *assumption*
**apply** *assumption*
**apply** (*cut-tac h=h* **and** *u=u* **and** *g=g* **and** *h1.0=h1* **and**
*t1.0=t1* **and** *hs1a=hs1a* **and** *hs2a=hs2a* **and**

```
            t2.0=t2 and hs1b=hs1b and hs2b=hs2b in regSkelOuter2-dConcFill)
  apply (simp (no-asm-simp))
  apply (simp (no-asm-simp))
  apply (simp (no-asm-simp))
  apply (simp (no-asm-simp))
  apply assumption
  apply assumption
apply (cut-tac h=h and u=u and g=g and h1.0=h1 and
             t1.0=t2 and hs1a=hs1b and hs2a=hs2b and
            t2.0=t1 and hs1b=hs1a and hs2b=hs2a in regSkelOuter2-dConcFill)
  apply (simp (no-asm-simp))
  apply (simp (no-asm-simp))
  apply (simp (no-asm-simp))
  apply (simp (no-asm-simp))
  apply assumption
  apply assumption
apply (simp (no-asm-simp))
apply (subst headers-vConc)
  apply assumption
  apply assumption
apply (subst headers-vConc)
  apply assumption
  apply assumption
apply (simp (no-asm-simp))
done

lemma headers-invLift[simp]:
 ⟦ ⋀ t . headers (h u t) = headers t;
   ⋀ t x . h u (h x t) = h u t;
   ⋀ t1 t2 . h u (hConc t1 t2) = hConc (h u t1) (h u t2);
   ⋀ x y t . headers (g x t) = headers (g y t)
  ⟧ ⟹
   headers (invLift u h g h1 t) = tFold (% h2 t2 . headers (g h1 t2)) append t
apply (induct-tac t rule: T-induct, simp)
apply (cut-tac u=u and h=h and g=g and h1.0=h1 and t=t1 in regSkelOuter2-invLift-EX)
    apply (simp (no-asm-simp))
    apply (simp (no-asm-simp))
    apply (simp (no-asm-simp))
    apply (simp (no-asm-simp))
    apply (erule exE)
apply (cut-tac u=u and h=h and g=g and h1.0=h1 and t=t2 in regSkelOuter2-invLift-EX)
    apply (simp (no-asm-simp))
    apply (simp (no-asm-simp))
    apply (simp (no-asm-simp))
    apply (simp (no-asm-simp))
    apply (erule exE)
apply (subst headers-invLift-hConc)
  apply (simp (no-asm-simp))
  apply (simp (no-asm-simp))
```

```
  apply (simp (no-asm-simp))
  apply (simp (no-asm-simp))
  apply assumption
  apply assumption
apply (simp (no-asm-simp))
done

lemma regSkelOuter2-invLift[simp]:
 ⟦ ⋀ t . headers (h u t) = headers t;
   ⋀ t x . h u (h x t) = h u t;
   ⋀ t1 t2 . h u (hConc t1 t2) = hConc (h u t1) (h u t2);
   ⋀ x y t . headers (g x t) = headers (g y t);
   ⋀ h0 t0 . h u (addH h0 (delH1 (g h1 t0))) = addH h0 (delH1 (g u t0))
  ⟧ ⟹
   regSkelOuter2 (invLift u h g h1 t) =
   Some (tFold (% h2 t2 . headers (g h1 t2)) append t, headers t)
apply (induct-tac t rule: T-induct, simp)
apply (cut-tac u=u and h=h and g=g and h1.0=h1 and t=t1 in regSkelOuter2-invLift-EX)
    apply (simp (no-asm-simp))
    apply (simp (no-asm-simp))
    apply (simp (no-asm-simp))
    apply (simp (no-asm-simp))
    apply (erule exE)
apply (cut-tac u=u and h=h and g=g and h1.0=h1 and t=t2 in regSkelOuter2-invLift-EX)
    apply (simp (no-asm-simp))
    apply (simp (no-asm-simp))
    apply (simp (no-asm-simp))
    apply (simp (no-asm-simp))
    apply (erule exE)
apply (subst invLift-hConc)
  apply (simp (no-asm-simp))
  apply (simp (no-asm-simp))
  apply (simp (no-asm-simp))
  apply (simp (no-asm-simp))
  apply assumption
  apply assumption
apply (subst regSkelOuter2-hConc)
apply (cut-tac h=h and u=u and g=g and h1.0=h1 and
             t1.0=t1 and hs1a=hs1 and hs2a=headers t1 and
         t2.0=t2 and hs1b=hs1a and hs2b=headers t2 in regSkelOuter2-dConcFill)
  apply (simp (no-asm-simp))
  apply (simp (no-asm-simp))
  apply (simp (no-asm-simp))
  apply (simp (no-asm-simp))
  apply assumption
  apply assumption
apply (cut-tac h=h and u=u and g=g and h1.0=h1 and
             t1.0=t2 and hs1a=hs1a and hs2a=headers t2 and
         t2.0=t1 and hs1b=hs1 and hs2b=headers t1 in regSkelOuter2-dConcFill)
```

apply (*simp* (*no-asm-simp*))
apply (*simp* (*no-asm-simp*))
apply (*simp* (*no-asm-simp*))
apply (*simp* (*no-asm-simp*))
apply *assumption*
apply *assumption*
apply (*subst regSkelOuter2-vConc*)
apply (*simp* (*no-asm-simp*))
apply (*rule conjI*)
apply (*simp* (*no-asm-simp*))
apply (*simp* (*no-asm-simp*))
apply *assumption*
apply (*simp* (*no-asm-simp*))
apply (*subst regSkelOuter2-vConc*)
apply *assumption*
apply (*simp* (*no-asm-simp*))
apply (*rule conjI*)
apply (*simp* (*no-asm-simp*))
apply (*simp* (*no-asm-simp*))
apply (*simp* (*no-asm-simp*))
apply (*simp* (*no-asm-simp*))
done

## 6.11 The Inversion Operator for Two Dimensions

**consts** *invOp2* :: $'a \Rightarrow 'a \Rightarrow ('b,('c,'d)\ T)\ T \Rightarrow ('c,('b,('a,'d)\ T)\ T)\ T$

**defs** *invOp2-def*: *invOp2 u* == *invLift u slimH2 addH2*

**lemma** *delH1-invOp2*:
*delH1* (*invOp2 u v t*) = *tFold* (% *h t* . *addH h* (*slimH1 v t*)) (*hConcSH2 u*) *t*
**apply** (*unfold invOp2-def*)
**apply** (*subst delH1-invLift*)
**apply** (*simp* (*no-asm-simp*))
**apply** (*simp* (*no-asm-simp*))
**apply** (*simp* (*no-asm-simp*))
**apply** (*simp* (*no-asm-simp*))
**apply** (*simp del*: *tFold-hConc hConcU-def slimH2-def*)
**apply** (*fold hConcSH2-def*)
**apply** (*rule refl*)
**done**

**lemma** *headers-invOp2*[*simp*]:
*headers* (*invOp2 u v t*) = *tFold* (% *h* . *headers*) *append t*
**apply** (*unfold invOp2-def*)
**apply** (*subst headers-invLift*)
**apply** (*simp* (*no-asm-simp*))
**apply** (*simp* (*no-asm-simp*))
**apply** (*simp* (*no-asm-simp*))

**apply** (*simp* (*no-asm-simp*))
**apply** (*simp* (*no-asm-simp*))
**done**

**lemma** *cs-invOp2*[*simp*]:
*slimH2 u* (*delH1* (*invOp2 u v t*)) = *tFold* (% *h t* . *addH h* (*slimH1 u t*)) *hConc t*
**apply** (*subst delH1-invOp2*)
**apply** (*induct-tac t rule*: *T-induct*, *simp-all del*: *slimH2-def*)
**done**

**lemma** *regSkelOuter2-invOp2*:
*regSkelOuter2* (*invOp2 u h1 t*) = *Some* (*tFold* (% *h* . *headers*) *append t*, *headers t*)
**apply** (*cut-tac u*=*u* **and** *h*=*slimH2* **and** *g*=*addH2* **and** *h1.0*=*h1* **and** *t*=*t* **in** *regSkelOuter2-invLift*)
**apply** (*simp* (*no-asm-simp*))
**apply** (*simp* (*no-asm-simp*))
**apply** (*simp* (*no-asm-simp*))
**apply** (*simp* (*no-asm-simp*))
**apply** (*simp* (*no-asm-simp*))
**apply** (*fold invOp2-def*)
**apply** (*simp* (*no-asm-simp*))
**done**

**lemma** *invOp2-hConc*[*simp*]:
⟦*regSkelOuter2* (*invOp2 u h1 t1*) = *Some* (*hs1a*, *hs2a*);
*regSkelOuter2* (*invOp2 u h1 t2*) = *Some* (*hs1b*, *hs2b*)
⟧ ⟹
*invOp2 u h1* (*hConc t1 t2*) =
*hConc* (*vConc* (*invOp2 u h1 t1*)
(*spread1 hs1a* (*slimH2 u t2*)))
(*vConc* (*spread1 hs1b* (*slimH2 u t1*))
(*invOp2 u h1 t2*))
**apply** (*unfold invOp2-def*, *subst invLift-hConc*)
**apply** (*simp* (*no-asm-simp*))
**apply** (*simp* (*no-asm-simp*))
**apply** (*simp* (*no-asm-simp*))
**apply** (*simp* (*no-asm-simp*))
**apply** *assumption*
**apply** *assumption*
**apply** (*fold invOp2-def*)
**apply** (*subst cs-invOp2*)
**apply** (*subst cs-invOp2*)
**apply** (*simp* (*no-asm-simp*))
**apply** (*subst tMap-CONST--headers*)
**apply** (*subst tMap-CONST--headers*)
**apply** (*subst regSkelOuter2-eq-Some* [*of invOp2 u h1 t1 hs1a hs2a*])
**apply** *assumption*
**apply** (*subst regSkelOuter2-eq-Some* [*of invOp2 u h1 t2*])

**apply** *assumption*
**apply** (*fold slimH2-def*)
**apply** (*unfold spread1-def*)
**apply** (*fold slimH1-def*)
**apply** (*simp* (*no-asm-simp*) *only*: *slimH2-as-tFold*)
**done**

### 6.12 Two-Dimensional Inversion

**consts** *inverse2* :: $'a \Rightarrow ('a,('b,('c,'d)\ T)\ T)\ T$
$\Rightarrow ('c,('b,('a,'d)\ T)\ T)\ T$

**defs** *inverse2-def*: *inverse2 u* == *tFold* (*invLift u slimH2 addH2*) *hConc*

**lemma** *inverse2-addH*[*simp*]: *inverse2 u* (*addH h t*) = *invLift u slimH2 addH2 h t*
**by** (*unfold inverse2-def*, *simp*)

**lemma** *inverse2-hConc*[*simp*]:
*inverse2 u* (*hConc t1 t2*) = *hConc* (*inverse2 u t1*) (*inverse2 u t2*)
**by** (*unfold inverse2-def*, *simp*)

**lemma** *tFold-tFold-tFold-inverse2*:
⟦ *assoc c2*; *assoc c3*; *assoc c1*; *assoc c5*; *assoc c6*;
⋀ *h ha hb t* . *a4 hb* (*a5 ha* (*a6 h t*)) = *a1 h* (*a2 ha* (*a3 hb t*));
⋀ *h ha t1 t2* .
*c1* (*a1 h* (*a2 ha t1*)) (*a1 h* (*a2 ha t2*)) = *a1 h* (*a2 ha* (*c3 t1 t2*));
(∗ *The next two are for tFold-tFold-vConc*: ∗)
⋀ *x y z* . *a4 x* (*c5 y z*) = *c5* (*a4 x y*) (*a4 x z*);
⋀ *x1 x2 y1 y2* . *c1* (*c5 x1 y1*) (*c5 x2 y2*) = *c5* (*c1 x1 x2*) (*c1 y1 y2*);
(∗ *units*: ∗)
⋀ *x* . *LRunit c1* (*a1 u x*);
⋀ *x y* . *LRunit c5* (*a5 y* (*a6 u x*));
⋀ *x y z* . *LRunit c1* (*a1 z* (*a5 y* (*a6 u x*)));
(∗ *finishing 1.2*: ∗)
⋀ *h x1 x2* . *c5* (*a1 h x1*) (*a1 h x2*) = *a1 h* (*c2 x1 x2*)
⟧ ⟹
*tFold a4 c1* (*tMap* (*tFold a5 c5 o* (*tMap* (*tFold a6 c6*))) (*inverse2 u t*)) =
*tFold a1 c1* (*tMap* (*tFold a2 c2 o* (*tMap* (*tFold a3 c3*))) *t*)
**apply** (*induct-tac t rule*: *T-induct*)

1: *addH h t0*

**apply** (*induct-tac t0 rule*: *T-induct*)

1.1: *t0* = *addH ha t0a*

**apply** (*induct-tac t0a rule*: *T-induct*, *simp*)

1.1.2: *t0a* = *hConc t1 t2*

**apply** *simp*

1.2: *t0* = *hConc t1 t2*

**apply** (*simp del*: *slimH2-def tFold-tMap*)
**apply** (*fold invOp2-def*)
**apply** (*cut-tac u=u* **and** *h1.0=h* **and** *t=t1* **in** *regSkelOuter2-invOp2*)
**apply** (*cut-tac u=u* **and** *h1.0=h* **and** *t=t2* **in** *regSkelOuter2-invOp2*)
**apply** (*subst invOp2-hConc*)
  **apply** *assumption*
  **apply** *assumption*
**apply** (*subgoal-tac ALL t . tFold a5 c5* (*tMap* (*tFold a6 c6*) *t*) =
          *tFold* (λ*h t0*. *a5 h* (*tFold a6 c6 t0*)) *c5 t*)
  **prefer** *2*
  **apply** (*rule allI*, *simp* (*no-asm-simp*))
**apply** (*simp* (*no-asm-simp*) *del*: *slimH2-def tFold-tMap*)
**apply** (*subst tFold-tFold-vConc* [*of c5 c1 a4*])
  **apply** *assumption*
  **apply** *assumption*
  **apply** (*simp* (*no-asm-simp*))
  **apply** (*simp* (*no-asm-simp*))
  **apply** (*simp* (*no-asm-simp*))
    **apply** (*rule conjI*)
    **apply** (*rule refl*)
    **apply** (*rule refl*)
  **apply** *assumption*
  **apply** (*rule refl*)
**apply** (*subst tFold-tFold-vConc* [*of c5 c1 a4*])
  **apply** *assumption*
  **apply** *assumption*
  **apply** (*simp* (*no-asm-simp*))
  **apply** (*simp* (*no-asm-simp*))
  **apply** *assumption*
  **apply** (*simp* (*no-asm-simp*) *del*: *slimH2-def*)
    **apply** (*rule conjI*)
    **apply** (*rule refl*)
    **apply** (*rule refl*)
  **apply** (*rule refl*)
**apply** (*simp* (*no-asm-simp*) *only*: *tMap-f-spread1*)
**apply** (*simp* (*no-asm-simp*) *only*: *tFold-slimH2 tFold-slimH1*)
**apply** (*simp* (*no-asm-simp*) *only*: *f-tFold-const*)
**apply** (*simp* (*no-asm-simp*) *only*: *tFold-spread1*)
**apply** (*subgoal-tac* (λ*h*. *a4 h* (*tFold* (λ*h0*. *tFold* (λ*h0a t0*. *a5 h0* (*a6 u t0*)) *const*) *c5 t1*)) =
   (λ*h*. *a4 h* (*tFold* (λ*h0*. *tFold* (λ*h0a t0*. *a5 arbitrary* (*a6 u arbitrary*)) *const*) *c5 t1*)))
  **prefer** *2*
  **apply** (*rule ext*)
  **apply** (*rule-tac f=a4 ha* **in** *arg-cong*)
  **apply** (*rule-tac x=t1* **in** *fun-cong*)
  **apply** (*rule-tac x=c5* **in** *fun-cong*)
  **apply** (*rule-tac f=tFold* **in** *arg-cong*)
  **apply** (*rule ext*)

**apply** (*rule fun-cong* [*of - - const*])
**apply** (*rule arg-cong* [*of - - tFold*])
**apply** (*rule ext*)
**apply** (*rule ext*)
**apply** (*cut-tac c=c5* **and** *F=%x . a5 x* (*a6 u t0*) **and** *x=h0* **in** *LRunit-const*)
**apply** (*simp* (*no-asm-simp*))
**apply** (*cut-tac c=c5* **and** *F=%x . a5 arbitrary* (*a6 u x*) **and** *x=t0* **in** *LRunit-const*)
**apply** (*simp* (*no-asm-simp*))
**apply** (*simp* (*no-asm-simp*))
**apply** (*subgoal-tac* (*λh. a4 h* (*tFold* (*λh0. tFold* (*λh0a t0. a5 h0* (*a6 u t0*)) *const*) *c5 t2*)) =
(*λh. a4 h* (*tFold* (*λh0. tFold* (*λh0a t0. a5 arbitrary* (*a6 u arbitrary*)) *const*) *c5 t2*)))
**prefer** *2*
**apply** (*rule ext*)
**apply** (*rule-tac f=a4 ha* **in** *arg-cong*)
**apply** (*rule-tac x=t2* **in** *fun-cong*)
**apply** (*rule-tac x=c5* **in** *fun-cong*)
**apply** (*rule-tac f=tFold* **in** *arg-cong*)
**apply** (*rule ext*)
**apply** (*rule fun-cong* [*of - - const*])
**apply** (*rule arg-cong* [*of - - tFold*])
**apply** (*rule ext*)
**apply** (*rule ext*)
**apply** (*cut-tac c=c5* **and** *F=%x . a5 x* (*a6 u t0*) **and** *x=h0* **in** *LRunit-const*)
**apply** (*simp* (*no-asm-simp*))
**apply** (*cut-tac c=c5* **and** *F=%x . a5 arbitrary* (*a6 u x*) **and** *x=t0* **in** *LRunit-const*)
**apply** (*simp* (*no-asm-simp*))
**apply** (*simp* (*no-asm-simp*))
**apply** (*rotate-tac −1*, *simp* (*no-asm-simp*) *del*: *slimH2-def tFold-tMap*)
**apply** (*thin-tac ALL t . tFold a5 c5* (*tMap* (*tFold a6 c6*) *t*) =
*tFold* (*λh t0. a5 h* (*tFold a6 c6 t0*)) *c5 t*)
**apply** (*subgoal-tac tFold* (*λh t0. a5 h* (*tFold a6 c6 t0*)) *c5* = (*% t . tFold a5 c5* (*tMap* (*tFold a6 c6*) *t*)))
**prefer** *2*
**apply** (*rule ext*)
**apply** (*subst tFold-tMap*)
**apply** *assumption*
**apply** (*rule refl*)
**apply** (*simp* (*no-asm-simp*) *del*: *tFold-tMap add*: *o-def*)
**apply** (*thin-tac tFold* (*λh t0. a5 h* (*tFold a6 c6 t0*)) *c5* = (*% t . tFold a5 c5* (*tMap* (*tFold a6 c6*) *t*)))
**apply** (*simp* (*no-asm-simp*) *add*: *o-def LRunit-left LRunit-right*)

2: *t = hConc t1 t2*

**apply** (*simp del*: *slimH2-def tFold-tMap*)
**done**

```
lemma tFold-tFold-tFold0-inverse2:
 ⟦ assoc c2; assoc c3; assoc c1; assoc c5; assoc c6;
    ⋀ h ha hb t . a4 hb (a5 ha h) = a1 h (a2 ha hb);
    ⋀ h ha t1 t2 .
      c1 (a1 h (a2 ha t1)) (a1 h (a2 ha t2)) = a1 h (a2 ha (c3 t1 t2));
   (* The next two are for tFold-tFold-vConc: *)
    ⋀ x y z . a4 x (c5 y z) = c5 (a4 x y) (a4 x z);
    ⋀ x1 x2 y1 y2 . c1 (c5 x1 y1) (c5 x2 y2) = c5 (c1 x1 x2) (c1 y1 y2);
   (* units: *)
    ⋀ x . LRunit c1 (a1 u x);
    ⋀ x y . LRunit c5 (a5 y u);
    ⋀ x y z . LRunit c1 (a1 z (a5 y u));
   (* finishing 1.2: *)
   ⋀ h x1 x2 . c5 (a1 h x1) (a1 h x2) = a1 h (c2 x1 x2)
  ⟧ ⟹
  tFold a4 c1 (tMap (tFold a5 c5 o (tMap (tFold0 c6))) (inverse2 u t)) =
  tFold a1 c1 (tMap (tFold a2 c2 o (tMap (tFold0 c3))) t)
apply (unfold tFold0-def)
apply (rule tFold-tFold-tFold-inverse2)
apply (simp-all (no-asm-simp))
done
```

Adapting for cells at lowest level:

```
lemma tFold-tFold-tFoldC-inverse2:
 ⟦ assoc c1; assoc c2; assoc c3; assoc c5; assoc c6;
    ⋀ h ha hb t . a4 hb (a5 ha (a6 h)) = a1 h (a2 ha (a3 hb));
    ⋀ h ha t1 t2 .
      c1 (a1 h (a2 ha t1)) (a1 h (a2 ha t2)) = a1 h (a2 ha (c3 t1 t2));
   (* The next two are for tFold-tFold-vConc: *)
    ⋀ x y z . a4 x (c5 y z) = c5 (a4 x y) (a4 x z);
    ⋀ x1 x2 y1 y2 . c1 (c5 x1 y1) (c5 x2 y2) = c5 (c1 x1 x2) (c1 y1 y2);
   (* units: *)
    ⋀ x . LRunit c1 (a1 u x);
    ⋀ x y . LRunit c5 (a5 y (a6 u));
    ⋀ x y z . LRunit c1 (a1 z (a5 y (a6 u)));
   (* finishing 1.2: *)
   ⋀ h x1 x2 . c5 (a1 h x1) (a1 h x2) = a1 h (c2 x1 x2)
  ⟧ ⟹
  tFold a4 c1 (tMap (tFold a5 c5 o (tMap (tFoldC a6 c6))) (inverse2 u t)) =
  tFold a1 c1 (tMap (tFold a2 c2 o (tMap (tFoldC a3 c3))) t)
apply (unfold tFoldC-def)
apply (rule tFold-tFold-tFold-inverse2)
apply assumption+
apply (simp-all (no-asm-simp))
done
```

```
lemma tFold-tFold-tFold-inverse2-wrapped:
 ⟦ assoc c1; assoc c2; assoc c3; assoc c4; assoc c5; assoc c6;
    ⋀ h ha hb t . w2 (a4 hb (a5 ha (a6 h t))) = w1 (a1 h (a2 ha (a3 hb t)));
```

$\bigwedge$ *x y . w2 (c4 x y) = c7 (w2 x) (w2 y)*;
$\bigwedge$ *x y . c7 (w1 x) (w1 y) = w1 (c1 x y)*;
$\bigwedge$ *x y . w2 (c5 x y) = c8 (w2 x) (w2 y)*;
$\bigwedge$ *h x y . c7 (w1 (a1 h x)) (w1 (a1 h y)) = w1 (a1 h (c2 x y))*;
$\bigwedge$ *h ha x y .*
*c7 (w1 (a1 h (a2 ha x))) (w1 (a1 h (a2 ha y))) =*
*w1 (a1 h (a2 ha (c3 x y)))*;
(∗ *The next two are for tFold-tFold-vConc*: ∗)
$\bigwedge$ *x y z . a4 x (c5 y z) = c5 (a4 x y) (a4 x z)*;
$\bigwedge$ *x1 x2 y1 y2 . c4 (c5 x1 y1) (c5 x2 y2) = c5 (c4 x1 x2) (c4 y1 y2)*;
(∗ *units*: ∗)
$\bigwedge$ *x . LRunit c8 (w1 (a1 u x))*;
$\bigwedge$ *x y . LRunit c5 (a5 y (a6 u x))*;
$\bigwedge$ *x y z . LRunit c4 (a4 z (a5 y (a6 u x)))*;
(∗ *finishing 1.2*: ∗)
$\bigwedge$ *h x1 x2 . c5 (a1 h x1) (a1 h x2) = a1 h (c2 x1 x2)*
$\rrbracket \Longrightarrow$
*w2 (tFold a4 c4 (tMap (tFold a5 c5 o (tMap (tFold a6 c6))) (inverse2 u t))) =*
*w1 (tFold a1 c1 (tMap (tFold a2 c2 o (tMap (tFold a3 c3))) t))*
**apply** (*induct-tac t rule*: *T-induct*)

1: *addH h t0*

**apply** (*induct-tac t0 rule*: *T-induct*)

1.1: *t0 = addH ha t0a*

**apply** (*induct-tac t0a rule*: *T-induct*, *simp* (*no-asm-simp*))

1.1.2: *t0a = hConc t1 t2*

**apply** *simp*

1.2: *t0 = hConc t1 t2*

**apply** (*simp del*: *slimH2-def tFold-tMap*)
**apply** (*fold invOp2-def*)
**apply** (*cut-tac u=u* **and** *h1.0=h* **and** *t=t1* **in** *regSkelOuter2-invOp2*)
**apply** (*cut-tac u=u* **and** *h1.0=h* **and** *t=t2* **in** *regSkelOuter2-invOp2*)
**apply** (*subst invOp2-hConc*)
**apply** *assumption*
**apply** *assumption*
**apply** (*subgoal-tac ALL t . tFold a5 c5 (tMap (tFold a6 c6) t) =*
*tFold* ($\lambda$*h t0. a5 h (tFold a6 c6 t0)) c5 t*)
**prefer** *2*
**apply** (*rule allI*, *simp* (*no-asm-simp*))
**apply** (*simp* (*no-asm-simp*) *del*: *slimH2-def tFold-tMap*)
**apply** (*subst tFold-tFold-vConc* [*of c5 c4 a4*])
**apply** *assumption*
**apply** *assumption*
**apply** (*simp* (*no-asm-simp*))
**apply** (*simp* (*no-asm-simp*))
**apply** (*simp* (*no-asm-simp*))

**apply** (*rule conjI*, *rule refl*, *rule refl*)
**apply** *assumption*
**apply** (*rule refl*)
**apply** (*subst tFold-tFold-vConc* [*of c5 c4 a4*])
**apply** *assumption*
**apply** *assumption*
**apply** (*simp* (*no-asm-simp*))
**apply** (*simp* (*no-asm-simp*))
**apply** *assumption*
**apply** (*simp* (*no-asm-simp*) *del*: *slimH2-def*)
**apply** (*rule conjI*, *rule refl*, *rule refl*)
**apply** (*rule refl*)
**apply** (*simp* (*no-asm-simp*) *only*: *tMap-f-spread1*)
**apply** (*simp* (*no-asm-simp*) *only*: *tFold-slimH2 tFold-slimH1*)
**apply** (*simp* (*no-asm-simp*) *only*: *f-tFold-const*)
**apply** (*subgoal-tac* (*λh0* . *tFold* (*λh0a t0* . *a5 h0* (*a6 u t0*)) *const*) = (*λ h0 t0* . *a5 arbitrary* (*a6 u arbitrary*)))
**prefer** *2*
**apply** (*subgoal-tac* (*λh0* . *tFold* (*λh0a t0* . *a5 h0* (*a6 u t0*)) *const*) = (*λh0* . *tFold* (*λ h0 t0* . *a5 arbitrary* (*a6 u arbitrary*)) *const*))
**prefer** *2*
**apply** (*rule ext*)
**apply** (*rule-tac x*=*const* **in** *fun-cong*)
**apply** (*rule-tac f*=*tFold* **in** *arg-cong*)
**apply** (*rule ext*)
**apply** (*rule ext*)
**apply** (*cut-tac c*=*c5* **and** *F*=%*x* . *a5 x* (*a6 u t0*) **and** *x*=*h0* **in** *LRunit-const*)
**apply** (*simp* (*no-asm-simp*))
**apply** (*cut-tac c*=*c5* **and** *F*=%*x* . *a5 arbitrary* (*a6 u x*) **and** *x*=*t0* **in** *LRunit-const*)
**apply** (*simp* (*no-asm-simp*))
**apply** (*simp* (*no-asm-simp*))
**apply** (*rotate-tac* −*1*, *erule trans*)
**apply** (*rule ext*)
**apply** (*rule ext*)
**apply** (*simp* (*no-asm-simp*))
**apply** (*rotate-tac* −*1*, *erule ssubst*)
**apply** (*simp* (*no-asm-simp*) *del*: *slimH2-def tFold-tMap*)
**apply** (*simp* (*no-asm-simp*) *only*: *tFold-spread1*)
**apply** (*subst foldr1-map1-LRunit*)
**apply** (*simp* (*no-asm-simp*))
**apply** *assumption*
**apply** (*subst foldr1-map1-LRunit*)
**apply** (*simp* (*no-asm-simp*))
**apply** *assumption*
**apply** (*thin-tac ALL t* . *tFold a5 c5* (*tMap* (*tFold a6 c6*) *t*) =
*tFold* (*λh t0*. *a5 h* (*tFold a6 c6 t0*)) *c5 t*)
**apply** (*subgoal-tac tFold* (*λh t0*. *a5 h* (*tFold a6 c6 t0*)) *c5* = (% *t* . *tFold a5 c5* (*tMap* (*tFold a6 c6*) *t*)))

**prefer** *2*
**apply** (*rule ext*)
**apply** (*subst tFold-tMap*)
**apply** *assumption*
**apply** (*rule refl*)
**apply** (*simp* (*no-asm-simp*) *del*: *tFold-tMap add*: *LRunit-left LRunit-right*)

2: *t* = *hConc t1 t2*

**apply** (*simp del*: *slimH2-def tFold-tMap*)
**done**

**lemma** *tFold-tFold-tFoldC-inverse2-wrapped*:
⟦ *assoc c1*; *assoc c2*; *assoc c3*; *assoc c4*; *assoc c5*; *assoc c6*;
⋀ *h ha hb t* . *w2* (*a4 hb* (*a5 ha* (*a6 h*))) = *w1* (*a1 h* (*a2 ha* (*a3 hb*)));
⋀ *x y* . *w2* (*c4 x y*) = *c7* (*w2 x*) (*w2 y*);
⋀ *x y* . *c7* (*w1 x*) (*w1 y*) = *w1* (*c1 x y*);
⋀ *x y* . *w2* (*c5 x y*) = *c8* (*w2 x*) (*w2 y*);
⋀ *h x y* . *c7* (*w1* (*a1 h x*)) (*w1* (*a1 h y*)) = *w1* (*a1 h* (*c2 x y*));
⋀ *h ha x y* .
*c7* (*w1* (*a1 h* (*a2 ha x*))) (*w1* (*a1 h* (*a2 ha y*))) =
*w1* (*a1 h* (*a2 ha* (*c3 x y*)));
(∗ *The next two are for tFold-tFold-vConc*: ∗)
⋀ *x y z* . *a4 x* (*c5 y z*) = *c5* (*a4 x y*) (*a4 x z*);
⋀ *x1 x2 y1 y2* . *c4* (*c5 x1 y1*) (*c5 x2 y2*) = *c5* (*c4 x1 x2*) (*c4 y1 y2*);
(∗ *units*: ∗)
⋀ *x* . *LRunit c8* (*w1* (*a1 u x*));
⋀ *x y* . *LRunit c5* (*a5 y* (*a6 u*));
⋀ *x y z* . *LRunit c4* (*a4 z* (*a5 y* (*a6 u*)));
(∗ *finishing 1.2*: ∗)
⋀ *h x1 x2* . *c5* (*a1 h x1*) (*a1 h x2*) = *a1 h* (*c2 x1 x2*)
⟧ ⟹
*w2* (*tFold a4 c4* (*tMap* (*tFold a5 c5 o* (*tMap* (*tFoldC a6 c6*))) (*inverse2 u t*))) =
*w1* (*tFold a1 c1* (*tMap* (*tFold a2 c2 o* (*tMap* (*tFoldC a3 c3*))) *t*))
**apply** (*unfold tFoldC-def*)
**apply** (*rule tFold-tFold-tFold-inverse2-wrapped*)
**apply** *assumption+*
**apply** (*simp-all* (*no-asm-simp*))
**apply** *simp-all*

Strange: Why doesn't the first *simp-all*, even without *no-asm-simp*, go through?

**done**

**lemma** *wrapped-tFold*:
⟦ *assoc c4*; *assoc c7*; ⋀ *x y* . *w2* (*c4 x y*) = *c7* (*w2 x*) (*w2 y*) ⟧ ⟹
*w2* (*tFold a4 c4 t*) = *tFold* (% *h t* . *w2* (*a4 h t*)) *c7 t*
**by** (*induct-tac t rule*: *T-induct*, *simp-all*)

**lemma** *wrapped-foldr1*:
⟦ *assoc c5*; *assoc c8*; ⋀ *x y* . *f* (*c5 x y*) = *c8* (*f x*) (*f y*) ⟧ ⟹

*f* (*foldr1 c5 xs*) = *foldr1 c8* (*map1 f xs*)
**by** (*induct-tac xs rule*: *neList-append-induct*, *simp-all*)

**lemma** *wrapped2-tFold*:
⟦ *assoc c5*; *assoc c7*;
⋀ *h x y* . *w2* (*a4 h* (*c5 x y*)) = *c7* (*w2* (*a4 h x*)) (*w2* (*a4 h y*)) ⟧ ⟹
(% *x* . *w2* (*a4 x* (*tFold a8 c5 t*))) = (% *x* . *tFold* (% *h t* . *w2* (*a4 x* (*a8 h t*)))
*c7 t*)
**apply** (*rule ext*)
**apply** (*rule wrapped-tFold* [*of c5 c7*])
**apply** *assumption*
**apply** *assumption*
**apply** (*simp-all* (*no-asm-simp*))
**done**

**lemma** *tFold-tFold-tFold-inverse2-gen*:
⟦ *assoc c1*; *assoc c2*; *assoc c3*; *assoc c4*; *assoc c5*; *assoc c6*; *assoc c7*;
⋀ *h ha hb t* . *w2* (*a4 hb* (*a5 ha* (*a6 h t*))) = *w1* (*a1 h* (*a2 ha* (*a3 hb t*)));
(∗ *finishing 2*, *and influencing the next rule*: ∗)
⋀ *x y* . *c7* (*w1 x*) (*w1 y*) = *w1* (*c1 x y*);
(∗ *finishing 1.1.2* ∗)
⋀ *h ha x y* .
*w1* (*c1* (*a1 h* (*a2 ha x*)) (*a1 h* (*a2 ha y*))) =
*w1* (*a1 h* (*a2 ha* (*c3 x y*)));
(∗ *The next few are for tFold-tFold-vConc-wrapped*: ∗)
⋀ *h x y* . *w2* (*a4 h* (*c5 x y*)) = *w2* (*c4* (*a4 h x*) (*a4 h y*));
⋀ *x y* . *w2* (*c4 x y*) = *c7* (*w2 x*) (*w2 y*);
⋀ *x1 x2 y1 y2* . *c7* (*w2* (*c4 x1 x2*)) (*w2* (*c4 y1 y2*)) =
*w2* (*c4* (*c4 x1 y1*) (*c4 x2 y2*));
(∗ *stronger*:
⋀ *x1 x2 y1 y2* . *c4* (*c4 x1 x2*) (*c4 y1 y2*) =
*c4* (*c4 x1 y1*) (*c4 x2 y2*);
∗)
(∗ *units*: ∗)
(∗
⋀ *x y* . *LRunit c5* (*a5 y* (*a6 u x*));
⋀ *h x y* . *LRunit c4* (*a4 h* (*a5 y* (*a6 u x*)));
∗)
⋀ *x* . *LRunit c7* (*w1* (*a1 u x*));
(∗ *finishing 1.2*: ∗)
⋀ *h x y* . *w1* (*c1* (*a1 h x*) (*a1 h y*)) = *w1* (*a1 h* (*c2 x y*))
⟧ ⟹
*w2* (*tFold a4 c4* (*tMap* (*tFold a5 c5 o* (*tMap* (*tFold a6 c6*))) (*inverse2 u t*))) =
*w1* (*tFold a1 c1* (*tMap* (*tFold a2 c2 o* (*tMap* (*tFold a3 c3*))) *t*))
**apply** (*induct-tac t rule*: *T-induct*)

1: *addH h t0*

**apply** (*induct-tac t0 rule*: *T-induct*)

1.1: *t0* = *addH ha t0a*

**apply** (*induct-tac t0a rule*: *T-induct*, *simp* (*no-asm-simp*))

1.1.2: *t0a* = *hConc t1 t2*

**apply** *simp*

1.2: *t0* = *hConc t1 t2*

**apply** (*simp del*: *slimH2-def tFold-tMap*)
**apply** (*fold invOp2-def*)
**apply** (*cut-tac u*=*u* **and** *h1.0*=*h* **and** *t*=*t1* **in** *regSkelOuter2-invOp2*)
**apply** (*cut-tac u*=*u* **and** *h1.0*=*h* **and** *t*=*t2* **in** *regSkelOuter2-invOp2*)
**apply** (*subst invOp2-hConc*)
  **apply** *assumption*
  **apply** *assumption*
**apply** (*subgoal-tac ALL t* . *tFold a5 c5* (*tMap* (*tFold a6 c6*) *t*) =
          *tFold* (λ*h t0*. *a5 h* (*tFold a6 c6 t0*)) *c5 t*)
  **prefer** *2*
  **apply** (*rule allI*, *simp* (*no-asm-simp*))
**apply** (*simp* (*no-asm-simp*) *del*: *slimH2-def tFold-tMap*)
**apply** (*subst tFold-tFold-vConc-wrapped* [*of c4 c5 c4 w2 a4 w2 a4 c7*])
  **apply** *assumption*
  **apply** *assumption*
  **apply** *assumption*
  **apply** (*simp* (*no-asm-simp*))
  **apply** (*simp* (*no-asm-simp*))
  **apply** (*simp* (*no-asm-simp*))
  **apply** (*simp* (*no-asm-simp*))
    **apply** (*rule conjI*, *rule refl*, *rule refl*)
  **apply** *assumption*
**apply** (*subst tFold-tFold-vConc-wrapped* [*of c4 c5 c4 w2 a4 w2 a4 c7*])
  **apply** *assumption*
  **apply** *assumption*
  **apply** *assumption*
  **apply** (*simp* (*no-asm-simp*))
  **apply** (*simp* (*no-asm-simp*))
  **apply** (*simp* (*no-asm-simp*))
  **apply** *assumption*
  **apply** (*simp* (*no-asm-simp*) *del*: *slimH2-def*)
**apply** (*simp* (*no-asm-simp*) *only*: *spread1-slimH2*)
**apply** (*simp* (*no-asm-simp*) *only*: *tMap-f-slimH3 tFold-slimH2 tFold-slimH1*)
**apply** (*subst tFold-tMap* [*of - -* (*tFold* (λ*h0 t0*. *a5 h0* (*tFold* (λ*h0*. *a6 u*) *const t0*)) *c5*)])
  **apply** *assumption*
**apply** (*subst tFold-tMap* [*of - -* (*tFold* (λ*h0 t0*. *a5 h0* (*tFold* (λ*h0*. *a6 u*) *const t0*)) *c5*)])
  **apply** *assumption*
**apply** (*subst wrapped-tFold* [*of c4 c7 w2* (λ*h t0*. *a4 h* (*tFold* (λ*h0 t0*. *a5 h0* (*tFold* (λ*h0*. *a6 u*) *const t0*)) *c5 t0*))])
  **apply** *assumption*
  **apply** *assumption*

**apply** (*simp* (*no-asm-simp*))
**apply** (*subst wrapped-tFold* [*of c4 c7 w2* (λ*h t0*. *a4 h* (*tFold* (λ*h0 t0*. *a5 h0* (*tFold* (λ*h0*. *a6 u*) *const t0*)) *c5 t0*))])
**apply** *assumption*
**apply** *assumption*
**apply** (*simp* (*no-asm-simp*))
**apply** (*subgoal-tac* (λ*h t*. *w2* (*a4 h* (*tFold* (λ*h0 t0*. *a5 h0* (*tFold* (λ*h0*. *a6 u*) *const t0*)) *c5 t*))) = (λ *h t* . *w1* (*a1 u arbitrary*)))
**prefer** *2*
**apply** (*rule ext*)
**apply** (*rule ext*)
**apply** (*cut-tac c4.0*=*c5* **and** *c7.0*=*c7* **and** *w2.0*=%*z* . *w2* (*a4 ha z*) **in** *wrapped-tFold*)
**apply** *assumption*
**apply** *assumption*
**apply** (*simp* (*no-asm-simp*))
**apply** (*rotate-tac* −*1*, *erule ssubst*)
**apply** (*simp* (*no-asm-simp*) *only*: *f-tFold-const*)
**apply** (*cut-tac c*=*c7* **and** *a*=(λ*h*. *tFold* (λ*h0 t0*. *w1* (*a1 u* (*a2 h* (*a3 ha t0*)))) *const*) **in** *tFold-LRunit*)
**apply** (*simp* (*no-asm-simp*))
**apply** (*simp* (*no-asm-simp*) *only*: *f-tFold-const*)
**apply** (*simp* (*no-asm-simp*) *only*: *tFold-const-const*)
**apply** *assumption*
**apply** (*rotate-tac* −*1*, *erule ssubst*)
**apply** (*subgoal-tac tFold* (λ*h0 t0*. *w1* (*a1 u* (*a2 arbitrary* (*a3 ha t0*)))) *const* = *tFold* (λ*h0 t0*. *w1* (*a1 u arbitrary*)) *const*)
**apply** (*rotate-tac* −*1*, *erule ssubst*)
**apply** (*simp* (*no-asm-simp*) *only*: *tFold-const-const*)
**apply** (*rule-tac x*=*const* **in** *fun-cong*)
**apply** (*rule-tac f*=*tFold* **in** *arg-cong*)
**apply** (*rule ext*)
**apply** (*rule ext*)
**apply** (*rule-tac c*=*c7* **and** *F*=%*x* . *w1* (*a1 u x*) **in** *LRunit-const*)
**apply** (*simp* (*no-asm-simp*))
**apply** (*rotate-tac* −*1*, *erule ssubst*)
**apply** (*simp* (*no-asm-simp*) *del*: *slimH2-def tFold-tMap*)
**apply** (*thin-tac ALL t* . *tFold a5 c5* (*tMap* (*tFold a6 c6*) *t*) =
*tFold* (λ*h t0*. *a5 h* (*tFold a6 c6 t0*)) *c5 t*)
**apply** (*subgoal-tac tFold* (λ*h t0*. *a5 h* (*tFold a6 c6 t0*)) *c5* = (% *t* . *tFold a5 c5* (*tMap* (*tFold a6 c6*) *t*)))
**prefer** *2*
**apply** (*rule ext*)
**apply** (*subst tFold-tMap*)
**apply** *assumption*
**apply** (*rule refl*)
**apply** (*simp* (*no-asm-simp*) *del*: *tFold-tMap add*: *LRunit-left LRunit-right*)

Now we expand some extra effort to be able to work with a single, wrapped-only unit assumption.

**apply** (*subgoal-tac w1* (*c1* (*a1 h* (*tFold a2 c2* (*tMap* (*tFold a3 c3*) *t1*)))
(*c1* (*a1 u arbitrary*)
(*c1* (*a1 u arbitrary*)
(*a1 h* (*tFold a2 c2* (*tMap* (*tFold a3 c3*) *t2*)))))) =
*c7* (*w1* (*a1 h* (*tFold a2 c2* (*tMap* (*tFold a3 c3*) *t1*))))
(*c7* (*w1* (*a1 u arbitrary*))
(*c7* (*w1* (*a1 u arbitrary*))
(*w1* (*a1 h* (*tFold a2 c2* (*tMap* (*tFold a3 c3*) *t2*)))))))
**prefer** *2*
**apply** (*simp* (*no-asm-simp*))
**apply** (*rotate-tac* −*1*, *erule trans*)
**apply** (*subst LRunit-left* [*of c7 w1* (*a1 u arbitrary*)])
**apply** (*simp* (*no-asm-simp*))
**apply** (*subst LRunit-left* [*of c7 w1* (*a1 u arbitrary*)])
**apply** (*simp* (*no-asm-simp*))
**apply** (*simp* (*no-asm-simp*))

2: *t* = *hConc t1 t2*

**apply** (*simp del*: *slimH2-def tFold-tMap*)
**done**

**lemma** *tFold-tFold-tFoldC-inverse2-gen*:
⟦ *assoc c1*; *assoc c2*; *assoc c3*; *assoc c4*; *assoc c5*; *assoc c6*; *assoc c7*;
⋀ *h ha hb* . *w2* (*a4 hb* (*a5 ha* (*a6 h*))) = *w1* (*a1 h* (*a2 ha* (*a3 hb*)));
(∗ *finishing 2*, *and influencing the next rule*: ∗)
⋀ *x y* . *c7* (*w1 x*) (*w1 y*) = *w1* (*c1 x y*);
(∗ *finishing 1.1.2* ∗)
⋀ *h ha x y* .
*w1* (*c1* (*a1 h* (*a2 ha x*)) (*a1 h* (*a2 ha y*))) =
*w1* (*a1 h* (*a2 ha* (*c3 x y*)));
(∗ *The next few are for tFold-tFold-vConc-wrapped*: ∗)
⋀ *h x y* . *w2* (*a4 h* (*c5 x y*)) = *w2* (*c4* (*a4 h x*) (*a4 h y*));
⋀ *x y* . *w2* (*c4 x y*) = *c7* (*w2 x*) (*w2 y*);
⋀ *x1 x2 y1 y2* . *c7* (*w2* (*c4 x1 x2*)) (*w2* (*c4 y1 y2*)) =
*w2* (*c4* (*c4 x1 y1*) (*c4 x2 y2*));
(∗ *stronger*:
⋀ *x1 x2 y1 y2* . *c4* (*c4 x1 x2*) (*c4 y1 y2*) =
*c4* (*c4 x1 y1*) (*c4 x2 y2*);
∗)
(∗ *units*: ∗)
(∗
⋀ *x y* . *LRunit c5* (*a5 y* (*a6 u*));
⋀ *h x y* . *LRunit c4* (*a4 h* (*a5 y* (*a6 u*)));
∗)
⋀ *x* . *LRunit c7* (*w1* (*a1 u x*));
(∗ *finishing 1.2*: ∗)
⋀ *h x y* . *w1* (*c1* (*a1 h x*) (*a1 h y*)) = *w1* (*a1 h* (*c2 x y*))
⟧ ⟹
*w2* (*tFold a4 c4* (*tMap* (*tFold a5 c5 o* (*tMap* (*tFoldC a6 c6*))) (*inverse2 u t*))) =

*w1* (*tFold a1 c1* (*tMap* (*tFold a2 c2 o* (*tMap* (*tFoldC a3 c3*))) *t*))
**apply** (*unfold tFoldC-def*)
**apply** (*rule tFold-tFold-tFold-inverse2-gen*)
**apply** *assumption+*
**apply** (*simp-all* (*no-asm-simp*))
**apply** *simp*
**apply** (*subgoal-tac c7* (*c7* (*w2 x1*) (*w2 x2*)) (*c7* (*w2 y1*) (*w2 y2*)) =
*c7* (*w2* (*c4 x1 x2*)) (*w2* (*c4 y1 y2*))))
**prefer** *2*
**apply** (*simp* (*no-asm-simp*))
**apply** *simp*
**done**

**lemma** *id-tFold-tFold-tFoldC-inverse2-gen*:
⟦ *assoc c1*; *assoc c2*; *assoc c3*; *assoc c5*; *assoc c6*;
⋀ *h ha hb* . *a4 hb* (*a5 ha* (*a6 h*)) = *a1 h* (*a2 ha* (*a3 hb*));
(∗ *finishing 1.1.2* ∗)
⋀ *h ha x y* .
*c1* (*a1 h* (*a2 ha x*)) (*a1 h* (*a2 ha y*)) = *a1 h* (*a2 ha* (*c3 x y*));
(∗ *The next few are for tFold-tFold-vConc-wrapped*: ∗)
⋀ *h x y* . *a4 h* (*c5 x y*) = *c1* (*a4 h x*) (*a4 h y*);
⋀ *x1 x2 y1 y2* . *c1* (*c1 x1 x2*) (*c1 y1 y2*) =
*c1* (*c1 x1 y1*) (*c1 x2 y2*);
(∗ *units*: ∗)
⋀ *x* . *LRunit c1* (*a1 u x*);
(∗ *finishing 1.2*: ∗)
⋀ *h x y* . *c1* (*a1 h x*) (*a1 h y*) = *a1 h* (*c2 x y*)
⟧ ⟹
*id* (*tFold a4 c1* (*tMap* (*tFold a5 c5 o* (*tMap* (*tFoldC a6 c6*))) (*inverse2 u t*))) =
*id* (*tFold a1 c1* (*tMap* (*tFold a2 c2 o* (*tMap* (*tFoldC a3 c3*))) *t*))
**apply** (*rule tFold-tFold-tFoldC-inverse2-gen*)
**apply** *assumption+*
**apply** (*simp-all* (*no-asm-simp*))
**apply** (*subgoal-tac a1 h* (*c2* (*a2 ha x*) (*a2 ha y*)) = *c1* (*a1 h* (*a2 ha x*)) (*a1 h* (*a2 ha y*)))
**prefer** *2*
**apply** *simp*
**apply** (*erule trans*)
**apply** (*rotate-tac* −*1*, *simp*)
**done**

**end**