

# Interval Temporal Logic on Natural Numbers

David Trachtenherz

April 11, 2026

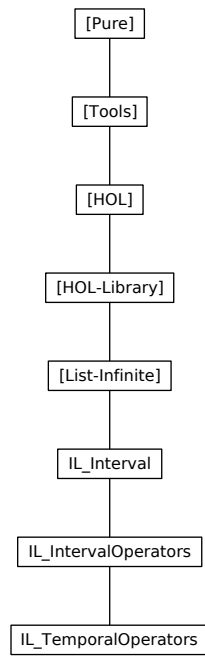
## Abstract

We introduce a theory of temporal logic operators using sets of natural numbers as time domain, formalized in a shallow embedding manner. The theory comprises special natural intervals (theory `IL_Interval`: open and closed intervals, continuous and modulo intervals, interval traversing results), operators for shifting intervals to left/right on the number axis as well as expanding/contracting intervals by constant factors (theory `IL_IntervalOperators.thy`), and ultimately definitions and results for unary and binary temporal operators on arbitrary natural sets (theory `IL_TemporalOperators`).

## Contents

<b>1</b>	<b>Intervals and operations for temporal logic declarations</b>	<b>2</b>
1.1	Time intervals – definitions and basic lemmata . . . . .	2
1.1.1	Definitions . . . . .	2
1.1.2	Membership in an interval . . . . .	3
1.1.3	Interval conversions . . . . .	6
1.1.4	Finiteness and emptiness of intervals . . . . .	7
1.1.5	<i>Min</i> and <i>Max</i> element of an interval . . . . .	8
1.2	Adding and subtracting constants to interval elements . . . . .	9
1.3	Relations between intervals . . . . .	13
1.3.1	Auxiliary lemmata . . . . .	13
1.3.2	Subset relation between intervals . . . . .	14
1.3.3	Equality of intervals . . . . .	23
1.3.4	Inequality of intervals . . . . .	24
1.4	Union and intersection of intervals . . . . .	25
1.5	Cutting intervals . . . . .	30
1.6	Cardinality of intervals . . . . .	41
1.7	Functions <i>inext</i> and <i>iprev</i> with intervals . . . . .	43
1.7.1	Mirroring of intervals . . . . .	47
1.7.2	Functions <i>inext-nth</i> and <i>iprev-nth</i> on intervals . . . . .	48
1.8	Induction with intervals . . . . .	50

<b>2</b>	<b>Arithmetic operators on natural intervals</b>	<b>51</b>
2.1	Arithmetic operations with intervals . . . . .	51
2.1.1	Addition of and multiplication by constants . . . . .	51
2.1.2	Some conversions between intervals using constant addition and multiplication . . . . .	57
2.1.3	Subtraction of constants . . . . .	58
2.1.4	Subtraction of intervals from constants . . . . .	65
2.1.5	Division of intervals by constants . . . . .	71
2.2	Interval cut operators with arithmetic interval operators . . .	83
2.3	<i>inext</i> and <i>iprev</i> with interval operators . . . . .	88
2.4	Cardinality of intervals with interval operators . . . . .	93
2.5	Results about sets of intervals . . . . .	103
2.5.1	Set of intervals without and with empty interval . . .	103
2.5.2	Interval sets are closed under cutting . . . . .	109
2.5.3	Interval sets are closed under addition and multiplication	110
2.5.4	Interval sets are closed with certain conditions under subtraction . . . . .	111
2.5.5	Interval sets are not closed under division . . . . .	112
2.5.6	Sets of intervals closed under division . . . . .	112
<b>3</b>	<b>Temporal logic operators on natural intervals</b>	<b>119</b>
3.1	Basic definitions . . . . .	120
3.2	Basic lemmata for temporal operators . . . . .	124
3.2.1	Intro/elim rules . . . . .	124
3.2.2	Rewrite rules for trivial simplification . . . . .	125
3.2.3	Empty sets and singletons . . . . .	133
3.2.4	Conversions between temporal operators . . . . .	134
3.2.5	Some implication results . . . . .	139
3.2.6	Congruence rules for temporal operators' predicates .	140
3.2.7	Temporal operators with set unions/intersections and subsets . . . . .	142
3.3	Further results for temporal operators . . . . .	143
3.4	Temporal operators and arithmetic interval operators . . . .	147



# 1 Intervals and operations for temporal logic declarations

```

theory IL-Interval
imports
  List-Infinite.InfiniteSet2
  List-Infinite.SetIntervalStep
begin

```

## 1.1 Time intervals – definitions and basic lemmata

### 1.1.1 Definitions

```

type-synonym Time = nat

```

```

type-synonym iT = Time set

```

Infinite interval starting at some natural  $n$ .

**definition**

```

iFROM :: Time  $\Rightarrow$  iT ( $\langle$ [ $\dots$ ] $\rangle$ )

```

**where**

```

[ $n\dots$ ]  $\equiv$  { $n\dots$ }

```

Finite interval starting at  $0$  and ending at some natural  $n$ .

**definition**

```

iTILL :: Time  $\Rightarrow$  iT ( $\langle$ [ $\dots$ ] $\rangle$ )

```

**where**

```

[ $\dots n$ ]  $\equiv$  { $\dots n$ }

```

Finite bounded interval containing the naturals between  $n$  and  $n + d$ .  $d$  denotes the difference between left and right interval bound. The number of elements is  $d + 1$  so that an empty interval cannot be defined.

**definition**

```

iIN :: Time  $\Rightarrow$  nat  $\Rightarrow$  iT ( $\langle$ [ $\dots$ , $\dots$ ] $\rangle$ )

```

**where**

```

[ $n\dots, d$ ]  $\equiv$  { $n\dots, n+d$ }

```

Infinite modulo interval containing all naturals having the same division remainder modulo  $m$  as  $r$ , and beginning at  $n$ .

**definition**

```

iMOD :: Time  $\Rightarrow$  nat  $\Rightarrow$  iT ( $\langle$ [ $\dots$ ,  $\text{mod } \dots$ ] $\rangle$ )

```

**where**

```

[ $r, \text{mod } m$ ]  $\equiv$  {  $x$ .  $x \text{ mod } m = r \text{ mod } m \wedge r \leq x$  }

```

Finite bounded modulo interval containing all naturals having the same division remainder modulo  $m$  as  $r$ , beginning at  $n$ , and ending after  $c$  cycles

at  $r + m * c$ . The number of elements is  $c + 1$  so that an empty interval cannot be defined.

**definition**

$iMODb :: Time \Rightarrow nat \Rightarrow nat \Rightarrow iT (\langle [-, mod -, -] \rangle)$

**where**

$[r, mod\ m, c] \equiv \{ x. x\ mod\ m = r\ mod\ m \wedge r \leq x \wedge x \leq r + m * c \}$

**1.1.2 Membership in an interval**

**lemmas**  $iT-defs = iFROM-def\ iTILL-def\ iIN-def\ iMOD-def\ iMODb-def$

**lemma**  $iFROM-iff: x \in [n..] = (n \leq x)$

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

**lemma**  $iTILL-iff: x \in [..n] = (x \leq n)$

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

**lemma**  $iIN-iff: x \in [n..,d] = (n \leq x \wedge x \leq n + d)$

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

**lemma**  $iMOD-iff: x \in [r, mod\ m] = (x\ mod\ m = r\ mod\ m \wedge r \leq x)$

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

**lemma**  $iMODb-iff: x \in [r, mod\ m, c] =$

$(x\ mod\ m = r\ mod\ m \wedge r \leq x \wedge x \leq r + m * c)$

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

**lemma**  $iFROM-D: x \in [n..] \Longrightarrow (n \leq x)$

**by** (*rule iFROM-iff[THEN iffD1]*)

**lemma**  $iTILL-D: x \in [..n] \Longrightarrow (x \leq n)$

**by** (*rule iTILL-iff[THEN iffD1]*)

**corollary**  $iIN-geD: x \in [n..,d] \Longrightarrow n \leq x$

**by** (*simp add: iIN-iff*)

**corollary**  $iIN-leD: x \in [n..,d] \Longrightarrow x \leq n + d$

**by** (*simp add: iIN-iff*)

**corollary**  $iMOD-modD: x \in [r, mod\ m] \Longrightarrow x\ mod\ m = r\ mod\ m$

**by** (*simp add: iMOD-iff*)

**corollary**  $iMOD-geD: x \in [r, mod\ m] \Longrightarrow r \leq x$

**by** (*simp add: iMOD-iff*)

**corollary**  $iMODb-modD: x \in [r, mod\ m, c] \Longrightarrow x\ mod\ m = r\ mod\ m$

**by** (*simp add: iMODb-iff*)

**corollary**  $iMODb-geD: x \in [r, mod\ m, c] \Longrightarrow r \leq x$

**by** (*simp add: iMODb-iff*)

**corollary**  $iMODb-leD: x \in [r, mod\ m, c] \Longrightarrow x \leq r + m * c$

**by** (*simp add: iMODb-iff*)

**lemmas**  $iT-iff = iFROM-iff\ iTILL-iff\ iIN-iff\ iMOD-iff\ iMODb-iff$

**lemmas**  $iT-drule =$

$iFROM-D$

$iTILL-D$

$iIN-geD\ iIN-leD$

$iMOD-modD\ iMOD-geD$

*iMODb-modD iMODb-geD iMODb-leD*

**lemma**

*iFROM-I* [intro]:  $n \leq x \implies x \in [n..]$  **and**  
*iTILL-I* [intro]:  $x \leq n \implies x \in [..n]$  **and**  
*iIN-I* [intro]:  $n \leq x \implies x \leq n + d \implies x \in [n..,d]$  **and**  
*iMOD-I* [intro]:  $x \bmod m = r \bmod m \implies r \leq x \implies x \in [r, \bmod m]$  **and**  
*iMODb-I* [intro]:  $x \bmod m = r \bmod m \implies r \leq x \implies x \leq r + m * c \implies x \in [r, \bmod m, c]$   
**by** (*simp add: iT-iff*)+

**lemma**

*iFROM-E* [elim]:  $x \in [n..] \implies (n \leq x \implies P) \implies P$  **and**  
*iTILL-E* [elim]:  $x \in [..n] \implies (x \leq n \implies P) \implies P$  **and**  
*iIN-E* [elim]:  $x \in [n..,d] \implies (n \leq x \implies x \leq n + d \implies P) \implies P$  **and**  
*iMOD-E* [elim]:  $x \in [r, \bmod m] \implies (x \bmod m = r \bmod m \implies r \leq x \implies P) \implies P$  **and**  
*iMODb-E* [elim]:  $x \in [r, \bmod m, c] \implies (x \bmod m = r \bmod m \implies r \leq x \implies x \leq r + m * c \implies P) \implies P$   
**by** (*simp add: iT-iff*)+

**lemma** *iIN-Suc-insert-conv*:

*insert (Suc (n + d)) [n..,d] = [n..,Suc d]*  
**by** (*fastforce simp: iIN-iff*)

**lemma** *iTILL-Suc-insert-conv*: *insert (Suc n) [..n] = [..Suc n]*

**by** (*fastforce simp: iIN-Suc-insert-conv[of 0 n]*)

**lemma** *iMODb-Suc-insert-conv*:

*insert (r + m \* Suc c) [r, mod m, c] = [r, mod m, Suc c]*  
**apply** (*rule set-eqI*)  
**apply** (*simp add: iMODb-iff add.commute[of - r]*)  
**apply** (*simp add: add.commute[of m]*)  
**apply** (*simp add: add.assoc[symmetric]*)  
**apply** (*rule iffI*)  
**apply** *fastforce*  
**apply** (*elim conjE*)  
**apply** (*drule-tac x=x in order-le-less[THEN iffD1, rule-format]*)  
**apply** (*erule disjE*)  
**apply** (*frule less-mod-eq-imp-add-divisor-le[where m=m], simp*)  
**apply** (*drule add-le-imp-le-right*)  
**apply** *simp*  
**apply** *simp*  
**done**

**lemma** *iFROM-pred-insert-conv*:  $\text{insert } (n - \text{Suc } 0) [n\dots] = [n - \text{Suc } 0\dots]$   
**by** (*fastforce simp: iFROM-iff*)

**lemma** *iIN-pred-insert-conv*:  
 $0 < n \implies \text{insert } (n - \text{Suc } 0) [n\dots, d] = [n - \text{Suc } 0\dots, \text{Suc } d]$   
**by** (*fastforce simp: iIN-iff*)

**lemma** *iMOD-pred-insert-conv*:  
 $m \leq r \implies \text{insert } (r - m) [r, \text{mod } m] = [r - m, \text{mod } m]$   
**apply** (*case-tac m = 0*)  
**apply** (*simp add: iMOD-iff insert-absorb*)  
**apply** *simp*  
**apply** (*rule set-eqI*)  
**apply** (*simp add: iMOD-iff mod-diff-self2*)  
**apply** (*rule iffI*)  
**apply** (*erule disjE*)  
**apply** (*simp add: mod-diff-self2*)  
**apply** (*simp add: le-imp-diff-le*)  
**apply** (*erule conjE*)  
**apply** (*drule order-le-less[THEN iffD1, of r-m], erule disjE*)  
**prefer** 2  
**apply** *simp*  
**apply** (*frule order-less-le-trans[of - m r], assumption*)  
**apply** (*drule less-mod-eq-imp-add-divisor-le[of r-m - m]*)  
**apply** (*simp add: mod-diff-self2*)  
**apply** *simp*  
**done**

**lemma** *iMODb-pred-insert-conv*:  
 $m \leq r \implies \text{insert } (r - m) [r, \text{mod } m, c] = [r - m, \text{mod } m, \text{Suc } c]$   
**apply** (*rule set-eqI*)  
**apply** (*frule iMOD-pred-insert-conv*)  
**apply** (*drule-tac f= $\lambda s. x \in s$  in arg-cong*)  
**apply** (*force simp: iMOD-iff iMODb-iff*)  
**done**

**lemma** *iFROM-Suc-pred-insert-conv*:  $\text{insert } n [\text{Suc } n\dots] = [n\dots]$   
**by** (*insert iFROM-pred-insert-conv[of Suc n], simp*)  
**lemma** *iIN-Suc-pred-insert-conv*:  $\text{insert } n [\text{Suc } n\dots, d] = [n\dots, \text{Suc } d]$   
**by** (*insert iIN-pred-insert-conv[of Suc n], simp*)  
**lemma** *iMOD-Suc-pred-insert-conv*:  $\text{insert } r [r + m, \text{mod } m] = [r, \text{mod } m]$   
**by** (*insert iMOD-pred-insert-conv[of m r + m], simp*)  
**lemma** *iMODb-Suc-pred-insert-conv*:  $\text{insert } r [r + m, \text{mod } m, c] = [r, \text{mod } m, \text{Suc } c]$   
**by** (*insert iMODb-pred-insert-conv[of m r + m], simp*)

**lemmas** *iT-Suc-insert* =  
*iIN-Suc-insert-conv*  
*iTILL-Suc-insert-conv*

*iMODb-Suc-insert-conv*  
**lemmas** *iT-pred-insert* =  
*iFROM-pred-insert-conv*  
*iIN-pred-insert-conv*  
*iMOD-pred-insert-conv*  
*iMODb-pred-insert-conv*  
**lemmas** *iT-Suc-pred-insert* =  
*iFROM-Suc-pred-insert-conv*  
*iIN-Suc-pred-insert-conv*  
*iMOD-Suc-pred-insert-conv*  
*iMODb-Suc-pred-insert-conv*

**lemma** *iMOD-mem-diff*:  $\llbracket a \in [r, \text{mod } m]; b \in [r, \text{mod } m] \rrbracket \implies (a - b) \text{ mod } m = 0$   
**by** (*simp add: iMOD-iff mod-eq-imp-diff-mod-0*)  
**lemma** *iMODb-mem-diff*:  $\llbracket a \in [r, \text{mod } m, c]; b \in [r, \text{mod } m, c] \rrbracket \implies (a - b) \text{ mod } m = 0$   
**by** (*simp add: iMODb-iff mod-eq-imp-diff-mod-0*)

### 1.1.3 Interval conversions

**lemma** *iIN-0-iTILL-conv*:  $[0..n] = [\dots n]$   
**by** (*simp add: iTILL-def iIN-def atLeastAtLeastAtMost-0-conv*)  
**lemma** *iIN-iTILL-iTILL-conv*:  $0 < n \implies [n..d] = [\dots n+d] - [\dots n - \text{Suc } 0]$   
**by** (*fastforce simp: iTILL-iff iIN-iff*)  
**lemma** *iIN-iFROM-iTILL-conv*:  $[n..d] = [n..] \cap [\dots n+d]$   
**by** (*simp add: iT-defs atLeastAtMost-def*)  
**lemma** *iMODb-iMOD-iTILL-conv*:  $[r, \text{mod } m, c] = [r, \text{mod } m] \cap [\dots r+m*c]$   
**by** (*force simp: iT-defs set-interval-defs*)  
**lemma** *iMODb-iMOD-iIN-conv*:  $[r, \text{mod } m, c] = [r, \text{mod } m] \cap [r..m*c]$   
**by** (*force simp: iT-defs set-interval-defs*)

**lemma** *iFROM-iTILL-iIN-conv*:  $n \leq n' \implies [n..] \cap [\dots n'] = [n.., n'-n]$   
**by** (*simp add: iT-defs atLeastAtMost-def*)

**lemma** *iMOD-iTILL-iMODb-conv*:  
 $r \leq n \implies [r, \text{mod } m] \cap [\dots n] = [r, \text{mod } m, (n - r) \text{ div } m]$   
**apply** (*rule set-eqI*)  
**apply** (*simp add: iT-iff minus-mod-eq-mult-div [symmetric]*)  
**apply** (*rule iffI*)  
**apply** *clarify*  
**apply** (*frule-tac x=x and y=n and m=m in le-imp-sub-mod-le*)  
**apply** (*simp add: mod-diff-right-eq*)  
**apply** *fastforce*  
**done**

**lemma** *iMOD-iIN-iMODb-conv*:  
 $[r, \text{mod } m] \cap [r..d] = [r, \text{mod } m, d \text{ div } m]$   
**apply** (*case-tac r = 0*)  
**apply** (*simp add: iIN-0-iTILL-conv iMOD-iTILL-iMODb-conv*)

**apply** (*simp add: iIN-iTILL-iTILL-conv Diff-Int-distrib iMOD-iTILL-iMODb-conv  
diff-add-inverse*)  
**apply** (*rule subst[of {} -  $\lambda t. \forall x.(x - t) = x$ , THEN spec]*)  
**prefer** 2  
**apply** *simp*  
**apply** (*rule sym*)  
**apply** (*fastforce simp: disjoint-iff-not-equal iMOD-iff iTILL-iff*)  
**done**

**lemma** *iFROM-0:  $[0..] = UNIV$*   
**by** (*simp add: iFROM-def*)

**lemma** *iTILL-0:  $[\dots 0] = \{0\}$*   
**by** (*simp add: iTILL-def*)

**lemma** *iIN-0:  $[n.., 0] = \{n\}$*   
**by** (*simp add: iIN-def*)

**lemma** *iMOD-0:  $[r, \text{mod } 0] = [r.., 0]$*   
**by** (*fastforce simp: iIN-0 iMOD-def*)

**lemma** *iMODb-mod-0:  $[r, \text{mod } 0, c] = [r.., 0]$*   
**by** (*fastforce simp: iMODb-def iIN-0*)

**lemma** *iMODb-0:  $[r, \text{mod } m, 0] = [r.., 0]$*   
**by** (*fastforce simp: iMODb-def iIN-0 set-eq-iff*)

**lemmas** *iT-0 =*  
*iFROM-0*  
*iTILL-0*  
*iIN-0*  
*iMOD-0*  
*iMODb-mod-0*  
*iMODb-0*

**lemma** *iMOD-1:  $[r, \text{mod } \text{Suc } 0] = [r..]$*   
**by** (*fastforce simp: iFROM-iff*)

**lemma** *iMODb-mod-1:  $[r, \text{mod } \text{Suc } 0, c] = [r.., c]$*   
**by** (*fastforce simp: iT-iff*)

#### 1.1.4 Finiteness and emptiness of intervals

**lemma**  
*iFROM-not-empty:  $[n..] \neq \{\}$  and*  
*iTILL-not-empty:  $[\dots n] \neq \{\}$  and*  
*iIN-not-empty:  $[n.., d] \neq \{\}$  and*  
*iMOD-not-empty:  $[r, \text{mod } m] \neq \{\}$  and*  
*iMODb-not-empty:  $[r, \text{mod } m, c] \neq \{\}$*

**by** (*fastforce simp: iT-iff*)+

**lemmas** *iT-not-empty* =  
*iFROM-not-empty*  
*iTILL-not-empty*  
*iIN-not-empty*  
*iMOD-not-empty*  
*iMODb-not-empty*

**lemma**  
*iTILL-finite: finite [..n]* **and**  
*iIN-finite: finite [n..,d]* **and**  
*iMODb-finite: finite [r, mod m, c]* **and**  
*iMOD-0-finite: finite [r, mod 0]*  
**by** (*simp add: iT-defs*)+

**lemma** *iFROM-infinite: infinite [n..]*  
**by** (*simp add: iT-defs infinite-atLeast*)

**lemma** *iMOD-infinite: 0 < m  $\implies$  infinite [r, mod m]*  
**apply** (*rule infinite-nat-iff-asc-chain[THEN iffD2]*)  
**apply** (*rule iT-not-empty*)  
**apply** (*rule ballI, rename-tac n*)  
**apply** (*rule-tac x=n+m in beqI, simp*)  
**apply** (*simp add: iMOD-iff*)  
**done**

**lemmas** *iT-finite* =  
*iTILL-finite*  
*iIN-finite*  
*iMODb-finite iMOD-0-finite*

**lemmas** *iT-infinite* =  
*iFROM-infinite*  
*iMOD-infinite*

### 1.1.5 *Min and Max* element of an interval

**lemma**  
*iTILL-Min: iMin [..n] = 0* **and**  
*iFROM-Min: iMin [n..] = n* **and**  
*iIN-Min: iMin [n..,d] = n* **and**  
*iMOD-Min: iMin [r, mod m] = r* **and**  
*iMODb-Min: iMin [r, mod m, c] = r*  
**by** (*rule iMin-equality, (simp add: iT-iff)*)+

**lemmas** *iT-Min* =  
*iIN-Min*  
*iTILL-Min*

*iFROM-Min*  
*iMOD-Min*  
*iMODb-Min*

**lemma**

*iTILL-Max*:  $\text{Max} [\dots n] = n$  **and**  
*iIN-Max*:  $\text{Max} [n\dots, d] = n+d$  **and**  
*iMODb-Max*:  $\text{Max} [r, \text{mod } m, c] = r + m * c$  **and**  
*iMOD-0-Max*:  $\text{Max} [r, \text{mod } 0] = r$

**by** (rule *Max-equality*, (*simp add: iT-iff iT-finite*))+

**lemmas** *iT-Max* =

*iTILL-Max*  
*iIN-Max*  
*iMODb-Max*  
*iMOD-0-Max*

**lemma**

*iTILL-iMax*:  $i\text{Max} [\dots n] = \text{enat } n$  **and**  
*iIN-iMax*:  $i\text{Max} [n\dots, d] = \text{enat } (n+d)$  **and**  
*iMODb-iMax*:  $i\text{Max} [r, \text{mod } m, c] = \text{enat } (r + m * c)$  **and**  
*iMOD-0-iMax*:  $i\text{Max} [r, \text{mod } 0] = \text{enat } r$  **and**  
*iFROM-iMax*:  $i\text{Max} [n\dots] = \infty$  **and**  
*iMOD-iMax*:  $0 < m \implies i\text{Max} [r, \text{mod } m] = \infty$

**by** (*simp add: iMax-def iT-finite iT-infinite iT-Max*)+

**lemmas** *iT-iMax* =

*iTILL-iMax*  
*iIN-iMax*  
*iMODb-iMax*  
*iMOD-0-iMax*  
*iFROM-iMax*  
*iMOD-iMax*

## 1.2 Adding and subtracting constants to interval elements

**lemma**

*iFROM-plus*:  $x \in [n\dots] \implies x + k \in [n\dots]$  **and**  
*iFROM-Suc*:  $x \in [n\dots] \implies \text{Suc } x \in [n\dots]$  **and**  
*iFROM-minus*:  $\llbracket x \in [n\dots]; k \leq x - n \rrbracket \implies x - k \in [n\dots]$  **and**  
*iFROM-pred*:  $n < x \implies x - \text{Suc } 0 \in [n\dots]$

**by** (*simp add: iFROM-iff*)+

**lemma**

*iTILL-plus*:  $\llbracket x \in [\dots n]; k \leq n - x \rrbracket \implies x + k \in [\dots n]$  **and**  
*iTILL-Suc*:  $x < n \implies \text{Suc } x \in [\dots n]$  **and**  
*iTILL-minus*:  $x \in [\dots n] \implies x - k \in [\dots n]$  **and**  
*iTILL-pred*:  $x \in [\dots n] \implies x - \text{Suc } 0 \in [\dots n]$

**by** (*simp add: iTILL-iff*)+

**lemma** *iIN-plus*:  $\llbracket x \in [n..d]; k \leq n + d - x \rrbracket \implies x + k \in [n..d]$   
**by** (*fastforce simp: iIN-iff*)

**lemma** *iIN-Suc*:  $\llbracket x \in [n..d]; x < n + d \rrbracket \implies \text{Suc } x \in [n..d]$   
**by** (*simp add: iIN-iff*)

**lemma** *iIN-minus*:  $\llbracket x \in [n..d]; k \leq x - n \rrbracket \implies x - k \in [n..d]$   
**by** (*fastforce simp: iIN-iff*)

**lemma** *iIN-pred*:  $\llbracket x \in [n..d]; n < x \rrbracket \implies x - \text{Suc } 0 \in [n..d]$   
**by** (*fastforce simp: iIN-iff*)

**lemma** *iMOD-plus-divisor-mult*:  $x \in [r, \text{mod } m] \implies x + k * m \in [r, \text{mod } m]$   
**by** (*simp add: iMOD-def*)

**corollary** *iMOD-plus-divisor*:  $x \in [r, \text{mod } m] \implies x + m \in [r, \text{mod } m]$   
**by** (*simp add: iMOD-def*)

**lemma** *iMOD-minus-divisor-mult*:  
 $\llbracket x \in [r, \text{mod } m]; k * m \leq x - r \rrbracket \implies x - k * m \in [r, \text{mod } m]$   
**by** (*fastforce simp: iMOD-def mod-diff-mult-self1*)

**corollary** *iMOD-minus-divisor-mult2*:  
 $\llbracket x \in [r, \text{mod } m]; k \leq (x - r) \text{ div } m \rrbracket \implies x - k * m \in [r, \text{mod } m]$   
**apply** (*rule iMOD-minus-divisor-mult, assumption*)  
**apply** (*clarsimp simp: iMOD-iff*)  
**apply** (*drule mult-le-mono1 [of - - m]*)  
**apply** (*simp add: mod-0-div-mult-cancel [THEN iffD1, OF mod-eq-imp-diff-mod-0]*)  
**done**

**corollary** *iMOD-minus-divisor*:  
 $\llbracket x \in [r, \text{mod } m]; m + r \leq x \rrbracket \implies x - m \in [r, \text{mod } m]$   
**apply** (*frule iMOD-geD*)  
**apply** (*insert iMOD-minus-divisor-mult [of x r m 1]*)  
**apply** *simp*  
**done**

**lemma** *iMOD-plus*:  
 $x \in [r, \text{mod } m] \implies (x + k \in [r, \text{mod } m]) = (k \text{ mod } m = 0)$   
**apply** *safe*  
**apply** (*drule iMOD-modD*)  
**apply** (*rule mod-add-eq-imp-mod-0 [of x, THEN iffD1]*)  
**apply** *simp*  
**apply** (*erule dvdE*)  
**apply** (*simp add: mult.commute iMOD-plus-divisor-mult*)  
**done**

**corollary** *iMOD-Suc*:  
 $x \in [r, \text{mod } m] \implies (\text{Suc } x \in [r, \text{mod } m]) = (m = \text{Suc } 0)$

**apply** (*simp add: iMOD-iff, safe*)  
**apply** (*simp add: mod-Suc, split if-split-asm*)  
**apply** *simp+*  
**done**

**lemma** *iMOD-minus:*

$\llbracket x \in [r, \text{mod } m]; k \leq x - r \rrbracket \implies (x - k \in [r, \text{mod } m]) = (k \text{ mod } m = 0)$   
**apply** *safe*  
**apply** (*clarsimp simp: iMOD-iff*)  
**apply** (*rule mod-add-eq-imp-mod-0[of x - k k, THEN iffD1]*)  
**apply** *simp*  
**apply** (*erule dvdE*)  
**apply** (*simp add: mult.commute iMOD-minus-divisor-mult*)  
**done**

**corollary** *iMOD-pred:*

$\llbracket x \in [r, \text{mod } m]; r < x \rrbracket \implies (x - \text{Suc } 0 \in [r, \text{mod } m]) = (m = \text{Suc } 0)$   
**apply** *safe*  
**apply** (*simp add: iMOD-Suc[of x - Suc 0 r, THEN iffD1]*)  
**apply** (*simp add: iMOD-iff*)  
**done**

**lemma** *iMODb-plus-divisor-mult:*

$\llbracket x \in [r, \text{mod } m, c]; k * m \leq r + m * c - x \rrbracket \implies x + k * m \in [r, \text{mod } m, c]$   
**by** (*fastforce simp: iMODb-def*)

**lemma** *iMODb-plus-divisor-mult2:*

$\llbracket x \in [r, \text{mod } m, c]; k \leq c - (x - r) \text{ div } m \rrbracket \implies$   
 $x + k * m \in [r, \text{mod } m, c]$   
**apply** (*rule iMODb-plus-divisor-mult, assumption*)  
**apply** (*clarsimp simp: iMODb-iff*)  
**apply** (*drule mult-le-mono1[of - - m]*)  
**apply** (*simp add: diff-mult-distrib*  
*mod-0-div-mult-cancel[THEN iffD1, OF mod-eq-imp-diff-mod-0]*  
*add.commute[of r] mult.commute[of c]*)  
**done**

**lemma** *iMODb-plus-divisor:*

$\llbracket x \in [r, \text{mod } m, c]; x < r + m * c \rrbracket \implies x + m \in [r, \text{mod } m, c]$   
**by** (*simp add: iMODb-iff less-mod-eq-imp-add-divisor-le*)

**lemma** *iMODb-minus-divisor-mult:*

$\llbracket x \in [r, \text{mod } m, c]; r + k * m \leq x \rrbracket \implies x - k * m \in [r, \text{mod } m, c]$   
**by** (*fastforce simp: iMODb-def mod-diff-mult-self1*)

**lemma** *iMODb-plus:*

$\llbracket x \in [r, \text{mod } m, c]; k \leq r + m * c - x \rrbracket \implies$   
 $(x + k \in [r, \text{mod } m, c]) = (k \text{ mod } m = 0)$   
**apply** *safe*  
**apply** (*rule mod-add-eq-imp-mod-0[of x, THEN iffD1]*)

**apply** (*simp add: iT-iff*)  
**apply** *fastforce*  
**done**

**corollary** *iMODb-Suc*:

$\llbracket x \in [r, \text{mod } m, c]; x < r + m * c \rrbracket \implies$   
 $(\text{Suc } x \in [r, \text{mod } m, c]) = (m = \text{Suc } 0)$

**apply** (*rule iffI*)  
**apply** (*simp add: iMODb-iMOD-iTILL-conv iMOD-Suc*)  
**apply** (*simp add: iMODb-iMOD-iTILL-conv iMOD-1 iFROM-Suc iTILL-Suc*)  
**done**

**lemma** *iMODb-minus*:

$\llbracket x \in [r, \text{mod } m, c]; k \leq x - r \rrbracket \implies$   
 $(x - k \in [r, \text{mod } m, c]) = (k \text{ mod } m = 0)$

**apply** (*rule iffI*)  
**apply** (*simp add: iMODb-iMOD-iTILL-conv iMOD-minus*)  
**apply** (*simp add: iMODb-iMOD-iTILL-conv iMOD-minus iTILL-minus*)  
**done**

**corollary** *iMODb-pred*:

$\llbracket x \in [r, \text{mod } m, c]; r < x \rrbracket \implies$   
 $(x - \text{Suc } 0 \in [r, \text{mod } m, c]) = (m = \text{Suc } 0)$

**apply** (*rule iffI*)  
**apply** (*subgoal-tac x \in [r, mod m] \wedge x - Suc 0 \in [r, mod m]*)  
**prefer** 2  
**apply** (*simp add: iT-iff*)  
**apply** (*clarsimp simp: iMOD-pred*)  
**apply** (*fastforce simp add: iMODb-iff*)  
**done**

**lemmas** *iFROM-plus-minus =*

*iFROM-plus*  
*iFROM-Suc*  
*iFROM-minus*  
*iFROM-pred*

**lemmas** *iTILL-plus-minus =*

*iTILL-plus*  
*iTILL-Suc*  
*iTILL-minus*  
*iTILL-pred*

**lemmas** *iIN-plus-minus =*

*iIN-plus*  
*iIN-Suc*  
*iTILL-minus*  
*iIN-pred*

**lemmas** *iMOD-plus-minus-divisor* =  
*iMOD-plus-divisor-mult*  
*iMOD-plus-divisor*  
*iMOD-minus-divisor-mult*  
*iMOD-minus-divisor-mult2*  
*iMOD-minus-divisor*

**lemmas** *iMOD-plus-minus* =  
*iMOD-plus*  
*iMOD-Suc*  
*iMOD-minus*  
*iMOD-pred*

**lemmas** *iMODb-plus-minus-divisor* =  
*iMODb-plus-divisor-mult*  
*iMODb-plus-divisor-mult2*  
*iMODb-plus-divisor*  
*iMODb-minus-divisor-mult*

**lemmas** *iMODb-plus-minus* =  
*iMODb-plus*  
*iMODb-Suc*  
*iMODb-minus*  
*iMODb-pred*

**lemmas** *iT-plus-minus* =  
*iFROM-plus-minus*  
*iTILL-plus-minus*  
*iIN-plus-minus*  
*iMOD-plus-minus-divisor*  
*iMOD-plus-minus*  
*iMODb-plus-minus-divisor*  
*iMODb-plus-minus*

### 1.3 Relations between intervals

#### 1.3.1 Auxiliary lemmata

**lemma** *Suc-in-imp-not-subset-iMOD*:  
 $\llbracket n \in S; \text{Suc } n \in S; m \neq \text{Suc } 0 \rrbracket \implies \neg S \subseteq [r, \text{mod } m]$   
**by** (*blast intro: iMOD-Suc[THEN iffD1]*)

**corollary** *Suc-in-imp-neq-iMOD*:  
 $\llbracket n \in S; \text{Suc } n \in S; m \neq \text{Suc } 0 \rrbracket \implies S \neq [r, \text{mod } m]$   
**by** (*blast dest: Suc-in-imp-not-subset-iMOD*)

**lemma** *Suc-in-imp-not-subset-iMODb*:  
 $\llbracket n \in S; \text{Suc } n \in S; m \neq \text{Suc } 0 \rrbracket \implies \neg S \subseteq [r, \text{mod } m, c]$   
**apply** (*rule ccontr, simp*)  
**apply** (*frule subsetD[of - - n], assumption*)

**apply** (*drule subsetD*[of - - *Suc n*], *assumption*)  
**apply** (*frule iMODb-Suc*[*THEN iffD1*])  
**apply** (*drule iMODb-leD*[of *Suc n*])  
**apply** *simp*  
**apply** *blast+*  
**done**  
**corollary** *Suc-in-imp-neq-iMODb*:  
 $\llbracket n \in S; \text{Suc } n \in S; m \neq \text{Suc } 0 \rrbracket \implies S \neq [r, \text{mod } m, c]$   
**by** (*blast dest: Suc-in-imp-not-subset-iMODb*)

### 1.3.2 Subset relation between intervals

**lemma**

*iIN-iFROM-subset-same*:  $[n..d] \subseteq [n..]$  **and**  
*iIN-iTILL-subset-same*:  $[n..d] \subseteq [..n+d]$  **and**  
*iMOD-iFROM-subset-same*:  $[r, \text{mod } m] \subseteq [r..]$  **and**  
*iMODb-iTILL-subset-same*:  $[r, \text{mod } m, c] \subseteq [..r+m*c]$  **and**  
*iMODb-iIN-subset-same*:  $[r, \text{mod } m, c] \subseteq [r..,m*c]$  **and**  
*iMODb-iMOD-subset-same*:  $[r, \text{mod } m, c] \subseteq [r, \text{mod } m]$   
**by** (*simp add: subset-iff iT-iff*)+

**lemmas** *iT-subset-same* =

*iIN-iFROM-subset-same*  
*iIN-iTILL-subset-same*  
*iMOD-iFROM-subset-same*  
*iMODb-iTILL-subset-same*  
*iMODb-iIN-subset-same*  
*iMODb-iTILL-subset-same*  
*iMODb-iMOD-subset-same*

**lemma** *iMODb-imp-iMOD*:  $x \in [r, \text{mod } m, c] \implies x \in [r, \text{mod } m]$   
**by** (*blast intro: iMODb-iMOD-subset-same*)

**lemma** *iMOD-imp-iMODb*:

$\llbracket x \in [r, \text{mod } m]; x \leq r + m * c \rrbracket \implies x \in [r, \text{mod } m, c]$   
**by** (*simp add: iT-iff*)

**lemma** *iMOD-singleton-subset-conv*:  $([r, \text{mod } m] \subseteq \{a\}) = (r = a \wedge m = 0)$

**apply** (*rule iffI*)

**apply** (*simp add: subset-singleton-conv iT-not-empty*)

**apply** (*simp add: set-eq-iff iT-iff*)

**apply** (*frule-tac x=r in spec, drule-tac x=r+m in spec*)

**apply** *simp*

**apply** (*simp add: iMOD-0 iIN-0*)

**done**

**lemma** *iMOD-singleton-eq-conv*:  $([r, \text{mod } m] = \{a\}) = (r = a \wedge m = 0)$

**apply** (*rule-tac t=[r, mod m] = {a} and s=[r, mod m]  $\subseteq$  {a} in subst*)

**apply** (*simp add: subset-singleton-conv iMOD-not-empty*)

**apply** (*simp add: iMOD-singleton-subset-conv*)

done

**lemma** *iMODb-singleton-subset-conv*:

$([r, \text{mod } m, c] \subseteq \{a\}) = (r = a \wedge (m = 0 \vee c = 0))$   
**apply** (*rule iffI*)  
**apply** (*simp add: subset-singleton-conv iT-not-empty*)  
**apply** (*simp add: set-eq-iff iT-iff*)  
**apply** (*frule-tac x=r in spec, drule-tac x=r+m in spec*)  
**apply** *clarsimp*  
**apply** (*fastforce simp: iMODb-0 iMODb-mod-0 iIN-0*)  
done

**lemma** *iMODb-singleton-eq-conv*:

$([r, \text{mod } m, c] = \{a\}) = (r = a \wedge (m = 0 \vee c = 0))$   
**apply** (*rule-tac t=[r, mod m, c] = {a} and s=[r, mod m, c] ⊆ {a} in subst*)  
**apply** (*simp add: subset-singleton-conv iMODb-not-empty*)  
**apply** (*simp add: iMODb-singleton-subset-conv*)  
done

**lemma** *iMODb-subset-imp-divisor-mod-0*:

$[0 < c'; [r', \text{mod } m', c'] \subseteq [r, \text{mod } m, c]] \implies m' \text{ mod } m = 0$   
**apply** (*simp add: subset-iff iMODb-iff*)  
**apply** (*drule gr0-imp-self-le-mult1[of - m']*)  
**apply** (*rule mod-add-eq-imp-mod-0[of r' m' m, THEN iffD1]*)  
**apply** (*frule-tac x=r' in spec, drule-tac x=r'+m' in spec*)  
**apply** *simp*  
done

**lemma** *iMOD-subset-imp-divisor-mod-0*:

$[r', \text{mod } m'] \subseteq [r, \text{mod } m] \implies m' \text{ mod } m = 0$   
**apply** (*simp add: subset-iff iMOD-iff*)  
**apply** (*rule mod-add-eq-imp-mod-0[of r' m' m, THEN iffD1]*)  
**apply** *simp*  
done

**lemma** *iMOD-subset-imp-iMODb-subset*:

$[ [r', \text{mod } m'] \subseteq [r, \text{mod } m]; r' + m' * c' \leq r + m * c ] \implies$   
 $[r', \text{mod } m', c'] \subseteq [r, \text{mod } m, c]$   
**by** (*simp add: subset-iff iT-iff*)

**lemma** *iMODb-subset-imp-iMOD-subset*:

$[ [r', \text{mod } m', c'] \subseteq [r, \text{mod } m, c]; 0 < c' ] \implies$   
 $[r', \text{mod } m'] \subseteq [r, \text{mod } m]$   
**apply** (*frule subsetD[of - - r']*)  
**apply** (*simp add: iMODb-iff*)  
**apply** (*rule subsetI*)  
**apply** (*simp add: iMOD-iff iMODb-iff, clarify*)  
**apply** (*drule mod-eq-mod-0-imp-mod-eq[where m=m and m'=m']*)  
**apply** (*simp add: iMODb-subset-imp-divisor-mod-0*)  
**apply** *simp*

done

**lemma** *iMODb-0-iMOD-subset-conv*:

$$([r', \text{mod } m', 0] \subseteq [r, \text{mod } m]) = (r' \text{ mod } m = r \text{ mod } m \wedge r \leq r')$$

**by** (*simp add: iMODb-0 iIN-0 singleton-subset-conv iMOD-iff*)

**lemma** *iFROM-subset-conv*:  $([n'..] \subseteq [n..]) = (n \leq n')$

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

**lemma** *iFROM-iMOD-subset-conv*:  $([n'..] \subseteq [r, \text{mod } m]) = (r \leq n' \wedge m = \text{Suc } 0)$

**apply** (*rule iffI*)

**apply** (*rule conjI*)

**apply** (*drule iMin-subset[OF iFROM-not-empty]*)

**apply** (*simp add: iT-Min*)

**apply** (*rule ccontr*)

**apply** (*cut-tac Suc-in-imp-not-subset-iMOD[of n' [n'..] m r]*)

**apply** (*simp add: iT-iff*)<sup>+</sup>

**apply** (*simp add: subset-iff iT-iff*)

done

**lemma** *iIN-subset-conv*:  $([n'..,d'] \subseteq [n..,d]) = (n \leq n' \wedge n'+d' \leq n+d)$

**apply** (*rule iffI*)

**apply** (*frule iMin-subset[OF iIN-not-empty]*)

**apply** (*drule Max-subset[OF iIN-not-empty - iIN-finite]*)

**apply** (*simp add: iIN-Min iIN-Max*)

**apply** (*simp add: subset-iff iIN-iff*)

done

**lemma** *iIN-iFROM-subset-conv*:  $([n'..,d'] \subseteq [n..]) = (n \leq n')$

**by** (*fastforce simp: subset-iff iFROM-iff iIN-iff*)

**lemma** *iIN-iTILL-subset-conv*:  $([n'..,d'] \subseteq [..n]) = (n' + d' \leq n)$

**by** (*fastforce simp: subset-iff iT-iff*)

**lemma** *iIN-iMOD-subset-conv*:

$$0 < d' \implies ([n'..,d'] \subseteq [r, \text{mod } m]) = (r \leq n' \wedge m = \text{Suc } 0)$$

**apply** (*rule iffI*)

**apply** (*frule iMin-subset[OF iIN-not-empty]*)

**apply** (*simp add: iT-Min*)

**apply** (*subgoal-tac n' ∈ [n'..,d']*)

**prefer** 2

**apply** (*simp add: iIN-iff*)

**apply** (*rule ccontr*)

**apply** (*frule Suc-in-imp-not-subset-iMOD[where r=r and m=m]*)

**apply** (*simp add: iIN-Suc*)<sup>+</sup>

**apply** (*simp add: iMOD-1 iIN-iFROM-subset-conv*)

done

**lemma** *iIN-iMODb-subset-conv*:

$0 < d' \implies$   
 $([n'..d'] \subseteq [r, \text{mod } m, c]) =$   
 $(r \leq n' \wedge m = \text{Suc } 0 \wedge n' + d' \leq r + m * c)$   
**apply** (*rule iffI*)  
**apply** (*frule subset-trans[OF - iMODb-iMOD-subset-same]*)  
**apply** (*simp add: iIN-iMOD-subset-conv iMODb-mod-1 iIN-subset-conv*)  
**apply** (*clarsimp simp: iMODb-mod-1 iIN-subset-conv*)  
**done**

**lemma** *iTILL-subset-conv*:  $([..n'] \subseteq [..n]) = (n' \leq n)$   
**by** (*simp add: iTILL-def*)

**lemma** *iTILL-iFROM-subset-conv*:  $([..n'] \subseteq [n..]) = (n = 0)$   
**apply** (*rule iffI*)  
**apply** (*drule subsetD[of - - 0]*)  
**apply** (*simp add: iT-iff*)  
**apply** (*simp add: iFROM-0*)  
**done**

**lemma** *iTILL-iIN-subset-conv*:  $([..n'] \subseteq [n..,d]) = (n = 0 \wedge n' \leq d)$   
**apply** (*rule iffI*)  
**apply** (*frule iMin-subset[OF iTILL-not-empty]*)  
**apply** (*drule Max-subset[OF iTILL-not-empty - iIN-finite]*)  
**apply** (*simp add: iT-Min iT-Max*)  
**apply** (*simp add: iIN-0-iTILL-conv iTILL-subset-conv*)  
**done**

**lemma** *iTILL-iMOD-subset-conv*:  
 $0 < n' \implies ([..n'] \subseteq [r, \text{mod } m]) = (r = 0 \wedge m = \text{Suc } 0)$   
**apply** (*drule iIN-iMOD-subset-conv[of n' 0 r m]*)  
**apply** (*simp add: iIN-0-iTILL-conv*)  
**done**

**lemma** *iTILL-iMODb-subset-conv*:  
 $0 < n' \implies ([..n'] \subseteq [r, \text{mod } m, c]) = (r = 0 \wedge m = \text{Suc } 0 \wedge n' \leq r + m * c)$   
**apply** (*drule iIN-iMODb-subset-conv[of n' 0 r m c]*)  
**apply** (*simp add: iIN-0-iTILL-conv*)  
**done**

**lemma** *iMOD-iFROM-subset-conv*:  $([r', \text{mod } m'] \subseteq [n..]) = (n \leq r')$   
**by** (*fastforce simp: subset-iff iT-iff*)

**lemma** *iMODb-iFROM-subset-conv*:  $([r', \text{mod } m', c'] \subseteq [n..]) = (n \leq r')$   
**by** (*fastforce simp: subset-iff iT-iff*)

**lemma** *iMODb-iIN-subset-conv*:

$([r', \text{mod } m', c'] \subseteq [n..d]) = (n \leq r' \wedge r' + m' * c' \leq n + d)$   
**by** (*fastforce simp: subset-iff iT-iff*)

**lemma** *iMODb-iTILL-subset-conv*:

$([r', \text{mod } m', c'] \subseteq [..n]) = (r' + m' * c' \leq n)$   
**by** (*fastforce simp: subset-iff iT-iff*)

**lemma** *iMOD-0-subset-conv*:  $([r', \text{mod } 0] \subseteq [r, \text{mod } m]) = (r' \text{ mod } m = r \text{ mod } m \wedge r \leq r')$

**by** (*fastforce simp: iMOD-0 iIN-0 singleton-subset-conv iMOD-iff*)

**lemma** *iMOD-subset-conv*:  $0 < m \implies$

$([r', \text{mod } m'] \subseteq [r, \text{mod } m]) =$   
 $(r' \text{ mod } m = r \text{ mod } m \wedge r \leq r' \wedge m' \text{ mod } m = 0)$

**apply** (*rule iffI*)

**apply** (*frule subsetD[of - r']*)

**apply** (*simp add: iMOD-iff*)

**apply** (*drule iMOD-subset-imp-divisor-mod-0*)

**apply** (*simp add: iMOD-iff*)

**apply** (*rule subsetI*)

**apply** (*simp add: iMOD-iff, elim conjE*)

**apply** (*drule mod-eq-mod-0-imp-mod-eq[where m'=m' and m=m]*)

**apply** *simp+*

**done**

**lemma** *iMODb-subset-mod-0-conv*:

$([r', \text{mod } m', c'] \subseteq [r, \text{mod } 0, c]) = (r'=r \wedge (m'=0 \vee c'=0))$   
**by** (*simp add: iMODb-mod-0 iIN-0 iMODb-singleton-subset-conv*)

**lemma** *iMODb-subset-0-conv*:

$([r', \text{mod } m', c'] \subseteq [r, \text{mod } m, 0]) = (r'=r \wedge (m'=0 \vee c'=0))$   
**by** (*simp add: iMODb-0 iIN-0 iMODb-singleton-subset-conv*)

**lemma** *iMODb-0-subset-conv*:

$([r', \text{mod } m', 0] \subseteq [r, \text{mod } m, c]) = (r' \in [r, \text{mod } m, c])$   
**by** (*simp add: iMODb-0 iIN-0*)

**lemma** *iMODb-mod-0-subset-conv*:

$([r', \text{mod } 0, c'] \subseteq [r, \text{mod } m, c]) = (r' \in [r, \text{mod } m, c])$   
**by** (*simp add: iMODb-mod-0 iIN-0*)

**lemma** *iMODb-subset-conv'*:  $\llbracket 0 < c; 0 < c' \rrbracket \implies$

$([r', \text{mod } m', c'] \subseteq [r, \text{mod } m, c]) =$   
 $(r' \text{ mod } m = r \text{ mod } m \wedge r \leq r' \wedge m' \text{ mod } m = 0 \wedge$   
 $r' + m' * c' \leq r + m * c)$

**apply** (*rule iffI*)

**apply** (*frule iMODb-subset-imp-iMOD-subset, assumption*)

**apply** (*drule iMOD-subset-imp-divisor-mod-0*)

**apply** (*frule subsetD[OF - iMinI-ex2[OF iMODb-not-empty]]*)

```

apply (drule Max-subset[OF iMODb-not-empty - iMODb-finite])
apply (simp add: iMODb-iff iMODb-Min iMODb-Max)
apply (elim conjE)
apply (case-tac m = 0, simp add: iMODb-mod-0)
apply (simp add: iMOD-subset-imp-iMODb-subset iMOD-subset-conv)
done

```

```

lemma iMODb-subset-conv:  $\llbracket 0 < m'; 0 < c' \rrbracket \implies$ 
   $([r', \text{mod } m', c'] \subseteq [r, \text{mod } m, c]) =$ 
   $(r' \text{ mod } m = r \text{ mod } m \wedge r \leq r' \wedge m' \text{ mod } m = 0 \wedge$ 
   $r' + m' * c' \leq r + m * c)$ 
apply (case-tac c = 0)
apply (simp add: iMODb-0 iIN-0 iMODb-singleton-subset-conv linorder-not-le,
intro impI)
apply (case-tac r' < r, simp)
apply (simp add: linorder-not-less)
apply (insert add-less-le-mono[of 0 m' * c' r r'])
apply simp
apply (simp add: iMODb-subset-conv')
done

```

```

lemma iMODb-iMOD-subset-conv:  $0 < c' \implies$ 
   $([r', \text{mod } m', c'] \subseteq [r, \text{mod } m]) =$ 
   $(r' \text{ mod } m = r \text{ mod } m \wedge r \leq r' \wedge m' \text{ mod } m = 0)$ 
apply (rule iffI)
apply (frule subsetD[OF - iMinI-ex2[OF iMODb-not-empty]])
apply (simp add: iMODb-Min iMOD-iff, elim conjE)
apply (simp add: iMODb-iMOD-iTILL-conv)
apply (subgoal-tac  $[r', \text{mod } m', c'] \subseteq [r, \text{mod } m] \cap [\dots r' + m' * c']$ )
prefer 2
apply (simp add: iMODb-iMOD-iTILL-conv)
apply (simp add: iMOD-iTILL-iMODb-conv iMODb-subset-imp-divisor-mod-0)
apply (rule subset-trans[OF iMODb-iMOD-subset-same])
apply (case-tac m = 0, simp)
apply (simp add: iMOD-subset-conv)
done

```

```

lemmas iT-subset-conv =
  iFROM-subset-conv
  iFROM-iMOD-subset-conv
  iTILL-subset-conv
  iTILL-iFROM-subset-conv
  iTILL-iIN-subset-conv
  iTILL-iMOD-subset-conv
  iTILL-iMODb-subset-conv
  iIN-subset-conv
  iIN-iFROM-subset-conv
  iIN-iTILL-subset-conv
  iIN-iMOD-subset-conv

```

*iIN-iMODb-subset-conv*  
*iMOD-subset-conv*  
*iMOD-iFROM-subset-conv*  
*iMODb-subset-conv'*  
*iMODb-subset-conv*  
*iMODb-iFROM-subset-conv*  
*iMODb-iIN-subset-conv*  
*iMODb-iTILL-subset-conv*  
*iMODb-iMOD-subset-conv*

**lemma** *iFROM-subset*:  $n \leq n' \implies [n'..] \subseteq [n..]$   
**by** (*simp add: iFROM-subset-conv*)

**lemma** *not-iFROM-iIN-subset*:  $\neg [n'..] \subseteq [n..,d]$   
**apply** (*rule ccontr, simp*)  
**apply** (*drule subsetD[of - - max n' (Suc (n + d))]*)  
**apply** (*simp add: iFROM-iff*)  
**apply** (*simp add: iIN-iff*)  
**done**

**lemma** *not-iFROM-iTILL-subset*:  $\neg [n'..] \subseteq [..n]$   
**by** (*simp add: iIN-0-iTILL-conv [symmetric] not-iFROM-iIN-subset*)

**lemma** *not-iFROM-iMOD-subset*:  $m \neq \text{Suc } 0 \implies \neg [n'..] \subseteq [r, \text{mod } m]$   
**apply** (*rule Suc-in-imp-not-subset-iMOD[of n']*)  
**apply** (*simp add: iT-iff*)  
**done**

**lemma** *not-iFROM-iMODb-subset*:  $\neg [n'..] \subseteq [r, \text{mod } m, c]$   
**by** (*rule infinite-not-subset-finite[OF iFROM-infinite iMODb-finite]*)

**lemma** *iIN-subset*:  $\llbracket n \leq n'; n' + d' \leq n + d \rrbracket \implies [n'..,d'] \subseteq [n..,d]$   
**by** (*simp add: iIN-subset-conv*)

**lemma** *iIN-iFROM-subset*:  $n \leq n' \implies [n'..,d'] \subseteq [n..]$   
**by** (*simp add: subset-iff iT-iff*)

**lemma** *iIN-iTILL-subset*:  $n' + d' \leq n \implies [n'..,d'] \subseteq [..n]$   
**by** (*simp add: iIN-0-iTILL-conv[symmetric] iIN-subset*)

**lemma** *not-iIN-iMODb-subset*:  $\llbracket 0 < d'; m \neq \text{Suc } 0 \rrbracket \implies \neg [n'..,d'] \subseteq [r, \text{mod } m, c]$   
**apply** (*rule Suc-in-imp-not-subset-iMODb[of n']*)  
**apply** (*simp add: iIN-iff*)  
**done**

**lemma** *not-iIN-iMOD-subset*:  $\llbracket 0 < d'; m \neq \text{Suc } 0 \rrbracket \implies \neg [n'..,d'] \subseteq [r, \text{mod } m]$   
**apply** (*rule ccontr, simp*)

**apply** (*case-tac*  $r \leq n' + d'$ )  
**apply** (*drule* *Int-greatest*[*OF* - *iIN-iTILL-subset*[*OF* *order-refl*]])  
**apply** (*simp* *add*: *iMOD-iTILL-iMODb-conv* *not-iIN-iMODb-subset*)  
**apply** (*drule* *subsetD*[*of* - -  $n'+d'$ ])  
**apply** (*simp* *add*: *iT-iff*)  
**done**

**lemma** *iTILL-subset*:  $n' \leq n \implies [\dots n'] \subseteq [\dots n]$   
**by** (*rule* *iTILL-subset-conv*[*THEN* *iffD2*])

**lemma** *iTILL-iFROM-subset*:  $([\dots n'] \subseteq [0\dots])$   
**by** (*simp* *add*: *iFROM-0*)

**lemma** *iTILL-iIN-subset*:  $n' \leq d \implies ([\dots n'] \subseteq [0\dots,d])$   
**by** (*simp* *add*: *iIN-0-iTILL-conv* *iTILL-subset*)

**lemma** *not-iTILL-iMOD-subset*:  
 $\llbracket 0 < n'; m \neq \text{Suc } 0 \rrbracket \implies \neg [\dots n'] \subseteq [r, \text{mod } m]$   
**by** (*simp* *add*: *iIN-0-iTILL-conv*[*symmetric*] *not-iIN-iMOD-subset*)

**lemma** *not-iTILL-iMODb-subset*:  
 $\llbracket 0 < n'; m \neq \text{Suc } 0 \rrbracket \implies \neg [\dots n'] \subseteq [r, \text{mod } m, c]$   
**by** (*simp* *add*: *iIN-0-iTILL-conv*[*symmetric*] *not-iIN-iMODb-subset*)

**lemma** *iMOD-iFROM-subset*:  $n \leq r' \implies [r', \text{mod } m'] \subseteq [n\dots]$   
**by** (*rule* *iMOD-iFROM-subset-conv*[*THEN* *iffD2*])

**lemma** *not-iMOD-iIN-subset*:  $0 < m' \implies \neg [r', \text{mod } m'] \subseteq [n\dots,d]$   
**by** (*rule* *infinite-not-subset-finite*[*OF* *iMOD-infinite* *iIN-finite*])

**lemma** *not-iMOD-iTILL-subset*:  $0 < m' \implies \neg [r', \text{mod } m'] \subseteq [\dots n]$   
**by** (*rule* *infinite-not-subset-finite*[*OF* *iMOD-infinite* *iTILL-finite*])

**lemma** *iMOD-subset*:  
 $\llbracket r \leq r'; r' \text{ mod } m = r \text{ mod } m; m' \text{ mod } m = 0 \rrbracket \implies [r', \text{mod } m'] \subseteq [r, \text{mod } m]$   
**apply** (*case-tac*  $m = 0$ , *simp*)  
**apply** (*simp* *add*: *iMOD-subset-conv*)  
**done**

**lemma** *not-iMOD-iMODb-subset*:  $0 < m' \implies \neg [r', \text{mod } m'] \subseteq [r, \text{mod } m, c]$   
**by** (*rule* *infinite-not-subset-finite*[*OF* *iMOD-infinite* *iMODb-finite*])

**lemma** *iMODb-iFROM-subset*:  $n \leq r' \implies [r', \text{mod } m', c'] \subseteq [n\dots]$   
**by** (*rule* *iMODb-iFROM-subset-conv*[*THEN* *iffD2*])

**lemma** *iMODb-iTILL-subset*:  
 $r' + m' * c' \leq n \implies [r', \text{mod } m', c'] \subseteq [\dots n]$   
**by** (*rule* *iMODb-iTILL-subset-conv*[*THEN* *iffD2*])

**lemma** *iMODb-iIN-subset*:

$\llbracket n \leq r'; r' + m' * c' \leq n + d \rrbracket \implies [r', \text{mod } m', c'] \subseteq [n \dots, d]$   
**by** (*simp add: iMODb-iIN-subset-conv*)

**lemma** *iMODb-iMOD-subset*:

$\llbracket r \leq r'; r' \text{ mod } m = r \text{ mod } m; m' \text{ mod } m = 0 \rrbracket \implies [r', \text{mod } m', c'] \subseteq [r, \text{mod } m]$

**apply** (*case-tac c' = 0*)  
**apply** (*simp add: iMODb-0 iIN-0 iMOD-iff*)  
**apply** (*simp add: iMODb-iMOD-subset-conv*)  
**done**

**lemma** *iMODb-subset*:

$\llbracket r \leq r'; r' \text{ mod } m = r \text{ mod } m; m' \text{ mod } m = 0; r' + m' * c' \leq r + m * c \rrbracket \implies [r', \text{mod } m', c'] \subseteq [r, \text{mod } m, c]$

**apply** (*case-tac m' = 0*)  
**apply** (*simp add: iMODb-mod-0 iIN-0 iMODb-iff*)  
**apply** (*case-tac c' = 0*)  
**apply** (*simp add: iMODb-0 iIN-0 iMODb-iff*)  
**apply** (*simp add: iMODb-subset-conv*)  
**done**

**lemma** *iFROM-trans*:  $\llbracket y \in [x \dots]; z \in [y \dots] \rrbracket \implies z \in [x \dots]$

**by** (*rule subsetD[OF iFROM-subset[OF iFROM-D]]*)

**lemma** *iTILL-trans*:  $\llbracket y \in [\dots x]; z \in [\dots y] \rrbracket \implies z \in [\dots x]$

**by** (*rule subsetD[OF iTILL-subset[OF iTILL-D]]*)

**lemma** *iIN-trans*:

$\llbracket y \in [x \dots, d]; z \in [y \dots, d']; d' \leq x + d - y \rrbracket \implies z \in [x \dots, d]$   
**by** *fastforce*

**lemma** *iMOD-trans*:

$\llbracket y \in [x, \text{mod } m]; z \in [y, \text{mod } m] \rrbracket \implies z \in [x, \text{mod } m]$   
**by** (*rule subsetD[OF iMOD-subset[OF iMOD-geD iMOD-modD mod-self]]*)

**lemma** *iMODb-trans*:

$\llbracket y \in [x, \text{mod } m, c]; z \in [y, \text{mod } m, c']; m * c' \leq x + m * c - y \rrbracket \implies z \in [x, \text{mod } m, c]$   
**by** *fastforce*

**lemma** *iMODb-trans'*:

$\llbracket y \in [x, \text{mod } m, c]; z \in [y, \text{mod } m, c']; c' \leq x \text{ div } m + c - y \text{ div } m \rrbracket \implies z \in [x, \text{mod } m, c]$

**apply** (*rule iMODb-trans[where c'=c', assumption+]*)  
**apply** (*frule iMODb-geD, frule div-le-mono[of x y m]*)  
**apply** (*simp add: add commute[of - c] add commute[of - m\*c]*)  
**apply** (*drule mult-le-mono[OF le-refl, of - - m]*)  
**apply** (*simp add: add-mult-distrib2 diff-mult-distrib2 minus-mod-eq-mult-div [symmetric]*)

**apply** (*simp add: iMODb-iff*)  
**done**

### 1.3.3 Equality of intervals

**lemma** *iFROM-eq-conv*:  $([n..] = [n'..]) = (n = n')$   
**apply** (*rule iffI*)  
**apply** (*drule set-eq-subset[THEN iffD1]*)  
**apply** (*simp add: iFROM-subset-conv*)  
**apply** *simp*  
**done**

**lemma** *iIN-eq-conv*:  $([n..,d] = [n'..,d']) = (n = n' \wedge d = d')$   
**apply** (*rule iffI*)  
**apply** (*drule set-eq-subset[THEN iffD1]*)  
**apply** (*simp add: iIN-subset-conv*)  
**apply** *simp*  
**done**

**lemma** *iTILL-eq-conv*:  $([..n] = [..n']) = (n = n')$   
**by** (*simp add: iIN-0-iTILL-conv[symmetric] iIN-eq-conv*)

**lemma** *iMOD-0-eq-conv*:  $([r, \text{mod } 0] = [r', \text{mod } m']) = (r = r' \wedge m' = 0)$   
**apply** (*simp add: iMOD-0 iIN-0*)  
**apply** (*simp add: iMOD-singleton-eq-conv eq-sym-conv[of {r}] eq-sym-conv[of r]*)  
**done**

**lemma** *iMOD-eq-conv*:  $0 < m \implies ([r, \text{mod } m] = [r', \text{mod } m']) = (r = r' \wedge m = m')$   
**apply** (*case-tac m' = 0*)  
**apply** (*simp add: eq-sym-conv[of [r, mod m]] iMOD-0-eq-conv*)  
**apply** (*rule iffI*)  
**apply** (*fastforce simp add: set-eq-subset iMOD-subset-conv*)  
**apply** *simp*  
**done**

**lemma** *iMODb-mod-0-eq-conv*:  
 $([r, \text{mod } 0, c] = [r', \text{mod } m', c']) = (r = r' \wedge (m' = 0 \vee c' = 0))$   
**apply** (*simp add: iMODb-mod-0 iIN-0*)  
**apply** (*fastforce simp: iMODb-singleton-eq-conv eq-sym-conv[of {r}]*)  
**done**

**lemma** *iMODb-0-eq-conv*:  
 $([r, \text{mod } m, 0] = [r', \text{mod } m', c']) = (r = r' \wedge (m' = 0 \vee c' = 0))$   
**apply** (*simp add: iMODb-0 iIN-0*)  
**apply** (*fastforce simp: iMODb-singleton-eq-conv eq-sym-conv[of {r}]*)  
**done**

**lemma** *iMODb-eq-conv*:  $\llbracket 0 < m; 0 < c \rrbracket \implies$

$([r, \text{mod } m, c] = [r', \text{mod } m', c']) = (r = r' \wedge m = m' \wedge c = c')$   
**apply** (*case-tac*  $c' = 0$ )  
**apply** (*simp add: iMODb-0 iIN-0 iMODb-singleton-eq-conv*)  
**apply** (*rule iffI*)  
**apply** (*fastforce simp: set-eq-subset iMODb-subset-conv'*)  
**apply** *simp*  
**done**

**lemma** *iMOD-iFROM-eq-conv*:  $([n..] = [r, \text{mod } m]) = (n = r \wedge m = \text{Suc } 0)$   
**by** (*fastforce simp: iMOD-1[symmetric] iMOD-eq-conv*)

**lemma** *iMODb-iIN-0-eq-conv*:  
 $([n.., 0] = [r, \text{mod } m, c]) = (n = r \wedge (m = 0 \vee c = 0))$   
**by** (*simp add: iIN-0 eq-commute[of {n}] eq-commute[of n] iMODb-singleton-eq-conv*)

**lemma** *iMODb-iIN-eq-conv*:  
 $0 < d \implies ([n.., d] = [r, \text{mod } m, c]) = (n = r \wedge m = \text{Suc } 0 \wedge c = d)$   
**by** (*fastforce simp: iMODb-mod-1[symmetric] iMODb-eq-conv*)

### 1.3.4 Inequality of intervals

**lemma** *iFROM-iIN-neq*:  $[n'..] \neq [n.., d]$   
**apply** (*rule ccontr*)  
**apply** (*insert iFROM-infinite[of n'], insert iIN-finite[of n d]*)  
**apply** *simp*  
**done**

**corollary** *iFROM-iTILL-neq*:  $[n'..] \neq [..n]$   
**by** (*simp add: iIN-0-iTILL-conv[symmetric] iFROM-iIN-neq*)

**corollary** *iFROM-iMOD-neq*:  $m \neq \text{Suc } 0 \implies [n..] \neq [r, \text{mod } m]$   
**apply** (*subgoal-tac*  $n \in [n..]$ )  
**prefer** 2  
**apply** (*simp add: iFROM-iff*)  
**apply** (*simp add: Suc-in-imp-neq-iMOD iFROM-Suc*)  
**done**

**corollary** *iFROM-iMODb-neq*:  $[n..] \neq [r, \text{mod } m, c]$   
**apply** (*rule ccontr*)  
**apply** (*insert iMODb-finite[of r m c], insert iFROM-infinite[of n]*)  
**apply** *simp*  
**done**

**corollary** *iIN-iMOD-neq*:  $0 < m \implies [n.., d] \neq [r, \text{mod } m]$   
**apply** (*rule ccontr*)  
**apply** (*insert iMOD-infinite[of m r], insert iIN-finite[of n d]*)  
**apply** *simp*  
**done**

**corollary** *iIN-iMODb-neq2*:  $\llbracket m \neq \text{Suc } 0; 0 < d \rrbracket \implies [n.., d] \neq [r, \text{mod } m, c]$

**apply** (*subgoal-tac*  $n \in [n\dots d]$ )  
**prefer** 2  
**apply** (*simp add: iIN-iff*)  
**apply** (*simp add: Suc-in-imp-neq-iMODb iIN-Suc*)  
**done**

**lemma** *iIN-iMODb-neq*:  $\llbracket 2 \leq m; 0 < c \rrbracket \implies [n\dots d] \neq [r, \text{mod } m, c]$   
**apply** (*simp add: nat-ge2-conv, elim conjE*)  
**apply** (*case-tac d=0*)  
**apply** (*rule not-sym*)  
**apply** (*simp add: iIN-0 iMODb-singleton-eq-conv*)  
**apply** (*simp add: iIN-iMODb-neq2*)  
**done**

**lemma** *iTILL-iIN-neq*:  $0 < n \implies [\dots n^\wedge] \neq [n\dots d]$   
**by** (*fastforce simp: set-eq-iff iT-iff*)

**corollary** *iTILL-iMOD-neq*:  $0 < m \implies [\dots n] \neq [r, \text{mod } m]$   
**by** (*simp add: iIN-0-iTILL-conv[symmetric] iIN-iMOD-neq*)

**corollary** *iTILL-iMODb-neq*:  
 $\llbracket m \neq \text{Suc } 0; 0 < n \rrbracket \implies [\dots n] \neq [r, \text{mod } m, c]$   
**by** (*simp add: iIN-0-iTILL-conv[symmetric] iIN-iMODb-neq2*)

**lemma** *iMOD-iMODb-neq*:  $0 < m \implies [r, \text{mod } m] \neq [r', \text{mod } m', c^\wedge]$   
**apply** (*rule ccontr*)  
**apply** (*insert iMODb-finite[of r' m' c^\wedge], insert iMOD-infinite[of m r]*)  
**apply** *simp*  
**done**

**lemmas** *iT-neq* =  
*iFROM-iTILL-neq iFROM-iIN-neq iFROM-iMOD-neq iFROM-iMODb-neq*  
*iTILL-iIN-neq iTILL-iMOD-neq iTILL-iMODb-neq*  
*iIN-iMOD-neq iIN-iMODb-neq iIN-iMODb-neq2*  
*iMOD-iMODb-neq*

## 1.4 Union and intersection of intervals

**lemma** *iFROM-union'*:  $[n\dots] \cup [n'\dots] = [\min n n'\dots]$   
**by** (*fastforce simp: iFROM-iff*)

**corollary** *iFROM-union*:  $n \leq n' \implies [n\dots] \cup [n'\dots] = [n\dots]$   
**by** (*simp add: iFROM-union' min-eqL*)

**lemma** *iFROM-inter'*:  $[n\dots] \cap [n'\dots] = [\max n n'\dots]$   
**by** (*fastforce simp: iFROM-iff*)

**corollary** *iFROM-inter*:  $n' \leq n \implies [n\dots] \cap [n'\dots] = [n\dots]$   
**by** (*simp add: iFROM-inter' max-eqL*)

**lemma** *iTILL-union'*:  $[\dots n] \cup [\dots n'] = [\dots \max n n']$

**by** (*fastforce simp: iTILL-iff*)

**corollary** *iTILL-union*:  $n' \leq n \implies [\dots n] \cup [\dots n'] = [\dots n]$

**by** (*simp add: iTILL-union' max-eqL*)

**lemma** *iTILL-iFROM-union*:  $n \leq n' \implies [\dots n'] \cup [n\dots] = UNIV$

**by** (*fastforce simp: iT-iff*)

**lemma** *iTILL-inter'*:  $[\dots n] \cap [\dots n'] = [\dots \min n n']$

**by** (*fastforce simp: iT-iff*)

**corollary** *iTILL-inter*:  $n \leq n' \implies [\dots n] \cap [\dots n'] = [\dots n]$

**by** (*simp add: iTILL-inter' min-eqL*)

Union and intersection for iIN only when there intersection is not empty and none of them is other's subset, for instance: .. n .. n+d .. n' .. n'+d'

**lemma** *iIN-union*:

$\llbracket n \leq n'; n' \leq \text{Suc } (n + d); n + d \leq n' + d' \rrbracket \implies$

$[n\dots, d] \cup [n'\dots, d'] = [n\dots, n' - n + d']$

**by** (*fastforce simp add: iIN-iff*)

**lemma** *iIN-append-union*:

$[n\dots, d] \cup [n + d\dots, d'] = [n\dots, d + d']$

**by** (*simp add: iIN-union*)

**lemma** *iIN-append-union-Suc*:

$[n\dots, d] \cup [\text{Suc } (n + d)\dots, d'] = [n\dots, \text{Suc } (d + d')]$

**by** (*simp add: iIN-union*)

**lemma** *iIN-append-union-pred*:

$0 < d \implies [n\dots, d - \text{Suc } 0] \cup [n + d\dots, d'] = [n\dots, d + d']$

**by** (*simp add: iIN-union*)

**lemma** *iIN-iFROM-union*:

$n' \leq \text{Suc } (n + d) \implies [n\dots, d] \cup [n'\dots] = [\min n n'\dots]$

**by** (*fastforce simp: iIN-iff*)

**lemma** *iIN-iFROM-append-union*:

$[n\dots, d] \cup [n + d\dots] = [n\dots]$

**by** (*simp add: iIN-iFROM-union min-eqL*)

**lemma** *iIN-iFROM-append-union-Suc*:

$[n\dots, d] \cup [\text{Suc } (n + d)\dots] = [n\dots]$

**by** (*simp add: iIN-iFROM-union min-eqL*)

**lemma** *iIN-iFROM-append-union-pred*:

$0 < d \implies [n\dots, d - \text{Suc } 0] \cup [n + d\dots] = [n\dots]$   
**by** (*simp add: iIN-iFROM-union min-eqL*)

**lemma** *iIN-inter*:

$\llbracket n \leq n'; n' \leq n + d; n + d \leq n' + d' \rrbracket \implies$   
 $[n\dots, d] \cap [n'\dots, d'] = [n'\dots, n + d - n']$   
**by** (*fastforce simp: iIN-iff*)

**lemma** *iMOD-union*:

$\llbracket r \leq r'; r \bmod m = r' \bmod m \rrbracket \implies$   
 $[r, \bmod m] \cup [r', \bmod m] = [r, \bmod m]$   
**by** (*fastforce simp: iT-iff*)

**lemma** *iMOD-union'*:

$r \bmod m = r' \bmod m \implies$   
 $[r, \bmod m] \cup [r', \bmod m] = [\min r r', \bmod m]$   
**apply** (*case-tac r ≤ r'*)  
**apply** (*fastforce simp: iMOD-union min-eq*)  
**done**

**lemma** *iMOD-inter*:

$\llbracket r \leq r'; r \bmod m = r' \bmod m \rrbracket \implies$   
 $[r, \bmod m] \cap [r', \bmod m] = [r', \bmod m]$   
**by** (*fastforce simp: iT-iff*)

**lemma** *iMOD-inter'*:

$r \bmod m = r' \bmod m \implies$   
 $[r, \bmod m] \cap [r', \bmod m] = [\max r r', \bmod m]$   
**apply** (*case-tac r ≤ r'*)  
**apply** (*fastforce simp: iMOD-inter max-eq*)  
**done**

**lemma** *iMODb-union*:

$\llbracket r \leq r'; r \bmod m = r' \bmod m; r' \leq r + m * c; r + m * c \leq r' + m * c' \rrbracket \implies$   
 $[r, \bmod m, c] \cup [r', \bmod m, c'] = [r, \bmod m, r' \text{ div } m - r \text{ div } m + c']$   
**apply** (*rule set-eqI*)  
**apply** (*simp add: iMODb-iff*)  
**apply** (*drule sym[of r mod m], simp*)  
**apply** (*fastforce simp: add-mult-distrib2 diff-mult-distrib2 minus-mod-eq-mult-div*  
*[symmetric]*)  
**done**

**lemma** *iMODb-append-union*:

$[r, \bmod m, c] \cup [r + m * c, \bmod m, c'] = [r, \bmod m, c + c']$   
**apply** (*insert iMODb-union[of r r + m \* c m c c']*)  
**apply** (*case-tac m = 0*)  
**apply** (*simp add: iMODb-mod-0*)  
**apply** *simp*  
**done**

**lemma** *iMODb-iMOD-append-union'*:  
 $\llbracket r \bmod m = r' \bmod m; r' \leq r + m * \text{Suc } c \rrbracket \implies$   
 $[r, \bmod m, c] \cup [r', \bmod m] = [\min r r', \bmod m]$   
**apply** (*subgoal-tac* (*min r r'*) *mod m = r' mod m*)  
**prefer** 2  
**apply** (*simp add: min-def*)  
**apply** (*rule set-eqI*)  
**apply** (*simp add: iT-iff*)  
**apply** (*drule sym*[*of r mod m*], *simp*)  
**apply** (*rule iffI*)  
**apply** *fastforce*  
**apply** (*clarsimp simp: linorder-not-le*)  
**apply** (*case-tac*  $r \leq r'$ )  
**apply** (*simp add: min-eqL*)  
**apply** (*rule add-le-imp-le-right*[*of - m*])  
**apply** (*rule less-mod-eq-imp-add-divisor-le*)  
**apply** *simp+*  
**done**

**lemma** *iMODb-iMOD-append-union*:  
 $\llbracket r \leq r'; r \bmod m = r' \bmod m; r' \leq r + m * \text{Suc } c \rrbracket \implies$   
 $[r, \bmod m, c] \cup [r', \bmod m] = [r, \bmod m]$   
**by** (*simp add: iMODb-iMOD-append-union' min-eqL*)

**lemma** *iMODb-append-union-Suc*:  
 $[r, \bmod m, c] \cup [r + m * \text{Suc } c, \bmod m, c'] = [r, \bmod m, \text{Suc } (c + c')]$   
**apply** (*subst insert-absorb*[*of r + m \* c*]  $[r, \bmod m, c] \cup [r + m * \text{Suc } c, \bmod m, c']$ , *symmetric*)  
**apply** (*simp add: iT-iff*)  
**apply** (*simp del: Un-insert-right add: Un-insert-right*[*symmetric*] *add.commute*[*of m*] *add.assoc*[*symmetric*] *iMODb-Suc-pred-insert-conv*)  
**apply** (*simp add: iMODb-append-union*)  
**done**

**lemma** *iMODb-append-union-pred*:  
 $0 < c \implies [r, \bmod m, c - \text{Suc } 0] \cup [r + m * c, \bmod m, c'] = [r, \bmod m, c + c']$   
**by** (*insert iMODb-append-union-Suc*[*of r m c - Suc 0 c'*], *simp*)

**lemma** *iMODb-inter*:  
 $\llbracket r \leq r'; r \bmod m = r' \bmod m; r' \leq r + m * c; r + m * c \leq r' + m * c' \rrbracket \implies$   
 $[r, \bmod m, c] \cap [r', \bmod m, c'] = [r', \bmod m, c - (r' - r) \text{ div } m]$   
**apply** (*rule set-eqI*)  
**apply** (*simp add: iMODb-iff*)  
**apply** (*simp add: diff-mult-distrib2*)  
**apply** (*simp add: mult.commute*[*of - (r' - r) div m*])  
**apply** (*simp add: mod-0-div-mult-cancel*[*THEN iffD1, OF mod-eq-imp-diff-mod-0*])

```

apply (simp add: add.commute[of - r])
apply fastforce
done

```

```

lemmas iT-union' =
  iFROM-union'
  iTILL-union'
  iMOD-union'
  iMODb-iMOD-append-union'

```

```

lemmas iT-union =
  iFROM-union
  iTILL-union
  iTILL-iFROM-union
  iIN-union
  iIN-iFROM-union
  iMOD-union
  iMODb-union

```

```

lemmas iT-union-append =
  iIN-append-union
  iIN-append-union-Suc
  iIN-append-union-pred
  iIN-iFROM-append-union
  iIN-iFROM-append-union-Suc
  iIN-iFROM-append-union-pred
  iMODb-append-union
  iMODb-iMOD-append-union
  iMODb-append-union-Suc
  iMODb-append-union-pred

```

```

lemmas iT-inter' =
  iFROM-inter'
  iTILL-inter'
  iMOD-inter'

```

```

lemmas iT-inter =
  iFROM-inter
  iTILL-inter
  iIN-inter
  iMOD-inter
  iMODb-inter

```

**lemma** *mod-partition-Union*:

$$0 < m \implies (\bigcup k. A \cap [k * m \dots, m - \text{Suc } 0]) = A$$

```

apply simp

```

```

apply (rule subst[where s=UNIV and P= $\lambda x. A \cap x = A$ ])

```

```

apply (rule set-eqI)

```

```

  apply (simp add: iT-iff)

```

```

  apply (rule-tac x=x div m in exI)

```

**apply** (*simp add: div-mult-cancel*)  
**apply** (*subst add commute*)  
**apply** (*rule le-add-diff*)  
**apply** (*simp add: Suc-mod-le-divisor*)  
**apply simp**  
**done**

**lemma** *finite-mod-partition-Union:*

$\llbracket 0 < m; \text{finite } A \rrbracket \implies$   
 $(\bigcup_{k \leq \text{Max } A \text{ div } m} A \cap [k * m .., m - \text{Suc } 0]) = A$   
**apply** (*rule subst[OF mod-partition-Union[of m], where*  
 $P = \lambda x. (\bigcup_{k \leq \text{Max } A \text{ div } m} A \cap [k * m .., m - \text{Suc } 0]) = x$ )  
**apply assumption**  
**apply** (*rule set-eqI*)  
**apply** (*simp add: iIN-iff*)  
**apply** (*rule iffI, blast*)  
**apply clarsimp**  
**apply** (*rename-tac x x1*)  
**apply** (*rule-tac x=x div m in bexI*)  
**apply** (*frule in-imp-not-empty[where A=A]*)  
**apply** (*frule-tac Max-ge, assumption*)  
**apply** (*cut-tac n=x and k=x div m and m=m in div-imp-le-less*)  
**apply clarsimp+**  
**apply** (*drule-tac m=x in less-imp-le-pred*)  
**apply** (*simp add: add commute[of m]*)  
**apply** (*simp add: div-le-mono*)  
**done**

**lemma** *mod-partition-is-disjoint:*

$\llbracket 0 < (m::\text{nat}); k \neq k' \rrbracket \implies$   
 $(A \cap [k * m .., m - \text{Suc } 0]) \cap (A \cap [k' * m .., m - \text{Suc } 0]) = \{\}$   
**apply** (*clarsimp simp add: all-not-in-conv[symmetric] iT-iff*)  
**apply** (*subgoal-tac  $\wedge k. \llbracket k * m \leq x; x \leq k * m + m - \text{Suc } 0 \rrbracket \implies x \text{ div } m = k$* )  
**apply blast**  
**apply** (*rule le-less-imp-div, assumption*)  
**apply simp**  
**done**

## 1.5 Cutting intervals

**lemma** *iTILL-cut-le:*  $[\dots n] \downarrow \leq t = (\text{if } t \leq n \text{ then } [\dots t] \text{ else } [\dots n])$   
**unfolding** *i-cut-defs iT-defs atMost-def*  
**by force**

**corollary** *iTILL-cut-le1:*  $t \in [\dots n] \implies [\dots n] \downarrow \leq t = [\dots t]$   
**by** (*simp add: iTILL-cut-le iT-iff*)

**corollary** *iTILL-cut-le2:*  $t \notin [\dots n] \implies [\dots n] \downarrow \leq t = [\dots n]$   
**by** (*simp add: iTILL-cut-le iT-iff*)

**lemma** *iFROM-cut-le*:

$[n..] \downarrow \leq t =$   
*(if*  $t < n$  *then*  $\{\}$  *else*  $[n.., t-n]$ *)*

**by** (*simp add: set-eq-iff i-cut-mem-iff iT-iff*)

**corollary** *iFROM-cut-le1*:  $t \in [n..] \implies [n..] \downarrow \leq t = [n.., t - n]$

**by** (*simp add: iFROM-cut-le iT-iff*)

**lemma** *iIN-cut-le*:

$[n.., d] \downarrow \leq t =$  (  
*if*  $t < n$  *then*  $\{\}$  *else*  
*if*  $t \leq n+d$  *then*  $[n.., t-n]$   
*else*  $[n.., d]$ *)*

**by** (*force simp: set-eq-iff i-cut-mem-iff iT-iff*)

**corollary** *iIN-cut-le1*:

$t \in [n.., d] \implies [n.., d] \downarrow \leq t = [n.., t - n]$

**by** (*simp add: iIN-cut-le iT-iff*)

**lemma** *iMOD-cut-le*:

$[r, \text{mod } m] \downarrow \leq t =$  (  
*if*  $t < r$  *then*  $\{\}$   
*else*  $[r, \text{mod } m, (t - r) \text{ div } m]$ *)*

**apply** (*case-tac m = 0*)

**apply** (*simp add: iMOD-0 iMODb-0 iIN-0 i-cut-empty i-cut-singleton*)

**apply** (*case-tac t < r*)

**apply** (*simp add: cut-le-Min-empty iMOD-Min*)

**apply** (*clarsimp simp: linorder-not-less set-eq-iff i-cut-mem-iff iT-iff*)

**apply** (*rule conj-cong, simp*) $+$

**apply** (*clarsimp simp: minus-mod-eq-mult-div [symmetric]*)

**apply** (*drule-tac x=r and y=x in le-imp-less-or-eq,erule disjE*)

**prefer** 2

**apply** *simp*

**apply** (*drule-tac x=r and y=x and m=m in less-mod-eq-imp-add-divisor-le, simp*)

**apply** (*rule iffI*)

**apply** (*rule-tac x=x in subst[OF mod-eq-imp-diff-mod-eq[of - m r t], rule-format], simp*) $+$

**apply** (*subgoal-tac r + (t - x) mod m ≤ t*)

**prefer** 2

**apply** (*simp add: order-trans[OF add-le-mono2[OF mod-le-divisor]]*)

**apply** *simp*

**apply** (*simp add: le-imp-sub-mod-le*)

**apply** (*subgoal-tac r + (t - r) mod m ≤ t*)

**prefer** 2

**apply** (*rule ccontr*)

**apply** *simp*

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

**lemma** *iMOD-cut-le1*:

$t \in [r, \text{mod } m] \implies$   
 $[r, \text{mod } m] \downarrow \leq t = [r, \text{mod } m, (t - r) \text{ div } m]$   
**by** (*simp add: iMOD-cut-le iT-iff*)

**lemma** *iMODb-cut-le*:

$[r, \text{mod } m, c] \downarrow \leq t =$  (  
  *if*  $t < r$  *then*  $\{\}$   
  *else if*  $t < r + m * c$  *then*  $[r, \text{mod } m, (t - r) \text{ div } m]$   
  *else*  $[r, \text{mod } m, c]$ )  
**apply** (*case-tac m = 0*)  
**apply** (*simp add: iMODb-mod-0 iIN-0 cut-le-singleton*)  
**apply** (*case-tac t < r*)  
**apply** (*simp add: cut-le-Min-empty iT-Min*)  
**apply** (*case-tac r + m \* c ≤ t*)  
**apply** (*simp add: cut-le-Max-all iT-Max iT-finite*)  
**apply** (*simp add: linorder-not-le linorder-not-less*)  
**apply** (*rule-tac t=c and s=(r + m \* c - r) div m in subst, simp*)  
**apply** (*subst iMOD-iTILL-iMODb-conv[symmetric], simp*)  
**apply** (*simp add: cut-le-Int-right iTILL-cut-le*)  
**apply** (*simp add: iMOD-iTILL-iMODb-conv*)  
**done**

**lemma** *iMODb-cut-le1*:

$t \in [r, \text{mod } m, c] \implies$   
 $[r, \text{mod } m, c] \downarrow \leq t = [r, \text{mod } m, (t - r) \text{ div } m]$   
**by** (*clarsimp simp: iMODb-cut-le iT-iff iMODb-mod-0*)

**lemma** *iTILL-cut-less*:

$[..n] \downarrow < t =$  (  
  *if*  $n < t$  *then*  $[..n]$  *else*  
  *if*  $t = 0$  *then*  $\{\}$   
  *else*  $[..t - \text{Suc } 0]$ )  
**apply** (*case-tac n < t*)  
**apply** (*simp add: cut-less-Max-all iT-Max iT-finite*)  
**apply** (*case-tac t = 0*)  
**apply** (*simp add: cut-less-0-empty*)  
**apply** (*fastforce simp: nat-cut-less-le-conv iTILL-cut-le*)  
**done**

**lemma** *iTILL-cut-less1*:

$\llbracket t \in [..n]; 0 < t \rrbracket \implies [..n] \downarrow < t = [..t - \text{Suc } 0]$   
**by** (*simp add: iTILL-cut-less iT-iff*)

**lemma** *iFROM-cut-less*:

```

[n...] ↓< t = (
  if t ≤ n then {}
  else [n..., t - Suc n])
apply (case-tac t ≤ n)
apply (simp add: cut-less-Min-empty iT-Min)
apply (fastforce simp: nat-cut-less-le-conv iFROM-cut-le)
done

```

**lemma** *iFROM-cut-less1*:

```

n < t ⇒ [n...] ↓< t = [n..., t - Suc n]
by (simp add: iFROM-cut-less)

```

**lemma** *iIN-cut-less*:

```

[n..., d] ↓< t = (
  if t ≤ n then {} else
  if t ≤ n + d then [n..., t - Suc n]
  else [n..., d])
apply (case-tac t ≤ n)
apply (simp add: cut-less-Min-empty iT-Min )
apply (case-tac n + d < t)
apply (simp add: cut-less-Max-all iT-Max iT-finite)
apply (fastforce simp: nat-cut-less-le-conv iIN-cut-le)
done

```

**lemma** *iIN-cut-less1*:

```

[[ t ∈ [n..., d]; n < t ]] ⇒ [n..., d] ↓< t = [n..., t - Suc n]
by (simp add: iIN-cut-less iT-iff)

```

**lemma** *iMOD-cut-less*:

```

[r, mod m] ↓< t = (
  if t ≤ r then {}
  else [r, mod m, (t - Suc r) div m])
apply (case-tac t = 0)
apply (simp add: cut-less-0-empty)
apply (simp add: nat-cut-less-le-conv iMOD-cut-le)
apply fastforce
done

```

**lemma** *iMOD-cut-less1*:

```

[[ t ∈ [r, mod m]; r < t ]] ⇒
[r, mod m] ↓< t = [r, mod m, (t - r) div m - Suc 0]
apply (case-tac m = 0)
apply (simp add: iMOD-0 iMODb-mod-0 iIN-0)
apply (simp add: iMOD-cut-less)
apply (simp add: mod-0-imp-diff-Suc-div-conv mod-eq-imp-diff-mod-0 iT-iff)
done

```

**lemma** *iMODb-cut-less*:

```

[r, mod m, c] ↓< t = (
  if t ≤ r then {} else
  if r + m * c < t then [r, mod m, c]
  else [r, mod m, (t - Suc r) div m])
apply (case-tac t = 0)
apply (simp add: cut-less-0-empty)
apply (simp add: nat-cut-less-le-conv iMODb-cut-le)
apply fastforce
done

```

**lemma** *iMODb-cut-less1*:  $\llbracket t \in [r, \text{mod } m, c]; r < t \rrbracket \implies$

```

[r, mod m, c] ↓< t = [r, mod m, (t - r) div m - Suc 0]
apply (case-tac m = 0)
apply (simp add: iMODb-mod-0 iIN-0)
apply (simp add: iMODb-cut-less)
apply (simp add: mod-0-imp-diff-Suc-div-conv mod-eq-imp-diff-mod-0 iT-iff)
done

```

**lemmas** *iT-cut-le* =

```

iTILL-cut-le
iFROM-cut-le
iIN-cut-le
iMOD-cut-le
iMODb-cut-le

```

**lemmas** *iT-cut-le1* =

```

iTILL-cut-le1
iFROM-cut-le1
iIN-cut-le1
iMOD-cut-le1
iMODb-cut-le1

```

**lemmas** *iT-cut-less* =

```

iTILL-cut-less
iFROM-cut-less
iIN-cut-less
iMOD-cut-less
iMODb-cut-less

```

**lemmas** *iT-cut-less1* =

```

iTILL-cut-less1
iFROM-cut-less1
iIN-cut-less1
iMOD-cut-less1
iMODb-cut-less1

```

**lemmas** *iT-cut-le-less* =

*iTILL-cut-le*  
*iTILL-cut-less*  
*iFROM-cut-le*  
*iFROM-cut-less*  
*iIN-cut-le*  
*iIN-cut-less*  
*iMOD-cut-le*  
*iMOD-cut-less*  
*iMODb-cut-le*  
*iMODb-cut-less*

**lemmas** *iT-cut-le-less1* =

*iTILL-cut-le1*  
*iTILL-cut-less1*  
*iFROM-cut-le1*  
*iFROM-cut-less1*  
*iIN-cut-le1*  
*iIN-cut-less1*  
*iMOD-cut-le1*  
*iMOD-cut-less1*  
*iMODb-cut-le1*  
*iMODb-cut-less1*

**lemma** *iTILL-cut-ge*:

$[\dots n] \downarrow_{\geq} t = (\text{if } n < t \text{ then } \{\} \text{ else } [t\dots n-t])$

**by** (*force simp: i-cut-mem-iff iT-iff*)

**corollary** *iTILL-cut-ge1*:  $t \in [\dots n] \implies [\dots n] \downarrow_{\geq} t = [t\dots n-t]$

**by** (*simp add: iTILL-cut-ge iT-iff*)

**corollary** *iTILL-cut-ge2*:  $t \notin [\dots n] \implies [\dots n] \downarrow_{\geq} t = \{\}$

**by** (*simp add: iTILL-cut-ge iT-iff*)

**lemma** *iTILL-cut-greater*:

$[\dots n] \downarrow_{>} t = (\text{if } n \leq t \text{ then } \{\} \text{ else } [Suc\ t\dots n - Suc\ t])$

**by** (*force simp: i-cut-mem-iff iT-iff*)

**corollary** *iTILL-cut-greater1*:

$t \in [\dots n] \implies t < n \implies [\dots n] \downarrow_{>} t = [Suc\ t\dots n - Suc\ t]$

**by** (*simp add: iTILL-cut-greater iT-iff*)

**corollary** *iTILL-cut-greater2*:  $t \notin [\dots n] \implies [\dots n] \downarrow_{>} t = \{\}$

**by** (*simp add: iTILL-cut-greater iT-iff*)

**lemma** *iFROM-cut-ge*:

$[n\dots] \downarrow_{\geq} t = (\text{if } n \leq t \text{ then } [t\dots] \text{ else } [n\dots])$

**by** (*force simp: i-cut-mem-iff iT-iff*)

**corollary** *iFROM-cut-ge1*:  $t \in [n..] \implies [n..] \downarrow_{\geq} t = [t..]$   
**by** (*simp add: iFROM-cut-ge iT-iff*)

**lemma** *iFROM-cut-greater*:

$[n..] \downarrow_{>} t = (\text{if } n \leq t \text{ then } [Suc\ t..] \text{ else } [n..])$

**by** (*force simp: i-cut-mem-iff iT-iff*)

**corollary** *iFROM-cut-greater1*:

$t \in [n..] \implies [n..] \downarrow_{>} t = [Suc\ t..]$

**by** (*simp add: iFROM-cut-greater iT-iff*)

**lemma** *iIN-cut-ge*:

$[n..,d] \downarrow_{\geq} t = (\text{if } t < n \text{ then } [n..,d] \text{ else } \text{if } t \leq n+d \text{ then } [t..,n+d-t] \text{ else } \{\})$

**by** (*force simp: i-cut-mem-iff iT-iff*)

**corollary** *iIN-cut-ge1*:  $t \in [n..,d] \implies$

$[n..,d] \downarrow_{\geq} t = [t..,n+d-t]$

**by** (*simp add: iIN-cut-ge iT-iff*)

**corollary** *iIN-cut-ge2*:  $t \notin [n..,d] \implies$

$[n..,d] \downarrow_{\geq} t = (\text{if } t < n \text{ then } [n..,d] \text{ else } \{\})$

**by** (*simp add: iIN-cut-ge iT-iff*)

**lemma** *iIN-cut-greater*:

$[n..,d] \downarrow_{>} t = (\text{if } t < n \text{ then } [n..,d] \text{ else } \text{if } t < n+d \text{ then } [Suc\ t..,n+d - Suc\ t] \text{ else } \{\})$

**by** (*force simp: i-cut-mem-iff iT-iff*)

**corollary** *iIN-cut-greater1*:

$\llbracket t \in [n..,d]; t < n + d \rrbracket \implies$

$[n..,d] \downarrow_{>} t = [Suc\ t..,n + d - Suc\ t]$

**by** (*simp add: iIN-cut-greater iT-iff*)

**lemma** *mod-cut-greater-aux-t-less*:

$\llbracket 0 < (m::nat); r \leq t \rrbracket \implies$

$t < t + m - (t - r) \bmod m$

**by** (*simp add: less-add-diff add commute*)

**lemma** *mod-cut-greater-aux-le-x*:

$\llbracket (r::nat) \leq t; t < x; x \bmod m = r \bmod m \rrbracket \implies$

$t + m - (t - r) \bmod m \leq x$

```

apply (insert diff-mod-le[of t r m])
apply (subst diff-add-assoc2, simp)
apply (rule less-mod-eq-imp-add-divisor-le, simp)
apply (simp add: sub-diff-mod-eq)
done

```

```

lemma iMOD-cut-greater:
  [r, mod m] ↓> t = (
    if t < r then [r, mod m] else
    if m = 0 then {}
    else [t + m - (t - r) mod m, mod m])
apply (case-tac m = 0)
apply (simp add: iMOD-0 iIN-0 i-cut-singleton)
apply (case-tac t < r)
apply (simp add: iT-Min cut-greater-Min-all)
apply (simp add: linorder-not-less)
apply (simp add: set-eq-iff i-cut-mem-iff iT-iff, clarify)
apply (subgoal-tac (t - r) mod m ≤ t)
prefer 2
apply (rule order-trans[OF mod-le-dividend], simp)
apply (simp add: diff-add-assoc2 del: add-diff-assoc2)
apply (simp add: sub-diff-mod-eq del: add-diff-assoc2)
apply (rule conj-cong, simp)
apply (rule iffI)
apply clarify
apply (rule less-mod-eq-imp-add-divisor-le)
apply simp
apply (simp add: sub-diff-mod-eq)
apply (subgoal-tac t < x)
prefer 2
apply (rule-tac y=t - (t - r) mod m + m in order-less-le-trans)
apply (simp add: mod-cut-greater-aux-t-less)
apply simp+
done

```

```

lemma iMOD-cut-greater1:
  t ∈ [r, mod m] ⇒
  [r, mod m] ↓> t = (
    if m = 0 then {}
    else [t + m, mod m])
by (simp add: iMOD-cut-greater iT-iff mod-eq-imp-diff-mod-0)

```

```

lemma iMODb-cut-greater-aux:
  [ 0 < m; t < r + m * c; r ≤ t ] ⇒
  (r + m * c - (t + m - (t - r) mod m)) div m =
  c - Suc ((t - r) div m)
apply (subgoal-tac r ≤ t + m - (t - r) mod m)
prefer 2
apply (rule order-trans[of - t], simp)

```

**apply** (*simp add: mod-cut-greater-aux-t-less less-imp-le*)  
**apply** (*rule-tac t=(r + m \* c - (t + m - (t - r) mod m)) and s=m \* (c - Suc ((t - r) div m)) in subst*)  
**apply** (*simp add: diff-mult-distrib2 minus-mod-eq-mult-div [symmetric] del: diff-diff-left*)  
**apply simp**  
**done**

**lemma** *iMODb-cut-greater:*

$[r, \text{mod } m, c] \downarrow > t = ($   
   *if*  $t < r$  *then*  $[r, \text{mod } m, c]$  *else*  
   *if*  $r + m * c \leq t$  *then*  $\{\}$   
   *else*  $[t + m - (t - r) \text{ mod } m, \text{mod } m, c - \text{Suc } ((t-r) \text{ div } m)]$   
**apply** (*case-tac m = 0*)  
**apply** (*simp add: iMODb-mod-0 iIN-0 i-cut-singleton*)  
**apply** (*case-tac r + m \* c ≤ t*)  
**apply** (*simp add: cut-greater-Max-empty iT-Max iT-finite*)  
**apply** (*case-tac t < r*)  
**apply** (*simp add: cut-greater-Min-all iT-Min*)  
**apply** (*simp add: linorder-not-less linorder-not-le*)  
**apply** (*rule-tac t=[ r, mod m, c ] and s=[ r, mod m ] ∩ [...r + m \* c] in subst*)  
**apply** (*simp add: iMOD-iTILL-iMODb-conv*)  
**apply** (*simp add: i-cut-Int-left iMOD-cut-greater*)  
**apply** (*subst iMOD-iTILL-iMODb-conv*)  
**apply** (*rule mod-cut-greater-aux-le-x, simp+*)  
**apply** (*simp add: iMODb-cut-greater-aux*)  
**done**

**lemma** *iMODb-cut-greater1:*

$t \in [r, \text{mod } m, c] \implies$   
 $[r, \text{mod } m, c] \downarrow > t = ($   
   *if*  $r + m * c \leq t$  *then*  $\{\}$   
   *else*  $[t + m, \text{mod } m, c - \text{Suc } ((t-r) \text{ div } m)]$   
**by** (*simp add: iMODb-cut-greater iT-iff mod-eq-imp-diff-mod-0*)

**lemma** *iMOD-cut-ge:*

$[r, \text{mod } m] \downarrow \geq t = ($   
   *if*  $t \leq r$  *then*  $[r, \text{mod } m]$  *else*  
   *if*  $m = 0$  *then*  $\{\}$   
   *else*  $[t + m - \text{Suc } (t - \text{Suc } r) \text{ mod } m, \text{mod } m]$   
**apply** (*case-tac t = 0*)  
**apply** (*simp add: cut-ge-0-all*)  
**apply** (*force simp: nat-cut-greater-ge-conv[symmetric] iMOD-cut-greater*)  
**done**

**lemma** *iMOD-cut-ge1:*

$t \in [r, \text{mod } m] \implies$   
 $[r, \text{mod } m] \downarrow \geq t = [t, \text{mod } m]$

by (*fastforce simp: iMOD-cut-ge*)

**lemma** *iMODb-cut-ge*:

```

[r, mod m, c] ↓ ≥ t = (
  if t ≤ r then [r, mod m, c] else
  if r + m * c < t then {}
  else [t + m - Suc ((t - Suc r) mod m), mod m, c - (t + m - Suc r) div m])
apply (case-tac m = 0)
apply (simp add: iMODb-mod-0 iIN-0 i-cut-singleton)
apply (case-tac r + m * c < t)
apply (simp add: cut-ge-Max-empty iT-Max iT-finite)
apply (case-tac t ≤ r)
apply (simp add: cut-ge-Min-all iT-Min)
apply (simp add: linorder-not-less linorder-not-le)
apply (case-tac r mod m = t mod m)
apply (simp add: diff-mod-pred)
apply (simp add: mod-0-imp-diff-Suc-div-conv mod-eq-diff-mod-0-conv diff-add-assoc2
del: add-diff-assoc2)
apply (subgoal-tac 0 < (t - r) div m)
prefer 2
apply (frule-tac x=r in less-mod-eq-imp-add-divisor-le)
apply (simp add: mod-eq-diff-mod-0-conv)
apply (drule add-le-imp-le-diff2)
apply (drule-tac m=m and k=m in div-le-mono)
apply simp
apply (simp add: set-eq-iff i-cut-mem-iff iT-iff, intro allI)
apply (simp add: mod-eq-diff-mod-0-conv[symmetric])
apply (rule conj-cong, simp)
apply (case-tac t ≤ x)
prefer 2
apply simp
apply (simp add: diff-mult-distrib2 minus-mod-eq-mult-div [symmetric] mod-eq-diff-mod-0-conv
add commute[of r])
apply (subgoal-tac Suc ((t - Suc r) mod m) = (t - r) mod m)
prefer 2
apply (clarsimp simp add: diff-mod-pred mod-eq-diff-mod-0-conv)
apply (rule-tac t=[ r, mod m, c ] and s=[ r, mod m ] ∩ [...r + m * c] in subst)
apply (simp add: iMOD-iTILL-iMODb-conv)
apply (simp add: i-cut-Int-left iMOD-cut-ge)
apply (subst iMOD-iTILL-iMODb-conv)
apply (drule-tac x=t in le-imp-less-or-eq, erule disjE)
apply (rule mod-cut-greater-aux-le-x, simp+)
apply (rule arg-cong [where y=c - (t + m - Suc r) div m])
apply (drule-tac x=t in le-imp-less-or-eq, erule disjE)
prefer 2
apply simp
apply (simp add: iMODb-cut-greater-aux)

```

```

apply (rule arg-cong[where f=(-) c])
apply (simp add: diff-add-assoc2 del: add-diff-assoc2)
apply (rule-tac t=t - Suc r and s=t - r - Suc 0 in subst, simp)
apply (subst div-diff1-eq[of - Suc 0])
apply (case-tac m = Suc 0, simp)
apply simp
done

```

```

lemma iMODb-cut-ge1:
  t ∈ [r, mod m, c] ⇒
  [r, mod m, c] ↓ ≥ t = (
    if r + m * c < t then {}
    else [t, mod m, c - (t - r) div m])
apply (case-tac m = 0)
apply (simp add: iMODb-mod-0 iT-iff iIN-0 i-cut-singleton)
apply (clarsimp simp: iMODb-cut-ge iT-iff)
apply (simp add: mod-eq-imp-diff-mod-eq-divisor)
apply (rule-tac t=t + m - Suc r and s=t - r + (m - Suc 0) in subst, simp)
apply (subst div-add1-eq)
apply (simp add: mod-eq-imp-diff-mod-0)
done

```

```

lemma iMOD-0-cut-greater:
  t ∈ [r, mod 0] ⇒ [r, mod 0] ↓ > t = {}
by (simp add: iT-iff iMOD-0 iIN-0 i-cut-singleton)
lemma iMODb-0-cut-greater: t ∈ [r, mod 0, c] ⇒
  [r, mod 0, c] ↓ > t = {}
by (simp add: iT-iff iMODb-mod-0 iIN-0 i-cut-singleton)

```

```

lemmas iT-cut-ge =
  iTILL-cut-ge
  iFROM-cut-ge
  iIN-cut-ge
  iMOD-cut-ge
  iMODb-cut-ge

```

```

lemmas iT-cut-ge1 =
  iTILL-cut-ge1
  iFROM-cut-ge1
  iIN-cut-ge1
  iMOD-cut-ge1
  iMODb-cut-ge1

```

```

lemmas iT-cut-greater =
  iTILL-cut-greater
  iFROM-cut-greater
  iIN-cut-greater
  iMOD-cut-greater
  iMODb-cut-greater

```

**lemmas** *iT-cut-greater1* =  
*iTILL-cut-greater1*  
*iFROM-cut-greater1*  
*iIN-cut-greater1*  
*iMOD-cut-greater1*  
*iMODb-cut-greater1*

**lemmas** *iT-cut-ge-greater* =  
*iTILL-cut-ge*  
*iTILL-cut-greater*  
*iFROM-cut-ge*  
*iFROM-cut-greater*  
*iIN-cut-ge*  
*iIN-cut-greater*  
*iMOD-cut-ge*  
*iMOD-cut-greater*  
*iMODb-cut-ge*  
*iMODb-cut-greater*

**lemmas** *iT-cut-ge-greater1* =  
*iTILL-cut-ge1*  
*iTILL-cut-greater1*  
*iFROM-cut-ge1*  
*iFROM-cut-greater1*  
*iIN-cut-ge1*  
*iIN-cut-greater1*  
*iMOD-cut-ge1*  
*iMOD-cut-greater1*  
*iMODb-cut-ge1*  
*iMODb-cut-greater1*

## 1.6 Cardinality of intervals

**lemma** *iFROM-card*:  $\text{card } [n..] = 0$   
**by** (*simp add: iFROM-infinite*)

**lemma** *iTILL-card*:  $\text{card } [..n] = \text{Suc } n$   
**by** (*simp add: iTILL-def*)

**lemma** *iIN-card*:  $\text{card } [n..,d] = \text{Suc } d$   
**by** (*simp add: iIN-def*)

**lemma** *iMOD-0-card*:  $\text{card } [r, \text{mod } 0] = \text{Suc } 0$   
**by** (*simp add: iMOD-0 iIN-card*)

**lemma** *iMOD-card*:  $0 < m \implies \text{card } [r, \text{mod } m] = 0$   
**by** (*simp add: iMOD-infinite*)

**lemma** *iMOD-card-if*:  $\text{card } [r, \text{mod } m] = (\text{if } m = 0 \text{ then } \text{Suc } 0 \text{ else } 0)$

**by** (*simp add: iMOD-0-card iMOD-card*)

**lemma** *iMODb-mod-0-card*:  $\text{card } [r, \text{mod } 0, c] = \text{Suc } 0$   
**by** (*simp add: iMODb-mod-0 iIN-card*)

**lemma** *iMODb-card*:  $0 < m \implies \text{card } [r, \text{mod } m, c] = \text{Suc } c$   
**apply** (*induct c*)  
**apply** (*simp add: iMODb-0 iIN-card*)  
**apply** (*subst iMODb-Suc-insert-conv[symmetric]*)  
**apply** (*subst card-insert-disjoint*)  
**apply** (*simp add: iT-finite iT-iff*)  
**done**

**lemma** *iMODb-card-if*:  
 $\text{card } [r, \text{mod } m, c] = (\text{if } m = 0 \text{ then } \text{Suc } 0 \text{ else } \text{Suc } c)$   
**by** (*simp add: iMODb-mod-0-card iMODb-card*)

**lemmas** *iT-card* =  
*iFROM-card*  
*iTILL-card*  
*iIN-card*  
*iMOD-card-if*  
*iMODb-card-if*

Cardinality with *icard*

**lemma** *iFROM-icard*:  $\text{icard } [n..] = \infty$   
**by** (*simp add: iFROM-infinite*)

**lemma** *iTILL-icard*:  $\text{icard } [..n] = \text{enat } (\text{Suc } n)$   
**by** (*simp add: icard-finite iT-finite iT-card*)

**lemma** *iIN-icard*:  $\text{icard } [n..,d] = \text{enat } (\text{Suc } d)$   
**by** (*simp add: icard-finite iT-finite iT-card*)

**lemma** *iMOD-0-icard*:  $\text{icard } [r, \text{mod } 0] = \text{eSuc } 0$   
**by** (*simp add: icard-finite iT-finite iT-card eSuc-enat*)

**lemma** *iMOD-icard*:  $0 < m \implies \text{icard } [r, \text{mod } m] = \infty$   
**by** (*simp add: iMOD-infinite*)

**lemma** *iMOD-icard-if*:  $\text{icard } [r, \text{mod } m] = (\text{if } m = 0 \text{ then } \text{eSuc } 0 \text{ else } \infty)$   
**by** (*simp add: icard-finite iT-finite iT-infinite eSuc-enat iT-card*)

**lemma** *iMODb-mod-0-icard*:  $\text{icard } [r, \text{mod } 0, c] = \text{eSuc } 0$   
**by** (*simp add: icard-finite iT-finite eSuc-enat iT-card*)

**lemma** *iMODb-icard*:  $0 < m \implies \text{icard } [r, \text{mod } m, c] = \text{enat } (\text{Suc } c)$   
**by** (*simp add: icard-finite iT-finite iMODb-card*)

**lemma** *iMODb-icard-if*:  $\text{icard } [r, \text{mod } m, c] = \text{enat}$  (if  $m = 0$  then  $\text{Suc } 0$  else  $\text{Suc } c$ )

**by** (*simp add: icard-finite iT-finite iMODb-card-if*)

**lemmas** *iT-icard* =

*iFROM-icard*

*iTILL-icard*

*iIN-icard*

*iMOD-icard-if*

*iMODb-icard-if*

## 1.7 Functions *inext* and *iprev* with intervals

**lemma**

*iFROM-inext*:  $t \in [n..] \implies \text{inext } t [n..] = \text{Suc } t$  **and**

*iTILL-inext*:  $t < n \implies \text{inext } t [..n] = \text{Suc } t$  **and**

*iIN-inext*:  $\llbracket n \leq t; t < n + d \rrbracket \implies \text{inext } t [n..,d] = \text{Suc } t$

**by** (*simp add: iT-defs inext-atLeast inext-atMost inext-atLeastAtMost*)<sup>+</sup>

**lemma**

*iFROM-iprev'*:  $t \in [n..] \implies \text{iprev } (\text{Suc } t) [n..] = t$  **and**

*iFROM-iprev*:  $n < t \implies \text{iprev } t [n..] = t - \text{Suc } 0$  **and**

*iTILL-iprev*:  $t \in [..n] \implies \text{iprev } t [..n] = t - \text{Suc } 0$  **and**

*iIN-iprev*:  $\llbracket n < t; t \leq n + d \rrbracket \implies \text{iprev } t [n..,d] = t - \text{Suc } 0$  **and**

*iIN-iprev'*:  $\llbracket n \leq t; t < n + d \rrbracket \implies \text{iprev } (\text{Suc } t) [n..,d] = t$

**by** (*simp add: iT-defs iprev-atLeast iprev-atMost iprev-atLeastAtMost*)<sup>+</sup>

**lemma** *iMOD-inext*:  $t \in [r, \text{mod } m] \implies \text{inext } t [r, \text{mod } m] = t + m$

**by** (*clarsimp simp add: inext-def iMOD-cut-greater iT-iff iT-Min iT-not-empty mod-eq-imp-diff-mod-0*)

**lemma** *iMOD-iprev*:  $\llbracket t \in [r, \text{mod } m]; r < t \rrbracket \implies \text{iprev } t [r, \text{mod } m] = t - m$

**apply** (*case-tac m = 0, simp add: iMOD-iff*)

**apply** (*clarsimp simp add: iprev-def iMOD-cut-less iT-iff iT-Max iT-not-empty minus-mod-eq-mult-div [symmetric]*)

**apply** (*simp del: add-Suc-right add: add-Suc-right[symmetric] mod-eq-imp-diff-mod-eq-divisor*)

**apply** (*simp add: less-mod-eq-imp-add-divisor-le*)

**done**

**lemma** *iMOD-iprev'*:  $t \in [r, \text{mod } m] \implies \text{iprev } (t + m) [r, \text{mod } m] = t$

**apply** (*case-tac m = 0*)

**apply** (*simp add: iMOD-0 iIN-0 iprev-singleton*)

**apply** (*simp add: iMOD-iprev iT-iff*)

**done**

**lemma** *iMODb-inext*:

$\llbracket t \in [r, \text{mod } m, c]; t < r + m * c \rrbracket \implies$

$\text{inext } t [r, \text{mod } m, c] = t + m$

**by** (*clarsimp simp add: inext-def iMODb-cut-greater iT-iff iT-Min iT-not-empty mod-eq-imp-diff-mod-0*)

**lemma** *iMODb-iprev*:

$\llbracket t \in [r, \text{mod } m, c]; r < t \rrbracket \implies$   
 $\text{iprev } t [r, \text{mod } m, c] = t - m$

**apply** (*case-tac*  $m = 0$ , *simp add: iMODb-iff*)

**apply** (*clarsimp simp add: iprev-def iMODb-cut-less iT-iff iT-Max iT-not-empty*  
*minus-mod-eq-mult-div [symmetric]*)

**apply** (*simp del: add-Suc-right add: add-Suc-right[symmetric] mod-eq-imp-diff-mod-eq-divisor*)

**apply** (*simp add: less-mod-eq-imp-add-divisor-le*)

**done**

**lemma** *iMODb-iprev'*:

$\llbracket t \in [r, \text{mod } m, c]; t < r + m * c \rrbracket \implies$   
 $\text{iprev } (t + m) [r, \text{mod } m, c] = t$

**apply** (*case-tac*  $m = 0$ )

**apply** (*simp add: iMODb-mod-0 iIN-0 iprev-singleton*)

**apply** (*simp add: iMODb-iprev iT-iff less-mod-eq-imp-add-divisor-le*)

**done**

**lemmas** *iT-inext* =

*iFROM-inext*

*iTILL-inext*

*iIN-inext*

*iMOD-inext*

*iMODb-inext*

**lemmas** *iT-iprev* =

*iFROM-iprev'*

*iFROM-iprev*

*iTILL-iprev*

*iIN-iprev*

*iIN-iprev'*

*iMOD-iprev*

*iMOD-iprev'*

*iMODb-iprev*

*iMODb-iprev'*

**lemma** *iFROM-inext-if*:

$\text{inext } t [n..] = (\text{if } t \in [n..] \text{ then } \text{Suc } t \text{ else } t)$

**by** (*simp add: iFROM-inext not-in-inext-fix*)

**lemma** *iTILL-inext-if*:

$\text{inext } t [..n] = (\text{if } t < n \text{ then } \text{Suc } t \text{ else } t)$

**by** (*simp add: iTILL-inext iT-finite iT-Max inext-ge-Max*)

**lemma** *iIN-inext-if*:

$\text{inext } t [n..d] = (\text{if } n \leq t \wedge t < n + d \text{ then } \text{Suc } t \text{ else } t)$

**by** (*fastforce simp: iIN-inext iT-iff not-in-inext-fix iT-finite iT-Max inext-ge-Max*)

**lemma** *iMOD-inext-if*:

$\text{inext } t [r, \text{mod } m] = (\text{if } t \in [r, \text{mod } m] \text{ then } t + m \text{ else } t)$   
**by** (*simp add: iMOD-inext not-in-inext-fix*)

**lemma** *iMODb-inext-if*:

$\text{inext } t [r, \text{mod } m, c] =$   
 $(\text{if } t \in [r, \text{mod } m, c] \wedge t < r + m * c \text{ then } t + m \text{ else } t)$   
**by** (*fastforce simp: iMODb-inext iT-iff not-in-inext-fix iT-finite iT-Max inext-ge-Max*)

**lemmas** *iT-inext-if* =

*iFROM-inext-if*  
*iTILL-inext-if*  
*iIN-inext-if*  
*iMOD-inext-if*  
*iMODb-inext-if*

**lemma** *iFROM-iprev-if*:

$\text{iprev } t [n..] = (\text{if } n < t \text{ then } t - \text{Suc } 0 \text{ else } t)$   
**by** (*simp add: iFROM-iprev iT-Min iprev-le-iMin*)

**lemma** *iTILL-iprev-if*:

$\text{iprev } t [..n] = (\text{if } t \in [..n] \text{ then } t - \text{Suc } 0 \text{ else } t)$   
**by** (*simp add: iTILL-iprev not-in-iprev-fix*)

**lemma** *iIN-iprev-if*:

$\text{iprev } t [n..d] = (\text{if } n < t \wedge t \leq n + d \text{ then } t - \text{Suc } 0 \text{ else } t)$   
**by** (*fastforce simp: iIN-iprev iT-iff not-in-iprev-fix iT-Min iprev-le-iMin*)

**lemma** *iMOD-iprev-if*:

$\text{iprev } t [r, \text{mod } m] =$   
 $(\text{if } t \in [r, \text{mod } m] \wedge r < t \text{ then } t - m \text{ else } t)$   
**by** (*fastforce simp add: iMOD-iprev iT-iff not-in-iprev-fix iT-Min iprev-le-iMin*)

**lemma** *iMODb-iprev-if*:

$\text{iprev } t [r, \text{mod } m, c] =$   
 $(\text{if } t \in [r, \text{mod } m, c] \wedge r < t \text{ then } t - m \text{ else } t)$   
**by** (*fastforce simp add: iMODb-iprev iT-iff not-in-iprev-fix iT-Min iprev-le-iMin*)

**lemmas** *iT-iprev-if* =

*iFROM-iprev-if*  
*iTILL-iprev-if*  
*iIN-iprev-if*  
*iMOD-iprev-if*  
*iMODb-iprev-if*

The difference between an element and the next/previous element is constant if the element is different from Min/Max of the interval

**lemma** *iFROM-inext-diff-const*:

$t \in [n..] \implies \text{inext } t [n..] - t = \text{Suc } 0$   
**by** (*simp add: iFROM-inext*)

**lemma** *iFROM-iprev-diff-const*:

$n < t \implies t - \text{iprev } t [n..] = \text{Suc } 0$   
**by** (*simp add: iFROM-iprev*)

**lemma** *iFROM-iprev-diff-const'*:

$$t \in [n..] \implies \text{Suc } t - \text{iprev } (\text{Suc } t) [n..] = \text{Suc } 0$$

**by** (*simp add: iFROM-iprev'*)

**lemma** *iTILL-inext-diff-const*:

$$t < n \implies \text{inext } t [..n] - t = \text{Suc } 0$$

**by** (*simp add: iTILL-inext*)

**lemma** *iTILL-iprev-diff-const*:

$$\llbracket t \in [..n]; 0 < t \rrbracket \implies t - \text{iprev } t [..n] = \text{Suc } 0$$

**by** (*simp add: iTILL-iprev*)

**lemma** *iIN-inext-diff-const*:

$$\llbracket n \leq t; t < n + d \rrbracket \implies \text{inext } t [n..,d] - t = \text{Suc } 0$$

**by** (*simp add: iIN-inext*)

**lemma** *iIN-iprev-diff-const*:

$$\llbracket n < t; t \leq n + d \rrbracket \implies t - \text{iprev } t [n..,d] = \text{Suc } 0$$

**by** (*simp add: iIN-iprev*)

**lemma** *iIN-iprev-diff-const'*:

$$\llbracket n \leq t; t < n + d \rrbracket \implies \text{Suc } t - \text{iprev } (\text{Suc } t) [n..,d] = \text{Suc } 0$$

**by** (*simp add: iIN-iprev*)

**lemma** *iMOD-inext-diff-const*:

$$t \in [r, \text{mod } m] \implies \text{inext } t [r, \text{mod } m] - t = m$$

**by** (*simp add: iMOD-inext*)

**lemma** *iMOD-iprev-diff-const'*:

$$t \in [r, \text{mod } m] \implies (t + m) - \text{iprev } (t + m) [r, \text{mod } m] = m$$

**by** (*simp add: iMOD-iprev'*)

**lemma** *iMOD-iprev-diff-const*:

$$\llbracket t \in [r, \text{mod } m]; r < t \rrbracket \implies t - \text{iprev } t [r, \text{mod } m] = m$$

**apply** (*simp add: iMOD-iprev iT-iff*)

**apply** (*drule less-mod-eq-imp-add-divisor-le[where m=m], simp+*)

**done**

**lemma** *iMODb-inext-diff-const*:

$$\llbracket t \in [r, \text{mod } m, c]; t < r + m * c \rrbracket \implies \text{inext } t [r, \text{mod } m, c] - t = m$$

**by** (*simp add: iMODb-inext*)

**lemma** *iMODb-iprev-diff-const'*:

$$\llbracket t \in [r, \text{mod } m, c]; t < r + m * c \rrbracket \implies (t + m) - \text{iprev } (t + m) [r, \text{mod } m, c] = m$$

**by** (*simp add: iMODb-iprev'*)

**lemma** *iMODb-iprev-diff-const*:

$$\llbracket t \in [r, \text{mod } m, c]; r < t \rrbracket \implies t - \text{iprev } t [r, \text{mod } m, c] = m$$

**apply** (*simp add: iMODb-iprev iT-iff*)

**apply** (*drule less-mod-eq-imp-add-divisor-le*[**where**  $m=m$ ], *simp+*)  
**done**

**lemmas** *iT-inext-diff-const* =  
*iFROM-inext-diff-const*  
*iTILL-inext-diff-const*  
*iIN-inext-diff-const*  
*iMOD-inext-diff-const*  
*iMODb-inext-diff-const*

**lemmas** *iT-iprev-diff-const* =  
*iFROM-iprev-diff-const*  
*iFROM-iprev-diff-const'*  
*iTILL-iprev-diff-const*  
*iIN-iprev-diff-const*  
*iIN-iprev-diff-const'*  
*iMOD-iprev-diff-const'*  
*iMOD-iprev-diff-const*  
*iMODb-iprev-diff-const'*  
*iMODb-iprev-diff-const*

### 1.7.1 Mirroring of intervals

**lemma**

*iIN-mirror-elem*: *mirror-elem*  $x [n..d] = n + n + d - x$  **and**  
*iTILL-mirror-elem*: *mirror-elem*  $x [..n] = n - x$  **and**  
*iMODb-mirror-elem*: *mirror-elem*  $x [r, \text{mod } m, c] = r + r + m * c - x$   
**by** (*simp add: mirror-elem-def nat-mirror-def iT-Min iT-Max*)**+**

**lemma** *iMODb-imirror-bounds*:

$r' + m' * c' \leq l + r \implies$   
*imirror-bounds*  $[r', \text{mod } m', c'] l r = [l + r - r' - m' * c', \text{mod } m', c']$   
**apply** (*clarsimp simp: set-eq-iff Bex-def imirror-bounds-iff iT-iff*)  
**apply** (*frule diff-le-mono*[*of - - r'*], *simp*)  
**apply** (*simp add: mod-diff-right-eq*)  
**apply** (*rule iffI*)  
**apply** (*clarsimp, rename-tac x'*)  
**apply** (*rule-tac a=x' in ssubst*[*OF mod-diff-right-eq, rule-format*], *simp+*)  
**apply** (*simp add: diff-le-mono2*)  
**apply** *clarsimp*  
**apply** (*rule-tac x=l+r-x in exI*)  
**apply** (*simp add: le-diff-swap*)  
**apply** (*simp add: le-diff-conv2*)  
**apply** (*subst mod-sub-eq-mod-swap, simp+*)  
**apply** (*simp add: mod-diff-right-eq*)  
**done**

**lemma** *iIN-imirror-bounds*:

$n + d \leq l + r \implies$  *imirror-bounds*  $[n..d] l r = [l + r - n - d..d]$   
**apply** (*insert iMODb-imirror-bounds*[*of n Suc 0 d l r*])

**apply** (*simp add: iMODb-mod-1*)  
**done**

**lemma** *iTILL-imirror-bounds*:

$n \leq l + r \implies \text{imirror-bounds } [..n] \ l \ r = [l + r - n..n]$   
**apply** (*insert iIN-imirror-bounds[of 0 n l r]*)  
**apply** (*simp add: iIN-0-iTILL-conv*)  
**done**

**lemmas** *iT-imirror-bounds =*

*iTILL-imirror-bounds*  
*iIN-imirror-bounds*  
*iMODb-imirror-bounds*

**lemma** *iMODb-imirror-ident*:  $\text{imirror } [r, \text{mod } m, c] = [r, \text{mod } m, c]$

**by** (*simp add: imirror-eq-imirror-bounds iMODb-Min iMODb-Max iMODb-imirror-bounds*)

**lemma** *iIN-imirror-ident*:  $\text{imirror } [n..d] = [n..d]$

**by** (*simp add: iMODb-mod-1[symmetric] iMODb-imirror-ident*)

**lemma** *iTILL-imirror-ident*:  $\text{imirror } [..n] = [..n]$

**by** (*simp add: iIN-0-iTILL-conv[symmetric] iIN-imirror-ident*)

**lemmas** *iT-imirror-ident =*

*iTILL-imirror-ident*  
*iIN-imirror-ident*  
*iMODb-imirror-ident*

### 1.7.2 Functions *inext-nth* and *iprev-nth* on intervals

**lemma** *iFROM-inext-nth* :  $[n..] \rightarrow a = n + a$

**by** (*simp add: iT-defs inext-nth-atLeast*)

**lemma** *iIN-inext-nth* :  $a \leq d \implies [n..d] \rightarrow a = n + a$

**by** (*simp add: iT-defs inext-nth-atLeastAtMost*)

**lemma** *iIN-iprev-nth*:  $a \leq d \implies [n..d] \leftarrow a = n + d - a$

**by** (*simp add: iT-defs iprev-nth-atLeastAtMost*)

**lemma** *iIN-inext-nth-if* :

$[n..d] \rightarrow a = (\text{if } a \leq d \text{ then } n + a \text{ else } n + d)$

**by** (*simp add: iIN-inext-nth inext-nth-card-Max iT-finite iT-not-empty iT-Max iT-card*)

**lemma** *iIN-iprev-nth-if*:

$[n..d] \leftarrow a = (\text{if } a \leq d \text{ then } n + d - a \text{ else } n)$

**by** (*simp add: iIN-iprev-nth iprev-nth-card-iMin iT-finite iT-not-empty iT-Min iT-card*)

**lemma** *iTILL-inext-nth* :  $a \leq n \implies [..n] \rightarrow a = a$

**by** (*simp add: iTILL-def inext-nth-atMost*)

**lemma** *iTILL-inext-nth-if* :  
 $[\dots n] \rightarrow a = (\text{if } a \leq n \text{ then } a \text{ else } n)$   
**by** (*insert iIN-inext-nth-if*[of 0 n a], *simp add: iIN-0-iTILL-conv*)

**lemma** *iTILL-iprev-nth*:  $a \leq n \implies [\dots n] \leftarrow a = n - a$   
**by** (*simp add: iTILL-def iprev-nth-atMost*)

**lemma** *iTILL-iprev-nth-if*:  
 $[\dots n] \leftarrow a = (\text{if } a \leq n \text{ then } n - a \text{ else } 0)$   
**by** (*insert iIN-iprev-nth-if*[of 0 n a], *simp add: iIN-0-iTILL-conv*)

**lemma** *iMOD-inext-nth*:  $[r, \text{mod } m] \rightarrow a = r + m * a$   
**apply** (*induct a*)  
**apply** (*simp add: iT-Min*)  
**apply** (*simp add: iMOD-inext-if iT-iff*)  
**done**

**lemma** *iMODb-inext-nth*:  $a \leq c \implies [r, \text{mod } m, c] \rightarrow a = r + m * a$   
**apply** (*case-tac m = 0*)  
**apply** (*simp add: iMODb-mod-0 iIN-0 inext-nth-singleton*)  
**apply** (*induct a*)  
**apply** (*simp add: iMODb-Min*)  
**apply** (*simp add: iMODb-inext-if iT-iff*)  
**done**

**lemma** *iMODb-inext-nth-if*:  
 $[r, \text{mod } m, c] \rightarrow a = (\text{if } a \leq c \text{ then } r + m * a \text{ else } r + m * c)$   
**by** (*simp add: iMODb-inext-nth inext-nth-card-Max iT-finite iT-not-empty iT-Max iT-card*)

**lemma** *iMODb-iprev-nth*:  
 $a \leq c \implies [r, \text{mod } m, c] \leftarrow a = r + m * (c - a)$   
**apply** (*case-tac m = 0*)  
**apply** (*simp add: iMODb-mod-0 iIN-0 iprev-nth-singleton*)  
**apply** (*induct a*)  
**apply** (*simp add: iMODb-Max*)  
**apply** (*simp add: iMODb-iprev-if iT-iff*)  
**apply** (*frule mult-left-mono*[of - - m], *simp*)  
**apply** (*simp add: diff-mult-distrib2*)  
**done**

**lemma** *iMODb-iprev-nth-if*:  
 $[r, \text{mod } m, c] \leftarrow a = (\text{if } a \leq c \text{ then } r + m * (c - a) \text{ else } r)$   
**by** (*simp add: iMODb-iprev-nth iprev-nth-card-iMin iT-finite iT-not-empty iT-Min iT-card*)

**lemma** *iIN-iFROM-inext-nth*:

$$a \leq d \implies [n..d] \rightarrow a = [n..] \rightarrow a$$

**by** (*simp add: iIN-inext-nth iFROM-inext-nth*)

**lemma** *iIN-iFROM-inext*:

$$a < n + d \implies \text{inext } a [n..d] = \text{inext } a [n..]$$

**by** (*simp add: iT-inext-if iT-iff*)

**lemma** *iMOD-iMODb-inext-nth*:

$$a \leq c \implies [r, \text{mod } m, c] \rightarrow a = [r, \text{mod } m] \rightarrow a$$

**by** (*simp add: iMOD-inext-nth iMODb-inext-nth*)

**lemma** *iMOD-iMODb-inext*:

$$a < r + m * c \implies \text{inext } a [r, \text{mod } m, c] = \text{inext } a [r, \text{mod } m]$$

**by** (*simp add: iT-inext-if iT-iff*)

**lemma** *iMOD-inext-nth-Suc-diff*:

$$([r, \text{mod } m] \rightarrow (\text{Suc } n)) - ([r, \text{mod } m] \rightarrow n) = m$$

**by** (*simp add: iMOD-inext-nth del: inext-nth.simps*)

**lemma** *iMOD-inext-nth-diff*:

$$([r, \text{mod } m] \rightarrow a) - ([r, \text{mod } m] \rightarrow b) = (a - b) * m$$

**by** (*simp add: iMOD-inext-nth diff-mult-distrib mult.commute[of m]*)

**lemma** *iMODb-inext-nth-diff*:  $\llbracket a \leq c; b \leq c \rrbracket \implies$

$$([r, \text{mod } m, c] \rightarrow a) - ([r, \text{mod } m, c] \rightarrow b) = (a - b) * m$$

**by** (*simp add: iMODb-inext-nth diff-mult-distrib mult.commute[of m]*)

## 1.8 Induction with intervals

**lemma** *iFROM-induct*:

$$\llbracket P n; \bigwedge k. \llbracket k \in [n..]; P k \rrbracket \implies P (\text{Suc } k); a \in [n..] \rrbracket \implies P a$$

**apply** (*rule inext-induct[of - [n..]]*)

**apply** (*simp add: iT-Min iT-inext-if*)

**done**

**lemma** *iIN-induct*:

$$\llbracket P n; \bigwedge k. \llbracket k \in [n..d]; k \neq n + d; P k \rrbracket \implies P (\text{Suc } k); a \in [n..d] \rrbracket \implies P a$$

**apply** (*rule inext-induct[of - [n..d]]*)

**apply** (*simp add: iT-Min iT-inext-if*)

**done**

**lemma** *iTILL-induct*:

$$\llbracket P 0; \bigwedge k. \llbracket k \in [..n]; k \neq n; P k \rrbracket \implies P (\text{Suc } k); a \in [..n] \rrbracket \implies P a$$

**apply** (*rule inext-induct[of - [..n]]*)

**apply** (*simp add: iT-Min iT-inext-if*)

**done**

**lemma** *iMOD-induct*:

```

[[ P r;  $\bigwedge k. [k \in [r, \text{mod } m]; P k] \implies P (k + m); a \in [r, \text{mod } m] ] \implies P a
apply (rule inext-induct[of - [r, mod m]])
apply (simp add: iT-Min iT-inext-if)+
done$ 
```

**lemma** *iMODb-induct*:

```

[[ P r;  $\bigwedge k. [k \in [r, \text{mod } m, c]; k \neq r + m * c; P k] \implies P (k + m); a \in [r, \text{mod } m, c] ] \implies P a
apply (rule inext-induct[of - [r, mod m, c]])
apply (simp add: iT-Min iT-inext-if)+
done$ 
```

**lemma** *iIN-rev-induct*:

```

[[ P (n + d);  $\bigwedge k. [k \in [n \dots, d]; k \neq n; P k] \implies P (k - \text{Suc } 0); a \in [n \dots, d] ] \implies P a
apply (rule iprev-induct[of - [n \dots, d]])
apply (simp add: iT-Max iT-finite iT-iprev-if)+
done$ 
```

**lemma** *iTILL-rev-induct*:

```

[[ P n;  $\bigwedge k. [k \in [\dots, n]; 0 < k; P k] \implies P (k - \text{Suc } 0); a \in [\dots, n] ] \implies P a
apply (rule iprev-induct[of - [\dots, n]])
apply (fastforce simp: iT-Max iT-finite iT-iprev-if)+
done$ 
```

**lemma** *iMODb-rev-induct*:

```

[[ P (r + m * c);  $\bigwedge k. [k \in [r, \text{mod } m, c]; k \neq r; P k] \implies P (k - m); a \in [r, \text{mod } m, c] ] \implies P a
apply (rule iprev-induct[of - [r, mod m, c]])
apply (simp add: iT-Max iT-finite iT-iprev-if)+
done$ 
```

**end**

## 2 Arithmetic operators on natural intervals

**theory** *IL-IntervalOperators*

**imports** *IL-Interval*

**begin**

### 2.1 Arithmetic operations with intervals

#### 2.1.1 Addition of and multiplication by constants

**definition** *iT-Plus* :: *iT*  $\Rightarrow$  *Time*  $\Rightarrow$  *iT* (**infixl**  $\langle \oplus \rangle$  55)  
 where  $I \oplus k \equiv (\lambda n. (n + k)) ' I$

**definition** *iT-Mult* :: *iT*  $\Rightarrow$  *Time*  $\Rightarrow$  *iT* (**infixl**  $\langle \otimes \rangle$  55)

**where** *iT-Mult-def* :  $I \otimes k \equiv (\lambda n.(n * k)) \text{ ' } I$

**lemma** *iT-Plus-image-conv*:  $I \oplus k = (\lambda n.(n + k)) \text{ ' } I$   
**by** (*simp add: iT-Plus-def*)

**lemma** *iT-Mult-image-conv*:  $I \otimes k = (\lambda n.(n * k)) \text{ ' } I$   
**by** (*simp add: iT-Mult-def*)

**lemma** *iT-Plus-empty*:  $\{\} \oplus k = \{\}$   
**by** (*simp add: iT-Plus-def*)

**lemma** *iT-Mult-empty*:  $\{\} \otimes k = \{\}$   
**by** (*simp add: iT-Mult-def*)

**lemma** *iT-Plus-not-empty*:  $I \neq \{\} \implies I \oplus k \neq \{\}$   
**by** (*simp add: iT-Plus-def*)

**lemma** *iT-Mult-not-empty*:  $I \neq \{\} \implies I \otimes k \neq \{\}$   
**by** (*simp add: iT-Mult-def*)

**lemma** *iT-Plus-empty-iff*:  $(I \oplus k = \{\}) = (I = \{\})$   
**by** (*simp add: iT-Plus-def*)

**lemma** *iT-Mult-empty-iff*:  $(I \otimes k = \{\}) = (I = \{\})$   
**by** (*simp add: iT-Mult-def*)

**lemma** *iT-Plus-mono*:  $A \subseteq B \implies A \oplus k \subseteq B \oplus k$   
**by** (*simp add: iT-Plus-def image-mono*)

**lemma** *iT-Mult-mono*:  $A \subseteq B \implies A \otimes k \subseteq B \otimes k$   
**by** (*simp add: iT-Mult-def image-mono*)

**lemma** *iT-Mult-0*:  $I \neq \{\} \implies I \otimes 0 = [\dots 0]$   
**by** (*fastforce simp add: iTILL-def iT-Mult-def*)

**corollary** *iT-Mult-0-if*:  $I \otimes 0 = (\text{if } I = \{\} \text{ then } \{\} \text{ else } [\dots 0])$   
**by** (*simp add: iT-Mult-empty iT-Mult-0*)

**lemma** *iT-Plus-mem-iff*:  $x \in (I \oplus k) = (k \leq x \wedge (x - k) \in I)$   
**apply** (*simp add: iT-Plus-def image-iff*)  
**apply** (*rule iffI*)  
**apply** *fastforce*  
**apply** (*rule-tac x=x - k in beXI, simp+*)  
**done**

**lemma** *iT-Plus-mem-iff2*:  $x + k \in (I \oplus k) = (x \in I)$

**by** (*simp add: iT-Plus-def image-iff*)

**lemma** *iT-Mult-mem-iff-0*:  $x \in (I \otimes 0) = (I \neq \{\}) \wedge x = 0$   
**apply** (*case-tac I = \{\}*)  
**apply** (*simp add: iT-Mult-empty*)  
**apply** (*simp add: iT-Mult-0 iT-iff*)  
**done**

**lemma** *iT-Mult-mem-iff*:  
 $0 < k \implies x \in (I \otimes k) = (x \bmod k = 0 \wedge x \operatorname{div} k \in I)$   
**by** (*fastforce simp: iT-Mult-def image-iff*)

**lemma** *iT-Mult-mem-iff2*:  $0 < k \implies x * k \in (I \otimes k) = (x \in I)$   
**by** (*simp add: iT-Mult-def image-iff*)

**lemma** *iT-Plus-singleton*:  $\{a\} \oplus k = \{a + k\}$   
**by** (*simp add: iT-Plus-def*)

**lemma** *iT-Mult-singleton*:  $\{a\} \otimes k = \{a * k\}$   
**by** (*simp add: iT-Mult-def*)

**lemma** *iT-Plus-Un*:  $(A \cup B) \oplus k = (A \oplus k) \cup (B \oplus k)$   
**by** (*simp add: iT-Plus-def image-Un*)

**lemma** *iT-Mult-Un*:  $(A \cup B) \otimes k = (A \otimes k) \cup (B \otimes k)$   
**by** (*simp add: iT-Mult-def image-Un*)

**lemma** *iT-Plus-Int*:  $(A \cap B) \oplus k = (A \oplus k) \cap (B \oplus k)$   
**by** (*simp add: iT-Plus-def image-Int*)

**lemma** *iT-Mult-Int*:  $0 < k \implies (A \cap B) \otimes k = (A \otimes k) \cap (B \otimes k)$   
**by** (*simp add: iT-Mult-def image-Int mult-right-inj*)

**lemma** *iT-Plus-image*:  $f ' I \oplus n = (\lambda x. f x + n) ' I$   
**by** (*fastforce simp: iT-Plus-def*)

**lemma** *iT-Mult-image*:  $f ' I \otimes n = (\lambda x. f x * n) ' I$   
**by** (*fastforce simp: iT-Mult-def*)

**lemma** *iT-Plus-commute*:  $I \oplus a \oplus b = I \oplus b \oplus a$   
**by** (*fastforce simp: iT-Plus-def*)

**lemma** *iT-Mult-commute*:  $I \otimes a \otimes b = I \otimes b \otimes a$   
**by** (*fastforce simp: iT-Mult-def*)

**lemma** *iT-Plus-assoc*:  $I \oplus a \oplus b = I \oplus (a + b)$   
**by** (*fastforce simp: iT-Plus-def*)

**lemma** *iT-Mult-assoc*:  $I \otimes a \otimes b = I \otimes (a * b)$   
**by** (*fastforce simp: iT-Mult-def*)

**lemma** *iT-Plus-Mult-distrib*:  $I \oplus n \otimes m = I \otimes m \oplus n * m$   
**by** (*simp add: iT-Plus-def iT-Mult-def image-image add-mult-distrib*)

**lemma** *iT-Plus-finite-iff*:  $\text{finite } (I \oplus k) = \text{finite } I$   
**by** (*simp add: iT-Plus-def inj-on-finite-image-iff*)

**lemma** *iT-Mult-0-finite*:  $\text{finite } (I \otimes 0)$   
**by** (*simp add: iT-Mult-0-if iTILL-0*)

**lemma** *iT-Mult-finite-iff*:  $0 < k \implies \text{finite } (I \otimes k) = \text{finite } I$   
**by** (*simp add: iT-Mult-def inj-on-finite-image-iff[OF inj-imp-inj-on] mult-right-inj*)

**lemma** *iT-Plus-Min*:  $I \neq \{\} \implies iMin (I \oplus k) = iMin I + k$   
**by** (*simp add: iT-Plus-def iMin-mono2 mono-def*)

**lemma** *iT-Mult-Min*:  $I \neq \{\} \implies iMin (I \otimes k) = iMin I * k$   
**by** (*simp add: iT-Mult-def iMin-mono2 mono-def*)

**lemma** *iT-Plus-Max*:  $\llbracket \text{finite } I; I \neq \{\} \rrbracket \implies Max (I \oplus k) = Max I + k$   
**by** (*simp add: iT-Plus-def Max-mono2 mono-def*)

**lemma** *iT-Mult-Max*:  $\llbracket \text{finite } I; I \neq \{\} \rrbracket \implies Max (I \otimes k) = Max I * k$   
**by** (*simp add: iT-Mult-def Max-mono2 mono-def*)

**corollary**

*iMOD-mult-0*:  $[r, \text{mod } m] \otimes 0 = [\dots 0]$  **and**  
*iMODb-mult-0*:  $[r, \text{mod } m, c] \otimes 0 = [\dots 0]$  **and**  
*iFROM-mult-0*:  $[n \dots] \otimes 0 = [\dots 0]$  **and**  
*iIN-mult-0*:  $[n \dots, d] \otimes 0 = [\dots 0]$  **and**  
*iTILL-mult-0*:  $[\dots n] \otimes 0 = [\dots 0]$   
**by** (*simp add: iT-not-empty iT-Mult-0*)+

**lemmas** *iT-mult-0* =  
*iTILL-mult-0*  
*iFROM-mult-0*  
*iIN-mult-0*  
*iMOD-mult-0*  
*iMODb-mult-0*

**lemma** *iT-Plus-0*:  $I \oplus 0 = I$   
**by** (*simp add: iT-Plus-def*)

**lemma** *iT-Mult-1*:  $I \otimes \text{Suc } 0 = I$   
**by** (*simp add: iT-Mult-def*)

**corollary**

*iFROM-add-Min*:  $iMin ([n..] \oplus k) = n + k$  **and**  
*iIN-add-Min*:  $iMin ([n..,d] \oplus k) = n + k$  **and**  
*iTILL-add-Min*:  $iMin ([..n] \oplus k) = k$  **and**  
*iMOD-add-Min*:  $iMin ([r, \text{mod } m] \oplus k) = r + k$  **and**  
*iMODb-add-Min*:  $iMin ([r, \text{mod } m, c] \oplus k) = r + k$   
**by** (*simp add: iT-not-empty iT-Plus-Min iT-Min*)+

**corollary**

*iFROM-mult-Min*:  $iMin ([n..] \otimes k) = n * k$  **and**  
*iIN-mult-Min*:  $iMin ([n..,d] \otimes k) = n * k$  **and**  
*iTILL-mult-Min*:  $iMin ([..n] \otimes k) = 0$  **and**  
*iMOD-mult-Min*:  $iMin ([r, \text{mod } m] \otimes k) = r * k$  **and**  
*iMODb-mult-Min*:  $iMin ([r, \text{mod } m, c] \otimes k) = r * k$   
**by** (*simp add: iT-not-empty iT-Mult-Min iT-Min*)+

**lemmas** *iT-add-Min* =

*iIN-add-Min*  
*iTILL-add-Min*  
*iFROM-add-Min*  
*iMOD-add-Min*  
*iMODb-add-Min*

**lemmas** *iT-mult-Min* =

*iIN-mult-Min*  
*iTILL-mult-Min*  
*iFROM-mult-Min*  
*iMOD-mult-Min*  
*iMODb-mult-Min*

**lemma** *iFROM-add*:  $[n..] \oplus k = [n+k..]$ 

**by** (*simp add: iFROM-def iT-Plus-def image-add-atLeast*)

**lemma** *iIN-add*:  $[n..,d] \oplus k = [n+k..,d]$ 

**by** (*fastforce simp add: iIN-def iT-Plus-def*)

**lemma** *iTILL-add*:  $[..i] \oplus k = [k..,i]$ 

**by** (*simp add: iIN-0-iTILL-conv[symmetric] iIN-add*)

**lemma** *iMOD-add*:  $[r, \text{mod } m] \oplus k = [r + k, \text{mod } m]$ 

**apply** (*clarsimp simp: set-eq-iff iMOD-def iT-Plus-def image-iff*)

**apply** (*rule iffI*)

**apply** (*clarsimp simp: mod-add*)

**apply** (*rule-tac x=x - k in exI*)

**apply** *clarsimp*

**apply** (*simp add: mod-sub-add*)

**done**

**lemma** *iMODb-add*:  $[r, \text{mod } m, c] \oplus k = [r + k, \text{mod } m, c]$   
**by** (*simp add: iMODb-iMOD-iIN-conv iT-Plus-Int iMOD-add iIN-add*)

**lemmas** *iT-add* =  
*iMOD-add*  
*iMODb-add*  
*iFROM-add*  
*iIN-add*  
*iTILL-add*  
*iT-Plus-singleton*

**lemma** *iFROM-mult*:  $[n..] \otimes k = [n * k, \text{mod } k]$   
**apply** (*case-tac k = 0*)  
**apply** (*simp add: iMOD-0 iT-mult-0 iIN-0 iTILL-0*)  
**apply** (*clarsimp simp: set-eq-iff iT-Mult-mem-iff iT-iff*)  
**apply** (*rule conj-cong, simp*)  
**apply** (*rule iffI*)  
**apply** (*drule mult-le-mono1[of - - k]*)  
**apply** (*rule order-trans, assumption*)  
**apply** (*simp add: div-mult-cancel*)  
**apply** (*drule div-le-mono[of - - k]*)  
**apply** *simp*  
**done**

**lemma** *iIN-mult*:  $[n..,d] \otimes k = [n * k, \text{mod } k, d]$   
**apply** (*case-tac k = 0*)  
**apply** (*simp add: iMODb-mod-0 iT-mult-0 iIN-0 iTILL-0*)  
**apply** (*clarsimp simp: set-eq-iff iT-Mult-mem-iff iT-iff*)  
**apply** (*rule conj-cong, simp*)  
**apply** (*rule iffI*)  
**apply** (*elim conjE*)  
**apply** (*drule mult-le-mono1[of - - k], drule mult-le-mono1[of - - k]*)  
**apply** (*rule conjI*)  
**apply** (*rule order-trans, assumption*)  
**apply** (*simp add: div-mult-cancel*)  
**apply** (*simp add: div-mult-cancel add-mult-distrib mult.commute[of k]*)  
**apply** (*erule conjE*)  
**apply** (*drule div-le-mono[of - - k], drule div-le-mono[of - - k]*)  
**apply** *simp*  
**done**

**lemma** *iTILL-mult*:  $[..n] \otimes k = [0, \text{mod } k, n]$   
**by** (*simp add: iIN-0-iTILL-conv[symmetric] iIN-mult*)

**lemma** *iMOD-mult*:  $[r, \text{mod } m] \otimes k = [r * k, \text{mod } m * k]$   
**apply** (*case-tac k = 0*)  
**apply** (*simp add: iMOD-0 iT-mult-0 iIN-0 iTILL-0*)

```

apply (clarsimp simp: set-eq-iff iT-Mult-mem-iff iT-iff)
apply (subst mult.commute[of m k])
apply (simp add: mod-mult2-eq)
apply (rule iffI)
  apply (elim conjE)
  apply (drule mult-le-mono1[of - - k])
  apply (simp add: div-mult-cancel)
apply (elim conjE)
apply (subgoal-tac x mod k = 0)
prefer 2
apply (drule-tac arg-cong[where f= $\lambda x. x \text{ mod } k$ ])
apply (simp add: mult.commute[of k])
apply (drule div-le-mono[of - - k])
apply simp
done

```

**lemma** *iMODb-mult*:

```

  [r, mod m, c]  $\otimes$  k = [r * k, mod m * k, c]
apply (case-tac k = 0)
apply (simp add: iMODb-mod-0 iT-mult-0 iIN-0 iTILL-0)
apply (subst iMODb-iMOD-iTILL-conv)
apply (simp add: iT-Mult-Int iMOD-mult iTILL-mult iMODb-iMOD-iTILL-conv)
apply (subst Int-assoc[symmetric])
apply (subst Int-absorb2)
  apply (simp add: iMOD-subset)
apply (simp add: iMOD-iTILL-iMODb-conv add-mult-distrib2)
done

```

**lemmas** *iT-mult* =

```

  iTILL-mult
  iFROM-mult
  iIN-mult
  iMOD-mult
  iMODb-mult
  iT-Mult-singleton

```

### 2.1.2 Some conversions between intervals using constant addition and multiplication

**lemma** *iFROM-conv*: [*n...*] = *UNIV*  $\oplus$  *n*  
**by** (*simp add: iFROM-0[symmetric] iFROM-add*)

**lemma** *iIN-conv*: [*n...d*] = [*...d*]  $\oplus$  *n*  
**by** (*simp add: iTILL-add*)

**lemma** *iMOD-conv*: [*r, mod m*] = [*0...*]  $\otimes$  *m*  $\oplus$  *r*  
**apply** (*case-tac m = 0*)  
**apply** (*simp add: iMOD-0 iT-mult-0 iTILL-add*)  
**apply** (*simp add: iFROM-mult iMOD-add*)

done

```
lemma iMODb-conv: [r, mod m, c] = [...c] ⊗ m ⊕ r
apply (case-tac m = 0)
  apply (simp add: iMODb-mod-0 iT-mult-0 iTILL-add)
apply (simp add: iTILL-mult iMODb-add)
done
```

Some examples showing the utility of iMODb\_conv

```
lemma [12, mod 10, 4] = {12, 22, 32, 42, 52}
apply (simp add: iT-defs)
apply safe
defer 1
apply simp+
```

The direct proof without iMODb\_conv fails

oops

```
lemma [12, mod 10, 4] = {12, 22, 32, 42, 52}
apply (simp only: iMODb-conv)
apply (simp add: iT-defs iT-Mult-def iT-Plus-def)
apply safe
apply simp+
done
```

```
lemma [12, mod 10, 4] = {12, 22, 32, 42, 52}
apply (simp only: iMODb-conv)
apply (simp add: iT-defs iT-Mult-def iT-Plus-def)
apply (simp add: atMost-def)
apply safe
apply simp+
done
```

```
lemma [r, mod m, 4] = {r, r+m, r+2*m, r+3*m, r+4*m}
apply (simp only: iMODb-conv)
apply (simp add: iT-defs iT-Mult-def iT-Plus-def atMost-def)
apply (simp add: image-Collect)
apply safe
apply fastforce+
done
```

```
lemma [2, mod 10, 4] = {2, 12, 22, 32, 42}
apply (simp only: iMODb-conv)
apply (simp add: iT-defs iT-Plus-def iT-Mult-def)
apply fastforce
done
```

### 2.1.3 Subtraction of constants

definition iT-Plus-neg :: iT ⇒ Time ⇒ iT (infixl ⟨⊕−⟩ 55) where

$I \oplus - k \equiv \{x. x + k \in I\}$

**lemma** *iT-Plus-neg-mem-iff*:  $(x \in I \oplus - k) = (x + k \in I)$   
**by** (*simp add: iT-Plus-neg-def*)

**lemma** *iT-Plus-neg-mem-iff2*:  $k \leq x \implies (x - k \in I \oplus - k) = (x \in I)$   
**by** (*simp add: iT-Plus-neg-def*)

**lemma** *iT-Plus-neg-image-conv*:  $I \oplus - k = (\lambda n. (n - k)) \text{ ` } (I \downarrow \geq k)$   
**apply** (*simp add: iT-Plus-neg-def cut-ge-def, safe*)  
**apply** (*rule-tac x=x+k in image-eqI*)  
**apply** *simp+*  
**done**

**lemma** *iT-Plus-neg-cut-eq*:  $t \leq k \implies (I \downarrow \geq t) \oplus - k = I \oplus - k$   
**by** (*simp add: set-eq-iff iT-Plus-neg-mem-iff cut-ge-mem-iff*)

**lemma** *iT-Plus-neg-mono*:  $A \subseteq B \implies A \oplus - k \subseteq B \oplus - k$   
**by** (*simp add: iT-Plus-neg-def subset-iff*)

**lemma** *iT-Plus-neg-empty*:  $\{\} \oplus - k = \{\}$   
**by** (*simp add: iT-Plus-neg-def*)

**lemma** *iT-Plus-neg-Max-less-empty*:  
 $\llbracket \text{finite } I; \text{Max } I < k \rrbracket \implies I \oplus - k = \{\}$   
**by** (*simp add: iT-Plus-neg-image-conv cut-ge-Max-empty*)

**lemma** *iT-Plus-neg-not-empty-iff*:  $(I \oplus - k \neq \{\}) = (\exists x \in I. k \leq x)$   
**by** (*simp add: iT-Plus-neg-image-conv cut-ge-not-empty-iff*)

**lemma** *iT-Plus-neg-empty-iff*:  
 $(I \oplus - k = \{\}) = (I = \{\} \vee (\text{finite } I \wedge \text{Max } I < k))$   
**apply** (*case-tac I = \{\}*)  
**apply** (*simp add: iT-Plus-neg-empty*)  
**apply** (*simp add: iT-Plus-neg-image-conv*)  
**apply** (*case-tac infinite I*)  
**apply** (*simp add: nat-cut-ge-infinite-not-empty*)  
**apply** (*simp add: cut-ge-empty-iff*)  
**done**

**lemma** *iT-Plus-neg-assoc*:  $(I \oplus - a) \oplus - b = I \oplus - (a + b)$   
**apply** (*simp add: iT-Plus-neg-def*)  
**apply** (*simp add: add.assoc add.commute[of b]*)  
**done**

**lemma** *iT-Plus-neg-commute*:  $I \oplus - a \oplus - b = I \oplus - b \oplus - a$   
**by** (*simp add: iT-Plus-neg-assoc add.commute[of b]*)

**lemma** *iT-Plus-neg-0*:  $I \oplus - 0 = I$   
**by** (*simp add: iT-Plus-neg-image-conv cut-ge-0-all*)

**lemma** *iT-Plus-Plus-neg-assoc*:  $b \leq a \implies I \oplus a \oplus - b = I \oplus (a - b)$   
**apply** (*simp add: iT-Plus-neg-image-conv*)  
**apply** (*clarsimp simp add: set-eq-iff image-iff Bex-def cut-ge-mem-iff iT-Plus-mem-iff*)  
**apply** (*rule iffI*)  
**apply** *fastforce*  
**apply** (*rule-tac x=x + b in exI*)  
**apply** (*simp add: le-diff-conv*)  
**done**

**lemma** *iT-Plus-Plus-neg-assoc2*:  $a \leq b \implies I \oplus a \oplus - b = I \oplus - (b - a)$   
**apply** (*simp add: iT-Plus-neg-image-conv*)  
**apply** (*clarsimp simp add: set-eq-iff image-iff Bex-def cut-ge-mem-iff iT-Plus-mem-iff*)  
**apply** (*rule iffI*)  
**apply** *fastforce*  
**apply** (*clarify, rename-tac x'*)  
**apply** (*rule-tac x=x' + a in exI*)  
**apply** *simp*  
**done**

**lemma** *iT-Plus-neg-Plus-le-cut-eq*:  
 $a \leq b \implies (I \oplus - a) \oplus b = (I \downarrow \geq a) \oplus (b - a)$   
**apply** (*simp add: iT-Plus-neg-image-conv*)  
**apply** (*clarsimp simp add: set-eq-iff image-iff Bex-def cut-ge-mem-iff iT-Plus-mem-iff*)  
**apply** (*rule iffI*)  
**apply** (*clarify, rename-tac x'*)  
**apply** (*subgoal-tac x' = x + a - b*)  
**prefer** 2  
**apply** *simp*  
**apply** (*simp add: le-imp-diff-le le-add-diff*)  
**apply** *fastforce*  
**done**

**corollary** *iT-Plus-neg-Plus-le-Min-eq*:  
 $\llbracket a \leq b; a \leq iMin I \rrbracket \implies (I \oplus - a) \oplus b = I \oplus (b - a)$   
**by** (*simp add: iT-Plus-neg-Plus-le-cut-eq cut-ge-Min-all*)

**lemma** *iT-Plus-neg-Plus-ge-cut-eq*:  
 $b \leq a \implies (I \oplus - a) \oplus b = (I \downarrow \geq a) \oplus - (a - b)$   
**apply** (*simp add: iT-Plus-neg-image-conv iT-Plus-def cut-cut-ge max-eqL*)  
**apply** (*subst image-comp*)  
**apply** (*rule image-cong, simp*)  
**apply** (*simp add: cut-ge-mem-iff*)  
**done**

**corollary** *iT-Plus-neg-Plus-ge-Min-eq*:  
 $\llbracket b \leq a; a \leq iMin I \rrbracket \implies (I \oplus - a) \oplus b = I \oplus - (a - b)$   
**by** (*simp add: iT-Plus-neg-Plus-ge-cut-eq cut-ge-Min-all*)

**lemma** *iT-Plus-neg-Mult-distrib*:

$0 < m \implies I \oplus - n \otimes m = I \otimes m \oplus - n * m$   
**apply** (*clarsimp simp: set-eq-iff iT-Plus-neg-image-conv image-iff iT-Plus-def iT-Mult-def*  
*Bex-def cut-ge-mem-iff*)  
**apply** (*rule iffI*)  
**apply** (*clarsimp, rename-tac x'*)  
**apply** (*rule-tac x=x' \* m in exI*)  
**apply** (*simp add: diff-mult-distrib*)  
**apply** (*clarsimp, rename-tac x'*)  
**apply** (*rule-tac x=x' - n in exI*)  
**apply** (*simp add: diff-mult-distrib*)  
**apply** *fastforce*  
**done**

**lemma** *iT-Plus-neg-Plus-le-inverse*:  $k \leq iMin I \implies I \oplus - k \oplus k = I$   
**by** (*simp add: iT-Plus-neg-Plus-le-Min-eq iT-Plus-0*)

**lemma** *iT-Plus-neg-Plus-inverse*:  $I \oplus - k \oplus k = I \downarrow \geq k$   
**by** (*simp add: iT-Plus-neg-Plus-ge-cut-eq iT-Plus-neg-0*)

**lemma** *iT-Plus-Plus-neg-inverse*:  $I \oplus k \oplus - k = I$   
**by** (*simp add: iT-Plus-Plus-neg-assoc iT-Plus-0*)

**lemma** *iT-Plus-neg-Un*:  $(A \cup B) \oplus - k = (A \oplus - k) \cup (B \oplus - k)$   
**by** (*fastforce simp: iT-Plus-neg-def*)

**lemma** *iT-Plus-neg-Int*:  $(A \cap B) \oplus - k = (A \oplus - k) \cap (B \oplus - k)$   
**by** (*fastforce simp: iT-Plus-neg-def*)

**lemma** *iT-Plus-neg-Max-singleton*:  $\llbracket \text{finite } I; I \neq \{\} \rrbracket \implies I \oplus - Max I = \{0\}$   
**apply** (*rule set-eqI*)  
**apply** (*simp add: iT-Plus-neg-def*)  
**apply** (*case-tac x = 0*)  
**apply** *simp*  
**apply** *fastforce*  
**done**

**lemma** *iT-Plus-neg-singleton*:  $\{a\} \oplus - k = (\text{if } k \leq a \text{ then } \{a - k\} \text{ else } \{\})$   
**by** (*force simp add: set-eq-iff iT-Plus-neg-mem-iff singleton-iff*)

**corollary** *iT-Plus-neg-singleton1*:  $k \leq a \implies \{a\} \oplus - k = \{a - k\}$   
**by** (*simp add: iT-Plus-neg-singleton*)

**corollary** *iT-Plus-neg-singleton2*:  $a < k \implies \{a\} \oplus - k = \{\}$   
**by** (*simp add: iT-Plus-neg-singleton*)

**lemma** *iT-Plus-neg-finite-iff*:  $\text{finite } (I \oplus - k) = \text{finite } I$   
**apply** (*simp add: iT-Plus-neg-image-conv*)

**apply** (*subst inj-on-finite-image-iff*)  
**apply** (*metis cut-geE inj-on-diff-nat*)  
**apply** (*simp add: nat-cut-ge-finite-iff*)  
**done**

**lemma** *iT-Plus-neg-Min*:

$I \oplus - k \neq \{\}$   $\implies iMin (I \oplus - k) = iMin (I \downarrow \geq k) - k$   
**apply** (*simp add: iT-Plus-neg-image-conv*)  
**apply** (*simp add: iMin-mono2 monoI*)  
**done**

**lemma** *iT-Plus-neg-Max*:

$\llbracket \text{finite } I; I \oplus - k \neq \{\} \rrbracket \implies Max (I \oplus - k) = Max I - k$   
**apply** (*simp add: iT-Plus-neg-image-conv*)  
**apply** (*simp add: Max-mono2 monoI cut-ge-finite cut-ge-Max-eq*)  
**done**

Subtractions of constants from intervals

**lemma** *iFROM-add-neg*:  $[n..] \oplus - k = [n - k..]$   
**by** (*fastforce simp: set-eq-iff iT-Plus-neg-mem-iff*)

**lemma** *iTILL-add-neg*:  $[..n] \oplus - k = (\text{if } k \leq n \text{ then } [..n - k] \text{ else } \{\})$   
**by** (*force simp add: set-eq-iff iT-Plus-neg-mem-iff iT-iff*)

**lemma** *iTILL-add-neg1*:  $k \leq n \implies [..n] \oplus - k = [..n - k]$   
**by** (*simp add: iTILL-add-neg*)

**lemma** *iTILL-add-neg2*:  $n < k \implies [..n] \oplus - k = \{\}$   
**by** (*simp add: iTILL-add-neg*)

**lemma** *iIN-add-neg*:

$[n..,d] \oplus - k = (\text{if } k \leq n \text{ then } [n - k..,d] \text{ else if } k \leq n + d \text{ then } [..n + d - k] \text{ else } \{\})$   
**by** (*simp add: iIN-iFROM-iTILL-conv iT-Plus-neg-Int iFROM-add-neg iTILL-add-neg iFROM-0*)

**lemma** *iIN-add-neg1*:  $k \leq n \implies [n..,d] \oplus - k = [n - k..,d]$   
**by** (*simp add: iIN-add-neg*)

**lemma** *iIN-add-neg2*:  $\llbracket n \leq k; k \leq n + d \rrbracket \implies [n..,d] \oplus - k = [..n + d - k]$   
**by** (*simp add: iIN-add-neg iIN-0-iTILL-conv*)

**lemma** *iIN-add-neg3*:  $n + d < k \implies [n..,d] \oplus - k = \{\}$   
**by** (*simp add: iT-Plus-neg-Max-less-empty iT-finite iT-Max*)

**lemma** *iMOD-0-add-neg*:  $[r, \text{mod } 0] \oplus - k = \{r\} \oplus - k$   
**by** (*simp add: iMOD-0 iIN-0*)

**lemma** *iMOD-gr0-add-neg*:

```

0 < m ==>
[r, mod m] ⊕- k = (
  if k ≤ r then [r - k, mod m]
  else [(m + r mod m - k mod m) mod m, mod m])
apply (rule set-eqI)
apply (simp add: iMOD-def iT-Plus-neg-def)
apply (simp add: eq-sym-conv[of - r mod m])
apply (intro conjI impI)
apply (simp add: eq-sym-conv[of - (r - k) mod m] mod-sub-add le-diff-conv)
apply (simp add: eq-commute[of r mod m] mod-add-eq-mod-conv)
apply safe
apply (drule sym)
apply simp
done

```

**lemma** *iMOD-add-neg*:

```

[r, mod m] ⊕- k = (
  if k ≤ r then [r - k, mod m]
  else if 0 < m then [(m + r mod m - k mod m) mod m, mod m] else {})
apply (case-tac 0 < m)
apply (simp add: iMOD-gr0-add-neg)
apply (simp add: iMOD-0 iIN-0 iT-Plus-neg-singleton)
done

```

**corollary** *iMOD-add-neg1*:

```

k ≤ r ==> [r, mod m] ⊕- k = [r - k, mod m]
by (simp add: iMOD-add-neg)

```

**lemma** *iMOD-add-neg2*:

```

[[ 0 < m; r < k ]] ==> [r, mod m] ⊕- k = [(m + r mod m - k mod m) mod m,
mod m]
by (simp add: iMOD-add-neg)

```

**lemma** *iMODb-mod-0-add-neg*:  $[r, \text{mod } 0, c] \oplus - k = \{r\} \oplus - k$

**by** (simp add: iMODb-mod-0 iIN-0)

**lemma** *iMODb-add-neg*:

```

[r, mod m, c] ⊕- k = (
  if k ≤ r then [r - k, mod m, c]
  else
    if k ≤ r + m * c then
      [(m + r mod m - k mod m) mod m, mod m, (r + m * c - k) div m]
    else {})
apply (clarsimp simp add: iMODb-iMOD-iIN-conv iT-Plus-neg-Int iMOD-add-neg
iIN-add-neg)

```

```

apply (simp add: iMOD-iIN-iMODb-conv)
apply (rule-tac t=(m + r mod m - k mod m) mod m and s=(r + m * c - k)
mod m in subst)
apply (simp add: mod-diff1-eq[of k])
apply (subst iMOD-iTILL-iMODb-conv, simp)
apply (subst sub-mod-div-eq-div, simp)
done

```

**lemma** *iMODb-add-neg'*:

```

[r, mod m, c] ⊕- k = (
  if k ≤ r then [r - k, mod m, c]
  else if k ≤ r + m * c then
    if k mod m ≤ r mod m
      then [ r mod m - k mod m, mod m, c + r div m - k div m ]
      else [ m + r mod m - k mod m, mod m, c + r div m - Suc (k div m) ]
  else {})
apply (clarsimp simp add: iMODb-add-neg)
apply (case-tac m = 0, simp+)
apply (case-tac k mod m ≤ r mod m)
apply (clarsimp simp: linorder-not-le)
apply (simp add: divisor-add-diff-mod-if)
apply (simp add: div-diff1-eq-if)
apply (clarsimp simp: linorder-not-le)
apply (simp add: div-diff1-eq-if)
done

```

**corollary** *iMODb-add-neg1*:

```

k ≤ r ⇒ [r, mod m, c] ⊕- k = [r - k, mod m, c]
by (simp add: iMODb-add-neg)

```

**corollary** *iMODb-add-neg2*:

```

[[ r < k; k ≤ r + m * c ]] ⇒
[r, mod m, c] ⊕- k =
[(m + r mod m - k mod m) mod m, mod m, (r + m * c - k) div m]
by (simp add: iMODb-add-neg)

```

**corollary** *iMODb-add-neg2-mod-le*:

```

[[ r < k; k ≤ r + m * c; k mod m ≤ r mod m ]] ⇒
[r, mod m, c] ⊕- k =
[r mod m - k mod m, mod m, c + r div m - k div m]
by (simp add: iMODb-add-neg)

```

**corollary** *iMODb-add-neg2-mod-less*:

```

[[ r < k; k ≤ r + m * c; r mod m < k mod m ]] ⇒
[r, mod m, c] ⊕- k =
[m + r mod m - k mod m, mod m, c + r div m - Suc (k div m) ]
by (simp add: iMODb-add-neg)

```

**lemma** *iMODb-add-neg3*:  $r + m * c < k \implies [r, \text{mod } m, c] \oplus - k = \{\}$

by (simp add: iMODb-add-neg)

lemmas iT-add-neg =  
 iFROM-add-neg  
 iIN-add-neg  
 iTILL-add-neg  
 iMOD-add-neg  
 iMODb-add-neg  
 iT-Plus-neg-singleton

#### 2.1.4 Subtraction of intervals from constants

**definition** *iT-Minus* :: *Time*  $\Rightarrow$  *iT*  $\Rightarrow$  *iT* (infixl  $\ominus$  55)  
 where  $k \ominus I \equiv \{x. x \leq k \wedge (k - x) \in I\}$

**lemma** *iT-Minus-mem-iff*:  $(x \in k \ominus I) = (x \leq k \wedge k - x \in I)$   
 by (simp add: iT-Minus-def)

**lemma** *iT-Minus-mono*:  $A \subseteq B \Longrightarrow k \ominus A \subseteq k \ominus B$   
 by (simp add: subset-iff iT-Minus-mem-iff)

**lemma** *iT-Minus-image-conv*:  $k \ominus I = (\lambda x. k - x) ` (I \downarrow \leq k)$   
 by (fastforce simp: iT-Minus-def cut-le-def image-iff)

**lemma** *iT-Minus-cut-eq*:  $k \leq t \Longrightarrow k \ominus (I \downarrow \leq t) = k \ominus I$   
 by (fastforce simp: set-eq-iff iT-Minus-mem-iff)

**lemma** *iT-Minus-Minus-cut-eq*:  $k \ominus (k \ominus (I \downarrow \leq k)) = I \downarrow \leq k$   
 by (fastforce simp: iT-Minus-def)

**lemma**  $10 \ominus [\dots 3] = [7 \dots, 3]$   
 by (fastforce simp: iT-Minus-def)

**lemma** *iT-Minus-empty*:  $k \ominus \{\} = \{\}$   
 by (simp add: iT-Minus-def)

**lemma** *iT-Minus-0*:  $k \ominus \{0\} = \{k\}$   
 by (simp add: iT-Minus-image-conv cut-le-def image-Collect)

**lemma** *iT-Minus-bound*:  $x \in k \ominus I \Longrightarrow x \leq k$   
 by (simp add: iT-Minus-def)

**lemma** *iT-Minus-finite*: finite  $(k \ominus I)$   
 apply (rule finite-nat-iff-bounded-le2[THEN iffD2])  
 apply (rule-tac x=k in exI)  
 apply (simp add: iT-Minus-bound)  
 done

**lemma** *iT-Minus-less-Min-empty*:  $k < iMin I \Longrightarrow k \ominus I = \{\}$

by (simp add: iT-Minus-image-conv cut-le-Min-empty)

**lemma** *iT-Minus-Min-singleton*:  $I \neq \{\}$   $\implies (iMin\ I) \ominus I = \{0\}$   
 apply (rule set-eqI)  
 apply (simp add: iT-Minus-mem-iff)  
 apply (fastforce intro: iMinI-ex2)  
 done

**lemma** *iT-Minus-empty-iff*:  $(k \ominus I = \{\}) = (I = \{\} \vee k < iMin\ I)$   
 apply (case-tac  $I = \{\}$ , simp add: iT-Minus-empty)  
 apply (simp add: iT-Minus-image-conv cut-le-empty-iff iMin-gr-iff)  
 done

**lemma** *iT-Minus-imirror-conv*:

$k \ominus I = imirror\ (I \downarrow \leq k) \oplus k \oplus - (iMin\ I + Max\ (I \downarrow \leq k))$   
 apply (case-tac  $I = \{\}$ )  
 apply (simp add: iT-Minus-empty cut-le-empty imirror-empty iT-Plus-empty iT-Plus-neg-empty)  
 apply (case-tac  $k < iMin\ I$ )  
 apply (simp add: iT-Minus-less-Min-empty cut-le-Min-empty imirror-empty iT-Plus-empty iT-Plus-neg-empty)  
 apply (simp add: linorder-not-less)  
 apply (frule cut-le-Min-not-empty[of - k], assumption)  
 apply (rule set-eqI)  
 apply (simp add: iT-Minus-image-conv iT-Plus-neg-image-conv iT-Plus-neg-mem-iff iT-Plus-mem-iff imirror-iff image-iff Bex-def i-cut-mem-iff cut-le-Min-eq)  
 apply (rule iffI)  
 apply (clarsimp, rename-tac  $x'$ )  
 apply (rule-tac  $x=k - x' + iMin\ I + Max\ (I \downarrow \leq k)$  in  $exI$ , simp)  
 apply (simp add: add.assoc le-add-diff)  
 apply (simp add: add.commute[of k] le-add-diff nat-cut-le-finite cut-leI trans-le-add2)  
 apply (rule-tac  $x=x'$  in  $exI$ , simp)  
 apply (clarsimp, rename-tac  $x1\ x2$ )  
 apply (rule-tac  $x=x2$  in  $exI$ )  
 apply simp  
 apply (drule add-right-cancel[THEN iffD2, of - - k], simp)  
 apply (simp add: trans-le-add2 nat-cut-le-finite cut-le-mem-iff)  
 done

**lemma** *iT-Minus-imirror-conv'*:

$k \ominus I = imirror\ (I \downarrow \leq k) \oplus k \oplus - (iMin\ (I \downarrow \leq k) + Max\ (I \downarrow \leq k))$   
 apply (case-tac  $I = \{\}$ )  
 apply (simp add: iT-Minus-empty cut-le-empty imirror-empty iT-Plus-empty iT-Plus-neg-empty)  
 apply (case-tac  $k < iMin\ I$ )  
 apply (simp add: iT-Minus-less-Min-empty cut-le-Min-empty imirror-empty iT-Plus-empty iT-Plus-neg-empty)  
 apply (simp add: cut-le-Min-not-empty cut-le-Min-eq iT-Minus-imirror-conv)  
 done

**lemma** *iT-Minus-Max*:

$\llbracket I \neq \{\}; iMin\ I \leq k \rrbracket \implies Max\ (k \ominus I) = k - (iMin\ I)$   
**apply** (*rule Max-equality*)  
**apply** (*simp add: iT-Minus-mem-iff iMinI-ex2*)  
**apply** (*simp add: iT-Minus-finite*)  
**apply** (*fastforce simp: iT-Minus-def*)  
**done**

**lemma** *iT-Minus-Min*:

$\llbracket I \neq \{\}; iMin\ I \leq k \rrbracket \implies iMin\ (k \ominus I) = k - (Max\ (I \downarrow \leq k))$   
**apply** (*insert nat-cut-le-finite[of I k]*)  
**apply** (*frule cut-le-Min-not-empty[of - k], assumption*)  
**apply** (*rule iMin-equality*)  
**apply** (*simp add: iT-Minus-mem-iff nat-cut-le-Max-le del: Max-le-iff*)  
**apply** (*simp add: subsetD[OF cut-le-subset, OF Max-in]*)  
**apply** (*clarsimp simp add: iT-Minus-image-conv image-iff, rename-tac x'*)  
**apply** (*rule diff-le-mono2*)  
**apply** (*simp add: Max-ge-iff cut-le-mem-iff*)  
**done**

**lemma** *iT-Minus-Minus-eq*:  $\llbracket finite\ I; Max\ I \leq k \rrbracket \implies k \ominus (k \ominus I) = I$

**apply** (*simp add: iT-Minus-cut-eq[of k k I, symmetric] iT-Minus-Minus-cut-eq*)  
**apply** (*simp add: cut-le-Max-all*)  
**done**

**lemma** *iT-Minus-Minus-eq2*:  $I \subseteq [\dots k] \implies k \ominus (k \ominus I) = I$

**apply** (*case-tac I = \{\}*)  
**apply** (*simp add: iT-Minus-empty*)  
**apply** (*rule iT-Minus-Minus-eq*)  
**apply** (*simp add: finite-subset iTILL-finite*)  
**apply** (*frule Max-subset*)  
**apply** (*simp add: iTILL-finite iTILL-Max*)  
**done**

**lemma** *iT-Minus-Minus*:  $a \ominus (b \ominus I) = (I \downarrow \leq b) \oplus a \oplus -\ b$

**apply** (*rule set-eqI*)  
**apply** (*simp add: iT-Minus-image-conv iT-Plus-image-conv iT-Plus-neg-image-conv image-iff Bex-def i-cut-mem-iff*)  
**apply** *fastforce*  
**done**

**lemma** *iT-Minus-Plus-empty*:  $k < n \implies k \ominus (I \oplus n) = \{\}$

**apply** (*case-tac I = \{\}*)  
**apply** (*simp add: iT-Plus-empty iT-Minus-empty*)  
**apply** (*simp add: iT-Minus-empty-iff iT-Plus-empty-iff iT-Plus-Min*)  
**done**

**lemma** *iT-Minus-Plus-commute*:  $n \leq k \implies k \ominus (I \oplus n) = (k - n) \ominus I$

**apply** (*rule set-eqI*)  
**apply** (*simp add: iT-Minus-image-conv iT-Plus-image-conv image-iff Bex-def i-cut-mem-iff*)  
**apply** *fastforce*  
**done**

**lemma** *iT-Minus-Plus-cut-assoc*:  $(k \ominus I) \oplus n = (k + n) \ominus (I \downarrow \leq k)$   
**apply** (*rule set-eqI*)  
**apply** (*simp add: iT-Plus-mem-iff iT-Minus-mem-iff cut-le-mem-iff*)  
**apply** *fastforce*  
**done**

**lemma** *iT-Minus-Plus-assoc*:  
 $\llbracket \text{finite } I; \text{Max } I \leq k \rrbracket \implies (k \ominus I) \oplus n = (k + n) \ominus I$   
**by** (*insert iT-Minus-Plus-cut-assoc[of k I n], simp add: cut-le-Max-all*)  
**lemma** *iT-Minus-Plus-assoc2*:  
 $I \subseteq [\dots k] \implies (k \ominus I) \oplus n = (k + n) \ominus I$   
**apply** (*case-tac I = {}*)  
**apply** (*simp add: iT-Minus-empty iT-Plus-empty*)  
**apply** (*rule iT-Minus-Plus-assoc*)  
**apply** (*simp add: finite-subset iTILL-finite*)  
**apply** (*frule Max-subset*)  
**apply** (*simp add: iTILL-finite iTILL-Max*)  
**done**

**lemma** *iT-Minus-Un*:  $k \ominus (A \cup B) = (k \ominus A) \cup (k \ominus B)$   
**by** (*fastforce simp: iT-Minus-def*)

**lemma** *iT-Minus-Int*:  $k \ominus (A \cap B) = (k \ominus A) \cap (k \ominus B)$   
**by** (*fastforce simp: set-eq-iff iT-Minus-mem-iff*)

**lemma** *iT-Minus-singleton*:  $k \ominus \{a\} = (\text{if } a \leq k \text{ then } \{k - a\} \text{ else } \{\})$   
**by** (*simp add: iT-Minus-image-conv cut-le-singleton*)  
**corollary** *iT-Minus-singleton1*:  $a \leq k \implies k \ominus \{a\} = \{k - a\}$   
**by** (*simp add: iT-Minus-singleton*)  
**corollary** *iT-Minus-singleton2*:  $k < a \implies k \ominus \{a\} = \{\}$   
**by** (*simp add: iT-Minus-singleton*)

**lemma** *iMOD-sub*:  
 $k \ominus [r, \text{mod } m] =$   
 $(\text{if } r \leq k \text{ then } [(k - r) \text{ mod } m, \text{mod } m, (k - r) \text{ div } m] \text{ else } \{\})$   
**apply** (*rule set-eqI*)  
**apply** (*simp add: iT-Minus-mem-iff iT-iff*)  
**apply** (*fastforce simp add: mod-sub-eq-mod-swap[of r, symmetric]*)  
**done**

**corollary** *iMOD-sub1*:  
 $r \leq k \implies k \ominus [r, \text{mod } m] = [(k - r) \text{ mod } m, \text{mod } m, (k - r) \text{ div } m]$

by (*simp add: iMOD-sub*)

**corollary** *iMOD-sub2*:  $k < r \implies k \ominus [r, \text{mod } m] = \{\}$   
 apply (*rule iT-Minus-less-Min-empty*)  
 apply (*simp add: iMOD-Min*)  
 done

**lemma** *iTILL-sub*:  $k \ominus [\dots n] = (\text{if } n \leq k \text{ then } [k - n, \dots, n] \text{ else } [\dots k])$   
 by (*force simp add: set-eq-iff iT-Minus-mem-iff iT-iff*)

**corollary** *iTILL-sub1*:  $n \leq k \implies k \ominus [\dots n] = [k - n, \dots, n]$   
 by (*simp add: iTILL-sub*)

**corollary** *iTILL-sub2*:  $k \leq n \implies k \ominus [\dots n] = [\dots k]$   
 by (*simp add: iTILL-sub iIN-0-iTILL-conv*)

**lemma** *iMODb-sub*:

$k \ominus [r, \text{mod } m, c] = (\text{if } r + m * c \leq k \text{ then } [k - r - m * c, \text{mod } m, c] \text{ else } \text{if } r \leq k \text{ then } [(k - r) \text{ mod } m, \text{mod } m, (k - r) \text{ div } m] \text{ else } \{\})$   
 apply (*case-tac m = 0*)  
 apply (*simp add: iMODb-mod-0 iIN-0 iT-Minus-singleton*)  
 apply (*subst iMODb-iMOD-iTILL-conv*)  
 apply (*subst iT-Minus-Int*)  
 apply (*simp add: iMOD-sub iTILL-sub*)  
 apply (*intro conjI impI*)  
 apply *simp*  
 apply (*subgoal-tac (k - r) mod m ≤ k - (r + m \* c)*)  
 prefer 2  
 apply (*subgoal-tac m \* c ≤ k - r - (k - r) mod m*)  
 prefer 2  
 apply (*drule add-le-imp-le-diff2*)  
 apply (*drule div-le-mono[of - - m], simp*)  
 apply (*drule mult-le-mono2[of - - m]*)  
 apply (*simp add: minus-mod-eq-mult-div [symmetric]*)  
 apply (*simp add: le-diff-conv2[OF mod-le-dividend] del: diff-diff-left*)  
 apply (*subst iMODb-iMOD-iIN-conv*)  
 apply (*simp add: Int-assoc minus-mod-eq-mult-div [symmetric]*)  
 apply (*subst iIN-inter, simp+*)  
 apply (*rule set-eqI*)  
 apply (*fastforce simp add: iT-iff mod-diff-mult-self2 diff-diff-left[symmetric] simp del: diff-diff-left*)  
 apply (*simp add: Int-absorb2 iMODb-iTILL-subset*)  
 done

**corollary** *iMODb-sub1*:

$\llbracket r \leq k; k \leq r + m * c \rrbracket \implies$

$k \ominus [r, \text{mod } m, c] = [(k - r) \text{ mod } m, \text{mod } m, (k - r) \text{ div } m]$   
**by** (*clarsimp simp: iMODb-sub iMODb-mod-0*)

**corollary** *iMODb-sub2*:  $k < r \implies k \ominus [r, \text{mod } m, c] = \{\}$   
**apply** (*rule iT-Minus-less-Min-empty*)  
**apply** (*simp add: iMODb-Min*)  
**done**

**corollary** *iMODb-sub3*:  
 $r + m * c \leq k \implies k \ominus [r, \text{mod } m, c] = [k - r - m * c, \text{mod } m, c]$   
**by** (*simp add: iMODb-sub*)

**lemma** *iFROM-sub*:  $k \ominus [n..] = (\text{if } n \leq k \text{ then } [\dots k - n] \text{ else } \{\})$   
**by** (*simp add: iMOD-1[symmetric] iMOD-sub iMODb-mod-1 iIN-0-iTILL-conv*)

**corollary** *iFROM-sub1*:  $n \leq k \implies k \ominus [n..] = [\dots k - n]$   
**by** (*simp add: iFROM-sub*)

**corollary** *iFROM-sub-empty*:  $k < n \implies k \ominus [n..] = \{\}$   
**by** (*simp add: iFROM-sub*)

**lemma** *iIN-sub*:  
 $k \ominus [n.., d] = (\text{if } n + d \leq k \text{ then } [k - (n + d).., d] \text{ else if } n \leq k \text{ then } [\dots k - n] \text{ else } \{\})$   
**apply** (*simp add: iMODb-mod-1[symmetric] iMODb-sub*)  
**apply** (*simp add: iMODb-mod-1 iIN-0-iTILL-conv*)  
**done**

**lemma** *iIN-sub1*:  $n + d \leq k \implies k \ominus [n.., d] = [k - (n + d).., d]$   
**by** (*simp add: iIN-sub*)

**lemma** *iIN-sub2*:  $\llbracket n \leq k; k \leq n + d \rrbracket \implies k \ominus [n.., d] = [\dots k - n]$   
**by** (*clarsimp simp: iIN-sub iIN-0-iTILL-conv*)

**lemma** *iIN-sub3*:  $k < n \implies k \ominus [n.., d] = \{\}$   
**by** (*simp add: iIN-sub*)

**lemmas** *iT-sub* =  
*iFROM-sub*  
*iIN-sub*  
*iTILL-sub*  
*iMOD-sub*  
*iMODb-sub*  
*iT-Minus-singleton*

### 2.1.5 Division of intervals by constants

Monotonicity and injectivity of arithmetic operators

**lemma** *iMOD-div-right-strict-mono-on*:

$\llbracket 0 < k; k \leq m \rrbracket \implies \text{strict-mono-on } (\lambda x. x \text{ div } k) [r, \text{mod } m]$   
**apply** (*rule div-right-strict-mono-on, assumption*)  
**apply** (*clarsimp simp: iT-iff*)  
**apply** (*drule-tac s=y mod m in sym, simp*)  
**apply** (*rule-tac y=x + m in order-trans, simp*)  
**apply** (*simp add: less-mod-eq-imp-add-divisor-le*)  
**done**

**corollary** *iMOD-div-right-inj-on*:

$\llbracket 0 < k; k \leq m \rrbracket \implies \text{inj-on } (\lambda x. x \text{ div } k) [r, \text{mod } m]$   
**by** (*rule strict-mono-on-imp-inj-on[OF iMOD-div-right-strict-mono-on]*)

**lemma** *iMOD-mult-div-right-inj-on*:

$\text{inj-on } (\lambda x. x \text{ div } (k::\text{nat})) [r, \text{mod } (k * m)]$   
**apply** (*case-tac k \* m = 0*)  
**apply** (*simp del: mult-is-0 mult-eq-0-iff add: iMOD-0 iIN-0*)  
**apply** (*simp add: iMOD-div-right-inj-on*)  
**done**

**lemma** *iMOD-mult-div-right-inj-on2*:

$m \text{ mod } k = 0 \implies \text{inj-on } (\lambda x. x \text{ div } k) [r, \text{mod } m]$   
**by** (*auto simp add: iMOD-mult-div-right-inj-on*)

**lemma** *iMODb-div-right-strict-mono-on*:

$\llbracket 0 < k; k \leq m \rrbracket \implies \text{strict-mono-on } (\lambda x. x \text{ div } k) [r, \text{mod } m, c]$   
**by** (*rule strict-mono-on-subset[OF iMOD-div-right-strict-mono-on iMODb-iMOD-subset-same]*)

**corollary** *iMODb-div-right-inj-on*:

$\llbracket 0 < k; k \leq m \rrbracket \implies \text{inj-on } (\lambda x. x \text{ div } k) [r, \text{mod } m, c]$   
**by** (*rule strict-mono-on-imp-inj-on[OF iMODb-div-right-strict-mono-on]*)

**lemma** *iMODb-mult-div-right-inj-on*:

$\text{inj-on } (\lambda x. x \text{ div } (k::\text{nat})) [r, \text{mod } (k * m), c]$   
**by** (*rule inj-on-subset[OF iMOD-mult-div-right-inj-on iMODb-iMOD-subset-same]*)

**corollary** *iMODb-mult-div-right-inj-on2*:

$m \text{ mod } k = 0 \implies \text{inj-on } (\lambda x. x \text{ div } k) [r, \text{mod } m, c]$   
**by** (*auto simp add: iMODb-mult-div-right-inj-on*)

**definition** *iT-Div* :: *iT*  $\Rightarrow$  *Time*  $\Rightarrow$  *iT* (**infixl**  $\langle \odot \rangle$  55)

**where**  $I \odot k \equiv (\lambda n. (n \text{ div } k)) \text{ ' } I$

**lemma** *iT-Div-image-conv*:  $I \odot k = (\lambda n. (n \text{ div } k)) \text{ ' } I$

**by** (*simp add: iT-Div-def*)

**lemma** *iT-Div-mono*:  $A \subseteq B \implies A \otimes k \subseteq B \otimes k$   
**by** (*simp add: iT-Div-def image-mono*)

**lemma** *iT-Div-empty*:  $\{\} \otimes k = \{\}$

**by** (*simp add: iT-Div-def*)

**lemma** *iT-Div-not-empty*:  $I \neq \{\} \implies I \otimes k \neq \{\}$

**by** (*simp add: iT-Div-def*)

**lemma** *iT-Div-empty-iff*:  $(I \otimes k = \{\}) = (I = \{\})$

**by** (*simp add: iT-Div-def*)

**lemma** *iT-Div-0*:  $I \neq \{\} \implies I \otimes 0 = [\dots 0]$

**by** (*force simp: iT-Div-def*)

**corollary** *iT-Div-0-if*:  $I \otimes 0 = (\text{if } I = \{\} \text{ then } \{\} \text{ else } [\dots 0])$

**by** (*force simp: iT-Div-def*)

**corollary**

*iFROM-div-0*:  $[n\dots] \otimes 0 = [\dots 0]$  **and**

*iTILL-div-0*:  $[\dots n] \otimes 0 = [\dots 0]$  **and**

*iIN-div-0*:  $[n\dots, d] \otimes 0 = [\dots 0]$  **and**

*iMOD-div-0*:  $[r, \text{mod } m] \otimes 0 = [\dots 0]$  **and**

*iMODb-div-0*:  $[r, \text{mod } m, c] \otimes 0 = [\dots 0]$

**by** (*simp add: iT-Div-0 iT-not-empty*)**+**

**lemmas** *iT-div-0* =

*iTILL-div-0*

*iFROM-div-0*

*iIN-div-0*

*iMOD-div-0*

*iMODb-div-0*

**lemma** *iT-Div-1*:  $I \otimes \text{Suc } 0 = I$

**by** (*simp add: iT-Div-def*)

**lemma** *iT-Div-mem-iff-0*:  $x \in (I \otimes 0) = (I \neq \{\} \wedge x = 0)$

**by** (*force simp: iT-Div-0-if*)

**lemma** *iT-Div-mem-iff*:

$0 < k \implies x \in (I \otimes k) = (\exists y \in I. y \text{ div } k = x)$

**by** (*force simp: iT-Div-def*)

**lemma** *iT-Div-mem-iff2*:

$0 < k \implies x \text{ div } k \in (I \otimes k) = (\exists y \in I. y \text{ div } k = x \text{ div } k)$

**by** (*rule iT-Div-mem-iff*)

**lemma** *iT-Div-mem-iff-Int*:

$0 < k \implies x \in (I \otimes k) = (I \cap [x * k \dots k - \text{Suc } 0] \neq \{\})$

**apply** (*simp add: ex-in-conv[symmetric] iT-Div-mem-iff iT-iff*)

**apply** (*simp add: le-less-div-conv*[*symmetric*] *add.commute*[*of k*])  
**apply** (*subst less-eq-le-pred, simp*)  
**apply** *blast*  
**done**

**lemma** *iT-Div-imp-mem*:  
 $0 < k \implies x \in I \implies x \text{ div } k \in (I \otimes k)$   
**by** (*force simp: iT-Div-mem-iff2*)

**lemma** *iT-Div-singleton*:  $\{a\} \otimes k = \{a \text{ div } k\}$   
**by** (*simp add: iT-Div-def*)

**lemma** *iT-Div-Un*:  $(A \cup B) \otimes k = (A \otimes k) \cup (B \otimes k)$   
**by** (*fastforce simp: iT-Div-def*)

**lemma** *iT-Div-insert*:  $(\text{insert } n \ I) \otimes k = \text{insert } (n \text{ div } k) \ (I \otimes k)$   
**by** (*fastforce simp: iT-Div-def*)

**lemma** *not-iT-Div-Int*:  $\neg (\forall k \ A \ B. (A \cap B) \otimes k = (A \otimes k) \cap (B \otimes k))$   
**apply** *simp*  
**apply** (  
  *rule-tac x=3 in exI,*  
  *rule-tac x={0} in exI,*  
  *rule-tac x={1} in exI*)  
**by** (*simp add: iT-Div-def*)

**lemma** *subset-iT-Div-Int*:  $A \subseteq B \implies (A \cap B) \otimes k = (A \otimes k) \cap (B \otimes k)$   
**by** (*simp add: iT-Div-def subset-image-Int*)

**lemma** *iFROM-iT-Div-Int*:  
 $\llbracket 0 < k; n \leq iMin \ A \rrbracket \implies (A \cap [n..]) \otimes k = (A \otimes k) \cap ([n..] \otimes k)$   
**apply** (*rule subset-iT-Div-Int*)  
**apply** (*blast intro: order-trans iMin-le*)  
**done**

**lemma** *iIN-iT-Div-Int*:  
 $\llbracket 0 < k; n \leq iMin \ A; \forall x \in A. x \text{ div } k \leq (n + d) \text{ div } k \implies x \leq n + d \rrbracket \implies$   
 $(A \cap [n..,d]) \otimes k = (A \otimes k) \cap ([n..,d] \otimes k)$   
**apply** (*rule set-eqI*)  
**apply** (*simp add: iT-Div-mem-iff Bex-def iIN-iff*)  
**apply** (*rule iffI*)  
  **apply** *blast*  
**apply** (*clarsimp, rename-tac x1 x2*)  
**apply** (*frule iMin-le*)  
**apply** (*rule-tac x=x1 in exI, simp*)  
**apply** (*drule-tac x=x1 in bspec, simp*)  
**apply** (*drule div-le-mono*[*of - n + d k*])

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

**corollary** *iTILL-iT-Div-Int*:

$\llbracket 0 < k; \forall x \in A. x \text{ div } k \leq n \text{ div } k \longrightarrow x \leq n \rrbracket \Longrightarrow$   
 $(A \cap [\dots n]) \otimes k = (A \otimes k) \cap ([\dots n] \otimes k)$

**by** (*simp add: iIN-0-iTILL-conv[symmetric] iIN-iT-Div-Int*)

**lemma** *iIN-iT-Div-Int-mod-0*:

$\llbracket 0 < k; n \bmod k = 0; \forall x \in A. x \text{ div } k \leq (n + d) \text{ div } k \longrightarrow x \leq n + d \rrbracket \Longrightarrow$   
 $(A \cap [n \dots, d]) \otimes k = (A \otimes k) \cap ([n \dots, d] \otimes k)$

**apply** (*rule set-eqI*)

**apply** (*simp add: iT-Div-mem-iff Bex-def iIN-iff*)

**apply** (*rule iffI*)

**apply** *blast*

**apply** (*elim conjE exE, rename-tac x1 x2*)

**apply** (*rule-tac x=x1 in exI, simp*)

**apply** (*rule conjI*)

**apply** (*rule ccontr, simp add: linorder-not-le*)

**apply** (*drule-tac m=n and n=x2 and k=k in div-le-mono*)

**apply** (*drule-tac a=x1 and m=k in less-mod-0-imp-div-less*)

**apply** *simp+*

**apply** (*drule-tac x=x1 in bspec, simp*)

**apply** (*drule div-le-mono[of - n + d k]*)

**apply** *simp*

**done**

**lemma** *mod-partition-iT-Div-Int*:

$\llbracket 0 < k; 0 < d \rrbracket \Longrightarrow$   
 $(A \cap [n * k \dots, d * k - \text{Suc } 0]) \otimes k =$   
 $(A \otimes k) \cap ([n * k \dots, d * k - \text{Suc } 0] \otimes k)$

**apply** (*rule iIN-iT-Div-Int-mod-0, simp+*)

**apply** (*clarify, rename-tac x*)

**apply** (*simp add: mod-0-imp-sub-1-div-conv*)

**apply** (*rule ccontr, simp add: linorder-not-le pred-less-eq-le*)

**apply** (*drule-tac n=x and k=k in div-le-mono*)

**apply** *simp*

**done**

**corollary** *mod-partition-iT-Div-Int2*:

$\llbracket 0 < k; 0 < d; n \bmod k = 0; d \bmod k = 0 \rrbracket \Longrightarrow$   
 $(A \cap [n \dots, d - \text{Suc } 0]) \otimes k =$   
 $(A \otimes k) \cap ([n \dots, d - \text{Suc } 0] \otimes k)$

**by** (*auto simp add: ac-simps mod-partition-iT-Div-Int elim!: dvdE*)

**corollary** *mod-partition-iT-Div-Int-one-segment*:

$0 < k \Longrightarrow$

$(A \cap [n * k \dots, k - \text{Suc } 0]) \otimes k = (A \otimes k) \cap ([n * k \dots, k - \text{Suc } 0] \otimes k)$

**by** (*insert mod-partition-iT-Div-Int[where d=1], simp*)

**corollary** *mod-partition-iT-Div-Int-one-segment2*:

$\llbracket 0 < k; n \bmod k = 0 \rrbracket \implies$   
 $(A \cap [n..,k - \text{Suc } 0]) \otimes k = (A \otimes k) \cap ([n..,k - \text{Suc } 0] \otimes k)$   
**using** *mod-partition-iT-Div-Int2*[**where**  $k=k$  **and**  $d=k$  **and**  $n=n$ ]  
**by** (*insert mod-partition-iT-Div-Int2*[**where**  $k=k$  **and**  $d=k$  **and**  $n=n$ ], *simp*)

**lemma** *iT-Div-assoc*:  $I \otimes a \otimes b = I \otimes (a * b)$   
**by** (*simp add: iT-Div-def image-image div-mult2-eq*)

**lemma** *iT-Div-commute*:  $I \otimes a \otimes b = I \otimes b \otimes a$   
**by** (*simp add: iT-Div-assoc mult.commute[of a]*)

**lemma** *iT-Mult-Div-self*:  $0 < k \implies I \otimes k \otimes k = I$   
**by** (*simp add: iT-Mult-def iT-Div-def image-image*)

**lemma** *iT-Mult-Div*:  
 $\llbracket 0 < d; k \bmod d = 0 \rrbracket \implies I \otimes k \otimes d = I \otimes (k \text{ div } d)$   
**by** (*auto simp add: ac-simps iT-Mult-assoc[symmetric] iT-Mult-Div-self*)

**lemma** *iT-Div-Mult-self*:  
 $0 < k \implies I \otimes k \otimes k = \{y. \exists x \in I. y = x - x \bmod k\}$   
**by** (*simp add: set-eq-iff iT-Mult-def iT-Div-def image-image image-iff div-mult-cancel*)

**lemma** *iT-Plus-Div-distrib-mod-less*:  
 $\forall x \in I. x \bmod m + n \bmod m < m \implies I \oplus n \otimes m = I \otimes m \oplus n \text{ div } m$   
**by** (*simp add: set-eq-iff iT-Div-def iT-Plus-def image-image image-iff div-add1-eq1*)  
**corollary** *iT-Plus-Div-distrib-mod-0*:  
 $n \bmod m = 0 \implies I \oplus n \otimes m = I \otimes m \oplus n \text{ div } m$   
**apply** (*case-tac m = 0, simp add: iT-Plus-0 iT-Div-0*)  
**apply** (*simp add: iT-Plus-Div-distrib-mod-less*)  
**done**

**lemma** *iT-Div-Min*:  $I \neq \{\}$   $\implies iMin (I \otimes k) = iMin I \text{ div } k$   
**by** (*simp add: iT-Div-def iMin-mono2 mono-def div-le-mono*)

**corollary**  
*iFROM-div-Min*:  $iMin ([n..] \otimes k) = n \text{ div } k$  **and**  
*iIN-div-Min*:  $iMin ([n..,d] \otimes k) = n \text{ div } k$  **and**  
*iTILL-div-Min*:  $iMin ([..n] \otimes k) = 0$  **and**  
*iMOD-div-Min*:  $iMin ([r, \text{mod } m] \otimes k) = r \text{ div } k$  **and**  
*iMODb-div-Min*:  $iMin ([r, \text{mod } m, c] \otimes k) = r \text{ div } k$   
**by** (*simp add: iT-not-empty iT-Div-Min iT-Min*)+

**lemmas** *iT-div-Min* =  
*iFROM-div-Min*  
*iIN-div-Min*  
*iTILL-div-Min*  
*iMOD-div-Min*  
*iMODb-div-Min*

**lemma** *iT-Div-Max*:  $\llbracket \text{finite } I; I \neq \{\} \rrbracket \implies \text{Max } (I \odot k) = \text{Max } I \text{ div } k$   
**by** (*simp add: iT-Div-def Max-mono2 mono-def div-le-mono*)

**corollary**

*iIN-div-Max*:  $\text{Max } ([n \dots, d] \odot k) = (n + d) \text{ div } k$  **and**  
*iTILL-div-Max*:  $\text{Max } ([\dots n] \odot k) = n \text{ div } k$  **and**  
*iMODb-div-Max*:  $\text{Max } ([r, \text{mod } m, c] \odot k) = (r + m * c) \text{ div } k$   
**by** (*simp add: iT-not-empty iT-finite iT-Div-Max iT-Max*)+

**lemma** *iT-Div-0-finite*:  $\text{finite } (I \odot 0)$   
**by** (*simp add: iT-Div-0-if iTILL-0*)

**lemma** *iT-Div-infinite-iff*:  $0 < k \implies \text{infinite } (I \odot k) = \text{infinite } I$   
**apply** (*unfold iT-Div-def*)  
**apply** (*rule iffI*)  
**apply** (*rule infinite-image-imp-infinite, assumption*)  
**apply** (*clarsimp simp: infinite-nat-iff-unbounded-le image-iff, rename-tac x1*)  
**apply** (*drule-tac x=x1 \* k in spec, clarsimp, rename-tac x2*)  
**apply** (*drule div-le-mono[of - - k], simp*)  
**apply** (*rule-tac x=x2 div k in exI*)  
**apply** *fastforce*  
**done**  
**lemma** *iT-Div-finite-iff*:  $0 < k \implies \text{finite } (I \odot k) = \text{finite } I$   
**by** (*insert iT-Div-infinite-iff, simp*)

**lemma** *iFROM-div*:  $0 < k \implies [n \dots] \odot k = [n \text{ div } k \dots]$   
**apply** (*clarsimp simp: set-eq-iff iT-Div-def image-iff Bex-def iFROM-iff, rename-tac x*)  
**apply** (*rule iffI*)  
**apply** (*clarsimp simp: div-le-mono*)  
**apply** (*rule-tac x=n mod k + k \* x in exI*)  
**apply** *simp*  
**apply** (*subst add commute, subst le-diff-conv[symmetric]*)  
**apply** (*subst minus-mod-eq-mult-div*)  
**apply** *simp*  
**done**

**lemma** *iIN-div*:

$0 < k \implies$   
 $[n \dots, d] \odot k = [n \text{ div } k \dots, d \text{ div } k + (n \text{ mod } k + d \text{ mod } k) \text{ div } k]$   
**apply** (*clarsimp simp: set-eq-iff iT-Div-def image-iff Bex-def iIN-iff, rename-tac x*)  
**apply** (*rule iffI*)  
**apply** *clarify*  
**apply** (*drule div-le-mono[of n - k]*)  
**apply** (*drule div-le-mono[of - n + d k]*)  
**apply** (*simp add: div-add1-eq[of n d]*)  
**apply** (*clarify, rename-tac x*)

```

apply (simp add: add.assoc[symmetric] div-add1-eq[symmetric])
apply (frule mult-le-mono1[of n div k - k])
apply (frule mult-le-mono1[of - (n + d) div k k])
apply (simp add: mult.commute[of - k] minus-mod-eq-mult-div [symmetric])
apply (simp add: le-diff-conv le-diff-conv2[OF mod-le-dividend])
apply (drule order-le-less[of - (n + d) div k, THEN iffD1], erule disjE)
  apply (rule-tac x=k * x + n mod k in exI)
  apply (simp add: add.commute[of - n mod k])
  apply (case-tac n mod k ≤ (n + d) mod k, simp)
  apply (simp add: linorder-not-le)
  apply (drule-tac m=x in less-imp-le-pred)
  apply (drule-tac i=x and k=k in mult-le-mono2)
  apply (simp add: diff-mult-distrib2 minus-mod-eq-mult-div [symmetric])
  apply (subst add.commute[of n mod k])
  apply (subst le-diff-conv2[symmetric])
    apply (simp add: trans-le-add1)
  apply (rule order-trans, assumption)
  apply (rule diff-le-mono2)
  apply (simp add: trans-le-add2)
apply (rule-tac x=n + d in exI, simp)
done

```

**corollary** *iIN-div-if*:

```

  0 < k ⇒ [n...d] ∘ k =
  [n div k..., d div k + (if n mod k + d mod k < k then 0 else Suc 0)]
apply (simp add: iIN-div)
apply (simp add: iIN-def add.assoc[symmetric] div-add1-eq[symmetric] div-add1-eq2[where
a=n])
done

```

**corollary** *iIN-div-eq1*:

```

  [ 0 < k; n mod k + d mod k < k ] ⇒
  [n...d] ∘ k = [n div k..., d div k]
by (simp add: iIN-div-if)

```

**corollary** *iIN-div-eq2*:

```

  [ 0 < k; k ≤ n mod k + d mod k ] ⇒
  [n...d] ∘ k = [n div k..., Suc (d div k)]
by (simp add: iIN-div-if)

```

**corollary** *iIN-div-mod-eq-0*:

```

  [ 0 < k; n mod k = 0 ] ⇒ [n...d] ∘ k = [n div k..., d div k]
by (simp add: iIN-div-eq1)

```

**lemma** *iTILL-div*:

```

  0 < k ⇒ [...n] ∘ k = [...n div k]
by (simp add: iIN-0-iTILL-conv[symmetric] iIN-div-if)

```

```

lemma iMOD-div-ge:
  
$$\llbracket 0 < m; m \leq k \rrbracket \implies [r, \text{mod } m] \odot k = [r \text{ div } k \dots]$$

  apply (frule less-le-trans[of - - k], assumption)
  apply (clarsimp simp: set-eq-iff iT-Div-mem-iff Bex-def iT-iff, rename-tac x)
  apply (rule iffI)
  apply (fastforce simp: div-le-mono)
  apply (rule-tac x=
    if x * k < r then r else
    
$$((\text{if } x * k \text{ mod } m \leq r \text{ mod } m \text{ then } 0 \text{ else } m) + r \text{ mod } m + (x * k - x * k \text{ mod } m))$$

    in exI)
  apply (case-tac x * k < r)
  apply simp
  apply (drule less-imp-le[of - r], drule div-le-mono[of - r k], simp)
  apply (simp add: linorder-not-less linorder-not-le)
  apply (simp add: div-le-conv add.commute[of k])
  apply (subst diff-add-assoc, simp)+
  apply (simp add: div-mult-cancel[symmetric] del: add-diff-assoc)
  apply (case-tac x * k mod m = 0)
  apply (clarsimp elim!: dvdE)
  apply (drule sym)
  apply (simp add: mult.commute[of m])
  apply (blast intro: div-less order-less-le-trans mod-less-divisor)
  apply simp
  apply (intro conjI impI)
  apply (simp add: div-mult-cancel)
  apply (simp add: div-mult-cancel)
  apply (subst add.commute, subst diff-add-assoc, simp)
  apply (subst add.commute, subst div-mult-self1, simp)
  apply (subst div-less)
  apply (rule order-less-le-trans[of - m], simp add: less-imp-diff-less)
  apply simp
  apply simp
  apply (rule-tac y=x * k in order-trans, assumption)
  apply (simp add: div-mult-cancel)
  apply (rule le-add-diff)
  apply (simp add: trans-le-add1)
  apply (simp add: div-mult-cancel)
  apply (subst diff-add-assoc2, simp add: trans-le-add1)
  apply simp
  done
corollary iMOD-div-self:
  
$$0 < m \implies [r, \text{mod } m] \odot m = [r \text{ div } m \dots]$$

  by (simp add: iMOD-div-ge)

```

```

lemma iMOD-div:
  
$$\llbracket 0 < k; m \text{ mod } k = 0 \rrbracket \implies$$

  
$$[r, \text{mod } m] \odot k = [r \text{ div } k, \text{mod } (m \text{ div } k)]$$

  apply (case-tac m = 0)

```

```

apply (simp add: iMOD-0 iIN-0 iT-Div-singleton)
apply (clarsimp elim!: dvdE)
apply (rename-tac q)
apply hypsubst-thin
apply (cut-tac r=r div k and k=k and m=q in iMOD-mult)
apply (drule arg-cong[where f= $\lambda x. x \oplus (r \bmod k)$ ])
apply (drule sym)
apply (simp add: iMOD-add mult.commute[of k])
apply (cut-tac I=[r div k, mod q]  $\otimes$  k and m=k and n=r mod k in iT-Plus-Div-distrib-mod-less)
apply (rule ballI)
apply (simp only: iMOD-mult iMOD-iff, elim conjE)
apply (drule mod-factor-imp-mod-0)
apply simp
apply (simp add: iT-Plus-0)
apply (simp add: iT-Mult-Div[OF - mod-self] iT-Mult-1)
done

```

**lemma** iMODb-div-self:

```

 $0 < m \implies [r, \bmod m, c] \odot m = [r \text{ div } m \dots, c]$ 
apply (subst iMODb-iMOD-iTILL-conv)
apply (subst iTILL-iT-Div-Int)
apply simp
apply (clarsimp simp: iT-iff simp del: div-mult-self1 div-mult-self2, rename-tac
x)
apply (drule div-le-mod-le-imp-le)
apply simp+
apply (simp add: iMOD-div-self iTILL-div iFROM-iTILL-iIN-conv)
done

```

**lemma** iMODb-div-ge:

```

 $\llbracket 0 < m; m \leq k \rrbracket \implies$ 
 $[r, \bmod m, c] \odot k = [r \text{ div } k \dots, (r + m * c) \text{ div } k - r \text{ div } k]$ 
apply (case-tac m = k)
apply (simp add: iMODb-div-self)
apply (drule le-neq-trans, simp+)
apply (induct c)
apply (simp add: iMODb-0 iIN-0 iT-Div-singleton)
apply (rule-tac t=[ r, mod m, Suc c ] and s=[ r, mod m, c ]  $\cup$  {r + m * c + m}
in subst)
apply (cut-tac c=c and c'=0 and r=r and m=m in iMODb-append-union-Suc[symmetric])
apply (simp add: iMODb-0 iIN-0 add.commute[of m] add.assoc)
apply (subst iT-Div-Un)
apply (simp add: iT-Div-singleton)
apply (simp add: add.commute[of m] add.assoc[symmetric])
apply (case-tac (r + m * c) mod k + m mod k < k)
apply (simp add: div-add1-eq1)
apply (rule insert-absorb)
apply (simp add: iIN-iff div-le-mono)
apply (simp add: linorder-not-less)

```

```

apply (simp add: div-add1-eq2)
apply (rule-tac t=Suc ((r + m * c) div k) and s=Suc (r div k + ((r + m * c)
div k - r div k)) in subst)
apply (simp add: div-le-mono)
apply (simp add: iN-Suc-insert-conv)
done

```

**corollary** *iMODb-div-ge-if*:

```

[[ 0 < m; m ≤ k ]] ==>
[r, mod m, c] ∘ k =
[r div k..., m * c div k + (if r mod k + m * c mod k < k then 0 else Suc 0)]
by (simp add: iMODb-div-ge div-add1-eq-if[of - r])

```

**lemma** *iMODb-div*:

```

[[ 0 < k; m mod k = 0 ]] ==>
[r, mod m, c] ∘ k = [r div k, mod (m div k), c]
apply (subst iMODb-iMOD-iTILL-conv)
apply (subst iTILL-iT-Div-Int)
apply simp
apply (simp add: Ball-def iMOD-iff, intro allI impI, elim conjE, rename-tac x)
apply (drule div-le-mod-le-imp-le)
apply (subst mod-add1-eq-if)
apply (simp add: mod-0-imp-mod-mult-right-0)
apply (drule mod-eq-mod-0-imp-mod-eq, simp+)
apply (simp add: iMOD-div iTILL-div)
apply (simp add: iMOD-iTILL-iMODb-conv div-le-mono)
apply (clarsimp simp: mult.assoc iMODb-mod-0 iMOD-0 elim!: dvdE)
done

```

**lemmas** *iT-div* =

```

iTILL-div
iFROM-div
iN-div
iMOD-div
iMODb-div
iT-Div-singleton

```

This lemma is valid for all  $k \leq m$ , i. e., also for  $k$  with  $m \bmod k \neq 0$ .

**lemma** *iMODb-div-unique*:

```

[[ 0 < k; k ≤ m; k ≤ c; [r', mod m', c'] = [r, mod m, c] ∘ k ]] ==>
r' = r div k ∧ m' = m div k ∧ c' = c
apply (case-tac r' ≠ r div k)
apply (drule arg-cong[where f=iMin])
apply (simp add: iT-Min iT-not-empty iT-Div-Min)
apply simp
apply (case-tac m' = 0 ∨ c' = 0)
apply (subgoal-tac [ r div k, mod m', c'] = {r div k})
prefer 2
apply (rule iMODb-singleton-eq-conv[THEN iffD2], simp)

```

```

apply simp
apply (drule arg-cong[where f=Max])
apply (simp add: iMODb-mod-0 iIN-0 iT-Max iT-Div-Max iT-Div-finite-iff iT-Div-not-empty
iT-finite iT-not-empty)
apply (subgoal-tac r div k < (r + m * c) div k, simp)
apply (subst div-add1-eq-if, simp)
apply clarsimp
apply (rule order-less-le-trans[of - k * k div k], simp)
apply (rule div-le-mono)
apply (simp add: mult-mono)
apply (subgoal-tac c' = c)
prefer 2
apply (drule arg-cong[where f= $\lambda A.$  card A])
apply (simp add: iT-Div-def card-image[OF iMODb-div-right-inj-on] iMODb-card)
apply clarsimp
apply (frule iMODb-div-right-strict-mono-on[of k m r c], assumption)
apply (frule-tac a=k and b=0 and m=m' and r=r div k and c=c in iMODb-inext-nth-diff,
simp)
apply (simp add: iT-Div-Min iT-not-empty iT-Min)
apply (simp add: iT-Div-def inext-nth-image[OF iMODb-not-empty])
apply (simp add: iMODb-inext-nth)
done

```

**lemma** iMODb-div-mod-gr0-is-0-not-ex0:

```

  [ 0 < k; k < m; 0 < m mod k; k ≤ c; r mod k = 0 ] ⇒
  ¬(∃ r' m' c'. [r', mod m', c'] = [r, mod m, c] ⊗ k)
apply (rule ccontr, simp, elim exE conjE)
apply (frule-tac r'=r' and m'=m' and c'=c' and r=r and k=k and m=m and
c=c
in iMODb-div-unique[OF - less-imp-le], simp+)
apply (drule arg-cong[where f=Max])
apply (simp add: iT-Max iT-Div-Max iT-Div-finite-iff iT-Div-not-empty iT-finite
iT-not-empty)
apply (simp add: div-add1-eq1)
apply (simp add: mult.commute[of m])
apply (simp add: div-mult1-eq[of c m] div-eq-0-conv)
apply (subgoal-tac c ≤ c * (m mod k))
apply simp+
done

```

**lemma** iMODb-div-mod-gr0-not-ex-arith-aux1:

```

  [ (0::nat) < k; k < m; 0 < x1 ] ⇒
  x1 * m + x2 - x mod k + x3 + x mod k = x1 * m + x2 + x3
apply (drule Suc-leI[of - x1])
apply (drule mult-le-mono1[of Suc 0 - m])
apply (subgoal-tac x mod k ≤ x1 * m)
prefer 2
apply (rule order-trans[OF mod-le-divisor], assumption)

```

**apply** (*rule order-less-imp-le*)  
**apply** (*rule order-less-le-trans*)  
**apply** *simp+*  
**done**

**lemma** *iMODb-div-mod-gr0-not-ex:*

$\llbracket 0 < k; k < m; 0 < m \bmod k; k \leq c \rrbracket \implies$   
 $\neg(\exists r' m' c'. [r', \bmod m', c] = [r, \bmod m, c] \otimes k)$   
**apply** (*case-tac r mod k = 0*)  
**apply** (*simp add: iMODb-div-mod-gr0-is-0-not-ex0*)  
**apply** (*rule ccontr, simp, elim exE conjE*)  
**apply** (*frule-tac r'=r' and m'=m' and c'=c' and r=r and k=k and m=m and c=c*)  
**in** *iMODb-div-unique[OF - less-imp-le], simp+*)  
**apply** *clarsimp*  
**apply** (*drule arg-cong[where f=Max]*)  
**apply** (*simp add: iT-Max iT-Div-Max iT-Div-finite-iff iT-Div-not-empty iT-finite iT-not-empty*)  
**apply** (*simp add: div-add1-eq[of r m \* c]*)  
**apply** (*simp add: mult.commute[of - c]*)  
**apply** (*clarsimp simp add: div-mult1-eq[of c m k]*)  
**apply** (*subgoal-tac Suc 0 ≤ c \* (m mod k) div k, simp*)  
**apply** (*thin-tac - = 0*)  
**apply** (*drule div-le-mono[of k c k], simp*)  
**apply** (*rule order-trans[of - c div k], simp*)  
**apply** (*rule div-le-mono, simp*)  
**done**

**lemma** *iMOD-div-eq-imp-iMODb-div-eq:*

$\llbracket 0 < k; k \leq m; [r', \bmod m] = [r, \bmod m] \otimes k \rrbracket \implies$   
 $[r', \bmod m', c] = [r, \bmod m, c] \otimes k$   
**apply** (*subgoal-tac r' = r div k*)  
**prefer** 2  
**apply** (*drule arg-cong[where f=iMin]*)  
**apply** (*simp add: iT-Div-Min iMOD-not-empty iMOD-Min*)  
**apply** *clarsimp*  
**apply** (*frule iMOD-div-right-strict-mono-on[of - m r], assumption*)  
**apply** (*frule card-image[OF strict-mono-on-imp-inj-on[OF iMODb-div-right-strict-mono-on[of k m r c]]], assumption*)  
**apply** (*simp add: iMODb-card*)  
**apply** (*subgoal-tac r + m \* c ∈ [r, mod m]*)  
**prefer** 2  
**apply** (*simp add: iMOD-iff*)  
**apply** (*subgoal-tac [r, mod m, c] = [r, mod m] ↓≤ (r + m \* c)*)  
**prefer** 2  
**apply** (*simp add: iMOD-cut-le1*)  
**apply** (*simp add: iT-Div-def*)  
**apply** (*simp add: cut-le-image[symmetric]*)

```

apply (drule sym)
apply (simp add: iMOD-cut-le)
apply (simp add: linorder-not-le[of r div k, symmetric])
apply (simp add: div-le-mono)
apply (case-tac m' = 0)
  apply (simp add: iMODb-mod-0-card)
apply (rule arg-cong[where f= $\lambda c. [r \text{ div } k, \text{ mod } m', c]$ ])
apply (simp add: iMODb-card)
done

```

```

lemma iMOD-div-unique:
   $\llbracket 0 < k; k \leq m; [r', \text{ mod } m'] = [r, \text{ mod } m] \circledast k \rrbracket \implies$ 
   $r' = r \text{ div } k \wedge m' = m \text{ div } k$ 
apply (frule iMOD-div-eq-imp-iMODb-div-eq[of k m r' m' r k], assumption+)
apply (simp add: iMODb-div-unique[of k - k])
done

```

```

lemma iMOD-div-mod-gr0-not-ex:
   $\llbracket 0 < k; k < m; 0 < m \text{ mod } k \rrbracket \implies$ 
   $\neg (\exists r' m'. [r', \text{ mod } m'] = [r, \text{ mod } m] \circledast k)$ 
apply (rule ccontr, clarsimp)
apply (frule-tac k=k and m=m and r'=r' and m'=m' and c=k
  in iMOD-div-eq-imp-iMODb-div-eq[OF - less-imp-le], assumption+)
apply (frule iMODb-div-mod-gr0-not-ex[of k m k r], simp+)
done

```

## 2.2 Interval cut operators with arithmetic interval operators

```

lemma
  iT-Plus-cut-le2:  $(I \oplus k) \downarrow \leq (t + k) = (I \downarrow \leq t) \oplus k$  and
  iT-Plus-cut-less2:  $(I \oplus k) \downarrow < (t + k) = (I \downarrow < t) \oplus k$  and
  iT-Plus-cut-ge2:  $(I \oplus k) \downarrow \geq (t + k) = (I \downarrow \geq t) \oplus k$  and
  iT-Plus-cut-greater2:  $(I \oplus k) \downarrow > (t + k) = (I \downarrow > t) \oplus k$ 
unfolding iT-Plus-def by fastforce+

```

```

lemma iT-Plus-cut-le:
   $(I \oplus k) \downarrow \leq t = (\text{if } t < k \text{ then } \{\} \text{ else } I \downarrow \leq (t - k) \oplus k)$ 
apply (case-tac t < k)
apply (simp add: cut-le-empty-iff iT-Plus-mem-iff)
apply (insert iT-Plus-cut-le2[of I k t - k], simp)
done

```

```

lemma iT-Plus-cut-less:  $(I \oplus k) \downarrow < t = I \downarrow < (t - k) \oplus k$ 
apply (case-tac t < k)
apply (simp add: cut-less-0-empty iT-Plus-empty cut-less-empty-iff iT-Plus-mem-iff)
apply (insert iT-Plus-cut-less2[of I k t - k], simp)
done

```

**lemma** *iT-Plus-cut-ge*:  $(I \oplus k) \downarrow_{\geq} t = I \downarrow_{\geq} (t - k) \oplus k$   
**apply** (*case-tac*  $t < k$ )  
**apply** (*simp add: cut-ge-0-all cut-ge-all-iff iT-Plus-mem-iff*)  
**apply** (*insert iT-Plus-cut-ge2*[of  $I k t - k$ ], *simp*)  
**done**

**lemma** *iT-Plus-cut-greater*:  
 $(I \oplus k) \downarrow_{>} t = (\text{if } t < k \text{ then } I \oplus k \text{ else } I \downarrow_{>} (t - k) \oplus k)$   
**apply** (*case-tac*  $t < k$ )  
**apply** (*simp add: cut-greater-all-iff iT-Plus-mem-iff*)  
**apply** (*insert iT-Plus-cut-greater2*[of  $I k t - k$ ], *simp*)  
**done**

**lemma**  
*iT-Mult-cut-le2*:  $0 < k \implies (I \otimes k) \downarrow_{\leq} (t * k) = (I \downarrow_{\leq} t) \otimes k$  **and**  
*iT-Mult-cut-less2*:  $0 < k \implies (I \otimes k) \downarrow_{<} (t * k) = (I \downarrow_{<} t) \otimes k$  **and**  
*iT-Mult-cut-ge2*:  $0 < k \implies (I \otimes k) \downarrow_{\geq} (t * k) = (I \downarrow_{\geq} t) \otimes k$  **and**  
*iT-Mult-cut-greater2*:  $0 < k \implies (I \otimes k) \downarrow_{>} (t * k) = (I \downarrow_{>} t) \otimes k$   
**unfolding** *iT-Mult-def* **by** *fastforce+*

**lemma** *iT-Mult-cut-le*:  
 $0 < k \implies (I \otimes k) \downarrow_{\leq} t = (I \downarrow_{\leq} (t \text{ div } k)) \otimes k$   
**apply** (*clarsimp simp: set-eq-iff iT-Mult-mem-iff cut-le-mem-iff*)  
**apply** (*rule conj-cong, simp*)  
**apply** (*rule iffI*)  
**apply** (*simp add: div-le-mono*)  
**apply** (*rule div-le-mod-le-imp-le, simp*)  
**done**

**lemma** *iT-Mult-cut-less*:  
 $0 < k \implies (I \otimes k) \downarrow_{<} t =$   
 $(\text{if } t \bmod k = 0 \text{ then } (I \downarrow_{<} (t \text{ div } k)) \text{ else } I \downarrow_{<} \text{Suc } (t \text{ div } k)) \otimes k$   
**apply** (*case-tac*  $t \bmod k = 0$ )  
**apply** (*clarsimp simp add: mult.commute*[of  $k$ ] *iT-Mult-cut-less2 elim!: dvdE*)  
**apply** (*clarsimp simp: set-eq-iff iT-Mult-mem-iff cut-less-mem-iff*)  
**apply** (*rule conj-cong, simp*)  
**apply** (*subst less-Suc-eq-le*)  
**apply** (*rule iffI*)  
**apply** (*rule div-le-mono, simp*)  
**apply** (*rule ccontr, simp add: linorder-not-less*)  
**apply** (*drule le-imp-less-or-eq*[of  $t$ ], *erule disjE*)  
**apply** (*fastforce dest: less-mod-0-imp-div-less*[of  $t - k$ ])  
**apply** *simp*  
**done**

**lemma** *iT-Mult-cut-greater*:  
 $0 < k \implies (I \otimes k) \downarrow_{>} t = (I \downarrow_{>} (t \text{ div } k)) \otimes k$   
**apply** (*clarsimp simp: set-eq-iff iT-Mult-mem-iff cut-greater-mem-iff*)

```

apply (rule conj-cong, simp)+
apply (rule iffI)
  apply (simp add: less-mod-ge-imp-div-less)
apply (rule ccontr, simp add: linorder-not-less)
apply (fastforce dest: div-le-mono[of - - k])
done

```

```

lemma iT-Mult-cut-ge:
   $0 < k \implies (I \otimes k) \downarrow_{\geq} t =$ 
  (if  $t \bmod k = 0$  then  $(I \downarrow_{\geq} (t \operatorname{div} k))$  else  $I \downarrow_{\geq} \operatorname{Suc} (t \operatorname{div} k) \otimes k$ )
apply (case-tac  $t \bmod k = 0$ )
  apply (clarsimp simp add: mult.commute[of k] iT-Mult-cut-ge2 elim!: dvdE)
apply (clarsimp simp: set-eq-iff iT-Mult-mem-iff cut-ge-mem-iff)
apply (rule conj-cong, simp)+
apply (rule iffI)
  apply (rule Suc-leI)
  apply (simp add: le-mod-greater-imp-div-less)
apply (rule ccontr)
apply (drule Suc-le-lessD)
apply (simp add: linorder-not-le)
apply (fastforce dest: div-le-mono[OF order-less-imp-le, of - t k])
done

```

```

lemma iT-Plus-neg-cut-le2:  $k \leq t \implies (I \oplus - k) \downarrow_{\leq} (t - k) = (I \downarrow_{\leq} t) \oplus - k$ 
apply (simp add: iT-Plus-neg-image-conv)
apply (simp add: i-cut-commute-disj[of ( $\downarrow_{\leq}$ ) ( $\downarrow_{\geq}$ )])
apply (rule i-cut-image[OF sub-left-strict-mono-on])
apply (simp add: cut-ge-Int-conv)+
done

```

```

lemma iT-Plus-neg-cut-less2:  $(I \oplus - k) \downarrow_{<} (t - k) = (I \downarrow_{<} t) \oplus - k$ 
apply (case-tac  $t \leq k$ )
  apply (simp add: cut-less-0-empty)
  apply (case-tac  $I \downarrow_{<} t = \{\}$ )
    apply (simp add: iT-Plus-neg-empty)
    apply (rule sym, rule iT-Plus-neg-Max-less-empty[OF nat-cut-less-finite])
    apply (rule order-less-le-trans[OF cut-less-Max-less[OF nat-cut-less-finite]], assumption+)
  apply (simp add: linorder-not-le iT-Plus-neg-image-conv)
apply (simp add: i-cut-commute-disj[of ( $\downarrow_{<}$ ) ( $\downarrow_{\geq}$ )])
apply (rule i-cut-image[OF sub-left-strict-mono-on])
apply (simp add: cut-ge-Int-conv)+
done

```

```

lemma iT-Plus-neg-cut-ge2:  $(I \oplus - k) \downarrow_{\geq} (t - k) = (I \downarrow_{\geq} t) \oplus - k$ 
apply (case-tac  $t \leq k$ )
  apply (simp add: cut-ge-0-all iT-Plus-neg-cut-ge)
apply (simp add: linorder-not-le iT-Plus-neg-image-conv)
apply (simp add: i-cut-commute-disj[of ( $\downarrow_{\geq}$ ) ( $\downarrow_{\geq}$ )])

```

**apply** (rule *i-cut-image*[*OF sub-left-strict-mono-on*])  
**apply** (simp add: *cut-ge-Int-conv*)  
**done**

**lemma** *iT-Plus-neg-cut-greater2*:  $k \leq t \implies (I \oplus - k) \downarrow > (t - k) = (I \downarrow > t) \oplus - k$   
**apply** (simp add: *iT-Plus-neg-image-conv*)  
**apply** (simp add: *i-cut-commute-disj*[of ( $\downarrow >$ ) ( $\downarrow \geq$ )])  
**apply** (rule *i-cut-image*[*OF sub-left-strict-mono-on*])  
**apply** (simp add: *cut-ge-Int-conv*)  
**done**

**lemma** *iT-Plus-neg-cut-le*:  $(I \oplus - k) \downarrow \leq t = I \downarrow \leq (t + k) \oplus - k$   
**by** (insert *iT-Plus-neg-cut-le2*[of  $k$   $t + k$   $I$ , *OF le-add2*], simp)

**lemma** *iT-Plus-neg-cut-less*:  $(I \oplus - k) \downarrow < t = I \downarrow < (t + k) \oplus - k$   
**by** (insert *iT-Plus-neg-cut-less2*[of  $I$   $k$   $t + k$ ], simp)

**lemma** *iT-Plus-neg-cut-ge*:  $(I \oplus - k) \downarrow \geq t = I \downarrow \geq (t + k) \oplus - k$   
**by** (insert *iT-Plus-neg-cut-ge2*[of  $I$   $k$   $t + k$ ], simp)

**lemma** *iT-Plus-neg-cut-greater*:  $(I \oplus - k) \downarrow > t = I \downarrow > (t + k) \oplus - k$   
**by** (insert *iT-Plus-neg-cut-greater2*[of  $k$   $t + k$   $I$ ], simp)

**lemma** *iT-Minus-cut-le2*:  $t \leq k \implies (k \ominus I) \downarrow \leq (k - t) = k \ominus (I \downarrow \geq t)$   
**by** (fastforce simp: *i-cut-mem-iff iT-Minus-mem-iff*)

**lemma** *iT-Minus-cut-less2*:  $(k \ominus I) \downarrow < (k - t) = k \ominus (I \downarrow > t)$   
**by** (fastforce simp: *i-cut-mem-iff iT-Minus-mem-iff*)

**lemma** *iT-Minus-cut-ge2*:  $(k \ominus I) \downarrow \geq (k - t) = k \ominus (I \downarrow \leq t)$   
**by** (fastforce simp: *i-cut-mem-iff iT-Minus-mem-iff*)

**lemma** *iT-Minus-cut-greater2*:  $t \leq k \implies (k \ominus I) \downarrow > (k - t) = k \ominus (I \downarrow < t)$   
**by** (fastforce simp: *i-cut-mem-iff iT-Minus-mem-iff*)

**lemma** *iT-Minus-cut-le*:  $(k \ominus I) \downarrow \leq t = k \ominus (I \downarrow \geq (k - t))$   
**by** (fastforce simp: *i-cut-mem-iff iT-Minus-mem-iff*)

**lemma** *iT-Minus-cut-less*:  
 $(k \ominus I) \downarrow < t = (\text{if } t \leq k \text{ then } k \ominus (I \downarrow > (k - t)) \text{ else } k \ominus I)$   
**apply** (case-tac  $t \leq k$ )  
**apply** (cut-tac *iT-Minus-cut-less2*[of  $k$   $I$   $k - t$ ], simp)  
**apply** (fastforce simp: *i-cut-mem-iff iT-Minus-mem-iff*)  
**done**

**lemma** *iT-Minus-cut-ge*:  
 $(k \ominus I) \downarrow \geq t = (\text{if } t \leq k \text{ then } k \ominus (I \downarrow \leq (k - t)) \text{ else } \{\})$

**apply** (*case-tac*  $t \leq k$ )  
**apply** (*cut-tac* *iT-Minus-cut-ge2*[*of*  $k$   $I$   $k - t$ ], *simp*)  
**apply** (*fastforce simp: i-cut-mem-iff iT-Minus-mem-iff*)  
**done**

**lemma** *iT-Minus-cut-greater*:  $(k \ominus I) \downarrow > t = k \ominus (I \downarrow < (k - t))$   
**apply** (*case-tac*  $t \leq k$ )  
**apply** (*cut-tac iT-Minus-cut-greater2*[*of*  $k - t$   $k$   $I$ ], *simp+*)  
**apply** (*fastforce simp: i-cut-mem-iff iT-Minus-mem-iff*)  
**done**

**lemma** *iT-Div-cut-le*:  
 $0 < k \implies (I \odot k) \downarrow \leq t = I \downarrow < (t * k + (k - \text{Suc } 0)) \odot k$   
**apply** (*simp add: set-eq-iff i-cut-mem-iff iT-Div-mem-iff Bex-def*)  
**apply** (*fastforce simp: div-le-conv*)  
**done**

**lemma** *iT-Div-cut-less*:  
 $0 < k \implies (I \odot k) \downarrow < t = I \downarrow < (t * k) \odot k$   
**apply** (*case-tac*  $t = 0$ )  
**apply** (*simp add: cut-less-0-empty iT-Div-empty*)  
**apply** (*simp add: nat-cut-less-le-conv iT-Div-cut-le diff-mult-distrib*)  
**done**

**lemma** *iT-Div-cut-ge*:  
 $0 < k \implies (I \odot k) \downarrow \geq t = I \downarrow \geq (t * k) \odot k$   
**apply** (*simp add: set-eq-iff i-cut-mem-iff iT-Div-mem-iff Bex-def*)  
**apply** (*fastforce simp: le-div-conv*)  
**done**

**lemma** *iT-Div-cut-greater*:  
 $0 < k \implies (I \odot k) \downarrow > t = I \downarrow > (t * k + (k - \text{Suc } 0)) \odot k$   
**by** (*simp add: nat-cut-ge-greater-conv[symmetric] iT-Div-cut-ge add.commute[*of*  $k$ ]*)

**lemma** *iT-Div-cut-le2*:  
 $0 < k \implies (I \odot k) \downarrow \leq (t \text{ div } k) = I \downarrow \leq (t - t \text{ mod } k + (k - \text{Suc } 0)) \odot k$   
**by** (*frule iT-Div-cut-le[*of*  $k$   $I$   $t \text{ div } k$ ], *simp add: div-mult-cancel*)*

**lemma** *iT-Div-cut-less2*:  
 $0 < k \implies (I \odot k) \downarrow < (t \text{ div } k) = I \downarrow < (t - t \text{ mod } k) \odot k$   
**by** (*frule iT-Div-cut-less[*of*  $k$   $I$   $t \text{ div } k$ ], *simp add: div-mult-cancel*)*

**lemma** *iT-Div-cut-ge2*:  
 $0 < k \implies (I \odot k) \downarrow \geq (t \text{ div } k) = I \downarrow \geq (t - t \text{ mod } k) \odot k$   
**by** (*frule iT-Div-cut-ge[*of*  $k$   $I$   $t \text{ div } k$ ], *simp add: div-mult-cancel*)*

**lemma** *iT-Div-cut-greater2*:

$0 < k \implies (I \otimes k) \downarrow > (t \text{ div } k) = I \downarrow > (t - t \text{ mod } k + (k - \text{Suc } 0)) \otimes k$   
**by** (*frule iT-Div-cut-greater*[of  $k$   $I$   $t \text{ div } k$ ], *simp add: div-mult-cancel*)

### 2.3 *inext* and *iprev* with interval operators

**lemma** *iT-Plus-inext*:  $\text{inext } (n + k) (I \oplus k) = (\text{inext } n I) + k$

**by** (*unfold iT-Plus-def*, *rule inext-image2*[*OF add-right-strict-mono*])

**lemma** *iT-Plus-iprev*:  $\text{iprev } (n + k) (I \oplus k) = (\text{iprev } n I) + k$

**by** (*unfold iT-Plus-def*, *rule iprev-image2*[*OF add-right-strict-mono*])

**lemma** *iT-Plus-inext2*:  $k \leq n \implies \text{inext } n (I \oplus k) = (\text{inext } (n - k) I) + k$

**by** (*insert iT-Plus-inext*[of  $n - k$   $k$   $I$ ], *simp*)

**lemma** *iT-Plus-prev2*:  $k \leq n \implies \text{iprev } n (I \oplus k) = (\text{iprev } (n - k) I) + k$

**by** (*insert iT-Plus-iprev*[of  $n - k$   $k$   $I$ ], *simp*)

**lemma** *iT-Mult-inext*:  $\text{inext } (n * k) (I \otimes k) = (\text{inext } n I) * k$

**apply** (*case-tac I = {}*)

**apply** (*simp add: iT-Mult-empty inext-empty*)

**apply** (*case-tac k = 0*)

**apply** (*simp add: iT-Mult-0 iTILL-0 inext-singleton*)

**apply** (*simp add: iT-Mult-def inext-image2*[*OF mult-right-strict-mono*])

**done**

**lemma** *iT-Mult-iprev*:  $\text{iprev } (n * k) (I \otimes k) = (\text{iprev } n I) * k$

**apply** (*case-tac I = {}*)

**apply** (*simp add: iT-Mult-empty iprev-empty*)

**apply** (*case-tac k = 0*)

**apply** (*simp add: iT-Mult-0 iTILL-0 iprev-singleton*)

**apply** (*simp add: iT-Mult-def iprev-image2*[*OF mult-right-strict-mono*])

**done**

**lemma** *iT-Mult-inext2-if*:

$\text{inext } n (I \otimes k) = (\text{if } n \text{ mod } k = 0 \text{ then } (\text{inext } (n \text{ div } k) I) * k \text{ else } n)$

**apply** (*case-tac I = {}*)

**apply** (*simp add: iT-Mult-empty inext-empty div-mult-cancel*)

**apply** (*case-tac k = 0*)

**apply** (*simp add: iT-Mult-0 iTILL-0 inext-singleton*)

**apply** (*case-tac n mod k = 0*)

**apply** (*clarsimp simp: mult.commute*[of  $k$ ] *iT-Mult-inext elim!: dvdE*)

**apply** (*simp add: not-in-inext-fix iT-Mult-mem-iff*)

**done**

**lemma** *iT-Mult-iprev2-if*:

$\text{iprev } n (I \otimes k) = (\text{if } n \text{ mod } k = 0 \text{ then } (\text{iprev } (n \text{ div } k) I) * k \text{ else } n)$

**apply** (*case-tac I = {}*)

**apply** (*simp add: iT-Mult-empty iprev-empty div-mult-cancel*)

```

apply (case-tac  $k = 0$ )
  apply (simp add: iT-Mult-0 iTILL-0 iprev-singleton)
apply (case-tac  $n \bmod k = 0$ )
  apply (clarsimp simp: mult.commute[of  $k$ ] iT-Mult-iprev elim!: dvdE)
apply (simp add: not-in-iprev-fix iT-Mult-mem-iff)
done

```

```

corollary iT-Mult-inext2:
   $n \bmod k = 0 \implies \text{inext } n (I \otimes k) = (\text{inext } (n \text{ div } k) I) * k$ 
by (simp add: iT-Mult-inext2-iff)

```

```

corollary iT-Mult-iprev2:
   $n \bmod k = 0 \implies \text{iprev } n (I \otimes k) = (\text{iprev } (n \text{ div } k) I) * k$ 
by (simp add: iT-Mult-iprev2-iff)

```

```

lemma iT-Plus-neg-inext:
   $k \leq n \implies \text{inext } (n - k) (I \oplus - k) = \text{inext } n I - k$ 
apply (case-tac  $I = \{\}$ )
  apply (simp add: iT-Plus-neg-empty inext-empty)
apply (case-tac  $n \in I$ )
  apply (simp add: iT-Plus-neg-image-conv)
  apply (rule subst[OF inext-cut-ge-conv, of  $k$ ], simp)
  apply (rule inext-image)
  apply (simp add: cut-ge-mem-iff)
  apply (subst cut-ge-Int-conv)
  apply (rule strict-mono-on-subset[OF - Int-lower2])
  apply (rule sub-left-strict-mono-on)
apply (subgoal-tac  $n - k \notin I \oplus - k$ )
  prefer 2
  apply (simp add: iT-Plus-neg-mem-iff)
apply (simp add: not-in-inext-fix)
done

```

```

lemma iT-Plus-neg-iprev:
   $\text{iprev } (n - k) (I \oplus - k) = \text{iprev } n (I \downarrow \geq k) - k$ 
apply (case-tac  $I = \{\}$ )
  apply (simp add: iT-Plus-neg-empty i-cut-empty iprev-empty)
apply (case-tac  $n < k$ )
  apply (simp add: iprev-le-iMin)
  apply (simp add: order-trans[OF iprev-mono])
apply (simp add: linorder-not-less)
apply (case-tac  $n \in I$ )
  apply (frule iT-Plus-neg-mem-iff2[THEN iffD2, of - - I], assumption)
  apply (simp add: iT-Plus-neg-image-conv)
  apply (rule iprev-image)
  apply (simp add: cut-ge-mem-iff)
  apply (subst cut-ge-Int-conv)
  apply (rule strict-mono-on-subset[OF - Int-lower2])
  apply (rule sub-left-strict-mono-on)

```

**apply** (*frule cut-ge-not-in-imp*[of - -  $k$ ])  
**apply** (*subgoal-tac*  $n - k \notin I \oplus - k$ )  
**prefer** 2  
**apply** (*simp add: iT-Plus-neg-mem-iff*)  
**apply** (*simp add: not-in-iprev-fix*)  
**done**

**corollary** *iT-Plus-neg-inext2*:  $\text{inext } n (I \oplus - k) = \text{inext } (n + k) I - k$   
**by** (*insert iT-Plus-neg-inext*[of  $k$   $n + k$   $I$ , *OF le-add2*], *simp*)

**corollary** *iT-Plus-neg-iprev2*:  $\text{iprev } n (I \oplus - k) = \text{iprev } (n + k) (I \downarrow \geq k) - k$   
**by** (*insert iT-Plus-neg-iprev*[of  $n + k$   $k$   $I$ ], *simp*)

**lemma** *iT-Minus-inext*:

$\llbracket k \ominus I \neq \{\}; n \leq k \rrbracket \implies \text{inext } (k - n) (k \ominus I) = k - \text{iprev } n I$   
**apply** (*subgoal-tac* *iMin*  $I \leq k$ )  
**prefer** 2  
**apply** (*simp add: iT-Minus-empty-iff*)  
**apply** (*subgoal-tac*  $I \downarrow \leq k \neq \{\}$ )  
**prefer** 2  
**apply** (*simp add: iT-Minus-empty-iff cut-le-Min-not-empty*)  
**apply** (*case-tac*  $n \in I$ )  
**apply** (*simp add: iT-Minus-imirror-conv*)  
**apply** (*simp add: iT-Plus-neg-inext2*)  
**apply** (*subgoal-tac*  $n \leq \text{iMin } I + \text{Max } (I \downarrow \leq k)$ )  
**prefer** 2  
**apply** (*rule trans-le-add2*)  
**apply** (*rule Max-ge*[*OF nat-cut-le-finite*])  
**apply** (*simp add: cut-le-mem-iff*)  
**apply** (*simp add: diff-add-assoc del: add-diff-assoc*)  
**apply** (*subst add.commute*[of  $k$ ], *subst iT-Plus-inext*)  
**apply** (*simp add: cut-le-Min-eq*[of  $I$ , *symmetric*])  
**apply** (*fold nat-mirror-def mirror-elem-def*)  
**apply** (*simp add: inext-imirror-iprev-conv*[*OF nat-cut-le-finite*])  
**apply** (*simp add: iprev-cut-le-conv*)  
**apply** (*simp add: mirror-elem-def nat-mirror-def*)  
**apply** (*frule iprev-mono*[*THEN order-trans*, of  $n$  *iMin*  $(I \downarrow \leq k) + \text{Max } (I \downarrow \leq k)$   $I$ ])  
**apply** *simp*  
**apply** (*subgoal-tac*  $k - n \notin k \ominus I$ )  
**prefer** 2  
**apply** (*simp add: iT-Minus-mem-iff*)  
**apply** (*simp add: not-in-inext-fix not-in-iprev-fix*)  
**done**

**corollary** *iT-Minus-inext2*:

$\llbracket k \ominus I \neq \{\}; n \leq k \rrbracket \implies \text{inext } n (k \ominus I) = k - \text{iprev } (k - n) I$   
**by** (*insert iT-Minus-inext*[of  $k$   $I$   $k - n$ ], *simp*)

**lemma** *iT-Minus-iprev*:

$\llbracket k \ominus I \neq \{\}; n \leq k \rrbracket \implies \text{iprev } (k - n) (k \ominus I) = k - \text{inext } n (I \downarrow \leq k)$   
**apply** (*subgoal-tac*  $I \downarrow \leq k \neq \{\}$ )  
**prefer** 2  
**apply** (*simp add: iT-Minus-empty-iff cut-le-Min-not-empty*)  
**apply** (*subst iT-Minus-cut-eq[OF le-refl, of - I, symmetric]*)  
**apply** (*insert iT-Minus-inext2[of k k  $\ominus$  (I  $\downarrow \leq$  k) n]*)  
**apply** (*simp add: iT-Minus-Minus-cut-eq*)  
**apply** (*rule diff-diff-cancel[symmetric]*)  
**apply** (*rule order-trans[OF iprev-mono]*)  
**apply** *simp*  
**done**

**lemma** *iT-Minus-iprev2*:

$\llbracket k \ominus I \neq \{\}; n \leq k \rrbracket \implies \text{iprev } n (k \ominus I) = k - \text{inext } (k - n) (I \downarrow \leq k)$   
**by** (*insert iT-Minus-iprev[of k I k - n], simp*)

**lemma** *iT-Plus-inext-nth*:  $I \neq \{\} \implies (I \oplus k) \rightarrow n = (I \rightarrow n) + k$

**apply** (*induct n*)  
**apply** (*simp add: iT-Plus-Min*)  
**apply** (*simp add: iT-Plus-inext*)  
**done**

**lemma** *iT-Plus-iprev-nth*:  $\llbracket \text{finite } I; I \neq \{\} \rrbracket \implies (I \oplus k) \leftarrow n = (I \leftarrow n) + k$

**apply** (*induct n*)  
**apply** (*simp add: iT-Plus-Max*)  
**apply** (*simp add: iT-Plus-iprev*)  
**done**

**lemma** *iT-Mult-inext-nth*:  $I \neq \{\} \implies (I \otimes k) \rightarrow n = (I \rightarrow n) * k$

**apply** (*induct n*)  
**apply** (*simp add: iT-Mult-Min*)  
**apply** (*simp add: iT-Mult-inext*)  
**done**

**lemma** *iT-Mult-iprev-nth*:  $\llbracket \text{finite } I; I \neq \{\} \rrbracket \implies (I \otimes k) \leftarrow n = (I \leftarrow n) * k$

**apply** (*induct n*)  
**apply** (*simp add: iT-Mult-Max*)  
**apply** (*simp add: iT-Mult-iprev*)  
**done**

**lemma** *iT-Plus-neg-inext-nth*:

$I \oplus - k \neq \{\} \implies (I \oplus - k) \rightarrow n = (I \downarrow \geq k \rightarrow n) - k$   
**apply** (*subgoal-tac*  $I \downarrow \geq k \neq \{\}$ )  
**prefer** 2  
**apply** (*simp add: cut-ge-not-empty-iff iT-Plus-neg-not-empty-iff*)  
**apply** (*induct n*)

```

apply (simp add: iT-Plus-neg-Min)
apply (simp add: iT-Plus-neg-cut-eq[of k k I, symmetric])
apply (rule iT-Plus-neg-inext)
apply (rule cut-ge-bound[of - I])
apply (simp add: inext-nth-closed)
done

```

```

lemma iT-Plus-neg-iprev-nth:
   $\llbracket \text{finite } I; I \oplus - k \neq \{\} \rrbracket \implies (I \oplus - k) \leftarrow n = (I \downarrow \leq k \leftarrow n) - k$ 
apply (subgoal-tac  $I \downarrow \leq k \neq \{\}$ )
prefer 2
apply (simp add: cut-ge-not-empty-iff iT-Plus-neg-not-empty-iff)
apply (induct n)
apply (simp add: iT-Plus-neg-Max cut-ge-Max-eq)
apply (simp add: iT-Plus-neg-iprev)
done

```

```

lemma iT-Minus-inext-nth:
   $k \ominus I \neq \{\} \implies (k \ominus I) \rightarrow n = k - ((I \downarrow \leq k) \leftarrow n)$ 
apply (subgoal-tac  $I \downarrow \leq k \neq \{\} \wedge I \neq \{\} \wedge iMin I \leq k$ )
prefer 2
apply (simp add: iT-Minus-empty-iff cut-le-Min-not-empty)
apply (elim conjE)
apply (induct n)
apply (simp add: iT-Minus-Min)
apply (simp add: iT-Minus-cut-eq[OF order-refl, of - I, symmetric])
apply (rule iT-Minus-inext)
apply simp
apply (rule cut-le-bound, rule iprev-nth-closed[OF nat-cut-le-finite])
apply assumption
done

```

```

lemma iT-Minus-iprev-nth:
   $k \ominus I \neq \{\} \implies (k \ominus I) \leftarrow n = k - ((I \downarrow \leq k) \rightarrow n)$ 
apply (subgoal-tac  $I \downarrow \leq k \neq \{\} \wedge I \neq \{\} \wedge iMin I \leq k$ )
prefer 2
apply (simp add: iT-Minus-empty-iff cut-le-Min-not-empty)
apply (elim conjE)
apply (induct n)
apply (simp add: iT-Minus-Max cut-le-Min-eq)
apply simp
apply (rule iT-Minus-iprev)
apply simp
apply (rule cut-le-bound, rule inext-nth-closed)
apply assumption
done

```

```

lemma iT-Div-ge-inext-nth:
   $\llbracket I \neq \{\}; \forall x \in I. \forall y \in I. x < y \longrightarrow x + k \leq y \rrbracket \implies$ 

```

$(I \otimes k) \rightarrow n = (I \rightarrow n) \text{ div } k$   
**apply** (*case-tac*  $k = 0$ )  
**apply** (*simp add: iT-Div-0 iTILL-0 inext-nth-singleton*)  
**apply** (*simp add: iT-Div-def*)  
**by** (*rule inext-nth-image[OF - div-right-strict-mono-on]*)

**lemma** *iT-Div-mod-inext-nth*:  
 $\llbracket I \neq \{\}; \forall x \in I. \forall y \in I. x \text{ mod } k = y \text{ mod } k \rrbracket \implies$   
 $(I \otimes k) \rightarrow n = (I \rightarrow n) \text{ div } k$   
**apply** (*case-tac*  $k = 0$ )  
**apply** (*simp add: iT-Div-0 iTILL-0 inext-nth-singleton*)  
**apply** (*simp add: iT-Div-def*)  
**by** (*rule inext-nth-image[OF - mod-eq-div-right-strict-mono-on]*)

**lemma** *iT-Div-ge-iprev-nth*:  
 $\llbracket \text{finite } I; I \neq \{\}; \forall x \in I. \forall y \in I. x < y \longrightarrow x + k \leq y \rrbracket \implies$   
 $(I \otimes k) \leftarrow n = (I \leftarrow n) \text{ div } k$   
**apply** (*case-tac*  $k = 0$ )  
**apply** (*simp add: iT-Div-0 iTILL-0 iprev-nth-singleton*)  
**apply** (*simp add: iT-Div-def*)  
**by** (*rule iprev-nth-image[OF - - div-right-strict-mono-on]*)

**lemma** *iT-Div-mod-iprev-nth*:  
 $\llbracket \text{finite } I; I \neq \{\}; \forall x \in I. \forall y \in I. x \text{ mod } k = y \text{ mod } k \rrbracket \implies$   
 $(I \otimes k) \leftarrow n = (I \leftarrow n) \text{ div } k$   
**apply** (*case-tac*  $k = 0$ )  
**apply** (*simp add: iT-Div-0 iTILL-0 iprev-nth-singleton*)  
**apply** (*simp add: iT-Div-def*)  
**by** (*rule iprev-nth-image[OF - - mod-eq-div-right-strict-mono-on]*)

## 2.4 Cardinality of intervals with interval operators

**lemma** *iT-Plus-card*:  $\text{card } (I \oplus k) = \text{card } I$   
**apply** (*unfold iT-Plus-def*)  
**apply** (*rule card-image*)  
**apply** (*rule inj-imp-inj-on*)  
**apply** (*rule add-right-inj*)  
**done**

**lemma** *iT-Mult-card*:  $0 < k \implies \text{card } (I \otimes k) = \text{card } I$   
**apply** (*unfold iT-Mult-def*)  
**apply** (*rule card-image*)  
**apply** (*rule inj-imp-inj-on*)  
**apply** (*rule mult-right-inj*)  
**apply** *assumption*  
**done**

**lemma** *iT-Plus-neg-card*:  $\text{card } (I \oplus - k) = \text{card } (I \downarrow \geq k)$   
**apply** (*simp add: iT-Plus-neg-image-conv*)  
**apply** (*rule card-image*)

```

apply (subst cut-ge-Int-conv)
apply (rule inj-on-subset[OF - Int-lower2])
apply (rule sub-left-inj-on)
done

```

```

lemma iT-Plus-neg-card-le: card (I  $\oplus$  - k)  $\leq$  card I
apply (simp add: iT-Plus-neg-card)
apply (case-tac finite I)
  apply (rule cut-ge-card, assumption)
apply (simp add: nat-cut-ge-finite-iff)
done

```

```

lemma iT-Minus-card: card (k  $\ominus$  I) = card (I  $\downarrow \leq$  k)
apply (simp add: iT-Minus-image-conv)
apply (rule card-image)
apply (subst cut-le-Int-conv)
apply (rule inj-on-subset[OF - Int-lower2])
apply (rule sub-right-inj-on)
done

```

```

lemma iT-Minus-card-le: finite I  $\implies$  card (k  $\ominus$  I)  $\leq$  card I
by (subst iT-Minus-card, rule cut-le-card)

```

```

lemma iT-Div-0-card-if:
  card (I  $\oslash$  0) = (if I = {} then 0 else Suc 0)
by (fastforce simp: iT-Div-empty iT-Div-0 iTILL-0)

```

```

lemma Int-empty-sum:
  ( $\sum k \leq (n::nat)$ . if {}  $\cap$  (I k) = {} then 0 else Suc 0) = 0
apply (rule sum-eq-0-iff[THEN iffD2])
  apply (rule finite-atMost)
apply simp
done

```

```

lemma iT-Div-mod-partition-card:
  card (I  $\cap$  [n * d...d - Suc 0]  $\oslash$  d) =
  (if I  $\cap$  [n * d...d - Suc 0] = {} then 0 else Suc 0)
apply (case-tac d = 0)
  apply (simp add: iIN-0 iTILL-0 iT-Div-0-if)
apply (split if-split, rule conjI)
  apply (simp add: iT-Div-empty)
apply clarsimp
apply (subgoal-tac I  $\cap$  [n * d...d - Suc 0]  $\oslash$  d = {n}, simp)
apply (rule set-eqI)
apply (simp add: iT-Div-mem-iff Bex-def iIN-iff)
apply (rule iffI)
  apply (clarsimp simp: le-less-imp-div)
apply (drule ex-in-conv[THEN iffD2], clarsimp simp: iIN-iff, rename-tac x')
apply (rule-tac x=x' in exI)

```

**apply** (*simp add: le-less-imp-div*)  
**done**

**lemma** *iT-Div-conv-count*:  
 $0 < d \implies I \odot d = \{k. I \cap [k * d \dots, d - \text{Suc } 0] \neq \{\}\}$   
**apply** (*case-tac I = \{\}*)  
**apply** (*simp add: iT-Div-empty*)  
**apply** (*rule set-eqI*)  
**apply** (*simp add: iT-Div-mem-iff-Int*)  
**done**

**lemma** *iT-Div-conv-count2*:  
 $\llbracket 0 < d; \text{finite } I; \text{Max } I \text{ div } d \leq n \rrbracket \implies$   
 $I \odot d = \{k. k \leq n \wedge I \cap [k * d \dots, d - \text{Suc } 0] \neq \{\}\}$   
**apply** (*simp add: iT-Div-conv-count*)  
**apply** (*rule set-eqI, simp*)  
**apply** (*rule iffI*)  
**apply** *simp*  
**apply** (*rule ccontr*)  
**apply** (*drule ex-in-conv[THEN iffD2], clarify, rename-tac x'*)  
**apply** (*clarsimp simp: linorder-not-le iIN-iff*)  
**apply** (*drule order-le-less-trans, simp*)  
**apply** (*drule div-less-conv[THEN iffD1, of - Max I], simp*)  
**apply** (*drule-tac x=x' in Max-ge, simp*)  
**apply** *simp+*  
**done**

**lemma** *mod-partition-count-Suc*:  
 $\{k. k \leq \text{Suc } n \wedge I \cap [k * d \dots, d - \text{Suc } 0] \neq \{\}\} =$   
 $\{k. k \leq n \wedge I \cap [k * d \dots, d - \text{Suc } 0] \neq \{\}\} \cup$   
 $(\text{if } I \cap [\text{Suc } n * d \dots, d - \text{Suc } 0] \neq \{\} \text{ then } \{\text{Suc } n\} \text{ else } \{\})$   
**apply** (*rule set-eqI, rename-tac x*)  
**apply** (*simp add: le-less[of - Suc n] less-Suc-eq-le*)  
**apply** (*simp add: conj-disj-distribR*)  
**apply** (*intro conjI impI*)  
**apply** *fastforce*  
**apply** (*rule iffI, clarsimp+*)  
**done**

**lemma** *iT-Div-card*:  
 $\llbracket I. \llbracket 0 < d; \text{finite } I; \text{Max } I \text{ div } d \leq n \rrbracket \implies$   
 $\text{card } (I \odot d) = (\sum k \leq n.$   
 $\text{if } I \cap [k * d \dots, d - \text{Suc } 0] = \{\} \text{ then } 0 \text{ else } \text{Suc } 0)$   
**apply** (*case-tac I = \{\}*)  
**apply** (*simp add: iT-Div-empty*)  
**apply** (*simp add: iT-Div-conv-count2*)  
**apply** (*induct n*)  
**apply** (*simp add: div-eq-0-conv iIN-0-iTILL-conv*)  
**apply** (*subgoal-tac I \cap [...d - Suc 0] \neq \{\}*)

```

prefer 2
apply (simp add: ex-in-conv[symmetric], fastforce)
apply (simp add: card-1-singleton-conv)
apply (rule-tac x=0 in exI)
apply (rule set-eqI)
apply (simp add: ex-in-conv[symmetric], fastforce)
apply simp
apply (simp add: mod-partition-count-Suc)
apply (drule-tac x=I ∩ [...n * d + d - Suc 0] in meta-spec)
apply simp
apply (case-tac I ∩ [...n * d + d - Suc 0] = {})
apply simp
apply (subgoal-tac {k. k ≤ n ∧ I ∩ [k * d..., d - Suc 0] ≠ {}} = {}, simp)
apply (clarsimp, rename-tac x)
apply (subgoal-tac I ∩ [x * d..., d - Suc 0] ⊆ I ∩ [...n * d + d - Suc 0], simp)
apply (rule Int-mono[OF order-refl])
apply (simp add: iIN-iTILL-subset-conv)
apply (simp add: diff-le-mono)
apply (subgoal-tac Max (I ∩ [...n * d + d - Suc 0]) div d ≤ n)
prefer 2
apply (simp add: div-le-conv add.commute[of d] iTILL-iff)
apply (subgoal-tac ∧k. k ≤ n ⇒ [...n * d + d - Suc 0] ∩ [k * d..., d - Suc 0]
= [k * d..., d - Suc 0])
prefer 2
apply (subst Int-commute)
apply (simp add: iTILL-def cut-le-Int-conv[symmetric])
apply (rule cut-le-Max-all[OF iIN-finite])
apply (simp add: iIN-Max diff-le-mono)
apply (simp add: Int-assoc)
apply (subgoal-tac
{k. k ≤ n ∧ I ∩ ([...n * d + d - Suc 0] ∩ [k * d..., d - Suc 0]) ≠ {}} =
{k. k ≤ n ∧ I ∩ [k * d..., d - Suc 0] ≠ {}})
prefer 2
apply (rule set-eqI, rename-tac x)
apply simp
apply (rule conj-cong, simp)
apply simp
apply simp
done

```

**corollary** *iT-Div-card-Suc*:

$\bigwedge I. [0 < d; \text{finite } I; \text{Max } I \text{ div } d \leq n] \implies$

$\text{card } (I \circlearrowleft d) = (\sum k < \text{Suc } n.$

$\text{if } I \cap [k * d..., d - \text{Suc } 0] = \{\} \text{ then } 0 \text{ else } \text{Suc } 0)$

**by** (simp add: lessThan-Suc-atMost iT-Div-card)

**corollary** *iT-Div-Max-card*:  $[0 < d; \text{finite } I] \implies$

$\text{card } (I \circlearrowleft d) = (\sum k \leq \text{Max } I \text{ div } d.$

$\text{if } I \cap [k * d..., d - \text{Suc } 0] = \{\} \text{ then } 0 \text{ else } \text{Suc } 0)$

**by** (simp add: iT-Div-card)

**lemma** *iT-Div-card-le*:  $0 < k \implies \text{card } (I \otimes k) \leq \text{card } I$   
**apply** (*case-tac finite I*)  
**apply** (*simp add: iT-Div-def card-image-le*)  
**apply** (*simp add: iT-Div-finite-iff*)  
**done**

**lemma** *iT-Div-card-inj-on*:  
*inj-on*  $(\lambda n. n \text{ div } k) I \implies \text{card } (I \otimes k) = \text{card } I$   
**by** (*unfold iT-Div-def, rule card-image*)

**lemma** *mod-Suc'*:  
 $0 < n \implies \text{Suc } m \text{ mod } n = (\text{if } m \text{ mod } n < n - \text{Suc } 0 \text{ then } \text{Suc } (m \text{ mod } n) \text{ else } 0)$   
**apply** (*simp add: mod-Suc*)  
**apply** (*intro conjI impI*)  
**apply** *simp*  
**apply** (*insert le-neq-trans[OF mod-less-divisor[THEN Suc-leI, of n m]], simp*)  
**done**

**lemma** *div-Suc*:  
 $0 < n \implies \text{Suc } m \text{ div } n = (\text{if } \text{Suc } (m \text{ mod } n) = n \text{ then } \text{Suc } (m \text{ div } n) \text{ else } m \text{ div } n)$   
**apply** (*drule Suc-leI, drule le-imp-less-or-eq*)  
**apply** (*case-tac n = Suc 0, simp*)  
**apply** (*split if-split, intro conjI impI*)  
**apply** (*rule-tac t=Suc m and s=m + 1 in subst, simp*)  
**apply** (*subst div-add1-eq2, simp+*)  
**apply** (*insert le-neq-trans[OF mod-less-divisor[THEN Suc-leI, of n m]], simp*)  
**apply** (*rule-tac t=Suc m and s=m + 1 in subst, simp*)  
**apply** (*subst div-add1-eq1, simp+*)  
**done**

**lemma** *div-Suc'*:  
 $0 < n \implies \text{Suc } m \text{ div } n = (\text{if } m \text{ mod } n < n - \text{Suc } 0 \text{ then } m \text{ div } n \text{ else } \text{Suc } (m \text{ div } n))$   
**apply** (*simp add: div-Suc*)  
**apply** (*intro conjI impI*)  
**apply** *simp*  
**apply** (*insert le-neq-trans[OF mod-less-divisor[THEN Suc-leI, of n m]], simp*)  
**done**

**lemma** *iT-Div-card-ge-aux*:  
 $\bigwedge I. \llbracket 0 < d; \text{finite } I; \text{Max } I \text{ div } d \leq n \rrbracket \implies$

```

    card I div d + (if card I mod d = 0 then 0 else Suc 0) ≤ card (I ⊙ d)
apply (induct n)
apply (case-tac I = {}, simp)
apply (frule-tac m=d and n=Max I and k=0 in div-le-conv[THEN iffD1, rule-format],
assumption)
apply (simp del: Max-le-iff)
apply (subgoal-tac I ⊙ d = {0})
prefer 2
apply (rule set-eqI)
apply (simp add: iT-Div-mem-iff)
apply (rule iffI)
apply (fastforce simp: div-eq-0-conv')
apply fastforce
apply (simp add: iT-Div-singleton card-singleton del: Max-le-iff)
apply (drule Suc-le-mono[THEN iffD2, of - d - Suc 0])
apply (drule order-trans[OF nat-card-le-Max])
apply (simp, intro conjI impI)
apply (drule div-le-mono[of - d d])
apply simp
apply (subgoal-tac card I ≠ d, simp)
apply clarsimp
apply (drule order-le-less[THEN iffD1], erule disjE)
apply simp
apply (rule-tac t=I and s=I ∩ [...n * d + d - Suc 0] ∪ I ∩ [Suc n * d..., d -
Suc 0] in subst)
apply (simp add: Int-Un-distrib[symmetric] add commute[of d])
apply (subst iIN-0-iTILL-conv[symmetric])
apply (simp add: iIN-union)
apply (rule Int-absorb2)
apply (simp add: iIN-0-iTILL-conv iTILL-def)
apply (case-tac I = {}, simp)
apply (simp add: subset-atMost-Max-le-conv le-less-div-conv[symmetric] less-eq-le-pred[symmetric]
add commute[of d])
apply (cut-tac A=I ∩ [...n * d + d - Suc 0] and B=I ∩ [Suc n * d..., d - Suc
0] in card-Un-disjoint)
apply simp
apply simp
apply (clarsimp simp: disjoint-iff-in-not-in1 iT-iff)
apply (case-tac I ∩ [...n * d + d - Suc 0] = {})
apply (simp add: iT-Div-mod-partition-card del: mult-Suc)
apply (intro conjI impI)
apply (rule div-le-conv[THEN iffD2], assumption)
apply simp
apply (rule order-trans[OF Int-card2[OF iIN-finite]])
apply (simp add: iIN-card)
apply (cut-tac A=I and n=Suc n * d and d=d - Suc 0 in Int-card2[OF
iIN-finite, rule-format])
apply (simp add: iIN-card)
apply (drule order-le-less[THEN iffD1], erule disjE)

```

```

apply simp
apply simp
apply (subgoal-tac Max ( $I \cap [\dots n * d + d - \text{Suc } 0]$ ) div  $d \leq n$ )
prefer 2
apply (rule div-le-conv[THEN iffD2], assumption)
apply (rule order-trans[OF Max-Int-le2[OF - iTILL-finite]], assumption)
apply (simp add: iTILL-Max)
apply (simp only: iT-Div-Un)
apply (cut-tac  $A=I \cap [\dots n * d + d - \text{Suc } 0] \odot d$  and  $B=I \cap [\text{Suc } n * d \dots, d - \text{Suc } 0] \odot d$  in card-Un-disjoint)
  apply (simp add: iT-Div-finite-iff)
  apply (simp add: iT-Div-finite-iff)
apply (subst iIN-0-iTILL-conv[symmetric])
apply (subst mod-partition-iT-Div-Int-one-segment, simp)
apply (cut-tac  $n=0$  and  $d=n * d+d$  and  $k=d$  and  $A=I$  in mod-partition-iT-Div-Int2, simp+)
apply (rule disjoint-iff-in-not-in1[THEN iffD2])
apply clarsimp
apply (simp add: iIN-div-mod-eq-0)
apply (simp add: mod-0-imp-sub-1-div-conv iIN-0-iTILL-conv iIN-0 iTILL-iff)
apply (simp only: iT-Div-mod-partition-card)
apply (subgoal-tac finite ( $I \cap [\dots n * d + d - \text{Suc } 0]$ ))
prefer 2
apply simp
apply simp
apply (intro conjI impI)
apply (rule add-le-divisor-imp-le-Suc-div)
  apply (rule add-leD1, blast)
apply (rule Int-card2[OF iIN-finite, THEN le-trans])
apply (simp add: iIN-card)
apply (cut-tac  $A=I$  and  $n=\text{Suc } n * d$  and  $d=d - \text{Suc } 0$  in Int-card2[OF iIN-finite, rule-format])
apply (simp add: iIN-card)
apply (rule-tac  $y=\text{let } I=I \cap [\dots n * d + d - \text{Suc } 0]$  in
  card  $I \text{ div } d + (\text{if } \text{card } I \text{ mod } d = 0 \text{ then } 0 \text{ else } \text{Suc } 0)$  in order-trans)
prefer 2
apply (simp add: Let-def)
apply (unfold Let-def)
apply (split if-split, intro conjI impI)
apply (subgoal-tac card ( $I \cap [\text{Suc } n * d \dots, d - \text{Suc } 0]$ )  $\neq d$ )
prefer 2
apply (rule ccontr, simp)
apply (simp add: div-add1-eq1-mod-0-left)
apply (simp add: add-le-divisor-imp-le-Suc-div)
done

```

**lemma** *iT-Div-card-ge*:

```

  card  $I \text{ div } d + (\text{if } \text{card } I \text{ mod } d = 0 \text{ then } 0 \text{ else } \text{Suc } 0) \leq \text{card } (I \odot d)$ 
apply (case-tac  $I = \{\}$ , simp)

```

```

apply (case-tac finite I)
  prefer 2
  apply simp
apply (case-tac d = 0)
  apply (simp add: iT-Div-0 iTILL-0)
apply (simp add: iT-Div-card-ge-aux[OF - - order-refl])
done

```

**corollary** *iT-Div-card-ge-div*:  $\text{card } I \text{ div } d \leq \text{card } (I \circlearrowright d)$   
**by** (rule iT-Div-card-ge[THEN add-leD1])

There is no better lower bound function  $f$  for  $i \circlearrowright d$  with  $\text{card } i$  and  $d$  as arguments.

**lemma** *iT-Div-card-ge--is-maximal-lower-bound*:

```

 $\forall I d. \text{card } I \text{ div } d + (\text{if } \text{card } I \bmod d = 0 \text{ then } 0 \text{ else } \text{Suc } 0) \leq f (\text{card } I) d \wedge$ 
 $f (\text{card } I) d \leq \text{card } (I \circlearrowright d) \implies$ 
 $f (\text{card } (I::\text{nat set})) d = \text{card } I \text{ div } d + (\text{if } \text{card } I \bmod d = 0 \text{ then } 0 \text{ else } \text{Suc } 0)$ 
apply (case-tac I = {})
  apply (erule-tac x=I in allE, erule-tac x=d in allE)
  apply (simp add: iT-Div-empty)
apply (case-tac d = 0)
  apply (frule-tac x={} in spec, erule-tac x=I in allE)
  apply (erule-tac x=d in allE, erule-tac x=d in allE)
  apply (clarsimp simp: iT-Div-0 iTILL-card iT-Div-empty)
apply (rule order-antisym)
  prefer 2
  apply simp
apply (case-tac finite I)
  prefer 2
  apply (erule-tac x=I in allE, erule-tac x=d in allE)
  apply (simp add: iT-Div-finite-iff)
apply (erule-tac x=[...card I - Suc 0] in allE, erule-tac x=d in allE, elim conjE)
apply (frule not-empty-card-gr0-conv[THEN iffD1], assumption)
apply (simp add: iTILL-card iTILL-div)
apply (intro conjI impI)
  apply (simp add: mod-0-imp-sub-1-div-conv)
  apply (subgoal-tac d ≤ card I)
  prefer 2
  apply (clarsimp elim!: dvdE)
  apply (drule div-le-mono[of d - d])
  apply simp
apply (case-tac d = Suc 0, simp)
apply (simp add: div-diff1-eq1)
done

```

**lemma** *iT-Plus-icard*:  $\text{icard } (I \oplus k) = \text{icard } I$   
**by** (simp add: iT-Plus-def icard-image)

**lemma** *iT-Mult-icard*:  $0 < k \implies \text{icard } (I \otimes k) = \text{icard } I$   
**apply** (*unfold iT-Mult-def*)  
**apply** (*rule icard-image*)  
**apply** (*rule inj-imp-inj-on*)  
**apply** (*simp add: mult-right-inj*)  
**done**

**lemma** *iT-Plus-neg-icard*:  $\text{icard } (I \oplus - k) = \text{icard } (I \downarrow_{\geq} k)$   
**apply** (*case-tac finite I*)  
**apply** (*simp add: iT-Plus-neg-finite-iff cut-ge-finite icard-finite iT-Plus-neg-card*)  
**apply** (*simp add: iT-Plus-neg-finite-iff nat-cut-ge-finite-iff*)  
**done**

**lemma** *iT-Plus-neg-icard-le*:  $\text{icard } (I \oplus - k) \leq \text{icard } I$   
**apply** (*case-tac finite I*)  
**apply** (*simp add: iT-Plus-neg-finite-iff icard-finite iT-Plus-neg-card-le*)  
**apply** *simp*  
**done**

**lemma** *iT-Minus-icard*:  $\text{icard } (k \ominus I) = \text{icard } (I \downarrow_{\leq} k)$   
**by** (*simp add: icard-finite iT-Minus-finite nat-cut-le-finite iT-Minus-card*)

**lemma** *iT-Minus-icard-le*:  $\text{icard } (k \ominus I) \leq \text{icard } I$   
**apply** (*case-tac finite I*)  
**apply** (*simp add: icard-finite iT-Minus-finite iT-Minus-card-le*)  
**apply** *simp*  
**done**

**lemma** *iT-Div-0-icard-if*:  $\text{icard } (I \oslash 0) = \text{enat } (\text{if } I = \{\} \text{ then } 0 \text{ else } \text{Suc } 0)$   
**by** (*simp add: icard-finite iT-Div-0-finite iT-Div-0-card-if*)

**lemma** *iT-Div-mod-partition-icard*:  
 $\text{icard } (I \cap [n * d \dots, d - \text{Suc } 0] \oslash d) =$   
 $\text{enat } (\text{if } I \cap [n * d \dots, d - \text{Suc } 0] = \{\} \text{ then } 0 \text{ else } \text{Suc } 0)$   
**apply** (*subgoal-tac finite (I \cap [n \* d \dots, d - Suc 0] \oslash d)*)  
**prefer** 2  
**apply** (*case-tac d = 0, simp add: iT-Div-0-finite*)  
**apply** (*simp add: iT-Div-finite-iff iIN-finite*)  
**apply** (*simp add: icard-finite iT-Div-mod-partition-card*)  
**done**

**lemma** *iT-Div-icard*:  
 $\llbracket 0 < d; \text{finite } I \implies \text{Max } I \text{ div } d \leq n \rrbracket \implies$   
 $\text{icard } (I \oslash d) =$   
 $(\text{if } \text{finite } I \text{ then } \text{enat } (\sum_{k \leq n}. \text{if } I \cap [k * d \dots, d - \text{Suc } 0] = \{\} \text{ then } 0 \text{ else } \text{Suc } 0) \text{ else } \infty)$   
**by** (*simp add: icard-finite iT-Div-finite-iff iT-Div-card*)

**corollary** *iT-Div-Max-icard*:  $0 < d \implies$

$icard (I \odot d) = (if \text{finite } I$   
 $\text{ then enat } (\sum_{k \leq \text{Max } I \text{ div } d. if I \cap [k * d.., d - \text{Suc } 0] = \{\}} \text{ then } 0 \text{ else Suc}$   
 $0) \text{ else } \infty)$   
**by** (*simp add: iT-Div-icard*)

**lemma** *iT-Div-icard-le*:  $0 < k \implies icard (I \odot k) \leq icard I$   
**apply** (*case-tac finite I*)  
**apply** (*simp add: iT-Div-finite-iff icard-finite iT-Div-card-le*)  
**apply** *simp*  
**done**

**lemma** *iT-Div-icard-inj-on*:  $inj\text{-on } (\lambda n. n \text{ div } k) I \implies icard (I \odot k) = icard I$   
**by** (*simp add: iT-Div-def icard-image*)

**lemma** *iT-Div-icard-ge*:  $icard I \text{ div } (enat d) + enat (if icard I \text{ mod } (enat d) = 0$   
 $\text{ then } 0 \text{ else Suc } 0) \leq icard (I \odot d)$   
**apply** (*case-tac d = 0*)  
**apply** (*simp add: icard-finite iT-Div-0-finite*)  
**apply** (*case-tac icard I*)  
**apply** (*fastforce simp: iT-Div-0-card-if*)  
**apply** (*simp add: iT-Div-0-card-if icard-infinite-conv infinite-imp-nonempty*)  
**apply** (*case-tac finite I*)  
**apply** (*simp add: iT-Div-finite-iff icard-finite iT-Div-card-ge*)  
**apply** (*simp add: iT-Div-finite-iff*)  
**done**

**corollary** *iT-Div-icard-ge-div*:  $icard I \text{ div } (enat d) \leq icard (I \odot d)$   
**by** (*rule iT-Div-icard-ge[THEN iadd-ileD1]*)

**lemma** *iT-Div-icard-ge--is-maximal-lower-bound*:

$\forall I d. icard I \text{ div } (enat d) + enat (if icard I \text{ mod } (enat d) = 0 \text{ then } 0 \text{ else Suc } 0)$   
 $\leq f (icard I) d \wedge$   
 $f (icard I) d \leq icard (I \odot d) \implies$   
 $f (icard (I::nat \text{ set})) d =$   
 $icard I \text{ div } (enat d) + enat (if icard I \text{ mod } (enat d) = 0 \text{ then } 0 \text{ else Suc } 0)$   
**apply** (*case-tac d = 0*)  
**apply** (*drule-tac x=I in spec, drule-tac x=d in spec, erule conjE*)  
**apply** (*simp add: iT-Div-0-icard-if icard-0-eq[unfolded zero-enat-def]*)  
**apply** (*case-tac finite I*)  
**prefer** 2  
**apply** (*drule-tac x=I in spec, drule-tac x=d in spec*)  
**apply** *simp*  
**apply** *simp*  
**apply** (*frule-tac iT-Div-finite-iff[THEN iffD2], assumption*)  
**apply** (*cut-tac f= $\lambda c d. the\text{-enat } (f (enat c) d)$  and  $I=I$  and  $d=d$  in iT-Div-card-ge--is-maximal-lower-bound*)  
**apply** (*intro allI, rename-tac I' d'*)  
**apply** (*subgoal-tac  $\wedge k. f 0 k = 0$* )  
**prefer** 2

```

apply (drule-tac x={ } in spec, drule-tac x=k in spec, erule conjE)
apply (simp add: iT-Div-empty)
apply (drule-tac x=I' in spec, drule-tac x=d' in spec, erule conjE)
apply (case-tac d' = 0)
apply (simp add: idiv-by-0 imod-by-0 iT-Div-0-card-if iT-Div-0-icard-if)
apply (case-tac I' = { }, simp)
apply (case-tac finite I')
  apply (simp add: icard-finite)
apply simp
apply simp
apply (case-tac finite I')
  apply (frule-tac I=I' and k=d' in iT-Div-finite-iff[THEN iffD2, rule-format],
assumption)
  apply (simp add: icard-finite)
  apply (subgoal-tac  $\exists n. f(\text{enat}(\text{card } I')) d' = \text{enat } n$ )
    prefer 2
    apply (rule enat-ile, assumption)
  apply clarsimp
apply (subgoal-tac infinite (I'  $\circ$  d'))
  prefer 2
  apply (simp add: iT-Div-finite-iff)
apply simp
apply (drule-tac x=I in spec, drule-tac x=d in spec, erule conjE)
apply (simp add: icard-finite)
apply (subgoal-tac  $\exists n. f(\text{enat}(\text{card } I)) d = \text{enat } n$ )
  prefer 2
  apply (rule enat-ile, assumption)
apply clarsimp
done

```

## 2.5 Results about sets of intervals

### 2.5.1 Set of intervals without and with empty interval

**definition** *iFROM-UN-set* :: (nat set) set  
**where** *iFROM-UN-set*  $\equiv \bigcup n. \{[n..]\}$

**definition** *iTILL-UN-set* :: (nat set) set  
**where** *iTILL-UN-set*  $\equiv \bigcup n. \{[..n]\}$

**definition** *iIN-UN-set* :: (nat set) set  
**where** *iIN-UN-set*  $\equiv \bigcup n d. \{[n..,d]\}$

**definition** *iMOD-UN-set* :: (nat set) set  
**where** *iMOD-UN-set*  $\equiv \bigcup r m. \{[r, \text{mod } m]\}$

**definition** *iMODb-UN-set* :: (nat set) set  
**where** *iMODb-UN-set*  $\equiv \bigcup r m c. \{[r, \text{mod } m, c]\}$

**definition** *iFROM-set* :: (nat set) set  
**where** *iFROM-set*  $\equiv \{[n..] \mid n. \text{True}\}$

**definition** *iTILL-set* :: (nat set) set  
**where** *iTILL-set*  $\equiv \{[..n] \mid n. \text{True}\}$

**definition** *iIN-set* :: (nat set) set  
**where** *iIN-set*  $\equiv \{[n..,d] \mid n \ d. \text{True}\}$

**definition** *iMOD-set* :: (nat set) set  
**where** *iMOD-set*  $\equiv \{[r, \text{mod } m] \mid r \ m. \text{True}\}$

**definition** *iMODb-set* :: (nat set) set  
**where** *iMODb-set*  $\equiv \{[r, \text{mod } m, c] \mid r \ m \ c. \text{True}\}$

**definition** *iMOD2-set* :: (nat set) set  
**where** *iMOD2-set*  $\equiv \{[r, \text{mod } m] \mid r \ m. 2 \leq m\}$

**definition** *iMODb2-set* :: (nat set) set  
**where** *iMODb2-set*  $\equiv \{[r, \text{mod } m, c] \mid r \ m \ c. 2 \leq m \wedge 1 \leq c\}$

**definition** *iMOD2-UN-set* :: (nat set) set  
**where** *iMOD2-UN-set*  $\equiv \bigcup r. \bigcup m \in \{2..\}. \{[r, \text{mod } m]\}$

**definition** *iMODb2-UN-set* :: (nat set) set  
**where** *iMODb2-UN-set*  $\equiv \bigcup r. \bigcup m \in \{2..\}. \bigcup c \in \{1..\}. \{[r, \text{mod } m, c]\}$

**lemmas** *i-set-defs* =  
*iFROM-set-def* *iTILL-set-def* *iIN-set-def*  
*iMOD-set-def* *iMODb-set-def*  
*iMOD2-set-def* *iMODb2-set-def*

**lemmas** *i-UN-set-defs* =  
*iFROM-UN-set-def* *iTILL-UN-set-def* *iIN-UN-set-def*  
*iMOD-UN-set-def* *iMODb-UN-set-def*  
*iMOD2-UN-set-def* *iMODb2-UN-set-def*

**lemma** *iFROM-set-UN-set-eq*: *iFROM-set* = *iFROM-UN-set*  
**by** (*fastforce simp: iFROM-set-def iFROM-UN-set-def*)

**lemma**  
*iTILL-set-UN-set-eq*: *iTILL-set* = *iTILL-UN-set* **and**  
*iIN-set-UN-set-eq*: *iIN-set* = *iIN-UN-set* **and**  
*iMOD-set-UN-set-eq*: *iMOD-set* = *iMOD-UN-set* **and**  
*iMODb-set-UN-set-eq*: *iMODb-set* = *iMODb-UN-set*  
**by** (*fastforce simp: i-set-defs i-UN-set-defs*)**+**

**lemma** *iMOD2-set-UN-set-eq*: *iMOD2-set* = *iMOD2-UN-set*

**by** (*fastforce simp: i-set-defs i-UN-set-defs*)

**lemma** *iMODb2-set-UN-set-eq*:  $iMODb2\text{-set} = iMODb2\text{-UN-set}$   
**by** (*fastforce simp: i-set-defs i-UN-set-defs*)

**lemmas** *i-set-i-UN-set-sets-eq* =  
*iFROM-set-UN-set-eq*  
*iTILL-set-UN-set-eq*  
*iIN-set-UN-set-eq*  
*iMOD-set-UN-set-eq*  
*iMODb-set-UN-set-eq*  
*iMOD2-set-UN-set-eq*  
*iMODb2-set-UN-set-eq*

**lemma** *iMOD2-set-iMOD-set-subset*:  $iMOD2\text{-set} \subseteq iMOD\text{-set}$   
**by** (*fastforce simp: i-set-defs*)

**lemma** *iMODb2-set-iMODb-set-subset*:  $iMODb2\text{-set} \subseteq iMODb\text{-set}$   
**by** (*fastforce simp: i-set-defs*)

**definition** *i-set* :: (nat set) set  
**where**  $i\text{-set} \equiv iFROM\text{-set} \cup iTILL\text{-set} \cup iIN\text{-set} \cup iMOD\text{-set} \cup iMODb\text{-set}$

**definition** *i-UN-set* :: (nat set) set  
**where**  $i\text{-UN-set} \equiv iFROM\text{-UN-set} \cup iTILL\text{-UN-set} \cup iIN\text{-UN-set} \cup iMOD\text{-UN-set} \cup iMODb\text{-UN-set}$

Minimal definitions for *i-set* and *i-set*

**definition** *i-set-min* :: (nat set) set  
**where**  $i\text{-set-min} \equiv iFROM\text{-set} \cup iIN\text{-set} \cup iMOD2\text{-set} \cup iMODb2\text{-set}$

**definition** *i-UN-set-min* :: (nat set) set  
**where**  $i\text{-UN-set-min} \equiv iFROM\text{-UN-set} \cup iIN\text{-UN-set} \cup iMOD2\text{-UN-set} \cup iMODb2\text{-UN-set}$

**definition** *i-set0* :: (nat set) set  
**where**  $i\text{-set0} \equiv \text{insert } \{\} i\text{-set}$

**lemma** *i-set-i-UN-set-eq*:  $i\text{-set} = i\text{-UN-set}$   
**by** (*simp add: i-set-def i-UN-set-def i-set-i-UN-set-sets-eq*)

**lemma** *i-set-min-i-UN-set-min-eq*:  $i\text{-set-min} = i\text{-UN-set-min}$   
**by** (*simp add: i-set-min-def i-UN-set-min-def i-set-i-UN-set-sets-eq*)

**lemma** *i-set-min-eq*:  $i\text{-set} = i\text{-set-min}$   
**apply** (*unfold i-set-def i-set-min-def*)  
**apply** (*rule equalityI*)  
**apply** (*rule subsetI*)

```

apply (simp add: i-set-defs)
apply (elim disjE)
  apply blast
  apply (fastforce simp: iIN-0-iTILL-conv[symmetric])
  apply blast
  apply (elim exE)
  apply (case-tac 2 ≤ m, blast)
  apply (simp add: nat-ge2-conv)
  apply (fastforce simp: iMOD-0 iMOD-1)
  apply (elim exE)
  apply (case-tac 1 ≤ c)
  apply (case-tac 2 ≤ m, fastforce)
  apply (simp add: nat-ge2-conv)
  apply (fastforce simp: iMODb-mod-0 iMODb-mod-1)
  apply (fastforce simp: linorder-not-le less-Suc-eq-le iMODb-0)
apply (rule Un-mono)+
apply (simp-all add: iMOD2-set-iMOD-set-subset iMODb2-set-iMODb-set-subset)
done

```

**corollary** *i-UN-set-i-UN-min-set-eq: i-UN-set = i-UN-set-min*

**by** (simp add: i-set-min-eq i-set-i-UN-set-eq[symmetric] i-set-min-i-UN-set-min-eq[symmetric])

**lemma** *i-set-min-is-minimal-let:*

*let s1 = iFROM-set; s2 = iIN-set; s3 = iMOD2-set; s4 = iMODb2-set in*  
 $s1 \cap s2 = \{\}$   $\wedge$   $s1 \cap s3 = \{\}$   $\wedge$   $s1 \cap s4 = \{\}$   $\wedge$   
 $s2 \cap s3 = \{\}$   $\wedge$   $s2 \cap s4 = \{\}$   $\wedge$   $s3 \cap s4 = \{\}$

**apply** (unfold Let-def i-set-defs, intro conjI)

**apply** (rule disjoint-iff-in-not-in1 [THEN iffD2], clarsimp simp: iT-neq)+

**done**

**lemmas** *i-set-min-is-minimal = i-set-min-is-minimal-let[simplified]*

**inductive-set** *i-set-ind:: (nat set) set*

**where**

*i-set-ind-FROM[intro!]: [n..] ∈ i-set-ind*  
 $|$  *i-set-ind-TILL[intro!]: [...n] ∈ i-set-ind*  
 $|$  *i-set-ind-IN[intro!]: [n..,d] ∈ i-set-ind*  
 $|$  *i-set-ind-MOD[intro!]: [r, mod m] ∈ i-set-ind*  
 $|$  *i-set-ind-MODb[intro!]: [r, mod m, c] ∈ i-set-ind*

**inductive-set** *i-set0-ind :: (nat set) set*

**where**

*i-set0-ind-empty[intro!]: {} ∈ i-set0-ind*  
 $|$  *i-set0-ind-i-set[intro!]: I ∈ i-set-ind  $\implies$  I ∈ i-set0-ind*

The introduction rule *i-set0-ind-i-set* is not declared a safe introduction rule, because it would disturb the correct usage of the *safe* method.

**lemma** *i-set-ind-subset-i-set0-ind: i-set-ind  $\subseteq$  i-set0-ind*

by (rule subsetI, rule i-set0-ind-i-set)

**lemma**

*i-set0-ind-FROM*[intro!] :  $[n..] \in i\text{-set0-ind}$  **and**  
*i-set0-ind-TILL*[intro!] :  $[\dots n] \in i\text{-set0-ind}$  **and**  
*i-set0-ind-IN*[intro!] :  $[n..,d] \in i\text{-set0-ind}$  **and**  
*i-set0-ind-MOD*[intro!] :  $[r, \text{mod } m] \in i\text{-set0-ind}$  **and**  
*i-set0-ind-MODb*[intro!] :  $[r, \text{mod } m, c] \in i\text{-set0-ind}$

by (rule subsetD[OF i-set-ind-subset-i-set0-ind], rule i-set-ind.intros)+

**lemmas** *i-set0-ind-intros2* =

*i-set0-ind-empty*  
*i-set0-ind-FROM*  
*i-set0-ind-TILL*  
*i-set0-ind-IN*  
*i-set0-ind-MOD*  
*i-set0-ind-MODb*

**lemma** *i-set-i-set-ind-eq*:  $i\text{-set} = i\text{-set-ind}$

apply (rule set-eqI, unfold i-set-def i-set-defs)

apply (rule iffI, blast)

apply (induct-tac x rule: i-set-ind.induct)

apply blast+

done

**lemma** *i-set0-i-set0-ind-eq*:  $i\text{-set0} = i\text{-set0-ind}$

apply (rule set-eqI, unfold i-set0-def)

apply (simp add: i-set-i-set-ind-eq)

apply (rule iffI)

  apply blast

apply (rule-tac a=x in i-set0-ind.cases)

apply blast+

done

**lemma** *i-set-imp-not-empty*:  $I \in i\text{-set} \implies I \neq \{\}$

apply (simp add: i-set-i-set-ind-eq)

apply (induct I rule: i-set-ind.induct)

apply (rule iT-not-empty)+

done

**lemma** *i-set0-i-set-mem-conv*:  $(I \in i\text{-set0}) = (I \in i\text{-set} \vee I = \{\})$

apply (simp add: i-set-i-set-ind-eq i-set0-i-set0-ind-eq)

apply (rule iffI)

apply (rule i-set0-ind.cases[of I])

apply blast+

done

**lemma** *i-set-i-set0-mem-conv*:  $(I \in i\text{-set}) = (I \in i\text{-set0} \wedge I \neq \{\})$

apply (insert i-set-imp-not-empty[of I])

**apply** (*fastforce simp: i-set0-i-set-mem-conv*)  
**done**

**lemma** *i-set0-i-set-conv*:  $i\text{-set0} - \{\{\}\} = i\text{-set}$   
**by** (*fastforce simp: i-set-i-set0-mem-conv*)

**corollary** *i-set-subset-i-set0*:  $i\text{-set} \subseteq i\text{-set0}$   
**by** (*simp add: i-set0-i-set-conv[symmetric]*)

**lemma** *i-set-singleton*:  $\{a\} \in i\text{-set}$   
**by** (*fastforce simp: i-set-def iIN-set-def iIN-0[symmetric]*)

**lemma** *i-set0-singleton*:  $\{a\} \in i\text{-set0}$   
**apply** (*rule subsetD[OF i-set-subset-i-set0]*)  
**apply** (*simp add: iIN-0[symmetric] i-set-i-set-ind-eq i-set-ind.intros*)  
**done**

**corollary**  
*i-set-FROM*[*intro!*] :  $[n..] \in i\text{-set}$  **and**  
*i-set-TILL*[*intro!*] :  $[..n] \in i\text{-set}$  **and**  
*i-set-IN*[*intro!*] :  $[n..,d] \in i\text{-set}$  **and**  
*i-set-MOD*[*intro!*] :  $[r, \text{mod } m] \in i\text{-set}$  **and**  
*i-set-MODb*[*intro!*] :  $[r, \text{mod } m, c] \in i\text{-set}$   
**by** (*rule ssubst[OF i-set-i-set-ind-eq], rule i-set-ind.intros*)+

**lemmas** *i-set-intros* =  
*i-set-FROM*  
*i-set-TILL*  
*i-set-IN*  
*i-set-MOD*  
*i-set-MODb*

**lemma**  
*i-set0-empty*[*intro!*]:  $\{\} \in i\text{-set0}$  **and**  
*i-set0-FROM*[*intro!*] :  $[n..] \in i\text{-set0}$  **and**  
*i-set0-TILL*[*intro!*] :  $[..n] \in i\text{-set0}$  **and**  
*i-set0-IN*[*intro!*] :  $[n..,d] \in i\text{-set0}$  **and**  
*i-set0-MOD*[*intro!*] :  $[r, \text{mod } m] \in i\text{-set0}$  **and**  
*i-set0-MODb*[*intro!*] :  $[r, \text{mod } m, c] \in i\text{-set0}$   
**by** (*rule ssubst[OF i-set0-i-set0-ind-eq], rule i-set0-ind-intros2*)+

**lemmas** *i-set0-intros* =  
*i-set0-empty*  
*i-set0-FROM*  
*i-set0-TILL*  
*i-set0-IN*  
*i-set0-MOD*  
*i-set0-MODb*

**lemma** *i-set-infinite-as-iMOD*:

```

  [ I ∈ i-set; infinite I ] ⇒ ∃ r m. I = [r, mod m]
  apply (simp add: i-set-i-set-ind-eq)
  apply (induct I rule: i-set-ind.induct)
  apply (simp-all add: iT-finite)
  apply (rule-tac x=n in exI, rule-tac x=Suc 0 in exI, simp add: iMOD-1)
  apply blast
  done

```

**lemma** *i-set-finite-as-iMODb*:

```

  [ I ∈ i-set; finite I ] ⇒ ∃ r m c. I = [r, mod m, c]
  apply (simp add: i-set-i-set-ind-eq)
  apply (induct I rule: i-set-ind.induct)
  apply (simp add: iT-infinite)
  apply (rule-tac x=0 in exI, rule-tac x=Suc 0 in exI, rule-tac x=n in exI)
  apply (simp add: iMODb-mod-1 iIN-0-iTILL-conv)
  apply (rule-tac x=n in exI, rule-tac x=Suc 0 in exI, rule-tac x=d in exI)
  apply (simp add: iMODb-mod-1)
  apply (case-tac m = 0)
  apply (rule-tac x=r in exI, rule-tac x=Suc 0 in exI, rule-tac x=0 in exI)
  apply (simp add: iMOD-0 iIN-0 iMODb-0)
  apply (simp add: iT-infinite)
  apply blast
  done

```

**lemma** *i-set-as-iMOD-iMODb*:

```

  I ∈ i-set ⇒ (∃ r m. I = [r, mod m]) ∨ (∃ r m c. I = [r, mod m, c])
  by (blast intro: i-set-finite-as-iMODb i-set-infinite-as-iMOD)

```

## 2.5.2 Interval sets are closed under cutting

**lemma** *i-set-cut-le-ge-closed-disj*:

```

  [ I ∈ i-set; t ∈ I; cut-op = (↓≤) ∨ cut-op = (↓≥) ] ⇒
  cut-op I t ∈ i-set
  apply (simp add: i-set-i-set-ind-eq)
  apply (induct rule: i-set-ind.induct)
  apply safe
  apply (simp-all add: iT-cut-le1 iT-cut-ge1 i-set-ind.intros iMODb-iff)
  done

```

**corollary**

```

  i-set-cut-le-closed: [ I ∈ i-set; t ∈ I ] ⇒ I ↓≤ t ∈ i-set and
  i-set-cut-ge-closed: [ I ∈ i-set; t ∈ I ] ⇒ I ↓≥ t ∈ i-set
  by (simp-all add: i-set-cut-le-ge-closed-disj)

```

**lemmas** *i-set-cut-le-ge-closed = i-set-cut-le-closed i-set-cut-ge-closed*

**lemma** *i-set0-cut-closed-disj*:

```

[[ I ∈ i-set0;
  cut-op = (↓≤) ∨ cut-op = (↓≥) ∨
  cut-op = (↓<) ∨ cut-op = (↓>) ]] ⇒
cut-op I t ∈ i-set0
apply (simp add: i-set0-i-set0-ind-eq)
apply (induct rule: i-set0-ind.induct)
apply (rule ssubst[OF set-restriction-empty, where P=λx. x ∈ i-set0-ind])
apply (rule i-cut-set-restriction-disj[of cut-op], blast)
apply blast
apply blast
apply (induct-tac I rule: i-set-ind.induct)
apply safe
apply (simp-all add: iT-cut-le iT-cut-ge iT-cut-less iT-cut-greater i-set0-ind-intros2)
done

```

**corollary**

```

i-set0-cut-le-closed: I ∈ i-set0 ⇒ I ↓≤ t ∈ i-set0 and
i-set0-cut-less-closed: I ∈ i-set0 ⇒ I ↓< t ∈ i-set0 and
i-set0-cut-ge-closed: I ∈ i-set0 ⇒ I ↓≥ t ∈ i-set0 and
i-set0-cut-greater-closed: I ∈ i-set0 ⇒ I ↓> t ∈ i-set0
by (simp-all add: i-set0-cut-closed-disj)

```

**lemmas** *i-set0-cut-closed* =

```

i-set0-cut-le-closed
i-set0-cut-less-closed
i-set0-cut-ge-closed
i-set0-cut-greater-closed

```

### 2.5.3 Interval sets are closed under addition and multiplication

**lemma** *i-set-Plus-closed*:  $I \in i\text{-set} \implies I \oplus k \in i\text{-set}$

```

apply (simp add: i-set-i-set-ind-eq)
apply (induct rule: i-set-ind.induct)
apply (simp-all add: iT-add i-set-ind.intros)
done

```

**lemma** *i-set-Mult-closed*:  $I \in i\text{-set} \implies I \otimes k \in i\text{-set}$

```

apply (case-tac k = 0)
apply (simp add: i-set-imp-not-empty iT-Mult-0-if i-set-intros)
apply (simp add: i-set-i-set-ind-eq)
apply (induct rule: i-set-ind.induct)
apply (simp-all add: iT-mult i-set-ind.intros)
done

```

**lemma** *i-set0-Plus-closed*:  $I \in i\text{-set0} \implies I \oplus k \in i\text{-set0}$

```

apply (simp add: i-set0-i-set0-ind-eq)
apply (induct I rule: i-set0-ind.induct)
apply (simp add: iT-Plus-empty i-set0-ind-empty)

```

```

apply (rule subsetD[OF i-set-ind-subset-i-set0-ind])
apply (simp add: i-set-i-set-ind-eq[symmetric] i-set-Plus-closed)
done

```

```

lemma i-set0-Mult-closed:  $I \in i\text{-set0} \implies I \otimes k \in i\text{-set0}$ 
apply (simp add: i-set0-i-set0-ind-eq)
apply (induct I rule: i-set0-ind.induct)
  apply (simp add: iT-Mult-empty i-set0-ind-empty)
apply (rule subsetD[OF i-set-ind-subset-i-set0-ind])
apply (simp add: i-set-i-set-ind-eq[symmetric] i-set-Mult-closed)
done

```

#### 2.5.4 Interval sets are closed with certain conditions under subtraction

```

lemma i-set-Plus-neg-closed:
   $\llbracket I \in i\text{-set}; \exists x \in I. k \leq x \rrbracket \implies I \oplus - k \in i\text{-set}$ 
apply (simp add: i-set-i-set-ind-eq)
apply (induct rule: i-set-ind.induct)
apply (fastforce simp: iT-iff iT-add-neg)+
done

```

```

lemma i-set-Minus-closed:
   $\llbracket I \in i\text{-set}; i\text{Min } I \leq k \rrbracket \implies k \ominus I \in i\text{-set}$ 
apply (simp add: i-set-i-set-ind-eq)
apply (induct rule: i-set-ind.induct)
apply (fastforce simp: iT-Min iT-sub)+
done

```

```

lemma i-set0-Plus-neg-closed:  $I \in i\text{-set0} \implies I \oplus - k \in i\text{-set0}$ 
apply (simp add: i-set0-i-set0-ind-eq)
apply (induct rule: i-set0-ind.induct)
  apply (fastforce simp: iT-Plus-neg-empty)
apply (induct-tac I rule: i-set-ind.induct)
apply (fastforce simp: iT-add-neg)+
done

```

```

lemma i-set0-Minus-closed:  $I \in i\text{-set0} \implies k \ominus I \in i\text{-set0}$ 
apply (simp add: i-set0-i-set0-ind-eq)
apply (induct rule: i-set0-ind.induct)
  apply (simp add: iT-Minus-empty i-set0-ind-empty)
apply (induct-tac I rule: i-set-ind.induct)
apply (fastforce simp: iT-sub)+
done

```

```

lemmas i-set-IntOp-closed =
  i-set-Plus-closed
  i-set-Mult-closed

```

```

i-set-Plus-neg-closed
i-set-Minus-closed
lemmas i-set0-IntOp-closed =
  i-set0-Plus-closed
  i-set0-Mult-closed
  i-set0-Plus-neg-closed
  i-set0-Minus-closed

```

### 2.5.5 Interval sets are not closed under division

```

lemma iMOD-div-mod-gr0-not-in-i-set:
  [  $0 < k; k < m; 0 < m \bmod k$  ]  $\implies [r, \bmod m] \odot k \notin i\text{-set}$ 
apply (rule ccontr, simp)
apply (drule i-set-infinite-as-iMOD)
  apply (simp add: iT-Div-finite-iff iMOD-infinite)
apply (elim exE, rename-tac r' m')
apply (frule iMOD-div-mod-gr0-not-ex[of k m r], assumption+)
apply fastforce
done

```

```

lemma iMODb-div-mod-gr0-not-in-i-set:
  [  $0 < k; k < m; 0 < m \bmod k; k \leq c$  ]  $\implies [r, \bmod m, c] \odot k \notin i\text{-set}$ 
apply (rule ccontr, simp)
apply (drule i-set-finite-as-iMODb)
  apply (simp add: iT-Div-finite-iff iMODb-finite)
apply (elim exE, rename-tac r' m' c')
apply (frule iMODb-div-mod-gr0-not-ex[of k m c r], assumption+)
apply fastforce
done

```

```

lemma  $[0, \bmod 3] \odot 2 \notin i\text{-set}$ 
by (rule iMOD-div-mod-gr0-not-in-i-set, simp+)

```

```

lemma i-set-Div-not-closed: Suc 0 < k  $\implies \exists I \in i\text{-set}. I \odot k \notin i\text{-set}$ 
apply (rule-tac x=[0, mod (Suc k)] in bexI)
apply (rule iMOD-div-mod-gr0-not-in-i-set)
apply (simp-all add: mod-Suc i-set-MOD)
done

```

```

lemma i-set0-Div-not-closed: Suc 0 < k  $\implies \exists I \in i\text{-set0}. I \odot k \notin i\text{-set0}$ 
apply (frule i-set-Div-not-closed, erule bexE)
apply (rule-tac x=I in bexI)
  apply (simp add: i-set0-def iT-Div-not-empty i-set-imp-not-empty)
apply (rule subsetD[OF i-set-subset-i-set0], assumption)
done

```

### 2.5.6 Sets of intervals closed under division

```

inductive-set NatMultiples :: nat set  $\Rightarrow$  nat set
  for F :: nat set
where

```

*NatMultiples-Factor*:  $k \in F \implies k \in \text{NatMultiples } F$   
 | *NatMultiples-Product*:  $\llbracket k \in F; m \in \text{NatMultiples } F \rrbracket \implies k * m \in \text{NatMultiples } F$

**lemma** *NatMultiples-ex-divisor*:  $m \in \text{NatMultiples } F \implies \exists k \in F. m \bmod k = 0$   
**apply** (*induct m rule: NatMultiples.induct*)  
**apply** (*rule-tac x=k in beXI, simp+*)  
**done**

**lemma** *NatMultiples-product-factor*:  $\llbracket a \in F; b \in F \rrbracket \implies a * b \in \text{NatMultiples } F$   
**by** (*drule NatMultiples-Factor[of b], rule NatMultiples-Product*)

**lemma** *NatMultiples-product-factor-multiple*:  
 $\llbracket a \in F; b \in \text{NatMultiples } F \rrbracket \implies a * b \in \text{NatMultiples } F$   
**by** (*rule NatMultiples-Product*)

**lemma** *NatMultiples-product-multiple-factor*:  
 $\llbracket a \in \text{NatMultiples } F; b \in F \rrbracket \implies a * b \in \text{NatMultiples } F$   
**by** (*simp add: mult.commute[of a] NatMultiples-Product*)

**lemma** *NatMultiples-product-multiple*:  
 $\llbracket a \in \text{NatMultiples } F; b \in \text{NatMultiples } F \rrbracket \implies a * b \in \text{NatMultiples } F$   
**apply** (*induct a rule: NatMultiples.induct*)  
**apply** (*simp add: NatMultiples-Product*)  
**apply** (*simp add: mult.assoc[of - - b] NatMultiples-Product*)  
**done**

Subset of *i-set* containing only continuous intervals, i. e., without *iMOD* and *iMODb*.

**inductive-set** *i-set-cont* :: (nat set) set  
**where**  
   *i-set-cont-FROM*[*intro*]:  $[n..] \in i\text{-set-cont}$   
   | *i-set-cont-TILL*[*intro*]:  $[\dots n] \in i\text{-set-cont}$   
   | *i-set-cont-IN*[*intro*]:  $[n..,d] \in i\text{-set-cont}$

**definition** *i-set0-cont* :: (nat set) set  
**where** *i-set0-cont*  $\equiv$  *insert* {} *i-set-cont*

**lemma** *i-set-cont-subset-i-set*:  $i\text{-set-cont} \subseteq i\text{-set}$   
**apply** (*unfold subset-eq, rule ballI, rename-tac x*)  
**apply** (*rule-tac a=x in i-set-cont.cases*)  
**apply** *blast+*  
**done**

**lemma** *i-set0-cont-subset-i-set0*:  $i\text{-set0-cont} \subseteq i\text{-set0}$   
**apply** (*unfold i-set0-cont-def i-set0-def*)  
**apply** (*rule insert-mono*)  
**apply** (*rule i-set-cont-subset-i-set*)  
**done**

Minimal definition of *i-set-cont*

**inductive-set** *i-set-cont-min* :: (nat set) set

**where**

*i-set-cont-FROM*[intro]:  $[n..] \in i\text{-set-cont-min}$   
 | *i-set-cont-IN*[intro]:  $[n..,d] \in i\text{-set-cont-min}$

**definition** *i-set0-cont-min* :: (nat set) set

**where** *i-set0-cont-min*  $\equiv$  insert {} *i-set-cont-min*

**lemma** *i-set-cont-min-eq*: *i-set-cont* = *i-set-cont-min*

**apply** (rule set-eqI, rule iffI)

**apply** (rename-tac x, rule-tac a=x **in** *i-set-cont.cases*)

**apply** (fastforce simp: *iIN-0-iTILL-conv*[symmetric])+

**apply** (rename-tac x, rule-tac a=x **in** *i-set-cont-min.cases*)

**apply** blast+

**done**

*inext* and *iprev* with continuous intervals

**lemma** *i-set-cont-inext*:

$\llbracket I \in i\text{-set-cont}; n \in I; \text{finite } I \implies n < \text{Max } I \rrbracket \implies \text{inext } n \ I = \text{Suc } n$

**apply** (simp add: *i-set-cont-min-eq*)

**apply** (rule *i-set-cont-min.cases*, assumption)

**apply** (simp-all add: *iT-finite iT-Max iT-inext-if iT-iff*)

**done**

**lemma** *i-set-cont-iprev*:

$\llbracket I \in i\text{-set-cont}; n \in I; i\text{Min } I < n \rrbracket \implies \text{iprev } n \ I = n - \text{Suc } 0$

**apply** (simp add: *i-set-cont-min-eq*)

**apply** (rule *i-set-cont-min.cases*, assumption)

**apply** (simp-all add: *iT-iprev-if iT-Min iT-iff*)

**done**

**lemma** *i-set-cont-inext--less*:

$\llbracket I \in i\text{-set-cont}; n \in I; n < n0; n0 \in I \rrbracket \implies \text{inext } n \ I = \text{Suc } n$

**apply** (case-tac finite I)

**apply** (rule *i-set-cont-inext*, assumption+)

**apply** (rule order-less-le-trans[OF - Max-ge], assumption+)

**apply** (rule *i-set-cont.cases*, assumption)

**apply** (simp-all add: *iT-finite iT-inext*)

**done**

**lemma** *i-set-cont-iprev--greater*:

$\llbracket I \in i\text{-set-cont}; n \in I; n0 < n; n0 \in I \rrbracket \implies \text{iprev } n \ I = n - \text{Suc } 0$

**apply** (rule *i-set-cont-iprev*, assumption+)

**apply** (rule order-le-less-trans[OF iMin-le, of n0], assumption+)

**done**

Sets of modulo intervals

**inductive-set** *i-set-mult* :: nat  $\Rightarrow$  (nat set) set

**for**  $k :: \text{nat}$   
**where**  
 $i\text{-set-mult-FROM}[intro!]: [n..] \in i\text{-set-mult } k$   
 $| i\text{-set-mult-TILL}[intro!]: [..n] \in i\text{-set-mult } k$   
 $| i\text{-set-mult-IN}[intro!]: [n..,d] \in i\text{-set-mult } k$   
 $| i\text{-set-mult-MOD}[intro!]: [r, \text{mod } m * k] \in i\text{-set-mult } k$   
 $| i\text{-set-mult-MODb}[intro!]: [r, \text{mod } m * k, c] \in i\text{-set-mult } k$

**definition**  $i\text{-set0-mult} :: \text{nat} \Rightarrow (\text{nat set}) \text{ set}$   
**where**  $i\text{-set0-mult } k \equiv \text{insert } \{\} (i\text{-set-mult } k)$

**lemma**  
 $i\text{-set0-mult-empty}[intro!]: \{\} \in i\text{-set0-mult } k$  **and**  
 $i\text{-set0-mult-FROM}[intro!]: [n..] \in i\text{-set0-mult } k$  **and**  
 $i\text{-set0-mult-TILL}[intro!]: [..n] \in i\text{-set0-mult } k$  **and**  
 $i\text{-set0-mult-IN}[intro!]: [n..,d] \in i\text{-set0-mult } k$  **and**  
 $i\text{-set0-mult-MOD}[intro!]: [r, \text{mod } m * k] \in i\text{-set0-mult } k$  **and**  
 $i\text{-set0-mult-MODb}[intro!]: [r, \text{mod } m * k, c] \in i\text{-set0-mult } k$   
**by** ( $\text{simp-all add: } i\text{-set0-mult-def } i\text{-set-mult.intros}$ )

**lemmas**  $i\text{-set0-mult-intros} =$   
 $i\text{-set0-mult-empty}$   
 $i\text{-set0-mult-FROM}$   
 $i\text{-set0-mult-TILL}$   
 $i\text{-set0-mult-IN}$   
 $i\text{-set0-mult-MOD}$   
 $i\text{-set0-mult-MODb}$

**lemma**  $i\text{-set-mult-subset-i-set0-mult}: i\text{-set-mult } k \subseteq i\text{-set0-mult } k$   
**by** ( $\text{unfold } i\text{-set0-mult-def}, \text{ rule subset-insertI}$ )

**lemma**  $i\text{-set-mult-subset-i-set}: i\text{-set-mult } k \subseteq i\text{-set}$   
**apply** ( $\text{clarsimp simp: subset-iff}$ )  
**apply** ( $\text{rule-tac } a=t \text{ in } i\text{-set-mult.cases}[of - k]$ )  
**apply** ( $\text{simp-all add: } i\text{-set-intros}$ )  
**done**

**lemma**  $i\text{-set0-mult-subset-i-set0}: i\text{-set0-mult } k \subseteq i\text{-set0}$   
**apply** ( $\text{simp add: } i\text{-set0-mult-def } i\text{-set0-empty}$ )  
**apply** ( $\text{rule order-trans}[OF - i\text{-set-subset-i-set0}, OF i\text{-set-mult-subset-i-set}]$ )  
**done**

**lemma**  $i\text{-set-mult-0-eq-i-set-cont}: i\text{-set-mult } 0 = i\text{-set-cont}$   
**apply** ( $\text{rule set-eqI}, \text{ rule iffI}$ )  
**apply** ( $\text{rename-tac } x, \text{ rule-tac } a=x \text{ in } i\text{-set-mult.cases}[of - 0]$ )  
**apply** ( $\text{simp-all add: } i\text{-set-cont.intros } i\text{MOD-0 } i\text{MODb-mod-0}$ )  
**apply** ( $\text{rename-tac } x, \text{ rule-tac } a=x \text{ in } i\text{-set-cont.cases}$ )  
**apply** ( $\text{simp-all add: } i\text{-set-mult.intros}$ )  
**done**

**lemma** *i-set0-mult-0-eq-i-set0-cont*:  $i\text{-set0-mult } 0 = i\text{-set0-cont}$   
**by** (*simp add: i-set0-mult-def i-set0-cont-def i-set-mult-0-eq-i-set-cont*)

**lemma** *i-set-mult-1-eq-i-set*:  $i\text{-set-mult } (\text{Suc } 0) = i\text{-set}$   
**apply** (*rule set-eqI, rule iffI*)  
**apply** (*rename-tac x, induct-tac x rule: i-set-mult.induct[of - 1]*)  
**apply** (*simp-all add: i-set-intros*)  
**apply** (*simp add: i-set-i-set-ind-eq*)  
**apply** (*rename-tac x, induct-tac x rule: i-set-ind.induct*)  
**apply** (*simp-all add: i-set-mult.intros*)  
**apply** (*rule-tac t=m and s=m \* Suc 0 in subst, simp, rule i-set-mult.intros*)+  
**done**

**lemma** *i-set0-mult-1-eq-i-set0*:  $i\text{-set0-mult } (\text{Suc } 0) = i\text{-set0}$   
**by** (*simp add: i-set0-mult-def i-set0-def i-set-mult-1-eq-i-set*)

**lemma** *i-set-mult-imp-not-empty*:  $I \in i\text{-set-mult } k \implies I \neq \{\}$   
**by** (*induct I rule: i-set-mult.induct, simp-all add: iT-not-empty*)

**lemma** *iMOD-in-i-set-mult-imp-divisor-mod-0*:  
 $\llbracket m \neq \text{Suc } 0; [r, \text{mod } m] \in i\text{-set-mult } k \rrbracket \implies m \text{ mod } k = 0$   
**apply** (*case-tac m = 0, simp*)  
**apply** (*rule i-set-mult.cases[of [r, mod m] k], assumption*)  
**apply** (*blast dest: iFROM-iMOD-neq*)  
**apply** (*blast dest: iTILL-iMOD-neq*)  
**apply** (*blast dest: iIN-iMOD-neq*)  
**apply** (*simp add: iMOD-eq-conv*)  
**apply** (*blast dest: iMOD-iMODb-neq*)  
**done**

**lemma**  
*divisor-mod-0-imp-iMOD-in-i-set-mult*:  $m \text{ mod } k = 0 \implies [r, \text{mod } m] \in i\text{-set-mult } k$  **and**  
*divisor-mod-0-imp-iMODb-in-i-set-mult*:  $m \text{ mod } k = 0 \implies [r, \text{mod } m, c] \in i\text{-set-mult } k$   
**by** (*auto simp add: ac-simps*)

**lemma** *iMOD-in-i-set-mult--divisor-mod-0-conv*:  
 $m \neq \text{Suc } 0 \implies ([r, \text{mod } m] \in i\text{-set-mult } k) = (m \text{ mod } k = 0)$   
**apply** (*rule iffI*)  
**apply** (*simp add: iMOD-in-i-set-mult-imp-divisor-mod-0*)  
**apply** (*simp add: divisor-mod-0-imp-iMOD-in-i-set-mult*)  
**done**

**lemma** *i-set-mult-neq1-subset-i-set*:  $k \neq \text{Suc } 0 \implies i\text{-set-mult } k \subset i\text{-set}$   
**apply** (*rule psubsetI, rule i-set-mult-subset-i-set*)  
**apply** (*simp add: set-eq-iff*)  
**apply** (*drule neq-iff[THEN iffD1], erule disjE*)

```

apply (simp add: i-set-mult-0-eq-i-set-cont)
apply (thin-tac k = 0)
apply (rule-tac x=[0, mod 2] in exI)
apply (rule ccontr)
apply (simp add: i-set-intros)
apply (drule i-set-cont.cases[where P=False])
  apply (drule sym, simp add: iT-neq)+
apply simp
apply (rule-tac x=[0, mod Suc k] in exI)
apply (rule ccontr)
apply (simp add: i-set-intros)
apply (insert iMOD-in-i-set-mult-imp-divisor-mod-0[of Suc k 0 k])
apply (simp add: mod-Suc)
done

```

```

lemma mod-0-imp-i-set-mult-subset:
  a mod b = 0  $\implies$  i-set-mult a  $\subseteq$  i-set-mult b
  apply (auto simp add: ac-simps elim!: dvdE)
apply (rule-tac a=x and k=k * b in i-set-mult.cases)
apply (simp-all add: i-set-mult.intros mult.assoc[symmetric])
done

```

```

lemma i-set-mult-subset-imp-mod-0:
   $\llbracket a \neq \text{Suc } 0; (i\text{-set-mult } a \subseteq i\text{-set-mult } b) \rrbracket \implies a \text{ mod } b = 0$ 
apply (simp add: subset-iff)
apply (erule-tac x=[0, mod a] in allE)
apply (simp add: divisor-mod-0-imp-iMOD-in-i-set-mult)
apply (simp add: iMOD-in-i-set-mult-imp-divisor-mod-0[of - 0 b])
done

```

```

lemma i-set-mult-subset-conv:
  a  $\neq$  Suc 0  $\implies (i\text{-set-mult } a \subseteq i\text{-set-mult } b) = (a \text{ mod } b = 0)$ 
apply (rule iffI)
  apply (simp add: i-set-mult-subset-imp-mod-0)
apply (simp add: mod-0-imp-i-set-mult-subset)
done

```

```

lemma i-set-mult-mod-0-div:
   $\llbracket I \in i\text{-set-mult } k; k \text{ mod } d = 0 \rrbracket \implies I \odot d \in i\text{-set-mult } (k \text{ div } d)$ 
apply (case-tac d = 0)
  apply (simp add: iT-Div-0[OF i-set-mult-imp-not-empty] i-set-mult.intros)
apply (induct I rule: i-set-mult.induct)
apply (simp-all add: iT-div i-set-mult.intros)
apply (simp-all add: iMOD-div iMODb-div mod-0-imp-mod-mult-left-0 mod-0-imp-div-mult-left-eq
i-set-mult.intros)
done

```

Intervals from  $i\text{-set-mult } k$  remain in  $i\text{-set}$  after division by  $d$  a divisor of  $k$ .

**corollary** *i-set-mult-mod-0-div-i-set*:

$\llbracket I \in i\text{-set-mult } k; k \bmod d = 0 \rrbracket \Longrightarrow I \odot d \in i\text{-set}$   
**by** (rule subsetD[OF *i-set-mult-subset-i-set*[of *k div d*]], rule *i-set-mult-mod-0-div*)  
**corollary** *i-set-mult-div-self-i-set*:  
 $I \in i\text{-set-mult } k \Longrightarrow I \odot k \in i\text{-set}$   
**by** (simp add: *i-set-mult-mod-0-div-i-set*)

**lemma** *i-set-mod-0-mult-in-i-set-mult*:

$\llbracket I \in i\text{-set}; m \bmod k = 0 \rrbracket \Longrightarrow I \otimes m \in i\text{-set-mult } k$   
**apply** (case-tac  $m = 0$ )  
**apply** (simp add: *iT-Mult-0 i-set-imp-not-empty i-set-mult.intros*)  
**apply** (clarsimp simp: *mult.commute*[of *k*] elim!: *dvdE*)  
**apply** (simp add: *i-set-i-set-ind-eq*)  
**apply** (rule-tac  $a=I$  in *i-set-ind.cases*)  
**apply** (simp-all add: *iT-mult mult.assoc*[*symmetric*] *i-set-mult.intros*)  
**done**

**lemma** *i-set-self-mult-in-i-set-mult*:

$I \in i\text{-set} \Longrightarrow I \otimes k \in i\text{-set-mult } k$   
**by** (rule *i-set-mod-0-mult-in-i-set-mult*[OF - *mod-self*])

**lemma** *i-set-mult-mod-gr0-div-not-in-i-set*:

$\llbracket 0 < k; 0 < d; 0 < k \bmod d \rrbracket \Longrightarrow \exists I \in i\text{-set-mult } k. I \odot d \notin i\text{-set}$   
**apply** (case-tac  $d = \text{Suc } 0$ , simp)  
**apply** (frule *iMOD-div-mod-gr0-not-ex*[of  $d \text{ Suc } d * k 0$ ])  
**apply** (rule *Suc-le-lessD*, rule *gr0-imp-self-le-mult1*, assumption)  
**apply** simp  
**apply** (rule-tac  $x=[0, \text{mod } \text{Suc } d * k]$  in *bezI*)  
**apply** (rule *ccontr*, simp)  
**apply** (frule *i-set-infinite-as-iMOD*)  
**apply** (simp add: *iT-Div-finite-iff iMOD-infinite*)  
**apply** fastforce  
**apply** (simp add: *i-set-mult.intros del: mult-Suc*)  
**done**

**lemma** *i-set0-mult-mod-0-div*:

$\llbracket I \in i\text{-set0-mult } k; k \bmod d = 0 \rrbracket \Longrightarrow I \odot d \in i\text{-set0-mult } (k \text{ div } d)$   
**apply** (simp add: *i-set0-mult-def*)  
**apply** (elim *disjE*)  
**apply** (simp add: *iT-Div-empty*)  
**apply** (simp add: *i-set-mult-mod-0-div*)  
**done**

**corollary** *i-set0-mult-mod-0-div-i-set0*:

$\llbracket I \in i\text{-set0-mult } k; k \bmod d = 0 \rrbracket \Longrightarrow I \odot d \in i\text{-set0}$   
**by** (rule subsetD[OF *i-set0-mult-subset-i-set0*[of *k div d*]], rule *i-set0-mult-mod-0-div*)

**corollary** *i-set0-mult-div-self-i-set0*:

$I \in i\text{-set0-mult } k \implies I \otimes k \in i\text{-set0}$   
**by** (*simp add: i-set0-mult-mod-0-div-i-set0*)

**lemma** *i-set0-mod-0-mult-in-i-set0-mult*:  
 $\llbracket I \in i\text{-set0}; m \bmod k = 0 \rrbracket \implies I \otimes m \in i\text{-set0-mult } k$   
**apply** (*simp add: i-set0-def*)  
**apply** (*erule disjE*)  
**apply** (*simp add: iT-Mult-empty i-set0-mult-empty*)  
**apply** (*rule subsetD[OF i-set-mult-subset-i-set0-mult]*)  
**apply** (*simp add: i-set-mod-0-mult-in-i-set-mult*)  
**done**

**lemma** *i-set0-self-mult-in-i-set0-mult*:  
 $I \in i\text{-set0} \implies I \otimes k \in i\text{-set0-mult } k$   
**by** (*simp add: i-set0-mod-0-mult-in-i-set0-mult*)

**lemma** *i-set0-mult-mod-gr0-div-not-in-i-set0*:  
 $\llbracket 0 < k; 0 < d; 0 < k \bmod d \rrbracket \implies \exists I \in i\text{-set0-mult } k. I \otimes d \notin i\text{-set0}$   
**apply** (*frule i-set-mult-mod-gr0-div-not-in-i-set[of k d], assumption+*)  
**apply** (*erule bexE, rename-tac I, rule-tac x=I in bexI*)  
**apply** (*simp add: i-set0-def iT-Div-not-empty i-set-mult-imp-not-empty*)  
**apply** (*simp add: subsetD[OF i-set-mult-subset-i-set0-mult]*)  
**done**

**end**

### 3 Temporal logic operators on natural intervals

**theory** *IL-TemporalOperators*  
**imports** *IL-IntervalOperators*  
**begin**

Bool : some additional properties

**instantiation** *bool* :: {*ord, zero, one, plus, times, order*}  
**begin**

**definition** *Zero-bool-def* [*simp*]:  $0 \equiv \text{False}$

**definition** *One-bool-def* [*simp*]:  $1 \equiv \text{True}$

**definition** *add-bool-def*:  $a + b \equiv a \vee b$

**definition** *mult-bool-def*:  $a * b \equiv a \wedge b$

**instance** ..

**end**

**value** *False* < *True*

**value** *True* < *True*

**value** *True*  $\leq$  *True*

```

lemmas bool-op-rel-defs =
  add-bool-def
  mult-bool-def
  less-bool-def
  le-bool-def

```

```

instance bool :: wellorder
apply (rule wf-wellorderI)
apply (rule-tac t={(x, y). x < y} and s={(False, True)} in subst)
apply fastforce
apply (unfold wf-def, blast)
apply intro-classes
done

```

```

instance bool :: comm-semiring-1
apply intro-classes
apply (unfold bool-op-rel-defs Zero-bool-def One-bool-def)
apply blast+
done

```

### 3.1 Basic definitions

```

lemma UNIV-nat:  $\mathbf{N} = (\text{UNIV}::\text{nat set})$ 
by (simp add: Nats-def)

```

Universal temporal operator: Always/Globally

```

definition iAll ::  $iT \Rightarrow (\text{Time} \Rightarrow \text{bool}) \Rightarrow \text{bool}$  — Always
where iAll I P  $\equiv \forall t \in I. P\ t$ 

```

Existential temporal operator: Eventually/Finally

```

definition iEx ::  $iT \Rightarrow (\text{Time} \Rightarrow \text{bool}) \Rightarrow \text{bool}$  — Eventually
where iEx I P  $\equiv \exists t \in I. P\ t$ 

```

**syntax**

```

-iAll ::  $\text{Time} \Rightarrow iT \Rightarrow (\text{Time} \Rightarrow \text{bool}) \Rightarrow \text{bool}$  ( $\langle \langle \exists \square \text{ - } \cdot \cdot / \text{ - } \rangle \rangle [0, 0, 10] 10$ )
-iEx ::  $\text{Time} \Rightarrow iT \Rightarrow (\text{Time} \Rightarrow \text{bool}) \Rightarrow \text{bool}$  ( $\langle \langle \exists \diamond \text{ - } \cdot \cdot / \text{ - } \rangle \rangle [0, 0, 10] 10$ )

```

**syntax-consts**

```

-iAll  $\equiv$  iAll and
-iEx  $\equiv$  iEx

```

**translations**

```

 $\square\ t\ I. P \equiv \text{CONST } iAll\ I\ (\lambda t. P)$ 
 $\diamond\ t\ I. P \equiv \text{CONST } iEx\ I\ (\lambda t. P)$ 

```

Future temporal operator: Next

```

definition iNext ::  $\text{Time} \Rightarrow iT \Rightarrow (\text{Time} \Rightarrow \text{bool}) \Rightarrow \text{bool}$  — Next
where iNext t0 I P  $\equiv P\ (\text{inext } t0\ I)$ 

```

Past temporal operator: Last/Previous

**definition**  $iLast :: Time \Rightarrow iT \Rightarrow (Time \Rightarrow bool) \Rightarrow bool$  — Last  
**where**  $iLast\ t0\ I\ P \equiv P\ (iprev\ t0\ I)$

**syntax**

$-iNext :: Time \Rightarrow Time \Rightarrow iT \Rightarrow (Time \Rightarrow bool) \Rightarrow bool$  ( $\langle (3\circ - - - / -) \rangle [0, 0, 10] 10$ )

$-iLast :: Time \Rightarrow Time \Rightarrow iT \Rightarrow (Time \Rightarrow bool) \Rightarrow bool$  ( $\langle (3\ominus - - - / -) \rangle [0, 0, 10] 10$ )

**syntax-consts**

$-iNext \equiv iNext$  **and**

$-iLast \equiv iLast$

**translations**

$\circ\ t\ t0\ I.\ P \equiv CONST\ iNext\ t0\ I\ (\lambda t.\ P)$

$\ominus\ t\ t0\ I.\ P \equiv CONST\ iLast\ t0\ I\ (\lambda t.\ P)$

**lemma**  $\circ\ t\ 10\ [0\dots].\ (t + 10 > 10)$

**by** (*simp add: iNext-def iT-inext-if*)

The following versions of Next and Last operator differ in the cases where no next/previous element exists or specified time point is not in interval: the weak versions return *True* and the strong versions return *False*.

**definition**  $iNextWeak :: Time \Rightarrow iT \Rightarrow (Time \Rightarrow bool) \Rightarrow bool$  — Weak Next  
**where**  $iNextWeak\ t0\ I\ P \equiv (\Box\ t\ \{inext\ t0\ I\} \downarrow >\ t0.\ P\ t)$

**definition**  $iNextStrong :: Time \Rightarrow iT \Rightarrow (Time \Rightarrow bool) \Rightarrow bool$  — Strong Next  
**where**  $iNextStrong\ t0\ I\ P \equiv (\Diamond\ t\ \{inext\ t0\ I\} \downarrow >\ t0.\ P\ t)$

**definition**  $iLastWeak :: Time \Rightarrow iT \Rightarrow (Time \Rightarrow bool) \Rightarrow bool$  — Weak Last  
**where**  $iLastWeak\ t0\ I\ P \equiv (\Box\ t\ \{iprev\ t0\ I\} \downarrow <\ t0.\ P\ t)$

**definition**  $iLastStrong :: Time \Rightarrow iT \Rightarrow (Time \Rightarrow bool) \Rightarrow bool$  — Strong Last  
**where**  $iLastStrong\ t0\ I\ P \equiv (\Diamond\ t\ \{iprev\ t0\ I\} \downarrow <\ t0.\ P\ t)$

**syntax**

$-iNextWeak :: Time \Rightarrow Time \Rightarrow iT \Rightarrow (Time \Rightarrow bool) \Rightarrow bool$  ( $\langle (3\circ_W - - - / -) \rangle [0, 0, 10] 10$ )

$-iNextStrong :: Time \Rightarrow Time \Rightarrow iT \Rightarrow (Time \Rightarrow bool) \Rightarrow bool$  ( $\langle (3\circ_S - - - / -) \rangle [0, 0, 10] 10$ )

$-iLastWeak :: Time \Rightarrow Time \Rightarrow iT \Rightarrow (Time \Rightarrow bool) \Rightarrow bool$  ( $\langle (3\ominus_W - - - / -) \rangle [0, 0, 10] 10$ )

$-iLastStrong :: Time \Rightarrow Time \Rightarrow iT \Rightarrow (Time \Rightarrow bool) \Rightarrow bool$  ( $\langle (3\ominus_S - - - / -) \rangle [0, 0, 10] 10$ )

**syntax-consts**

$-iNextWeak \equiv iNextWeak$  **and**

$-iNextStrong \equiv iNextStrong$  **and**

$-iLastWeak \equiv iLastWeak$  **and**

$-iLastStrong \equiv iLastStrong$

**translations**

$$\begin{aligned} \circ_W t t0 I. P &\equiv \text{CONST } iNextWeak t0 I (\lambda t. P) \\ \circ_S t t0 I. P &\equiv \text{CONST } iNextStrong t0 I (\lambda t. P) \\ \ominus_W t t0 I. P &\equiv \text{CONST } iLastWeak t0 I (\lambda t. P) \\ \ominus_S t t0 I. P &\equiv \text{CONST } iLastStrong t0 I (\lambda t. P) \end{aligned}$$

Some examples for Next and Last operator

**lemma**  $\circ t 5 [0 \dots, 10]. ([0 :: int, 10, 20, 30, 40, 50, 60, 70, 80, 90] ! t < 80)$   
**by** (*simp add: iNext-def iIN-inext*)

**lemma**  $\ominus t 5 [0 \dots, 10]. ([0 :: int, 10, 20, 30, 40, 50, 60, 70, 80, 90] ! t < 80)$   
**by** (*simp add: iLast-def iIN-iprev*)

Temporal Until operator

**definition**  $iUntil :: iT \Rightarrow (Time \Rightarrow bool) \Rightarrow (Time \Rightarrow bool) \Rightarrow bool$  — Until  
**where**  $iUntil I P Q \equiv \diamond t I. Q t \wedge (\square t' (I \downarrow < t). P t')$

Temporal Since operator (past operator corresponding to Until)

**definition**  $iSince :: iT \Rightarrow (Time \Rightarrow bool) \Rightarrow (Time \Rightarrow bool) \Rightarrow bool$  — Since  
**where**  $iSince I P Q \equiv \diamond t I. Q t \wedge (\square t' (I \downarrow > t). P t')$

**syntax**

$$\begin{aligned} -iUntil &:: Time \Rightarrow Time \Rightarrow iT \Rightarrow (Time \Rightarrow bool) \Rightarrow (Time \Rightarrow bool) \Rightarrow bool \\ &(\langle \langle \cdot / - (3U - \cdot) / - \rangle \rangle [10, 0, 0, 0, 10] 10) \\ -iSince &:: Time \Rightarrow Time \Rightarrow iT \Rightarrow (Time \Rightarrow bool) \Rightarrow (Time \Rightarrow bool) \Rightarrow bool \\ &(\langle \langle \cdot / - (3S - \cdot) / - \rangle \rangle [10, 0, 0, 0, 10] 10) \end{aligned}$$

**syntax-consts**

$$\begin{aligned} -iUntil &\equiv iUntil \text{ and} \\ -iSince &\equiv iSince \end{aligned}$$

**translations**

$$\begin{aligned} P. t U t' I. Q &\equiv \text{CONST } iUntil I (\lambda t. P) (\lambda t'. Q) \\ P. t S t' I. Q &\equiv \text{CONST } iSince I (\lambda t. P) (\lambda t'. Q) \end{aligned}$$

**definition**  $iWeakUntil :: iT \Rightarrow (Time \Rightarrow bool) \Rightarrow (Time \Rightarrow bool) \Rightarrow bool$  —  
Weak Until/Wating-for/Unless

**where**  $iWeakUntil I P Q \equiv (\square t I. P t) \vee (\diamond t I. Q t \wedge (\square t' (I \downarrow < t). P t'))$

**definition**  $iWeakSince :: iT \Rightarrow (Time \Rightarrow bool) \Rightarrow (Time \Rightarrow bool) \Rightarrow bool$  —  
Weak Since/Back-to

**where**  $iWeakSince I P Q \equiv (\square t I. P t) \vee (\diamond t I. Q t \wedge (\square t' (I \downarrow > t). P t'))$

**syntax**

$$\begin{aligned} -iWeakUntil &:: Time \Rightarrow Time \Rightarrow iT \Rightarrow (Time \Rightarrow bool) \Rightarrow (Time \Rightarrow bool) \Rightarrow bool \\ &(\langle \langle \cdot / - (3W - \cdot) / - \rangle \rangle [10, 0, 0, 0, 10] 10) \\ -iWeakSince &:: Time \Rightarrow Time \Rightarrow iT \Rightarrow (Time \Rightarrow bool) \Rightarrow (Time \Rightarrow bool) \Rightarrow bool \\ &(\langle \langle \cdot / - (3B - \cdot) / - \rangle \rangle [10, 0, 0, 0, 10] 10) \end{aligned}$$

**syntax-consts**

$$-iWeakUntil \equiv iWeakUntil \text{ and}$$

*-iWeakSince*  $\equiv$  *iWeakSince*

**translations**

$P. t \mathcal{W} t' I. Q \equiv \text{CONST } i\text{WeakUntil } I (\lambda t. P) (\lambda t'. Q)$

$P. t \mathcal{B} t' I. Q \equiv \text{CONST } i\text{WeakSince } I (\lambda t. P) (\lambda t'. Q)$

**definition** *iRelease*  $:: iT \Rightarrow (\text{Time} \Rightarrow \text{bool}) \Rightarrow (\text{Time} \Rightarrow \text{bool}) \Rightarrow \text{bool}$  —  
Release

**where** *iRelease*  $I P Q \equiv (\Box t I. Q t) \vee (\Diamond t I. P t \wedge (\Box t' (I \Downarrow \leq t). Q t'))$

**definition** *iTrigger*  $:: iT \Rightarrow (\text{Time} \Rightarrow \text{bool}) \Rightarrow (\text{Time} \Rightarrow \text{bool}) \Rightarrow \text{bool}$  —  
Trigger

**where** *iTrigger*  $I P Q \equiv (\Box t I. Q t) \vee (\Diamond t I. P t \wedge (\Box t' (I \Downarrow \geq t). Q t'))$

**syntax**

*-iRelease*  $:: \text{Time} \Rightarrow \text{Time} \Rightarrow iT \Rightarrow (\text{Time} \Rightarrow \text{bool}) \Rightarrow (\text{Time} \Rightarrow \text{bool}) \Rightarrow \text{bool}$   
 $\langle \langle \cdot / - (3\mathcal{R} - \cdot) / - \rangle [10, 0, 0, 0, 10] 10 \rangle$

*-iTrigger*  $:: \text{Time} \Rightarrow \text{Time} \Rightarrow iT \Rightarrow (\text{Time} \Rightarrow \text{bool}) \Rightarrow (\text{Time} \Rightarrow \text{bool}) \Rightarrow \text{bool}$   
 $\langle \langle \cdot / - (3\mathcal{T} - \cdot) / - \rangle [10, 0, 0, 0, 10] 10 \rangle$

**syntax-consts**

*-iRelease*  $\equiv$  *iRelease and*

*-iTrigger*  $\equiv$  *iTrigger*

**translations**

$P. t \mathcal{R} t' I. Q \equiv \text{CONST } i\text{Release } I (\lambda t. P) (\lambda t'. Q)$

$P. t \mathcal{T} t' I. Q \equiv \text{CONST } i\text{Trigger } I (\lambda t. P) (\lambda t'. Q)$

**lemmas** *iTL-Next-defs* =

*iNext-def* *iLast-def*

*iNextWeak-def* *iLastWeak-def*

*iNextStrong-def* *iLastStrong-def*

**lemmas** *iTL-defs* =

*iAll-def* *iEx-def*

*iUntil-def* *iSince-def*

*iWeakUntil-def* *iWeakSince-def*

*iRelease-def* *iTrigger-def*

*iTL-Next-defs*

**typed-print-translation**  $\langle$

$[(\text{const-syntax } \langle iAll \rangle, \text{Syntax-Trans.preserve-binder-abs2-tr}' \text{ syntax-const } \langle -iAll \rangle),$   
 $(\text{const-syntax } \langle iEx \rangle, \text{Syntax-Trans.preserve-binder-abs2-tr}' \text{ syntax-const } \langle -iEx \rangle)]$   
 $\rangle$

**print-translation**  $\langle$

*let*

*fun* *btr'* *syn* *ctxt* [*i*, *Abs* *abs*, *Abs* *abs'*] =

*let*

*val* (*t*, *P*) = *Syntax-Trans.atomic-abs-tr'* *ctxt* *abs*;

```

    val (t',Q) = Syntax-Trans.atomic-abs-tr' ctxt abs'
  in Syntax.const syn $ P $ t $ t' $ i $ Q end
in
  [(@{const-syntax iUntil}, btr' -iUntil),
   (@{const-syntax iSince}, btr' -iSince)]
end
>

```

### print-translation <

```

let
  fun btr' syn ctxt [i,Abs abs,Abs abs'] =
    let
      val (t,P) = Syntax-Trans.atomic-abs-tr' ctxt abs;
      val (t',Q) = Syntax-Trans.atomic-abs-tr' ctxt abs'
    in Syntax.const syn $ P $ t $ t' $ i $ Q end
in
  [(@{const-syntax iWeakUntil}, btr' -iWeakUntil),
   (@{const-syntax iWeakSince}, btr' -iWeakSince)]
end
>

```

### print-translation <

```

let
  fun btr' syn ctxt [i,Abs abs,Abs abs'] =
    let
      val (t,P) = Syntax-Trans.atomic-abs-tr' ctxt abs;
      val (t',Q) = Syntax-Trans.atomic-abs-tr' ctxt abs'
    in Syntax.const syn $ P $ t $ t' $ i $ Q end
in
  [(@{const-syntax iRelease}, btr' -iRelease),
   (@{const-syntax iTrigger}, btr' -iTrigger)]
end
>

```

## 3.2 Basic lemmata for temporal operators

### 3.2.1 Intro/elim rules

#### lemma

*iexI[intro]*:  $\llbracket P t; t \in I \rrbracket \Longrightarrow \Diamond t I. P t$  **and**  
*rev-iexI[intro?]*:  $\llbracket t \in I; P t \rrbracket \Longrightarrow \Diamond t I. P t$  **and**  
*iexE[elim!]*:  $\llbracket \Diamond t I. P t; \bigwedge t. \llbracket t \in I; P t \rrbracket \Longrightarrow Q \rrbracket \Longrightarrow Q$   
**by** (*unfold iEx-def*, *blast+*)

#### lemma

*iallI[intro!]*:  $(\bigwedge t. t \in I \Longrightarrow P t) \Longrightarrow \Box t I. P t$  **and**  
*ispec[dest?]*:  $\llbracket \Box t I. P t; t \in I \rrbracket \Longrightarrow P t$  **and**  
*iallE[elim]*:  $\llbracket \Box t I. P t; P t \Longrightarrow Q; t \notin I \Longrightarrow Q \rrbracket \Longrightarrow Q$   
**by** (*unfold iAll-def*, *blast+*)

**lemma**

*iuntilI[intro]:*

$\llbracket Q\ t; (\bigwedge t'. t' \in I \downarrow < t \implies P\ t'); t \in I \rrbracket \implies P\ t'. t' \mathcal{U}\ t\ I. Q\ t$  **and**

*rev-iuntilI[intro?]:*

$\llbracket t \in I; Q\ t; (\bigwedge t'. t' \in I \downarrow < t \implies P\ t') \rrbracket \implies P\ t'. t' \mathcal{U}\ t\ I. Q\ t$

**by** (*unfold iUntil-def, blast+*)

**lemma**

*iuntilE[elim]:*

$\llbracket P'\ t'. t' \mathcal{U}\ t\ I. P\ t; \bigwedge t. \llbracket t \in I; P\ t \rrbracket \implies Q \rrbracket \implies Q$

**by** (*unfold iUntil-def, blast*)

**lemma**

*isinceI[intro]:*

$\llbracket Q\ t; (\bigwedge t'. t' \in I \downarrow > t \implies P\ t'); t \in I \rrbracket \implies P\ t'. t' \mathcal{S}\ t\ I. Q\ t$  **and**

*rev-isinceI[intro?]:*

$\llbracket t \in I; Q\ t; (\bigwedge t'. t' \in I \downarrow > t \implies P\ t') \rrbracket \implies P\ t'. t' \mathcal{S}\ t\ I. Q\ t$

**by** (*unfold iSince-def, blast+*)

**lemma**

*isinceE[elim]:*

$\llbracket P'\ t'. t' \mathcal{S}\ t\ I. P\ t; \bigwedge t. \llbracket t \in I; P\ t \rrbracket \implies Q \rrbracket \implies Q$

**by** (*unfold iSince-def, blast*)

### 3.2.2 Rewrite rules for trivial simplification

**lemma** *iAll-triv[simp]:*  $(\Box\ t\ I. P) = ((\exists\ t. t \in I) \longrightarrow P)$

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

**lemma** *iEx-triv[simp]:*  $(\Diamond\ t\ I. P) = ((\exists\ t. t \in I) \wedge P)$

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

**lemma** *iEx-conjL1:*

$(\Diamond\ t1\ I1. (P1\ t1 \wedge (\Diamond\ t2\ I2. P2\ t1\ t2))) =$   
 $(\Diamond\ t1\ I1. \Diamond\ t2\ I2. P1\ t1 \wedge P2\ t1\ t2)$

**by** *blast*

**lemma** *iEx-conjR1:*

$(\Diamond\ t1\ I1. ((\Diamond\ t2\ I2. P2\ t1\ t2) \wedge P1\ t1)) =$   
 $(\Diamond\ t1\ I1. \Diamond\ t2\ I2. P2\ t1\ t2 \wedge P1\ t1)$

**by** *blast*

**lemma** *iEx-conjL2:*

$(\Diamond\ t1\ I1. (P1\ t1 \wedge (\Diamond\ t2\ (I2\ t1). P2\ t1\ t2))) =$   
 $(\Diamond\ t1\ I1. \Diamond\ t2\ (I2\ t1). P1\ t1 \wedge P2\ t1\ t2)$

**by** *blast*

**lemma** *iEx-conjR2:*

$(\Diamond\ t1\ I1. ((\Diamond\ t2\ (I2\ t1). P2\ t1\ t2) \wedge P1\ t1)) =$   
 $(\Diamond\ t1\ I1. \Diamond\ t2\ (I2\ t1). P2\ t1\ t2 \wedge P1\ t1)$

by *blast*

**lemma** *iex-commute*:

$$\begin{aligned} (\diamond t1 I1. \diamond t2 I2. P t1 t2) = \\ (\diamond t2 I2. \diamond t1 I1. P t1 t2) \end{aligned}$$

by *blast*

**lemma** *iall-conjL1*:

$$\begin{aligned} I2 \neq \{\} \implies \\ (\Box t1 I1. (P1 t1 \wedge (\Box t2 I2. P2 t1 t2))) = \\ (\Box t1 I1. \Box t2 I2. P1 t1 \wedge P2 t1 t2) \end{aligned}$$

by *blast*

**lemma** *iall-conjR1*:

$$\begin{aligned} I2 \neq \{\} \implies \\ (\Box t1 I1. ((\Box t2 I2. P2 t1 t2) \wedge P1 t1)) = \\ (\Box t1 I1. \Box t2 I2. P2 t1 t2 \wedge P1 t1) \end{aligned}$$

by *blast*

**lemma** *iall-conjL2*:

$$\begin{aligned} \Box t1 I1. I2 t1 \neq \{\} \implies \\ (\Box t1 I1. (P1 t1 \wedge (\Box t2 (I2 t1). P2 t1 t2))) = \\ (\Box t1 I1. \Box t2 (I2 t1). P1 t1 \wedge P2 t1 t2) \end{aligned}$$

by *blast*

**lemma** *iall-conjR2*:

$$\begin{aligned} \Box t1 I1. I2 t1 \neq \{\} \implies \\ (\Box t1 I1. ((\Box t2 (I2 t1). P2 t1 t2) \wedge P1 t1)) = \\ (\Box t1 I1. \Box t2 (I2 t1). P2 t1 t2 \wedge P1 t1) \end{aligned}$$

by *blast*

**lemma** *iall-commute*:

$$\begin{aligned} (\Box t1 I1. \Box t2 I2. P t1 t2) = \\ (\Box t2 I2. \Box t1 I1. P t1 t2) \end{aligned}$$

by *blast*

**lemma** *iall-conj-distrib*:

$$(\Box t I. P t \wedge Q t) = ((\Box t I. P t) \wedge (\Box t I. Q t))$$

by *blast*

**lemma** *iex-disj-distrib*:

$$(\diamond t I. P t \vee Q t) = ((\diamond t I. P t) \vee (\diamond t I. Q t))$$

by *blast*

**lemma** *iall-conj-distrib2*:

$$\begin{aligned} (\Box t1 I1. \Box t2 (I2 t1). P t1 t2 \wedge Q t1 t2) = \\ ((\Box t1 I1. \Box t2 (I2 t1). P t1 t2) \wedge (\Box t1 I1. \Box t2 (I2 t1). Q t1 t2)) \end{aligned}$$

by *blast*

**lemma** *iex-disj-distrib2*:

$$(\diamond t1 I1. \diamond t2 (I2 t1). P t1 t2 \vee Q t1 t2) = ((\diamond t1 I1. \diamond t2 (I2 t1). P t1 t2) \vee (\diamond t1 I1. \diamond t2 (I2 t1). Q t1 t2))$$

**by** *blast*

**lemma** *iUntil-disj-distrib*:

$$(P t1. t1 \mathcal{U} t2 I. (Q1 t2 \vee Q2 t2)) = ((P t1. t1 \mathcal{U} t2 I. Q1 t2) \vee (P t1. t1 \mathcal{U} t2 I. Q2 t2))$$

**unfolding** *iUntil-def* **by** *blast*

**lemma** *iSince-disj-distrib*:

$$(P t1. t1 \mathcal{S} t2 I. (Q1 t2 \vee Q2 t2)) = ((P t1. t1 \mathcal{S} t2 I. Q1 t2) \vee (P t1. t1 \mathcal{S} t2 I. Q2 t2))$$

**unfolding** *iSince-def* **by** *blast*

**lemma**

$$iNext\text{-iff}: (\circ t t0 I. P t) = (\square t [\dots 0] \oplus (inext t0 I). P t) \text{ and}$$

$$iLast\text{-iff}: (\ominus t t0 I. P t) = (\square t [\dots 0] \oplus (iprev t0 I). P t)$$

**by** (*fastforce simp: iTL-Next-defs iT-add iIN-0*)**+**

**lemma**

$$iNext\text{-iEx-iff}: (\circ t t0 I. P t) = (\diamond t [\dots 0] \oplus (inext t0 I). P t) \text{ and}$$

$$iLast\text{-iEx-iff}: (\ominus t t0 I. P t) = (\diamond t [\dots 0] \oplus (iprev t0 I). P t)$$

**by** (*fastforce simp: iTL-Next-defs iT-add iIN-0*)**+**

**lemma** *inext-singleton-cut-greater-not-empty-iff*:

$$(\{inext t0 I\} \downarrow > t0 \neq \{\}) = (I \downarrow > t0 \neq \{\} \wedge t0 \in I)$$

**apply** (*simp add: cut-greater-singleton*)

**apply** (*case-tac t0 \in I*)

**prefer** 2

**apply** (*simp add: not-in-inext-fix*)

**apply** *simp*

**apply** (*case-tac I \downarrow > t0 = \{\}*)

**apply** (*simp add: cut-greater-empty-iff inext-all-le-fix*)

**apply** (*simp add: cut-greater-not-empty-iff inext-mono2*)

**done**

**lemma** *iprev-singleton-cut-less-not-empty-iff*:

$$(\{iprev t0 I\} \downarrow < t0 \neq \{\}) = (I \downarrow < t0 \neq \{\} \wedge t0 \in I)$$

**apply** (*simp add: cut-less-singleton*)

**apply** (*case-tac t0 \in I*)

**prefer** 2

**apply** (*simp add: not-in-iprev-fix*)

**apply** *simp*

**apply** (*case-tac I \downarrow < t0 = \{\}*)

**apply** (*simp add: cut-less-empty-iff iprev-all-ge-fix*)

**apply** (*simp add: cut-less-not-empty-iff iprev-mono2*)

**done**

**lemma** *inext-singleton-cut-greater-empty-iff*:  
 $(\{inext\ t0\ I\} \downarrow > t0 = \{\}) = (I \downarrow > t0 = \{\} \vee t0 \notin I)$   
**apply** (*subst Not-eq-iff[symmetric]*)  
**apply** (*simp add: inext-singleton-cut-greater-not-empty-iff*)  
**done**

**lemma** *iprev-singleton-cut-less-empty-iff*:  
 $(\{iprev\ t0\ I\} \downarrow < t0 = \{\}) = (I \downarrow < t0 = \{\} \vee t0 \notin I)$   
**apply** (*subst Not-eq-iff[symmetric]*)  
**apply** (*simp add: iprev-singleton-cut-less-not-empty-iff*)  
**done**

**lemma** *iNextWeak-iff* :  $(\circ_W\ t\ t0\ I.\ P\ t) = ((\circ\ t\ t0\ I.\ P\ t) \vee (I \downarrow > t0 = \{\}) \vee t0 \notin I)$   
**apply** (*unfold iTL-defs*)  
**apply** (*simp add: inext-singleton-cut-greater-empty-iff[symmetric] cut-greater-singleton*)  
**done**

**lemma** *iNextStrong-iff* :  $(\circ_S\ t\ t0\ I.\ P\ t) = ((\circ\ t\ t0\ I.\ P\ t) \wedge (I \downarrow > t0 \neq \{\}) \wedge t0 \in I)$   
**apply** (*unfold iTL-defs*)  
**apply** (*simp add: inext-singleton-cut-greater-not-empty-iff[symmetric] cut-greater-singleton*)  
**done**

**lemma** *iLastWeak-iff* :  $(\ominus_W\ t\ t0\ I.\ P\ t) = ((\ominus\ t\ t0\ I.\ P\ t) \vee (I \downarrow < t0 = \{\}) \vee t0 \notin I)$   
**apply** (*unfold iTL-defs*)  
**apply** (*simp add: iprev-singleton-cut-less-empty-iff[symmetric] cut-less-singleton*)  
**done**

**lemma** *iLastStrong-iff* :  $(\ominus_S\ t\ t0\ I.\ P\ t) = ((\ominus\ t\ t0\ I.\ P\ t) \wedge (I \downarrow < t0 \neq \{\}) \wedge t0 \in I)$   
**apply** (*unfold iTL-defs*)  
**apply** (*simp add: iprev-singleton-cut-less-not-empty-iff[symmetric] cut-less-singleton*)  
**done**

**lemmas** *iTL-Next-iff* =  
*iNext-iff iLast-iff*  
*iNextWeak-iff iNextStrong-iff*  
*iLastWeak-iff iLastStrong-iff*

**lemma**  
*iNext-iff-singleton* :  $(\circ\ t\ t0\ I.\ P\ t) = (\square\ t\ \{inext\ t0\ I\}.\ P\ t)$  **and**  
*iLast-iff-singleton* :  $(\ominus\ t\ t0\ I.\ P\ t) = (\square\ t\ \{iprev\ t0\ I\}.\ P\ t)$   
**by** (*fastforce simp: iTL-Next-defs iT-add iIN-0*)  
**lemmas** *iNextLast-iff-singleton* = *iNext-iff-singleton iLast-iff-singleton*

**lemma**

*iNext-iEx-iff-singleton* :  $(\bigcirc t t0 I. P t) = (\diamond t \{inext t0 I\}. P t)$  **and**

*iLast-iEx-iff-singleton* :  $(\ominus t t0 I. P t) = (\diamond t \{iprev t0 I\}. P t)$

**by** (*fastforce simp: iTL-Next-defs iT-add iIN-0*)**+**

**lemma**

*iNextWeak-iAll-conv*:  $(\bigcirc_W t t0 I. P t) = (\square t (\{inext t0 I\} \downarrow > t0). P t)$  **and**

*iNextStrong-iEx-conv*:  $(\bigcirc_S t t0 I. P t) = (\diamond t (\{inext t0 I\} \downarrow > t0). P t)$  **and**

*iLastWeak-iAll-conv*:  $(\ominus_W t t0 I. P t) = (\square t (\{iprev t0 I\} \downarrow < t0). P t)$  **and**

*iLastStrong-iEx-conv*:  $(\ominus_S t t0 I. P t) = (\diamond t (\{iprev t0 I\} \downarrow < t0). P t)$

**by** (*simp-all add: iTL-Next-defs*)

**lemma**

*iAll-True[simp]*:  $\square t I. True$  **and**

*iAll-False[simp]*:  $(\square t I. False) = (I = \{\})$  **and**

*iEx-True[simp]*:  $(\diamond t I. True) = (I \neq \{\})$  **and**

*iEx-False[simp]*:  $\neg (\diamond t I. False)$

**by** *blast+*

**lemma** *empty-iff-iAll-False*:  $(I = \{\}) = (\square t I. False)$  **by** *blast*

**lemma** *not-empty-iff-iEx-True*:  $(I \neq \{\}) = (\diamond t I. True)$  **by** *blast*

**lemma**

*iNext-True*:  $\bigcirc t t0 I. True$  **and**

*iNextWeak-True*:  $(\bigcirc_W t t0 I. True)$  **and**

*iNext-False*:  $\neg (\bigcirc t t0 I. False)$  **and**

*iNextStrong-False*:  $\neg (\bigcirc_S t t0 I. False)$

**by** (*simp-all add: iTL-defs*)

**lemma**

*iNextStrong-True*:  $(\bigcirc_S t t0 I. True) = (I \downarrow > t0 \neq \{\} \wedge t0 \in I)$  **and**

*iNextWeak-False*:  $(\neg (\bigcirc_W t t0 I. False)) = (I \downarrow > t0 \neq \{\} \wedge t0 \in I)$

**by** (*simp-all add: iTL-defs ex-in-conv inext-singleton-cut-greater-not-empty-iff*)

**lemma**

*iLast-True*:  $\ominus t t0 I. True$  **and**

*iLastWeak-True*:  $(\ominus_W t t0 I. True)$  **and**

*iLast-False*:  $\neg (\ominus t t0 I. False)$  **and**

*iLastStrong-False*:  $\neg (\ominus_S t t0 I. False)$

**by** (*simp-all add: iTL-defs*)

**lemma**

*iLastStrong-True*:  $(\ominus_S t t0 I. True) = (I \downarrow < t0 \neq \{\} \wedge t0 \in I)$  **and**

*iLastWeak-False*:  $(\neg (\ominus_W t t0 I. False)) = (I \downarrow < t0 \neq \{\} \wedge t0 \in I)$

**by** (*simp-all add: iTL-defs ex-in-conv iprev-singleton-cut-less-not-empty-iff*)

**lemma** *iUntil-True-left[simp]*:  $(\text{True}. t' \mathcal{U} t I. Q t) = (\diamond t I. Q t)$   
**by** *blast*

**lemma** *iUntil-True[simp]*:  $(P t'. t' \mathcal{U} t I. \text{True}) = (I \neq \{\})$   
**apply** (*unfold iTL-defs*)  
**apply** (*rule iffI*)  
**apply** *blast*  
**apply** (*rule-tac x=iMin I in bexI*)  
**apply** (*simp add: cut-less-Min-empty iMinI-ex2*)  
**done**

**lemma** *iUntil-False-left[simp]*:  $(\text{False}. t' \mathcal{U} t I. Q t) = (I \neq \{\} \wedge Q (iMin I))$   
**apply** (*case-tac I = \{\}, blast*)  
**apply** (*simp add: iTL-defs*)  
**apply** (*rule iffI*)  
**apply** *clarsimp*  
**apply** (*drule iMin-equality*)  
**apply** (*simp add: cut-less-empty-iff*)  
**apply** *simp*  
**apply** (*rule-tac x=iMin I in bexI*)  
**apply** (*simp add: cut-less-Min-empty*)  
**apply** (*simp add: iMinI-ex2*)  
**done**

**lemma** *iUntil-False[simp]*:  $\neg (P t'. t' \mathcal{U} t I. \text{False})$   
**by** *blast*

**lemma** *iSince-True-left[simp]*:  $(\text{True}. t' \mathcal{S} t I. Q t) = (\diamond t I. Q t)$   
**by** *blast*

**lemma** *iSince-True-if*:  
 $(P t'. t' \mathcal{S} t I. \text{True}) =$   
 $(\text{if finite } I \text{ then } I \neq \{\} \text{ else } \diamond t1 I. \square t2 (I \downarrow > t1). P t2)$   
**apply** (*clarsimp simp: iTL-defs*)  
**apply** (*rule iffI*)  
**apply** *clarsimp*  
**apply** (*rule-tac x=Max I in bexI*)  
**apply** (*simp add: cut-greater-Max-empty*)  
**apply** *simp*  
**done**

**corollary** *iSince-True-finite[simp]*:  $\text{finite } I \implies (P t'. t' \mathcal{S} t I. \text{True}) = (I \neq \{\})$   
**by** (*simp add: iSince-True-if*)

**lemma** *iSince-False-left[simp]*:  $(\text{False}. t' \mathcal{S} t I. Q t) = (\text{finite } I \wedge I \neq \{\} \wedge Q (Max I))$   
**apply** (*simp add: iTL-defs*)  
**apply** (*case-tac I = \{\}, simp*)  
**apply** (*case-tac infinite I*)

```

apply (simp add: nat-cut-greater-infinite-not-empty)
apply (rule iffI)
apply clarsimp
apply (drule Max-equality)
  apply simp
  apply (simp add: cut-greater-empty-iff)
apply simp
apply (rule-tac x=Max I in bexI)
apply (simp add: cut-greater-Max-empty)
apply simp
done

```

```

lemma iSince-False[simp]:  $\neg (P \ t'. \ t' \ \mathcal{S} \ t \ I. \ \text{False})$ 
by blast

```

```

lemma iWeakUntil-True-left[simp]:  $\text{True}. \ t' \ \mathcal{W} \ t \ I. \ Q \ t$ 
by (simp add: iWeakUntil-def)

```

```

lemma iWeakUntil-True[simp]:  $P \ t'. \ t' \ \mathcal{W} \ t \ I. \ \text{True}$ 
apply (simp add: iTL-defs)
apply (case-tac I = {}, simp)
apply (rule disjI2)
apply (rule-tac x=iMin I in bexI)
  apply (simp add: cut-less-Min-empty)
apply (simp add: iMinI-ex2)
done

```

```

lemma iWeakUntil-False-left[simp]:  $(\text{False}. \ t' \ \mathcal{W} \ t \ I. \ Q \ t) = (I = \{\} \vee Q \ (iMin \ I))$ 
apply (simp add: iTL-defs)
apply (case-tac I = {}, simp)
apply (rule iffI)
  apply (clarsimp simp: cut-less-empty-iff)
  apply (frule iMin-equality)
  apply simp+
apply (rule-tac x=iMin I in bexI)
  apply (simp add: cut-less-Min-empty)
apply (simp add: iMinI-ex2)
done

```

```

lemma iWeakUntil-False[simp]:  $(P \ t'. \ t' \ \mathcal{W} \ t \ I. \ \text{False}) = (\Box \ t \ I. \ P \ t)$ 
by (simp add: iWeakUntil-def)

```

```

lemma iWeakSince-True-left[simp]:  $\text{True}. \ t' \ \mathcal{B} \ t \ I. \ Q \ t$ 
by (simp add: iTL-defs)

```

```

lemma iWeakSince-True-disj:
   $(P \ t'. \ t' \ \mathcal{B} \ t \ I. \ \text{True}) =$ 
   $(I = \{\} \vee (\Diamond \ t1 \ I. \ \Box \ t2 \ (I \ \Downarrow \ t1). \ P \ t2))$ 

```

**unfolding** *iTL-defs* **by** *blast*

**lemma** *iWeakSince-True-finite[simp]*:  $\text{finite } I \implies (P \ t'. \ t' \ \mathcal{B} \ t \ I. \ \text{True})$   
**apply** (*simp add: iTL-defs*)  
**apply** (*case-tac I = {}, simp*)  
**apply** (*rule disjI2*)  
**apply** (*rule-tac x=Max I in bexI*)  
**apply** (*simp add: cut-greater-Max-empty*)  
**apply** *simp*  
**done**

**lemma** *iWeakSince-False-left[simp]*:  $(\text{False}. \ t' \ \mathcal{B} \ t \ I. \ Q \ t) = (I = \{\} \vee (\text{finite } I \wedge Q \ (\text{Max } I)))$   
**apply** (*simp add: iTL-defs*)  
**apply** (*case-tac I = {}, simp*)  
**apply** (*case-tac in finite I*)  
**apply** (*simp add: nat-cut-greater-infinite-not-empty*)  
**apply** (*rule iffI*)  
**apply** *clarsimp*  
**apply** (*drule Max-equality*)  
**apply** *simp*  
**apply** (*simp add: cut-greater-empty-iff*)  
**apply** *simp*  
**apply** *simp*  
**apply** (*rule-tac x=Max I in bexI*)  
**apply** (*simp add: cut-greater-Max-empty*)  
**apply** *simp*  
**done**

**lemma** *iWeakSince-False[simp]*:  $(P \ t'. \ t' \ \mathcal{B} \ t \ I. \ \text{False}) = (\Box \ t \ I. \ P \ t)$   
**by** (*simp add: iWeakSince-def*)

**lemma** *iRelease-True-left[simp]*:  $(\text{True}. \ t' \ \mathcal{R} \ t \ I. \ Q \ t) = (I = \{\} \vee Q \ (\text{iMin } I))$   
**apply** (*simp add: iTL-defs*)  
**apply** (*case-tac I = {}, simp*)  
**apply** (*rule iffI*)  
**apply** (*erule disjE*)  
**apply** (*blast intro: iMinI2-ex2*)  
**apply** *clarsimp*  
**apply** (*drule-tac x=iMin I in bspec*)  
**apply** (*blast intro: iMinI-ex2*)  
**apply** *simp*  
**apply** (*rule disjI2*)  
**apply** (*rule-tac x=iMin I in bexI*)  
**apply** *fastforce*  
**apply** (*simp add: iMinI-ex2*)  
**done**

**lemma** *iRelease-True[simp]*:  $P \ t'. \ t' \ \mathcal{R} \ t \ I. \ \text{True}$

by (*simp add: iTL-defs*)

**lemma** *iRelease-False-left[simp]*:  $(False. t' \mathcal{R} t I. Q t) = (\Box t I. Q t)$   
 by (*simp add: iTL-defs*)

**lemma** *iRelease-False[simp]*:  $(P t'. t' \mathcal{R} t I. False) = (I = \{\})$   
**unfolding** *iTL-defs* by *blast*

**lemma** *iTrigger-True-left[simp]*:  $(True. t' \mathcal{T} t I. Q t) = (I = \{\} \vee (\Diamond t1 I. \Box t2 (I \Downarrow t1). Q t2))$   
**unfolding** *iTL-defs* by *blast*

**lemma** *iTrigger-True[simp]*:  $P t'. t' \mathcal{T} t I. True$   
 by (*simp add: iTL-defs*)

**lemma** *iTrigger-False-left[simp]*:  $(False. t' \mathcal{T} t I. Q t) = (\Box t I. Q t)$   
 by (*simp add: iTL-defs*)

**lemma** *iTrigger-False[simp]*:  $(P t'. t' \mathcal{T} t I. False) = (I = \{\})$   
**unfolding** *iTL-defs* by *blast*

**lemma**

*iUntil-TrueTrue[simp]*:  $(True. t' \mathcal{U} t I. True) = (I \neq \{\})$  **and**  
*iSince-TrueTrue[simp]*:  $(True. t' \mathcal{S} t I. True) = (I \neq \{\})$  **and**  
*iWeakUntil-TrueTrue[simp]*:  $True. t' \mathcal{W} t I. True$  **and**  
*iWeakSince-TrueTrue[simp]*:  $True. t' \mathcal{B} t I. True$  **and**  
*iRelease-TrueTrue[simp]*:  $True. t' \mathcal{R} t I. True$  **and**  
*iTrigger-TrueTrue[simp]*:  $True. t' \mathcal{T} t I. True$   
 by (*simp-all add: iTL-defs ex-in-conv*)

### 3.2.3 Empty sets and singletons

**lemma** *iAll-empty[simp]*:  $\Box t \{\}. P t$  by *blast*

**lemma** *iEx-empty[simp]*:  $\neg (\Diamond t \{\}. P t)$  by *blast*

**lemma** *iUntil-empty[simp]*:  $\neg (P t0. t0 \mathcal{U} t1 \{\}. Q t1)$  by *blast*

**lemma** *iSince-empty[simp]*:  $\neg (P t0. t0 \mathcal{S} t1 \{\}. Q t1)$  by *blast*

**lemma** *iWeakUntil-empty[simp]*:  $P t0. t0 \mathcal{W} t1 \{\}. Q t1$  by (*simp add: iWeakUntil-def*)

**lemma** *iWeakSince-empty[simp]*:  $P t0. t0 \mathcal{B} t1 \{\}. Q t1$  by (*simp add: iWeakSince-def*)

**lemma** *iRelease-empty[simp]*:  $P t0. t0 \mathcal{R} t1 \{\}. Q t1$  by (*simp add: iRelease-def*)

**lemma** *iTrigger-empty[simp]*:  $P t0. t0 \mathcal{T} t1 \{\}. Q t1$  by (*simp add: iTrigger-def*)

**lemmas** *iTL-empty* =

*iAll-empty iEx-empty*

*iUntil-empty iSince-empty*

*iWeakUntil-empty iWeakSince-empty*

*iRelease-empty iTrigger-empty*

**lemma** *iAll-singleton[simp]*:  $(\Box t' \{t\}. P t') = P t$  **by** *blast*

**lemma** *iEx-singleton[simp]*:  $(\Diamond t' \{t\}. P t') = P t$  **by** *blast*

**lemma** *iUntil-singleton[simp]*:  $(P t0. t0 \mathcal{U} t1 \{t\}. Q t1) = Q t$   
**by** (*simp add: iUntil-def cut-less-singleton*)

**lemma** *iSince-singleton[simp]*:  $(P t0. t0 \mathcal{S} t1 \{t\}. Q t1) = Q t$   
**by** (*simp add: iSince-def cut-greater-singleton*)

**lemma** *iWeakUntil-singleton[simp]*:  $(P t0. t0 \mathcal{W} t1 \{t\}. Q t1) = (P t \vee Q t)$   
**by** (*simp add: iWeakUntil-def cut-less-singleton*)

**lemma** *iWeakSince-singleton[simp]*:  $(P t0. t0 \mathcal{B} t1 \{t\}. Q t1) = (P t \vee Q t)$   
**by** (*simp add: iWeakSince-def cut-greater-singleton*)

**lemma** *iRelease-singleton[simp]*:  $(P t0. t0 \mathcal{R} t1 \{t\}. Q t1) = Q t$   
**unfolding** *iRelease-def* **by** *blast*

**lemma** *iTrigger-singleton[simp]*:  $(P t0. t0 \mathcal{T} t1 \{t\}. Q t1) = Q t$   
**unfolding** *iTrigger-def* **by** *blast*

**lemmas** *iTL-singleton* =  
*iAll-singleton iEx-singleton*  
*iUntil-singleton iSince-singleton*  
*iWeakUntil-singleton iWeakSince-singleton*  
*iRelease-singleton iTrigger-singleton*

### 3.2.4 Conversions between temporal operators

**lemma** *iAll-iEx-conv*:  $(\Box t I. P t) = (\neg (\Diamond t I. \neg P t))$  **by** *blast*

**lemma** *iEx-iAll-conv*:  $(\Diamond t I. P t) = (\neg (\Box t I. \neg P t))$  **by** *blast*

**lemma** *not-iAll[simp]*:  $(\neg (\Box t I. P t)) = (\Diamond t I. \neg P t)$  **by** *blast*

**lemma** *not-iEx[simp]*:  $(\neg (\Diamond t I. P t)) = (\Box t I. \neg P t)$  **by** *blast*

**lemma** *iUntil-iEx-conv*:  $(\text{True}. t' \mathcal{U} t I. P t) = (\Diamond t I. P t)$  **by** *blast*

**lemma** *iSince-iEx-conv*:  $(\text{True}. t' \mathcal{S} t I. P t) = (\Diamond t I. P t)$  **by** *blast*

**lemma** *iRelease-iAll-conv*:  $(\text{False}. t' \mathcal{R} t I. P t) = (\Box t I. P t)$   
**by** (*simp add: iRelease-def*)

**lemma** *iTrigger-iAll-conv*:  $(\text{False}. t' \mathcal{T} t I. P t) = (\Box t I. P t)$   
**by** (*simp add: iTrigger-def*)

**lemma** *iWeakUntil-iUntil-conv*:  $(P t'. t' \mathcal{W} t I. Q t) = ((P t'. t' \mathcal{U} t I. Q t) \vee (\Box t I. P t))$

**unfolding** *iWeakUntil-def iUntil-def* **by** *blast*

**lemma** *iWeakSince-iSince-conv*:  $(P\ t'.\ t'\ \mathcal{B}\ t\ I.\ Q\ t) = ((P\ t'.\ t'\ \mathcal{S}\ t\ I.\ Q\ t) \vee (\Box\ t\ I.\ P\ t))$

**unfolding** *iWeakSince-def iSince-def* **by** *blast*

**lemma** *iUntil-iWeakUntil-conv*:  $(P\ t'.\ t'\ \mathcal{U}\ t\ I.\ Q\ t) = ((P\ t'.\ t'\ \mathcal{W}\ t\ I.\ Q\ t) \wedge (\Diamond\ t\ I.\ Q\ t))$

**by** (*subst iWeakUntil-iUntil-conv, blast*)

**lemma** *iSince-iWeakSince-conv*:  $(P\ t'.\ t'\ \mathcal{S}\ t\ I.\ Q\ t) = ((P\ t'.\ t'\ \mathcal{B}\ t\ I.\ Q\ t) \wedge (\Diamond\ t\ I.\ Q\ t))$

**by** (*subst iWeakSince-iSince-conv, blast*)

**lemma** *iRelease-iWeakUntil-conv*:  $(P\ t'.\ t'\ \mathcal{R}\ t\ I.\ Q\ t) = (Q\ t'.\ t'\ \mathcal{W}\ t\ I.\ (Q\ t \wedge P\ t))$

**apply** (*unfold iRelease-def iWeakUntil-def*)

**apply** (*simp add: cut-le-less-conv-if*)

**apply** *blast*

**done**

**lemma** *iRelease-iUntil-conv*:  $(P\ t'.\ t'\ \mathcal{R}\ t\ I.\ Q\ t) = ((\Box\ t\ I.\ Q\ t) \vee (Q\ t'.\ t'\ \mathcal{U}\ t\ I.\ (Q\ t \wedge P\ t)))$

**by** (*fastforce simp: iRelease-iWeakUntil-conv iWeakUntil-iUntil-conv*)

**lemma** *iTrigger-iWeakSince-conv*:  $(P\ t'.\ t'\ \mathcal{T}\ t\ I.\ Q\ t) = (Q\ t'.\ t'\ \mathcal{B}\ t\ I.\ (Q\ t \wedge P\ t))$

**apply** (*unfold iTrigger-def iWeakSince-def*)

**apply** (*simp add: cut-ge-greater-conv-if*)

**apply** *blast*

**done**

**lemma** *iTrigger-iSince-conv*:  $(P\ t'.\ t'\ \mathcal{T}\ t\ I.\ Q\ t) = ((\Box\ t\ I.\ Q\ t) \vee (Q\ t'.\ t'\ \mathcal{S}\ t\ I.\ (Q\ t \wedge P\ t)))$

**by** (*fastforce simp: iTrigger-iWeakSince-conv iWeakSince-iSince-conv*)

**lemma** *iRelease-not-iUntil-conv*:  $(P\ t'.\ t'\ \mathcal{R}\ t\ I.\ Q\ t) = (\neg (\neg P\ t'.\ t'\ \mathcal{U}\ t\ I.\ \neg Q\ t))$

**apply** (*simp only: iUntil-def iRelease-def not-iAll not-iEx de-Morgan-conj not-not*)

**apply** (*case-tac  $\Box\ t\ I.\ Q\ t$ , blast*)

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

**apply** *clarsimp*

**apply** (*rule iffI*)

**apply** (*elim iexE, intro iallI, rename-tac t1 t2*)

**apply** (*case-tac  $t2 \leq t1$ , blast*)

**apply** (*simp add: linorder-not-le, blast*)

**apply** (*frule-tac  $t=t$  in ispec, assumption*)

**apply** *clarsimp*

**apply** (*rule-tac  $t=iMin\ \{t \in I.\ P\ t\}$  in iexI*)

**prefer** 2

```

apply (blast intro: subsetD[OF - iMinI-ex])
apply (rule conjI)
apply (blast intro: iMinI2)
apply (clarsimp simp: cut-le-mem-iff, rename-tac t1 t2)
apply (drule-tac t=t2 in ispec, assumption)
apply (clarsimp simp: cut-less-mem-iff)
apply (frule-tac x=t' in order-less-le-trans, assumption)
apply (drule not-less-iMin)
apply simp
done
lemma iUntil-not-iRelease-conv:  $(P t'. t' \mathcal{U} t I. Q t) = (\neg (\neg P t'. t' \mathcal{R} t I. \neg Q t))$ 
by (simp add: iRelease-not-iUntil-conv)

```

The Trigger operator  $\mathcal{T}$  is a past operator, so that it is used for time intervals, that are bounded by a current time point, and thus are finite. For an infinite interval the stated relation to the Since operator  $\mathcal{S}$  would not be fulfilled.

```

lemma iTrigger-not-iSince-conv:  $finite\ I \implies (P t'. t' \mathcal{T} t I. Q t) = (\neg (\neg P t'. t' \mathcal{S} t I. \neg Q t))$ 
apply (unfold iTrigger-def iSince-def)
apply (case-tac  $\square t I. Q t$ , blast)
apply (simp (no-asm-simp))
apply clarsimp
apply (rule iffI)
apply (elim iexE conjE, rule iallI, rename-tac t1 t2)
apply (case-tac  $t2 \geq t1$ , blast)
apply (simp add: linorder-not-le, blast)
apply (frule-tac t=t in ispec, assumption)
apply (erule disjE, blast)
apply (erule iexE)
apply (subgoal-tac finite  $\{t \in I. P t\}$ )
prefer 2
apply (blast intro: subset-finite-imp-finite)
apply (rule-tac t=Max  $\{t \in I. P t\}$  in iexI)
prefer 2
apply (blast intro: subsetD[OF - MaxI])
apply (rule conjI)
apply (blast intro: MaxI2)
apply (clarsimp simp: cut-ge-mem-iff, rename-tac t1 t2)
apply (drule-tac t=t2 in ispec, assumption)
apply (clarsimp simp: cut-greater-mem-iff, rename-tac t')
apply (frule-tac z=t' in order-le-less-trans, assumption)
apply (drule-tac  $A=\{t \in I. P t\}$  in not-greater-Max[rotated 1])
apply simp+
done

lemma iSince-not-iTrigger-conv:  $finite\ I \implies (P t'. t' \mathcal{S} t I. Q t) = (\neg (\neg P t'. t' \mathcal{T} t I. \neg Q t))$ 
by (simp add: iTrigger-not-iSince-conv)

```

**lemma not-iUntil:**

$$\begin{aligned} & (\neg (P \ t1. \ t1 \ \mathcal{U} \ t2 \ I. \ Q \ t2)) = \\ & (\Box \ t \ I. \ (Q \ t \longrightarrow (\Diamond \ t' \ (I \ \downarrow < \ t). \ \neg \ P \ t'))) \end{aligned}$$

**unfolding iTL-defs by blast**

**lemma not-iSince:**

$$\begin{aligned} & (\neg (P \ t1. \ t1 \ \mathcal{S} \ t2 \ I. \ Q \ t2)) = \\ & (\Box \ t \ I. \ (Q \ t \longrightarrow (\Diamond \ t' \ (I \ \downarrow > \ t). \ \neg \ P \ t'))) \end{aligned}$$

**unfolding iTL-defs by blast**

**lemma iWeakUntil-conj-iUntil-conv:**

$$(P \ t1. \ t1 \ \mathcal{W} \ t2 \ I. \ (P \ t2 \ \wedge \ Q \ t2)) = (\neg (\neg \ Q \ t1. \ t1 \ \mathcal{U} \ t2 \ I. \ \neg \ P \ t2))$$

**by (simp add: iRelease-not-iUntil-conv[symmetric] iRelease-iWeakUntil-conv)**

**lemma iUntil-disj-iUntil-conv:**

$$\begin{aligned} & (P \ t1 \ \vee \ Q \ t1. \ t1 \ \mathcal{U} \ t2 \ I. \ Q \ t2) = \\ & (P \ t1. \ t1 \ \mathcal{U} \ t2 \ I. \ Q \ t2) \end{aligned}$$

**apply (unfold iUntil-def)**

**apply (rule iffI)**

**prefer 2**

**apply blast**

**apply (clarsimp, rename-tac t1)**

**apply (rule-tac t=iMin {t ∈ I. Q t} in iexI)**

**apply (subgoal-tac Q (iMin {t ∈ I. Q t}))**

**prefer 2**

**apply (blast intro: iMinI2)**

**apply (clarsimp, rename-tac t2)**

**apply (frule Collect-not-less-iMin, simp)**

**apply (subgoal-tac t2 < t1)**

**prefer 2**

**apply (rule order-less-le-trans, assumption)**

**apply (simp add: Collect-iMin-le)**

**apply blast**

**apply (rule subsetD[OF - iMinI])**

**apply blast+**

**done**

**lemma iWeakUntil-disj-iWeakUntil-conv:**

$$\begin{aligned} & (P \ t1 \ \vee \ Q \ t1. \ t1 \ \mathcal{W} \ t2 \ I. \ Q \ t2) = \\ & (P \ t1. \ t1 \ \mathcal{W} \ t2 \ I. \ Q \ t2) \end{aligned}$$

**apply (simp only: iWeakUntil-iUntil-conv iUntil-disj-iUntil-conv)**

**apply (case-tac P t1. t1 U t2 I. Q t2, simp+)**

**apply (case-tac □ t I. P t, blast)**

**apply (simp add: not-iUntil)**

**apply (clarsimp, rename-tac t1)**

**apply (case-tac ¬ Q t1, blast)**

```

apply (subgoal-tac iMin {t ∈ I. Q t} ∈ I)
prefer 2
apply (blast intro: subsetD[OF - iMinI])
apply (frule-tac t=iMin {t ∈ I. Q t} in ispec, assumption)
apply (drule mp)
apply (blast intro: iMinI2)
apply (clarsimp, rename-tac t2)
apply (subgoal-tac ¬ Q t2)
prefer 2
apply (drule Collect-not-less-iMin)
apply (simp add: cut-less-mem-iff)
apply blast
done

```

```

lemma iWeakUntil-iUntil-conj-conv:
  (P t1. t1 W t2 I. Q t2) =
  (¬ (¬ Q t1. t1 U t2 I. (¬ P t2 ∧ ¬ Q t2)))
apply (subst iWeakUntil-disj-iWeakUntil-conv[symmetric])
apply (subst de-Morgan-disj[symmetric])
apply (subst iWeakUntil-conj-iUntil-conv[symmetric])
apply (simp add: conj-disj-distribR conj-disj-absorb)
done

```

Negation and temporal operators

```

lemma
  not-iNext[simp]: (¬ (○ t t0 I. P t)) = (○ t t0 I. ¬ P t) and
  not-iNextWeak[simp]: (¬ (○W t t0 I. P t)) = (○S t t0 I. ¬ P t) and
  not-iNextStrong[simp]: (¬ (○S t t0 I. P t)) = (○W t t0 I. ¬ P t) and
  not-iLast[simp]: (¬ (⊖ t t0 I. P t)) = (⊖ t t0 I. ¬ P t) and
  not-iLastWeak[simp]: (¬ (⊖W t t0 I. P t)) = (⊖S t t0 I. ¬ P t) and
  not-iLastStrong[simp]: (¬ (⊖S t t0 I. P t)) = (⊖W t t0 I. ¬ P t)
by (simp-all add: iTL-Next-defs)

```

```

lemma not-iWeakUntil:
  (¬ (P t1. t1 W t2 I. Q t2)) =
  ((□ t I. (Q t → (◇ t' (I ↓< t). ¬ P t'))) ∧ (◇ t I. ¬ P t))
by (simp add: iWeakUntil-iUntil-conv not-iUntil)

```

```

lemma not-iWeakSince:
  (¬ (P t1. t1 B t2 I. Q t2)) =
  ((□ t I. (Q t → (◇ t' (I ↓> t). ¬ P t'))) ∧ (◇ t I. ¬ P t))
by (simp add: iWeakSince-iSince-conv not-iSince)

```

```

lemma not-iRelease:
  (¬ (P t'. t' R t I. Q t)) =
  ((◇ t I. ¬ Q t) ∧ (□ t I. P t → (◇ t I ↓≤ t. ¬ Q t)))
by (simp add: iRelease-def)

```

```

lemma not-iRelease-iUntil-conv:
  (¬ (P t'. t' R t I. Q t)) = (¬ P t'. t' U t I. ¬ Q t)

```

**by** (*simp add: iUntil-not-iRelease-conv*)

**lemma** *not-iTrigger*:

$(\neg (P t'. t' \mathcal{T} t I. Q t)) =$   
 $((\diamond t I. \neg Q t) \wedge (\Box t I. \neg P t \vee (\diamond t I \downarrow \geq t. \neg Q t)))$   
**by** (*simp add: iTrigger-def*)

**lemma** *not-iTrigger-iSince-conv*:

$finite I \implies (\neg (P t'. t' \mathcal{T} t I. Q t)) = (\neg P t'. t' \mathcal{S} t I. \neg Q t)$   
**by** (*simp add: iSince-not-iTrigger-conv*)

### 3.2.5 Some implication results

**lemma** *all-imp-iall*:  $\forall x. P x \implies \Box t I. P t$  **by** *blast*

**lemma** *bex-imp-lex*:  $\diamond t I. P t \implies \exists x. P x$  **by** *blast*

**lemma** *iAll-imp-iEx*:  $I \neq \{\}$   $\implies \Box t I. P t \implies \diamond t I. P t$  **by** *blast*

**lemma** *i-set-iAll-imp-iEx*:  $I \in i\text{-set} \implies \Box t I. P t \implies \diamond t I. P t$   
**by** (*rule iAll-imp-iEx, rule i-set-imp-not-empty*)

**lemmas** *iT-iAll-imp-iEx = iT-not-empty[THEN iAll-imp-iEx]*

**lemma** *iUntil-imp-iEx*:  $P t1. t1 \mathcal{U} t2 I. Q t2 \implies \diamond t I. Q t$   
**unfolding** *iTL-defs* **by** *blast*

**lemma** *iSince-imp-iEx*:  $P t1. t1 \mathcal{S} t2 I. Q t2 \implies \diamond t I. Q t$   
**unfolding** *iTL-defs* **by** *blast*

**lemma** *iall-subset-imp-iall*:  $\llbracket \Box t B. P t; A \subseteq B \rrbracket \implies \Box t A. P t$   
**by** *blast*

**lemma** *iex-subset-imp-iex*:  $\llbracket \diamond t A. P t; A \subseteq B \rrbracket \implies \diamond t B. P t$   
**by** *blast*

**lemma** *iall-mp*:  $\llbracket \Box t I. P t \longrightarrow Q t; \Box t I. P t \rrbracket \implies \Box t I. Q t$  **by** *blast*

**lemma** *iex-mp*:  $\llbracket \Box t I. P t \longrightarrow Q t; \diamond t I. P t \rrbracket \implies \diamond t I. Q t$  **by** *blast*

**lemma** *iUntil-imp*:

$\llbracket P1 t1. t1 \mathcal{U} t2 I. Q t2; \Box t I. P1 t \longrightarrow P2 t \rrbracket \implies P2 t1. t1 \mathcal{U} t2 I. Q t2$   
**unfolding** *iTL-defs* **by** *blast*

**lemma** *iSince-imp*:

$\llbracket P1 t1. t1 \mathcal{S} t2 I. Q t2; \Box t I. P1 t \longrightarrow P2 t \rrbracket \implies P2 t1. t1 \mathcal{S} t2 I. Q t2$   
**unfolding** *iTL-defs* **by** *blast*

**lemma** *iWeakUntil-imp*:

$\llbracket P1 t1. t1 \mathcal{W} t2 I. Q t2; \Box t I. P1 t \longrightarrow P2 t \rrbracket \implies P2 t1. t1 \mathcal{W} t2 I. Q t2$   
**unfolding** *iTL-defs* **by** *blast*

**lemma** *iWeakSince-imp*:

$\llbracket P1\ t1.\ t1\ \mathcal{B}\ t2\ I.\ Q\ t2; \Box\ t\ I.\ P1\ t \longrightarrow P2\ t \rrbracket \Longrightarrow P2\ t1.\ t1\ \mathcal{B}\ t2\ I.\ Q\ t2$   
**unfolding** *iTL-defs* **by** *blast*

**lemma** *iRelease-imp*:

$\llbracket P1\ t1.\ t1\ \mathcal{R}\ t2\ I.\ Q\ t2; \Box\ t\ I.\ P1\ t \longrightarrow P2\ t \rrbracket \Longrightarrow P2\ t1.\ t1\ \mathcal{R}\ t2\ I.\ Q\ t2$   
**unfolding** *iTL-defs* **by** *blast*

**lemma** *iTrigger-imp*:

$\llbracket P1\ t1.\ t1\ \mathcal{T}\ t2\ I.\ Q\ t2; \Box\ t\ I.\ P1\ t \longrightarrow P2\ t \rrbracket \Longrightarrow P2\ t1.\ t1\ \mathcal{T}\ t2\ I.\ Q\ t2$   
**unfolding** *iTL-defs* **by** *blast*

**lemma** *iMin-imp-iUntil*:

$\llbracket I \neq \{\}; Q\ (iMin\ I) \rrbracket \Longrightarrow P\ t'.\ t'\ \mathcal{U}\ t\ I.\ Q\ t$   
**apply** (*unfold iUntil-def*)  
**apply** (*rule-tac t=iMin I in iexI*)  
**apply** (*simp add: cut-less-Min-empty*)  
**apply** (*blast intro: iMinI-ex2*)  
**done**

**lemma** *Max-imp-iSince*:

$\llbracket finite\ I; I \neq \{\}; Q\ (Max\ I) \rrbracket \Longrightarrow P\ t'.\ t'\ \mathcal{S}\ t\ I.\ Q\ t$   
**apply** (*unfold iSince-def*)  
**apply** (*rule-tac t=Max I in iexI*)  
**apply** (*simp add: cut-greater-Max-empty*)  
**apply** (*blast intro: Max-in*)  
**done**

### 3.2.6 Congruence rules for temporal operators' predicates

**lemma** *iAll-cong*:  $\Box\ t\ I.\ f\ t = g\ t \Longrightarrow (\Box\ t\ I.\ P\ (f\ t)\ t) = (\Box\ t\ I.\ P\ (g\ t)\ t)$   
**unfolding** *iTL-defs* **by** *simp*

**lemma** *iEx-cong*:  $\Diamond\ t\ I.\ f\ t = g\ t \Longrightarrow (\Diamond\ t\ I.\ P\ (f\ t)\ t) = (\Diamond\ t\ I.\ P\ (g\ t)\ t)$   
**unfolding** *iTL-defs* **by** *simp*

**lemma** *iUntil-cong1*:

$\Box\ t\ I.\ f\ t = g\ t \Longrightarrow$   
 $(P\ (f\ t1)\ t1.\ t1\ \mathcal{U}\ t2\ I.\ Q\ t2) = (P\ (g\ t1)\ t1.\ t1\ \mathcal{U}\ t2\ I.\ Q\ t2)$   
**apply** (*unfold iUntil-def*)  
**apply** (*rule iEx-cong*)  
**apply** (*rule iallI*)  
**apply** (*rule-tac f= $\lambda x.$  (Q t  $\wedge$  x) in arg-cong, rename-tac t*)  
**apply** (*rule iAll-cong[OF iall-subset-imp-iall[OF - cut-less-subset]]*)  
**apply** (*rule iallI, rename-tac t'*)  
**apply** (*drule-tac t=t' in ispec*)  
**apply** *simp+*

done

**lemma** *iUntil-cong2*:

$\square t I. f t = g t \implies$   
 $(P t1. t1 \mathcal{U} t2 I. Q (f t2) t2) = (P t1. t1 \mathcal{U} t2 I. Q (g t2) t2)$   
**apply** (*unfold iUntil-def*)  
**apply** (*rule iEx-cong*)  
**apply** (*rule iallI, rename-tac t*)  
**apply** (*drule-tac t=t in ispec*)  
**apply** *simp+*  
**done**

**lemma** *iSince-cong1*:

$\square t I. f t = g t \implies$   
 $(P (f t1) t1. t1 \mathcal{S} t2 I. Q t2) = (P (g t1) t1. t1 \mathcal{S} t2 I. Q t2)$   
**apply** (*unfold iSince-def*)  
**apply** (*rule iEx-cong*)  
**apply** (*rule iallI, rename-tac t*)  
**apply** (*rule-tac f= $\lambda x. (Q t \wedge x)$  in arg-cong*)  
**apply** (*rule iAll-cong[OF iall-subset-imp-iall[OF - cut-greater-subset]]*)  
**apply** (*rule iallI, rename-tac t'*)  
**apply** (*drule-tac t=t' in ispec*)  
**apply** *simp+*  
**done**

**lemma** *iSince-cong2*:

$\square t I. f t = g t \implies$   
 $(P t1. t1 \mathcal{S} t2 I. Q (f t2) t2) = (P t1. t1 \mathcal{S} t2 I. Q (g t2) t2)$   
**apply** (*unfold iSince-def*)  
**apply** (*rule iEx-cong*)  
**apply** (*rule iallI, rename-tac t*)  
**apply** (*drule-tac t=t in ispec*)  
**apply** *simp+*  
**done**

**lemma** *bex-subst*:

$\forall x \in A. P x \longrightarrow (Q x = Q' x) \implies$   
 $(\exists x \in A. P x \wedge Q x) = (\exists x \in A. P x \wedge Q' x)$   
**by** *blast*

**lemma** *iEx-subst*:

$\square t I. P t \longrightarrow (Q t = Q' t) \implies$   
 $(\diamond t I. P t \wedge Q t) = (\diamond t I. P t \wedge Q' t)$   
**by** *blast*

### 3.2.7 Temporal operators with set unions/intersections and subsets

**lemma** *iAll-subset*:  $\llbracket A \subseteq B; \Box t B. P t \rrbracket \Longrightarrow \Box t A. P t$   
**by** (*rule iall-subset-imp-iall*)

**lemma** *iEx-subset*:  $\llbracket A \subseteq B; \Diamond t A. P t \rrbracket \Longrightarrow \Diamond t B. P t$   
**by** (*rule iex-subset-imp-iex*)

**lemma** *iUntil-append*:

$\llbracket \text{finite } A; \text{Max } A \leq \text{iMin } B \rrbracket \Longrightarrow$   
 $P t1. t1 \mathcal{U} t2 A. Q t2 \Longrightarrow P t1. t1 \mathcal{U} t2 (A \cup B). Q t2$   
**apply** (*case-tac*  $A = \{\}$ , *simp*)  
**apply** (*unfold iUntil-def*)  
**apply** (*rule iEx-subset[OF Un-upper1]*)  
**apply** (*rule-tac*  $f=\lambda t. A \downarrow < t$  **and**  $g=\lambda t. (A \cup B) \downarrow < t$  **in** *subst[OF iEx-cong, rule-format]*)  
**apply** (*clarsimp simp: cut-less-Un, rename-tac*  $t t'$ )  
**apply** (*cut-tac*  $t=t$  **and**  $I=B$  **in** *cut-less-Min-empty*)  
**apply** *simp+*  
**done**

**lemma** *iSince-prepend*:

$\llbracket \text{finite } A; \text{Max } A \leq \text{iMin } B \rrbracket \Longrightarrow$   
 $P t1. t1 \mathcal{S} t2 B. Q t2 \Longrightarrow P t1. t1 \mathcal{S} t2 (A \cup B). Q t2$   
**apply** (*case-tac*  $B = \{\}$ , *simp*)  
**apply** (*unfold iSince-def*)  
**apply** (*rule iEx-subset[OF Un-upper2]*)  
**apply** (*rule-tac*  $f=\lambda t. B \downarrow > t$  **and**  $g=\lambda t. (A \cup B) \downarrow > t$  **in** *subst[OF iEx-cong, rule-format]*)  
**apply** (*clarsimp simp: cut-greater-Un, rename-tac*  $t t'$ )  
**apply** (*cut-tac*  $t=t$  **and**  $I=A$  **in** *cut-greater-Max-empty*)  
**apply** (*simp add: iMin-ge-iff*)  
**done**

**lemma**

*iAll-union*:  $\llbracket \Box t A. P t; \Box t B. P t \rrbracket \Longrightarrow \Box t (A \cup B). P t$  **and**  
*iAll-union-conv*:  $(\Box t A \cup B. P t) = ((\Box t A. P t) \wedge (\Box t B. P t))$   
**by** *blast+*

**lemma**

*iEx-union*:  $(\Diamond t A. P t) \vee (\Diamond t B. P t) \Longrightarrow \Diamond t (A \cup B). P t$  **and**  
*iEx-union-conv*:  $(\Diamond t (A \cup B). P t) = ((\Diamond t A. P t) \vee (\Diamond t B. P t))$   
**by** *blast+*

**lemma** *iAll-inter*:  $(\Box t A. P t) \vee (\Box t B. P t) \Longrightarrow \Box t (A \cap B). P t$  **by** *blast*

**lemma** *not-iEx-inter*:

$\exists A B P. (\Diamond t A. P t) \wedge (\Diamond t B. P t) \wedge \neg (\Diamond t (A \cap B). P t)$   
**by** (*rule-tac*  $x=\{0\}$  **in** *exI*, *rule-tac*  $x=\{\text{Suc } 0\}$  **in** *exI*, *blast*)

**lemma**

*iAll-insert*:  $\llbracket P t; \Box t I. P t \rrbracket \Longrightarrow \Box t' (\text{insert } t I). P t'$  **and**  
*iAll-insert-conv*:  $(\Box t' (\text{insert } t I). P t') = (P t \wedge (\Box t' I. P t'))$   
**by** *blast+*

**lemma**

*iEx-insert*:  $\llbracket P t \vee (\Diamond t I. P t) \rrbracket \Longrightarrow \Diamond t' (\text{insert } t I). P t'$  **and**  
*iEx-insert-conv*:  $(\Diamond t' (\text{insert } t I). P t') = (P t \vee (\Diamond t' I. P t'))$   
**by** *blast+*

### 3.3 Further results for temporal operators

**lemma** *Collect-minI-iEx*:  $\Diamond t I. P t \Longrightarrow \Diamond t I. P t \wedge (\Box t' (I \downarrow < t). \neg P t')$   
**by** (*unfold iAll-def iEx-def, rule Collect-minI-ex-cut*)

**lemma** *iUntil-disj-conv1*:

$I \neq \{\}$   $\Longrightarrow$   
 $(P t'. t' \mathcal{U} t I. Q t) = (Q (iMin I) \vee (P t'. t' \mathcal{U} t I. Q t \wedge iMin I < t))$   
**apply** (*case-tac Q (iMin I)*)  
**apply** (*simp add: iMin-imp-iUntil*)  
**apply** (*unfold iUntil-def, blast*)  
**done**

**lemma** *iSince-disj-conv1*:

$\llbracket \text{finite } I; I \neq \{\} \rrbracket \Longrightarrow$   
 $(P t'. t' \mathcal{S} t I. Q t) = (Q (Max I) \vee (P t'. t' \mathcal{S} t I. Q t \wedge t < Max I))$   
**apply** (*case-tac Q (Max I)*)  
**apply** (*simp add: Max-imp-iSince*)  
**apply** (*unfold iSince-def, blast*)  
**done**

**lemma** *iUntil-next*:

$I \neq \{\}$   $\Longrightarrow$   
 $(P t'. t' \mathcal{U} t I. Q t) =$   
 $(Q (iMin I) \vee (P (iMin I) \wedge (P t'. t' \mathcal{U} t (I \downarrow > (iMin I)). Q t)))$   
**apply** (*case-tac Q (iMin I)*)  
**apply** (*simp add: iMin-imp-iUntil*)  
**apply** (*simp add: iUntil-def*)  
**apply** (*frule iMinI-ex2*)  
**apply** *blast*  
**done**

**lemma** *iSince-prev*:  $\llbracket \text{finite } I; I \neq \{\} \rrbracket \Longrightarrow$

$(P t'. t' \mathcal{S} t I. Q t) =$   
 $(Q (Max I) \vee (Max I) \wedge (P t'. t' \mathcal{S} t (I \downarrow < Max I). Q t))$   
**apply** (*case-tac Q (Max I)*)  
**apply** (*simp add: Max-imp-iSince*)  
**apply** (*simp add: iSince-def*)

```

apply (frule Max-in, assumption)
apply blast
done

```

```

lemma iNext-induct-rule:
   $\llbracket P (iMin I); \Box t I. (P t \longrightarrow (\bigcirc t' t I. P t')) \rrbracket \Longrightarrow P t$ 
apply (rule inext-induct[of - I])
  apply simp
  apply (drule-tac t=n in ispec, assumption)
  apply (simp add: iNext-def)
apply assumption
done

```

```

lemma iNext-induct:
   $\llbracket P (iMin I); \Box t I. (P t \longrightarrow (\bigcirc t' t I. P t')) \rrbracket \Longrightarrow \Box t I. P t$ 
by (rule iallI, rule iNext-induct-rule)

```

```

lemma iLast-induct-rule:
   $\llbracket P (Max I); \Box t I. (P t \longrightarrow (\ominus t' t I. P t')) \rrbracket \Longrightarrow P t$ 
apply (rule iprev-induct[of - I])
  apply assumption
  apply (drule-tac t=n in ispec, assumption)
  apply (simp add: iLast-def)
apply assumption+
done

```

```

lemma iLast-induct:
   $\llbracket P (Max I); \Box t I. (P t \longrightarrow (\ominus t' t I. P t')) \rrbracket \Longrightarrow \Box t I. P t$ 
by (rule iallI, rule iLast-induct-rule)

```

```

lemma iUntil-conj-not:  $((P t1 \wedge \neg Q t1). t1 \mathcal{U} t2 I. Q t2) = (P t1. t1 \mathcal{U} t2 I. Q t2)$ 
apply (unfold iUntil-def)
apply (rule iffI)
  apply blast
apply (clarsimp, rename-tac t)
apply (rule-tac t=iMin {x \in I. Q x} in iexI)
apply (rule conjI)
  apply (blast intro: iMinI2)
apply (clarsimp simp: cut-less-mem-iff, rename-tac t1)
apply (subgoal-tac iMin {x \in I. Q x} \le t)
  prefer 2
  apply (simp add: iMin-le)
apply (frule order-less-le-trans, assumption)
apply (drule-tac t=t1 in ispec, simp add: cut-less-mem-iff)
apply (rule ccontr, simp)
apply (subgoal-tac t1 \in {x \in I. Q x})
  prefer 2

```

**apply** *blast*  
**apply** (*drule-tac*  $k=t1$  **and**  $I=\{x \in I. Q x\}$  **in** *iMin-le*)  
**apply** *simp*  
**apply** (*blast intro: subsetD*[*OF - iMinI*])  
**done**

**lemma** *iWeakUntil-conj-not*:  $((P t1 \wedge \neg Q t1). t1 \mathcal{W} t2 I. Q t2) = (P t1. t1 \mathcal{W} t2 I. Q t2)$   
**by** (*simp only: iWeakUntil-iUntil-conv iUntil-conj-not, blast*)

**lemma** *iSince-conj-not: finite I  $\implies$*   
 $((P t1 \wedge \neg Q t1). t1 \mathcal{S} t2 I. Q t2) = (P t1. t1 \mathcal{S} t2 I. Q t2)$   
**apply** (*simp only: iSince-def*)  
**apply** (*case-tac*  $I = \{\}$ , *simp*)  
**apply** (*rule iffI*)  
**apply** *blast*  
**apply** (*clarsimp, rename-tac t*)  
**apply** (*subgoal-tac finite*  $\{x \in I. Q x\}$ )  
**prefer** 2  
**apply** *fastforce*  
**apply** (*rule-tac*  $t=Max \{x \in I. Q x\}$  **in** *ieXI*)  
**apply** (*rule conjI*)  
**apply** (*blast intro: MaxI2*)  
**apply** (*clarsimp simp: cut-greater-mem-iff, rename-tac t1*)  
**apply** (*subgoal-tac*  $t \leq Max \{x \in I. Q x\}$ )  
**prefer** 2  
**apply** *simp*  
**apply** (*frule order-le-less-trans, assumption*)  
**apply** (*drule-tac*  $t=t1$  **in** *ispec, simp add: cut-greater-mem-iff*)  
**apply** (*rule ccontr, simp*)  
**apply** (*subgoal-tac*  $t1 \in \{x \in I. Q x\}$ )  
**prefer** 2  
**apply** *blast*  
**apply** (*drule not-greater-Max*[*rotated 1*], *simp+*)  
**apply** (*rule subsetD*[*OF - MaxI*], *fastforce+*)  
**done**

**lemma** *iWeakSince-conj-not: finite I  $\implies$*   
 $((P t1 \wedge \neg Q t1). t1 \mathcal{B} t2 I. Q t2) = (P t1. t1 \mathcal{B} t2 I. Q t2)$   
**by** (*simp only: iWeakSince-iSince-conv iSince-conj-not, blast*)

**lemma** *iNextStrong-imp-iNextWeak*:  $(\bigcirc_S t t0 I. P t) \longrightarrow (\bigcirc_W t t0 I. P t)$

**unfolding** *iTL-Next-defs* **by** *blast*

**lemma** *iLastStrong-imp-iLastWeak*:  $(\ominus_S t t0 I. P t) \longrightarrow (\ominus_W t t0 I. P t)$

**unfolding** *iTL-Next-defs* **by** *blast*

**lemma** *infin-imp-iNextWeak-iNextStrong-eq-iNext*:

$\llbracket \textit{infinite } I; t0 \in I \rrbracket \implies$

$((\circ_W t t0 I. P t) = (\circ t t0 I. P t)) \wedge ((\circ_S t t0 I. P t) = (\circ t t0 I. P t))$   
**by** (*simp add: iTL-Next-iff nat-cut-greater-infinite-not-empty*)

**lemma** *infin-imp-iNextWeak-eq-iNext*:  $\llbracket \text{infinite } I; t0 \in I \rrbracket \implies (\circ_W t t0 I. P t) = (\circ t t0 I. P t)$

**by** (*simp add: infin-imp-iNextWeak-iNextStrong-eq-iNext*)

**lemma** *infin-imp-iNextStrong-eq-iNext*:  $\llbracket \text{infinite } I; t0 \in I \rrbracket \implies (\circ_S t t0 I. P t) = (\circ t t0 I. P t)$

**by** (*simp add: infin-imp-iNextWeak-iNextStrong-eq-iNext*)

**lemma** *infin-imp-iNextStrong-eq-iNextWeak*:  $\llbracket \text{infinite } I; t0 \in I \rrbracket \implies (\circ_S t t0 I. P t) = (\circ_W t t0 I. P t)$

**by** (*simp add: infin-imp-iNextWeak-eq-iNext infin-imp-iNextStrong-eq-iNext*)

**lemma**

*not-in-iNext-eq*:  $t0 \notin I \implies (\circ t t0 I. P t) = (P t0)$  **and**

*not-in-iLast-eq*:  $t0 \notin I \implies (\ominus t t0 I. P t) = (P t0)$

**by** (*simp-all add: iTL-defs not-in-inext-fix not-in-iprev-fix*)

**lemma**

*not-in-iNextWeak-eq*:  $t0 \notin I \implies (\circ_W t t0 I. P t)$  **and**

*not-in-iLastWeak-eq*:  $t0 \notin I \implies (\ominus_W t t0 I. P t)$

**by** (*simp-all add: iNextWeak-iff iLastWeak-iff*)

**lemma**

*not-in-iNextStrong-eq*:  $t0 \notin I \implies \neg (\circ_S t t0 I. P t)$  **and**

*not-in-iLastStrong-eq*:  $t0 \notin I \implies \neg (\ominus_S t t0 I. P t)$

**by** (*simp-all add: iNextStrong-iff iLastStrong-iff*)

**lemma**

*iNext-UNIV*:  $(\circ t t0 UNIV. P t) = P (Suc t0)$  **and**

*iNextWeak-UNIV*:  $(\circ_W t t0 UNIV. P t) = P (Suc t0)$  **and**

*iNextStrong-UNIV*:  $(\circ_S t t0 UNIV. P t) = P (Suc t0)$

**by** (*simp-all add: iTL-Next-defs inext-UNIV cut-greater-singleton*)

**lemma**

*iLast-UNIV*:  $(\ominus t t0 UNIV. P t) = P (t0 - Suc 0)$  **and**

*iLastWeak-UNIV*:  $(\ominus_W t t0 UNIV. P t) = (\text{if } 0 < t0 \text{ then } P (t0 - Suc 0) \text{ else True})$  **and**

*iLastStrong-UNIV*:  $(\ominus_S t t0 UNIV. P t) = (\text{if } 0 < t0 \text{ then } P (t0 - Suc 0) \text{ else False})$

**by** (*simp-all add: iTL-Next-defs iprev-UNIV cut-less-singleton*)

**lemmas** *iTL-Next-UNIV* =

*iNext-UNIV iNextWeak-UNIV iNextStrong-UNIV*

*iLast-UNIV iLastWeak-UNIV iLastStrong-UNIV*

**lemma** *inext-nth-iNext-Suc*:  $(\circ t (I \rightarrow n) I. P t) = P (I \rightarrow Suc n)$

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

**lemma** *iprev-nth-iLast-Suc*:  $(\ominus t (I \leftarrow n) I. P t) = P (I \leftarrow \text{Suc } n)$   
**by** (*simp add: iLast-def*)

### 3.4 Temporal operators and arithmetic interval operators

Shifting intervals through addition and subtraction of constants. Mirroring intervals through subtraction of intervals from constants. Expanding and compressing intervals through multiplication and division by constants.

Always operator

**lemma** *iT-Plus-iAll-conv*:  $(\Box t I \oplus k. P t) = (\Box t I. P (t + k))$   
**apply** (*unfold iAll-def Ball-def*)  
**apply** (*rule iffI*)  
**apply** (*clarify, rename-tac x*)  
**apply** (*drule-tac x=x + k in spec*)  
**apply** (*simp add: iT-Plus-mem-iff2*)  
**apply** (*clarify, rename-tac x*)  
**apply** (*drule-tac x=x - k in spec*)  
**apply** (*simp add: iT-Plus-mem-iff*)  
**done**

**lemma** *iT-Mult-iAll-conv*:  $(\Box t I \otimes k. P t) = (\Box t I. P (t * k))$   
**apply** (*unfold iAll-def Ball-def*)  
**apply** (*case-tac I = {}*)  
**apply** (*simp add: iT-Mult-empty*)  
**apply** (*case-tac k = 0*)  
**apply** (*force simp: iT-Mult-0 iTILL-0*)  
**apply** (*rule iffI*)  
**apply** (*clarify, rename-tac x*)  
**apply** (*drule-tac x=x \* k in spec*)  
**apply** (*simp add: iT-Mult-mem-iff2*)  
**apply** (*clarify, rename-tac x*)  
**apply** (*drule-tac x=x div k in spec*)  
**apply** (*simp add: iT-Mult-mem-iff mod-0-div-mult-cancel*)  
**done**

**lemma** *iT-Plus-neg-iAll-conv*:  $(\Box t I \oplus - k. P t) = (\Box t (I \Downarrow \geq k). P (t - k))$   
**apply** (*unfold iAll-def Ball-def*)  
**apply** (*rule iffI*)  
**apply** (*clarify, rename-tac x*)  
**apply** (*drule-tac x=x - k in spec*)  
**apply** (*simp add: iT-Plus-neg-mem-iff2*)  
**apply** (*clarify, rename-tac x*)  
**apply** (*drule-tac x=x + k in spec*)  
**apply** (*simp add: iT-Plus-neg-mem-iff cut-ge-mem-iff*)  
**done**

**lemma** *iT-Minus-iAll-conv*:  $(\Box t k \ominus I. P t) = (\Box t (I \Downarrow \leq k). P (k - t))$   
**apply** (*unfold iAll-def Ball-def*)

```

apply (rule iffI)
  apply (clarify, rename-tac x)
  apply (drule-tac x=k - x in spec)
  apply (simp add: iT-Minus-mem-iff)
apply (clarify, rename-tac x)
apply (drule-tac x=k - x in spec)
apply (simp add: iT-Minus-mem-iff cut-le-mem-iff)
done

```

```

lemma iT-Div-iAll-conv: ( $\square t I \otimes k. P t$ ) = ( $\square t I. P (t \text{ div } k)$ )
apply (case-tac I = {})
  apply (simp add: iT-Div-empty)
apply (case-tac k = 0)
  apply (force simp: iT-Div-0 iTILL-0)
apply (unfold iAll-def Ball-def)
apply (rule iffI)
  apply (clarify, rename-tac x)
  apply (drule-tac x=x div k in spec)
  apply (simp add: iT-Div-imp-mem)
apply (blast dest: iT-Div-mem-iff[THEN iffD1])
done

```

```

lemmas iT-arith-iAll-conv =
  iT-Plus-iAll-conv
  iT-Mult-iAll-conv
  iT-Plus-neg-iAll-conv
  iT-Minus-iAll-conv
  iT-Div-iAll-conv

```

Eventually operator

```

lemma
  iT-Plus-iEx-conv: ( $\diamond t I \oplus k. P t$ ) = ( $\diamond t I. P (t + k)$ ) and
  iT-Mult-iEx-conv: ( $\diamond t I \otimes k. P t$ ) = ( $\diamond t I. P (t * k)$ ) and
  iT-Plus-neg-iEx-conv: ( $\diamond t I \oplus - k. P t$ ) = ( $\diamond t (I \downarrow \geq k). P (t - k)$ ) and
  iT-Minus-iEx-conv: ( $\diamond t k \ominus I. P t$ ) = ( $\diamond t (I \downarrow \leq k). P (k - t)$ ) and
  iT-Div-iEx-conv: ( $\diamond t I \otimes k. P t$ ) = ( $\diamond t I. P (t \text{ div } k)$ )
by (simp-all only: iEx-iAll-conv iT-arith-iAll-conv)

```

Until and Since operators

```

lemma iT-Plus-iUntil-conv: ( $P t1. t1 \mathcal{U} t2 (I \oplus k). Q t2$ ) = ( $P (t1 + k). t1 \mathcal{U} t2 I. Q (t2 + k)$ )
by (simp add: iUntil-def iT-Plus-iAll-conv iT-Plus-iEx-conv iT-Plus-cut-less2)

```

```

lemma iT-Mult-iUntil-conv: ( $P t1. t1 \mathcal{U} t2 (I \otimes k). Q t2$ ) = ( $P (t1 * k). t1 \mathcal{U} t2 I. Q (t2 * k)$ )
apply (case-tac I = {})
  apply (simp add: iT-Mult-empty)
apply (case-tac k = 0)
  apply (force simp add: iT-Mult-0 iTILL-0)

```

**apply** (*simp add: iUntil-def iT-Mult-iAll-conv iT-Mult-iEx-conv iT-Mult-cut-less2*)  
**done**

**lemma** *iT-Plus-neg-iUntil-conv*:  $(P\ t1.\ t1\ \mathcal{U}\ t2\ (I\ \oplus\ -\ k).\ Q\ t2) = (P\ (t1\ -\ k).\ t1\ \mathcal{U}\ t2\ (I\ \downarrow_{\geq}\ k).\ Q\ (t2\ -\ k))$

**apply** (*simp add: iUntil-def iT-Plus-neg-iAll-conv iT-Plus-neg-iEx-conv iT-Plus-neg-cut-less2*)  
**apply** (*simp add: i-cut-commute-disj*)  
**done**

**lemma** *iT-Minus-iUntil-conv*:  $(P\ t1.\ t1\ \mathcal{U}\ t2\ (k\ \ominus\ I).\ Q\ t2) = (P\ (k\ -\ t1).\ t1\ \mathcal{S}\ t2\ (I\ \downarrow_{\leq}\ k).\ Q\ (k\ -\ t2))$

**apply** (*simp add: iUntil-def iSince-def iT-Minus-iAll-conv iT-Minus-iEx-conv iT-Minus-cut-less2*)  
**apply** (*simp add: i-cut-commute-disj*)  
**done**

**lemma** *iT-Div-iUntil-conv*:  $(P\ t1.\ t1\ \mathcal{U}\ t2\ (I\ \odot\ k).\ Q\ t2) = (P\ (t1\ \text{div}\ k).\ t1\ \mathcal{U}\ t2\ I.\ Q\ (t2\ \text{div}\ k))$

**apply** (*case-tac I = {}*)  
**apply** (*simp add: iT-Div-empty*)  
**apply** (*case-tac k = 0*)  
**apply** (*force simp add: iT-Div-0 iTILL-0*)  
**apply** (*simp add: iUntil-def iT-Div-iAll-conv iT-Div-iEx-conv iT-Div-cut-less2*)  
**apply** (*rule iffI*)  
**apply** (*clarsimp, rename-tac t*)  
**apply** (*subgoal-tac I \downarrow\_{\geq} (t - t mod k) \neq {}*)  
**prefer** 2  
**apply** (*simp add: cut-ge-not-empty-iff*)  
**apply** (*rule-tac x=t in bexI*)  
**apply** *simp+*  
**apply** (*case-tac t mod k = 0*)  
**apply** (*rule-tac t=t in iexI*)  
**apply** *simp+*  
**apply** (*rule-tac t=iMin (I \downarrow\_{\geq} (t - t mod k)) in iexI*)  
**apply** (*subgoal-tac*  
 $t - t\ \text{mod}\ k \leq iMin\ (I\ \downarrow_{\geq}\ (t - t\ \text{mod}\ k)) \wedge$   
 $iMin\ (I\ \downarrow_{\geq}\ (t - t\ \text{mod}\ k)) \leq t$ )  
**prefer** 2  
**apply** (*rule conjI*)  
**apply** (*blast intro: cut-ge-Min-greater*)  
**apply** (*simp add: iMin-le cut-ge-mem-iff*)  
**apply** *clarify*  
**apply** (*rule-tac t=iMin (I \downarrow\_{\geq} (t - t mod k)) div k and s=t div k in subst*)  
**apply** (*rule order-antisym*)  
**apply** (*drule-tac m=t - t mod k and k=k in div-le-mono*)  
**apply** (*simp add: sub-mod-div-eq-div*)  
**apply** (*rule div-le-mono, assumption*)  
**apply** (*clarsimp, rename-tac t1*)  
**apply** (*subgoal-tac t1 \in I \downarrow\_{<} (t - t mod k) \cup I \downarrow\_{\geq} (t - t mod k)*)  
**prefer** 2

```

apply (simp add: cut-less-cut-ge-ident)
apply (subgoal-tac t1 ∉ I ↓ ≥ (t - t mod k))
prefer 2
apply (blast dest: not-less-iMin)
apply blast
apply (blast intro: subsetD[OF - iMinI-ex2])
apply (clarsimp, rename-tac t)
apply (rule-tac t=t in iexI)
apply simp
apply (rule-tac B=I ↓ < t in iAll-subset)
apply (simp add: cut-less-mono)
apply simp+
done

```

Until and Since operators can be converted into each other through sub-  
straction of intervals from constants

```

lemma iUntil-iSince-conv:
   $\llbracket \text{finite } I; \text{Max } I \leq k \rrbracket \implies$ 
   $(P \ t1. \ t1 \ \mathcal{U} \ t2 \ I. \ Q \ t2) = (P \ (k - t1). \ t1 \ \mathcal{S} \ t2 \ (k \ominus I). \ Q \ (k - t2))$ 
apply (case-tac I = {})
apply (simp add: iT-Minus-empty)
apply (frule le-trans[OF iMin-le-Max], assumption+)
apply (subgoal-tac Max (k ⊖ I) ≤ k)
prefer 2
apply (simp add: iT-Minus-Max)
apply (subgoal-tac iMin (k ⊖ I) ≤ k)
prefer 2
apply (rule order-trans[OF iMin-le-Max])
apply (simp add: iT-Minus-finite iT-Minus-empty-iff del: Max-le-iff)
apply (rule-tac t=P t1. t1 U t2 I. Q t2 and s=P t1. t1 U t2 (k ⊖ (k ⊖ I)). Q  
t2 in subst)
apply (simp add: iT-Minus-Minus-eq)
apply (simp add: iT-Minus-iUntil-conv cut-le-Max-all iT-Minus-finite)
done

```

```

lemma iSince-iUntil-conv:
   $\llbracket \text{finite } I; \text{Max } I \leq k \rrbracket \implies$ 
   $(P \ t1. \ t1 \ \mathcal{S} \ t2 \ I. \ Q \ t2) = (P \ (k - t1). \ t1 \ \mathcal{U} \ t2 \ (k \ominus I). \ Q \ (k - t2))$ 
apply (case-tac I = {})
apply (simp add: iT-Minus-empty)
apply (simp (no-asm-simp) add: iT-Minus-iUntil-conv)
apply (simp (no-asm-simp) add: cut-le-Max-all)
apply (unfold iSince-def)
apply (rule iffI)
apply (clarsimp, rename-tac t)
apply (rule-tac t=t in iexI)
apply (frule-tac x=t in bspec, assumption)
apply (clarsimp, rename-tac t1)
apply (drule-tac t=t1 in ispec)

```

```

  apply (simp add: cut-greater-mem-iff)
  apply simp+
  apply (clarsimp, rename-tac t)
  apply (rule-tac t=t in iexI)
  apply (clarsimp, rename-tac t')
  apply (drule-tac t=t' in ispec)
  apply (simp add: cut-greater-mem-iff)
  apply simp+
done

```

**lemma** *iT-Plus-iSince-conv*:  $(P\ t1.\ t1\ \mathcal{S}\ t2\ (I \oplus k). Q\ t2) = (P\ (t1 + k). t1\ \mathcal{S}\ t2\ I.\ Q\ (t2 + k))$   
**by** (simp add: *iSince-def iT-Plus-iAll-conv iT-Plus-iEx-conv iT-Plus-cut-greater2*)

**lemma** *iT-Mult-iSince-conv*:  $0 < k \implies (P\ t1.\ t1\ \mathcal{S}\ t2\ (I \otimes k). Q\ t2) = (P\ (t1 * k). t1\ \mathcal{S}\ t2\ I.\ Q\ (t2 * k))$   
**by** (simp add: *iSince-def iT-Mult-iAll-conv iT-Mult-iEx-conv iT-Mult-cut-greater2*)

**lemma** *iT-Plus-neg-iSince-conv*:  $(P\ t1.\ t1\ \mathcal{S}\ t2\ (I \oplus - k). Q\ t2) = (P\ (t1 - k). t1\ \mathcal{S}\ t2\ (I \downarrow \geq k). Q\ (t2 - k))$   
**apply** (simp add: *iSince-def iT-Plus-neg-iAll-conv iT-Plus-neg-iEx-conv*)  
**apply** (rule *iffI*)  
**apply** (clarsimp, rename-tac t)  
**apply** (simp add: *iT-Plus-neg-cut-greater2*)  
**apply** (rule-tac t=t in iexI)  
**apply** (clarsimp, rename-tac t')  
**apply** (drule-tac t=t' - k in ispec)  
**apply** (simp add: *iT-Plus-neg-mem-iff2 cut-greater-mem-iff*)  
**apply** simp  
**apply** blast  
**apply** (clarsimp, rename-tac t)  
**apply** (rule-tac t=t in iexI)  
**apply** (clarsimp, rename-tac t')  
**apply** (drule-tac t=t' + k in ispec)  
**apply** (simp add: *iT-Plus-neg-mem-iff i-cut-mem-iff*)  
**apply** simp  
**apply** blast  
done

**lemma** *iT-Minus-iSince-conv*:  
 $(P\ t1.\ t1\ \mathcal{S}\ t2\ (k \ominus I). Q\ t2) = (P\ (k - t1). t1\ \mathcal{U}\ t2\ (I \downarrow \leq k). Q\ (k - t2))$   
**apply** (case-tac  $I = \{\}$ )  
**apply** (simp add: *iT-Minus-empty cut-le-empty*)  
**apply** (case-tac  $I \downarrow \leq k = \{\}$ )  
**apply** (simp add: *iT-Minus-image-conv*)  
**apply** (subst *iT-Minus-cut-eq[OF order-refl, symmetric]*)  
**apply** (subst *iSince-iUntil-conv[where k=k]*)  
**apply** (rule *iT-Minus-finite*)  
**apply** (subst *iT-Minus-Max*)

```

apply simp
apply (rule cut-le-bound, rule iMinI-ex2, simp)
apply simp
apply (simp add: iT-Minus-Minus-cut-eq)
done

```

**lemma** *iT-Div-iSince-conv*:

```

 $0 < k \implies (P \ t1. \ t1 \ \mathcal{S} \ t2 \ (I \ \circledast \ k). \ Q \ t2) = (P \ (t1 \ \text{div} \ k). \ t1 \ \mathcal{S} \ t2 \ I. \ Q \ (t2 \ \text{div} \ k))$ 
apply (case-tac I = {})
apply (simp add: iT-Div-empty)
apply (simp add: iSince-def iT-Div-iAll-conv iT-Div-iEx-conv)
apply (simp add: iT-Div-cut-greater)
apply (subgoal-tac  $\forall t. t \leq t \ \text{div} \ k * k + (k - \text{Suc } 0)$ )
prefer 2
apply clarsimp
apply (simp add: div-mult-cancel add.commute[of - k])
apply (simp add: le-add-diff Suc-mod-le-divisor)
apply (rule iffI)
apply (clarsimp, rename-tac t)
apply (drule-tac x=t in spec)
apply (subgoal-tac  $I \downarrow \leq (t \ \text{div} \ k * k + (k - \text{Suc } 0)) \neq \{\}$ )
prefer 2
apply (simp add: cut-le-not-empty-iff)
apply (rule-tac x=t in bexI, assumption+)
apply (subgoal-tac  $t \leq \text{Max} \ (I \downarrow \leq (t \ \text{div} \ k * k + (k - \text{Suc } 0)))$ )
prefer 2
apply (simp add: nat-cut-le-finite cut-le-mem-iff)
apply (subgoal-tac  $\text{Max} \ (I \downarrow \leq (t \ \text{div} \ k * k + (k - \text{Suc } 0))) \leq t \ \text{div} \ k * k + (k - \text{Suc } 0)$ )
prefer 2
apply (simp add: nat-cut-le-finite cut-le-mem-iff)
apply (subgoal-tac  $\text{Max} \ (I \downarrow \leq (t \ \text{div} \ k * k + (k - \text{Suc } 0))) \ \text{div} \ k = t \ \text{div} \ k$ )
prefer 2
apply (rule order-antisym)
apply (rule-tac  $t=t \ \text{div} \ k$  and  $s=(t \ \text{div} \ k * k + (k - \text{Suc } 0)) \ \text{div} \ k$  in subst)
apply (simp only: div-add1-eq1-mod-0-left[OF mod-mult-self2-is-0])
apply simp
apply (rule div-le-mono)
apply (simp only: div-add1-eq1-mod-0-left[OF mod-mult-self2-is-0])
apply simp
apply (rule div-le-mono, assumption)
apply (rule-tac  $t=\text{Max} \ (I \downarrow \leq (t \ \text{div} \ k * k + (k - \text{Suc } 0)))$  in iexI)
apply (clarsimp, rename-tac t1)
apply (subgoal-tac  $t1 \in I$ )
prefer 2
apply assumption
apply (subgoal-tac  $t \ \text{div} \ k * k + (k - \text{Suc } 0) < t1$ )
prefer 2

```

```

apply (rule ccontr)
apply (drule not-greater-Max[OF nat-cut-le-finite])
apply (simp add: i-cut-mem-iff)
apply (drule-tac t=t1 div k in ispec)
apply (simp add: iT-Div-imp-mem cut-greater-mem-iff)
apply assumption
apply (blast intro: subsetD[OF - Max-in[OF nat-cut-le-finite]])
apply (clarsimp, rename-tac t)
apply (drule-tac x=t in spec)
apply (rule-tac t=t in iexI)
apply (clarsimp simp: iT-Div-mem-iff, rename-tac t1 t2)
apply (drule-tac t=t2 in ispec)
apply (simp add: cut-greater-mem-iff)
apply simp+
done

```

Weak Until and Weak Since operators

**lemma** *iT-Plus-iWeakUntil-conv*:  $(P\ t1.\ t1\ \mathcal{W}\ t2\ (I \oplus k).\ Q\ t2) = (P\ (t1 + k).\ t1\ \mathcal{W}\ t2\ I.\ Q\ (t2 + k))$   
**by** (simp add: iWeakUntil-iUntil-conv iT-Plus-iUntil-conv iT-Plus-iAll-conv)

**lemma** *iT-Mult-iWeakUntil-conv*:  $(P\ t1.\ t1\ \mathcal{W}\ t2\ (I \otimes k).\ Q\ t2) = (P\ (t1 * k).\ t1\ \mathcal{W}\ t2\ I.\ Q\ (t2 * k))$   
**by** (simp add: iWeakUntil-iUntil-conv iT-Mult-iUntil-conv iT-Mult-iAll-conv)

**lemma** *iT-Plus-neg-iWeakUntil-conv*:  $(P\ t1.\ t1\ \mathcal{W}\ t2\ (I \oplus - k).\ Q\ t2) = (P\ (t1 - k).\ t1\ \mathcal{W}\ t2\ (I \downarrow \geq k).\ Q\ (t2 - k))$   
**by** (simp add: iWeakUntil-iUntil-conv iT-Plus-neg-iUntil-conv iT-Plus-neg-iAll-conv)

**lemma** *iT-Minus-iWeakUntil-conv*:  $(P\ t1.\ t1\ \mathcal{W}\ t2\ (k \ominus I).\ Q\ t2) = (P\ (k - t1).\ t1\ \mathcal{B}\ t2\ (I \downarrow \leq k).\ Q\ (k - t2))$   
**by** (simp add: iWeakUntil-iUntil-conv iWeakSince-iSince-conv iT-Minus-iUntil-conv iT-Minus-iAll-conv)

**lemma** *iT-Div-iWeakUntil-conv*:  $(P\ t1.\ t1\ \mathcal{W}\ t2\ (I \oslash k).\ Q\ t2) = (P\ (t1\ \text{div}\ k).\ t1\ \mathcal{W}\ t2\ I.\ Q\ (t2\ \text{div}\ k))$   
**by** (simp add: iWeakUntil-iUntil-conv iT-Div-iUntil-conv iT-Div-iAll-conv)

**lemma** *iT-Plus-iWeakSince-conv*:  $(P\ t1.\ t1\ \mathcal{B}\ t2\ (I \oplus k).\ Q\ t2) = (P\ (t1 + k).\ t1\ \mathcal{B}\ t2\ I.\ Q\ (t2 + k))$   
**by** (simp add: iWeakSince-iSince-conv iT-Plus-iSince-conv iT-Plus-iAll-conv)

**lemma** *iT-Mult-iWeakSince-conv*:  $0 < k \implies (P\ t1.\ t1\ \mathcal{B}\ t2\ (I \otimes k).\ Q\ t2) = (P\ (t1 * k).\ t1\ \mathcal{B}\ t2\ I.\ Q\ (t2 * k))$   
**by** (simp add: iWeakSince-iSince-conv iT-Mult-iSince-conv iT-Mult-iAll-conv)

**lemma** *iT-Plus-neg-iWeakSince-conv*:  $(P\ t1.\ t1\ \mathcal{B}\ t2\ (I \oplus - k).\ Q\ t2) = (P\ (t1 - k).\ t1\ \mathcal{B}\ t2\ (I \downarrow \geq k).\ Q\ (t2 - k))$

by (simp add: *iWeakSince-iSince-conv iT-Plus-neg-iSince-conv iT-Plus-neg-iAll-conv*)

**lemma** *iT-Minus-iWeakSince-conv*:

$(P\ t1.\ t1\ \mathcal{B}\ t2\ (k \ominus I). Q\ t2) = (P\ (k - t1).\ t1\ \mathcal{W}\ t2\ (I \downarrow \leq k). Q\ (k - t2))$

by (simp add: *iWeakSince-iSince-conv iT-Minus-iSince-conv iT-Minus-iAll-conv iWeakUntil-iUntil-conv*)

**lemma** *iT-Div-iWeakSince-conv*:

$0 < k \implies (P\ t1.\ t1\ \mathcal{B}\ t2\ (I \oslash k). Q\ t2) = (P\ (t1\ \text{div}\ k).\ t1\ \mathcal{B}\ t2\ I.\ Q\ (t2\ \text{div}\ k))$

by (simp add: *iWeakSince-iSince-conv iT-Div-iSince-conv iT-Div-iAll-conv*)

Release and Trigger operators

**lemma** *iT-Plus-iRelease-conv*:  $(P\ t1.\ t1\ \mathcal{R}\ t2\ (I \oplus k). Q\ t2) = (P\ (t1 + k).\ t1\ \mathcal{R}\ t2\ I.\ Q\ (t2 + k))$

by (simp add: *iRelease-iWeakUntil-conv iT-Plus-iWeakUntil-conv*)

**lemma** *iT-Mult-iRelease-conv*:  $(P\ t1.\ t1\ \mathcal{R}\ t2\ (I \otimes k). Q\ t2) = (P\ (t1 * k).\ t1\ \mathcal{R}\ t2\ I.\ Q\ (t2 * k))$

by (simp add: *iRelease-iWeakUntil-conv iT-Mult-iWeakUntil-conv*)

**lemma** *iT-Plus-neg-iRelease-conv*:  $(P\ t1.\ t1\ \mathcal{R}\ t2\ (I \oplus - k). Q\ t2) = (P\ (t1 - k).\ t1\ \mathcal{R}\ t2\ (I \downarrow \geq k). Q\ (t2 - k))$

by (simp add: *iRelease-iWeakUntil-conv iT-Plus-neg-iWeakUntil-conv*)

**lemma** *iT-Minus-iRelease-conv*:  $(P\ t1.\ t1\ \mathcal{R}\ t2\ (k \ominus I). Q\ t2) = (P\ (k - t1).\ t1\ \mathcal{T}\ t2\ (I \downarrow \leq k). Q\ (k - t2))$

by (simp add: *iRelease-iWeakUntil-conv iT-Minus-iWeakUntil-conv iTrigger-iSince-conv iWeakSince-iSince-conv disj-commute*)

**lemma** *iT-Div-iRelease-conv*:  $(P\ t1.\ t1\ \mathcal{R}\ t2\ (I \oslash k). Q\ t2) = (P\ (t1\ \text{div}\ k).\ t1\ \mathcal{R}\ t2\ I.\ Q\ (t2\ \text{div}\ k))$

by (simp add: *iRelease-iWeakUntil-conv iT-Div-iWeakUntil-conv*)

**lemma** *iT-Plus-iTrigger-conv*:  $(P\ t1.\ t1\ \mathcal{T}\ t2\ (I \oplus k). Q\ t2) = (P\ (t1 + k).\ t1\ \mathcal{T}\ t2\ I.\ Q\ (t2 + k))$

by (simp add: *iTrigger-iWeakSince-conv iT-Plus-iWeakSince-conv*)

**lemma** *iT-Mult-iTrigger-conv*:  $0 < k \implies (P\ t1.\ t1\ \mathcal{T}\ t2\ (I \otimes k). Q\ t2) = (P\ (t1 * k).\ t1\ \mathcal{T}\ t2\ I.\ Q\ (t2 * k))$

by (simp add: *iTrigger-iWeakSince-conv iT-Mult-iWeakSince-conv*)

**lemma** *iT-Plus-neg-iTrigger-conv*:  $(P\ t1.\ t1\ \mathcal{T}\ t2\ (I \oplus - k). Q\ t2) = (P\ (t1 - k).\ t1\ \mathcal{T}\ t2\ (I \downarrow \geq k). Q\ (t2 - k))$

by (simp add: *iTrigger-iWeakSince-conv iT-Plus-neg-iWeakSince-conv*)

**lemma** *iT-Minus-iTrigger-conv*:

$(P\ t1.\ t1\ \mathcal{T}\ t2\ (k \ominus I). Q\ t2) = (P\ (k - t1).\ t1\ \mathcal{R}\ t2\ (I \downarrow \leq k). Q\ (k - t2))$

**by** (*fastforce simp add: iT-Trigger-iWeakSince-conv iT-Minus-iWeakSince-conv iRelease-iUntil-conv iWeakUntil-iUntil-conv*)

**lemma** *iT-Div-iTrigger-conv*:

$0 < k \implies (P\ t1.\ t1\ \mathcal{T}\ t2\ (I\ \otimes\ k).\ Q\ t2) = (P\ (t1\ \text{div}\ k).\ t1\ \mathcal{T}\ t2\ I.\ Q\ (t2\ \text{div}\ k))$

**by** (*simp add: iT-Trigger-iWeakSince-conv iT-Div-iWeakSince-conv*)

**end**