T U M

I N S T I T U T F Ü R I N F O R M A T I K

Formal Validation of Core SALT Translation to LTL in Isabelle/HOL

David Trachtenherz

TUM-I1105
März 11

T E C H N I S C H E U N I V E R S I T Ä T M Ü N C H E N

TUM-INFO-03-I1105-0/1.-FI

Druck: Institut für Informatik der
Technischen Universität München

Formal Validation of Core SALT Translation to LTL in Isabelle/HOL

Formal semantics definition, translation to LTL, and formal translation validation for core SALT in the Isabelle/HOL theorem prover

David Trachtenherz

February 2011

Abstract

Temporal notations are widely accepted for formal specification of functional properties amenable to automated formal verification. The SALT temporal specification language was developed as an extension of the popular LTL notation to simplify creating temporal specifications: it provides, among others, concise operators and restricted regular expressions. SALT formulas can be translated to LTL by a freely available compiler and thereby directly used for model checking. Clearly defined semantics of the specification notation is indispensable for creating precise unambiguous descriptions of the desired behavioural properties and for making subsequent formal verification meaningful. SALT semantics has been given through translation to LTL so far, which is in parts rather sophisticated and not easily comprehensible. This report presents a clear and explicit semantics formalisation for a substantial language subset of SALT through translation to an expressive interval temporal logic with explicit time variables. The formal definition and validation is performed in the Isabelle/HOL theorem prover. In the course of the formal validation we particularly prove that the semantics resulting from translation to LTL is equivalent to the explicit semantics definition.

Contents

1 Introduction and motivation **3**

2 ILET **4**
2.1 Shallow embedding 4
2.2 Regular expressions (deep embedding) 7
2.2.1 Syntax 7
2.2.2 Semantics 7
2.2.3 Sequence operators and expressions matching empty words 8

3 LTL **11**
3.1 Syntax 11
3.2 Semantics 11

4 Core SALT **12**
4.1 Syntax 12
4.2 Translation to LTL 13
4.2.1 Translation of regular expressions to LTL 13
4.2.2 Translation of *until* and *from* operators to LTL 15
4.2.3 Translation of core SALT formulas to LTL 16
4.3 Semantics 18
4.3.1 Translation of regular expressions to ILET 18
4.3.2 Translation of *until* and *from* operators to ILET 19
4.3.3 Translation of core SALT formulas to ILET 19
4.4 Sequence operators and expressions matching empty words 20
4.5 Formal validation of core SALT translation to LTL 22
4.5.1 Selected auxiliary translation validation lemmas 22
4.5.2 Main translation validation theorem 23

5 Additional results for core SALT **23**
5.1 LTL operators *until*, *weak until*, *release* in core SALT 23
5.2 Expressive equivalence of core SALT and LTL 24

1 Introduction and motivation

SALT [BLS06] is a temporal specification language based on the linear temporal logic LTL [Pnu77] and incorporating aspects of further specification formalisms and frameworks [BBDE+01, DAC99], e.g., restricted regular expressions, specification patterns and further operators. SALT is meant particularly as a bridge to formal but not always user-friendly LTL specification – allowing macro definitions and using textual operator names it much more resembles a programming language than LTL does, and furthermore the operators provided by SALT make it possible to conveniently specify requirements, which can hardly be formulated in LTL without errors due to the complexity of corresponding LTL formulas – compare, for instance, a simple SALT regular expression and the corresponding LTL formula:

$$/\,p\,;\,q*[\geq 3]\,;\,r\,/\;\Leftrightarrow\; p \,\wedge\, \bigcirc\,(q\,\mathcal{U}\,(q \,\wedge\, \bigcirc\,(q \,\wedge\, \bigcirc\,(q \,\wedge\, \bigcirc\, r))))$$

This example also shows an important and critical point about SALT translation to LTL – the concise and quite intuitive SALT operators have to be expressed using the well-defined but minimalist set of LTL operators so that the translation is in parts complex and therefore not easy to comprehend and especially being checked for correctness.

The meaning of SALT operators is informally explained in [Str06]. The translation of SALT to LTL, described in [Str06] and implemented by the SALT compiler [SAL], implicitly gives a SALT semantics. However, there existed no explicit formal semantics definition so far. The advantages of creating such an explicit semantics definition are manifold. [Gor03, Section 2] discusses several aspects that motivated the semantics validation for PSL [Acc04]. This discussion largely applies also to our work, particularly the issues of obtaining a machine-processible semantics as well as the prospect of combining model checking and theorem proving for formal verification of temporal properties of programs.

The main motivation concerns the actual purpose of SALT as language for formal specification of program properties. Similarly to LTL, SALT and many other formal notations are intended to be used for clear and unambiguous specification of functional properties, and in many instances for a subsequent verification. It is thus of crucial importance that their own semantics is clearly and precisely defined. Formalising SALT in a mechanized theorem prover and proving the correctness of its translation to LTL provides both a clear, machine-processible semantics definition of SALT and a formal evidence for the fact that formal verification (e.g., by model checking) of an LTL specification generated from a SALT specification is equivalent to formally verifying the original SALT specification. This would give the firm confidence that we can, instead of manually creating LTL specifications, safely use SALT for creating formal functional specifications and then automatically translate them by means of the SALT compiler into LTL for further applications, especially model checking.

Our first goal is an explicit definition of semantics for a selected SALT subset, comprising most of the core SALT operators (cf. Section 4), performed by translation to the expressive temporal logic ILET [Tra09, Chapter 4.2] [Tra11]. We have chosen ILET for several reasons. Firstly, it makes use of simple syntax and semantics with few basic constructs and allows explicit access to time variables, thus simplifying definitions of both further temporal operators and complete temporal logic notations. Secondly, there already exists a developed Isabelle/HOL theory for its temporal operators including verified results for time intervals and temporal operators, which are directly transferable to temporal logic notations defined through translation to ILET. Finally, it includes operators and verification results for working with bounded time intervals, which is significant with regard to future work comprising translation validation for SALT operators that simulate bounded time intervals (e.g., the ***upto*** operator).

The explicit semantics definition through translation to ILET prepares the ground for the second goal of formally validating the translation of the selected SALT subset to LTL by verifying that the semantics resulting from translation to LTL is equivalent to the explicit semantics definition.

We perform the semantics definition and the formal translation validation in the Isabelle/HOL interactive theorem prover. Familiarity with higher-order logic and Isabelle/HOL notation or similar ones is not required to understand the proof documentation in the presented work, though it would be helpful when reading it. A detailed tutorial on Isabelle/HOL can be found in [NPW02].

2 ILET

ILET (*Interval Logic with Explicit Time*, [Tra11], BPDL in [Tra09, Chapter 4.2]) is a propositional interval temporal logic providing explicit access to time variables and intervals and using natural numbers as time domain.

The propositional part of ILET provides atomic propositions on system computation states and the common Boolean operators. Due to explicity of time variables, propositions can be evaluated on states for any point of time given by an arithmetic expression on time variables.

The temporal part of ILET has a simple syntax and semantics with three basic constructs:[1]

- Temporal operators $\Box$ and $\Diamond$ corresponding to universal and existential quantification on time domain.
- Interval step operator inext calculating the next element of an interval $I \subseteq \mathbb{N}$ with respect to a given element $n \in I$.
- Interval cut operators $\downarrow<$ and $\downarrow\leq$ restricting an interval $I \subseteq \mathbb{N}$ to its elements less / less or equal a given cutting point $n \in \mathbb{N}$.

These constructs are sufficient to define further operators, common to various linear temporal logics, e.g., next or until.

2.1 Shallow embedding

Selected definitions and results for ILET.

Interval cut operators

Cutting intervals/sets at given point. The resulting interval contains all elements of original intervals less / less or equal the cutting point.

consts
  cut-le :: $'a$::*linorder set* $\Rightarrow$ $'a$ $\Rightarrow$ $'a$ *set* (**infixl** $\downarrow\leq$ *100*)
  cut-less :: $'a$::*linorder set* $\Rightarrow$ $'a$ $\Rightarrow$ $'a$ *set* (**infixl** $\downarrow<$ *100*)
defs
  cut-le-def: $I \downarrow\leq t \equiv \{ x{\in}I.\ x \leq t \}$
  cut-less-def: $I \downarrow< t \equiv \{ x{\in}I.\ x < t \}$

Relations between cut operators:

lemma *cut-less-le-conv*: $I \downarrow< t = (I \downarrow\leq t) - \{t\}$
lemma *cut-less-le-conv-if*: $I \downarrow< t = (\mathit{if}\ t \in I\ \mathit{then}\ (I \downarrow\leq t) - \{t\}\ \mathit{else}\ (I \downarrow\leq t))$
lemma *nat-cut-le-less-conv*: $I \downarrow\leq t = I \downarrow< \mathit{Suc}\ t$
lemma *nat-cut-less-le-conv*: $0 < t \Longrightarrow I \downarrow< t = I \downarrow\leq (t - \mathit{Suc}\ 0)$

Operator *inext* for stepping forwards through intervals

Minimal element of a well-ordered set.

constdefs
  iMin :: $'a$::*wellorder set* $\Rightarrow$ $'a$
  $\mathit{iMin}\ I \equiv \mathit{LEAST}\ x.\ x \in I$

Function returning the next element of a natural interval/set *I* with respect to a given number *n*. If *I* contains no greater elements (*n* is maximal element) or *n* is not in *I*, then *n* is returned.

constdefs
  inext :: *nat* $\Rightarrow$ *nat set* $\Rightarrow$ *nat*
  $\mathit{inext}\ n\ I \equiv ($
    $\mathit{if}\ (n \in I \wedge (I \downarrow> n \neq \{\}))$

[1] Here ILET constructs required below are introduced. A complete ILET definition (including further constructs, e.g., operator iprev, which is dual to inext and calculates the previous element of $I \subseteq \mathbb{N}$ w.r.t. some $n \in I$) is given in [Tra09, Chapter 4].

  then $iMin$ $(I \downarrow> n)$
  else $n)$

Operator *inext* on continuous natural intervals.

lemma $inext\text{-}atLeast$: $n \le t \Longrightarrow inext\ t\ \{n..\} = Suc\ t$
lemma $inext\text{-}atMost$: $t < n \Longrightarrow inext\ t\ \{..n\} = Suc\ t$
lemma $inext\text{-}lessThan$: $Suc\ t < n \Longrightarrow inext\ t\ \{..<n\} = Suc\ t$
lemma $inext\text{-}atLeastAtMost$: $[\![\ m \le t;\ t < n\ ]\!] \Longrightarrow inext\ t\ \{m..n\} = Suc\ t$

Temporal operators

ILET uses natural numbers as time domain.

types $Time = nat$
types $iT = Time\ set$

Basic operators *always* and *eventually* corresponding to universal/existential quantification for time variables over time intervals.

consts
  $iAll$ :: $iT \Rightarrow (Time \Rightarrow bool) \Rightarrow bool$ — Always
  iEx :: $iT \Rightarrow (Time \Rightarrow bool) \Rightarrow bool$ — Eventually
defs
  $iAll\text{-}def$: $iAll\ I\ P \equiv \forall t{\in}I.\ P\ t$
  $iEx\text{-}def$: $iEx\ I\ P \equiv \exists t{\in}I.\ P\ t$
syntax $(xsymbols)$
  $\text{-}iAll$ :: $Time \Rightarrow iT \Rightarrow (Time \Rightarrow bool) \Rightarrow bool$ $((3\Box\ \text{-}\ \text{-}./\ \text{-})\ [0,\ 0,\ 10]\ 10)$
  $\text{-}iEx$ :: $Time \Rightarrow iT \Rightarrow (Time \Rightarrow bool) \Rightarrow bool$ $((3\Diamond\ \text{-}\ \text{-}./\ \text{-})\ [0,\ 0,\ 10]\ 10)$
translations
  $\Box\ t\ I.\ P \rightleftharpoons iAll\ I\ (\lambda t.\ P)$
  $\Diamond\ t\ I.\ P \rightleftharpoons iEx\ I\ (\lambda t.\ P)$

Weak and strong *next* operator. The bound formula is evaluated at the next time point in I relatively to t_0. If *inext t0 I* $=$ *t0* (i.e., t_0 is maximal element or $t_0 \notin I$) then *weak next* evaluates to *true* and *strong next* to *false*.

consts
  $iNextWeak$ :: $Time \Rightarrow iT \Rightarrow (Time \Rightarrow bool) \Rightarrow bool$
  $iNextStrong$ :: $Time \Rightarrow iT \Rightarrow (Time \Rightarrow bool) \Rightarrow bool$
defs
  $iNextWeak\text{-}def$: $iNextWeak\ t0\ I\ P \equiv (\Box\ t\ \{inext\ t0\ I\} \downarrow> t0.\ P\ t)$
  $iNextStrong\text{-}def$: $iNextStrong\ t0\ I\ P \equiv (\Diamond\ t\ \{inext\ t0\ I\} \downarrow> t0.\ P\ t)$
syntax $(xsymbols)$
  $\text{-}iNextWeak$ :: $Time \Rightarrow Time \Rightarrow iT \Rightarrow (Time \Rightarrow bool) \Rightarrow bool$
   $((3\bigcirc_W\ \text{-}\ \text{-}\ \text{-}./\ \text{-})\ [0,\ 0,\ 10]\ 10)$
  $\text{-}iNextStrong$ :: $Time \Rightarrow Time \Rightarrow iT \Rightarrow (Time \Rightarrow bool) \Rightarrow bool$
   $((3\bigcirc_S\ \text{-}\ \text{-}\ \text{-}./\ \text{-})\ [0,\ 0,\ 10]\ 10)$
translations
  $\bigcirc_W\ t\ t0\ I.\ P \rightleftharpoons iNextWeak\ t0\ I\ (\lambda t.\ P)$
  $\bigcirc_S\ t\ t0\ I.\ P \rightleftharpoons iNextStrong\ t0\ I\ (\lambda t.\ P)$

Operator *until*: the second formula Q must hold at some time $t \in I$ and the first formula P must hold until this time point.

consts
  $iUntil$ :: $iT \Rightarrow (Time \Rightarrow bool) \Rightarrow (Time \Rightarrow bool) \Rightarrow bool$
defs
  $iUntil\text{-}def$: $iUntil\ I\ P\ Q \equiv \Diamond\ t\ I.\ Q\ t \wedge (\Box\ t'\ (I \downarrow< t).\ P\ t')$
syntax $(xsymbols)$
  $\text{-}iUntil$ :: $Time \Rightarrow Time \Rightarrow iT \Rightarrow (Time \Rightarrow bool) \Rightarrow (Time \Rightarrow bool) \Rightarrow bool$
   $((\text{-}./\ \text{-}\ (3\mathcal{U}\ \text{-}\ \text{-}).\!/\ \text{-})\ [10,\ 0,\ 0,\ 0,\ 10]\ 10)$
translations
  $P.\ t\ \mathcal{U}\ t'\ I.\ Q \rightleftharpoons iUntil\ I\ (\lambda t.\ P)\ (\lambda t'.\ Q)$

Operator *weak until* (also *waiting for*, *unless*): either the previously defined *until* operator must hold, or the first formula P must always hold in I.

consts
$\mathit{iWeakUntil} :: \mathit{iT} \Rightarrow (\mathit{Time} \Rightarrow \mathit{bool}) \Rightarrow (\mathit{Time} \Rightarrow \mathit{bool}) \Rightarrow \mathit{bool}$
defs
$\mathit{iWeakUntil\text{-}def} : \mathit{iWeakUntil}\ I\ P\ Q \equiv$
$(\Box\ t\ I.\ P\ t) \vee (\Diamond\ t\ I.\ Q\ t \wedge (\Box\ t'\ (I \downarrow< t).\ P\ t'))$
syntax $(\mathit{xsymbols})$
$\text{-}\mathit{iWeakUntil} ::\ \mathit{Time} \Rightarrow \mathit{Time} \Rightarrow \mathit{iT} \Rightarrow (\mathit{Time} \Rightarrow \mathit{bool}) \Rightarrow (\mathit{Time} \Rightarrow \mathit{bool}) \Rightarrow \mathit{bool}$
$((\text{-}./\ \text{-}\ (3\mathcal{W}\ \text{-}\ \text{-})./\ \text{-})\ [10,\ 0,\ 0,\ 0,\ 10]\ 10)$
translations
$P.\ t\ \mathcal{W}\ t'\ I.\ Q \rightleftharpoons \mathit{iWeakUntil}\ I\ (\lambda t.\ P)\ (\lambda t'.\ Q)$

Operator *release*: the second formula Q must always hold in I or it must hold until it is released by the first formula P.

consts
$\mathit{iRelease} :: \mathit{iT} \Rightarrow (\mathit{Time} \Rightarrow \mathit{bool}) \Rightarrow (\mathit{Time} \Rightarrow \mathit{bool}) \Rightarrow \mathit{bool}$
defs
$\mathit{iRelease\text{-}def} : \mathit{iRelease}\ I\ P\ Q \equiv$
$(\Box\ t\ I.\ Q\ t) \vee (\Diamond\ t\ I.\ P\ t \wedge (\Box\ t'\ (I \downarrow\leq t).\ Q\ t'))$
syntax $(\mathit{xsymbols})$
$\text{-}\mathit{iRelease} ::\ \mathit{Time} \Rightarrow \mathit{Time} \Rightarrow \mathit{iT} \Rightarrow (\mathit{Time} \Rightarrow \mathit{bool}) \Rightarrow (\mathit{Time} \Rightarrow \mathit{bool}) \Rightarrow \mathit{bool}$
$((\text{-}./\ \text{-}\ (3\mathcal{R}\ \text{-}\ \text{-})./\ \text{-})\ [10,\ 0,\ 0,\ 0,\ 10]\ 10)$
translations
$P.\ t\ \mathcal{R}\ t'\ I.\ Q \rightleftharpoons \mathit{iRelease}\ I\ (\lambda t.\ P)\ (\lambda t'.\ Q)$

Selected results for temporal operators

The interval conversions given below hold for arbitrary intervals/sets of natural numbers $I \subseteq \mathbb{N}$.

Conversion between basic operators *always* and *eventually*.

lemma $\mathit{iAll\text{-}iEx\text{-}conv}$: $(\Box\ t\ I.\ P\ t) = (\neg\ (\Diamond\ t\ I.\ \neg\ P\ t))$
lemma $\mathit{iEx\text{-}iAll\text{-}conv}$: $(\Diamond\ t\ I.\ P\ t) = (\neg\ (\Box\ t\ I.\ \neg\ P\ t))$

Expressing *eventually* operator through *until* operator analogously to the LTL rule $\Diamond\ \varphi = \mathit{true}\ \mathcal{U}\ \varphi$.

lemma $\mathit{iUntil\text{-}iEx\text{-}conv}$: $(\mathit{True}.\ t'\ \mathcal{U}\ t\ I.\ P\ t) = (\Diamond\ t\ I.\ P\ t)$

Conversions between *until* and *weak until*.

lemma $\mathit{iWeakUntil\text{-}iUntil\text{-}conv}$:
$(P\ t'.\ t'\ \mathcal{W}\ t\ I.\ Q\ t) = ((P\ t'.\ t'\ \mathcal{U}\ t\ I.\ Q\ t) \vee (\Box\ t\ I.\ P\ t))$
lemma $\mathit{iUntil\text{-}iWeakUntil\text{-}conv}$:
$(P\ t'.\ t'\ \mathcal{U}\ t\ I.\ Q\ t) = ((P\ t'.\ t'\ \mathcal{W}\ t\ I.\ Q\ t) \wedge (\Diamond\ t\ I.\ Q\ t))$
lemma $\mathit{iWeakUntil\text{-}conj\text{-}iUntil\text{-}conv}$:
$(P\ t1.\ t1\ \mathcal{W}\ t2\ I.\ (P\ t2 \wedge Q\ t2)) = (\neg\ (\neg\ Q\ t1.\ t1\ \mathcal{U}\ t2\ I.\ \neg\ P\ t2))$

Conversion between *release* and *weak until*.

lemma $\mathit{iRelease\text{-}iWeakUntil\text{-}conv}$: $(P\ t'.\ t'\ \mathcal{R}\ t\ I.\ Q\ t) = (Q\ t'.\ t'\ \mathcal{W}\ t\ I.\ (Q\ t \wedge P\ t))$

Weak and strong *next* operators are dual.

lemma $\mathit{not\text{-}iNextWeak}$: $(\neg\ (\bigcirc_W\ t\ t0\ I.\ P\ t)) = (\bigcirc_S\ t\ t0\ I.\ \neg\ P\ t)$
lemma $\mathit{not\text{-}iNextStrong}$: $(\neg\ (\bigcirc_S\ t\ t0\ I.\ P\ t)) = (\bigcirc_W\ t\ t0\ I.\ \neg\ P\ t)$

Weak and strong *next* operators are equivalent for infinite intervals (provided the evaluation time point is in the interval, which is always true for the interval *{0..}* used in LTL and core SALT semantics definition, cf. functions *ltl-valid* in Sec. 3.2 and *core-salt-valid* in Sec. 4.3.3).

lemma $\mathit{infin\text{-}imp\text{-}iNextStrong\text{-}eq\text{-}iNextWeak}$:
$[\![\ \mathit{infinite}\ I;\ t0 \in I\ ]\!] \Longrightarrow (\bigcirc_S\ t\ t0\ I.\ P\ t) = (\bigcirc_W\ t\ t0\ I.\ P\ t)$

On the interval *{0..}* weak and strong *next* operators are equivalent to adding 1 to the time point of evaluation.

lemma

iNextWeak-atLeast-0: $(\bigcirc_W\ t\ t0\ \{0..\}.\ P\ t) = P\ (Suc\ t0)$ **and**
iNextStrong-atLeast-0: $(\bigcirc_S\ t\ t0\ \{0..\}.\ P\ t) = P\ (Suc\ t0)$

2.2 Regular expressions (deep embedding)

Though ILET is formalised in shallow embedding manner, the regular expressions are first deeply embedded using standalone data types, which allow imposing syntactical restrictions (needed for the repetition operator $*[\leq n]$), and dedicated evaluation functions, which process a regular expression recursively over its structure. The ILET regular expressions can then be translated to the conventional shallow embedding formulas. We use the deep embedding of regular expressions as an intermediate step that especially facilitates working with ILET regular expressions because the shallow embedded ILET formulas giving the semantics of regular expressions do not resemble familiar regular expression notations.

2.2.1 Syntax

Data types for ILET regular expressions.

```
datatype ilet-reg-exp-bool =
    BREAtom² Time ⇒ bool                              ( BREAtom - [115] 115)
  | BRENot   ilet-reg-exp-bool                        ( ¬bre - [40] 40)
  | BREAnd   ilet-reg-exp-bool ilet-reg-exp-bool      ( (- ∧bre -) [36, 35] 35)
  | BREOr    ilet-reg-exp-bool ilet-reg-exp-bool      ( (- ∨bre -) [31, 30] 30)
  | BREImp   ilet-reg-exp-bool ilet-reg-exp-bool      ( (- →bre -) [26, 25] 25)
  | BREEquiv ilet-reg-exp-bool ilet-reg-exp-bool      ( (- ↔bre -) [26, 25] 25)
datatype ilet-reg-exp =
    BREBool          ilet-reg-exp-bool                ( BREBool - [115] 115)
  | BREEmpty                                          ( ε )
  | BRERegExpOr      ilet-reg-exp ilet-reg-exp        ( (- ∨ - ) [30, 31] 30)
  | BRESeqSubsequent ilet-reg-exp ilet-reg-exp        ( (- '';''bre -) [111, 110] 110)
  | BRESeqOverlap    ilet-reg-exp ilet-reg-exp        ( (- '':''bre -) [111, 110] 110)
  | BRERegOp-StarGe  ilet-reg-exp-bool nat            ( (- ''*'' [≥ -]bre) [110, 110] 111)
```

2.2.2 Semantics

Validity function for Boolean terms in ILET regular expressions.

```
consts
  ilet-reg-exp-bool-valid :: Time ⇒ ilet-reg-exp-bool ⇒ bool
  ( (⊨brebool - - ) [80, 80] 80)
primrec
  ⊨brebool t (BREAtom a) = a t
  ⊨brebool t (¬bre f) = (¬ ⊨brebool t f)
  ⊨brebool t (f1 ∧bre f2) = (⊨brebool t f1 ∧ ⊨brebool t f2)
  ⊨brebool t (f1 ∨bre f2) = (⊨brebool t f1 ∨ ⊨brebool t f2)
  ⊨brebool t (f1 →bre f2) = (⊨brebool t f1 ⟶ ⊨brebool t f2)
  ⊨brebool t (f1 ↔bre f2) = (⊨brebool t f1 = ⊨brebool t f2)
```

We now define the evaluation function for ILET regular expressions. Though evaluation of ILET regular expressions is principally possible for all intervals (e.g. for modulo-intervals of the form $\{n \mid n \geq n_0 \ \wedge\ n\ mod\ m = r\}$), we consider for reasons of simplicity only continuous intervals of the form $[n_1 \ldots n_2) = \{n_1, n_1 + 1, \ldots, n_2 - 1\}$. Thus, passing lower and upper bounds of a time interval suffices for matching a regular expression to this interval. $t_2 - t_1$ indicates the length of the regular expression: the expression begins at time point t_1 and ends exactly before time point t_2.

```
consts
  ilet-reg-exp-match :: Time ⇒ Time ⇒ ilet-reg-exp ⇒ bool
  ( (⊨bre - - - ) [80, 80, 80] 80)
```

[2] The abbreviation *BRE* stands for Boolean Regular Expression, the identifier *BREAtom* is required merely for technical syntactical reasons in Isabelle/HOL.

primrec
$\models_{bre}$ *t1 t2* (*BREBool b*) = ($\models_{brebool}$ *t1 b* $\wedge$ *t2* = *Suc t1*)
$\models_{bre}$ *t1 t2* ε = (*t2* = *t1*)
$\models_{bre}$ *t1 t2* (*a* $\vee$ *b*) = ($\models_{bre}$ *t1 t2 a* $\vee$ $\models_{bre}$ *t1 t2 b*)
$\models_{bre}$ *t1 t2* (*b* $'*'$ $[\geq n]_{bre}$) = (($\Box$ *t* {*t1*..<*t2*}. $\models_{brebool}$ *t b*) $\wedge$ *t1* + *n* $\leq$ *t2*)
$\models_{bre}$ *t1 t2* (*a* $';'_{bre}$ *b*) = ($\Diamond$ *t* {*t1*..*t2*}. ($\models_{bre}$ *t1 t a* $\wedge$ $\models_{bre}$ *t t2 b*))
$\models_{bre}$ *t1 t2* (*a* $':'_{bre}$ *b*) = ($\Diamond$ *t* {*t1*..<*t2*}. ($\models_{bre}$ *t1* (*Suc t*) *a* $\wedge$ $\models_{bre}$ *t t2 b*))

For example, $/\ a*\ ;\ b*\ /$ matches "aaabbb.." with $t_1 = 0, t_2 = 6$ as follows:
ilet_reg_exp_match 0 6 /a;b*/* returns true with $t = 3$, because
ilet_reg_exp_match 0 3 /a/* matches "aaa", as s[0]=s[1]=s[2]=a and
ilet_reg_exp_match 3 6 /b/* matches "bbb", as s[3]=s[4]=s[5]=b.

The regular expressions are, similar to derived operators like *until*, directly translatable to basic ILET operators and hence represent convenience constructs. Here, for example, a simple protocol pattern for data transfer, once as regular expression $/\ start; data*[\geq 3]; finish\ /$ and once using basic ILET operators.

lemma *ilet-RegExp1-start-data-finish*:
($\models_{bre}$ *t t'*
(*BREBool* (*BREAtom start*)) $';'_{bre}$
((*BREAtom data*) $'*'$ $[\geq 3]_{bre}$) $';'_{bre}$
(*BREBool* (*BREAtom finish*))) =
($\Diamond$ *t1* {*t*..*t'*}.
start t $\wedge$ *t1* = *t* + *1* $\wedge$
($\Diamond$ *t2* {*t1*..*t'*}.
($\Box$ *t3* {*t1*..<*t2*}. *data t3*) $\wedge$
t1 + *3* $\leq$ *t2* $\wedge$ *finish t2* $\wedge$ *t'* = *t2* + *1*))

2.2.3 Sequence operators and expressions matching empty words

The sequence overlap operator : requires additional considerations for expressions able to match the empty word ε with regard to well-formed ILET and SALT formulas, as explained later in this section and in Section 4.

Results for sequence operators with the empty word ε as left operand.

lemma *bre-reg-exp-overlap-epsilon*:
$\neg$ ($\models_{bre}$ *t1 t2* (ε $':'_{bre}$ *b*))
lemma *bre-reg-exp-subsequent-epsilon*:
($\models_{bre}$ *t1 t2* (ε $';'_{bre}$ *b*)) = ($\models_{bre}$ *t1 t2 b*)

Empty word ε matches any interval of length 0.

definition *ilet-reg-exp-matches-epsilon* :: *ilet-reg-exp* $\Rightarrow$ *bool* **where**
ilet-reg-exp-matches-epsilon r = $\models_{bre}$ *0 0 r*
lemma *ilet-reg-exp-matches-epsilon-any-time*: $\models_{bre}$ *t t r* = *ilet-reg-exp-matches-epsilon r*

All expressions matching ε.

lemma *ilet-reg-exp-matches-epsilon-conv*:
((*r* = ε) $\vee$
($\exists$ *b*. *r* = (*b* $'*'$ $[\geq 0]_{bre}$)) $\vee$
($\exists$ *a b*. (*r* = (*a* $\vee$ *b*) $\wedge$
(*ilet-reg-exp-matches-epsilon a* $\vee$ *ilet-reg-exp-matches-epsilon b*))) $\vee$
($\exists$ *a b*. (*r* = (*a* $';'_{bre}$ *b*) $\wedge$
ilet-reg-exp-matches-epsilon a $\wedge$ *ilet-reg-exp-matches-epsilon b*))) =
(*ilet-reg-exp-matches-epsilon r*)

Function determining regular expressions where there is at least one sequence whose last element matches ε.

fun
ilet-reg-exp-seq-last-matches-epsilon :: *ilet-reg-exp* $\Rightarrow$ *bool*
where

```
  ilet-reg-exp-seq-last-matches-epsilon (a ';'_bre b) =
    ilet-reg-exp-seq-last-matches-epsilon b
| ilet-reg-exp-seq-last-matches-epsilon (a ':'_bre b) =
    ilet-reg-exp-seq-last-matches-epsilon b
| ilet-reg-exp-seq-last-matches-epsilon (a ∨ b) =
    (ilet-reg-exp-seq-last-matches-epsilon a ∨ ilet-reg-exp-seq-last-matches-epsilon b)
| ilet-reg-exp-seq-last-matches-epsilon r = ilet-reg-exp-matches-epsilon r
```

lemma

```
  ilet-reg-exp-seq-last-matches-epsilon--bool:
    ¬ ilet-reg-exp-seq-last-matches-epsilon (BREBool b) and
  ilet-reg-exp-seq-last-matches-epsilon--epsilon:
    ilet-reg-exp-seq-last-matches-epsilon (ε) and
  ilet-reg-exp-seq-last-matches-epsilon--star:
    (ilet-reg-exp-seq-last-matches-epsilon (b '*' [≥ n]_bre)) = (n = 0)
```

Analogue function determining regular expressions where there is at least one sequence whose first element matches ε.

fun

```
  ilet-reg-exp-seq-first-matches-epsilon :: ilet-reg-exp ⇒ bool
```

where

```
  ilet-reg-exp-seq-first-matches-epsilon (a ';'_bre b) =
    ilet-reg-exp-seq-first-matches-epsilon a
| ilet-reg-exp-seq-first-matches-epsilon (a ':'_bre b) =
    ilet-reg-exp-seq-first-matches-epsilon a
| ilet-reg-exp-seq-first-matches-epsilon (a ∨ b) =
    (ilet-reg-exp-seq-first-matches-epsilon a ∨ ilet-reg-exp-seq-first-matches-epsilon b)
| ilet-reg-exp-seq-first-matches-epsilon r = ilet-reg-exp-matches-epsilon r
```

lemma

```
  ilet-reg-exp-seq-first-matches-epsilon--bool:
    ¬ ilet-reg-exp-seq-first-matches-epsilon (BREBool b) and
  ilet-reg-exp-seq-first-matches-epsilon--epsilon:
    ilet-reg-exp-seq-first-matches-epsilon (ε) and
  ilet-reg-exp-seq-first-matches-epsilon--star:
    (ilet-reg-exp-seq-first-matches-epsilon (b '*' [≥ n]_bre)) = (n = 0)
```

Function determining regular expressions, in which there is at least one sequence overlap operator :, for which at least one operand matches ε.

fun

```
  ilet-reg-exp-overlap-with-epsilon :: ilet-reg-exp ⇒ bool
```

where

```
  ilet-reg-exp-overlap-with-epsilon (a ':'_bre b) =
    (ilet-reg-exp-seq-last-matches-epsilon a ∨ ilet-reg-exp-seq-first-matches-epsilon b ∨
     ilet-reg-exp-overlap-with-epsilon a ∨ ilet-reg-exp-overlap-with-epsilon b)
| ilet-reg-exp-overlap-with-epsilon (a ';'_bre b) =
    (ilet-reg-exp-overlap-with-epsilon a ∨ ilet-reg-exp-overlap-with-epsilon b)
| ilet-reg-exp-overlap-with-epsilon (a ∨ b) =
    (ilet-reg-exp-overlap-with-epsilon a ∨ ilet-reg-exp-overlap-with-epsilon b)
| ilet-reg-exp-overlap-with-epsilon r = False
```

Some examples of ILET regular expressions with and without overlaps with empty words:

lemma

```
  let
  a1 = BREBool a1;  a2 = BREBool a2;  a3 = BREBool a3;  a4 = BREBool a4;
  a5 = BREBool a5;  a6 = BREBool a6;  a7 = BREBool a7
  in
  (ilet-reg-exp-overlap-with-epsilon ((a1 ';'_bre a2) ';'_bre (a3 ':'_bre (a4 ';'_bre a5) ';'_bre
    (a6 ':'_bre a7))) = False) ∧
  (ilet-reg-exp-overlap-with-epsilon ((a1 ';'_bre a2) ∨ (a3 ':'_bre (a4 ';'_bre a5) ';'_bre
    (a6 ':'_bre a7))) = False) ∧
  (ilet-reg-exp-overlap-with-epsilon ((a1 ';'_bre a2) ∨ (ε ':'_bre (a4 ';'_bre a5) ';'_bre
```

$(a6\ ':'_{bre}\ a7))) = True) \wedge$
$(ilet\text{-}reg\text{-}exp\text{-}overlap\text{-}with\text{-}epsilon\ ((a1\ ':'_{bre}\ \varepsilon) \vee (a3\ ':'_{bre}\ (a4\ ';'_{bre}\ a5)\ ';'_{bre}$
$(a6\ ':'_{bre}\ a7))) = True) \wedge$
$(ilet\text{-}reg\text{-}exp\text{-}overlap\text{-}with\text{-}epsilon\ ((a1\ ';'_{bre}\ a2) \vee (a3\ ':'_{bre}\ ((b\ '*'\ [\geq 1]_{bre})\ ';'_{bre}\ a5)$
$';'_{bre}\ (a6\ ':'_{bre}\ a7))) = False) \wedge$
$(ilet\text{-}reg\text{-}exp\text{-}overlap\text{-}with\text{-}epsilon\ ((a1\ ';'_{bre}\ a2) \vee (a3\ ':'_{bre}\ ((b\ '*'\ [\geq 0]_{bre})\ ';'_{bre}\ a5)$
$';'_{bre}\ (a6\ ':'_{bre}\ a7))) = True)$

Definition of well-formedness condition w.r.t. proper overlaps: an ILET regular expression is considered well-formed w.r.t. to overlap operator if for every overlap operator both operands cannot match the empty word/interval.

definition $ilet\text{-}reg\text{-}exp\text{-}proper\text{-}overlap :: ilet\text{-}reg\text{-}exp \Rightarrow bool$ **where**
$ilet\text{-}reg\text{-}exp\text{-}proper\text{-}overlap\ r \equiv \neg\ (ilet\text{-}reg\text{-}exp\text{-}overlap\text{-}with\text{-}epsilon\ r)$

The sequence and overlap operators are associative.

lemma $ILETRegExp\text{-}subsequent\text{-}assoc$:
$(\models_{bre}\ t1\ t2\ ((r1\ ';'_{bre}\ r2)\ ';'_{bre}\ r3)) =$
$(\models_{bre}\ t1\ t2\ (r1\ ';'_{bre}\ r2\ ';'_{bre}\ r3))$
lemma $ILETRegExp\text{-}overlap\text{-}assoc$:
$(\models_{bre}\ t1\ t2\ ((r1\ ':'_{bre}\ r2)\ ':'_{bre}\ r3)) =$
$(\models_{bre}\ t1\ t2\ (r1\ ':'_{bre}\ r2\ ':'_{bre}\ r3))$

The sequence and overlap operators are associative with each other only if the middle operand cannot match the empty word.

lemma $ILETRegExp\text{-}subsequent\text{-}overlap\text{-}assoc$:
$\neg\ ilet\text{-}reg\text{-}exp\text{-}matches\text{-}epsilon\ r2 \Longrightarrow$
$(\models_{bre}\ t1\ t2\ ((r1\ ';'_{bre}\ r2)\ ':'_{bre}\ r3)) =$
$(\models_{bre}\ t1\ t2\ (r1\ ';'_{bre}\ r2\ ':'_{bre}\ r3))$
lemma $ILETRegExp\text{-}overlap\text{-}subsequent\text{-}assoc$:
$\neg\ ilet\text{-}reg\text{-}exp\text{-}matches\text{-}epsilon\ r2 \Longrightarrow$
$(\models_{bre}\ t1\ t2\ ((r1\ ':'_{bre}\ r2)\ ';'_{bre}\ r3)) =$
$(\models_{bre}\ t1\ t2\ (r1\ ':'_{bre}\ r2\ ';'_{bre}\ r3))$

It follows as corollaries that sequence and overlap operator are associative with each other on regular expressions with proper overlap operators.

corollary $ILETRegExp\text{-}subsequent\text{-}overlap\text{-}assoc\text{-}proper\text{-}overlap$:
$ilet\text{-}reg\text{-}exp\text{-}proper\text{-}overlap\ (r1\ ';'_{bre}\ r2\ ':'_{bre}\ r3) \Longrightarrow$
$(\models_{bre}\ t1\ t2\ ((r1\ ';'_{bre}\ r2)\ ':'_{bre}\ r3)) =$
$(\models_{bre}\ t1\ t2\ (r1\ ';'_{bre}\ r2\ ':'_{bre}\ r3))$
corollary $ILETRegExp\text{-}overlap\text{-}subsequent\text{-}assoc\text{-}proper\text{-}overlap$:
$ilet\text{-}reg\text{-}exp\text{-}proper\text{-}overlap\ (r1\ ':'_{bre}\ r2\ ';'_{bre}\ r3) \Longrightarrow$
$(\models_{bre}\ t1\ t2\ ((r1\ ':'_{bre}\ r2)\ ';'_{bre}\ r3)) =$
$(\models_{bre}\ t1\ t2\ (r1\ ':'_{bre}\ r2\ ';'_{bre}\ r3))$

If a regular expression matching the empty word neighbours an overlap operator (improper overlap) then different parenthesis of the sequence can result in different formula meaning:

lemma
$ILETRegExp\text{-}subsequent\text{-}overlap\text{-}epsilon\text{-}left$:
$(\models_{bre}\ t1\ t2\ ((r1\ ';'_{bre}\ \varepsilon)\ ':'_{bre}\ r3)) = (\models_{bre}\ t1\ t2\ (r1\ ':'_{bre}\ r3))$ **and**
$ILETRegExp\text{-}subsequent\text{-}overlap\text{-}epsilon\text{-}right$:
$(\models_{bre}\ t1\ t2\ (r1\ ';'_{bre}\ \varepsilon\ ':'_{bre}\ r3)) = False$

Consequently sequence and overlap operator can in general be non-associative with each other:

lemma $NOT\text{-}ILETRegExp\text{-}subsequent\text{-}overlap\text{-}assoc$:
$\neg\ (\forall r1\ r2\ r3\ t1\ t2.$
$(\models_{bre}\ t1\ t2\ ((r1\ ';'_{bre}\ r2)\ ':'_{bre}\ r3)) =$
$(\models_{bre}\ t1\ t2\ (r1\ ';'_{bre}\ r2\ ':'_{bre}\ r3)))$
lemma $NOT\text{-}ILETRegExp\text{-}overlap\text{-}subsequent\text{-}assoc$:
$\neg\ (\forall r1\ r2\ r3\ t1\ t2.$
$(\models_{bre}\ t1\ t2\ ((r1\ ':'_{bre}\ r2)\ ';'_{bre}\ r3)) =$
$(\models_{bre}\ t1\ t2\ (r1\ ':'_{bre}\ r2\ ';'_{bre}\ r3)))$

3 LTL

3.1 Syntax

Syntax of deep embedding of LTL.

Data type for LTL formulas:

```
datatype 'a ltl-formula =
    LTLAtom       'a ⇒ bool                          ( LTLAtom - [60] 60)
  | LTLNot        'a ltl-formula                     ( ¬_ltl - [40] 40)
  | LTLAnd        'a ltl-formula 'a ltl-formula      ( (- ∧_ltl - ) [35, 36] 35)
  | LTLOr         'a ltl-formula 'a ltl-formula      ( (- ∨_ltl - ) [30, 31] 30)
  | LTLImp        'a ltl-formula 'a ltl-formula      ( (- →_ltl - ) [26, 25] 25)
  | LTLEquiv      'a ltl-formula 'a ltl-formula      ( (- ↔_ltl - ) [26, 25] 25)
  | LTLNext       'a ltl-formula                     ( (○_ltl -) [50] 50 )
  | LTLAlways     'a ltl-formula                     ( (□_ltl -) [50] 50 )
  | LTLEventually 'a ltl-formula                     ( (◇_ltl -) [50] 50 )
  | LTLUntil      'a ltl-formula 'a ltl-formula      ( (- U_ltl -) [50, 51] 50 )
  | LTLUntilWeak  'a ltl-formula 'a ltl-formula      ( (- W_ltl -) [50, 51] 50 )
  | LTLRelease    'a ltl-formula 'a ltl-formula      ( (- R_ltl -) [50, 51] 50 )
```

3.2 Semantics

Validity function for LTL formulas – definition through translation to (shallow embedding of) ILET:

```
consts
  ltl-valid :: (Time ⇒ 'a) ⇒ Time ⇒ 'a ltl-formula ⇒ bool
    ( (- ⊨_ltl - -) [80,80] 80)
primrec
  s ⊨_ltl t (LTLAtom a) = a (s t)
  s ⊨_ltl t (¬_ltl f) = (¬(s ⊨_ltl t f))
  s ⊨_ltl t (f1 ∧_ltl f2) = ((s ⊨_ltl t f1) ∧ (s ⊨_ltl t f2))
  s ⊨_ltl t (f1 ∨_ltl f2) = ((s ⊨_ltl t f1) ∨ (s ⊨_ltl t f2))
  s ⊨_ltl t (f1 →_ltl f2) = ((s ⊨_ltl t f1) ⟶ (s ⊨_ltl t f2))
  s ⊨_ltl t (f1 ↔_ltl f2) = ((s ⊨_ltl t f1) = (s ⊨_ltl t f2))
  s ⊨_ltl t (○_ltl f) = (○_S t1 t {0..}. s ⊨_ltl t1 f)
  s ⊨_ltl t (□_ltl f) = (□ t1 {t..}. (s ⊨_ltl t1 f ))
  s ⊨_ltl t (◇_ltl f) = (◇ t1 {t..}. (s ⊨_ltl t1 f ))
  s ⊨_ltl t (f1 U_ltl f2) = ((s ⊨_ltl t1 f1. t1 𝒰 t2 {t..}. s ⊨_ltl t2 f2))
  s ⊨_ltl t (f1 W_ltl f2) = ((s ⊨_ltl t1 f1. t1 𝒲 t2 {t..}. s ⊨_ltl t2 f2))
  s ⊨_ltl t (f1 R_ltl f2) = ((s ⊨_ltl t1 f1. t1 ℛ t2 {t..}. s ⊨_ltl t2 f2))
```

Convenience shortcuts for Boolean constants in LTL formulas:

```
consts
  LTLTrue  :: 'a ltl-formula
  LTLFalse :: 'a ltl-formula
defs
  LTLTrue-def[simp]  : LTLTrue  ≡ LTLAtom (λx. True)
  LTLFalse-def[simp] : LTLFalse ≡ LTLAtom (λx. False)
lemma
  LTLTrue-conv:  (s ⊨_ltl t LTLTrue) = True and
  LTLFalse-conv: (s ⊨_ltl t LTLFalse) = False
```

LTL is often defined on basis of Boolean operators *not*, *and* and temporal operators *until*, *next*. Further Boolean operators *or*, *implies*, *equiv* and temporal operators *eventually*, *always*, *weak until*, *release* can be then defined as abbreviations. The commonly used abbreviations and the explicit semantics definition through translation to ILET are equivalent:

```
lemma
  ltl-disj-equiv: (s ⊨_ltl t (f1 ∨_ltl f2)) = (s ⊨_ltl t ¬_ltl ((¬_ltl f1) ∧_ltl ¬_ltl f2)) and
```

ltl-imp-equiv: $(s \models_{ltl} t\ (f1 \rightarrow_{ltl} f2)) = (s \models_{ltl} t\ ((\neg_{ltl}\ f1) \vee_{ltl} f2))$ **and**
ltl-equiv-equiv: $(s \models_{ltl} t\ (f1 \leftrightarrow_{ltl} f2)) = (s \models_{ltl} t\ ((f1 \rightarrow_{ltl} f2) \wedge_{ltl} (f2 \rightarrow_{ltl} f1)))$

lemma
ltl-eventually-equiv: $(s \models_{ltl} t\ (\Diamond_{ltl}\ f)) = (s \models_{ltl} t\ (LTLTrue\ U_{ltl}\ f))$ **and**
ltl-always-equiv: $(s \models_{ltl} t\ (\Box_{ltl}\ f)) = (s \models_{ltl} t\ (\neg_{ltl}\ \Diamond_{ltl}\ (\neg_{ltl}\ f)))$

lemma *ltl-untilweak-equiv*: $(s \models_{ltl} t\ (f1\ W_{ltl}\ f2)) = (s \models_{ltl} t\ ((f1\ U_{ltl}\ f2) \vee_{ltl} \Box_{ltl}\ f1))$

lemma *ltl-release-equiv*: $(s \models_{ltl} t\ (f1\ R_{ltl}\ f2)) = (s \models_{ltl} t\ (f2\ W_{ltl}\ (f2 \wedge_{ltl} f1)))$

4 Core SALT

We consider following core SALT language constructs:

- Boolean operators ***not***, ***and***, ***or***, ***implies***, ***equals***.
- Common temporal operators ***next***, ***always***, ***eventually***.
- Extended ***until*** operator capable of encoding LTL operators until, until weak, release.
- ***from*** operator.
- Restricted regular expressions
  - Boolean operators on propositions.
  - Disjunction on regular expressions.
  - Repetition operator $*[\geq n]$ with $n \in \mathbb{N}$ for propositional expressions.
  - Operators ; and : expressing successive and overlapping sequences, respectively.

Few core SALT constructs are not treated here and are considered part of future work:

- Scope operators using the SALT-- stop operators (e.g., ***upto***).
- Exception operators ***accepton***, ***rejecton***.

As the SALT-- translation step [Str06, Section 6.2] is only needed for translation of the omitted operators, we do not have to consider it and can translate core SALT directly to LTL.

4.1 Syntax

Syntax of deep embedding of core SALT.

Data types for parameters of some core SALT operators.

```
datatype SALT-req-opt-weak =
  req  ( req )
| opt  ( opt )
| weak ( weak )
datatype SALT-req-opt =
  req2  ( req )
| opt2  ( opt )
datatype SALT-excl-incl =
  excl ( excl )
| incl ( incl )
```

Data types for core SALT regular expressions:

```
datatype 'a core-salt-reg-exp-bool =
    CoreSREAtom    'a ⇒ bool                    ( CoreSREAtom - [115] 115)
  | CoreSRENot     'a core-salt-reg-exp-bool     ( not - [40] 40)
  | CoreSREAnd     'a core-salt-reg-exp-bool 'a core-salt-reg-exp-bool
    ( (- and -) [36, 35] 35)
  | CoreSREOr      'a core-salt-reg-exp-bool 'a core-salt-reg-exp-bool
    ( (- or -) [31, 30] 30)
  | CoreSREImp     'a core-salt-reg-exp-bool 'a core-salt-reg-exp-bool
    ( (- implies -) [26, 25] 25)
```

| *CoreSREEquiv* *'a core-salt-reg-exp-bool* *'a core-salt-reg-exp-bool*
((- **equals** -) [26, 25] 25)

datatype
'a core-salt-reg-exp =
CoreSREBool *'a core-salt-reg-exp-bool* (*CoreSREBool* - [115] 115)
| *CoreSREEmpty* (ε)
| *CoreSRERegExpOr* *'a core-salt-reg-exp* *'a core-salt-reg-exp*
((- **or** -) [30, 31] 30)
| *CoreSRESeqSubsequent* *'a core-salt-reg-exp* *'a core-salt-reg-exp*
((- '';'' -) [111, 110] 110)
| *CoreSRESeqOverlap* *'a core-salt-reg-exp* *'a core-salt-reg-exp*
((- '':'' -) [111, 110] 110)
| *CoreSRERegOp-StarGe* *'a core-salt-reg-exp-bool nat*
((- ''*'' [$\geq$ -]$_{coresre}$) [110, 110] 111)

Data type for core SALT formulas:

datatype
'a core-salt-formula =
CoreSALTAtom *'a* $\Rightarrow$ *bool* (*CoreSALTAtom* - [115] 115)
| *CoreSALTNot* *'a core-salt-formula* (**not**- [40] 40)
| *CoreSALTAnd* *'a core-salt-formula* *'a core-salt-formula*
((- **and** -) [36, 35] 35)
| *CoreSALTOr* *'a core-salt-formula* *'a core-salt-formula*
((- **or** -) [31, 30] 30)
| *CoreSALTImp* *'a core-salt-formula* *'a core-salt-formula*
((- **implies** -) [26, 25] 25)
| *CoreSALTEquiv* *'a core-salt-formula* *'a core-salt-formula*
((- **equals** -) [26, 25] 25)
| *CoreSALTNext* *'a core-salt-formula* ((**next** -) [50] 50)
| *CoreSALTAlways* *'a core-salt-formula* ((**always** -) [50] 50)
| *CoreSALTEventually* *'a core-salt-formula* ((**eventually** -) [50] 50)
| *CoreSALTUntilExt* *'a core-salt-formula SALT-excl-incl SALT-req-opt-weak*
'a core-salt-formula ((- **until** - - -) [50, 50, 50, 51] 50)
| *CoreSALTFrom* *'a core-salt-formula SALT-excl-incl SALT-req-opt*
'a $\Rightarrow$ *bool* ((- **from** - - -) [50, 50, 50, 51] 50)
| *CoreSALTRegExp* *'a core-salt-reg-exp*
((''|'' - ''|''$_{coresre}$) [50] 50)
| *CoreSALTRegExpSeqSaltFinish* *'a core-salt-reg-exp* *'a core-salt-formula*
((''|'' - '';''$_{end}$ - ''|''$_{coresre}$) [51,50] 50)

4.2 Translation to LTL

Translation of core SALT to LTL according to [Str06, Section 6].

4.2.1 Translation of regular expressions to LTL

Translating Boolean regular expressions to LTL.

consts
core-salt-reg-exp-bool-to-ltl :: *'a core-salt-reg-exp-bool* $\Rightarrow$ *'a ltl-formula*

primrec
core-salt-reg-exp-bool-to-ltl (*CoreSREAtom a*) = *LTLAtom a*
core-salt-reg-exp-bool-to-ltl (**not** *f*) =
($\neg_{ltl}$ *core-salt-reg-exp-bool-to-ltl f*)
core-salt-reg-exp-bool-to-ltl (*f1* **and** *f2*) =
(*core-salt-reg-exp-bool-to-ltl f1* $\wedge_{ltl}$ *core-salt-reg-exp-bool-to-ltl f2*)
core-salt-reg-exp-bool-to-ltl (*f1* **or** *f2*) =
(*core-salt-reg-exp-bool-to-ltl f1* $\vee_{ltl}$ *core-salt-reg-exp-bool-to-ltl f2*)
core-salt-reg-exp-bool-to-ltl (*f1* **implies** *f2*) =

$(core\text{-}salt\text{-}reg\text{-}exp\text{-}bool\text{-}to\text{-}ltl\ f1 \rightarrow_{ltl} core\text{-}salt\text{-}reg\text{-}exp\text{-}bool\text{-}to\text{-}ltl\ f2)$
$core\text{-}salt\text{-}reg\text{-}exp\text{-}bool\text{-}to\text{-}ltl\ (f1\ \textbf{equals}\ f2) =$
$(core\text{-}salt\text{-}reg\text{-}exp\text{-}bool\text{-}to\text{-}ltl\ f1 \leftrightarrow_{ltl} core\text{-}salt\text{-}reg\text{-}exp\text{-}bool\text{-}to\text{-}ltl\ f2)$

Function for constructing an LTL formula consisting of n subsequent *next* operators applied to a parameter LTL formula *f*. The resulting formula states that *f* holds n steps after current time.

consts
$nextn\text{-}ltl :: nat \Rightarrow {}'a\ ltl\text{-}formula \Rightarrow {}'a\ ltl\text{-}formula$
$(\ (\bigcirc_{ltl}{}^{[\text{-}]}\ \text{-})\ [50,50]\ 50\ )$
primrec
$(\bigcirc_{ltl}{}^{[0]}\ f) = f$
$(\bigcirc_{ltl}{}^{[Suc\ n]}\ f) = (\bigcirc_{ltl}\ (\bigcirc_{ltl}{}^{[n]}\ f))$

Function expressing a bounded *always* operator in LTL. It constructs an LTL formula stating that *f* holds for n steps (in an interval $[t\ \dots\ t+n)$ when evaluated at time t).

consts
$alwaysn\text{-}ltl :: nat \Rightarrow {}'a\ ltl\text{-}formula \Rightarrow {}'a\ ltl\text{-}formula$
$(\ (\Box_{ltl}{}^{[\text{-}]}\ \text{-})\ [50,50]\ 50\ )$
primrec
$(\Box_{ltl}{}^{[0]}\ f) = LTLTrue$
$(\Box_{ltl}{}^{[Suc\ n]}\ f) = (f \wedge_{ltl} \bigcirc_{ltl}\ (\Box_{ltl}{}^{[n]}\ f))$

Properties of *nextn-ltl* and *alwaysn-ltl*.

lemma $nextn\text{-}ltl\text{-}conv$: $\bigwedge t.\ (s \models_{ltl} t\ (\bigcirc_{ltl}{}^{[n]}\ f)) = (s \models_{ltl} (t + n)\ f)$
lemma $alwaysn\text{-}ltl\text{-}conv$: $\bigwedge t.\ (s \models_{ltl} t\ (\Box_{ltl}{}^{[n]}\ f)) = (\Box\ t'\ \{t..<t + n\}.\ (s \models_{ltl} t'\ f))$

Translating sequence operators to LTL (mutually recursive function definitions):

fun
$sre\text{-}subsequent\text{-}to\text{-}ltl :: {}'a\ core\text{-}salt\text{-}reg\text{-}exp \Rightarrow {}'a\ ltl\text{-}formula \Rightarrow {}'a\ ltl\text{-}formula$ **and**
$sre\text{-}overlap\text{-}to\text{-}ltl :: {}'a\ core\text{-}salt\text{-}reg\text{-}exp \Rightarrow {}'a\ ltl\text{-}formula \Rightarrow {}'a\ ltl\text{-}formula$
where
$(sre\text{-}subsequent\text{-}to\text{-}ltl\ (CoreSREBool\ b)\ f) =$
$((core\text{-}salt\text{-}reg\text{-}exp\text{-}bool\text{-}to\text{-}ltl\ b) \wedge_{ltl} (\bigcirc_{ltl}\ f))$
$|\ (sre\text{-}subsequent\text{-}to\text{-}ltl\ \varepsilon\ f) = f$
$|\ (sre\text{-}subsequent\text{-}to\text{-}ltl\ (a\ \textbf{or}\ b)\ f) =$
$((sre\text{-}subsequent\text{-}to\text{-}ltl\ a\ f) \vee_{ltl} (sre\text{-}subsequent\text{-}to\text{-}ltl\ b\ f))$
$|\ sre\text{-}subsequent\text{-}to\text{-}ltl\text{-}simp\text{-}subseq$:
$(sre\text{-}subsequent\text{-}to\text{-}ltl\ (a\ {}';{}'\ b)\ f) =$
$(sre\text{-}subsequent\text{-}to\text{-}ltl\ a\ (sre\text{-}subsequent\text{-}to\text{-}ltl\ b\ f))$
$|\ sre\text{-}overlap\text{-}subsequent\text{-}to\text{-}ltl\text{-}simp\text{-}subseq$:
$(sre\text{-}subsequent\text{-}to\text{-}ltl\ (a\ {}':{}'\ b)\ f) =$
$(sre\text{-}overlap\text{-}to\text{-}ltl\ a\ (sre\text{-}subsequent\text{-}to\text{-}ltl\ b\ f))$
$|\ (sre\text{-}subsequent\text{-}to\text{-}ltl\ (b\ {}'{*}{}'\ [\geq n]_{coresre})\ f) =$
$((core\text{-}salt\text{-}reg\text{-}exp\text{-}bool\text{-}to\text{-}ltl\ b)\ \mathcal{U}_{ltl}$
$((\Box_{ltl}{}^{[n]}\ (core\text{-}salt\text{-}reg\text{-}exp\text{-}bool\text{-}to\text{-}ltl\ b)) \wedge_{ltl} (\bigcirc_{ltl}{}^{[n]}\ f)))$

$|\ (sre\text{-}overlap\text{-}to\text{-}ltl\ (CoreSREBool\ b)\ f) = ((core\text{-}salt\text{-}reg\text{-}exp\text{-}bool\text{-}to\text{-}ltl\ b) \wedge_{ltl} f)$
$|\ (sre\text{-}overlap\text{-}to\text{-}ltl\ \varepsilon\ f) = f$
$|\ (sre\text{-}overlap\text{-}to\text{-}ltl\ (a\ \textbf{or}\ b)\ f) =$
$((sre\text{-}overlap\text{-}to\text{-}ltl\ a\ f) \vee_{ltl} (sre\text{-}overlap\text{-}to\text{-}ltl\ b\ f))$
$|\ sre\text{-}subsequent\text{-}overlap\text{-}to\text{-}ltl\text{-}simp\text{-}subseq$:
$(sre\text{-}overlap\text{-}to\text{-}ltl\ (a\ {}';{}'\ b)\ f) =$
$(sre\text{-}subsequent\text{-}to\text{-}ltl\ a\ (sre\text{-}overlap\text{-}to\text{-}ltl\ b\ f))$
$|\ sre\text{-}overlap\text{-}to\text{-}ltl\text{-}simp\text{-}subseq$:
$(sre\text{-}overlap\text{-}to\text{-}ltl\ (a\ {}':{}'\ b)\ f) =$
$(sre\text{-}overlap\text{-}to\text{-}ltl\ a\ (sre\text{-}overlap\text{-}to\text{-}ltl\ b\ f))$
$|\ (sre\text{-}overlap\text{-}to\text{-}ltl\ (b\ {}'{*}{}'\ [\geq n]_{coresre})\ f) =$
$(if\ n = 0\ then$

$(f \vee_{ltl}$
$((\mathit{core\text{-}salt\text{-}reg\text{-}exp\text{-}bool\text{-}to\text{-}ltl}\ b)\ U_{ltl}\ ((\mathit{core\text{-}salt\text{-}reg\text{-}exp\text{-}bool\text{-}to\text{-}ltl}\ b)\ \wedge_{ltl}\ f)))$
else
$((\mathit{core\text{-}salt\text{-}reg\text{-}exp\text{-}bool\text{-}to\text{-}ltl}\ b)\ U_{ltl}$
$((\Box_{ltl}{}^{[n]}\ (\mathit{core\text{-}salt\text{-}reg\text{-}exp\text{-}bool\text{-}to\text{-}ltl}\ b))\ \wedge_{ltl}\ (\bigcirc_{ltl}{}^{[n\,-\,1]}\ f))))$

Translating all core SALT regular operators:

fun
$\mathit{sre\text{-}core\text{-}to\text{-}ltl} :: {}'a\ \mathit{core\text{-}salt\text{-}reg\text{-}exp} \Rightarrow {}'a\ \mathit{ltl\text{-}formula}$
where
$(\mathit{sre\text{-}core\text{-}to\text{-}ltl}\ (\mathit{CoreSREBool}\ b)) = (\mathit{core\text{-}salt\text{-}reg\text{-}exp\text{-}bool\text{-}to\text{-}ltl}\ b)$
$|\ (\mathit{sre\text{-}core\text{-}to\text{-}ltl}\ \varepsilon) = (\mathit{LTLTrue})$
$|\ (\mathit{sre\text{-}core\text{-}to\text{-}ltl}\ (a\ \mathbf{or}\ b)) = ((\mathit{sre\text{-}core\text{-}to\text{-}ltl}\ a) \vee_{ltl} (\mathit{sre\text{-}core\text{-}to\text{-}ltl}\ b))$
$|\ \mathit{sre\text{-}core\text{-}to\text{-}ltl\text{-}simp\text{-}subseq}$:
$(\mathit{sre\text{-}core\text{-}to\text{-}ltl}\ (a\ ';'\ b)) = (\mathit{sre\text{-}subsequent\text{-}to\text{-}ltl}\ a\ (\mathit{sre\text{-}core\text{-}to\text{-}ltl}\ b))$
$|\ \mathit{sre\text{-}core\text{-}to\text{-}ltl\text{-}simp\text{-}overlap}$:
$(\mathit{sre\text{-}core\text{-}to\text{-}ltl}\ (a\ ':'\ b)) = (\mathit{sre\text{-}overlap\text{-}to\text{-}ltl}\ a\ (\mathit{sre\text{-}core\text{-}to\text{-}ltl}\ b))$
$|\ (\mathit{sre\text{-}core\text{-}to\text{-}ltl}\ (b\ '{*}'\ [\geq\ n]_{coresre})) = (\Box_{ltl}{}^{[n]}\ (\mathit{core\text{-}salt\text{-}reg\text{-}exp\text{-}bool\text{-}to\text{-}ltl}\ b))$

Core SALT sequence operators are associative, not only w.r.t. to the semantical equivalence of the LTL formulas resulting from the translation but even syntactically, i.e., the resulting LTL formulas are syntactically equal.

lemma
$\mathit{sre\text{-}core\text{-}to\text{-}ltl\text{-}subsequent\text{-}assoc}$:
$\mathit{sre\text{-}core\text{-}to\text{-}ltl}\ ((a\ ';'\ b)\ ';'\ c) = \mathit{sre\text{-}core\text{-}to\text{-}ltl}\ (a\ ';'\ b\ ';'\ c)$ **and**
$\mathit{sre\text{-}core\text{-}to\text{-}ltl\text{-}overlap\text{-}assoc}$:
$\mathit{sre\text{-}core\text{-}to\text{-}ltl}\ ((a\ ':'\ b)\ ':'\ c) = \mathit{sre\text{-}core\text{-}to\text{-}ltl}\ (a\ ':'\ b\ ':'\ c)$ **and**
$\mathit{sre\text{-}core\text{-}to\text{-}ltl\text{-}subsequent\text{-}overlap\text{-}assoc}$:
$\mathit{sre\text{-}core\text{-}to\text{-}ltl}\ ((a\ ';'\ b)\ ':'\ c) = \mathit{sre\text{-}core\text{-}to\text{-}ltl}\ (a\ ';'\ b\ ':'\ c)$ **and**
$\mathit{sre\text{-}core\text{-}to\text{-}ltl\text{-}overlap\text{-}subsequent\text{-}assoc}$:
$\mathit{sre\text{-}core\text{-}to\text{-}ltl}\ ((a\ ':'\ b)\ ';'\ c) = \mathit{sre\text{-}core\text{-}to\text{-}ltl}\ (a\ ':'\ b\ ';'\ c)$

Contrary to ILET, core SALT sequence operators are associative without well-formedness preconditions. The reason is that due to translation definition all sequences are considered right-associative independently of the actual parenthesis. Consider the translation of the sequences $(a;\varepsilon) : c$ and $a;(\varepsilon : c)$, which are both translated according to the right-associative interpretation $a;\varepsilon : c = a;c$ where the empty word ε is "consumed" by c (which corresponds to the LTL formula $a\ \wedge\ \circ\ c$ if a and c are Boolean expressions).

lemma
$\mathit{sre\text{-}core\text{-}subsequent\text{-}overlap\text{-}epsilon\text{-}left}$:
$\mathit{sre\text{-}core\text{-}to\text{-}ltl}\ ((a\ ';'\ \varepsilon)\ ':'\ c) = \mathit{sre\text{-}core\text{-}to\text{-}ltl}\ (a\ ';'\ c)$ **and**
$\mathit{sre\text{-}core\text{-}subsequent\text{-}overlap\text{-}epsilon\text{-}right}$:
$\mathit{sre\text{-}core\text{-}to\text{-}ltl}\ (a\ ';'\ \varepsilon\ ':'\ c) = \mathit{sre\text{-}core\text{-}to\text{-}ltl}\ (a\ ';'\ c)$

Obviously we cannot provide a sound semantics for all formulas if the translation syntactically forces the sequence operators to be right associative and at the same time the semantics of the expressions $(a;\varepsilon) : c = a : c$ and $a;(\varepsilon : c) = a;c$ are different (the interpretation $\varepsilon : c = c$ corresponds to the description in [Str06, p. 42]; in ILET the semantics of $\varepsilon : c$ is False, cf. lemma *ILETRegExp-subsequent-overlap-epsilon-right* in Section 2.2.3).

Hence, for proving the correctness of the translation of core SALT to LTL we will have to restrict the set of well-formed core SALT regular expressions by the condition that an expression matching the empty word ε may not neighbour the overlap operator : (cf. Section 4.4).

4.2.2 Translation of *until* and *from* operators to LTL

Translating the extended *until* operator to LTL.

consts
$\mathit{ltl\text{-}untilext} :: \mathit{SALT\text{-}excl\text{-}incl} \Rightarrow \mathit{SALT\text{-}req\text{-}opt\text{-}weak} \Rightarrow {}'a\ \mathit{ltl\text{-}formula} \Rightarrow$
${}'a\ \mathit{ltl\text{-}formula} \Rightarrow {}'a\ \mathit{ltl\text{-}formula}$

ltl-untilext-exclincl :: *SALT-excl-incl* $\Rightarrow$ *'a ltl-formula* $\Rightarrow$
'a ltl-formula $\Rightarrow$ *'a ltl-formula*

primrec
ltl-untilext-exclincl ***excl*** *f1 f2* = *f2*
ltl-untilext-exclincl ***incl*** *f1 f2* = (*f1* $\wedge_{ltl}$ *f2*)

primrec
ltl-untilext exclincl ***req*** *f1 f2* =
(*f1* U_{ltl} (*ltl-untilext-exclincl exclincl f1 f2*))
ltl-untilext exclincl ***opt*** *f1 f2* =
(($\Diamond_{ltl}$ *f2*) $\rightarrow_{ltl}$ (*f1* U_{ltl} (*ltl-untilext-exclincl exclincl f1 f2*)))
ltl-untilext exclincl ***weak*** *f1 f2* =
(*f1* W_{ltl} (*ltl-untilext-exclincl exclincl f1 f2*))

The translation function for the extended *until* operator returns exactly the LTL formulas given in the SALT language reference [Str06, p. 40].

lemma
ltl-untilext-excl-req: *ltl-untilext* ***excl req*** *f1 f2* = (*f1* U_{ltl} *f2*) **and**
ltl-untilext-excl-opt: *ltl-untilext* ***excl opt*** *f1 f2* = ($\Diamond_{ltl}$ *f2* $\rightarrow_{ltl}$ (*f1* U_{ltl} *f2*)) **and**
ltl-untilext-excl-weak: *ltl-untilext* ***excl weak*** *f1 f2* = (*f1* W_{ltl} *f2*) **and**
ltl-untilext-incl-req: *ltl-untilext* ***incl req*** *f1 f2* = (*f1* U_{ltl} (*f1* $\wedge_{ltl}$ *f2*)) **and**
ltl-untilext-incl-opt: *ltl-untilext* ***incl opt*** *f1 f2* =
($\Diamond_{ltl}$ *f2* $\rightarrow_{ltl}$ (*f1* U_{ltl} (*f1* $\wedge_{ltl}$ *f2*))) **and**
ltl-untilext-incl-weak: *ltl-untilext* ***incl weak*** *f1 f2* = (*f1* W_{ltl} (*f1* $\wedge_{ltl}$ *f2*))

Translating the *from* operator to LTL.

consts
ltl-from-exclincl :: *SALT-excl-incl* $\Rightarrow$ *'a ltl-formula* $\Rightarrow$ *'a ltl-formula*

primrec
ltl-from-exclincl ***incl*** *f* = *f*
ltl-from-exclincl ***excl*** *f* = ($\bigcirc_{ltl}$ *f*)

constdefs
ltl-from :: *SALT-excl-incl* $\Rightarrow$ *SALT-req-opt* $\Rightarrow$ *'a ltl-formula* $\Rightarrow$
(*'a* $\Rightarrow$ *bool*) $\Rightarrow$ *'a ltl-formula*
ltl-from exclincl reqopt f a $\equiv$
(*case reqopt of* ***req*** $\Rightarrow$ *LTLUntil* | ***opt*** $\Rightarrow$ *LTLUntilWeak*)
($\neg_{ltl}$ *LTLAtom a*)
(*LTLAtom a* $\wedge_{ltl}$ (*ltl-from-exclincl exclincl f*))

The translation function for the *from* operator returns exactly the LTL formulas given in the SALT language reference [Str06, p. 42].

lemma
ltl-from-excl-req: *ltl-from* ***excl req*** *f a* =
(($\neg_{ltl}$ *LTLAtom a*) U_{ltl} (*LTLAtom a* $\wedge_{ltl}$ $\bigcirc_{ltl}$ *f*)) **and**
ltl-from-excl-opt: *ltl-from* ***excl opt*** *f a* =
(($\neg_{ltl}$ *LTLAtom a*) W_{ltl} (*LTLAtom a* $\wedge_{ltl}$ $\bigcirc_{ltl}$ *f*)) **and**
ltl-from-incl-req: *ltl-from* ***incl req*** *f a* =
(($\neg_{ltl}$ *LTLAtom a*) U_{ltl} (*LTLAtom a* $\wedge_{ltl}$ *f*)) **and**
ltl-from-incl-opt: *ltl-from* ***incl opt*** *f a* =
(($\neg_{ltl}$ *LTLAtom a*) W_{ltl} (*LTLAtom a* $\wedge_{ltl}$ *f*))

4.2.3 Translation of core SALT formulas to LTL

Main function for translation of core SALT to LTL.

consts
core-salt-to-ltl :: *'a core-salt-formula* $\Rightarrow$ *'a ltl-formula*

primrec
core-salt-to-ltl (*CoreSALTAtom a*) = *LTLAtom a*
core-salt-to-ltl (***not*** *f*) = ($\neg_{ltl}$ *core-salt-to-ltl f*)
core-salt-to-ltl (*f1* ***and*** *f2*) = (*core-salt-to-ltl f1* $\wedge_{ltl}$ *core-salt-to-ltl f2*)

core-salt-to-ltl (f1 **or** f2) = (core-salt-to-ltl f1 $\vee_{ltl}$ core-salt-to-ltl f2)
core-salt-to-ltl (f1 **implies** f2) = (core-salt-to-ltl f1 $\rightarrow_{ltl}$ core-salt-to-ltl f2)
core-salt-to-ltl (f1 **equals** f2) = (core-salt-to-ltl f1 $\leftrightarrow_{ltl}$ core-salt-to-ltl f2)
core-salt-to-ltl (**next** f) = ($\bigcirc_{ltl}$ core-salt-to-ltl f)
core-salt-to-ltl (**always** f) = ($\Box_{ltl}$ core-salt-to-ltl f)
core-salt-to-ltl (**eventually** f) = ($\Diamond_{ltl}$ core-salt-to-ltl f)
core-salt-to-ltl (f1 **until** exclincl reqoptweak f2) =
  (ltl-untilext exclincl reqoptweak (core-salt-to-ltl f1) (core-salt-to-ltl f2))
core-salt-to-ltl (f **from** exclincl reqopt a) =
  (ltl-from exclincl reqopt (core-salt-to-ltl f) a)
core-salt-to-ltl ('|' r '|'$_{coresre}$) = (sre-core-to-ltl r)
core-salt-to-ltl ('|' r ';'$_{end}$ f '|'$_{coresre}$) =
  (sre-subsequent-to-ltl r (core-salt-to-ltl f))
core-salt-to-ltl ('|' r ':'$_{end}$ f '|'$_{coresre}$) =
  (sre-overlap-to-ltl r (core-salt-to-ltl f))

Below we define auxiliary functions for showing the equivalence of the translation definition used here and the translation definition in the SALT language reference [Str06, p. 42] for the regular operators star $*$, sequence ;, and overlap ':'[3].

Function for constructing a sequence of $n+1$ repetitions of a regular expression.

consts
  subsequentn-coresre :: nat $\Rightarrow$ 'a core-salt-reg-exp $\Rightarrow$ 'a core-salt-reg-exp
primrec
  subsequentn-coresre 0 r = r
  subsequentn-coresre (Suc n) r = (r ';' subsequentn-coresre n r)

Function for constructing a SALT formula, which is a regular expression containing n repetitions of a Boolean expression (empty word for $n = 0$).

constdefs
  subsequentn-core-salt :: nat $\Rightarrow$ 'a core-salt-reg-exp-bool $\Rightarrow$ 'a core-salt-formula
  subsequentn-core-salt n b $\equiv$ '|' (case n of
    0 $\Rightarrow$ ε | Suc n' $\Rightarrow$ subsequentn-coresre n' (CoreSREBool b)) '|'$_{coresre}$

Function for constructing a SALT formula, which is a regular expression containing n repetitions of a Boolean expression, followed by a further core SALT formula.

constdefs
  subsequentn-tail-core-salt :: nat $\Rightarrow$ 'a core-salt-reg-exp-bool $\Rightarrow$
    'a core-salt-formula $\Rightarrow$ 'a core-salt-formula
  subsequentn-tail-core-salt n b f $\equiv$ (case n of
    0 $\Rightarrow$ f | Suc n' $\Rightarrow$ '|' subsequentn-coresre n' (CoreSREBool b) ';'$_{end}$ f '|'$_{coresre}$)

Function for constructing a SALT formula, which is a regular expression containing n repetitions of a Boolean expression, followed by an overlapping core SALT formula.

constdefs
  subsequentn-tail-overlap-core-salt :: nat $\Rightarrow$ 'a core-salt-reg-exp-bool $\Rightarrow$
    'a core-salt-formula $\Rightarrow$ 'a core-salt-formula
  subsequentn-tail-overlap-core-salt n b f $\equiv$ (case n of
    0 $\Rightarrow$ f | Suc n' $\Rightarrow$ '|' subsequentn-coresre n' (CoreSREBool b) ':'$_{end}$ f '|'$_{coresre}$)

Some examples of generating regular expressions representing *n* subsequent repetitions of a given regular expression *r* or a Boolean regular expression *b*, possibly followed by a core SALT formula *f*.

lemma subsequentn-coresre-3:
  subsequentn-coresre 3 r = r ';' r ';' r ';' r
lemma subsequentn-core-salt-0:
  subsequentn-core-salt 0 b = ('|' ε '|'$_{coresre}$)
lemma subsequentn-core-salt-3: let r = CoreSREBool b in
  subsequentn-core-salt 3 b = ('|' r ';' r ';' r '|'$_{coresre}$)
lemma subsequentn-tail-core-salt-0:

[3]The operator : is written in apostrophes solely to distinguish it from punctuation marks.

$subsequentn\text{-}tail\text{-}core\text{-}salt\ 0\ b\ f = f$

lemma $subsequentn\text{-}tail\text{-}core\text{-}salt\text{-}3$: $let\ r = CoreSREBool\ b\ in$

$subsequentn\text{-}tail\text{-}core\text{-}salt\ 3\ b\ f = ('|'\ r\ ';'\ r\ ';'\ r\ ';'_{end}\ f\ '|'_{coresre})$

For translating the star operator $*[\geq n]$ to LTL we use the function alwaysn-ltl (syntax $\Box_{ltl}{}^{[n]}$) (cf. *sre-core-to-ltl*). Here the equivalence of this definition and the definition in the SALT language reference [Str06, p. 42] is shown.

lemma $core\text{-}salt\text{-}StarGe\text{-}equiv\text{-}alwaysn\text{-}ltl$: $\bigwedge t.$

$s \models_{ltl} t\ (core\text{-}salt\text{-}to\text{-}ltl\ ('|'\ b\ '*'\ [\geq n]_{coresre}\ '|'_{coresre})) =$

$s \models_{ltl} t\ (core\text{-}salt\text{-}to\text{-}ltl\ (subsequentn\text{-}core\text{-}salt\ n\ b))$

For translating the star operator $*[\geq n]$ with the sequence operator ; to LTL we use the functions *alwaysn-ltl* and *nextn-ltl* (syntax $\bigcirc_{ltl}{}^{[-]}$) (cf. *sre-subsequent-to-ltl*). Here the equivalence of this definition and the definition in the SALT language reference [Str06, p. 42] is shown.

lemma $core\text{-}salt\text{-}StarGe\text{-}Subsequent\text{-}equiv\text{-}alwaysn\text{-}ltl$: $\bigwedge t.$

$s \models_{ltl} t\ (core\text{-}salt\text{-}to\text{-}ltl\ ('|'\ b\ '*'\ [\geq n]_{coresre}\ ';'_{end}\ f\ '|'_{coresre})) =$

$s \models_{ltl} t\ (core\text{-}salt\text{-}reg\text{-}exp\text{-}bool\text{-}to\text{-}ltl\ b)\ \mathcal{U}_{ltl}$

$(core\text{-}salt\text{-}to\text{-}ltl\ (subsequentn\text{-}tail\text{-}core\text{-}salt\ n\ b\ f))$

Finally, the analogue equivalence of the translation definition of the star operator $*[\geq n]$ with the overlap operator : to LTL and the definition in the SALT language reference [Str06, p. 42] is shown.

lemma $core\text{-}salt\text{-}StarGe\text{-}Overlap\text{-}equiv\text{-}alwaysn\text{-}ltl$: $\bigwedge t.$

$0 < n \Longrightarrow$

$s \models_{ltl} t\ (core\text{-}salt\text{-}to\text{-}ltl\ ('|'\ b\ '*'\ [\geq n]_{coresre}\ ':'_{end}\ f\ '|'_{coresre})) =$

$s \models_{ltl} t\ (core\text{-}salt\text{-}reg\text{-}exp\text{-}bool\text{-}to\text{-}ltl\ b)\ \mathcal{U}_{ltl}$

$(core\text{-}salt\text{-}to\text{-}ltl\ (subsequentn\text{-}tail\text{-}overlap\text{-}core\text{-}salt\ n\ b\ f))$

4.3 Semantics

Definition of core SALT semantics by translation of core SALT formulas to ILET. A formula is first translated to an ILET formula, which can contain ILET regular expressions if the core SALT formula contains regular expressions. In the final step the ILET regular expressions are translated to ILET – the resulting formula gives the formal semantics of the core SALT formula.

4.3.1 Translation of regular expressions to ILET

Translating Boolean regular expressions to ILET.

consts

$core\text{-}salt\text{-}reg\text{-}exp\text{-}bool\text{-}to\text{-}ilet$ ::

$(Time \Rightarrow 'a) \Rightarrow 'a\ core\text{-}salt\text{-}reg\text{-}exp\text{-}bool \Rightarrow ilet\text{-}reg\text{-}exp\text{-}bool$

primrec

$core\text{-}salt\text{-}reg\text{-}exp\text{-}bool\text{-}to\text{-}ilet\ s\ (CoreSREAtom\ a) = (BREAtom\ (\lambda x.\ a\ (s\ x)))$

$core\text{-}salt\text{-}reg\text{-}exp\text{-}bool\text{-}to\text{-}ilet\ s\ (\mathbf{not}\ f) =$

$(\neg_{bre}\ core\text{-}salt\text{-}reg\text{-}exp\text{-}bool\text{-}to\text{-}ilet\ s\ f)$

$core\text{-}salt\text{-}reg\text{-}exp\text{-}bool\text{-}to\text{-}ilet\ s\ (f1\ \mathbf{and}\ f2) =$

$(core\text{-}salt\text{-}reg\text{-}exp\text{-}bool\text{-}to\text{-}ilet\ s\ f1 \wedge_{bre} core\text{-}salt\text{-}reg\text{-}exp\text{-}bool\text{-}to\text{-}ilet\ s\ f2)$

$core\text{-}salt\text{-}reg\text{-}exp\text{-}bool\text{-}to\text{-}ilet\ s\ (f1\ \mathbf{or}\ f2) =$

$(core\text{-}salt\text{-}reg\text{-}exp\text{-}bool\text{-}to\text{-}ilet\ s\ f1 \vee_{bre} core\text{-}salt\text{-}reg\text{-}exp\text{-}bool\text{-}to\text{-}ilet\ s\ f2)$

$core\text{-}salt\text{-}reg\text{-}exp\text{-}bool\text{-}to\text{-}ilet\ s\ (f1\ \mathbf{implies}\ f2) =$

$(core\text{-}salt\text{-}reg\text{-}exp\text{-}bool\text{-}to\text{-}ilet\ s\ f1 \rightarrow_{bre} core\text{-}salt\text{-}reg\text{-}exp\text{-}bool\text{-}to\text{-}ilet\ s\ f2)$

$core\text{-}salt\text{-}reg\text{-}exp\text{-}bool\text{-}to\text{-}ilet\ s\ (f1\ \mathbf{equals}\ f2) =$

$(core\text{-}salt\text{-}reg\text{-}exp\text{-}bool\text{-}to\text{-}ilet\ s\ f1 \leftrightarrow_{bre} core\text{-}salt\text{-}reg\text{-}exp\text{-}bool\text{-}to\text{-}ilet\ s\ f2)$

Translating regular expressions to ILET.

consts

$sre\text{-}core\text{-}to\text{-}ilet :: (Time \Rightarrow 'a) \Rightarrow 'a\ core\text{-}salt\text{-}reg\text{-}exp \Rightarrow ilet\text{-}reg\text{-}exp$

primrec

$(sre\text{-}core\text{-}to\text{-}ilet\ s\ (CoreSREBool\ b)) = BREBool\ (core\text{-}salt\text{-}reg\text{-}exp\text{-}bool\text{-}to\text{-}ilet\ s\ b)$

$(sre\text{-}core\text{-}to\text{-}ilet\ s\ \varepsilon) = \varepsilon$

$(\mathit{sre\text{-}core\text{-}to\text{-}ilet}\ s\ (a\ \mathbf{or}\ b)) = (\mathit{sre\text{-}core\text{-}to\text{-}ilet}\ s\ a \lor \mathit{sre\text{-}core\text{-}to\text{-}ilet}\ s\ b)$
$(\mathit{sre\text{-}core\text{-}to\text{-}ilet}\ s\ (a\ ';'\ b)) = (\mathit{sre\text{-}core\text{-}to\text{-}ilet}\ s\ a\ ';'_{bre}\ \mathit{sre\text{-}core\text{-}to\text{-}ilet}\ s\ b)$
$(\mathit{sre\text{-}core\text{-}to\text{-}ilet}\ s\ (a\ ':'\ b)) = (\mathit{sre\text{-}core\text{-}to\text{-}ilet}\ s\ a\ ':'_{bre}\ \mathit{sre\text{-}core\text{-}to\text{-}ilet}\ s\ b)$
$(\mathit{sre\text{-}core\text{-}to\text{-}ilet}\ s\ (b\ '{*}'\ [\geq n]_{coresre})) =$
$((\mathit{core\text{-}salt\text{-}reg\text{-}exp\text{-}bool\text{-}to\text{-}ilet}\ s\ b)\ '{*}'\ [\geq n]_{bre})$

4.3.2 Translation of *until* and *from* operators to ILET

Translating the extended *until* operator to ILET.

consts
$\mathit{salt\text{-}exclincl\text{-}to\text{-}cut} :: \mathit{SALT\text{-}excl\text{-}incl} \Rightarrow (\mathit{iT} \Rightarrow \mathit{Time} \Rightarrow \mathit{iT})$
$\mathit{salt\text{-}reqoptweak\text{-}to\text{-}ilet} ::$
$\mathit{SALT\text{-}req\text{-}opt\text{-}weak} \Rightarrow (\mathit{Time} \Rightarrow \mathit{bool}) \Rightarrow (\mathit{Time} \Rightarrow \mathit{bool}) \Rightarrow (\mathit{Time} \Rightarrow \mathit{bool})$
primrec
$\mathit{salt\text{-}exclincl\text{-}to\text{-}cut}\ \mathbf{excl} = (op\ \downarrow<)$
$\mathit{salt\text{-}exclincl\text{-}to\text{-}cut}\ \mathbf{incl} = (op\ \downarrow\leq)$
primrec
$\mathit{salt\text{-}reqoptweak\text{-}to\text{-}ilet}\ \mathbf{req}\ f1\ f2 = (\lambda t.\ \mathit{False})$
$\mathit{salt\text{-}reqoptweak\text{-}to\text{-}ilet}\ \mathbf{opt}\ f1\ f2 = (\lambda t.\ \Box\ t2\ \{t..\}.\ \neg\ f2\ t2)$
$\mathit{salt\text{-}reqoptweak\text{-}to\text{-}ilet}\ \mathbf{weak}\ f1\ f2 = (\lambda t.\ \Box\ t1\ \{t..\}.\ f1\ t1)$
constdefs
$\mathit{salt\text{-}untilext\text{-}to\text{-}ilet} ::$
$\mathit{Time} \Rightarrow \mathit{SALT\text{-}excl\text{-}incl} \Rightarrow \mathit{SALT\text{-}req\text{-}opt\text{-}weak} \Rightarrow$
$(\mathit{Time} \Rightarrow \mathit{bool}) \Rightarrow (\mathit{Time} \Rightarrow \mathit{bool}) \Rightarrow$
bool
$\mathit{salt\text{-}untilext\text{-}to\text{-}ilet}\ t\ \mathit{exclincl}\ \mathit{reqoptweak}\ f1\ f2 \equiv ($
$(\Diamond\ t2\ \{t..\}.\ (f2\ t2 \land (\Box\ t1\ ((\mathit{salt\text{-}exclincl\text{-}to\text{-}cut}\ \mathit{exclincl})\ \{t..\}\ t2).\ f1\ t1))) \lor$
$(\mathit{salt\text{-}reqoptweak\text{-}to\text{-}ilet}\ \mathit{reqoptweak}\ f1\ f2)\ t)$

The *excl/incl* parameter for the extended *until* operator specifies, whether the time point, at which f_2 becomes true, is excluded from the interval, in which f_1 must hold. This is done by selecting the corresponding interval cut operator, excluding ($\downarrow\leq$) or including ($\downarrow<$) the time point, where the interval is cut.

lemma
$\mathit{salt\text{-}exclincl\text{-}to\text{-}cut\text{-}\text{-}excl}$: $(\mathit{salt\text{-}exclincl\text{-}to\text{-}cut}\ \mathbf{excl}\ I\ t) = (I \downarrow< t)$ **and**
$\mathit{salt\text{-}exclincl\text{-}to\text{-}cut\text{-}\text{-}incl}$: $(\mathit{salt\text{-}exclincl\text{-}to\text{-}cut}\ \mathbf{incl}\ I\ t) = (I \downarrow\leq t)$

Translating the *from* operator to ILET. Here the *excl/incl* parameter specifies, whether f must become true at the time point, where a is valid, or at the next time point. The *req/opt* parameter specifies, whether the formula is also fulfilled, if a never becomes true (parameter value *opt*).

constdefs
$\mathit{salt\text{-}from\text{-}to\text{-}ilet} ::$
$\mathit{Time} \Rightarrow \mathit{SALT\text{-}excl\text{-}incl} \Rightarrow \mathit{SALT\text{-}req\text{-}opt} \Rightarrow$
$(\mathit{Time} \Rightarrow \mathit{bool}) \Rightarrow (\mathit{Time} \Rightarrow \mathit{bool}) \Rightarrow$
bool
$\mathit{salt\text{-}from\text{-}to\text{-}ilet}\ t\ \mathit{exclincl}\ \mathit{reqopt}\ f\ a \equiv$
$(\Diamond\ t2\ \{t..\}.\ ($
$a\ t2 \land (\Box\ t1\ (\{t..\} \downarrow< t2).\ \neg\ a\ t1) \land$
$(f\ (\mathit{case}\ \mathit{exclincl}\ \mathit{of}\ \mathbf{excl} \Rightarrow \mathit{Suc}\ t2 \mid \mathbf{incl} \Rightarrow t2)))) \lor$
$(\mathit{case}\ \mathit{reqopt}\ \mathit{of}\ \mathbf{req} \Rightarrow \mathit{False} \mid \mathbf{opt} \Rightarrow (\Box\ t1\ \{t..\}.\ \neg\ a\ t1))$

4.3.3 Translation of core SALT formulas to ILET

The semantics of a core SALT formula is given by its translation to ILET.

consts
$\mathit{core\text{-}salt\text{-}valid} :: (\mathit{Time} \Rightarrow 'a) \Rightarrow \mathit{Time} \Rightarrow 'a\ \mathit{core\text{-}salt\text{-}formula} \Rightarrow \mathit{bool}$
$(\ (\text{-} \models_{coresalt} \text{-}\ \text{-})\ [80,80]\ 80)$
primrec

$s \models_{coresalt}\ t\ (CoreSALTAtom\ a) = (a\ (s\ t))$
$s \models_{coresalt}\ t\ (\mathbf{not}\ f) = (\neg\ s \models_{coresalt}\ t\ f)$
$s \models_{coresalt}\ t\ (f1\ \mathbf{and}\ f2) = (s \models_{coresalt}\ t\ f1 \wedge s \models_{coresalt}\ t\ f2)$
$s \models_{coresalt}\ t\ (f1\ \mathbf{or}\ f2) = (s \models_{coresalt}\ t\ f1 \vee s \models_{coresalt}\ t\ f2)$
$s \models_{coresalt}\ t\ (f1\ \mathbf{implies}\ f2) = (s \models_{coresalt}\ t\ f1 \longrightarrow s \models_{coresalt}\ t\ f2)$
$s \models_{coresalt}\ t\ (f1\ \mathbf{equals}\ f2) = (s \models_{coresalt}\ t\ f1 \longleftrightarrow s \models_{coresalt}\ t\ f2)$
$s \models_{coresalt}\ t\ (\mathbf{next}\ f) = (\bigcirc_S\ t1\ t\ \{0..\}.\ s \models_{coresalt}\ t1\ f)$
$s \models_{coresalt}\ t\ (\mathbf{always}\ f) = (\Box\ t1\ \{t..\}.\ s \models_{coresalt}\ t1\ f)$
$s \models_{coresalt}\ t\ (\mathbf{eventually}\ f) = (\Diamond\ t1\ \{t..\}.\ s \models_{coresalt}\ t1\ f)$
$s \models_{coresalt}\ t\ (f1\ \mathbf{until}\ exclincl\ reqoptweak\ f2) =$
$\quad (salt\text{-}untilext\text{-}to\text{-}ilet\ t\ exclincl\ reqoptweak\ (\lambda t.\ s \models_{coresalt}\ t\ f1)\ (\lambda t.\ s \models_{coresalt}\ t\ f2))$
$s \models_{coresalt}\ t\ (f\ \mathbf{from}\ exclincl\ reqopt\ a) =$
$\quad (salt\text{-}from\text{-}to\text{-}ilet\ t\ exclincl\ reqopt\ (\lambda t.\ s \models_{coresalt}\ t\ f)\ (\lambda t.\ a\ (s\ t)))$
$s \models_{coresalt}\ t\ ('|'\ r\ '|'_{coresre}) = (\Diamond\ t2\ \{t..\}.\ (\models_{bre}\ t\ t2\ (sre\text{-}core\text{-}to\text{-}ilet\ s\ r)))$
$s \models_{coresalt}\ t\ ('|'\ r\ ';'_{end}\ f\ '|'_{coresre}) =$
$\quad (\Diamond\ t2\ \{t..\}.\ (\models_{bre}\ t\ t2\ (sre\text{-}core\text{-}to\text{-}ilet\ s\ r)) \wedge (s \models_{coresalt}\ t2\ f))$
$s \models_{coresalt}\ t\ ('|'\ r\ ':'_{end}\ f\ '|'_{coresre}) =$
$\quad (\Diamond\ t2\ \{t..\}.\ (\models_{bre}\ t\ (Suc\ t2)\ (sre\text{-}core\text{-}to\text{-}ilet\ s\ r)) \wedge (s \models_{coresalt}\ t2\ f))$

4.4 Sequence operators and expressions matching empty words

Definition of well-formedness condition w.r.t. proper overlaps: a core SALT regular expression is considered well-formed w.r.t. the overlap operator if for every overlap operator both operands cannot match the empty word/interval.

Core SALT expressions matching the empty word, i.e., an interval of length 0.

primrec
$\quad core\text{-}salt\text{-}reg\text{-}exp\text{-}matches\text{-}epsilon :: 'a\ core\text{-}salt\text{-}reg\text{-}exp \Rightarrow bool$
where
$\quad core\text{-}salt\text{-}reg\text{-}exp\text{-}matches\text{-}epsilon\ (CoreSREBool\ b) = False$
$|\ core\text{-}salt\text{-}reg\text{-}exp\text{-}matches\text{-}epsilon\ \varepsilon = True$
$|\ core\text{-}salt\text{-}reg\text{-}exp\text{-}matches\text{-}epsilon\ (a\ \mathbf{or}\ b) =$
$\quad (core\text{-}salt\text{-}reg\text{-}exp\text{-}matches\text{-}epsilon\ a \vee core\text{-}salt\text{-}reg\text{-}exp\text{-}matches\text{-}epsilon\ b)$
$|\ core\text{-}salt\text{-}reg\text{-}exp\text{-}matches\text{-}epsilon\ (a\ ';'\ b) =$
$\quad (core\text{-}salt\text{-}reg\text{-}exp\text{-}matches\text{-}epsilon\ a \wedge core\text{-}salt\text{-}reg\text{-}exp\text{-}matches\text{-}epsilon\ b)$
$|\ core\text{-}salt\text{-}reg\text{-}exp\text{-}matches\text{-}epsilon\ (a\ ':'\ b) =$
$\quad (core\text{-}salt\text{-}reg\text{-}exp\text{-}matches\text{-}epsilon\ a \wedge core\text{-}salt\text{-}reg\text{-}exp\text{-}matches\text{-}epsilon\ b)$
$|\ core\text{-}salt\text{-}reg\text{-}exp\text{-}matches\text{-}epsilon\ (b\ '*'\ [\geq n]_{coresre}) = (n = 0)$

Function indicating whether the first expression in a sequence matches the empty word.

fun
$\quad core\text{-}salt\text{-}reg\text{-}exp\text{-}seq\text{-}first\text{-}matches\text{-}epsilon :: 'a\ core\text{-}salt\text{-}reg\text{-}exp \Rightarrow bool$
where
$\quad core\text{-}salt\text{-}reg\text{-}exp\text{-}seq\text{-}first\text{-}matches\text{-}epsilon\ (a\ ';'\ b) =$
$\quad\ core\text{-}salt\text{-}reg\text{-}exp\text{-}seq\text{-}first\text{-}matches\text{-}epsilon\ a$
$|\ core\text{-}salt\text{-}reg\text{-}exp\text{-}seq\text{-}first\text{-}matches\text{-}epsilon\ (a\ ':'\ b) =$
$\quad\ core\text{-}salt\text{-}reg\text{-}exp\text{-}seq\text{-}first\text{-}matches\text{-}epsilon\ a$
$|\ core\text{-}salt\text{-}reg\text{-}exp\text{-}seq\text{-}first\text{-}matches\text{-}epsilon\ (a\ \mathbf{or}\ b) =$
$\quad (core\text{-}salt\text{-}reg\text{-}exp\text{-}seq\text{-}first\text{-}matches\text{-}epsilon\ a \vee$
$\quad\ core\text{-}salt\text{-}reg\text{-}exp\text{-}seq\text{-}first\text{-}matches\text{-}epsilon\ b)$
$|\ core\text{-}salt\text{-}reg\text{-}exp\text{-}seq\text{-}first\text{-}matches\text{-}epsilon\ r = core\text{-}salt\text{-}reg\text{-}exp\text{-}matches\text{-}epsilon\ r$

Analogue function indicating whether the last expression in a sequence matches the empty word.

fun
$\quad core\text{-}salt\text{-}reg\text{-}exp\text{-}seq\text{-}last\text{-}matches\text{-}epsilon :: 'a\ core\text{-}salt\text{-}reg\text{-}exp \Rightarrow bool$
where
$\quad core\text{-}salt\text{-}reg\text{-}exp\text{-}seq\text{-}last\text{-}matches\text{-}epsilon\ (a\ ';'\ b) =$
$\quad\ core\text{-}salt\text{-}reg\text{-}exp\text{-}seq\text{-}last\text{-}matches\text{-}epsilon\ b$
$|\ core\text{-}salt\text{-}reg\text{-}exp\text{-}seq\text{-}last\text{-}matches\text{-}epsilon\ (a\ ':'\ b) =$
$\quad\ core\text{-}salt\text{-}reg\text{-}exp\text{-}seq\text{-}last\text{-}matches\text{-}epsilon\ b$

```
| core-salt-reg-exp-seq-last-matches-epsilon (a or b) =
   (core-salt-reg-exp-seq-last-matches-epsilon a ∨
    core-salt-reg-exp-seq-last-matches-epsilon b)
| core-salt-reg-exp-seq-last-matches-epsilon r = core-salt-reg-exp-matches-epsilon r
```

Function determining whether the core SALT regular expression contains a sequence where an expression matching the empty word neighbours the overlap operator.

```
fun
  core-salt-reg-exp-overlap-with-epsilon :: 'a core-salt-reg-exp ⇒ bool
where
  core-salt-reg-exp-overlap-with-epsilon (a ':' b) =
    (core-salt-reg-exp-seq-last-matches-epsilon a ∨
     core-salt-reg-exp-seq-first-matches-epsilon b ∨
     core-salt-reg-exp-overlap-with-epsilon a ∨ core-salt-reg-exp-overlap-with-epsilon b)
| core-salt-reg-exp-overlap-with-epsilon (a ';' b) =
    (core-salt-reg-exp-overlap-with-epsilon a ∨ core-salt-reg-exp-overlap-with-epsilon b)
| core-salt-reg-exp-overlap-with-epsilon (a or b) =
    (core-salt-reg-exp-overlap-with-epsilon a ∨ core-salt-reg-exp-overlap-with-epsilon b)
| core-salt-reg-exp-overlap-with-epsilon r = False
```

Some examples of core SALT regular expressions with and without overlaps with empty words.

```
lemma
  let
  a1 = CoreSREBool a1;  a2 = CoreSREBool a2;  a3 = CoreSREBool a3;  a4 = CoreSREBool a4;
  a5 = CoreSREBool a5;  a6 = CoreSREBool a6;  a7 = CoreSREBool a7
  in
  (core-salt-reg-exp-overlap-with-epsilon ((a1 ';' a2) ';' (a3 ':' (a4 ';' a5) ';'
    (a6 ':' a7))) = False) ∧
  (core-salt-reg-exp-overlap-with-epsilon ((a1 ';' a2) or (a3 ':' (a4 ';' a5) ';'
    (a6 ':' a7))) = False) ∧
  (core-salt-reg-exp-overlap-with-epsilon ((a1 ';' a2) or (ε ':' (a4 ';' a5) ';'
    (a6 ':' a7))) = True) ∧
  (core-salt-reg-exp-overlap-with-epsilon ((a1 ':' ε) or (a3 ':' (a4 ';' a5) ';'
    (a6 ':' a7))) = True) ∧
  (core-salt-reg-exp-overlap-with-epsilon ((a1 ';' a2) or (a3 ':' ((b '*' [≥ 1]_coresre) ';' a5) ';'
    (a6 ':' a7))) = False) ∧
  (core-salt-reg-exp-overlap-with-epsilon ((a1 ';' a2) or (a3 ':' ((b '*' [≥ 0]_coresre) ';' a5) ';'
    (a6 ':' a7))) = True)
```

A core SALT regular expression is well-formed if no regular expression matching the empty word neighbours the overlap operator.

```
definition core-salt-reg-exp-proper-overlap :: 'a core-salt-reg-exp ⇒ bool where
  core-salt-reg-exp-proper-overlap r ≡ ¬ (core-salt-reg-exp-overlap-with-epsilon r)
```

Remarkably, a core SALT regular expression is well-formed iff its translation to ILET is well-formed.

```
lemma core-salt-reg-exp-to-ilet--proper-overlap-eq:
  (ilet-reg-exp-proper-overlap (sre-core-to-ilet s r)) =
  (core-salt-reg-exp-proper-overlap r)
```

A core SALT formula is well-formed if all regular expressions in it are well-formed w.r.t. overlaps with expressions matching empty words.

```
consts
  core-salt-proper-overlap :: 'a core-salt-formula ⇒ bool
primrec
  core-salt-proper-overlap (CoreSALTAtom a) = True
  core-salt-proper-overlap (not f) = (core-salt-proper-overlap f)
  core-salt-proper-overlap (f1 and f2) =
    (core-salt-proper-overlap f1 ∧ core-salt-proper-overlap f2)
  core-salt-proper-overlap (f1 or f2) =
    (core-salt-proper-overlap f1 ∧ core-salt-proper-overlap f2)
  core-salt-proper-overlap (f1 implies f2) =
```

$(\mathit{core\text{-}salt\text{-}proper\text{-}overlap}\ f1 \wedge \mathit{core\text{-}salt\text{-}proper\text{-}overlap}\ f2)$
$\mathit{core\text{-}salt\text{-}proper\text{-}overlap}\ (f1\ \textbf{equals}\ f2) =$
$(\mathit{core\text{-}salt\text{-}proper\text{-}overlap}\ f1 \wedge \mathit{core\text{-}salt\text{-}proper\text{-}overlap}\ f2)$
$\mathit{core\text{-}salt\text{-}proper\text{-}overlap}\ (\textbf{next}\ f) = (\mathit{core\text{-}salt\text{-}proper\text{-}overlap}\ f)$
$\mathit{core\text{-}salt\text{-}proper\text{-}overlap}\ (\textbf{always}\ f) = (\mathit{core\text{-}salt\text{-}proper\text{-}overlap}\ f)$
$\mathit{core\text{-}salt\text{-}proper\text{-}overlap}\ (\textbf{eventually}\ f) = (\mathit{core\text{-}salt\text{-}proper\text{-}overlap}\ f)$
$\mathit{core\text{-}salt\text{-}proper\text{-}overlap}\ (f1\ \textbf{until}\ \mathit{exclincl}\ \mathit{reqoptweak}\ f2) =$
$(\mathit{core\text{-}salt\text{-}proper\text{-}overlap}\ f1 \wedge \mathit{core\text{-}salt\text{-}proper\text{-}overlap}\ f2)$
$\mathit{core\text{-}salt\text{-}proper\text{-}overlap}\ (f\ \textbf{from}\ \mathit{exclincl}\ \mathit{reqopt}\ a) = (\mathit{core\text{-}salt\text{-}proper\text{-}overlap}\ f)$
$\mathit{core\text{-}salt\text{-}proper\text{-}overlap}\ ('|'\ r\ '|'_{coresre}) = (\mathit{core\text{-}salt\text{-}reg\text{-}exp\text{-}proper\text{-}overlap}\ r)$
$\mathit{core\text{-}salt\text{-}proper\text{-}overlap}\ ('|'\ r\ ';'_{end}\ f\ '|'_{coresre}) =$
$(\mathit{core\text{-}salt\text{-}reg\text{-}exp\text{-}proper\text{-}overlap}\ r \wedge \mathit{core\text{-}salt\text{-}proper\text{-}overlap}\ f)$
$\mathit{core\text{-}salt\text{-}proper\text{-}overlap}\ ('|'\ r\ ':'_{end}\ f\ '|'_{coresre}) =$
$(\mathit{core\text{-}salt\text{-}reg\text{-}exp\text{-}proper\text{-}overlap}\ r \wedge \mathit{core\text{-}salt\text{-}proper\text{-}overlap}\ f \wedge$
$\neg\ \mathit{core\text{-}salt\text{-}reg\text{-}exp\text{-}seq\text{-}last\text{-}matches\text{-}epsilon}\ r)$

The well-formedness precondition *core-salt-proper-overlap f* will be employed in the main translation validation theorem *core-salt-to-ltl-equiv-core-salt-valid* in Sec. 4.5.2, because this theorem will consider core SALT formulas with proper overlaps in regular expressions and hence well-defined semantics. It will not state anything about core SALT formulas with improper overlaps, e.g., $/\ a; b * [\geq 0] : c\ /$ because for them no well-defined semantics exist. Consider the example of the two formulas $/\ (a;\varepsilon) : c\ /$ and $/\ a;\varepsilon : c\ /$. They are mapped to two different ILET regular expressions and thus assigned two different meanings: $/\ (a;\varepsilon) : c\ / = /\ a\ :\ c\ /_{\mathsf{ilet}}$, because the empty word ε is "consumed" by a, while $/\ a;\varepsilon : c\ / = /\ a; \mathsf{False}\ /$ $= \mathsf{False}$, because the empty word in the sub-expression $\varepsilon\ :\ c$ cannot match any interval of length > 0, as required by the sub-formula $\models_{bre} t\ (\mathit{Suc}\ t)\ \varepsilon$ in the definition of *ilet-reg-exp-match*. At the same time the translation to LTL yields the right associative interpretation $/\ a;\varepsilon : c\ / = a\ \wedge\ \bigcirc\ c$ for both formulas therefore mapping two different core SALT formulas with different meanings to the same LTL formula. Thus, a proper semantics definition for such cases is not possible, unless we use a semantics definition that cannot distinguish such formulas, e.g., by forcing all regular expressions to be right associative and hence ignoring parentheses in sequences, which would be a purely syntactic solution, reasonable for a pragmatic compiler but not suitable for formal semantics definition.

4.5 Formal validation of core SALT translation to LTL

The translation of core SALT to LTL is validated by proving that the semantics of an LTL formula obtained by translating a core SALT formula is equivalent to the semantics of the core SALT formula directly given by its ILET translation.

4.5.1 Selected auxiliary translation validation lemmas

Translation validation for the regular repetition operator $*$:

lemma $\mathit{core\text{-}salt\text{-}to\text{-}ltl\text{-}equiv\text{-}core\text{-}salt\text{-}valid\text{-}\text{-}RegExp\text{-}StarGe}$:
$(s \models_{ltl} t\ (\mathit{sre\text{-}core\text{-}to\text{-}ltl}\ (b\ '*'\ [\geq n]_{coresre}))) =$
$(\Diamond\ t2\ \{t..\}.\ \models_{bre} t\ t2\ (\mathit{sre\text{-}core\text{-}to\text{-}ilet}\ s\ (b\ '*'\ [\geq n]_{coresre})))$

Translation validation for the sequence operator ; and the sequence overlap operator ':':

lemma
$\mathit{core\text{-}salt\text{-}to\text{-}ltl\text{-}equiv\text{-}core\text{-}salt\text{-}valid\text{-}\text{-}RegExp\text{-}Subsequent}$: $\bigwedge t.$
$\mathit{core\text{-}salt\text{-}reg\text{-}exp\text{-}proper\text{-}overlap}\ r \Longrightarrow$
$(s \models_{ltl} t\ (\mathit{sre\text{-}subsequent\text{-}to\text{-}ltl}\ r\ f)) =$
$(\Diamond\ t2\ \{t..\}.\ ((\models_{bre} t\ t2\ (\mathit{sre\text{-}core\text{-}to\text{-}ilet}\ s\ r)) \wedge (s \models_{ltl} t2\ f)))$ **and**
$\mathit{core\text{-}salt\text{-}to\text{-}ltl\text{-}equiv\text{-}core\text{-}salt\text{-}valid\text{-}\text{-}RegExp\text{-}Overlap}$: $\bigwedge t.$
$[\![\ \mathit{core\text{-}salt\text{-}reg\text{-}exp\text{-}proper\text{-}overlap}\ r;\ \neg\ \mathit{core\text{-}salt\text{-}reg\text{-}exp\text{-}seq\text{-}last\text{-}matches\text{-}epsilon}\ r\ ]\!] \Longrightarrow$
$(s \models_{ltl} t\ (\mathit{sre\text{-}overlap\text{-}to\text{-}ltl}\ r\ f)) =$
$(\Diamond\ t2\ \{t..\}.\ ((\models_{bre} t\ (\mathit{Suc}\ t2)\ (\mathit{sre\text{-}core\text{-}to\text{-}ilet}\ s\ r)) \wedge (s \models_{ltl} t2\ f)))$

Translation validation for regular expressions:

lemma $\mathit{core\text{-}salt\text{-}to\text{-}ltl\text{-}equiv\text{-}core\text{-}salt\text{-}valid\text{-}\text{-}RegExp}$: $\bigwedge t.$

$\mathit{core\text{-}salt\text{-}reg\text{-}exp\text{-}proper\text{-}overlap}\ sre \Longrightarrow$
$(s \models_{ltl} t\ (\mathit{sre\text{-}core\text{-}to\text{-}ltl}\ sre)) = (\Diamond\ t2\ \{t..\}.\ \models_{bre} t\ t2\ (\mathit{sre\text{-}core\text{-}to\text{-}ilet}\ s\ sre))$

Translation validation for regular expressions ending with a core SALT formula:

lemma $\mathit{core\text{-}salt\text{-}to\text{-}ltl\text{-}equiv\text{-}core\text{-}salt\text{-}valid\text{-}\text{-}RegExp\text{-}Subsequent\text{-}SeqSaltFinish}$: $\bigwedge f\ t.$
$[\![\ \mathit{core\text{-}salt\text{-}reg\text{-}exp\text{-}proper\text{-}overlap}\ r;$
$\bigwedge t.\ (s \models_{ltl} t\ (\mathit{core\text{-}salt\text{-}to\text{-}ltl}\ f)) = s \models_{coresalt} t\ f\ ]\!] \Longrightarrow$
$(s \models_{ltl} t\ (\mathit{sre\text{-}subsequent\text{-}to\text{-}ltl}\ r\ (\mathit{core\text{-}salt\text{-}to\text{-}ltl}\ f))) =$
$(\Diamond\ t2\ \{t..\}.\ \models_{bre} t\ t2\ (\mathit{sre\text{-}core\text{-}to\text{-}ilet}\ s\ r) \wedge s \models_{coresalt} t2\ f)$

lemma $\mathit{core\text{-}salt\text{-}to\text{-}ltl\text{-}equiv\text{-}core\text{-}salt\text{-}valid\text{-}\text{-}RegExp\text{-}Overlap\text{-}SeqSaltFinish}$: $\bigwedge f\ t.$
$[\![\ \mathit{core\text{-}salt\text{-}reg\text{-}exp\text{-}proper\text{-}overlap}\ r;\ \neg\ \mathit{core\text{-}salt\text{-}reg\text{-}exp\text{-}seq\text{-}last\text{-}matches\text{-}epsilon}\ r;$
$\bigwedge t.\ (s \models_{ltl} t\ (\mathit{core\text{-}salt\text{-}to\text{-}ltl}\ f)) = s \models_{coresalt} t\ f\ ]\!] \Longrightarrow$
$(s \models_{ltl} t\ (\mathit{sre\text{-}overlap\text{-}to\text{-}ltl}\ r\ (\mathit{core\text{-}salt\text{-}to\text{-}ltl}\ f))) =$
$(\Diamond\ t2\ \{t..\}.\ \models_{bre} t\ (\mathit{Suc}\ t2)\ (\mathit{sre\text{-}core\text{-}to\text{-}ilet}\ s\ r) \wedge s \models_{coresalt} t2\ f)$

Translation validation for the extended *until* operator:

lemma $\mathit{core\text{-}salt\text{-}to\text{-}ltl\text{-}equiv\text{-}core\text{-}salt\text{-}valid\text{-}\text{-}UntilExt}$:
$[\![\ \bigwedge t.\ (s \models_{ltl} t\ (\mathit{core\text{-}salt\text{-}to\text{-}ltl}\ f1)) = s \models_{coresalt} t\ f1;$
$\bigwedge t.\ (s \models_{ltl} t\ (\mathit{core\text{-}salt\text{-}to\text{-}ltl}\ f2)) = s \models_{coresalt} t\ f2\ ]\!] \Longrightarrow$
$(s \models_{ltl} t\ (\mathit{ltl\text{-}untilext}\ \mathit{exclincl}\ \mathit{reqoptweak}$
$(\mathit{core\text{-}salt\text{-}to\text{-}ltl}\ f1)\ (\mathit{core\text{-}salt\text{-}to\text{-}ltl}\ f2))) =$
$(\mathit{salt\text{-}untilext\text{-}to\text{-}ilet}\ t\ \mathit{exclincl}\ \mathit{reqoptweak}$
$(\lambda t.\ s \models_{coresalt} t\ f1)\ (\lambda t.\ s \models_{coresalt} t\ f2))$

Translation validation for the *from* operator:

lemma $\mathit{core\text{-}salt\text{-}to\text{-}ltl\text{-}equiv\text{-}core\text{-}salt\text{-}valid\text{-}\text{-}From}$:
$[\![\ \bigwedge t.\ (s \models_{ltl} t\ (\mathit{core\text{-}salt\text{-}to\text{-}ltl}\ f)) = s \models_{coresalt} t\ f\ ]\!] \Longrightarrow$
$(s \models_{ltl} t\ (\mathit{ltl\text{-}from}\ \mathit{exclincl}\ \mathit{reqopt}\ (\mathit{core\text{-}salt\text{-}to\text{-}ltl}\ f)\ a)) =$
$(\mathit{salt\text{-}from\text{-}to\text{-}ilet}\ t\ \mathit{exclincl}\ \mathit{reqopt}\ (\lambda t.\ s \models_{coresalt} t\ f)\ (\lambda t.\ a\ (s\ t)))$

4.5.2 Main translation validation theorem

Core SALT translation to LTL yields the same semantics as the core SALT semantics given by direct translation to ILET:

theorem $\mathit{core\text{-}salt\text{-}to\text{-}ltl\text{-}equiv\text{-}core\text{-}salt\text{-}valid}$: $\bigwedge t.$
$\mathit{core\text{-}salt\text{-}proper\text{-}overlap}\ f \Longrightarrow$
$(s \models_{ltl} t\ (\mathit{core\text{-}salt\text{-}to\text{-}ltl}\ f)) = (s \models_{coresalt} t\ f)$

The precondition *core-salt-proper-overlap f* indicates that we consider core SALT formulas with proper overlaps in regular expressions and hence well-defined semantics.[4]

5 Additional results for core SALT

5.1 LTL operators *until*, *weak until*, *release* in core SALT

Lemmas about expressing LTL operators $\mathcal{U}$, $\mathcal{W}$, $\mathcal{R}$ using the extended *until* operator in core SALT.

lemma $\mathit{core\text{-}salt\text{-}until\text{-}excl\text{-}req\text{-}ltl\text{-}until\text{-}equiv}$:
$[\![\ \mathit{core\text{-}salt\text{-}proper\text{-}overlap}\ f1;\ \mathit{core\text{-}salt\text{-}proper\text{-}overlap}\ f2\ ]\!] \Longrightarrow$
$(s \models_{coresalt} t\ (f1\ \mathbf{until}\ \mathbf{excl}\ \mathbf{req}\ f2)) =$
$(s \models_{ltl} t\ (\mathit{core\text{-}salt\text{-}to\text{-}ltl}\ f1)\ \mathcal{U}_{ltl}\ (\mathit{core\text{-}salt\text{-}to\text{-}ltl}\ f2))$

lemma $\mathit{core\text{-}salt\text{-}until\text{-}excl\text{-}weak\text{-}ltl\text{-}until\text{-}weak\text{-}equiv}$:
$[\![\ \mathit{core\text{-}salt\text{-}proper\text{-}overlap}\ f1;\ \mathit{core\text{-}salt\text{-}proper\text{-}overlap}\ f2\ ]\!] \Longrightarrow$
$(s \models_{coresalt} t\ (f1\ \mathbf{until}\ \mathbf{excl}\ \mathbf{weak}\ f2)) =$
$(s \models_{ltl} t\ (\mathit{core\text{-}salt\text{-}to\text{-}ltl}\ f1)\ \mathcal{W}_{ltl}\ (\mathit{core\text{-}salt\text{-}to\text{-}ltl}\ f2))$

lemma $\mathit{core\text{-}salt\text{-}until\text{-}incl\text{-}weak\text{-}ltl\text{-}release\text{-}equiv}$:
$[\![\ \mathit{core\text{-}salt\text{-}proper\text{-}overlap}\ f1;\ \mathit{core\text{-}salt\text{-}proper\text{-}overlap}\ f2\ ]\!] \Longrightarrow$

[4] The theorem does not state anything about core SALT formulas with improper overlaps, e.g., $/\ a; b * [\geq 0] : c\ /$, because for them no well-defined semantics exist.

$(s \models_{coresalt}$ t (*f1* **until incl weak** *f2*)) =
$(s \models_{ltl}$ t (core-salt-to-ltl *f2*) R_{ltl} (core-salt-to-ltl *f1*))

5.2 Expressive equivalence of core SALT and LTL

The expressiveness of core SALT (for well-formed formulas) and LTL is equivalent.

Translation function from LTL to core SALT:

consts
ltl-to-core-salt :: 'a ltl-formula ⇒ 'a core-salt-formula
primrec
ltl-to-core-salt (LTLAtom a) = CoreSALTAtom a
ltl-to-core-salt ($\neg_{ltl}$ f) = (**not** (ltl-to-core-salt f))
ltl-to-core-salt (f1 $\wedge_{ltl}$ f2) = ((ltl-to-core-salt f1) **and** (ltl-to-core-salt f2))
ltl-to-core-salt (f1 $\vee_{ltl}$ f2) = ((ltl-to-core-salt f1) **or** (ltl-to-core-salt f2))
ltl-to-core-salt (f1 $\rightarrow_{ltl}$ f2) = ((ltl-to-core-salt f1) **implies** (ltl-to-core-salt f2))
ltl-to-core-salt (f1 $\leftrightarrow_{ltl}$ f2) = ((ltl-to-core-salt f1) **equals** (ltl-to-core-salt f2))
ltl-to-core-salt ($\bigcirc_{ltl}$ f) = (**next** ltl-to-core-salt f)
ltl-to-core-salt ($\Box_{ltl}$ f) = (**always** (ltl-to-core-salt f))
ltl-to-core-salt ($\Diamond_{ltl}$ f) = (**eventually** (ltl-to-core-salt f))
ltl-to-core-salt (f1 U_{ltl} f2) = (
(ltl-to-core-salt f1) **until excl req** (ltl-to-core-salt f2))
ltl-to-core-salt (f1 W_{ltl} f2) = (
(ltl-to-core-salt f1) **until excl weak** (ltl-to-core-salt f2))
ltl-to-core-salt (f1 R_{ltl} f2) = (
(ltl-to-core-salt f2) **until incl weak** (ltl-to-core-salt f1))

Translation functions from core SALT to LTL and vice versa are inverse:

lemma ltl-to-core-salt-to-ltl-equiv: $\bigwedge t$.
$(s \models_{ltl}$ t core-salt-to-ltl (ltl-to-core-salt f)) = $(s \models_{ltl}$ t f)
lemma core-salt-to-ltl-to-core-salt-equiv:
core-salt-proper-overlap f $\Longrightarrow$
$(s \models_{coresalt}$ t ltl-to-core-salt (core-salt-to-ltl f)) = $(s \models_{coresalt}$ t f)

Each core SALT property can be expressed in LTL:

lemma core-salt-subset-ltl:
$\forall$(f::'a core-salt-formula). core-salt-proper-overlap f $\longrightarrow$
($\exists$(f'::'a ltl-formula). $(s \models_{coresalt}$ t f) = $(s \models_{ltl}$ t f'))

Each LTL property can be expressed in core SALT:

lemma ltl-subset-core-salt:
$\forall$(f::'a ltl-formula). $\exists$(f'::'a core-salt-formula). $(s \models_{ltl}$ t f) = $(s \models_{coresalt}$ t f')

Core SALT and LTL have equivalent expressiveness, i.e., the sets of properties on system runs s for a given time point t expressible in core SALT (considering well-formed formulas) and in LTL are equal:

theorem core-salt-ltl-equiv:
{p. $\exists$(f::'a core-salt-formula). core-salt-proper-overlap f $\wedge$ p s t = $(s \models_{coresalt}$ t f)} =
{p. $\exists$(f::'a ltl-formula). p s t = $(s \models_{ltl}$ t f)}

References

[Acc04] Accelera. *Property Specification Language Reference Manual, Version 1.1*, Jun 2004.

[BBDE+01] Ilan Beer, Shoham Ben-David, Cindy Eisner, Dana Fisman, Anna Gringauze, and Yoav Rodeh. The Temporal Logic Sugar. In *Computer Aided Verification, 13th International Conference, CAV 2001*, pages 363–367, 2001.

[BLS06] Andreas Bauer, Martin Leucker, and Jonathan Streit. SALT – Structured Assertion Language for Temporal Logic. In Zhiming Liu and Jifeng He, editors, *Formal Methods and Software*

Engineering, 8th International Conference on Formal Engineering Methods (ICFEM 2006), Proceedings, volume 4260 of *Lecture Notes in Computer Science*, pages 757–775. Springer, 2006.

[DAC99] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Patterns in Property Specifications for Finite-State Verification. In *ICSE 1999*, pages 411–420. IEEE Computer Society, 1999.

[Gor03] Michael J. C. Gordon. Validating the PSL/Sugar Semantics Using Automated Reasoning. *Formal Aspects of Computing*, 15(4):406–421, 2003.

[NPW02] T. Nipkow, L.C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[Pnu77] Amir Pnueli. The Temporal Logic of Programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 46–57. IEEE Computer Society, 1977.

[SAL] SALT Compiler. `http://salt.in.tum.de/`.

[Str06] Jonathan Streit. *SALT – Language Reference and Compiler Manual*, Apr 2006.

[Tra09] David Trachtenherz. *Eigenschaftsorientierte Beschreibung der logischen Architektur eingebetteter Systeme (Property-Oriented Description of Logical Architecture of Embedded Systems).* PhD thesis, Institut für Informatik, Technische Universität München, 2009.

[Tra11] David Trachtenherz. Interval Temporal Logic on Natural Numbers. In Gerwin Klein, Tobias Nipkow, and Lawrence Paulson, editors, *The Archive of Formal Proofs*. `http://afp.sourceforge.net/entries/AutoFocus-Stream.shtml`, February 2011.