GRATchk: Verified (UN)SAT Certificate Checker

Peter Lammich

January 28, 2023

Abstract

GRATchk is a formally verified and efficient checker for satisfiability and unsatisfiability certificates for Boolean formulas.

The verification covers the actual efficient implementation, and the semantics of a formula down to the integer sequences that represents it.

The satisfiability certificates are non-contradictory lists of literals, as output by any standard SAT solver. The unsatisfiability certificates are GRAT certificates, which can be generated from standard DRAT certificates by the GRATgen tool.

Contents

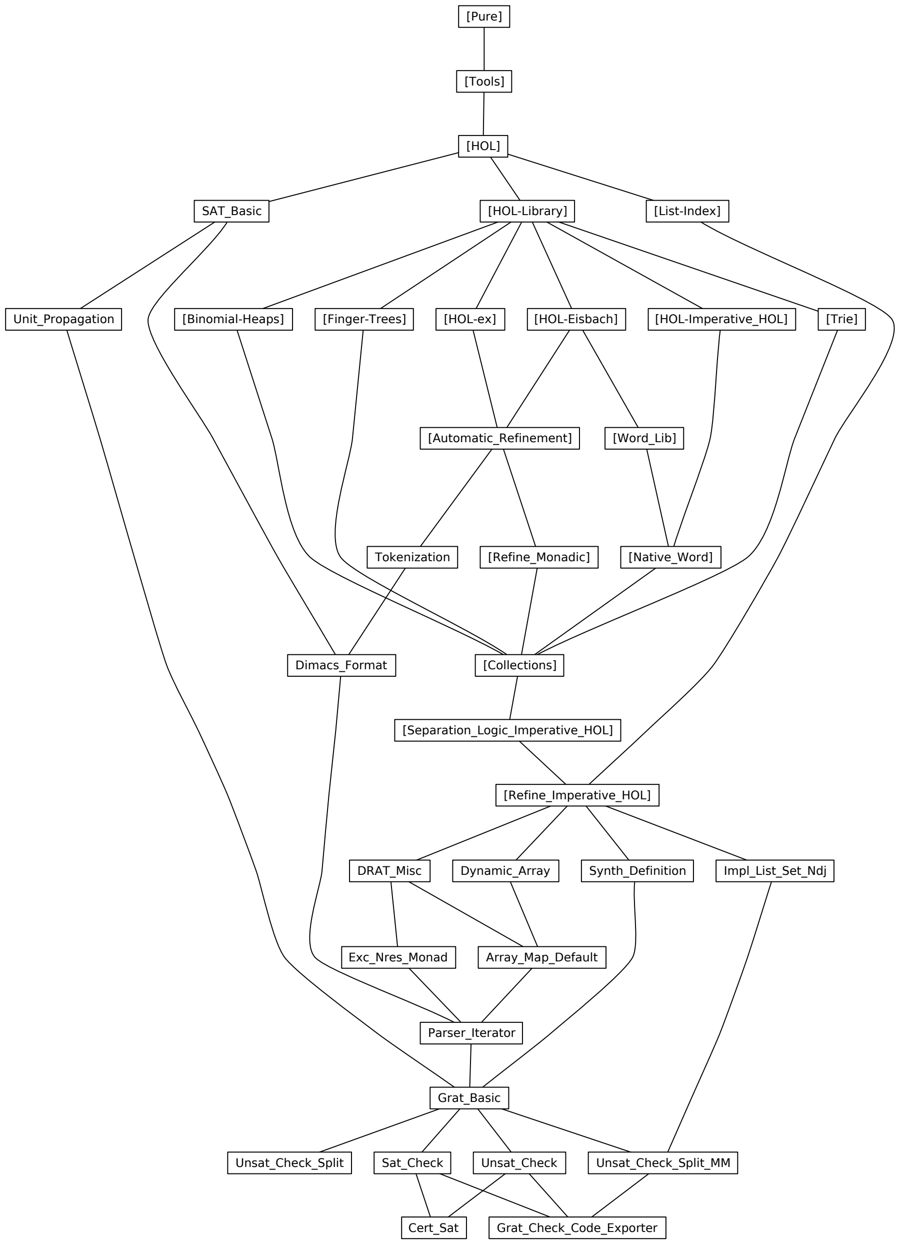

Figure 1: Theory dependency graph

1 Introduction

We present an efficient verified checker for satisfiability and unsatisfiability certificates obtained from SAT solvers.

Our sat certificates are lists of non-contradictory literals, as produced by virtually any SAT solver.

The de facto standard for unsat certificates is DRAT. Here, our checker uses a two step approach: The unverified GRATgen tool converts the DRAT certificates into GRAT certificates, which are then checked against the original formula by the verified GRATchk, presented in this formalization.

The GRAT certificates are engineered to admit a simple and efficient checker algorithm, which is well suited for formal verification. We use the Isabelle Refinement Framework to verify an efficient imperative implementation of the checker algorithm.

Our verification covers the semantics of a formula down to the integer sequence that represents it. This way, only a simple untrusted parser is required to read the formula from a file to an integer array. In Section 6.2, we give a complete and self-contained summary of what we actually proved.

2 Unit Propagation and RUP/RAT Checks

theory *Unit_Propagation*
imports *SAT_Basic*
begin

This theory formalizes the basics of unit propagation and RUP/RAT redundancy checks.

2.1 Partial Assignments

primrec *sem_lit'* :: *'a literal* ⇒ (*'a*⇀*bool*) ⇀ *bool* **where**
  sem_lit' (*Pos x*) *A* = *A x*
| *sem_lit'* (*Neg x*) *A* = *map_option Not* (*A x*)

definition *sem_clause'* :: *'a literal set* ⇒ (*'a* ⇀ *bool*) ⇀ *bool* **where**
  sem_clause' C A ≡
    if ∃ *l*∈*C*. *sem_lit' l A* = *Some True then Some True*
    else if ∀ *l*∈*C*. *sem_lit' l A* = *Some False then Some False*
    else None

definition *compat_assignment* :: (*'a* ⇀ *bool*) ⇒ (*'a* ⇒ *bool*) ⇒ *bool*
  where *compat_assignment A σ* ≡ ∀ *x v*. *A x* = *Some v* ⟶ *σ x* = *v*

lemma *sem_neg_lit'*[*simp*]:
  sem_lit' (*neg_lit l*) *A* = *map_option Not* (*sem_lit' l A*)
  ⟨*proof*⟩

lemma (**in** −) *sem_lit'_empty*[*simp*]: *sem_lit' l Map.empty* = *None*
  ⟨*proof*⟩

We install a custom case distinction rule for *bool option*, which has the cases *undec*, *false*, and *true*.

fun *boolopt_cases_aux* **where**
  boolopt_cases_aux None = ()
| *boolopt_cases_aux* (*Some False*) = ()
| *boolopt_cases_aux* (*Some True*) = ()

lemmas *boolopt_cases*[*case_names undec false true*, *cases type*]
  = *boolopt_cases_aux.cases*

lemma *not_Some_bool_if*: ⟦ *a*≠*Some False*; *a*≠*Some True* ⟧ ⟹ *a*=*None*
  ⟨*proof*⟩

Rules to trigger case distinctions on the semantics of a clause with a distinguished literal.

lemma *sem_clause_insert_eq_complete*:
  sem_clause' (*insert l C*) *A* = (*case sem_lit' l A of*

Some True ⇒ Some True
| Some False ⇒ sem_clause′ C A
| None ⇒ (case sem_clause′ C A of
None ⇒ None
| Some False ⇒ None
| Some True ⇒ Some True))
⟨proof⟩

lemma *sem_clause_empty*[*simp*]: *sem_clause′* {} *A* = *Some False*
⟨proof⟩

lemma *sem_clause′_insert_true*: *sem_clause′* (*insert l C*) *A* = *Some True* ⟷
sem_lit′ l A = *Some True* ∨ *sem_clause′ C A* = *Some True*
⟨proof⟩

lemma *sem_clause′_insert_false*[*simp*]:
sem_clause′ (*insert l C*) *A* = *Some False*
⟷ *sem_lit′ l A* = *Some False* ∧ *sem_clause′ C A* = *Some False*
⟨proof⟩

lemma *sem_clause′_union_false*[*simp*]:
sem_clause′ (*C1*∪*C2*) *A* = *Some False*
⟷ *sem_clause′ C1 A* = *Some False* ∧ *sem_clause′ C2 A* = *Some False*
⟨proof⟩

lemma *compat_assignment_empty*[*simp*]: *compat_assignment Map.empty* σ
⟨proof⟩

Assign variable such that literal becomes true

definition *assign_lit A l* ≡ *A*(*var_of_lit l* ↦ *is_pos l*)

lemma *assign_lit_simps*[*simp*]:
assign_lit A (*Pos x*) = *A*(*x*↦*True*)
assign_lit A (*Neg x*) = *A*(*x*↦*False*)
⟨proof⟩

lemma *assign_lit_dom*[*simp*]:
dom (*assign_lit A l*) = *insert* (*var_of_lit l*) (*dom A*)
⟨proof⟩

lemma *sem_lit_assign*[*simp*]: *sem_lit′ l* (*assign_lit A l*) = *Some True*
⟨proof⟩

lemma *sem_lit′_none_conv*: *sem_lit′ l A* = *None* ⟷ *A* (*var_of_lit l*) = *None*
⟨proof⟩

lemma *assign_undec_pres_dec_lit*:
⟦ *sem_lit′ l A* = *None*; *sem_lit′ l′ A* = *Some v* ⟧
⟹ *sem_lit′ l′* (*assign_lit A l*) = *Some v*
⟨proof⟩

lemma *assign_undec_pres_dec_clause*:
⟦ *sem_lit′ l A* = *None*; *sem_clause′ C A* = *Some v* ⟧
⟹ *sem_clause′ C* (*assign_lit A l*) = *Some v*
⟨proof⟩

lemma *sem_lit′_assign_conv*: *sem_lit′ l′* (*assign_lit A l*) = (
if l′=*l then Some True*
else if l′=*neg_lit l then Some False*
else sem_lit′ l′ A)
⟨proof⟩

Predicates for unit clauses

definition *is_unit_lit A C l*
  $\equiv$ *l*$\in$*C* $\wedge$ *sem_lit′ l A = None* $\wedge$ (*sem_clause′* (*C*$-${*l*}) *A = Some False*)
definition *is_unit_clause A C* $\equiv$ $\exists$ *l. is_unit_lit A C l*
definition *the_unit_lit A C* $\equiv$ *THE l. is_unit_lit A C l*

abbreviation (*input*) *is_conflict_clause A C* $\equiv$ *sem_clause′ C A = Some False*
abbreviation (*input*) *is_true_clause A C* $\equiv$ *sem_clause′ C A = Some True*

lemma *sem_clause′_false_conv*:
  sem_clause′ C A = Some False $\longleftrightarrow$ ($\forall$ *l*$\in$*C. sem_lit′ l A = Some False*)
  ⟨*proof*⟩

lemma *sem_clause′_true_conv*:
  sem_clause′ C A = Some True $\longleftrightarrow$ ($\exists$ *l*$\in$*C. sem_lit′ l A = Some True*)
  ⟨*proof*⟩

lemma *the_unit_lit_eq*[*simp*]: *is_unit_lit A C l* $\Longrightarrow$ *the_unit_lit A C = l*
  ⟨*proof*⟩

lemma *is_unit_lit_unique*: ⟦*is_unit_lit C A l1*; *is_unit_lit C A l2*⟧ $\Longrightarrow$ *l1=l2*
  ⟨*proof*⟩

lemma *is_unit_clauseE*:
  assumes *is_unit_clause A C*
  obtains *l C′* **where**
    C=insert l C′
    l$\notin$*C′*
    sem_lit′ l A = None
    sem_clause′ C′ A = Some False
    the_unit_lit A C = l
  ⟨*proof*⟩

lemma *is_unit_clauseE′*:
  assumes *is_unit_clause A C*
  obtains *l C′* **where**
    C=insert l C′
    l$\notin$*C′*
    sem_lit′ l A = None
    sem_clause′ C′ A = Some False
  ⟨*proof*⟩

lemma *sem_not_false_the_unit_lit*:
  assumes *is_unit_lit A C l*
  assumes *l′*$\in$*C*
  assumes *sem_lit′ l′ A* $\neq$ *Some False*
  shows *l′=l*
  ⟨*proof*⟩

lemma *sem_none_the_unit_lit*:
  assumes *is_unit_lit A C l*
  assumes *l′*$\in$*C*
  assumes *sem_lit′ l′ A = None*
  shows *l′=l*
  ⟨*proof*⟩

lemma *is_unit_lit_unique_ss*:
  ⟦*C′*$\subseteq$*C*; *is_unit_lit A C′ l′*; *is_unit_lit A C l*⟧ $\Longrightarrow$ *l′=l*
  ⟨*proof*⟩

lemma *is_unit_litI*:
  ⟦*l*$\in$*C*; *sem_clause′* (*C*$-${*l*}) *A = Some False*; *sem_lit′ l A = None*⟧
  $\Longrightarrow$ *is_unit_lit A C l*
  ⟨*proof*⟩

lemma *is_unit_clauseI*: *is_unit_lit A C l* $\Longrightarrow$ *is_unit_clause A C*
⟨*proof*⟩

lemma *unit_other_false*:
assumes *is_unit_lit A C l*
assumes $l' \in C$ $l \neq l'$
shows *sem_lit′ l′ A* = *Some False*
⟨*proof*⟩

lemma *unit_clause_sem′*: *is_unit_lit A C l* $\Longrightarrow$ *sem_clause′ C A* = *None*
⟨*proof*⟩

lemma *unit_clause_assign_dec*:
is_unit_lit A C l $\Longrightarrow$ *sem_clause′ C* (*assign_lit A l*) = *Some True*
⟨*proof*⟩

lemma *unit_clause_sem*: *is_unit_clause A C* $\Longrightarrow$ *sem_clause′ C A* = *None*
⟨*proof*⟩

lemma *sem_not_unit_clause*: *sem_clause′ C A* $\neq$ *None* $\Longrightarrow$ ¬*is_unit_clause A C*
⟨*proof*⟩

lemma *unit_contains_no_true*:
assumes *is_unit_clause A C*
assumes $l \in C$
shows *sem_lit′ l A* $\neq$ *Some True*
⟨*proof*⟩

lemma *two_nfalse_not_unit*:
assumes $l1 \in C$ **and** $l2 \in C$ **and** $l1 \neq l2$
assumes *sem_lit′ l1 A* $\neq$ *Some False* **and** *sem_lit′ l2 A* $\neq$ *Some False*
shows ¬*is_unit_clause A C*
⟨*proof*⟩

lemma *conflict_clause_assign_indep*:
assumes *sem_clause′ C* (*assign_lit A l*) = *Some False*
assumes *neg_lit l* $\notin$ *C*
shows *sem_clause′ C A* = *Some False*
⟨*proof*⟩

lemma *sem_lit′_assign_undec_conv*:
sem_lit′ l′ (*assign_lit A l*) = *None*
$\longleftrightarrow$ *sem_lit′ l′ A* = *None* ∧ *var_of_lit l* $\neq$ *var_of_lit l′*
⟨*proof*⟩

lemma *unit_clause_assign_indep*:
assumes *is_unit_clause* (*assign_lit A l*) *C*
assumes *neg_lit l* $\notin$ *C*
shows *is_unit_clause A C*
⟨*proof*⟩

lemma *clause_assign_false_cases*[*consumes 1*, *case_names no_lit lit*]:
assumes *sem_clause′ C* (*assign_lit A l*) = *Some False*
obtains *neg_lit l* $\notin$ *C sem_clause′ C A* = *Some False*
| *neg_lit l* $\in$ *C sem_clause′* (*C*−{*neg_lit l*}) *A* = *Some False*
⟨*proof*⟩

lemma *clause_assign_unit_cases*[*consumes 1*, *case_names no_lit lit*]:
assumes *is_unit_clause* (*assign_lit A l*) *C*
obtains *neg_lit l* $\notin$ *C is_unit_clause A C*
| *neg_lit l* $\in$ *C*

⟨*proof*⟩

lemma *sem_clause_ins_assign_not_false*[*simp*]:
sem_clause' (*insert l C*) (*assign_lit A l*) $\neq$ *Some False*
⟨*proof*⟩

lemma *sem_clause_ins_assign_not_unit*[*simp*]:
$\neg$*is_unit_clause* (*assign_lit A l*) (*insert l* C')
⟨*proof*⟩

context
fixes $A :: 'a \rightharpoonup bool$ **and** $\sigma :: 'a \Rightarrow bool$
assumes *C*: *compat_assignment A* σ
begin
lemma *compat_lit*: sem_lit' $l\ A = Some\ v \Longrightarrow sem\_lit\ l\ \sigma = v$
⟨*proof*⟩

lemma *compat_clause*: sem_clause' $C\ A = Some\ v \Longrightarrow sem\_clause\ C\ \sigma = v$
⟨*proof*⟩
end

2.1.1 Models, Equivalence, and Redundancy

definition $models'$ $F\ A \equiv \{\ \sigma.\ compat\_assignment\ A\ \sigma \wedge sem\_cnf\ F\ \sigma\}$
definition sat' $F\ A \equiv models'\ F\ A \neq \{\}$
definition $equiv'$ $F\ A\ A' \equiv models'\ F\ A = models'\ F\ A'$

Alternative definition of models', which may be suited for presentation in paper.

lemma $models'\ F\ A = models\ F \cap Collect\ (compat\_assignment\ A)$
⟨*proof*⟩

lemma *equiv'_refl*[*simp*]: $equiv'\ F\ A\ A$ ⟨*proof*⟩
lemma *equiv'_sym*: $equiv'\ F\ A\ A' \Longrightarrow equiv'\ F\ A'\ A$
⟨*proof*⟩
lemma *equiv'_trans*[*trans*]: $[\![\ equiv'\ F\ A\ B;\ equiv'\ F\ B\ C\ ]\!] \Longrightarrow equiv'\ F\ A\ C$
⟨*proof*⟩

lemma *models_antimono*: $C' \subseteq C \Longrightarrow models'\ C\ A \subseteq models'\ C'\ A$
⟨*proof*⟩

lemma *conflict_clause_imp_no_models*:
$[\![\ C \in F;\ is\_conflict\_clause\ A\ C\ ]\!] \Longrightarrow models'\ F\ A = \{\}$
⟨*proof*⟩

lemma *sat'_empty_iff*[*simp*]: $sat'\ F\ Map.empty = sat\ F$
⟨*proof*⟩

lemma *sat'_antimono*: $F \subseteq F' \Longrightarrow sat'\ F'\ A \Longrightarrow sat'\ F\ A$
⟨*proof*⟩

lemma *sat'_equiv*: $equiv'\ F\ A\ A' \Longrightarrow sat'\ F\ A = sat'\ F\ A'$
⟨*proof*⟩

lemma *sat_iff_sat'*: $sat\ F \longleftrightarrow (\exists A.\ sat'\ F\ A)$
⟨*proof*⟩

definition $implied\_clause\ F\ A\ C \equiv models'\ (insert\ C\ F)\ A = models'\ F\ A$
definition $redundant\_clause\ F\ A\ C$
$\equiv (models'\ (insert\ C\ F)\ A = \{\}) \longleftrightarrow (models'\ F\ A = \{\})$

lemma *redundant_clause_alt*: $redundant\_clause\ F\ A\ C \longleftrightarrow sat'\ (insert\ C\ F)\ A = sat'\ F\ A$

$\langle proof \rangle$

lemma *redundant_clauseI*[*intro?*]:
assumes $\bigwedge\sigma.\ [\![compat\_assignment\ A\ \sigma;\ sem\_cnf\ F\ \sigma]\!]$
$\Longrightarrow \exists\sigma'.\ compat\_assignment\ A\ \sigma' \wedge sem\_clause\ C\ \sigma' \wedge sem\_cnf\ F\ \sigma'$
shows *redundant_clause F A C*
$\langle proof \rangle$

lemma *implied_clauseI*[*intro?*]:
assumes $\bigwedge\sigma.\ [\![compat\_assignment\ A\ \sigma;\ sem\_cnf\ F\ \sigma]\!] \Longrightarrow sem\_clause\ C\ \sigma$
shows *implied_clause F A C*
$\langle proof \rangle$

lemma *implied_is_redundant*: $implied\_clause\ F\ A\ C \Longrightarrow redundant\_clause\ F\ A\ C$
$\langle proof \rangle$

lemma *add_redundant_sat_iff*[*simp*]:
$redundant\_clause\ F\ A\ C \Longrightarrow sat'\ (insert\ C\ F)\ A = sat'\ F\ A$
$\langle proof \rangle$

lemma *true_clause_implied*:
$sem\_clause'\ C\ A = Some\ True \Longrightarrow implied\_clause\ F\ A\ C$
$\langle proof \rangle$

lemma *equiv'_map_empty_sym*:
$NO\_MATCH\ Map.empty\ A \Longrightarrow equiv'\ F\ Map.empty\ A \longleftrightarrow equiv'\ F\ A\ Map.empty$
$\langle proof \rangle$

lemma *tautology*: $[\![l{\in}C;\ neg\_lit\ l \in C]\!] \Longrightarrow sem\_clause\ C\ \sigma$
$\langle proof \rangle$

lemma *implied_taut*: $[\![l{\in}C;\ neg\_lit\ l \in C]\!] \Longrightarrow implied\_clause\ F\ A\ C$
$\langle proof \rangle$

definition $is\_syn\_taut\ C \equiv C \cap neg\_lit\ `\ C \neq \{\}$
definition $is\_blocked\ A\ C \equiv sem\_clause'\ C\ A = Some\ True \vee is\_syn\_taut\ C$
lemma *is_blocked_alt*:
$is\_blocked\ A\ C \longleftrightarrow sem\_clause'\ C\ A = Some\ True \vee C \cap neg\_lit\ `\ C \neq \{\}$
$\langle proof \rangle$

lemma *is_syn_taut_empty*[*simp*]: $\neg is\_syn\_taut\ \{\}$
$\langle proof \rangle$

lemma *is_syn_taut_conv*: $is\_syn\_taut\ C \longleftrightarrow (\exists l.\ l{\in}C \wedge neg\_lit\ l \in C)$
$\langle proof \rangle$

lemma *empty_not_blocked*[*simp*]: $\neg is\_blocked\ A\ \{\}$
$\langle proof \rangle$

lemma *is_blocked_insert_iff*:
$is\_blocked\ A\ (insert\ l\ C)$
$\longleftrightarrow is\_blocked\ A\ C \vee sem\_lit'\ l\ A = Some\ True \vee neg\_lit\ l \in C$
$\langle proof \rangle$

lemma *is_blockedI1*: $[\![l{\in}C;\ sem\_lit'\ l\ A = Some\ True]\!] \Longrightarrow is\_blocked\ A\ C$
$\langle proof \rangle$

lemma *is_blockedI2*: $[\![l{\in}C;\ neg\_lit\ l \in C]\!] \Longrightarrow is\_blocked\ A\ C$
$\langle proof \rangle$

lemma *syn_taut_true*[*simp*]: *is_syn_taut* $C \Longrightarrow$ *sem_clause* $C\ \sigma = $ *True*
⟨*proof*⟩

lemma *syn_taut_imp_blocked*: *is_syn_taut* $C \Longrightarrow$ *is_blocked* $A\ C$
⟨*proof*⟩

lemma *blocked_redundant*: *is_blocked* $A\ C \Longrightarrow$ *redundant_clause* $F\ A\ C$
⟨*proof*⟩

lemma *blocked_clause_true*:
⟦*is_blocked* $A\ C$; *compat_assignment* $A\ \sigma$⟧ $\Longrightarrow$ *sem_clause* $C\ \sigma$
⟨*proof*⟩

2.2 Unit Propagation

lemma *unit_propagation*:
assumes $C{\in}F$
assumes *UNIT*: *is_unit_lit* $A\ C\ l$
shows *equiv'* $F\ A$ (*assign_lit* $A\ l$)
⟨*proof*⟩

inductive-set *prop_unit_R* :: *'a cnf* $\Rightarrow$ ((*'a*⇀*bool*) $\times$ (*'a*⇀*bool*)) *set* **for** F
where
step: ⟦ $C{\in}F$; *is_unit_lit* $A\ C\ l$ ⟧ $\Longrightarrow$ (A,*assign_lit* $A\ l$)∈*prop_unit_R* F

lemma *prop_unit_R_Domain*[*simp*]:
$A \in$ *Domain* (*prop_unit_R* F) $\longleftrightarrow$ ($\exists\, C{\in}F$. *is_unit_clause* $A\ C$)
⟨*proof*⟩

lemma *prop_unit_R_equiv*:
assumes $(A,A')\in$(*prop_unit_R* F)*
shows *equiv'* $F\ A\ A'$
⟨*proof*⟩

lemma *wf_prop_unit_R*: *finite* $F \Longrightarrow$ *wf* ((*prop_unit_R* F)$^{-1}$)
⟨*proof*⟩

2.3 RUP and RAT Criteria

RAT-criterion to check for a redundant clause: Pick a *resolution literal* l from the clause, which is not assigned to false, and then check that all resolvents of the clause are implied clauses.

Note: We include l in the resolvents here, as drat-trim does.

lemma *abs_rat_criterion*:
assumes *LIC*: $l{\in}C$
assumes *NFALSE*: *sem_lit'* $l\ A \neq$ *Some False*
assumes *CANDS*: $\forall\, D{\in}F$. *neg_lit* $l \in D$
$\longrightarrow$ *implied_clause* $F\ A$ ($C \cup$ ($D -$ {*neg_lit* l}))
shows *redundant_clause* $F\ A\ C$
⟨*proof*⟩

lemma *abs_rat_criterion'*:
assumes *RAT*: $\exists\, l{\in}C$.
sem_lit' $l\ A \neq$ *Some False*
$\wedge$ ($\forall\, D{\in}F$. *neg_lit* $l \in D \longrightarrow$ *implied_clause* $F\ A$ ($C \cup$ ($D-$ {*neg_lit* l})))
shows *redundant_clause* $F\ A\ C$
⟨*proof*⟩

Assign all literals of clause to false.

definition *and_not_C* $A\ C \equiv \lambda v$.

$\textit{if Pos } v \in C \textit{ then Some False else if Neg } v \in C \textit{ then Some True else } A\ v$

lemma *compat_and_not_C*:
assumes *compat_assignment A* σ
assumes $\neg$*sem_clause C* σ
shows *compat_assignment* (*and_not_C A C*) σ
⟨*proof*⟩

lemma *and_not_empty*[*simp*]: *and_not_C A* {} = *A*
⟨*proof*⟩

lemma *and_not_insert_None*: *sem_lit′ l* (*and_not_C A C*) = *None*
$\Longrightarrow$ *and_not_C A* (*insert l C*) = *assign_lit* (*and_not_C A C*) (*neg_lit l*)
⟨*proof*⟩

lemma *and_not_insert_False*: *sem_lit′ l* (*and_not_C A C*) = *Some False*
$\Longrightarrow$ *and_not_C A* (*insert l C*) = *and_not_C A C*
⟨*proof*⟩

lemma *sem_lit_and_not_C_conv*: *sem_lit′ l* (*and_not_C A C*) = *Some v* $\longleftrightarrow$ (
($l \notin C \wedge$ *neg_lit* $l \notin C \wedge$ *sem_lit′ l A* = *Some v*)
$\vee$ ($l \in C \wedge$ *neg_lit* $l \notin C \wedge v$=*False*)
$\vee$ ($l \notin C \wedge$ *neg_lit* $l \in C \wedge v$=*True*)
$\vee$ ($l \in C \wedge$ *neg_lit* $l \in C \wedge v$=($\neg$*is_pos l*))
)
⟨*proof*⟩

lemma *sem_lit_and_not_C_None_conv*: *sem_lit′ l* (*and_not_C A C*) = *None* $\longleftrightarrow$
sem_lit′ l A = *None* $\wedge\ l \notin C \wedge$ *neg_lit* $l \notin C$
⟨*proof*⟩

Check for implied clause by RUP: If the clause is not blocked, assign all literals of the clause to false, and search for an equivalent assignment (usually by unit-propagation), which has a conflict.

lemma *one_step_implied*:
assumes *RC*: $\neg$*is_blocked A C* $\Longrightarrow$
$\exists A_1$. *equiv′ F* (*and_not_C A C*) $A_1 \wedge$ ($\exists E \in F$. *is_conflict_clause* A_1 *E*)
shows *implied_clause F A C*
⟨*proof*⟩

The unit-propagation steps of ($\neg$ *is_blocked ?A ?C* $\Longrightarrow \exists A_1$. *equiv′ ?F* (*and_not_C ?A ?C*) $A_1 \wedge$ ($\exists E \in$*?F*. *sem_clause′ E* A_1 = *Some False*)) $\Longrightarrow$ *implied_clause ?F ?A ?C* can also be distributed over between the assignments of the negated literals. This is an optimization used for the RAT-check, where an initial set of unit-propagations can be shared between all candidate checks.

lemma *two_step_implied*:
assumes $\neg$*is_blocked A C*
$\Longrightarrow \exists A_1$. *equiv′ F* (*and_not_C A C*) $A_1 \wedge$ ($\neg$*is_blocked* A_1 *D*
$\longrightarrow$ ($\exists A_2$. *equiv′ F* (*and_not_C* A_1 *D*) $A_2 \wedge$ ($\exists E \in F$. *is_conflict_clause* A_2 *E*)))

shows *implied_clause F A* ($C \cup D$)
⟨*proof*⟩

2.4 Old *assign_all_negated* Formulation

definition *assign_all_negated A C* $\equiv$ *let UD* = {$l \in C$. *sem_lit′ l A* = *None*} *in*
A ++ (λl. *if Pos* $l \in$ *UD then Some False*
else if Neg $l \in$ *UD then Some True*
else None)

lemma *abs_rup_criterion*:
assumes *models′ F* (*assign_all_negated A C*) = {}
shows *implied_clause F A C*
⟨*proof*⟩

2.4.1 Properties of *assign_all_negated*

lemma *sem_lit_assign_all_negated_cases*[*consumes 1*, *case_names None Neg Pos*]:
assumes $sem\_lit'\ l\ (assign\_all\_negated\ A\ C) = Some\ v$
obtains $sem\_lit'\ l\ A = Some\ v$
$\mid sem\_lit'\ l\ A = None\ neg\_lit\ l \in C\ v{=}True$
$\mid sem\_lit'\ l\ A = None\ l \in C\ v{=}False$
⟨*proof*⟩

lemma *sem_lit_assign_all_negated_none_iff*:
$sem\_lit'\ l\ (assign\_all\_negated\ A\ C) = None$
$\longleftrightarrow (sem\_lit'\ l\ A = None \wedge l{\notin}C \wedge neg\_lit\ l \notin C)$
⟨*proof*⟩

lemma *sem_lit_assign_all_negated_pres_decided*:
assumes $sem\_lit'\ l\ A = Some\ v$
shows $sem\_lit'\ l\ (assign\_all\_negated\ A\ C) = Some\ v$
⟨*proof*⟩

lemma *sem_lit_assign_all_negated_assign*:
assumes $\forall\, l{\in}C.\ neg\_lit\ l{\notin}C\ l \in C\ sem\_lit'\ l\ A = None$
shows $sem\_lit'\ l\ (assign\_all\_negated\ A\ C) = Some\ False$
⟨*proof*⟩

lemma *sem_lit_assign_all_negated_neqv*:
$sem\_lit'\ l\ (assign\_all\_negated\ A\ C) \neq Some\ v \Longrightarrow sem\_lit'\ l\ A \neq Some\ v$
⟨*proof*⟩

lemma *aan_idem*[*simp*]:
$assign\_all\_negated\ (assign\_all\_negated\ A\ C)\ C = assign\_all\_negated\ A\ C$
⟨*proof*⟩

lemma *aan_dbl*:
assumes $\forall\, l{\in}C{\cup}C'.\ neg\_lit\ l \notin C{\cup}C'$
shows $assign\_all\_negated\ (assign\_all\_negated\ A\ C)\ C'$
$= assign\_all\_negated\ A\ (C{\cup}C')$
⟨*proof*⟩

lemma *aan_mono2*:
$[\![C{\subseteq}C';\ \forall\, l{\in}C'.\ neg\_lit\ l \notin C']\!]$
$\Longrightarrow assign\_all\_negated\ A\ C \subseteq_m assign\_all\_negated\ A\ C'$
⟨*proof*⟩

lemma *aan_empty*[*simp*]: $assign\_all\_negated\ A\ \{\} = A$
⟨*proof*⟩

lemma *aan_restrict*:
$assign\_all\_negated\ A\ C\ |`\ (-\ var\_of\_lit\ `\ \{l \in C.\ sem\_lit'\ l\ A = None\}) = A$
⟨*proof*⟩

lemma *aan_insert*:
assumes $\forall\, l'{\in}C.\ sem\_lit'\ l'\ A \neq Some\ True \wedge neg\_lit\ l' \notin C$
assumes $sem\_lit'\ l\ A \neq Some\ True \wedge neg\_lit\ l \notin C$
shows $assign\_lit\ (assign\_all\_negated\ A\ C)\ (neg\_lit\ l)$
$= assign\_all\_negated\ A\ (insert\ l\ C)$
⟨*proof*⟩

lemma *aan_insert_set*:
assumes $sem\_lit'\ l\ A \neq None$
shows $assign\_all\_negated\ A\ (insert\ l\ C) = assign\_all\_negated\ A\ C$
⟨*proof*⟩

end

3 Basic Notions for the GRAT Format

theory *Grat_Basic*
imports
Unit_Propagation
Refine_Imperative_HOL.Sepref_ICF_Bindings
Exc_Nres_Monad
DRAT_Misc
Synth_Definition
Dynamic_Array
Array_Map_Default
Parser_Iterator
DRAT_Misc
Automatic_Refinement.Misc
begin

hide-const (**open**) *Word.slice*

lemma *list_set_assn_finite*[*simp*, *intro*]:
⟦*rdomp* (*list_set_assn* (*pure R*)) *s*; *single_valued R*⟧ ⟹ *finite s*
⟨*proof*⟩

lemma *list_set_assn_IS_TO_SORTED_LIST_GA′*[*sepref_gen_algo_rules*]:
⟦*CONSTRAINT* (*IS_PURE IS_LEFT_UNIQUE*) *A*;
CONSTRAINT (*IS_PURE IS_RIGHT_UNIQUE*) *A* ⟧
⟹ *GEN_ALGO* (*return*) (*IS_TO_SORTED_LIST* (λ_ _. *True*) (*list_set_assn A*) *A*)
⟨*proof*⟩

3.1 Input Parser

locale *input_pre* =
iterator it_invar′ it_next it_peek
for *it_invar′ it_next* **and** *it_peek* :: *′it*::*linorder* ⇒ *int* +
fixes
it_end :: *′it*

begin
definition *it_invar it* ≡ *itran it it_end*
lemma *it_invar_imp′*[*simp*, *intro*]: *it_invar it* ⟹ *it_invar′ it*
⟨*proof*⟩
lemma *it_invar_imp_ran*[*simp*, *intro*]: *it_invar it* ⟹ *itran it it_end*
⟨*proof*⟩
lemma *itran_invarD*: *itran it it_end* ⟹ *it_invar it*
⟨*proof*⟩
lemma *itran_invarI*: ⟦*itran it it′*; *it_invar it′*⟧ ⟹ *it_invar it*
⟨*proof*⟩

end

type-synonym *′it error* = *String.literal* × *int option* × *′it option*

locale *input* = *input_pre it_invar′ it_next it_peek it_end*
for *it_invar′*::*′it*::*linorder* ⇒ _ **and** *it_next it_peek it_end* +
assumes
it_end_invar[*simp*, *intro*!]: *it_invar it_end*
begin

definition WF ≡ { (it_next it, it) | it. it_invar it ∧ it ≠ it_end}
lemma wf_WF[simp, intro!]: wf WF
⟨proof⟩

lemmas wf_WF_trancl[simp, intro!] = wf_trancl[OF wf_WF]

lemma it_next_invar[simp, intro!]:
⟦ it_invar it; it ≠ it_end ⟧ ⟹ it_invar (it_next it)
⟨proof⟩

lemma it_next_wf[simp, intro]:
⟦ it_invar it; it ≠ it_end ⟧ ⟹ (it_next it, it) ∈ WF
⟨proof⟩

lemma seg_wf[simp, intro]: ⟦seg it l it'; it_invar it'⟧ ⟹ (it',it)∈ WF^*
⟨proof⟩

lemma lz_string_wf[simp, intro]:
⟦lz_string 0 it l ita; it_invar ita⟧ ⟹ (ita, it) ∈ WF^+
⟨proof⟩

Some abbreviations to conveniently construct error messages.

abbreviation mk_err :: String.literal ⇒ 'it error
where mk_err msg ≡ (msg, None, None)
abbreviation mk_errN :: String.literal ⇒ _ ⇒ 'it error
where mk_errN msg n ≡ (msg, Some (int n), None)
abbreviation mk_errI :: _ ⇒_ ⇒ 'it error
where mk_errI msg i ≡ (msg, Some i, None)
abbreviation mk_errit :: _ ⇒ _ ⇒ 'it error
where mk_errit msg it ≡ (msg, None, Some it)
abbreviation mk_errNit :: _ ⇒ _ ⇒_ ⇒ 'it error
where mk_errNit msg n it ≡ (msg, Some (int n), Some it)
abbreviation mk_errIit :: _ ⇒ _ ⇒_ ⇒ 'it error
where mk_errIit msg i it ≡ (msg, Some i, Some it)

Check that iterator has not reached the end.

definition check_not_end it
≡ CHECK (it ≠ it_end) (mk_err STR ''Parsed beyond end'')

lemma check_not_end_correct[THEN ESPEC_trans, refine_vcg]:
it_invar it ⟹ check_not_end it ≤ ESPEC (λ_. True) (λ_. it ≠ it_end)
⟨proof⟩

Skip one element.

definition skip it ≡ doE {
EASSERT (it_invar it);
check_not_end it;
ERETURN (it_next it)
}

Read a literal

definition parse_literal it ≡ doE {
EASSERT(it_invar it ∧ it ≠ it_end ∧ it_peek it ≠ litZ);
ERETURN (lit_α (it_peek it), it_next it)
}

Read an integer

definition parse_int it ≡ doE {
EASSERT (it_invar it);
check_not_end it;
ERETURN (it_peek it, it_next it)

}

Read a natural number

definition *parse_nat* it_0 ≡ *doE* {
  (x,it) ← *parse_int* it_0;
  CHECK $(x \geq 0)$ (*mk_errIit STR* ''*Invalid nat*'' *x* it_0);
  ERETURN (*nat x*,*it*)
}

lemma *parse_literal_spec*[*THEN ESPEC_trans*,*refine_vcg*]:
  ⟦*it_invar it*; *it* ≠ *it_end*; *it_peek it* ≠ *litZ*⟧
  ⟹ *parse_literal it*
    ≤ *ESPEC* (λ_. *True*) (λ(*l*,*it'*). *it_invar it'* ∧ (*it'*,*it*)∈WF^+)
  ⟨*proof*⟩

lemma *skip_spec*[*THEN ESPEC_trans*,*refine_vcg*]:
  ⟦*it_invar it*⟧
  ⟹ *skip it* ≤ *ESPEC* (λ_. *True*) (λ*it'*. *it_invar it'* ∧ (*it'*,*it*)∈WF^+)
  ⟨*proof*⟩

lemma *parse_int_spec*[*THEN ESPEC_trans*,*refine_vcg*]:
  ⟦*it_invar it*⟧
  ⟹ *parse_int it* ≤ *ESPEC* (λ_. *True*) (λ(*x*,*it'*). *it_invar it'* ∧ (*it'*,*it*)∈WF^+)
  ⟨*proof*⟩

lemma *parse_nat_spec*[*THEN ESPEC_trans*,*refine_vcg*]:
  ⟦*it_invar it*⟧
  ⟹ *parse_nat it* ≤ *ESPEC* (λ_. *True*) (λ(*x*,*it'*). *it_invar it'* ∧ (*it'*,*it*)∈WF^+)
  ⟨*proof*⟩

We inline many of the specifications on breaking down the exception monad

lemmas [*enres_inline*] = *check_not_end_def skip_def parse_literal_def*
  parse_int_def parse_nat_def

end

3.2 Implementation

3.2.1 Literals

definition *lit_rel* ≡ *br lit_α lit_invar*
abbreviation *lit_assn* ≡ *pure lit_rel*

interpretation *lit_dflt_option*: *dflt_option pure lit_rel 0 return oo* (=)
  ⟨*proof*⟩
  applyS *sep_auto*
  ⟨*proof*⟩

lemma *neg_lit_refine*[*sepref_import_param*]:
  (*uminus*, *neg_lit*) ∈ *lit_rel* → *lit_rel*
  ⟨*proof*⟩

lemma *lit_α_refine*[*sepref_import_param*]:
  (λ*x*. *x*, *lit_α*) ∈ $[\lambda x.\ x \neq 0]_f$ *int_rel* → *lit_rel*
  ⟨*proof*⟩

3.2.2 Assignment

definition *vv_rel* ≡ {(*1::nat*, *False*), (*2*, *True*)}

definition *assignment_assn* ≡ *amd_assn 0 id_assn* (*pure vv_rel*)

lemmas [*safe_constraint_rules*] = *CN_FALSEI*[*of is_pure assignment_assn*]
type-synonym *i_assignment* = (*nat*,*bool*) *i_map*

lemmas [*intf_of_assn*]
= *intf_of_assnI*[**where** *R*=*assignment_assn* **and** *'a*=(*nat*,*bool*) *i_map*]

sepref-decl-op *lit_is_true*: λ(*l*::*nat literal*) *A*. *sem_lit' l A* = *Some True*
:: (*Id*::(*nat literal*×_) *set*) → ⟨*nat_rel*,*bool_rel*⟩*map_rel* → *bool_rel* ⟨*proof*⟩

sepref-decl-op *lit_is_false*: λ(*l*::*nat literal*) *A*. *sem_lit' l A* = *Some False*
:: (*Id*::(*nat literal*×_) *set*) → ⟨*nat_rel*,*bool_rel*⟩*map_rel* → *bool_rel* ⟨*proof*⟩

sepref-decl-op (*no_def*)
assign_lit :: _ ⇒ *nat literal* ⇒ _
:: ⟨*nat_rel*,*bool_rel*⟩*map_rel* → (*Id*::(*nat literal*×_) *set*)
→ ⟨*nat_rel*,*bool_rel*⟩*map_rel* ⟨*proof*⟩

sepref-decl-op
unset_lit: λ(*A*::*nat*⇀*bool*) *l*. *A*(*var_of_lit l* := *None*)
:: ⟨*nat_rel*,*bool_rel*⟩*map_rel* → (*Id*::(*nat literal*×_) *set*)
→ ⟨*nat_rel*,*bool_rel*⟩*map_rel* ⟨*proof*⟩

lemma [*def_pat_rules*]:
(=)\$(*sem_lit'*\$*l*\$*A*)\$(*Some*\$*True*) ≡ *op_lit_is_true*\$*l*\$*A*
(=)\$(*sem_lit'*\$*l*\$*A*)\$(*Some*\$*False*) ≡ *op_lit_is_false*\$*l*\$*A*
⟨*proof*⟩

lemma *lit_eq_impl*[*sepref_import_param*]:
((=),(=)) ∈ *lit_rel* → *lit_rel* → *bool_rel*
⟨*proof*⟩

lemma *var_of_lit_refine*[*sepref_import_param*]:
(*nat o abs*,*var_of_lit*) ∈ *lit_rel* → *nat_rel*
⟨*proof*⟩

lemma *is_pos_refine*[*sepref_import_param*]:
(λ*x*. *x*>*0*, *is_pos*) ∈ *lit_rel* → *bool_rel*
⟨*proof*⟩

lemma *op_lit_is_true_alt*: *op_lit_is_true l A* = (*let*
x = *A* (*var_of_lit l*);
p = *is_pos l*
in
if x = *None then False*
else (*p* ∧ *the x* = *True* ∨ ¬*p* ∧ *the x* = *False*)
)
⟨*proof*⟩

lemma *op_lit_is_false_alt*: *op_lit_is_false l A* = (*let*
x = *A* (*var_of_lit l*);
p = *is_pos l*
in
if x = *None then False*
else (*p* ∧ *the x* = *False* ∨ ¬*p* ∧ *the x* = *True*)
)
⟨*proof*⟩

definition [*simp*,*code_unfold*]: *vv_eq_bool x y* ≡ *y*⟷*x*=*2*

lemma [*sepref_opt_simps*]:
vv_eq_bool x True ⟷ *x*=*2*

vv_eq_bool x False ⟷ *x≠2*
⟨*proof*⟩

lemma *vv_bool_eq_refine*[*sepref_import_param*]:
(*vv_eq_bool*, (=)) ∈ *vv_rel* → *bool_rel* → *bool_rel*
⟨*proof*⟩

sepref-definition *op_lit_is_true_impl* **is** *uncurry* (*RETURN oo op_lit_is_true*)
:: (*pure lit_rel*)^k $*_a$ *assignment_assn*^{k} $\rightarrow_a$ *bool_assn*
⟨*proof*⟩

sepref-definition *op_lit_is_false_impl* **is** *uncurry* (*RETURN oo op_lit_is_false*)
:: (*pure lit_rel*)^k $*_a$ *assignment_assn*^{k} $\rightarrow_a$ *bool_assn*
⟨*proof*⟩

definition [*simp*]: *b2vv_conv b* ≡ *b*
definition [*code_unfold*]: *b2vv_conv_impl b* ≡ *if b then 2 else 1*::*nat*

lemma *b2vv_conv_impl_refine*[*sepref_import_param*]:
(*b2vv_conv_impl*,*b2vv_conv*) ∈ *bool_rel* → *vv_rel*
⟨*proof*⟩

lemma *vv_unused0*[*safe_constraint_rules*]: (*is_unused_elem 0*) (*pure vv_rel*)
⟨*proof*⟩

sepref-definition *assign_lit_impl*
is *uncurry* (*RETURN oo assign_lit*)
:: *assignment_assn*^{d} $*_a$ (*pure lit_rel*)^k $\rightarrow_a$ *assignment_assn*
⟨*proof*⟩

term *op_unset_lit*
sepref-definition *unset_lit_impl*
is *uncurry* (*RETURN oo op_unset_lit*)
:: *assignment_assn*^{d} $*_a$ (*pure lit_rel*)^k $\rightarrow_a$ *assignment_assn*
⟨*proof*⟩

sepref-definition *unset_var_impl*
is *uncurry* (*RETURN oo op_map_delete*)
:: (*pure nat_rel*)^k $*_a$ *assignment_assn*^{d} $\rightarrow_a$ *assignment_assn*
⟨*proof*⟩

sepref-definition *assignment_empty_impl* **is** *uncurry0* (*RETURN op_map_empty*)
:: *unit_assn*^{k} $\rightarrow_a$ *assignment_assn*
⟨*proof*⟩

lemma *assignment_assn_id_map_rel_fold*:
hr_comp assignment_assn (⟨*nat_rel*, *bool_rel*⟩*map_rel*) = *assignment_assn*
⟨*proof*⟩

context
notes [*fcomp_norm_unfold*] = *assignment_assn_id_map_rel_fold*
begin
sepref-decl-impl *op_lit_is_true_impl.refine* ⟨*proof*⟩
sepref-decl-impl *op_lit_is_false_impl.refine* ⟨*proof*⟩
sepref-decl-impl *assign_lit_impl.refine* ⟨*proof*⟩
sepref-decl-impl *unset_lit_impl.refine* ⟨*proof*⟩
sepref-decl-impl *unset_var_impl.refine*
uses *op_map_delete.fref*[**where** *K*=*Id* **and** *V*=*Id*] ⟨*proof*⟩
sepref-decl-impl (*no_register*) *assignment_empty*: *assignment_empty_impl.refine*
uses *op_map_empty.fref*[**where** *K*=*Id* **and** *V*=*Id*] ⟨*proof*⟩
end

definition [*simp*]: *op_assignment_empty* ≡ *op_map_empty*
interpretation *assignment*: *map_custom_empty op_assignment_empty*
⟨*proof*⟩
lemmas [*sepref_fr_rules*] = *assignment_empty_hnr*[*folded op_assignment_empty_def*]

3.2.3 Clause Database

type-synonym *clausedb2* = *int list*

locale *DB2_def_loc* =
fixes *DB* :: *clausedb2*
fixes *frml_end* :: *nat*
begin
lemmas *amtx_pats*[*pat_rules del*]
sublocale *liti*: *array_iterator DB* ⟨*proof*⟩

lemmas *liti.a_assn_rdompD*[*dest*!]

abbreviation *error_assn*
≡ *id_assn* $\times_a$ *option_assn int_assn* $\times_a$ *option_assn liti.it_assn*

end

locale *DB2_loc* = *DB2_def_loc* +
assumes *DB_not_Nil*[*simp*]: *DB* ≠ []
begin
sublocale *input_pre liti.I liti.next liti.peek liti.end*
⟨*proof*⟩

sublocale *input liti.I liti.next liti.peek liti.end*
⟨*proof*⟩

end

3.2.4 Clausemap

definition (**in** −) *abs_cr_register*
:: *'a literal* ⇒ *'id* ⇒ (*'a literal* ⇀ *'id list*) ⇒ (*'a literal* ⇀ *'id list*)
where *abs_cr_register l cid cr* ≡ *case cr l of*
None ⇒ *cr* | *Some s* ⇒ *cr*(*l* ↦ *mbhd_insert cid s*)

type-synonym *creg* = (*nat list option*) *array*

term *int_encode* **term** *int_decode*
term *map_option*

definition *is_creg* :: (*nat literal* ⇀ *nat list*) ⇒ *creg* ⇒ *assn* **where**
is_creg cr a ≡ $\exists_A f$. *is_nff None f a*
∗ ↑(*cr* = *f o int_encode o lit_γ*)

lemmas [*intf_of_assn*]
= *intf_of_assnI*[**where** *R*=*is_creg* **and** *'a*=(*nat literal*,*nat list*) *i_map*]

definition *creg_dflt_size* ≡ *16*::*nat*
definition *creg_empty* :: *creg Heap*
where *creg_empty* ≡ *dyn_array_new_sz None creg_dflt_size*

lemma *creg_empty_rule*[*sep_heap_rules*]: <*emp*> *creg_empty* <*is_creg Map.empty*>
⟨*proof*⟩

definition [*simp*]: *op_creg_empty* ≡ *op_map_empty* :: *nat literal* ⇀ *nat list*
interpretation *creg*: *map_custom_empty op_creg_empty* ⟨*proof*⟩
lemma *creg_empty_hnr*[*sepref_fr_rules*]:
  (*uncurry0 creg_empty*, *uncurry0* (*RETURN op_creg_empty*))
  $\in$ *unit_assn*^{k} $\rightarrow_a$ *is_creg*
  ⟨*proof*⟩

definition *creg_initialize* :: *int* ⇒ *creg* ⇒ *creg Heap* **where**
  creg_initialize l cr = *do* {
    cr ← *array_set_dyn None cr* (*int_encode l*) (*Some* []);
    return cr
  }

lemma *creg_initialize_rule*[*sep_heap_rules*]:
  ⟦ (*i*,*l*)∈*lit_rel* ⟧
  ⟹ <*is_creg cr a*> *creg_initialize i a* <λ*r*. *is_creg* (*cr*(*l* ↦ [])) *r*>$_t$
  ⟨*proof*⟩

definition *creg_register l cid cr* ≡ *do* {
  x ← *array_get_dyn None cr* (*int_encode l*);
  case x of
    None ⇒ *return cr*
  | *Some s* ⇒ *array_set_dyn None cr* (*int_encode l*) (*Some* (*mbhd_insert cid s*))
}

lemma *creg_register_rule*[*sep_heap_rules*]:
  ⟦ (*i*,*l*) ∈ *lit_rel* ⟧
  ⟹ <*is_creg cr a*>
        creg_register i cid a
      <*is_creg* (*abs_cr_register l cid cr*)>$_t$
  ⟨*proof*⟩

lemma *creg_register_hnr*[*sepref_fr_rules*]:
  (*uncurry2 creg_register*, *uncurry2* (*RETURN ooo abs_cr_register*))
    $\in$ (*pure lit_rel*)^k $*_a$ *nat_assn*^{k} $*_a$ *is_creg*^{d} $\rightarrow_a$ *is_creg*
  ⟨*proof*⟩

definition *op_creg_initialize* :: *nat literal* ⇒ (*nat literal* ⇀ *nat list*) ⇒ _
  where [*simp*]: *op_creg_initialize l cr* ≡ *cr*(*l* ↦ [])

lemma *creg_initialize_hnr*[*sepref_fr_rules*]:
  (*uncurry creg_initialize*, *uncurry* (*RETURN oo op_creg_initialize*))
  $\in$ (*pure lit_rel*)^k $*_a$ *is_creg*^{d} $\rightarrow_a$ *is_creg*
  ⟨*proof*⟩

sepref-register *op_creg_initialize*
  :: *nat literal* ⇒ (*nat literal*,*nat list*) *i_map*
      ⇒ (*nat literal*,*nat list*) *i_map*

sepref-register *abs_cr_register* :: *nat literal* ⇒ *nat* ⇒ _
  :: *nat literal* ⇒ *nat* ⇒ (*nat literal*,*nat list*) *i_map*
      ⇒ (*nat literal*,*nat list*) *i_map*

term *op_map_lookup*
definition *op_creg_lookup i a* ≡ *array_get_dyn None a* (*int_encode i*)

lemma *creg_lookup_rule*[*sep_heap_rules*]:
  ⟦ (*i*,*l*) ∈ *lit_rel* ⟧
  ⟹ <*is_creg cr a*> *op_creg_lookup i a* <λ*r*. *is_creg cr a* ∗ ↑(*r* = *cr l*)>
  ⟨*proof*⟩

lemma *creg_lookup_hnr*[*sepref_fr_rules*]:
  (*uncurry op_creg_lookup*, *uncurry* (*RETURN oo op_map_lookup*))
  $\in$ (*pure lit_rel*)^k $*_a$ *is_creg*^{k} $\rightarrow_a$ *option_assn* (*list_assn id_assn*)
  ⟨*proof*⟩

3.2.5 Clause Database

context
  fixes *DB* :: *clausedb2*
  fixes *frml_end* :: *nat*
begin
  definition *item_next it* ≡
    let sz = *DB*!(*it*−*1*) *in*
    if sz > *0* ∧ *nat* (*sz*) + *1* < *it then*
      Some (*it* − *nat* (*sz*) − *1*)
    else
      None

  definition *at_item_end it* ≡ *it* ≤ *frml_end*

  definition *peek_int it* ≡ *DB*!*it*
end

context *DB2_def_loc*
begin
  abbreviation *cm_assn* ≡ *prod_assn* (*amd_assn 0 nat_assn liti.it_assn*) *is_creg*
  type-synonym *i_cm* = (*nat*,*nat*) *i_map* × (*nat literal*, *nat list*) *i_map*

  abbreviation *state_assn* ≡ *nat_assn* $\times_a$ *cm_assn* $\times_a$ *assignment_assn*
  type-synonym *i_state* = *nat* × *i_cm* × *i_assignment*

  definition *item_next_impl a it* ≡ *do* {
    sz ← *Array.nth a* (*it*−*1*);
    if sz > *0* ∧ *nat* (*sz*) + *1* < *it then*
      return (*it* − *nat* (*sz*) − *1*)
    else
      return 0
  }

  lemma *item_next_hnr*[*sepref_fr_rules*]:
    (*uncurry item_next_impl*, *uncurry* (*RETURN oo item_next*))
    $\in$ *liti.a_assn*^{k} $*_a$ *liti.it_assn*^{k} $\rightarrow_a$ *dflt_option_assn 0 liti.it_assn*
    ⟨*proof*⟩

  lemma *at_item_end_hnr*[*sepref_fr_rules*]:
    (*uncurry* (*return oo at_item_end*), *uncurry* (*RETURN oo at_item_end*))
    $\in$ *nat_assn*^{k} $*_a$ *liti.it_assn*^{k} $\rightarrow_a$ *bool_assn*
    ⟨*proof*⟩

end

3.3 Common GRAT Stuff

datatype *item_type* =
    INVALID
  | *UNIT_PROP*
  | *DELETION*
  | *RUP_LEMMA*
  | *RAT_LEMMA*
  | *CONFLICT*

| *RAT_COUNTS*

type-synonym *id* = *nat*

3.3.1 Clause Map

3.3.2 Correctness

The input to the verified part of the checker is an array of integers *DB* and an index *F_end*, such that the range from index *1::'a* (inclusive) to index *F_end* (exclusive) contains the formula in DIMACs format.

The array is represented as a list here.

We phrase an invariant that expressed a valid formula, and a characterization whether the represented formula is satisfiable.

definition *clause_DB_valid DB F_end* ≡
  1 ≤ *F_end* ∧ *F_end* ≤ *length DB*
∧ *F_invar* (*tl* (*take F_end DB*))

definition *clause_DB_sat DB F_end* ≡ *sat* (*F_α* (*tl* (*take F_end DB*)))

definition *verify_sat_spec DB F_end*
≡ *clause_DB_valid DB F_end* ∧ *clause_DB_sat DB F_end*

definition *verify_unsat_spec DB F_end*
≡ *clause_DB_valid DB F_end* ∧ ¬*clause_DB_sat DB F_end*

lemma *verify_sat_spec DB F_end* ⟷ *1* ≤ *F_end* ∧ *F_end* ≤ *length DB* ∧
(*let lst* = *tl* (*take F_end DB*) *in F_invar lst* ∧ *sat* (*F_α lst*))
⟨*proof*⟩

lemma *verify_unsat_spec DB F_end* ⟷ *1* ≤ *F_end* ∧ *F_end* ≤ *length DB* ∧
(*let lst* = *tl* (*take F_end DB*) *in F_invar lst* ∧ ¬*sat* (*F_α lst*))
⟨*proof*⟩

Concise version only using elementary list operations

lemma *clause_DB_valid_concise*: *clause_DB_valid DB F_end* ≡
  1 ≤ *F_end* ∧ *F_end* ≤ *length DB*
∧ (*let lst*=*tl* (*take F_end DB*) *in lst* ≠[] ⟶ *last lst* = *0*)
⟨*proof*⟩

lemma *clause_DB_sat_concise*:
clause_DB_sat DB F_end ≡ ∃σ. *assn_consistent* σ
  ∧ (∀ *C*∈*set* ‘ *set* (*tokenize 0* (*tl* (*take F_end DB*))). ∃ *l*∈*C*. σ *l*)
⟨*proof*⟩

The input describes a satisfiable formula, iff *F_end* is in range, the described DIMACS string is empty or ends with zero, and there exists a consistent assignment such that each clause contains a literal assigned to true.

lemma *verify_sat_spec_concise*:
shows *verify_sat_spec DB F_end* ≡ *1*≤*F_end* ∧ *F_end* ≤ *length DB* ∧ (
  let lst = *tl* (*take F_end DB*) *in*
    (*lst*≠[] ⟶ *last lst* = *0*)
  ∧ (∃σ. *assn_consistent* σ ∧ (∀ *C*∈*set* (*tokenize 0 lst*). ∃ *l*∈*set C*. σ *l*)))
⟨*proof*⟩

The input describes an unsatisfiable formula, iff *F_end* is in range and does not describe the empty DIMACS string, the DIMACS string ends with zero, and there exists no consistent assignment such that every clause contains at least one literal assigned to true.

lemma *verify_unsat_spec_concise*:
verify_unsat_spec DB F_end ≡ *1* < *F_end* ∧ *F_end* ≤ *length DB* ∧ (
  let lst = *tl* (*take F_end DB*) *in*
    last lst = *0*

∧ (∄σ. *assn_consistent σ* ∧ (∀ *C*∈*set* (*tokenize 0 lst*). ∃ *l*∈*set C*. *σ l*)))
⟨*proof*⟩

end
theory *Impl_List_Set_Ndj*
imports
Collections.Refine_Dflt_ICF
Refine_Imperative_HOL.IICF
Refine_Imperative_HOL.Sepref_ICF_Bindings
begin

definition [*simp*]: *ndls_rel* ≡ *br set* (λ_. *True*)
definition *nd_list_set_assn A* ≡ *pure* (*ndls_rel O* ⟨*the_pure A*⟩*set_rel*)

context
notes [*fcomp_norm_unfold*] = *nd_list_set_assn_def*[*symmetric*]
notes [*fcomp_norm_unfold*] = *list_set_assn_def*[*symmetric*]
begin

lemma *ndls_empty_hnr_aux*: ([],*op_set_empty*) ∈ *ndls_rel* ⟨*proof*⟩
sepref-decl-impl (*no_register*) *ndls_empty*: *ndls_empty_hnr_aux*[*sepref_param*] ⟨*proof*⟩

lemma *ndls_is_empty_hnr_aux*: ((=) [], *op_set_is_empty*) ∈ *ndls_rel* → *bool_rel*
⟨*proof*⟩
sepref-decl-impl *ndls_is_empty*: *ndls_is_empty_hnr_aux*[*sepref_param*] ⟨*proof*⟩

lemma *ndls_insert_hnr_aux*: ((#), *op_set_insert*) ∈ *Id* → *ndls_rel* → *ndls_rel*
⟨*proof*⟩

sepref-decl-impl *ndls_insert*: *ndls_insert_hnr_aux*[*sepref_param*] ⟨*proof*⟩

sepref-decl-op *ndls_ls_copy*: λ*x*::′*a set*. *x* :: ⟨*A*⟩*set_rel* → ⟨*A*⟩*set_rel* ⟨*proof*⟩
lemma *op_ndls_ls_copy_hnr_aux*:
(*remdups*, *op_ndls_ls_copy*)∈*ndls_rel* → ⟨*Id*⟩*list_set_rel*
⟨*proof*⟩

sepref-decl-impl *op_ndls_ls_copy_hnr_aux*[*sepref_param*] ⟨*proof*⟩
end

definition [*simp*]: *op_ndls_empty* = *op_set_empty*
interpretation *ndls*: *set_custom_empty return* [] *op_ndls_empty*
⟨*proof*⟩
sepref-register *op_ndls_empty*
lemmas [*sepref_fr_rules*] = *ndls_empty_hnr*[*folded op_ndls_empty_def*]

lemma *fold_ndls_ls_copy*: *x* = *op_ndls_ls_copy x* ⟨*proof*⟩

end

4 Unsat Checker

theory *Unsat_Check_Split_MM*
imports *Impl_List_Set_Ndj Grat_Basic*
begin
// Test for flexible memory management. // Next id can be any free id. // Problem with re-using IDs: // It's expensive to delete ids from collected RAT candidate lists. // Probably resort to filter candidate lists afterwards. // Currently, we use maybe-head insert to update RAT candidate lists. // That is, if we re-use an ID, it may end up with a duplicate

~~entry in candidate list? RL ≠ [...] Try: Use non-distinct list for RAT candidate lists? TODO: Allow memory management on clause-db, by re-using space of deleted clauses? TODO: Declare max-ids in advance?~~

hide-const (**open**) *Word.slice*

This theory provides a formally verified unsat certificate checker.

The checker accepts an integer array whose prefix contains a cnf formula (encoded as a list of null-terminated clauses), and the suffix contains a certificate in the GRAT format.

4.1 Abstract level

```
definition mkp_raw_err :: _ ⇒ _ ⇒ _ ⇒ (nat×'prf) error where
  mkp_raw_err msg I p ≡ (msg, I, p)

locale unsat_input = input it_invar' for it_invar'::'it::linorder ⇒ _ +
  fixes prf_next :: 'prf ⇒ int × 'prf
begin
  abbreviation mkp_err :: _ ⇒ (nat×'prf) error
    where mkp_err msg ≡ mkp_raw_err (msg) None None
  abbreviation mkp_errN :: _ ⇒_ ⇒ (nat×'prf) error
    where mkp_errN msg n ≡ mkp_raw_err (msg) (Some (int n)) None
  abbreviation mkp_errI :: _ ⇒_ ⇒ (nat×'prf) error
    where mkp_errI msg i ≡ mkp_raw_err (msg) (Some i) None

  abbreviation mkp_errprf :: _ ⇒ _ ⇒ (nat×'prf) error
    where mkp_errprf msg prf ≡ mkp_raw_err (msg) None (Some prf)
  abbreviation mkp_errNprf :: _ ⇒ _ ⇒_ ⇒ (nat×'prf) error
    where mkp_errNprf msg n prf ≡ mkp_raw_err (msg) (Some (int n)) (Some prf)
  abbreviation mkp_errIprf :: _ ⇒ _ ⇒_ ⇒ (nat×'prf) error
    where mkp_errIprf msg i prf ≡ mkp_raw_err (msg) (Some i) (Some prf)


  definition parse_prf :: nat × 'prf ⇒ (_,int × (nat × 'prf)) enres
    where parse_prf ≡ λ(fuel,prf). doE {
      CHECK (fuel > 0) (mkp_errprf STR ''Out of fuel'' (fuel,prf));
      let (x,prf) = prf_next prf;
      ERETURN (x,(fuel − 1,prf))
    }

  definition parse_id prf ≡ doE {
    (x,prf) ← parse_prf prf;
    CHECK (x>0) (mkp_errIprf STR ''Invalid id'' x prf);
    ERETURN (nat x,prf)
  }

  definition parse_idZ prf ≡ doE {
    (x,prf) ← parse_prf prf;
    CHECK (x≥0) (mkp_errIprf STR ''Invalid idZ'' x prf);
    ERETURN (nat x,prf)
  }


  definition parse_type prf ≡ doE {
    (v,prf) ← parse_prf prf;
         if v=1 then ERETURN (UNIT_PROP, prf)
    else if v=2 then ERETURN (DELETION, prf)
    else if v=3 then ERETURN (RUP_LEMMA, prf)
    else if v=4 then ERETURN (RAT_LEMMA, prf)
    else if v=5 then ERETURN (CONFLICT, prf)
    else if v=6 then ERETURN (RAT_COUNTS, prf)
    else THROW (mkp_errIprf STR ''Invalid item type'' v prf)
  }
```

definition *parse_prf_literal prf* ≡ *doE* {
  (*i*,*prf*) ← *parse_prf prf*;
  CHECK (*i* ≠ *0*) (*mkp_errprf STR ''Expected literal but found 0'' prf*);
  ERETURN (*lit_α i*, *prf*)
}

definition *parse_prf_literalZ prf* ≡ *doE* {
  (*i*,*prf*) ← *parse_prf prf*;
  if (*i*=*0*) *then ERETURN* (*None*,*prf*)
  else ERETURN (*Some* (*lit_α i*), *prf*)
}

abbreviation *at_end it* ≡ *it* = *it_end*
abbreviation *at_Z it* ≡ *it_peek it* = *litZ*

definition *prfWF* :: ((*nat*×'*prf*) × (*nat*×'*prf*)) *set*
  where *prfWF* ≡ *measure fst*
lemma *wf_prfWF*[*simp*, *intro*!]: *wf prfWF* ⟨*proof*⟩
lemma *wf_prfWFtrcl*[*simp*, *intro*!]: *wf* (*prfWF*$^+$)
  ⟨*proof*⟩

lemma *parse_prf_spec*[*THEN ESPEC_trans*,*refine_vcg*]:
  parse_prf prf ≤ *ESPEC* (λ_. *True*) (λ(_,*prf*'). (*prf*',*prf*)∈*prfWF*$^+$)
  ⟨*proof*⟩

lemma *parse_id_spec*[*THEN ESPEC_trans*,*refine_vcg*]:
  parse_id prf
    ≤ *ESPEC* (λ_. *True*) (λ(*x*,*prf*'). (*prf*',*prf*)∈*prfWF*$^+$ ∧ *x*>*0*)
  ⟨*proof*⟩

lemma *parse_idZ_spec*[*THEN ESPEC_trans*,*refine_vcg*]:
  parse_idZ prf
    ≤ *ESPEC* (λ_. *True*) (λ(*x*,*prf*'). (*prf*',*prf*)∈*prfWF*$^+$)
  ⟨*proof*⟩

lemma *parse_type_spec*[*THEN ESPEC_trans*,*refine_vcg*]:
  parse_type prf
    ≤ *ESPEC* (λ_. *True*) (λ(*x*,*prf*'). (*prf*',*prf*)∈*prfWF*$^+$)
  ⟨*proof*⟩

lemma *parse_prf_literal_spec*[*THEN ESPEC_trans*,*refine_vcg*]:
  parse_prf_literal prf
    ≤ *ESPEC* (λ_. *True*) (λ(_,*prf*'). (*prf*',*prf*)∈*prfWF*$^+$)
  ⟨*proof*⟩

lemma *parse_prf_literalZ_spec*[*THEN ESPEC_trans*,*refine_vcg*]:
  parse_prf_literalZ prf
    ≤ *ESPEC* (λ_. *True*) (λ(_,*prf*'). (*prf*',*prf*)∈*prfWF*$^+$)
  ⟨*proof*⟩

end

type-synonym *clausemap* = (*id* ⇀ *var clause*) × (*var literal* ⇀ *id set*)
type-synonym *state* = *clausemap* × (*var* ⇀ *bool*)

definition *cm_invar* ≡ λ(*CM*,*RL*).
    (∀ *C*∈*ran CM*. ¬*is_syn_taut C*)
  ∧ (∀ *l s*. *RL l* = *Some s* ⟶ *s* ⊇ {*i*. ∃ *C*. *CM i* = *Some C* ∧ *l*∈*C*})
definition *cm_F* ≡ λ(*CM*,*RL*). *ran CM*

definition *cm_ids* ≡ λ(*CM*, *RL*). *dom CM*

```
context unsat_input begin

Map Interface
definition resolve_id :: clausemap ⇒ id ⇒ (_,var clause) enres
  where resolve_id ≡ λ(CM,RL) i. doE {
    CHECK (i∈dom CM) (mkp_errN STR ''Invalid clause id'' i);
    ERETURN (the (CM i))
  }

definition remove_id :: id ⇒ clausemap ⇒ (_, clausemap) enres
  where remove_id ≡ λi (CM,RL). ERETURN (CM(i:=None),RL)

definition remove_ids CMRL₀ prf ≡ doE {
  (i,prf) ← parse_idZ prf;
  (CMRL,i,prf) ← EWHILEIT
    (λ(CMRL,i,it).  cm_invar CMRL
              ∧ cm_F CMRL ⊆ cm_F CMRL₀
              ∧ cm_ids CMRL ⊆ cm_ids CMRL₀)
    (λ(_,i,_). i≠0)
    (λ(CMRL,i,prf). doE {
      CMRL ← remove_id i CMRL;
      (i,prf) ← parse_idZ prf;
      ERETURN (CMRL,i,prf)
    }) (CMRL₀,i,prf);
  ERETURN (CMRL,prf)
}


definition add_clause
  :: id ⇒ var clause ⇒ clausemap ⇒ (_, clausemap) enres
  where add_clause ≡ λi C (CM,RL). doE {
    EASSERT (¬is_syn_taut C);
    EASSERT (i∉cm_ids (CM,RL));
    let CM = CM(i ↦ C);
    let RL = (λl. case RL l of
        None ⇒ None
      | Some s ⇒ if l∈C then Some (insert i s) else Some s);
    ERETURN (CM,RL)
  }

definition get_rat_candidates
  :: clausemap ⇒ (var ⇀ bool) ⇒ var literal ⇒ (_,id set) enres
  where
  get_rat_candidates ≡ λ(CM,RL) A l. doE {
    let l = neg_lit l;
    CHECK (RL l ≠ None) (mkp_err STR ''Resolution literal not declared'');
    Get collected candidates
    let cands_raw = the (RL l);
    Filter out deleted, not containing l, and being blocked
    let cands = { i∈cands_raw.
                  ∃ C. CM i = Some C
                    ∧ l∈C ∧ sem_clause' (C − {l}) A ≠ Some True };
    ERETURN cands
  }


lemma resolve_id_correct[THEN ESPEC_trans,refine_vcg]:
  resolve_id CMRL i
    ≤ ESPEC (λ_. i∉dom (fst CMRL)) (λC. C ∈ cm_F CMRL ∧ fst CMRL i = Some C)
  ⟨proof⟩
```

lemma *remove_id_correct*[*THEN ESPEC_trans*,*refine_vcg*]:
  cm_invar CMRL
  $\Longrightarrow$ *remove_id i CMRL*
    $\leq$ *ESPEC*
      (λ_. *False*)
      (λ*CMRL'*. *cm_invar CMRL'*
          $\wedge$ *cm_F CMRL'* $\subseteq$ *cm_F CMRL*
          $\wedge$ *cm_ids CMRL'* $\subseteq$ *cm_ids CMRL*)
  ⟨*proof*⟩
lemma *rtrancl_inv_image_ss*: (*inv_image R f*)* $\subseteq$ *inv_image* (R^*) *f*
⟨*proof*⟩

lemmas *rtrancl_inv_image_ssI* = *rtrancl_inv_image_ss*[*THEN set_mp*]

lemma *remove_ids_correct*[*THEN ESPEC_trans*,*refine_vcg*]:
  ⟦*cm_invar CMRL*⟧
  $\Longrightarrow$ *remove_ids CMRL prf*
    $\leq$ *ESPEC*
      (λ_. *True*)
      (λ(*CMRL'*,*prf'*). *cm_invar CMRL'*
          $\wedge$ *cm_F CMRL'* $\subseteq$ *cm_F CMRL*
          $\wedge$ *cm_ids CMRL'* $\subseteq$ *cm_ids CMRL*
          $\wedge$ (*prf'*,*prf*)$\in$*prfWF*$^+$
)
  ⟨*proof*⟩

lemma *add_clause_correct*[*THEN ESPEC_trans*,*refine_vcg*]:
  ⟦*cm_invar CM*; *i*$\notin$*cm_ids CM*; $\neg$*is_syn_taut C*⟧ $\Longrightarrow$
    add_clause i C CM $\leq$ *ESPEC* (λ_. *False*) (λ*CM'*.
      cm_F CM' = *insert C* (*cm_F CM*)
    $\wedge$ *cm_invar CM'*
    $\wedge$ *cm_ids CM'* = *insert i* (*cm_ids CM*)
)
  ⟨*proof*⟩

definition *rat_candidates CM A reslit*
  $\equiv$ {*i*. $\exists$ *C*. *CM i* = *Some C*
        $\wedge$ *neg_lit reslit* $\in$ *C*
        $\wedge$ $\neg$*is_blocked A* (*C*$-${*neg_lit reslit*})}

lemma *is_syn_taut_mono_aux*: *is_syn_taut* (*C*$-$*X*) $\Longrightarrow$ *is_syn_taut C*
  ⟨*proof*⟩

lemma *get_rat_candidates_correct*[*THEN ESPEC_trans*,*refine_vcg*]:
  ⟦ *cm_invar CM* ⟧
  $\Longrightarrow$ *get_rat_candidates CM A reslit*
    $\leq$ *ESPEC* (λ_. *True*) (λ*r*. *r*=*rat_candidates* (*fst CM*) *A reslit*)
  ⟨*proof*⟩

definition *check_unit_clause A C*
  $\equiv$ *ESPEC* (λ_. $\neg$*is_unit_clause A C*) (λ*l*. *is_unit_lit A C l*)

definition *apply_unit i CM A* $\equiv$ *doE* {
  C $\leftarrow$ *resolve_id CM i*;
  l $\leftarrow$ *check_unit_clause A C*;
  EASSERT (*sem_lit' l A* = *None*);
  ERETURN (*assign_lit A l*)
}

```
definition apply_units CM A prf ≡ doE {
  (i,prf) ← parse_idZ prf;
  (A,i,prf) ← EWHILET
    (λ(A,i,prf). i≠0)
    (λ(A,i,prf). doE {
      A ← apply_unit i CM A;
      (i,prf) ← parse_idZ prf;
      ERETURN (A,i,prf)
    }) (A,i,prf);
  ERETURN (A,prf)
}
```

```
lemma apply_unit_correct[THEN ESPEC_trans, refine_vcg]:
  apply_unit i CM A ≤ ESPEC (λ_. True) (λA'. equiv' (cm_F CM) A A')
  ⟨proof⟩
```

```
lemma apply_units_correct[THEN ESPEC_trans, refine_vcg]:
  apply_units CM A prf
        ≤ ESPEC
            (λ_. True)
            (λ(A',prf'). equiv' (cm_F CM) A A' ∧ (prf',prf)∈prfWF⁺)
  ⟨proof⟩
```

Parse a clause and check that it is not blocked.

```
definition parse_check_blocked A it ≡ doE {EASSERT (it_invar it); ESPEC
  (λ_. True)
  (λ(C,A',it'). (∃l.
        lz_string litZ it l it'
      ∧ it_invar it'
      ∧ C=clause_α l
      ∧ ¬is_blocked A C
      ∧ A' = and_not_C A C))}
```

~~abbreviation pbERR ≡ (STR ''Parsed beyond end'',None,None) :: it error~~

```
definition parse_skip_listZ :: (nat×'prf) ⇒ (_,nat×'prf) enres where
  parse_skip_listZ prf ≡ doE {
    (x,prf) ← parse_prf prf;
    (_,prf) ← EWHILET (λ(x,prf). x≠0) (λ(x,prf). parse_prf prf) (x,prf);
    ERETURN prf
  }
```

```
lemma parse_skip_listZ_correct[THEN ESPEC_trans, refine_vcg]:
  shows parse_skip_listZ prf
          ≤ ESPEC (λ_. True) (λprf'. (prf',prf)∈prfWF⁺)
  ⟨proof⟩
```

Too keep proofs more readable, we extract the logic used to check that a RAT-proof provides an exhaustive list of the expected candidates.

```
definition check_candidates candidates prf check ≡ doE {
  (cand,prf) ← parse_idZ prf;
  (candidates,cand,prf) ← EWHILET
    (λ(_,cand,_). cand≠0)
    (λ(candidates,cand,prf). doE {
      if cand ∈ candidates then doE {
        let candidates = candidates - {cand};
        prf ← check cand prf;
        (cand,prf) ← parse_idZ prf;
        ERETURN (candidates,cand,prf)
      } else doE {
        prf ← parse_skip_listZ prf; Skip over unit propagation
        (_,prf) ← parse_prf prf; Skip over conflict clause
        (cand,prf) ← parse_idZ prf;
```

```
      ERETURN (candidates,cand,prf)
    }
  }) (candidates,cand,prf);

  CHECK (candidates = {}) (mkp_errprf STR ''Too few RAT−candidates in proof'' prf);
  ERETURN prf
}

lemma check_candidates_rule[THEN ESPEC_trans, zero_var_indexes]:
  assumes check_correct: ⋀cand prf.
    ⟦ cand∈candidates ⟧
      ⟹ check cand prf
          ≤ ESPEC (λ_. True) (λprf'. Φ cand ∧ (prf',prf)∈prfWF⁺)
  shows check_candidates candidates prf check
          ≤ ESPEC
              (λ_. True)
              (λprf'. (∀ cand∈candidates. Φ cand) ∧ (prf',prf)∈prfWF⁺)
  ⟨proof⟩
definition check_rup_proof :: state ⇒ 'it ⇒ (nat×'prf) ⇒ (_, state × 'it × (nat×'prf)) enres where
  check_rup_proof ≡ λ(CM,A₀) it prf. doE {
    (i,prf) ← parse_id prf;
    CHECK (i∉cm_ids CM) (mkp_errNprf STR ''Duplicate ID'' i prf);
    (C,A',it) ← parse_check_blocked A₀ it;
    (A',prf) ← apply_units CM A' prf;
    (confl_id,prf) ← parse_id prf;
    confl ← resolve_id CM confl_id;
    CHECK (is_conflict_clause A' confl)
          (mkp_errNprf STR ''Expected conflict clause'' confl_id prf);
    EASSERT (redundant_clause (cm_F CM) A₀ C);
    EASSERT (i ∉ cm_ids CM);
    CM ← add_clause i C CM;
    ERETURN ((CM,A₀),it,prf)
  }

lemma check_rup_proof_correct[THEN ESPEC_trans, refine_vcg]:
  assumes [simp]: s=(CM,A)
  assumes cm_invar CM
  assumes it_invar it
  shows
  check_rup_proof s it prf ≤ ESPEC (λ_. True) (λ((CM',A'),it',prf').
      cm_invar CM'
    ∧ (sat' (cm_F CM) A ⟶ sat' (cm_F CM') A')
    ∧ (it_invar it') ∧ (prf',prf)∈prfWF⁺
  )
  ⟨proof⟩
definition check_rat_proof :: state ⇒ 'it ⇒ (nat × 'prf) ⇒ (_, state × 'it × (nat × 'prf)) enres where
  check_rat_proof ≡ λ(CM,A₀) it prf. doE {
    (reslit,prf) ← parse_prf_literal prf;

    CHECK (sem_lit' reslit A₀ ≠ Some False)
          (mkp_errprf STR ''Resolution literal is false'' prf);
    (i,prf) ← parse_id prf;
    CHECK (i∉cm_ids CM) (mkp_errNprf STR ''Duplicate ID'' i prf);
    ~~(C,it) ← lift_parser parse_clause it;~~
    (C,A',it) ← parse_check_blocked A₀ it;
    CHECK (reslit ∈ C) (mkp_errprf STR ''Resolution literal not in clause'' prf);
    (A',prf) ← apply_units CM A' prf;
    candidates ← get_rat_candidates CM A' reslit;
    prf ← check_candidates candidates prf (λcand_id prf. doE {
      cand ← resolve_id CM cand_id;

      EASSERT (¬is_blocked A' (cand−{neg_lit reslit}));
      let A'' = and_not_C A' (cand−{neg_lit reslit});
```

```
      (A'',prf) ← apply_units CM A'' prf;
      (confl_id,prf) ← parse_id prf;
      confl ← resolve_id CM confl_id;
      CHECK (is_conflict_clause A'' confl)
          (mkp_errprf STR ''Expected conflict clause'' prf);
      EASSERT (implied_clause (cm_F CM) A₀ (C ∪ (cand−{neg_lit reslit})));
      ERETURN prf
    });

    EASSERT (redundant_clause (cm_F CM) A₀ C);
    EASSERT (i ∉ cm_ids CM);
    CM ← add_clause i C CM;
    ERETURN ((CM,A₀),it,prf)
  }
```

```
lemma rat_criterion:
  assumes LIC: reslit∈C
  assumes NFALSE: sem_lit' reslit A ≠ Some False
  assumes EQ1: equiv' (cm_F (CM, RL)) (and_not_C A C) A'
  assumes CANDS: ∀ cand∈rat_candidates CM A' reslit.
                    implied_clause
                      (cm_F (CM,RL))
                      A
                      (C ∪ ((the (CM cand)) − {neg_lit reslit}))
  shows redundant_clause (cm_F (CM,RL)) A C
⟨proof⟩
```

```
lemma check_rat_proof_correct[THEN ESPEC_trans, refine_vcg]:
  assumes [simp]: s=(CM,A)
  assumes cm_invar CM
  assumes it_invar it
  shows
  check_rat_proof s it prf ≤ ESPEC (λ_. True) (λ((CM',A'),it',prf').
      cm_invar CM'
    ∧ (sat' (cm_F CM) A ⟶ sat' (cm_F CM') A')
    ∧ it_invar it' ∧ (prf',prf)∈prfWF⁺
  )
  ⟨proof⟩
      applyS (auto simp: rat_candidates_def)
      ⟨proof⟩
      applyS auto []
      ⟨proof⟩
      applyS (rule CMI)
      ⟨proof⟩
```

```
definition check_item :: state ⇒ 'it ⇒ (nat × 'prf) ⇒ (_, (state × 'it × (nat × 'prf)) option) enres
  where check_item ≡ λ(CM,A) it prf. doE {
    (ty,prf) ← parse_type prf;
    case ty of
      INVALID ⇒ THROW (mkp_err STR ''Invalid item'')
    | UNIT_PROP ⇒ doE {
        (A,prf) ← apply_units CM A prf;
        ERETURN (Some ((CM,A),it,prf))
      }
    | DELETION ⇒ doE {
        (CM,prf) ← remove_ids CM prf;
        ERETURN (Some ((CM,A),it,prf))
      }
    | RUP_LEMMA ⇒ doE {
```

```
      s ← check_rup_proof (CM,A) it prf;
      ERETURN (Some s)
    }
  | RAT_LEMMA ⇒ doE {
      s ← check_rat_proof (CM,A) it prf;
      ERETURN (Some s)
    }
  | CONFLICT ⇒ doE {
      (i,prf) ← parse_id prf;
      C ← resolve_id CM i;
      CHECK (is_conflict_clause A C)
          (mkp_errNprf STR ''Conflict clause has no conflict'' i prf);
      ERETURN None
    }
  | RAT_COUNTS ⇒
      THROW (mkp_errprf STR ''Not expecting rat−counts in the middle of proof'' prf)
}

lemma check_item_correct_pre:
  assumes [simp]: s = (CM,A)
  assumes cm_invar CM
  assumes [simp]: it_invar it
  shows check_item s it prf ≤ ESPEC (λ_. True) (λ
      Some ((CM',A'),it',prf') ⇒
          cm_invar CM'
        ∧ (sat' (cm_F CM) A ⟶ sat' (cm_F CM') A')
        ∧ it_invar it' ∧ (prf',prf)∈prfWF⁺
    | None ⇒ ¬sat' (cm_F CM) A
  )
  ⟨proof⟩
  applyS (refine_vcg; auto)
  applyS (refine_vcg; auto simp: sat'_equiv)
  applyS (refine_vcg; auto simp: sat'_antimono)
  applyS (refine_vcg; auto)
  applyS (refine_vcg; auto)
  applyS (refine_vcg; auto simp: conflict_clause_imp_no_models sat'_def)
  applyS (refine_vcg; auto)
  ⟨proof⟩


lemma check_item_correct[THEN ESPEC_trans, refine_vcg]:
  assumes case s of (CM,A) ⇒ cm_invar CM
  assumes it_invar it
  shows check_item s it prf ≤ ESPEC (λ_. True) (case s of (CM,A) ⇒ (λ
      Some ((CM',A'),it',prf') ⇒
          cm_invar CM'
        ∧ (sat' (cm_F CM) A ⟶ sat' (cm_F CM') A')
        ∧ it_invar it' ∧ (prf',prf)∈prfWF⁺
    | None ⇒ ¬sat' (cm_F CM) A)
  )
  ⟨proof⟩


definition cm_empty :: clausemap where cm_empty ≡ (Map.empty, Map.empty)
lemma cm_empty_invar[simp]: cm_invar cm_empty
  ⟨proof⟩
lemma cm_F_empty[simp]: cm_F cm_empty = {}
  ⟨proof⟩
lemma cm_ids_empty[simp]: cm_ids cm_empty = {}
  ⟨proof⟩

lemma cm_ids_empty_imp_F_empty: cm_ids CM = {} ⟹ cm_F CM = {}
  ⟨proof⟩
```

```
definition read_clause_check_taut itE it A ≡ doE {
  EASSERT (A = Map.empty);
  EASSERT (it_invar it ∧ it_invar itE ∧ itran itE it_end);
  (it',(t,A)) ← parse_lz
    (mkp_err STR ''Parsed beyond end'')
    litZ itE it (λ_. True) (λx (t,A). doE {
      let l = lit_α x;
      if (sem_lit' l A = Some False) then ERETURN (True,A)
      else ERETURN (t,assign_lit A l)
    }) (False,A);

  A ← iterate_lz litZ itE it (λ_. True) (λx A. doE {
    let A = A(var_of_lit (lit_α x) := None);
    ERETURN A
  }) A;

  ERETURN (it',(t,A))
}

lemma clause_assignment_syn_taut_aux:
  ⟦∀ l. (sem_lit' l A = Some True) = (l ∈ C); is_syn_taut C⟧ ⟹ False
  ⟨proof⟩



lemma read_clause_check_taut_correct[THEN ESPEC_trans,refine_vcg]:
  ⟦itran it itE; it_invar itE; A = Map.empty⟧ ⟹
  read_clause_check_taut itE it A
  ≤ ESPEC
      (λ_. True)
      (λ(it',(t,A)). A = Map.empty
                  ∧ (∃ l. lz_string litZ it l it'
                        ∧ itran it' itE
                        ∧ (t = is_syn_taut (clause_α l))))
  ⟨proof⟩
  applyS auto
  applyS (auto simp: is_syn_taut_def)
  applyS (auto simp: assign_lit_def split: if_splits)
  applyS (auto simp: is_syn_taut_def)
  applyS (force simp: sem_lit'_assign_conv split: if_splits)
  applyS (auto)
  applyS (auto simp: itran_ord)
  applyS (auto)
  applyS (auto)
  applyS (auto dest: clause_assignment_syn_taut_aux)
  ⟨proof⟩

definition read_cnf_new
  :: 'it ⇒ 'it ⇒ clausemap ⇒ (_, clausemap) enres
  where read_cnf_new itE it CM ≡ doE {
    (CM,next_id,A) ← tok_fold itE it (λit (CM,next_id,A). doE {
      (it',(t,A)) ← read_clause_check_taut itE it A;
      if t then ERETURN (it', (CM,next_id+1,A))
      else doE {
        EASSERT (∃ l it'. lz_string litZ it l it' ∧ it_invar it');
        let C = clause_α (the_lz_string litZ it);
        CM ← add_clause next_id C CM;
        ERETURN (it',(CM,next_id+1,A))
      }
    }) (CM,1,Map.empty);
    ERETURN (CM)
  }
```

```
lemma read_cnf_new_correct[THEN ESPEC_trans, refine_vcg]:
  ⟦seg it lst itE; cm_invar CM; cm_ids CM = {}; it_invar itE⟧
  ⟹ read_cnf_new itE it CM
  ≤ ESPEC (λ_. True) (λ(CM).
        (lst ≠ [] ⟶ last lst = litZ)
      ∧ cm_invar CM
      ∧ sat (cm_F CM) = sat (set (map clause_α (tokenize litZ lst)))
  )
  ⟨proof⟩
```

```
definition cm_init_lit
  :: var literal ⇒ clausemap ⇒ (_,clausemap) enres
  where cm_init_lit ≡ λl (CM,RL). ERETURN (CM,RL(l ↦ {}))
```

```
lemma cm_init_lit_correct[THEN ESPEC_trans, refine_vcg]:
  ⟦ cm_invar CMRL; cm_ids CMRL = {} ⟧ ⟹
    cm_init_lit l CMRL
    ≤ ESPEC (λ_. False) (λCMRL'. cm_invar CMRL' ∧ cm_ids CMRL' = {})
  ⟨proof⟩
```

```
definition init_rat_counts prf ≡ doE {
  (ty,prf) ← parse_type prf;
  CHECK (ty = RAT_COUNTS) (mkp_errprf STR ''Expected RAT counts item'' prf);

  (l,prf) ← parse_prf_literalZ prf;
  (CM,_,prf) ← EWHILET (λ(CM,l,prf). l≠None) (λ(CM,l,prf). doE {
    EASSERT (l≠None);
    let l = the l;
    (_,prf) ← parse_prf prf;   Just ignoring count, silently assuming it to be >0. TODO: Add count-down and stop
optimization?

    let l = neg_lit l;
    CM ← cm_init_lit l CM;

    (l,prf) ← parse_prf_literalZ prf;
    ERETURN (CM,l,prf)
  }) (cm_empty,l,prf);

  ERETURN (CM,prf)
}
```

```
lemma init_rat_counts_correct[THEN ESPEC_trans, refine_vcg]:
  init_rat_counts prf
    ≤ ESPEC (λ_. True) (λ(CM,prf'). cm_invar CM ∧ cm_ids CM = {} ∧ (prf',prf)∈prfWF⁺)
  ⟨proof⟩
```

```
definition verify_unsat F_begin F_end it prf ≡ doE {
  EASSERT (it_invar it);

  (CM,prf) ← init_rat_counts prf;

  CM ← read_cnf_new F_end F_begin CM;

  let s = (CM,Map.empty);

  EWHILEIT
    (λSome (_,it,_) ⇒ it_invar it | None ⇒ True)
    (λs. s≠None)
    (λs. doE {
      EASSERT (s≠None);
```

```
    let (s,it,prf) = the s;

    EASSERT (it_invar it);

    check_item s it prf
  }) (Some (s,it,prf));

  ERETURN ()
  CHECK (is_None s) (mk_err ''Proof did not contain conflict declaration'')
}
```

lemma *verify_unsat_correct*:

```
  ⟦seg F_begin lst F_end; it_invar F_end; it_invar it⟧ ⟹
    verify_unsat F_begin F_end it prf
    ≤ ESPEC (λ_. True) (λ_. F_invar lst ∧ ¬sat (F_α lst))
  ⟨proof⟩
  applyS (auto)
  applyS (auto simp: F_α_def F_invar_def)
  applyS (clarsimp split: option.splits; auto)
  applyS (auto split!: option.split_asm)
  applyS (auto simp: F_α_def F_invar_def)
  applyS (auto split: option.split_asm)
  applyS (auto split: option.split_asm)
  ⟨proof⟩
```

end — proof parser

4.2 Refinement — Backtracking

type-synonym *bt_assignment* = (*var* ⇀ *bool*) × *var set*

```
definition backtrack A T ≡ A|`(−T)
lemma backtrack_empty[simp]: backtrack A {} = A
  ⟨proof⟩

definition is_backtrack A′ T′ A ≡ T′⊆dom A′ ∧ A = backtrack A′ T′
lemma is_backtrack_empty[simp]: is_backtrack A {} A
  ⟨proof⟩
lemma is_backtrack_not_undec:
  ⟦ is_backtrack A′ T′ A; var_of_lit l∈T′ ⟧ ⟹ sem_lit′ l A′ ≠ None
  ⟨proof⟩

lemma is_backtrack_assignI:
  ⟦is_backtrack A′ T′ A; sem_lit′ l A′ = None; x = var_of_lit l⟧
  ⟹ is_backtrack (assign_lit A′ l) (insert x T′) A
  ⟨proof⟩
```

context *unsat_input* **begin**

```
definition assign_lit_bt ≡ λA T l. doE {
  EASSERT (sem_lit′ l A = None ∧ var_of_lit l ∉ T);
  ERETURN (assign_lit A l, insert (var_of_lit l) T)
}

definition apply_unit_bt i CM A T ≡ doE {
  C ← resolve_id CM i;
  l ← check_unit_clause A C;
  assign_lit_bt A T l
}

definition apply_units_bt CM A T prf ≡ doE {
  (i,prf) ← parse_idZ prf;
```

```
  ((A,T),i,prf) ← EWHILET
    (λ((A,T),i,prf). i≠0)
    (λ((A,T),i,prf). doE {
      (A,T) ← apply_unit_bt i CM A T;
      (i,prf) ← parse_idZ prf;
      ERETURN ((A,T),i,prf)
    }) ((A,T),i,prf);
  ERETURN ((A,T),prf)
}

definition parse_check_blocked_bt A it ≡ doE {EASSERT (it_invar it); ESPEC
  (λ_. True ~~∧ (∀e. parse_clause it ≠ Inl e)~~)
  (λ(C,(A',T'),it'). ∃l.
      lz_string litZ it l it'
    ∧ it_invar it'
    ∧ C=clause_α l
    ∧ ¬is_blocked A C
    ∧ A' = and_not_C A C
    ∧ T' = { v. v∈var_of_lit`C ∧ A v = None })}

definition and_not_C_bt A C ≡ doE {
  EASSERT (¬is_blocked A C);
  ERETURN (and_not_C A C, { v. v∈var_of_lit`C ∧ A v = None })
}

definition check_candidates' candidates A prf check ≡ doE {
  (cand,prf) ← parse_idZ prf;
  (candidates,A,cand,prf) ← EWHILET
    (λ(_,_,cand,_). cand≠0)
    (λ(candidates,A,cand,prf). doE {
      if cand ∈ candidates then doE {
        let candidates = candidates − {cand};
        (A,prf) ← check cand A prf;
        (cand,prf) ← parse_idZ prf;
        ERETURN (candidates,A,cand,prf)
      } else doE {
        prf ← parse_skip_listZ prf;
        (_,prf) ← parse_prf prf;
        (cand,prf) ← parse_idZ prf;
        ERETURN (candidates,A,cand,prf)
      }
    }) (candidates,A,cand,prf);

  CHECK (candidates = {}) (mkp_errprf STR ''Too few RAT−candidates in proof'' prf);
  ERETURN (A,prf)
}

lemma check_candidates'_refine_ca[refine]:
  assumes [simplified,simp]: (candidatesi,candidates)∈Id (prfi,prf)∈Id
  assumes [refine]: ⋀candi prfi cand prf A'.
      ⟦(candi,cand)∈Id; (prfi,prf)∈Id; (A',A)∈Id⟧
      ⟹ check' candi A' prfi
            ≤⇓E UNIV ({((A,prf),prf) | prf. True })
          (check cand prf)
  shows check_candidates' candidatesi A prfi check'
          ≤⇓E UNIV {((A,prf),prf) | prf. True }
        (check_candidates candidates prf check)
  ⟨proof⟩

lemma check_candidates'_refine[refine]:
  assumes [simplified,simp]:
    (candidatesi,candidates)∈Id (prfi,prf)∈Id (Ai,A)∈Id
  assumes ERID: Id ⊆ ER
```

```
  assumes [refine]:
    ⋀candi prfi cand prf A′ A. ⟦(candi,cand)∈Id; (prfi,prf)∈Id; (A′,A)∈Id⟧
      ⟹ check′ candi A′ prfi ≤⇓E ER (Id×rId) (check cand A prf)
  shows check_candidates′ candidatesi Ai prfi check′
        ≤⇓E ER (Id×rId) (check_candidates′ candidates A prf check)
  ⟨proof⟩
```

```
definition check_rup_proof_bt :: state ⇒ ′it ⇒ (nat×′prf) ⇒ (_, state × ′it × (nat×′prf)) enres where
  check_rup_proof_bt ≡ λ(CM,A) it prf. doE {
    (i,prf) ← parse_id prf;
    CHECK (i∉cm_ids CM) (mkp_errNprf STR ′′Duplicate ID′′ i prf);
    (C,(A,T),it) ← parse_check_blocked_bt A it;
    ((A,T),prf) ← apply_units_bt CM A T prf;
    (confl_id,prf) ← parse_id prf;
    confl ← resolve_id CM confl_id;
    CHECK (is_conflict_clause A confl)
          (mkp_errNprf STR ′′Expected conflict clause′′ confl_id prf);
    EASSERT (i ∉ cm_ids CM);
    CM ← add_clause i C CM;
    ERETURN ((CM,backtrack A T),it,prf)
  }
```

```
definition check_rat_proof_bt :: state ⇒ ′it ⇒ (nat × ′prf) ⇒ (_,state × ′it × (nat × ′prf)) enres where
  check_rat_proof_bt ≡ λ(CM,A) it prf. doE {
    (reslit,prf) ← parse_prf_literal prf;

    CHECK (sem_lit′ reslit A ≠ Some False)
          (mkp_errprf STR ′′Resolution literal is false′′ prf);
    (i,prf) ← parse_id prf;
    CHECK (i∉cm_ids CM) (mkp_errNprf STR ′′Duplicate ID′′ i prf);
    (C,(A,T),it) ← parse_check_blocked_bt A it;
    CHECK (reslit ∈ C) (mkp_errprf STR ′′Resolution literal not in clause′′ prf);
    ((A,T),prf) ← apply_units_bt CM A T prf;
    candidates ← get_rat_candidates CM A reslit;
    (A,prf) ← check_candidates′ candidates A prf (λcand_id A prf. doE {
      cand ← resolve_id CM cand_id;

      (A,T2) ← and_not_C_bt A (cand−{neg_lit reslit});
      ((A,T2),prf) ← apply_units_bt CM A T2 prf;
      (confl_id,prf) ← parse_id prf;
      confl ← resolve_id CM confl_id;
      CHECK (is_conflict_clause A confl)
            (mkp_errprf STR ′′Expected conflict clause′′ prf);
      ERETURN (backtrack A T2,prf)
    });

    EASSERT (i ∉ cm_ids CM);
    CM ← add_clause i C CM;
    ERETURN ((CM,backtrack A T),it,prf)
  }
```

```
definition bt_assign_rel A
  ≡ { ((A′,T),A′) | A′ T. T ⊆ dom A′ ∧ A = A′|‘(−T) }
definition bt_need_bt_rel A₀
  ≡ br (λ_. A₀) (λ(A′,T′). T′⊆dom A′ ∧ backtrack A′ T′ = A₀)
```

~~definition bt_anccb_rel A₀ ≡ ({(False,False)} ×r Id ×r bt_assign_rel A₀ ×r Id) ∪ ({(True,True)} ×r UNIV ×r bt_need_bt_rel A₀ ×r UNIV)~~

```
lemma bt_rel_simps:
  ((Ai,T),A)∈bt_assign_rel A₀ ⟹ Ai=A ∧ backtrack A T = A₀ ∧ T⊆dom A
```

$((Ai,T),A) \in bt\_need\_bt\_rel\ A_0 \Longrightarrow A = A_0 \wedge backtrack\ Ai\ T = A_0 \wedge T \subseteq dom\ Ai$
⟨proof⟩

lemma $bt\_in\_bta\_rel$: $T \subseteq dom\ A \Longrightarrow ((A,T),A) \in bt\_assign\_rel\ (backtrack\ A\ T)$
⟨proof⟩

lemma $and\_not\_C\_bt\_refine[refine]$: $[\![\neg is\_blocked\ A\ C;\ (Ai,A) \in Id;\ (Ci,C) \in Id]\!]$
$\Longrightarrow and\_not\_C\_bt\ Ai\ Ci \le \Downarrow_E\ UNIV\ (bt\_assign\_rel\ A)\ (ERETURN\ (and\_not\_C\ A\ C))$
⟨proof⟩

lemma $parse\_check\_blocked\_bt\_refine[refine]$: $[\![(Ai,A) \in Id;\ (iti,it) \in Id]\!]$
$\Longrightarrow parse\_check\_blocked\_bt\ Ai\ iti$
$\le \Downarrow_E\ UNIV\ (Id \times_r bt\_assign\_rel\ A \times_r Id)\ (parse\_check\_blocked\ A\ it)$
⟨proof⟩

lemma $assign\_lit\_bt\_refine[refine]$:
$[\![sem\_lit'\ l\ A = None;\ ((Ai,Ti),A) \in bt\_assign\_rel\ A_0;\ (li,l) \in Id]\!]$
$\Longrightarrow assign\_lit\_bt\ Ai\ Ti\ li$
$\le \Downarrow_E\ UNIV\ (bt\_assign\_rel\ A_0)\ (ERETURN\ (assign\_lit\ A\ l))$
⟨proof⟩
applyS *simp*
⟨proof⟩

lemma $apply\_unit\_bt\_refine[refine]$:
$[\![(ii,i) \in Id;\ (CMi,CM) \in Id;\ ((Ai,Ti),A) \in bt\_assign\_rel\ A_0]\!]$
$\Longrightarrow apply\_unit\_bt\ ii\ CMi\ Ai\ Ti$
$\le \Downarrow_E\ UNIV\ (bt\_assign\_rel\ A_0)\ (apply\_unit\ i\ CM\ A)$
⟨proof⟩

lemma $apply\_units\_bt\_refine[refine]$:
$[\![(CMi,CM) \in Id;\ ((Ai,Ti),A) \in bt\_assign\_rel\ A_0;\ (iti,it) \in Id]\!]$
$\Longrightarrow apply\_units\_bt\ CMi\ Ai\ Ti\ iti$
$\le \Downarrow_E\ UNIV\ (bt\_assign\_rel\ A_0 \times_r Id)\ (apply\_units\ CM\ A\ it)$
⟨proof⟩

term *check_rup_proof*
lemma $check\_rup\_proof\_bt\_refine[refine]$:
$[\![(si,s) \in Id;\ (iti,it) \in Id;\ (prfi,prf) \in Id]\!]$
$\Longrightarrow check\_rup\_proof\_bt\ si\ iti\ prfi \le \Downarrow_E\ UNIV\ Id\ (check\_rup\_proof\ s\ it\ prf)$
⟨proof⟩

lemma $check\_rat\_proof\_bt\_refine[refine]$:
$[\![(si,s) \in Id;\ (iti,it) \in Id;\ (prfi,prf) \in Id]\!]$
$\Longrightarrow check\_rat\_proof\_bt\ si\ iti\ prfi \le \Downarrow_E\ UNIV\ Id\ (check\_rat\_proof\ s\ it\ prf)$
⟨proof⟩

definition $check\_item\_bt :: state \Rightarrow 'it \Rightarrow (nat \times 'prf) \Rightarrow (\_, (state \times 'it \times (nat \times 'prf))\ option)\ enres$
where $check\_item\_bt \equiv \lambda(CM,A)\ it\ prf.\ doE\ \{$
$(ty,prf) \leftarrow parse\_type\ prf;$
$case\ ty\ of$
$INVALID \Rightarrow THROW\ (mkp\_err\ STR\ ''Invalid\ item'')$
$|\ UNIT\_PROP \Rightarrow doE\ \{$
$(A,prf) \leftarrow apply\_units\ CM\ A\ prf;$
$ERETURN\ (Some\ ((CM,A),it,prf))$
$\}$
$|\ DELETION \Rightarrow doE\ \{$
$(CM,prf) \leftarrow remove\_ids\ CM\ prf;$
$ERETURN\ (Some\ ((CM,A),it,prf))$
$\}$
$|\ RUP\_LEMMA \Rightarrow doE\ \{$
$s \leftarrow check\_rup\_proof\_bt\ (CM,A)\ it\ prf;$

ERETURN (Some s)
}
| RAT_LEMMA ⇒ doE {
s ← check_rat_proof_bt (CM,A) it prf;
ERETURN (Some s)
}
| CONFLICT ⇒ doE {
(i,prf) ← parse_id prf;
C ← resolve_id CM i;
CHECK (is_conflict_clause A C)
(mkp_errNprf STR ''Conflict clause has no conflict'' i prf);
ERETURN None
}
| RAT_COUNTS ⇒
THROW (mkp_errprf STR ''Not expecting rat−counts in the middle of proof'' prf)
}

lemma *check_item_bt_refine*[*refine*]: ⟦(si,s)∈Id; (iti,it)∈Id; (prfi,prf)∈Id⟧
⟹ check_item_bt si iti prfi ≤⇓$_E$ UNIV Id (check_item s it prf)
⟨proof⟩
applyS *simp*

⟨proof⟩

definition *verify_unsat_bt F_begin F_end it prf* ≡ *doE* {
EASSERT (it_invar it);

(CM,prf) ← init_rat_counts prf;

CM ← read_cnf_new F_end F_begin CM;

let s = (CM,Map.empty);

EWHILEIT
(λSome (_,it,_) ⇒ it_invar it | None ⇒ True)
(λs. s≠None)
(λs.
doE {
EASSERT (s≠None);
let (s,it,prf) = the s;

EASSERT (it_invar it);

check_item_bt s it prf
}) (Some (s,it,prf));
ERETURN ()
~~CHECK (is_None s) (mkp_err ''Proof did not contain conflict declaration'')~~
}

lemma *verify_unsat_bt_refine*[*refine*]:
⟦ (F_begini,F_begin)∈Id; (F_endi,F_end)∈Id; (iti,it)∈Id; (prfi,prf)∈Id ⟧
⟹ verify_unsat_bt F_begini F_endi iti prfi
≤⇓$_E$ UNIV Id (verify_unsat F_begin F_end it prf)
⟨proof⟩

end — proof parser

4.3 Refinement 1

Model clauses by iterators to their starting position

type-synonym ('it) *clausemap1* = (id ⇀ 'it) × (var literal ⇀ id list)
type-synonym ('it) *state1* = ('it) *clausemap1* × (var ⇀ bool)

```
context unsat_input begin

  definition cref_rel
    ≡ { (cref,C). ∃ l it'. lz_string litZ cref l it'
                        ∧ it_invar it'
                        ∧ C = clause_α l}
  definition next_it_rel
    ≡ { (cref,it'). ∃ l. lz_string litZ cref l it' ∧ it_invar it'}

  definition clausemap1_rel
    ≡ (Id → ⟨cref_rel⟩option_rel) ×r (Id → ⟨br set (λ_. True)⟩option_rel)
  abbreviation state1_rel ≡ clausemap1_rel ×r Id

  definition parse_check_clause cref c f s ≡ doE {
    (it,s) ← parse_lz (mkp_err STR ''Parsed beyond end'') litZ it_end cref c (λx s. doE {
      EASSERT (x ≠ litZ);
      let l = lit_α x;
      f l s
    }) s;
    ERETURN (s,it)
  }

  lemma parse_check_clause_rule_aux:
    assumes I[simp]: I {} s
    assumes F_RL:
      ⋀C l s. ⟦I C s; c s⟧ ⟹ f l s ≤ ESPEC (λ_. True) (I (insert l C))
    assumes [simp]: it_invar cref
    shows parse_check_clause cref c f s ≤ ESPEC
      (/TODO: Spec that parsing failed/ λ_. True)
      (λ(s,it'). ∃ C.
          I C s
        ∧ (c s ⟶ it_invar it'
                ∧ (cref,C) ∈ cref_rel
                ∧ (cref,it') ∈ next_it_rel)
      )
    ⟨proof⟩

  lemma parse_check_clause_rule:
    assumes I0: I {} s
    assumes [simp]: it_invar cref
    assumes F_RL:
      ⋀C l s. ⟦I C s; c s⟧ ⟹ f l s ≤ ESPEC (λ_. True) (I (insert l C))
    assumes ⋀C s it'. ⟦I C s; ¬c s ⟧ ⟹ Q (s,it')
    assumes ⋀C s it'.
      ⟦ I C s; c s; (cref,it')∈next_it_rel; (cref,C)∈cref_rel ⟧ ⟹ Q (s,it')
    shows parse_check_clause cref c f s ≤ ESPEC (λ_. True) Q
    ⟨proof⟩
  definition iterate_clause cref c f s ≡
    iterate_lz litZ it_end cref c (λx s. f (lit_α x) s) s

  lemma iterate_clause_rule:
    assumes CR: (cref,C)∈cref_rel
    assumes I0: I {} s
    assumes F_RL: ⋀C1 l s.
      ⟦ I C1 s; C1⊆C; l∈C; c s ⟧ ⟹ f l s ≤ ESPEC E (I (insert l C1))
    assumes T_IMP: ⋀s. ⟦ c s; I C s ⟧ ⟹ P s
    assumes C_IMP: ⋀s C1. ⟦ ¬c s; C1⊆C; I C1 s ⟧ ⟹ P s
    shows iterate_clause cref c f s ≤ ESPEC E P
  ⟨proof⟩
      applyS (simp add: INV itran_ord)
      applyS (simp add: I0)
      applyS (rule F_RL; auto)
      applyS (erule C_IMP; assumption?; auto)
```

applyS (*auto intro*: *T_IMP*)
⟨*proof*⟩

definition *check_unit_clause1 A cref* ≡ *doE* {
ul ← *iterate_clause cref* (λ*ul*. *True*) (λ*l ul*. *doE* {
CHECK (*sem_lit′ l A* ≠ *Some True*)
(*mkp_err STR ′′True literal in clause assumed to be unit′′*);
if (*sem_lit′ l A* = *Some False*) *then ERETURN ul*
else doE {
CHECK (*ul* = *None* ∨ *ul* = *Some l*)
(*mkp_err STR ′′2*−*undec in clause assumed to be unit′′*);
ERETURN (*Some l*)
}
}) *None*;
CHECK (*ul* ≠ *None*) (*mkp_err STR ′′Conflict in clause assumed to be unit′′*);
EASSERT (*ul* ≠ *None*);
ERETURN (*the ul*)
}

lemma *check_unit_clause1_refine*[*refine*]:
assumes [*simplified*, *simp*]: (*Ai*,*A*)∈*Id*
assumes *CR*: (*cref*,*C*)∈*cref_rel*
shows *check_unit_clause1 Ai cref* ≤⇓$_E$ *UNIV Id* (*check_unit_clause A C*)
⟨*proof*⟩

definition *resolve_id1* ≡ λ(*CM*,_) *i*. *doE* {
CHECK (*i*∈*dom CM*) (*mkp_errN STR ′′Invalid clause id′′ i*);
ERETURN (*the* (*CM i*))
}

lemma *resolve_id1_refine*[*refine*]:
⟦ (*CMi*,*CM*)∈*clausemap1_rel*; (*ii*,*i*)∈*Id* ⟧
⟹ *resolve_id1 CMi ii* ≤⇓$_E$ *UNIV cref_rel* (*resolve_id CM i*)
⟨*proof*⟩

definition *apply_unit1_bt i CM A T* ≡ *doE* {
C ← *resolve_id1 CM i*;
l ← *check_unit_clause1 A C*;
assign_lit_bt A T l
}

lemma *apply_unit1_bt_refine*[*refine*]:
⟦ (*ii*,*i*)∈*Id*; (*CMi*,*CM*)∈*clausemap1_rel*; (*Ai*,*A*)∈*Id*; (*Ti*,*T*)∈*Id* ⟧
⟹ *apply_unit1_bt ii CMi Ai Ti* ≤ ⇓$_E$ *UNIV Id* (*apply_unit_bt i CM A T*)
⟨*proof*⟩

definition *apply_units1_bt CM A T prf* ≡ *doE* {
(*i*,*prf*) ← *parse_idZ prf*;
((*A*,*T*),*i*,*prf*) ← *EWHILET*
(λ((*A*,*T*),*i*,*prf*). *i*≠*0*)
(λ((*A*,*T*),*i*,*prf*). *doE* {
(*A*,*T*) ← *apply_unit1_bt i CM A T*;
(*i*,*prf*) ← *parse_idZ prf*;
ERETURN ((*A*,*T*),*i*,*prf*)
}) ((*A*,*T*),*i*,*prf*);
ERETURN ((*A*,*T*),*prf*)
}

lemma *apply_units1_bt_refine*[*refine*]:
⟦ (*CMi*,*CM*)∈*clausemap1_rel*; (*Ai*,*A*)∈*Id*; (*Ti*,*T*)∈*Id*; (*iti*,*it*)∈*Id* ⟧
⟹ *apply_units1_bt CMi Ai Ti iti* ≤ ⇓$_E$ *UNIV Id* (*apply_units_bt CM A T it*)
⟨*proof*⟩

definition *apply_unit1 i CM A* ≡ *doE* {
  C ← *resolve_id1 CM i*;
  l ← *check_unit_clause1 A C*;
  ERETURN (*assign_lit A l*)
}

lemma *apply_unit1_refine*[*refine*]:
  ⟦ (*ii*,*i*)∈*Id*; (*CMi*,*CM*)∈*clausemap1_rel*; (*Ai*,*A*)∈*Id* ⟧
  ⟹ *apply_unit1 ii CMi Ai* ≤ $\Downarrow_E$ *UNIV Id* (*apply_unit i CM A*)
  ⟨*proof*⟩

definition *apply_units1 CM A prf* ≡ *doE* {
  (*i*,*prf*) ← *parse_idZ prf*;
  (*A*,*i*,*prf*) ← *EWHILET*
    (λ(*A*,*i*,*prf*). *i*≠*0*)
    (λ(*A*,*i*,*prf*). *doE* {
      A ← *apply_unit1 i CM A*;
      (*i*,*prf*) ← *parse_idZ prf*;
      ERETURN (*A*,*i*,*prf*)
    }) (*A*,*i*,*prf*);
  ERETURN (*A*,*prf*)
}

lemma *apply_units1_refine*[*refine*]:
  ⟦ (*CMi*,*CM*)∈*clausemap1_rel*; (*Ai*,*A*)∈*Id*; (*iti*,*it*)∈*Id* ⟧
  ⟹ *apply_units1 CMi Ai iti* ≤ $\Downarrow_E$ *UNIV Id* (*apply_units CM A it*)
  ⟨*proof*⟩

lemma *dom_and_not_C_diff_aux*: ⟦¬*is_blocked A C*⟧
  ⟹ *dom* (*and_not_C A C*) − *dom A* = {*v* ∈ *var_of_lit* ‘ *C*. *A v* = *None*}
  ⟨*proof*⟩

lemma *dom_and_not_C_eq*: *dom* (*and_not_C A C*) = *dom A* ∪ *var_of_lit*‘*C*
  ⟨*proof*⟩

lemma *backtrack_and_not_C_trail_eq*: ⟦ *is_backtrack* (*and_not_C A C*) *T A*⟧
      ⟹ *T* = {*v* ∈ *var_of_lit* ‘ *C*. *A v* = *None*}
  ⟨*proof*⟩

definition *parse_check_blocked1* A_0 *cref* ≡ *doE* {
  ((*A*,*T*),*it*′) ← *parse_check_clause cref* (λ_. *True*) (λ*l* (*A*,*T*). *doE* {
    CHECK (*sem_lit*′ *l A* ≠ *Some True*) (*mkp_err STR* ′′*Blocked lemma clause*′′);
    if (*sem_lit*′ *l A* = *Some False*) *then ERETURN* (*A*,*T*)
    else doE {
      EASSERT (*sem_lit*′ *l A* = *None*);
      EASSERT (*var_of_lit l* ∉ *T*);
      ERETURN (*assign_lit A* (*neg_lit l*),*insert* (*var_of_lit l*) *T*)
    }
  }) (A_0,{});
  ERETURN (*cref*,(*A*,*T*),*it*′)
}

lemma *parse_check_blocked1_refine*[*refine*]:
  assumes [*simplified*, *simp*]: (*Ai*,*A*)∈*Id* (*iti*,*it*)∈*Id*
  shows *parse_check_blocked1 Ai iti*
    ≤ $\Downarrow_E$ *UNIV* (*cref_rel* $\times_r$ *Id* $\times_r$ *Id*) (*parse_check_blocked_bt A it*)
  ⟨*proof*⟩

definition *check_conflict_clause1* prf_0 *A cref*
  ≡ *iterate_clause cref* (λ_. *True*) (λ*l* _. *doE* {
      CHECK (*sem_lit*′ *l A* = *Some False*)

(mkp_errprf STR ''Expected conflict clause'' prf_0)
}) ()

lemma check_conflict_clause1_refine[refine]:
assumes [simplified,simp]: (Ai,A)∈Id
assumes CR: (cref,C)∈cref_rel
shows check_conflict_clause1 it_0 Ai cref
$\le\Downarrow_E$ UNIV Id (CHECK (is_conflict_clause A C) msg)
⟨proof⟩

definition lit_in_clause1 cref l ≡ doE {
iterate_clause cref (λf. ¬f) (λlx _. doE {
ERETURN (l=lx)
}) False
}

lemma lit_in_clause1_correct[THEN ESPEC_trans, refine_vcg]:
assumes CR: (cref,C) ∈ cref_rel
shows lit_in_clause1 cref lc ≤ ESPEC (λ_. False) (λr. r ⟷ lc∈C)
⟨proof⟩

definition lit_in_clause_and_not_true A cref lc ≡ doE {
f ← iterate_clause cref (λf. f≠2) (λl f. doE {
if (l=lc) then ERETURN 1
else if (sem_lit' l A = Some True) then ERETURN 2
else ERETURN f
}) (0::nat);
ERETURN (f=1)
}

lemma lit_in_clause_and_not_true_correct[THEN ESPEC_trans, refine_vcg]:
assumes CR: (cref,C) ∈ cref_rel
shows lit_in_clause_and_not_true A cref lc
≤ ESPEC (λ_. False)
(λr. r ⟷ lc∈C ∧ sem_clause' (C−{lc}) A ≠ Some True)
⟨proof⟩

definition and_not_C_excl A cref exl ≡ doE {
iterate_clause cref (λ_. True) (λl (A,T). doE {
if (l ≠ exl) then doE {
EASSERT (sem_lit' l A ≠ Some True);
if (sem_lit' l A ≠ Some False) then doE {
EASSERT (var_of_lit l ∉ T);
ERETURN (assign_lit A (neg_lit l), insert (var_of_lit l) T)
} else
ERETURN (A,T)
} else
ERETURN (A,T)
}) (A,{})
}

lemma and_not_C_excl_refine[refine]:
assumes [simplified,simp]: (Ai,A)∈Id
assumes CR: (cref,C) ∈ cref_rel
assumes [simplified,simp]: (exli,exl)∈Id
~~assumes NBLK: ¬is_blocked A (C−{exl})~~
shows and_not_C_excl Ai cref exli
$\le\Downarrow_E$ UNIV ($Id\times_r Id$) (and_not_C_bt A (C−{exl}))
⟨proof⟩

```
definition get_rat_candidates1
  :: ('it) clausemap1 ⇒ (var ⇀ bool) ⇒ var literal ⇒ (_,id set) enres
  where
  get_rat_candidates1 ≡ λ(CM,RL) A l. doE {
    let l = neg_lit l;
    let cands_raw = RL l;
    CHECK (¬is_None cands_raw) (mkp_err STR ''Resolution literal not declared'');
    ~~// Get collected candidates~~
    let cands_raw = the cands_raw;
    ~~EASSERT (distinct cands_raw);~~
    ~~// Filter deleted, blocked, and those not containing resolution literal~~
    cands ← enfoldli cands_raw (λ_. True) (λi s. doE {
      let cref = CM i;
      if ¬is_None cref then doE {
        let cref = the cref;
        lant ← lit_in_clause_and_not_true A cref l;
        if lant then doE {
          ~~EASSERT (i ∉ s);~~
          ERETURN (insert i s)
        } else ERETURN s
      } else ERETURN s
    }) {};
    ERETURN cands
  }
```

~~xxx: Choice: We could 1) either remove duplicates after all candidates have been gathered, or 2) from RL-list before deleted/blocked/contained check 1) In case of massive duplicates, checks will be repeated. However, typically, only a few RAT candidates remain, such that simple $O(n^2)$ remdup impl can be used. Moreover, we do not expect massive duplicates. 2) In case of long candidate lists, remdups may be expensive or requires efficient DS.~~

```
lemma get_rat_candidates1_refine[refine]:
  assumes CMR: (CMi,CM)∈clausemap1_rel
  assumes [simplified, simp]: (Ai,A)∈Id (resliti,reslit)∈Id
  shows get_rat_candidates1 CMi Ai resliti
        ≤⇓E UNIV (Id) (get_rat_candidates CM A reslit)
  ⟨proof⟩
    focus ⟨proof⟩
    solved
    ⟨proof⟩


definition backtrack1 A T ≡ do {
  ASSUME (finite T);
  FOREACH T (λx A. RETURN (A(x:=None))) A
}

lemma backtrack1_correct[THEN SPEC_trans, refine_vcg]:
  backtrack1 A T ≤ SPEC (λr. r = backtrack A T)
  ⟨proof⟩


definition (in −) abs_cr_register_ndj
:: 'a literal ⇒ 'id ⇒ ('a literal ⇀ 'id list) ⇒ ('a literal ⇀ 'id list)
where abs_cr_register_ndj l cid cr ≡ case cr l of
  None ⇒ cr | Some s ⇒ cr(l ↦ cid#s)


definition register_clause1 cid cref RL ≡ doE {
    iterate_clause cref (λ_. True) (λl RL. doE {
      ERETURN (abs_cr_register_ndj l cid RL)
    }) RL
  }
```

~~XXX: Do we really need mbhd–insert? We iterate over literals of clause, which should not contain duplicates!~~

definition *RL_upd cid C RL* ≡ (λ*l*. *case RL l of*
None ⇒ *None*
| *Some s* ⇒ *if l*∈*C then Some* (*insert cid s*) *else Some s*)

lemma *RL_upd_empty*[*simp*]: *RL_upd cid* {} *RL* = *RL*
⟨*proof*⟩

lemma *RL_upd_insert_eff*:
RL_upd cid C RL l = *Some s*
⟹ *RL_upd cid* (*insert l C*) *RL* = (*RL_upd cid C RL*)(*l* ↦ *insert cid s*)
⟨*proof*⟩

lemma *RL_upd_insert_noeff*:
RL_upd cid C RL l = *None* ⟹ *RL_upd cid* (*insert l C*) *RL* = *RL_upd cid C RL*
⟨*proof*⟩

lemma *register_clause1_correct*[*THEN ESPEC_trans*, *refine_vcg*]:
assumes *CR*: (*cref*,*C*)∈*cref_rel*
assumes *RL*: (*RLi*,*RL*)∈*Id* → ⟨*br set* (λ_. *True*)⟩*option_rel*
~~assumes fresh_id: cid ∉ ⋃ ran RL~~
shows *register_clause1 cid cref RLi*
≤ *ESPEC* (λ_. *False*)
(λ*RLi*′. (*RLi*′, *RL_upd cid C RL*) ∈ *Id* → ⟨*br set* (λ_. *True*)⟩*option_rel*)
⟨*proof*⟩
apply1 (*frule fun_relD*[*OF _ IdI*[*of l*]])
apply1 (*frule fun_relD*[*OF _ IdI*[*of l*′]])
apply1 (*erule option_relE*;
simp add: *RL_upd_insert_eff RL_upd_insert_noeff*)
applyS (*auto simp*: *in_br_conv mbhd_insert_correct mbhd_insert_invar*) []
⟨*proof*⟩
apply1 (*drule fun_relD*[*OF _ IdI*[*of l*′]])
apply1 (*erule set_rev_mp*[*OF _ option_rel_mono*])
applyS (*auto simp*: *in_br_conv mbhd_invar_exit*)
⟨*proof*⟩

definition *add_clause1*
:: *id* ⇒ ′*it* ⇒ (′*it*) *clausemap1* ⇒ (_,(′*it*) *clausemap1*) *enres*
where *add_clause1* ≡ λ*i cref* (*CM*,*RL*). *doE* {
let CM = *CM*(*i* ↦ *cref*);
RL ← *register_clause1 i cref RL*;
ERETURN (*CM*,*RL*)
}

lemma *add_clause1_refine*[*refine*]:
⟦ (*ii*,*i*)∈*Id*; (*cref*,*C*)∈*cref_rel*; (*CMi*,*CM*)∈*clausemap1_rel* ⟧ ⟹
add_clause1 ii cref CMi $\leq\Downarrow_E$ *UNIV clausemap1_rel* (*add_clause i C CM*)
⟨*proof*⟩
applyS *assumption*
applyS (*erule fun_relD*[*rotated*, **where** *f*=*RLi* **and** *f*′=*RL*];
auto simp: *clausemap1_rel_def*)
~~applyS (auto simp: cm_ids_def)~~
apply1 *clarsimp* ⟨*proof*⟩
applyS (*auto simp*: *RL_upd_def split*: *if_split_asm*) []
applyS (*auto simp*: *RL_upd_def split*: *if_split_asm*) []
applyS (*auto*
simp: *RL_upd_def cref_rel_def in_br_conv*

```
          split: if_split_asm)
    ⟨proof⟩


definition check_rup_proof1
  :: ('it) state1 ⇒ 'it ⇒ (nat×'prf) ⇒ (_,('it) state1 × 'it × (nat×'prf)) enres
  where
  check_rup_proof1 ≡ λ(CM,A) it prf. doE {
    (i,prf) ← parse_id prf;
    CHECK (i∉cm_ids CM) (mkp_errNprf STR ''Duplicate ID'' i prf);
    (cref,(A,T),it) ← parse_check_blocked1 A it;

    ((A,T),prf) ← apply_units1_bt CM A T prf;
    (confl_id,prf) ← parse_id prf;
    confl ← resolve_id1 CM confl_id;
    check_conflict_clause1 prf A confl;
    CM ← add_clause1 i cref CM;
    A ← enres_lift (backtrack1 A T);
    ERETURN ((CM,A),it,prf)
  }


lemma cm1_rel_imp_eq_ids[simp]:
  assumes (cm1,cm)∈clausemap1_rel
  shows cm_ids cm1 = cm_ids cm
⟨proof⟩

lemma check_rup_proof1_refine[refine]:
  assumes SR: (si,s)∈state1_rel
  assumes [simplified, simp]: (iti,it)∈Id (prfi,prf)∈Id
  shows check_rup_proof1 si iti prfi
        ≤⇓E UNIV (state1_rel ×r Id ×r Id) (check_rup_proof_bt s it prf)
⟨proof⟩

definition check_rat_candidates_part1 CM reslit candidates A prf ≡
  check_candidates' candidates A prf (λcand_id A prf. doE {
    cand ← resolve_id1 CM cand_id;

    (A,T2) ← and_not_C_excl A cand (neg_lit reslit);
    ((A,T2),prf) ← apply_units1_bt CM A T2 prf;
    (confl_id,prf) ← parse_id prf;
    confl ← resolve_id1 CM confl_id;
    check_conflict_clause1 prf A confl;
    A ← enres_lift (backtrack1 A T2);
    ERETURN (A,prf)
  })

definition check_rat_proof1
  :: ('it) state1 ⇒ 'it ⇒ (nat × 'prf) ⇒ (_,('it) state1 × 'it × (nat × 'prf)) enres
  where
  check_rat_proof1 ≡ λ(CM,A) it prf. doE {
    (reslit,prf) ← parse_prf_literal prf;
    CHECK (sem_lit' reslit A ≠ Some False)
          (mkp_errprf STR ''Resolution literal is false'' prf);
    (i,prf) ← parse_id prf;
    CHECK (i∉cm_ids CM) (mkp_errNprf STR ''Ids must be strictly increasing'' i prf);
    (cref,(A,T),it) ← parse_check_blocked1 A it;

    CHECK_monadic (lit_in_clause1 cref reslit)
                (mkp_errprf STR ''Resolution literal not in clause'' prf);
    ((A,T),prf) ← apply_units1_bt CM A T prf;
    candidates ← get_rat_candidates1 CM A reslit;
    (A,prf) ← check_rat_candidates_part1 CM reslit candidates A prf;
```

```
    CM ← add_clause1 i cref CM;
    A ← enres_lift (backtrack1 A T);
    ERETURN ((CM,A),it,prf)
  }

lemma check_rat_proof1_refine[refine]:
  assumes SR: (si,s)∈state1_rel
  assumes [simplified, simp]: (iti,it)∈Id (prfi,prf)∈Id
  shows check_rat_proof1 si iti prfi
        ≤⇓E UNIV (state1_rel ×r Id ×r Id) (check_rat_proof_bt s it prf)
⟨proof⟩


definition remove_id1
  :: id ⇒ ('cref) clausemap1 ⇒ (_,('cref) clausemap1) enres
  where remove_id1 ≡ λi (CM,RL). ERETURN (CM(i:=None),RL)

lemma remove_id1_refine[refine]:
  ⟦ (ii,i)∈Id; (CMi,CM)∈clausemap1_rel ⟧
    ⟹ remove_id1 ii CMi ≤⇓E UNIV clausemap1_rel (remove_id i CM)
  ⟨proof⟩

definition remove_ids1
    :: ('cref) clausemap1 ⇒ (nat × 'prf) ⇒ (_,('cref) clausemap1 × (nat × 'prf)) enres
  where
  remove_ids1 CM prf ≡ doE {
    (i,prf) ← parse_idZ prf;
    (CM,i,prf) ← EWHILET
      (λ(_,i,_). i≠0)
      (λ(CM,i,prf). doE {
        CM ← remove_id1 i CM;
        (i,prf) ← parse_idZ prf;
        ERETURN (CM,i,prf)
      }) (CM,i,prf);
    ERETURN (CM,prf)
}

lemma remove_ids1_refine[refine]:
  ⟦ (CMi,CM)∈clausemap1_rel; (prfi,prf)∈Id ⟧
    ⟹ remove_ids1 CMi prfi ≤⇓E UNIV (clausemap1_rel ×r Id) (remove_ids CM prf)
  ⟨proof⟩

definition check_item1
  :: ('it) state1 ⇒ 'it ⇒ (nat × 'prf) ⇒ (_,(('it) state1 × 'it × (nat × 'prf)) option) enres
  where check_item1 ≡ λ(CM,A) it prf. doE {
    (ty,prf) ← parse_type prf;
    case ty of
      INVALID ⇒ THROW (mkp_err STR ''Invalid item'')
    | UNIT_PROP ⇒ doE {
        (A,prf) ← apply_units1 CM A prf;
        ERETURN (Some ((CM,A),it,prf))
      }
    | DELETION ⇒ doE {
        (CM,prf) ← remove_ids1 CM prf;
        ERETURN (Some ((CM,A),it,prf))
      }
    | RUP_LEMMA ⇒ doE {
        s ← check_rup_proof1 (CM,A) it prf;
        ERETURN (Some s)
      }
    | RAT_LEMMA ⇒ doE {
        s ← check_rat_proof1 (CM,A) it prf;
        ERETURN (Some s)
```

```
      }
    | CONFLICT ⇒ doE {
        (i,prf) ← parse_id prf;
        cref ← resolve_id1 CM i;
        check_conflict_clause1 prf A cref;
        ERETURN None
      }
    | RAT_COUNTS ⇒ THROW (mkp_errprf
          STR ''Not expecting rat−counts in the middle of proof'' prf)
  }
```

```
lemma check_item1_refine[refine]:
  assumes SR: (si,s)∈state1_rel
  assumes [simplified, simp]: (iti,it)∈Id (prfi,prf)∈Id
  shows check_item1 si iti prfi
        ≤⇓_E UNIV (⟨state1_rel ×_r Id ×_r Id⟩option_rel) (check_item_bt s it prf)
  ⟨proof⟩
  applyS simp
  ⟨proof⟩
```

```
lemma check_item1_deforest: check_item1 = (λ(CM,A) it prf. doE {
    (ty,prf) ← parse_prf prf;
    if ty=1 then doE {
        (A,prf) ← apply_units1 CM A prf;
        ERETURN (Some ((CM,A),it,prf))
      }
    else if ty=2 then doE {
        (CM,prf) ← remove_ids1 CM prf;
        ERETURN (Some ((CM,A),it,prf))
      }
    else if ty=3 then doE {
        s ← check_rup_proof1 (CM,A) it prf;
        ERETURN (Some s)
      }
    else if ty=4 then doE {
        s ← check_rat_proof1 (CM,A) it prf;
        ERETURN (Some s)
      }
    else if ty=5 then doE {
        (i,prf) ← parse_id prf;
        cref ← resolve_id1 CM i;
        check_conflict_clause1 prf A cref;
        ERETURN None
      }
    else if ty=6 then
      THROW (mkp_errprf STR ''Not expecting rat−counts in the middle of proof'' prf)
    else
      THROW (mkp_errIprf STR ''Invalid item type'' ty prf)
  })
  ⟨proof⟩
```

```
definition (in −) cm_empty1 :: ('cref) clausemap1
  where cm_empty1 ≡ (Map.empty, Map.empty)
lemma cm_empty_refine[refine]: (cm_empty1, cm_empty) ∈ clausemap1_rel
  ⟨proof⟩
```

```
definition is_syn_taut1 cref A ≡ doE {
  EASSERT (A = Map.empty);
  (t,A) ← iterate_clause cref (λ(t,A). ¬t) (λl (t,A). doE {
    if (sem_lit' l A = Some False) then ERETURN (True,A)
    else if sem_lit' l A = Some True then ERETURN (False,A) DUP literal. Perhaps check for it?
```

```
    else doE {
      EASSERT (sem_lit' l A = None);
      ERETURN (False,assign_lit A l)
    }
  }) (False,A);

  ~~Iterate again over clause to reset assignment~~
  A ← iterate_clause cref (λ_. True) (λl A. doE {
    let A = A(var_of_lit l := None);
    ERETURN A
  }) A;

  ERETURN (t,A)
}

lemma is_syn_taut1_correct[THEN ESPEC_trans, refine_vcg]:
  assumes CR: (cref,C)∈cref_rel
  assumes [simp]: A = Map.empty
  shows is_syn_taut1 cref A
          ≤ ESPEC (λ_. False) (λ(t,A). (t ⟷ is_syn_taut C) ∧ A=Map.empty)
⟨proof⟩

definition read_cnf_new1
  :: 'it ⇒ 'it ⇒ 'it clausemap1 ⇒ (_, 'it clausemap1) enres
  where read_cnf_new1 itE it CM ≡ doE {
    (CM,next_id,A) ← tok_fold itE it (λit (CM,next_id,A). doE {
      (it',(t,A)) ← read_clause_check_taut itE it A;
      if t then ERETURN (it', (CM,next_id+1,A))
      else doE {
        EASSERT (∃ l it'. lz_string litZ it l it');
        let C = it;
        CM ← add_clause1 next_id C CM;
        ERETURN (it',(CM,next_id+1,A))
      }
    }) (CM,1,Map.empty);
    ERETURN (CM)
  }

lemma read_cnf_new1_refine[refine]:
  assumes [simplified,simp]: (F_begini, F_begin)∈Id (F_endi,F_end)∈Id
  assumes [simp]: (CMi,CM)∈clausemap1_rel
  shows read_cnf_new1 F_endi F_begini CMi
          ≤ ⇓E UNIV (clausemap1_rel)
          (read_cnf_new F_end F_begin CM)
  ⟨proof⟩
  applyS auto
  ⟨proof⟩

definition cm_init_lit1
  :: var literal ⇒ ('it) clausemap1 ⇒ (_,('it) clausemap1) enres
  where cm_init_lit1 ≡ λl (CM,RL). ERETURN (CM,RL(l ↦ []))

definition init_rat_counts1 prf ≡ doE {
  (ty,prf) ← parse_type prf;
  CHECK (ty = RAT_COUNTS) (mkp_errprf STR ''Expected RAT counts item'' prf);

  (l,prf) ← parse_prf_literalZ prf;
  (CM,_,prf) ← EWHILET (λ(CM,l,prf). l≠None) (λ(CM,l,prf). doE {
    EASSERT (l≠None);
    let l = the l;
    (_,prf) ← parse_prf prf;

    let l = neg_lit l;
```

```
    CM ← cm_init_lit1 l CM;

    (l,prf) ← parse_prf_literalZ prf;
    ERETURN (CM,l,prf)
  }) (cm_empty1,l,prf);

  ERETURN (CM,prf)
}
```

lemma *init_rat_counts1_refine*[*refine*]:
assumes [*simplified*,*simp*]: (*prfi*,*prf*)∈*Id*
shows *init_rat_counts1 prfi* $\leq\Downarrow_E$ *UNIV* (*clausemap1_rel* $\times_r$ *Id*) (*init_rat_counts prf*)
⟨*proof*⟩

lemma *init_rat_counts1_deforest*: *init_rat_counts1 prf* = *doE* {

```
  (ty,prf) ← parse_prf prf;
  CHECK (ty = 1 ∨ ty = 2 ∨ ty = 3 ∨ ty = 4 ∨ ty = 5 ∨ ty = 6)
        (mkp_errIprf STR ''Invalid item type'' ty prf);
  CHECK (ty = 6) (mkp_errprf STR ''Expected RAT counts item'' prf);
  (l,prf) ← parse_prf_literalZ prf;
  (CM,l,prf) ← EWHILET
    (λ(CM,l,prf). l≠None)
    (λ(CM,l,prf). doE {
      EASSERT (l≠None);
      let l = the l;

      (_,prf) ← parse_prf prf;
      let l = neg_lit l;
      CM ← cm_init_lit1 l CM;

      (l,prf) ← parse_prf_literalZ prf;
      ERETURN (CM,l,prf)
    }) (cm_empty1,l,prf);
  ERETURN (CM,prf)
}
```

⟨*proof*⟩

definition *verify_unsat1 F_begin F_end it prf* ≡ *doE* {

```
  EASSERT (it_invar it);

  (CM,prf) ← init_rat_counts1 prf;

  CM ← read_cnf_new1 F_end F_begin CM;

  let s = (CM,Map.empty);

  EWHILEIT
    (λSome (_,it,_) ⇒ it_invar it | None ⇒ True)
    (λs. s≠None)
    (λs. doE {
      EASSERT (s≠None);
      let (s,it,prf) = the s;

      EASSERT (it_invar it);

      check_item1 s it prf
    }) (Some (s,it,prf));
  ERETURN ()
  CHECK (is_None s) (mkp_err ''Proof did not contain conflict declaration'')
}
```

lemma *verify_unsat1_refine*[*refine*]:
  ⟦ (*F_begini*,*F_begin*)∈*Id*; (*F_endi*,*F_end*)∈*Id*; (*iti*,*it*)∈*Id*; (*prfi*,*prf*)∈*Id* ⟧
    ⟹ *verify_unsat1 F_begini F_endi iti prfi*
        ≤⇓$_E$ *UNIV Id* (*verify_unsat_bt F_begin F_end it prf*)
  ⟨*proof*⟩

end

4.4 Refinement 2

4.4.1 Getting Out of Exception Monad

context *unsat_input*
begin
  lemmas [*enres_inline*] = *parse_id_def parse_idZ_def parse_prf_literal_def parse_prf_literalZ_def*

  synth-definition *parse_prf_bd* **is** [*enres_unfolds*]: *parse_prf prf* = ¤
  ⟨*proof*⟩

  synth-definition *check_unit_clause1_bd*
    is [*enres_unfolds*]: *check_unit_clause1 A cref* = ¤
  ⟨*proof*⟩
  lemma *resolve_id1_alt*: *resolve_id1* = (λ(*CM*,_) *i*. *doE* {
      let x = *CM i*;
      if (*x*=*None*) *then THROW* (*mkp_errN STR ''Invalid clause id'' i*)
      else ERETURN (*the x*)
    })
  ⟨*proof*⟩

  synth-definition *resolve_id1_bd* **is** [*enres_unfolds*]: *resolve_id1 CM cid* = ¤
  ⟨*proof*⟩

  synth-definition *apply_unit1_bt_bd*
    is [*enres_unfolds*]: *apply_unit1_bt i CM A T* = ¤
  ⟨*proof*⟩

  synth-definition *apply_units1_bt_bd*
    is [*enres_unfolds*]: *apply_units1_bt CM A T units* = ¤
  ⟨*proof*⟩

  synth-definition *apply_unit1_bd* **is** [*enres_unfolds*]: *apply_unit1 i CM A* = ¤
  ⟨*proof*⟩

  synth-definition *apply_units1_bd*
    is [*enres_unfolds*]: *apply_units1 CM A units* = ¤
  ⟨*proof*⟩

  synth-definition *remove_ids1_bd*
    is [*enres_unfolds*]: *remove_ids1 CM prf* = ¤
  ⟨*proof*⟩

  synth-definition *parse_check_blocked1_bd*
    is [*enres_unfolds*]: *parse_check_blocked1 A cref* = ¤
  ⟨*proof*⟩

  synth-definition *check_conflict_clause1_bd*
    is [*enres_unfolds*]: *check_conflict_clause1* prf_0 *A cref* = ¤
  ⟨*proof*⟩

  synth-definition *and_not_C_excl_bd*
    is [*enres_breakdown*]: *and_not_C_excl A cref exl* = *enres_lift* ¤
  ⟨*proof*⟩

synth-definition *lit_in_clause_and_not_true_bd*
is [*enres_breakdown*]: *lit_in_clause_and_not_true A cref lc* = *enres_lift* ¤
⟨*proof*⟩

synth-definition *lit_in_clause_bd*
is [*enres_breakdown*]: *lit_in_clause1 cref lc* = *enres_lift* ¤
⟨*proof*⟩

synth-definition *get_rat_candidates1_bd*
is [*enres_unfolds*]: *get_rat_candidates1 CM A l* = ¤
⟨*proof*⟩

synth-definition *add_clause1_bd*
is [*enres_breakdown*]: *add_clause1 i it CM* = *enres_lift* ¤
⟨*proof*⟩

synth-definition *check_rup_proof1_bd*
is [*enres_unfolds*]: *check_rup_proof1 s it prf* = ¤
⟨*proof*⟩

term *check_rat_candidates_part1*
synth-definition *check_rat_candidates_part1_bd*
is [*enres_unfolds*]:
check_rat_candidates_part1 CM reslit candidates A prf = ¤
⟨*proof*⟩

synth-definition *check_rat_proof1_bd*
is [*enres_unfolds*]: *check_rat_proof1 s it prf* = ¤
⟨*proof*⟩

synth-definition *check_item1_bd* **is** [*enres_unfolds*]: *check_item1 s it prf* = ¤
⟨*proof*⟩

synth-definition *is_syn_taut1_bd*
is [*enres_breakdown*]: *is_syn_taut1 cref A* = *enres_lift* ¤
⟨*proof*⟩

synth-definition *read_clause_check_taut_bd*
is [*enres_unfolds*]: *read_clause_check_taut F_end F_begin A* = ¤
⟨*proof*⟩

synth-definition *read_cnf_new1_bd*
is [*enres_unfolds*]: *read_cnf_new1 F_begin F_end CM* = ¤
⟨*proof*⟩

synth-definition *init_rat_counts1_bd*
is [*enres_unfolds*]: *init_rat_counts1 prf* = ¤
⟨*proof*⟩

synth-definition *verify_unsat1_bd*
is [*enres_unfolds*]: *verify_unsat1 F_begin F_end it prf* = ¤
⟨*proof*⟩

end

4.4.2 Instantiating Input Locale

locale *GRAT_def_loc* = *DB2_def_loc* +
fixes *prf_next* :: $'prf \Rightarrow int \times 'prf$

locale *GRAT_loc* = *DB2_loc DB frml_end* + *GRAT_def_loc DB frml_end prf_next*

for *DB frml_end* **and** *prf_next* :: *'prf* ⇒ *int* × *'prf*

context *GRAT_loc*
begin
sublocale *unsat_input liti.next liti.peek liti.end liti.I prf_next*
⟨*proof*⟩

end

4.4.3 Extraction from Locale

named-theorems *extrloc_unfolds*

concrete-definition (**in** *GRAT_loc*) *parse_prf2_loc*
uses *parse_prf_bd_def*[*unfolded extrloc_unfolds*]
declare (**in** *GRAT_loc*) *parse_prf2_loc.refine*[*extrloc_unfolds*]
concrete-definition *parse_prf2*
uses *GRAT_loc.parse_prf2_loc_def*[*unfolded extrloc_unfolds*]
declare (**in** *GRAT_loc*) *parse_prf2.refine*[*OF GRAT_loc_axioms*, *extrloc_unfolds*]

concrete-definition (**in** *GRAT_loc*) *parse_check_blocked2_loc*
uses *parse_check_blocked1_bd_def*[*unfolded extrloc_unfolds*]
declare (**in** *GRAT_loc*) *parse_check_blocked2_loc.refine*[*extrloc_unfolds*]
concrete-definition *parse_check_blocked2*
uses *GRAT_loc.parse_check_blocked2_loc_def*[*unfolded extrloc_unfolds*]
declare (**in** *GRAT_loc*) *parse_check_blocked2.refine*[*OF GRAT_loc_axioms*, *extrloc_unfolds*]

concrete-definition (**in** *GRAT_loc*) *check_unit_clause2_loc*
uses *check_unit_clause1_bd_def*[*unfolded extrloc_unfolds*]
declare (**in** *GRAT_loc*) *check_unit_clause2_loc.refine*[*extrloc_unfolds*]
concrete-definition *check_unit_clause2* **uses** *GRAT_loc.check_unit_clause2_loc_def*[*unfolded extrloc_unfolds*]
declare (**in** *GRAT_loc*) *check_unit_clause2.refine*[*OF GRAT_loc_axioms*, *extrloc_unfolds*]

concrete-definition (**in** *GRAT_loc*) *resolve_id2_loc*
uses *resolve_id1_bd_def*[*unfolded extrloc_unfolds*]
declare (**in** *GRAT_loc*) *resolve_id2_loc.refine*[*extrloc_unfolds*]
concrete-definition *resolve_id2* **uses** *GRAT_loc.resolve_id2_loc_def*[*unfolded extrloc_unfolds*]
declare (**in** *GRAT_loc*) *resolve_id2.refine*[*OF GRAT_loc_axioms*, *extrloc_unfolds*]

concrete-definition (**in** *GRAT_loc*) *apply_units2_loc*
uses *apply_units1_bd_def*[*unfolded apply_unit1_bd_def extrloc_unfolds*]
declare (**in** *GRAT_loc*) *apply_units2_loc.refine*[*extrloc_unfolds*]
concrete-definition *apply_units2* **uses** *GRAT_loc.apply_units2_loc_def*[*unfolded extrloc_unfolds*]
declare (**in** *GRAT_loc*) *apply_units2.refine*[*OF GRAT_loc_axioms*, *extrloc_unfolds*]

concrete-definition (**in** *GRAT_loc*) *apply_units2_bt_loc*
uses *apply_units1_bt_bd_def*[*unfolded apply_unit1_bt_bd_def extrloc_unfolds*]
declare (**in** *GRAT_loc*) *apply_units2_bt_loc.refine*[*extrloc_unfolds*]
concrete-definition *apply_units2_bt* **uses** *GRAT_loc.apply_units2_bt_loc_def*[*unfolded extrloc_unfolds*]
declare (**in** *GRAT_loc*) *apply_units2_bt.refine*[*OF GRAT_loc_axioms*, *extrloc_unfolds*]

concrete-definition (**in** *GRAT_loc*) *remove_ids2_loc*
uses *remove_ids1_bd_def*[*unfolded extrloc_unfolds*]
declare (**in** *GRAT_loc*) *remove_ids2_loc.refine*[*extrloc_unfolds*]
concrete-definition *remove_ids2* **uses** *GRAT_loc.remove_ids2_loc_def*[*unfolded extrloc_unfolds*]
declare (**in** *GRAT_loc*) *remove_ids2.refine*[*OF GRAT_loc_axioms*, *extrloc_unfolds*]

concrete-definition (**in** *GRAT_loc*) *check_conflict_clause2_loc*
uses *check_conflict_clause1_bd_def*[*unfolded extrloc_unfolds*]
declare (**in** *GRAT_loc*) *check_conflict_clause2_loc.refine*[*extrloc_unfolds*]
concrete-definition *check_conflict_clause2* **uses** *GRAT_loc.check_conflict_clause2_loc_def*[*unfolded extrloc_unfolds*]
declare (**in** *GRAT_loc*) *check_conflict_clause2.refine*[*OF GRAT_loc_axioms*, *extrloc_unfolds*]

concrete-definition (**in** *GRAT_loc*) *and_not_C_excl2_loc*
uses *and_not_C_excl_bd_def*[*unfolded extrloc_unfolds*]
declare (**in** *GRAT_loc*) *and_not_C_excl2_loc.refine*[*extrloc_unfolds*]
concrete-definition *and_not_C_excl2* **uses** *GRAT_loc.and_not_C_excl2_loc_def*[*unfolded extrloc_unfolds*]
declare (**in** *GRAT_loc*) *and_not_C_excl2.refine*[*OF GRAT_loc_axioms*, *extrloc_unfolds*]

concrete-definition (**in** *GRAT_loc*) *lit_in_clause_and_not_true2_loc*
uses *lit_in_clause_and_not_true_bd_def*[*unfolded extrloc_unfolds*]
declare (**in** *GRAT_loc*) *lit_in_clause_and_not_true2_loc.refine*[*extrloc_unfolds*]
concrete-definition *lit_in_clause_and_not_true2* **uses** *GRAT_loc.lit_in_clause_and_not_true2_loc_def*[*unfolded extrloc_unfolds*]
declare (**in** *GRAT_loc*) *lit_in_clause_and_not_true2.refine*[*OF GRAT_loc_axioms*, *extrloc_unfolds*]

concrete-definition (**in** *GRAT_loc*) *get_rat_candidates2_loc*
uses *get_rat_candidates1_bd_def*[*unfolded extrloc_unfolds*]
declare (**in** *GRAT_loc*) *get_rat_candidates2_loc.refine*[*extrloc_unfolds*]
concrete-definition *get_rat_candidates2* **uses** *GRAT_loc.get_rat_candidates2_loc_def*[*unfolded extrloc_unfolds*]
declare (**in** *GRAT_loc*) *get_rat_candidates2.refine*[*OF GRAT_loc_axioms*, *extrloc_unfolds*]

concrete-definition (**in** *GRAT_loc*) *backtrack2_loc*
uses *backtrack1_def*[*unfolded extrloc_unfolds*]
declare (**in** *GRAT_loc*) *backtrack2_loc.refine*[*extrloc_unfolds*]
concrete-definition *backtrack2* **uses** *GRAT_loc.backtrack2_loc_def*[*unfolded extrloc_unfolds*]
declare (**in** *GRAT_loc*) *backtrack2.refine*[*OF GRAT_loc_axioms*, *extrloc_unfolds*]

concrete-definition (**in** *GRAT_loc*) *add_clause2_loc*
uses *add_clause1_bd_def*[*unfolded extrloc_unfolds*]
declare (**in** *GRAT_loc*) *add_clause2_loc.refine*[*extrloc_unfolds*]
concrete-definition *add_clause2* **uses** *GRAT_loc.add_clause2_loc_def*[*unfolded extrloc_unfolds*]
declare (**in** *GRAT_loc*) *add_clause2.refine*[*OF GRAT_loc_axioms*, *extrloc_unfolds*]

concrete-definition (**in** *GRAT_loc*) *check_rup_proof2_loc*
uses *check_rup_proof1_bd_def*[*unfolded extrloc_unfolds*]
declare (**in** *GRAT_loc*) *check_rup_proof2_loc.refine*[*extrloc_unfolds*]
concrete-definition *check_rup_proof2* **uses** *GRAT_loc.check_rup_proof2_loc_def*[*unfolded extrloc_unfolds*]
declare (**in** *GRAT_loc*) *check_rup_proof2.refine*[*OF GRAT_loc_axioms*, *extrloc_unfolds*]

concrete-definition (**in** *GRAT_loc*) *lit_in_clause2_loc*
uses *lit_in_clause_bd_def*[*unfolded extrloc_unfolds*]
declare (**in** *GRAT_loc*) *lit_in_clause2_loc.refine*[*extrloc_unfolds*]
concrete-definition *lit_in_clause2* **uses** *GRAT_loc.lit_in_clause2_loc_def*[*unfolded extrloc_unfolds*]
declare (**in** *GRAT_loc*) *lit_in_clause2.refine*[*OF GRAT_loc_axioms*, *extrloc_unfolds*]

concrete-definition (**in** *GRAT_loc*) *check_rat_candidates_part2_loc*
uses *check_rat_candidates_part1_bd_def*[*unfolded extrloc_unfolds*]
declare (**in** *GRAT_loc*) *check_rat_candidates_part2_loc.refine*[*extrloc_unfolds*]
concrete-definition *check_rat_candidates_part2* **uses** *GRAT_loc.check_rat_candidates_part2_loc_def*[*unfolded extrloc_unfolds*]
declare(**in** *GRAT_loc*) *check_rat_candidates_part2.refine*[*OF GRAT_loc_axioms*, *extrloc_unfolds*]

concrete-definition (**in** *GRAT_loc*) *check_rat_proof2_loc*
uses *check_rat_proof1_bd_def*[*unfolded extrloc_unfolds*]
declare (**in** *GRAT_loc*) *check_rat_proof2_loc.refine*[*extrloc_unfolds*]
concrete-definition *check_rat_proof2* **uses** *GRAT_loc.check_rat_proof2_loc_def*[*unfolded extrloc_unfolds*]
declare (**in** *GRAT_loc*) *check_rat_proof2.refine*[*OF GRAT_loc_axioms*, *extrloc_unfolds*]

concrete-definition (**in** *GRAT_loc*) *check_item2_loc*
uses *check_item1_bd_def*[*unfolded extrloc_unfolds*]
declare (**in** *GRAT_loc*) *check_item2_loc.refine*[*extrloc_unfolds*]

concrete-definition check_item2 **uses** GRAT_loc.check_item2_loc_def[unfolded extrloc_unfolds]
declare (**in** GRAT_loc) check_item2.refine[OF GRAT_loc_axioms, extrloc_unfolds]

concrete-definition (**in** GRAT_loc) is_syn_taut2_loc
  uses is_syn_taut1_bd_def[unfolded extrloc_unfolds]
declare (**in** GRAT_loc) is_syn_taut2_loc.refine[extrloc_unfolds]
concrete-definition is_syn_taut2 **uses** GRAT_loc.is_syn_taut2_loc_def[unfolded extrloc_unfolds]
declare (**in** GRAT_loc) is_syn_taut2.refine[OF GRAT_loc_axioms, extrloc_unfolds]

~~concrete_definition (in GRAT_loc) read_cnf2_loc uses read_cnf1_bd_def[unfolded extrloc_unfolds] declare (in GRAT_loc) read_cnf2_loc.refine[extrloc_unfolds] concrete_definition read_cnf2 uses GRAT_loc.read_cnf2_loc_def[unfolded extrloc_unfolds] declare (in GRAT_loc) read_cnf2.refine[OF GRAT_loc_axioms, extrloc_unfolds]~~

concrete-definition (**in** GRAT_loc) read_clause_check_taut2_loc
  uses read_clause_check_taut_bd_def[unfolded extrloc_unfolds]
declare (**in** GRAT_loc) read_clause_check_taut2_loc.refine[extrloc_unfolds]
concrete-definition read_clause_check_taut2 **uses** GRAT_loc.read_clause_check_taut2_loc_def[unfolded extrloc_unfolds]
declare (**in** GRAT_loc) read_clause_check_taut2.refine[OF GRAT_loc_axioms, extrloc_unfolds]

concrete-definition (**in** GRAT_loc) read_cnf_new2_loc
  uses read_cnf_new1_bd_def[unfolded extrloc_unfolds]
declare (**in** GRAT_loc) read_cnf_new2_loc.refine[extrloc_unfolds]
concrete-definition read_cnf_new2 **uses** GRAT_loc.read_cnf_new2_loc_def[unfolded extrloc_unfolds]
declare (**in** GRAT_loc) read_cnf_new2.refine[OF GRAT_loc_axioms, extrloc_unfolds]

~~concrete_definition (in GRAT_loc) goto_next_item2_loc uses goto_next_item_bd_def[unfolded extrloc_unfolds] declare (in GRAT_loc) goto_next_item2_loc.refine[extrloc_unfolds] concrete_definition goto_next_item2 uses GRAT_loc.goto_next_item2_ extrloc_unfolds] declare (in GRAT_loc) goto_next_item2.refine[OF GRAT_loc_axioms, extrloc_unfolds]~~

concrete-definition (**in** GRAT_loc) init_rat_counts2_loc
  uses init_rat_counts1_bd_def[unfolded extrloc_unfolds]
declare (**in** GRAT_loc) init_rat_counts2_loc.refine[extrloc_unfolds]
concrete-definition init_rat_counts2 **uses** GRAT_loc.init_rat_counts2_loc_def[unfolded extrloc_unfolds]
declare (**in** GRAT_loc) init_rat_counts2.refine[OF GRAT_loc_axioms, extrloc_unfolds]

concrete-definition (**in** GRAT_loc) verify_unsat2_loc
  uses verify_unsat1_bd_def[unfolded extrloc_unfolds]
declare (**in** GRAT_loc) verify_unsat2_loc.refine[extrloc_unfolds]
concrete-definition verify_unsat2 **uses** GRAT_loc.verify_unsat2_loc_def[unfolded extrloc_unfolds]
declare (**in** GRAT_loc) verify_unsat2.refine[OF GRAT_loc_axioms, extrloc_unfolds]

~~concrete_definition (in GRAT_loc) XXX2_loc uses XXX1_bd_def[unfolded extrloc_unfolds] declare (in GRAT_loc) XXX2_loc.refine[extrloc_unfolds] concrete_definition XXX2 uses GRAT_loc.XXX2_loc_def[unfolded extrloc_unfolds] declare (in GRAT_loc) XXX2.refine[OF GRAT_loc_axioms, extrloc_unfolds]~~

4.4.4 Synthesis of Imperative Code

definition creg_register_ndj l cid cr ≡ do {
  x ← array_get_dyn None cr (int_encode l);
  case x of
    None ⇒ return cr
  | Some s ⇒ array_set_dyn None cr (int_encode l) (Some (cid#s))
}

lemma creg_register_ndj_rule[sep_heap_rules]:
  ⟦ (i,l) ∈ lit_rel ⟧
  ⟹ <is_creg cr a>
        creg_register_ndj i cid a
      <is_creg (abs_cr_register_ndj l cid cr)>$_t$
  ⟨proof⟩

lemma *creg_register_hnr*[*sepref_fr_rules*]:
  (*uncurry2 creg_register_ndj*, *uncurry2* (*RETURN ooo abs_cr_register_ndj*))
    $\in$ (*pure lit_rel*)^k $*_a$ *nat_assn*^{k} $*_a$ *is_creg*^{d} $\rightarrow_a$ *is_creg*
  ⟨*proof*⟩

sepref-register *abs_cr_register_ndj* :: *nat literal* ⇒ *nat* ⇒ _
  :: *nat literal* ⇒ *nat* ⇒ (*nat literal*,*nat list*) *i_map*
      ⇒ (*nat literal*,*nat list*) *i_map*

context *GRAT_def_loc*
begin
  lemma *pr_next_hnr*[*sepref_import_param*]: (*prf_next*,*prf_next*) $\in$ *Id* $\rightarrow$ *Id* $\times_r$ *Id*
    ⟨*proof*⟩

  definition *prfi_assn* :: *nat* × *'prf* ⇒ _ **where** *prfi_assn* ≡ *id_assn*

  definition *prfn_assn* :: (*'prf* ⇒ *int* × *'prf*) ⇒ _ **where** *prfn_assn* ≡ *id_assn*

  abbreviation *errorp_assn*
    ≡ (*id_assn* :: *String.literal* ⇒ _) $\times_a$ *option_assn int_assn* $\times_a$ *option_assn prfi_assn*

  lemma *prfi_assn_pure*[*safe_constraint_rules*]: *is_pure prfi_assn* ⟨*proof*⟩

  term *prf_next*

end

sepref-decl-intf *'prf i_prfi* **is** *nat* × *'prf*
sepref-decl-intf *'prf i_prfn* **is** *'prf* ⇒ *int* × *'prf*

context
  fixes *DB* :: *clausedb2*
  fixes *frml_end* :: *nat*
  fixes *prf_next* :: *'prf* ⇒ *int* × *'prf*
begin
  interpretation *GRAT_def_loc DB frml_end prf_next* ⟨*proof*⟩

  abbreviation *state_assn'* ≡ *cm_assn* $\times_a$ *assignment_assn*
  type-synonym *i_state'* = *i_cm* × *i_assignment*

  term *parse_prf2* **thm** *parse_prf2_def*

  lemmas [*intf_of_assn*] =
    intf_of_assnI[**where** *R*=*prfi_assn* **and** *'a*=*'prf i_prfi*]
    intf_of_assnI[**where** *R*=*prfn_assn* **and** *'a*=*'prf i_prfn*]

  term *mkp_raw_err*
  lemma *mkp_raw_err_hnr*[*sepref_fr_rules*]:
    (*uncurry2* (*return ooo mkp_raw_err*), *uncurry2* (*RETURN ooo mkp_raw_err*))
      $\in$ *id_assn*^{k} $*_a$ (*option_assn int_assn*)^k $*_a$ (*option_assn prfi_assn*)^k $\rightarrow_a$ *errorp_assn*
    ⟨*proof*⟩

  sepref-register *mkp_raw_err* ::
    String.literal ⇒ *int option* ⇒ *'prf i_prfi option*
      ⇒ *String.literal* × *int option* × *'prf i_prfi option*

  definition *parse_prf_impl* (*prfn* :: *'prf* ⇒ *int* × *'prf*) ≡ λ(*fuel*::*nat*,*prf*).
    if fuel > *0 then do* {
      let (*x*,*prf*) = *prfn prf*;

return (Inr (x,(fuel−1,prf)))
} else
return (Inl (mkp_raw_err (STR ''Out of fuel'') None (Some (fuel, prf))))

lemma parse_prf_impl_hnr[sepref_fr_rules]:
(uncurry parse_prf_impl, uncurry parse_prf2) ∈ prfn_assn^k $*_a$ prfi_assn^d
$\rightarrow_a$ errorp_assn $+_a$ int_assn $\times_a$ prfi_assn
⟨proof⟩
sepref-register parse_prf2
:: 'prf i_prfn ⇒ 'prf i_prfi ⇒ ('prf i_prfi error + int × 'prf i_prfi) nres

term read_clause_check_taut2

sepref-definition read_clause_check_taut3 **is** uncurry3 read_clause_check_taut2
:: liti.a_assn^k $*_a$ liti.it_assn^k $*_a$ liti.it_assn^k $*_a$ assignment_assn^d
$\rightarrow_a$ errorp_assn $+_a$ liti.it_assn $\times_a$ bool_assn $\times_a$ assignment_assn
⟨proof⟩
lemmas [sepref_fr_rules] = read_clause_check_taut3.refine
sepref-register read_clause_check_taut2
:: int list ⇒ nat ⇒ nat ⇒ i_assignment
⇒ ('prf i_prfi error + nat × bool × i_assignment) nres

sepref-definition add_clause3 **is** uncurry3 add_clause2
:: liti.a_assn^k $*_a$ nat_assn^k $*_a$ liti.it_assn^k $*_a$ cm_assn^d $\rightarrow_a$ cm_assn
⟨proof⟩
sepref-register add_clause2 :: int list ⇒ nat ⇒ nat ⇒ i_cm ⇒ i_cm nres
lemmas [sepref_fr_rules] = add_clause3.refine

~~TODO: Why can we rewrite 1::nat to int→itype? Realized this oddity during debugging read_cnf_new2 sepref~~

sepref-definition read_cnf_new3 **is** uncurry3 read_cnf_new2
:: liti.a_assn^k $*_a$ liti.it_assn^k $*_a$ liti.it_assn^k $*_a$ cm_assn^d
$\rightarrow_a$ errorp_assn $+_a$ cm_assn
⟨proof⟩
sepref-register read_cnf_new2
:: int list ⇒ nat ⇒ nat ⇒ i_cm ⇒ ('prf i_prfi error + i_cm) nres
lemmas [sepref_fr_rules] = read_cnf_new3.refine

sepref-definition parse_check_blocked3 **is** uncurry2 parse_check_blocked2
:: liti.a_assn^k $*_a$ assignment_assn^d $*_a$ liti.it_assn^k
$\rightarrow_a$ errorp_assn $+_a$
liti.it_assn
$\times_a$ (assignment_assn $\times_a$ list_set_assn id_assn)
$\times_a$ liti.it_assn
⟨proof⟩

term parse_check_blocked2
sepref-register parse_check_blocked2
:: int list ⇒ i_assignment ⇒ nat
⇒ ('prf i_prfi error + nat × (i_assignment × nat set) × nat) nres
lemmas [sepref_fr_rules] = parse_check_blocked3.refine

sepref-definition check_unit_clause3 **is** uncurry2 check_unit_clause2
:: liti.a_assn^k $*_a$ assignment_assn^k $*_a$ (liti.it_assn)^k
$\rightarrow_a$ sum_assn errorp_assn (pure lit_rel)
⟨proof⟩
lemmas [sepref_fr_rules] = check_unit_clause3.refine
sepref-register check_unit_clause2
:: int list ⇒ i_assignment ⇒ nat ⇒ ('prf i_prfi error + nat literal) nres

sepref-definition resolve_id3 **is** uncurry resolve_id2

:: cm_assn^k $*_a$ nat_assn^k $\rightarrow_a$ sum_assn errorp_assn liti.it_assn
⟨proof⟩
term resolve_id2
sepref-register resolve_id2
:: (nat) clausemap1 ⇒ nat ⇒ _ :: i_cm ⇒ nat ⇒ ('prf i_prfi error + nat) nres
lemmas [sepref_fr_rules] = resolve_id3.refine

term apply_units2
sepref-definition apply_units3 **is** uncurry4 apply_units2
:: liti.a_assn^k $*_a$ prfn_assn^k $*_a$ cm_assn^k $*_a$ (assignment_assn)^d $*_a$ prfi_assn^d
$\rightarrow_a$ errorp_assn $+_a$ assignment_assn $\times_a$ prfi_assn
⟨proof⟩
sepref-register apply_units2 :: _ ⇒ _ ⇒ (nat) clausemap1 ⇒ _
:: int list ⇒ 'prf i_prfn ⇒ i_cm ⇒ i_assignment ⇒ 'prf i_prfi
⇒ ('prf i_prfi error + i_assignment × 'prf i_prfi) nres
lemmas [sepref_fr_rules] = apply_units3.refine

~~TODO: Use array-based list instead of list_set_assn!~~
sepref-definition apply_units3_bt **is** uncurry5 apply_units2_bt
:: liti.a_assn^k
$*_a$ prfn_assn^k
$*_a$ cm_assn^k
$*_a$ (assignment_assn)^d
$*_a$ (list_set_assn nat_assn)^d
$*_a$ prfi_assn^d
$\rightarrow_a$ errorp_assn $+_a$
(assignment_assn $\times_a$ list_set_assn nat_assn) $\times_a$ prfi_assn
⟨proof⟩

sepref-register apply_units2_bt :: _ ⇒ _ ⇒ (nat) clausemap1 ⇒ _
:: int list ⇒ 'prf i_prfn ⇒ i_cm ⇒ i_assignment ⇒ nat set ⇒ 'prf i_prfi
⇒ ('prf i_prfi error + (i_assignment × nat set) × 'prf i_prfi) nres
lemmas [sepref_fr_rules] = apply_units3_bt.refine

term remove_ids2
sepref-definition remove_ids3 **is** uncurry2 remove_ids2
:: prfn_assn^k $*_a$ cm_assn^d $*_a$ prfi_assn^d
$\rightarrow_a$ errorp_assn $+_a$ cm_assn $\times_a$ prfi_assn
⟨proof⟩
sepref-register remove_ids2 :: _ ⇒ (nat) clausemap1 ⇒ _
:: 'prf i_prfn ⇒ i_cm ⇒ 'prf i_prfi ⇒ ('prf i_prfi error + i_cm × 'prf i_prfi) nres
lemmas [sepref_fr_rules] = remove_ids3.refine

term check_conflict_clause2
sepref-definition check_conflict_clause3 **is** uncurry3 check_conflict_clause2
:: liti.a_assn^k $*_a$ prfi_assn^k $*_a$ assignment_assn^k $*_a$ liti.it_assn^k
$\rightarrow_a$ sum_assn errorp_assn unit_assn
⟨proof⟩
sepref-register check_conflict_clause2
:: int list ⇒ 'prf i_prfi ⇒ i_assignment ⇒ nat ⇒ ('prf i_prfi error + unit) nres
lemmas [sepref_fr_rules] = check_conflict_clause3.refine

term and_not_C_excl2
sepref-definition and_not_C_excl3 **is** uncurry3 and_not_C_excl2
:: liti.a_assn^k $*_a$ (assignment_assn)^d $*_a$ (liti.it_assn)^k $*_a$ (pure lit_rel)^k
$\rightarrow_a$ prod_assn assignment_assn (list_set_assn nat_assn)
⟨proof⟩
sepref-register and_not_C_excl2
:: int list ⇒ i_assignment ⇒ nat ⇒ nat literal
⇒ (i_assignment × nat set) nres
lemmas [sepref_fr_rules] = and_not_C_excl3.refine

sepref-definition *lit_in_clause_and_not_true3*
is *uncurry3 lit_in_clause_and_not_true2*
:: *liti.a_assn*^{k} $*_a$ (*assignment_assn*)^k $*_a$ *liti.it_assn*^{k} $*_a$ (*pure lit_rel*)^k
$\rightarrow_a$ *bool_assn*
⟨*proof*⟩
lemmas [*sepref_fr_rules*] = *lit_in_clause_and_not_true3.refine*
sepref-register *lit_in_clause_and_not_true2*
:: *int list* ⇒ (*nat*,*bool*) *i_map* ⇒ *nat* ⇒ *nat literal* ⇒ *bool nres*

sepref-definition *get_rat_candidates3* **is** *uncurry3 get_rat_candidates2*
:: *liti.a_assn*^{k} $*_a$ *cm_assn*^{k} $*_a$ (*assignment_assn*)^k $*_a$ (*pure lit_rel*)^k
$\rightarrow_a$ *sum_assn errorp_assn* (*list_set_assn nat_assn*)
⟨*proof*⟩

sepref-register *get_rat_candidates2*
:: *int list* ⇒ *i_cm* ⇒ *i_assignment* ⇒ *nat literal*
⇒ ('*prf i_prfi error* + *nat set*) *nres*
lemmas [*sepref_fr_rules*] = *get_rat_candidates3.refine*

sepref-definition *backtrack3* **is** *uncurry backtrack2*
:: (*assignment_assn*)^d $*_a$ (*list_set_assn nat_assn*)^k $\rightarrow_a$ *assignment_assn*
⟨*proof*⟩
sepref-register *backtrack2* :: (*nat*⇀*bool*) ⇒ _
:: *i_assignment* ⇒ *nat set* ⇒ *i_assignment nres*
lemmas [*sepref_fr_rules*] = *backtrack3.refine*

~~TODO: Make this a proper operation on CM?~~
lemma *not_in_cm_ids_unf*: *i*∉*cm_ids CM* ⟷ (*case CM of* (*CM*,_) ⇒ *is_None* (*CM i*))
⟨*proof*⟩

sepref-definition *check_rup_proof3* **is** *uncurry4 check_rup_proof2*
:: *liti.a_assn*^{k} $*_a$ *prfn_assn*^{k} $*_a$ (*state_assn'*)^d $*_a$ *liti.it_assn*^{k} $*_a$ *prfi_assn*^{d}
$\rightarrow_a$ *errorp_assn* $+_a$ *state_assn'* $\times_a$ *liti.it_assn* $\times_a$ *prfi_assn*
⟨*proof*⟩
sepref-register *check_rup_proof2*
:: *int list* ⇒ '*prf i_prfn* ⇒ *i_state'* ⇒ *nat* ⇒ '*prf i_prfi*
⇒ ('*prf i_prfi error* + *i_state'* × *nat* × '*prf i_prfi*) *nres*
lemmas [*sepref_fr_rules*] = *check_rup_proof3.refine*

term *lit_in_clause2*
sepref-definition *lit_in_clause3* **is** *uncurry2 lit_in_clause2*
:: *liti.a_assn*^{k} $*_a$ *liti.it_assn*^{k} $*_a$ *lit_assn*^{k} $\rightarrow_a$ *bool_assn*
⟨*proof*⟩
sepref-register *lit_in_clause2* :: *int list* ⇒ *nat* ⇒ *nat literal* ⇒ *bool nres*
lemmas [*sepref_fr_rules*] = *lit_in_clause3.refine*

term *check_rat_candidates_part2*
sepref-definition *check_rat_candidates_part3*
is *uncurry6 check_rat_candidates_part2* ::
liti.a_assn^k
$*_a$ *prfn_assn*^{k}
$*_a$ *cm_assn*^{k}
$*_a$ *lit_assn*^{k}
$*_a$ (*list_set_assn nat_assn*)^d
$*_a$ *assignment_assn*^{d}
$*_a$ *prfi_assn*^{d}
$\rightarrow_a$ *errorp_assn* $+_a$ (*assignment_assn* $\times_a$ *prfi_assn*)
⟨*proof*⟩
sepref-register *check_rat_candidates_part2* :: _ ⇒ _ ⇒ (*nat*) *clausemap1* ⇒ _
:: *int list* ⇒ '*prf i_prfn* ⇒ *i_cm* ⇒ *nat literal* ⇒ *nat set* ⇒ *i_assignment* ⇒ '*prf i_prfi*

$\Rightarrow$ ('prf i_prfi error + i_assignment × 'prf i_prfi) nres
lemmas [sepref_fr_rules] = check_rat_candidates_part3.refine

term check_rat_proof2
sepref-definition check_rat_proof3 **is** uncurry4 check_rat_proof2
:: liti.a_assn^k $*_a$ prfn_assn^k $*_a$ (state_assn')^d $*_a$ liti.it_assn^k $*_a$ prfi_assn^d
$\rightarrow_a$ errorp_assn $+_a$ state_assn' $\times_a$ liti.it_assn $\times_a$ prfi_assn
⟨proof⟩
sepref-register check_rat_proof2
:: int list ⇒ 'prf i_prfn ⇒ i_state' ⇒ nat ⇒ 'prf i_prfi
⇒ ('prf i_prfi error + i_state' × nat × 'prf i_prfi) nres
lemmas [sepref_fr_rules] = check_rat_proof3.refine

term check_item2
sepref-definition check_item3 **is** uncurry4 check_item2
:: liti.a_assn^k $*_a$ prfn_assn^k $*_a$ (state_assn')^d $*_a$ liti.it_assn^k $*_a$ prfi_assn^d
$\rightarrow_a$ errorp_assn $+_a$ option_assn (state_assn' $\times_a$ liti.it_assn $\times_a$ prfi_assn)
⟨proof⟩
sepref-register check_item2
:: int list ⇒ 'prf i_prfn ⇒ i_state' ⇒ nat ⇒ 'prf i_prfi
⇒ ('prf i_prfi error + (i_state' × nat × 'prf i_prfi) option) nres
lemmas [sepref_fr_rules] = check_item3.refine

term is_syn_taut2
sepref-definition is_syn_taut3 **is** uncurry2 is_syn_taut2
:: liti.a_assn^k $*_a$ liti.it_assn^k $*_a$ assignment_assn^d
$\rightarrow_a$ bool_assn $\times_a$ assignment_assn
⟨proof⟩
sepref-register is_syn_taut2
:: int list ⇒ nat ⇒ i_assignment ⇒ (bool × i_assignment) nres
lemmas [sepref_fr_rules] = is_syn_taut3.refine

~~term read_cnf2 sepref_definition read_cnf3 is uncurry2 read_cnf2 :: liti.a_assn^k $*_a$ (list_assn liti.it_assn)^k $*_a$ cm_assn^d $\rightarrow_a$ cm_assn $\times_a$ nat_assn unfolding read_cnf2_def supply [[goals_limit = 1, id_debug]] apply (rewrite at (_,1,□) assignment.fold_custom_empty) by sepref sepref_register read_cnf2 :: int list ⇒ nat list ⇒ i_cm ⇒ (i_cm × nat) nres lemmas [sepref_fr_rules] = read_cnf3.refine~~

~~term goto_next_item2 thm goto_next_item2_def **cancel** ‹ TODO: frml_end parameter only gets in by assertion! Remove assertion before! Currently solved by inlining goto_next_item2 ›~~

term init_rat_counts2
sepref-definition init_rat_counts3 **is** uncurry init_rat_counts2
:: prfn_assn^k $*_a$ prfi_assn^d $\rightarrow_a$ errorp_assn $+_a$ (cm_assn $\times_a$ prfi_assn)
⟨proof⟩
sepref-register init_rat_counts2
:: 'prf i_prfn ⇒ 'prf i_prfi ⇒ ('prf i_prfi error + i_cm × 'prf i_prfi) nres
lemmas [sepref_fr_rules] = init_rat_counts3.refine

term verify_unsat2
sepref-definition verify_unsat3 **is** uncurry5 verify_unsat2
:: liti.a_assn^k
$*_a$ prfn_assn^k
$*_a$ liti.it_assn^k
$*_a$ liti.it_assn^k
$*_a$ liti.it_assn^k
$*_a$ prfi_assn^d
$\rightarrow_a$ errorp_assn $+_a$ unit_assn
⟨proof⟩

end

definition verify_unsat_split_impl_wrapper DBi prf_next F_end it prf ≡ do {

```
  lenDBi ← Array.len DBi;

  if (0 < F_end ∧ F_end ≤ lenDBi ∧ 0 < it ∧ it ≤ lenDBi) then
    verify_unsat3 DBi prf_next 1 F_end it prf
  else
    return (Inl (STR ''Invalid arguments'',None,None))
}

lemmas [code] = DB2_def_loc.item_next_impl_def
export-code verify_unsat_split_impl_wrapper checking SML_imp
```

4.5 Correctness Theorem

```
context GRAT_loc begin

  lemma verify_unsat3_correct_aux[sep_heap_rules]:
    ~~assumes F_ref: (Fi,F) ∈ ⟨cref_rel⟩list_rel~~
    assumes SEG: liti.seg F_begin lst F_end
    assumes itI[simp]: it_invar F_end it_invar it
    shows
      <DBi ↦a DB>
        verify_unsat3 DBi prf_next F_begin F_end it prf
      <λr. DBi ↦a DB * ↑(¬isl r ⟶ F_invar lst ∧ ¬sat (F_α lst))>t
  ⟨proof⟩
      applyS (sep_auto simp: prfi_assn_def prfn_assn_def pure_def)
      applyS (sep_auto dest!: 1 simp: sum.disc_eq_case split: sum.splits)
      applyS (simp add: I_begin)
      ⟨proof⟩
end
```

Main correctness theorem: Given an array *DBi* that contains the integers *DB*, the verification algorithm does not change the array, and if it returns a non-*Inl* value, the formula in the array is unsatisfiable.

```
theorem verify_unsat_split_impl_wrapper_correct[sep_heap_rules]:
  shows
    <DBi ↦a DB>
      verify_unsat_split_impl_wrapper DBi prf_next F_end it prf
    <λresult. DBi ↦a DB * ↑(¬isl result ⟶ verify_unsat_spec DB F_end)>t
⟨proof⟩

end
```

5 Satisfiability Check

```
theory Sat_Check
imports Grat_Basic
begin
```

5.1 Abstract Specification

```
  locale sat_input = input it_invar' it_next it_peek it_end for it_invar' :: 'it::linorder ⇒ bool
    and it_next it_peek it_end

  context sat_input begin

    definition read_assignment it ≡ doE {
      let A = Map.empty;
      check_not_end it;
      (A,_) ← EWHILEIT (λ(_,it). it_invar it ∧ it≠it_end) (λ(_,it). it_peek it ≠ litZ) (λ(A,it). doE {
        (l,it) ← parse_literal it;
        check_not_end it;
        CHECK (sem_lit' l A ≠ Some False) (mk_errit STR ''Contradictory assignment'' it);
        let A = assign_lit A l;
        ERETURN (A,it)
```

```
    }) (A,it);
    ERETURN A
  }
```

We merely specify that this does not fail, i.e. termination and assertions.

```
  lemma read_assignment_correct[THEN ESPEC_trans, refine_vcg]:
    it_invar it ⟹ read_assignment it ≤ ESPEC (λ_. True) (λ_. True)
    ⟨proof⟩

  definition read_clause_check_sat itE it A ≡ doE {
    EASSERT (it_invar it ∧ it_invar itE ∧ itran itE it_end);
    parse_lz
      (mk_errit STR ''Parsed beyond end'' it)
      litZ itE it (λ_. True) (λx r. doE {
        let l = lit_α x;
        ERETURN (r ∨ (sem_lit' l A = Some True))
      }) False
  }

  lemma read_clause_check_sat_correct[THEN ESPEC_trans, refine_vcg]:
    ⟦itran it itE; it_invar itE⟧ ⟹
    read_clause_check_sat itE it A
    ≤ ESPEC
        (λ_. True)
        (λ(it',r). ∃l. lz_string litZ it l it' ∧ itran it' itE
                     ∧ (r ⟷ sem_clause' (clause_α l) A = Some True))
    ⟨proof⟩


  definition check_sat it itE A ≡ doE {
    tok_fold itE it (λit _. doE {
      (it',r) ← read_clause_check_sat itE it A;
      CHECK (r) (mk_errit STR ''Clause not satisfied by given assignment'' it);
      ERETURN (it',())
    }) ()
  }

  term sem_cnf


  lemma obtain_compat_assignment: obtains σ where compat_assignment A σ
  ⟨proof⟩

  lemma check_sat_correct[THEN ESPEC_trans, refine_vcg]:
    ⟦seg it lst itE; it_invar itE⟧ ⟹ check_sat it itE A
    ≤ ESPEC (λ_. True) (λ_. F_invar lst ∧ sat (F_α lst))
    ⟨proof⟩

  definition verify_sat F_begin F_end it ≡ doE {
    A ← read_assignment it;
    check_sat F_begin F_end A
  }

  lemma verify_sat_correct[THEN ESPEC_trans, refine_vcg]:
    ⟦seg F_begin lst F_end; it_invar F_end; it_invar it⟧
      ⟹ verify_sat F_begin F_end it ≤ ESPEC (λ_. True) (λ_. F_invar lst ∧ sat (F_α lst))
    ⟨proof⟩

end
```

5.2 Implementation

```
context sat_input begin
```

5.2.1 Getting Out of Exception Monad

synth-definition *read_assignment_bd* **is** [*enres_unfolds*]: *read_assignment it* = ⌑
⟨*proof*⟩

synth-definition *read_clause_check_sat_bd* **is** [*enres_unfolds*]: *read_clause_check_sat itE it A* = ⌑
⟨*proof*⟩

synth-definition *check_sat_bd* **is** [*enres_unfolds*]: *check_sat it itE* = ⌑
⟨*proof*⟩

synth-definition *verify_sat_bd* **is** [*enres_unfolds*]: *verify_sat F_begin F_end it* = ⌑
⟨*proof*⟩

end

5.3 Extraction from Locales

named-theorems *extrloc_unfolds*

context *DB2_loc* **begin**
sublocale *sat_input liti.I liti.next liti.peek liti.end*
⟨*proof*⟩
end

concrete-definition (**in** *DB2_loc*) *read_assignment2_loc*
uses *read_assignment_bd_def*[*unfolded extrloc_unfolds*]
declare (**in** *DB2_loc*) *read_assignment2_loc.refine*[*extrloc_unfolds*]
concrete-definition *read_assignment2* **uses** *DB2_loc.read_assignment2_loc_def*[*unfolded extrloc_unfolds*]
declare (**in** *DB2_loc*) *read_assignment2.refine*[*OF DB2_loc_axioms*, *extrloc_unfolds*]

concrete-definition (**in** *DB2_loc*) *read_clause_check_sat2_loc*
uses *read_clause_check_sat_bd_def*[*unfolded extrloc_unfolds*]
declare (**in** *DB2_loc*) *read_clause_check_sat2_loc.refine*[*extrloc_unfolds*]
concrete-definition *read_clause_check_sat2* **uses** *DB2_loc.read_clause_check_sat2_loc_def*[*unfolded extrloc_unfolds*]
declare (**in** *DB2_loc*) *read_clause_check_sat2.refine*[*OF DB2_loc_axioms*, *extrloc_unfolds*]

concrete-definition (**in** *DB2_loc*) *check_sat2_loc*
uses *check_sat_bd_def*[*unfolded extrloc_unfolds*]
declare (**in** *DB2_loc*) *check_sat2_loc.refine*[*extrloc_unfolds*]
concrete-definition *check_sat2* **uses** *DB2_loc.check_sat2_loc_def*[*unfolded extrloc_unfolds*]
declare (**in** *DB2_loc*) *check_sat2.refine*[*OF DB2_loc_axioms*, *extrloc_unfolds*]

concrete-definition (**in** *DB2_loc*) *verify_sat2_loc*
uses *verify_sat_bd_def*[*unfolded extrloc_unfolds*]
declare (**in** *DB2_loc*) *verify_sat2_loc.refine*[*extrloc_unfolds*]
concrete-definition *verify_sat2* **uses** *DB2_loc.verify_sat2_loc_def*[*unfolded extrloc_unfolds*]
declare (**in** *DB2_loc*) *verify_sat2.refine*[*OF DB2_loc_axioms*, *extrloc_unfolds*]

5.3.1 Synthesis of Imperative Code

context
fixes *DB* :: *clausedb2*
fixes *frml_end* :: *nat*
begin
interpretation *DB2_def_loc DB frml_end* ⟨*proof*⟩

term *read_assignment2*

sepref-definition *read_assignment3* **is** *uncurry read_assignment2*
:: $liti.a\_assn^k *_a liti.it\_assn^k \rightarrow_a error\_assn +_a assignment\_assn$
⟨*proof*⟩

sepref-register *read_assignment2* :: *int list* ⇒ *nat* ⇒ (*nat error* + *i_assignment*) *nres*

lemmas [*sepref_fr_rules*] = *read_assignment3.refine*

term *read_clause_check_sat2*
sepref-definition *read_clause_check_sat3* **is** *uncurry3 read_clause_check_sat2*
:: $liti.a\_assn^k *_a liti.it\_assn^k *_a liti.it\_assn^k *_a assignment\_assn^k \to_a error\_assn +_a liti.it\_assn \times_a bool\_assn$
⟨*proof*⟩
sepref-register *read_clause_check_sat2* :: *int list* ⇒ *nat* ⇒ *nat* ⇒ *i_assignment* ⇒ (*nat error* + *nat*×*bool*) *nres*
lemmas [*sepref_fr_rules*] = *read_clause_check_sat3.refine*

term *check_sat2*
sepref-definition *check_sat3* **is** *uncurry3 check_sat2*
:: $liti.a\_assn^k *_a liti.it\_assn^k *_a liti.it\_assn^k *_a assignment\_assn^k \to_a error\_assn +_a unit\_assn$
⟨*proof*⟩
sepref-register *check_sat2* :: *int list* ⇒ *nat* ⇒ *nat* ⇒ *i_assignment* ⇒ (*nat error* + *unit*) *nres*
lemmas [*sepref_fr_rules*] = *check_sat3.refine*

term *verify_sat2*
sepref-definition *verify_sat3* **is** *uncurry3 verify_sat2*
:: $liti.a\_assn^k *_a liti.it\_assn^k *_a liti.it\_assn^k *_a liti.it\_assn^k \to_a error\_assn +_a unit\_assn$
⟨*proof*⟩
sepref-register *verify_sat2* :: *int list* ⇒ *nat* ⇒ *nat* ⇒ *nat* ⇒ (*nat error* + *unit*) *nres*
lemmas [*sepref_fr_rules*] = *verify_sat3.refine*

end

definition *verify_sat_impl_wrapper DBi F_end* ≡ *do* {
lenDBi ← *Array.len DBi*;
if (*0*<*F_end* ∧ *F_end* ≤ *lenDBi*) *then*
verify_sat3 DBi 1 F_end F_end
else
return (*Inl* (*STR ''Invalid arguments''*,*None*,*None*))

}

export-code *verify_sat_impl_wrapper* **checking** *SML_imp*

5.4 Correctness Theorem

context *DB2_loc* **begin**
lemma *verify_sat3_correct*:
assumes *SEG*: *liti.seg F_begin lst F_end*
assumes *itI*[*simp*]: *it_invar F_end it_invar it*
shows <*DBi* $\mapsto_a$ *DB*> *verify_sat3 DBi F_begin F_end it* <λ*r*. *DBi* $\mapsto_a$ *DB* ∗ ↑(¬*isl r* ⟶ *F_invar lst* ∧ *sat* (*F_α lst*)) $>_t$
⟨*proof*⟩
applyS *sep_auto*
applyS (*sep_auto dest*!: *1 simp*: *sum.disc_eq_case split*: *sum.splits*)
applyS (*simp add*: *I_begin*)
⟨*proof*⟩

end

theorem *verify_sat_impl_wrapper_correct*[*sep_heap_rules*]:
shows
<*DBi* $\mapsto_a$ *DB*>
verify_sat_impl_wrapper DBi F_end
<λ*result*. *DBi* $\mapsto_a$ *DB* ∗ ↑(¬*isl result* ⟶ *verify_sat_spec DB F_end*)$>_t$
⟨*proof*⟩

end

6 Code Generation and Summary of Correctness Theorems

theory *Grat_Check_Code_Exporter*
imports *Unsat_Check Unsat_Check_Split_MM Sat_Check*
begin

6.1 Code Generation

We generate code for *verify_unsat_impl_wrapper* and *verify_sat_impl_wrapper*.

The first statement is a sanity check, that will make our automated regression tests fail if the generated code does not compile.

The second statement actually exports the two main functions, and some auxiliary functions to convert between SML and Isabelle integers, and to access the sum data type of Isabelle, which is used to encode the checker's result.

export-code
verify_unsat_impl_wrapper
verify_unsat_split_impl_wrapper
verify_sat_impl_wrapper
checking *SML_imp*

export-code
verify_sat_impl_wrapper
verify_unsat_impl_wrapper
verify_unsat_split_impl_wrapper
int_of_integer
integer_of_int
integer_of_nat
nat_of_integer

isl projl projr Inr Inl Pair
in *SML_imp* **module-name** *Grat_Check* **file** *code/gratchk_export.sml*

6.2 Summary of Correctness Theorems

In this section, we summarize the correctness theorems for our checker

The precondition of the triples just state that their is an integer array, which contains the DIMACS representation of the formula in the segment from indexes $[1..<F\_end]$. The postcondition states that the array is not changed, and, if the checker does not fail, the *F_end* index will be in range, the DIMACS representation of the formula is valid, and the represented formula is satisfiable or unsatisfiable, respectively.

Note that this only proved soundness of the checker, that is, the checker may always fail, but if it does not, we guarantee a valid and (un)satisfiable formula.

theorem
$<DBi \mapsto_a DB>$
verify_sat_impl_wrapper DBi F_end
$<\lambda result.\ DBi \mapsto_a DB * \uparrow(\neg isl\ result \longrightarrow verify\_sat\_spec\ DB\ F\_end)>_t$
⟨*proof*⟩

theorem
$<DBi \mapsto_a DB>$
verify_unsat_impl_wrapper DBi F_end it
$<\lambda result.\ DBi \mapsto_a DB * \uparrow(\neg isl\ result \longrightarrow verify\_unsat\_spec\ DB\ F\_end)>_t$
⟨*proof*⟩

theorem
shows
$<DBi \mapsto_a DB>$
verify_unsat_split_impl_wrapper DBi prf_next F_end it prf
$<\lambda result.\ DBi \mapsto_a DB * \uparrow(\neg isl\ result \longrightarrow verify\_unsat\_spec\ DB\ F\_end)>_t$
⟨*proof*⟩

The specifications for a formula being valid and satisfiable/unsatisfiable can be written up in a very concise way, only relying on basic list operations and the notion of a consistent assignment of truth values to integers.

An assignment is consistent, if each non-zero integer is assigned the opposite of its negated value.

lemma $assn\_consistent\ \sigma \longleftrightarrow (\forall\, l.\ l{\neq}0 \longrightarrow \sigma\ l = (\neg\ \sigma\ (-l)))$
$\langle proof \rangle$

The input described a valid and satisfiable formula, iff the F_end index is in range, the corresponding DIMACS string is empty or ends with a zero, and there is a consistent assignment such that each represented clause contains a true literal.

lemma
$verify\_sat\_spec\ DB\ F\_end \equiv 1{\leq}F\_end \wedge F\_end \leq length\ DB \wedge ($
$let\ lst = tl\ (take\ F\_end\ DB)\ in$
$(lst{\neq}[] \longrightarrow last\ lst = 0)$
$\wedge (\exists\, \sigma.\ assn\_consistent\ \sigma \wedge (\forall\, C{\in}set\ (tokenize\ 0\ lst).\ \exists\, l{\in}set\ C.\ \sigma\ l)))$
$\langle proof \rangle$

The input describes a valid and unsatisfiable formula, iff F_end is in range and does not describe the empty DIMACS string, the DIMACS string ends with zero, and there exists no consistent assignment such that every clause contains at least one literal assigned to true.

lemma
$verify\_unsat\_spec\ DB\ F\_end \equiv 1 < F\_end \wedge F\_end \leq length\ DB \wedge ($
$let\ lst = tl\ (take\ F\_end\ DB)\ in$
$last\ lst = 0$
$\wedge (\nexists\, \sigma.\ assn\_consistent\ \sigma \wedge (\forall\, C{\in}set\ (tokenize\ 0\ lst).\ \exists\, l{\in}set\ C.\ \sigma\ l)))$
$\langle proof \rangle$

end