GRATchk: Verified (UN)SAT Certificate Checker

Peter Lammich

September 28, 2020

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

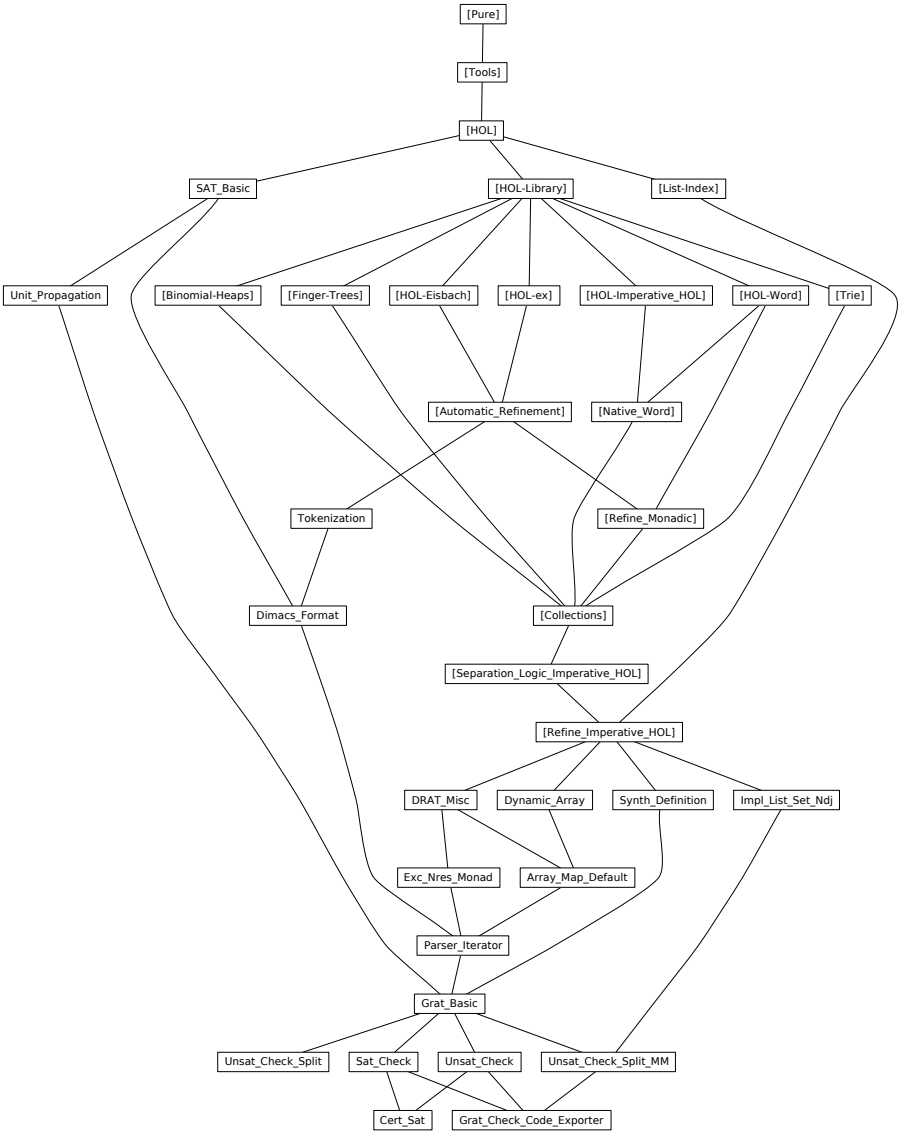

Figure 1: Theory dependency graph

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)
  by (*cases l*) (*auto simp*: *option.map_comp o_def option.map_ident*)

lemma (**in** −) *sem_lit'_empty*[*simp*]: *sem_lit' l Map.empty = None*
  by (*cases l*) *auto*

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*
  by (*cases a*) *auto*

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))
  by (auto simp: sem_clause'_def split: option.split bool.split)


lemma sem_clause_empty[simp]: sem_clause' {} A = Some False
  unfolding sem_clause'_def by auto

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
  by (auto simp: sem_clause_insert_eq_complete split: option.split bool.split)

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
  unfolding sem_clause'_def by auto

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
  unfolding sem_clause'_def by auto

lemma compat_assignment_empty[simp]: compat_assignment Map.empty σ
  unfolding compat_assignment_def by simp
```

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)
  unfolding assign_lit_def by auto

lemma assign_lit_dom[simp]:
  dom (assign_lit A l) = insert (var_of_lit l) (dom A)
  unfolding assign_lit_def by auto

lemma sem_lit_assign[simp]: sem_lit' l (assign_lit A l) = Some True
  unfolding assign_lit_def by (cases l) auto

lemma sem_lit'_none_conv: sem_lit' l A = None ⟷ A (var_of_lit l) = None
  by (cases l) auto

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
  unfolding assign_lit_def
  apply (cases l)
  apply auto
  apply (cases l'; auto)
  apply (cases l'; clarsimp)
  done

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
  unfolding sem_clause'_def
  by (force split: if_split_asm simp: assign_undec_pres_dec_lit)
```

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)
unfolding *assign_lit_def*
by (*cases l*; *cases l'*; *auto*)

Predicates for unit clauses

definition *is_unit_lit A C l*
≡ l∈C ∧ sem_lit' l A = None ∧ (sem_clause' (C−{l}) A = Some False)
definition *is_unit_clause A C ≡ ∃ l. is_unit_lit A C l*
definition *the_unit_lit A C ≡ THE l. is_unit_lit A C l*

abbreviation (*input*) *is_conflict_clause A C ≡ sem_clause' C A = Some False*
abbreviation (*input*) *is_true_clause A C ≡ sem_clause' C A = Some True*

lemma *sem_clause'_false_conv*:
sem_clause' C A = Some False ⟷ (∀ l∈C. sem_lit' l A = Some False)
unfolding *sem_clause'_def* **by** *auto*

lemma *sem_clause'_true_conv*:
sem_clause' C A = Some True ⟷ (∃ l∈C. sem_lit' l A = Some True)
unfolding *sem_clause'_def* **by** *auto*

lemma *the_unit_lit_eq*[*simp*]: *is_unit_lit A C l ⟹ the_unit_lit A C = l*
unfolding *is_unit_lit_def the_unit_lit_def sem_clause'_false_conv*
by *force*

lemma *is_unit_lit_unique*: ⟦*is_unit_lit C A l1*; *is_unit_lit C A l2*⟧ *⟹ l1=l2*
using *the_unit_lit_eq* **by** *blast*

lemma *is_unit_clauseE*:
assumes *is_unit_clause A C*
obtains *l C'* **where**
C=insert l C'
l∉C'
sem_lit' l A = None
sem_clause' C' A = Some False
the_unit_lit A C = l
using *assms*
proof −
from *assms* **obtain** *l* **where** *IUL*: *is_unit_lit A C l*
unfolding *is_unit_clause_def* **by** *blast*
note [*simp*] = *the_unit_lit_eq*[*OF IUL*]

from *IUL*
have *1*: *l∈C sem_lit' l A = None sem_clause' (C−{l}) A = Some False*
unfolding *is_unit_lit_def* **by** *blast+*
show *thesis*
apply (*rule that*[*of l C−{l}*])
using *1*
by *auto*
qed

lemma *is_unit_clauseE'*:
assumes *is_unit_clause A C*
obtains *l C'* **where**
C=insert l C'
l∉C'
sem_lit' l A = None
sem_clause' C' A = Some False
by (*rule is_unit_clauseE*[*OF assms*])

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 $l'=l$
by (*metis assms insert_Diff insert_iff*
is_unit_lit_def sem_clause'_insert_false)

lemma *sem_none_the_unit_lit*:
assumes *is_unit_lit A C l*
assumes $l' \in C$
assumes *sem_lit' l' A* = *None*
shows $l'=l$
using *sem_not_false_the_unit_lit*[*OF assms*(*1*,*2*)] *assms*(*3*) **by** *auto*

lemma *is_unit_lit_unique_ss*:
$[\![C' \subseteq C$; *is_unit_lit A C' l'*; *is_unit_lit A C l*$]\!] \Longrightarrow l'=l$
by (*simp add*: *is_unit_lit_def sem_none_the_unit_lit subsetD*)

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*
by (*auto simp*: *is_unit_lit_def*)

lemma *is_unit_clauseI*: *is_unit_lit A C l* $\Longrightarrow$ *is_unit_clause A C*
by (*auto simp*: *is_unit_clause_def*)

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*
using *assms* **by** (*auto simp*: *is_unit_lit_def sem_clause'_false_conv*)

lemma *unit_clause_sem'*: *is_unit_lit A C l* $\Longrightarrow$ *sem_clause' C A* = *None*
unfolding *is_unit_lit_def sem_clause'_def*
using *mk_disjoint_insert* **by** (*fastforce split*: *if_split_asm*)

lemma *unit_clause_assign_dec*:
is_unit_lit A C l $\Longrightarrow$ *sem_clause' C* (*assign_lit A l*) = *Some True*
unfolding *is_unit_lit_def sem_clause'_def*
by (*force split*: *if_split_asm simp*: *sem_lit'_assign_conv*)

lemma *unit_clause_sem*: *is_unit_clause A C* $\Longrightarrow$ *sem_clause' C A* = *None*
by (*auto simp*: *is_unit_clause_def unit_clause_sem'*)

lemma *sem_not_unit_clause*: *sem_clause' C A* $\neq$ *None* $\Longrightarrow$ $\neg$*is_unit_clause A C*
by (*auto simp*: *is_unit_clause_def unit_clause_sem'*)

lemma *unit_contains_no_true*:
assumes *is_unit_clause A C*
assumes $l \in C$
shows *sem_lit' l A* $\neq$ *Some True*
using *assms* **unfolding** *is_unit_clause_def is_unit_lit_def*
by (*force simp*: *sem_clause'_false_conv*)

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 $\neg$*is_unit_clause A C*
using *assms*
unfolding *is_unit_clause_def is_unit_lit_def*
by (*auto simp*: *sem_clause'_false_conv*)

lemma *conflict_clause_assign_indep*:
assumes *sem_clause'* *C* (*assign_lit* *A* *l*) = *Some False*
assumes *neg_lit* *l* ∉ *C*
shows *sem_clause'* *C* *A* = *Some False*
using *assms*
by (*auto simp*: *sem_clause'_def* *sem_lit'_assign_conv* *split*: *if_split_asm*)

lemma *sem_lit'_assign_undec_conv*:
sem_lit' *l'* (*assign_lit* *A* *l*) = *None*
⟷ *sem_lit'* *l'* *A* = *None* ∧ *var_of_lit* *l* ≠ *var_of_lit* *l'*
by (*cases* *l*; *cases* *l'*; *auto*)

lemma *unit_clause_assign_indep*:
assumes *is_unit_clause* (*assign_lit* *A* *l*) *C*
assumes *neg_lit* *l* ∉ *C*
shows *is_unit_clause* *A* *C*
using *assms*
unfolding *is_unit_clause_def* *is_unit_lit_def*
by (*auto*
dest!: *conflict_clause_assign_indep*
simp: *sem_lit'_assign_undec_conv*)

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* ∉ *C* *sem_clause'* *C* *A* = *Some False*
| *neg_lit* *l* ∈ *C* *sem_clause'* (*C*−{*neg_lit* *l*}) *A* = *Some False*
proof (*cases*)
assume *A*: *neg_lit* *l* ∈ *C*
with *assms* **have** *sem_clause'* (*C*−{*neg_lit* *l*}) *A* = *Some False*
by (*auto simp*: *sem_clause'_def* *sem_lit'_assign_conv* *split*: *if_split_asm*)
with *A* **show** *?thesis* **by** (*rule that*)
next
assume *A*: *neg_lit* *l* ∉ *C*
with *assms* **have** *sem_clause'* *C* *A* = *Some False*
by (*auto simp*: *sem_clause'_def* *sem_lit'_assign_conv* *split*: *if_split_asm*)
with *A* **show** *?thesis* **by** (*rule that*)
qed

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* ∉ *C* *is_unit_clause* *A* *C*
| *neg_lit* *l* ∈ *C*
proof (*cases*)
assume *neg_lit* *l* ∈ *C* **thus** *?thesis* **by** (*rule that*)
next
assume *A*: *neg_lit* *l* ∉ *C*
from *assms* **obtain** *lu* *C'* **where**
[*simp*]: *C* = *insert* *lu* *C'* *lu*∉*C'*
and *LUN*: *sem_lit'* *lu* (*assign_lit* *A* *l*) = *None*
and *SCF*: *sem_clause'* *C'* (*assign_lit* *A* *l*) = *Some False*
by (*blast elim*: *is_unit_clauseE*)

from *clause_assign_false_cases*[*OF* *SCF*] *A*
have *sem_clause'* *C'* *A* = *Some False* **by** *auto*
moreover from *LUN* **have** *sem_lit'* *lu* *A* = *None*
by (*simp add*: *sem_lit'_assign_undec_conv*)
ultimately have *is_unit_clause* *A* *C*
by (*auto simp*: *is_unit_clause_def* *is_unit_lit_def*)
with *A* **show** *?thesis* **by** (*rule that*)
qed

lemma *sem_clause_ins_assign_not_false*[*simp*]:

sem_clause′ (insert l C) (assign_lit A l) ≠ Some False
unfolding *sem_clause′_def* **by** *auto*

lemma *sem_clause_ins_assign_not_unit*[*simp*]:
¬is_unit_clause (assign_lit A l) (insert l C′)
apply (*clarsimp simp*: *is_unit_clause_def is_unit_lit_def sem_lit′_assign_undec_conv sem_clause′_false_conv*)
apply *force*
done

context
fixes *A* :: *′a ⇀ bool* **and** *σ* :: *′a ⇒ bool*
assumes *C*: *compat_assignment A σ*
begin
lemma *compat_lit*: *sem_lit′ l A = Some v ⟹ sem_lit l σ = v*
using *C*
by (*cases l*) (*auto simp*: *compat_assignment_def*)

lemma *compat_clause*: *sem_clause′ C A = Some v ⟹ sem_clause C σ = v*
unfolding *sem_clause_def sem_clause′_def*
by (*force simp*: *compat_lit split*: *if_split_asm*)
end

2.1.1 Models, Equivalence, and Redundancy

definition *models′ F A ≡ { σ. compat_assignment A σ ∧ sem_cnf F σ}*
definition *sat′ F A ≡ models′ F A ≠ {}*
definition *equiv′ F A A′ ≡ models′ F A = models′ F A′*

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

lemma *models′ F A = models F ∩ Collect* (*compat_assignment A*)
unfolding *models′_def models_def* **by** *auto*

lemma *equiv′_refl*[*simp*]: *equiv′ F A A* **unfolding** *equiv′_def* **by** *simp*
lemma *equiv′_sym*: *equiv′ F A A′ ⟹ equiv′ F A′ A*
unfolding *equiv′_def* **by** *simp*
lemma *equiv′_trans*[*trans*]: ⟦ *equiv′ F A B*; *equiv′ F B C* ⟧ *⟹ equiv′ F A C*
unfolding *equiv′_def* **by** *simp*

lemma *models_antimono*: *C′⊆C ⟹ models′ C A ⊆ models′ C′ A*
unfolding *models′_def* **by** (*auto simp*: *sem_cnf_def*)

lemma *conflict_clause_imp_no_models*:
⟦ *C∈F*; *is_conflict_clause A C* ⟧ *⟹ models′ F A = {}*
by (*auto simp*: *models′_def sem_cnf_def dest*: *compat_clause*)

lemma *sat′_empty_iff*[*simp*]: *sat′ F Map.empty = sat F*
unfolding *sat′_def sat_def models′_def*
by *auto*

lemma *sat′_antimono*: *F⊆F′ ⟹ sat′ F′ A ⟹ sat′ F A*
unfolding *sat′_def* **using** *models_antimono* **by** *blast*

lemma *sat′_equiv*: *equiv′ F A A′ ⟹ sat′ F A = sat′ F A′*
unfolding *equiv′_def sat′_def* **by** *blast*

lemma *sat_iff_sat′*: *sat F ⟷ (∃ A. sat′ F A)*
by (*metis* (*no_types*, *lifting*) *Collect_empty_eq models′_def models_def*
sat′_def sat′_empty_iff sat_iff_has_models)

definition *implied_clause F A C ≡ 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*: $redundant\_clause\ F\ A\ C \longleftrightarrow sat'\ (insert\ C\ F)\ A = sat'\ F\ A$
unfolding *redundant_clause_def sat'_def* **by** *blast*

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*
using *assms* **unfolding** *redundant_clause_def models'_def*
by *auto*

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*
using *assms* **unfolding** *implied_clause_def models'_def*
by *auto*

lemma *implied_is_redundant*: $implied\_clause\ F\ A\ C \Longrightarrow redundant\_clause\ F\ A\ C$
unfolding *implied_clause_def redundant_clause_def* **by** *blast*

lemma *add_redundant_sat_iff*[*simp*]:
$redundant\_clause\ F\ A\ C \Longrightarrow sat'\ (insert\ C\ F)\ A = sat'\ F\ A$
unfolding *redundant_clause_def sat'_def* **by** *auto*

lemma *true_clause_implied*:
$sem\_clause'\ C\ A = Some\ True \Longrightarrow implied\_clause\ F\ A\ C$
unfolding *implied_clause_def models'_def*
by (*auto simp*: *compat_clause*)

lemma *equiv'_map_empty_sym*:
$NO\_MATCH\ Map.empty\ A \Longrightarrow equiv'\ F\ Map.empty\ A \longleftrightarrow equiv'\ F\ A\ Map.empty$
using *equiv'_sym* **by** *auto*

lemma *tautology*: $[\![l{\in}C;\ neg\_lit\ l \in C]\!] \Longrightarrow sem\_clause\ C\ \sigma$
by (*cases sem_lit l σ*; *cases l*; *force simp*: *sem_clause_def*)

lemma *implied_taut*: $[\![l{\in}C;\ neg\_lit\ l \in C]\!] \Longrightarrow implied\_clause\ F\ A\ C$
unfolding *implied_clause_def models'_def* **using** *tautology*[*of l C*]
by *auto*

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*:
$is\_blocked\ A\ C \longleftrightarrow sem\_clause'\ C\ A = Some\ True \vee C \cap neg\_lit\ `\ C \neq \{\}$
unfolding *is_syn_taut_def is_blocked_def* **by** *auto*

lemma *is_syn_taut_empty*[*simp*]: $\neg is\_syn\_taut\ \{\}$
by (*auto simp*: *is_syn_taut_def*)

lemma *is_syn_taut_conv*: $is\_syn\_taut\ C \longleftrightarrow (\exists l.\ l{\in}C \wedge neg\_lit\ l \in C)$
unfolding *is_syn_taut_def* **by** *auto*

lemma *empty_not_blocked*[*simp*]: $\neg is\_blocked\ A\ \{\}$
unfolding *is_blocked_alt* **by** (*auto simp*: *sem_clause'_true_conv*)

lemma *is_blocked_insert_iff*:
$is\_blocked\ A\ (insert\ l\ C)$
$\longleftrightarrow is\_blocked\ A\ C \vee sem\_lit'\ l\ A = Some\ True \vee neg\_lit\ l \in C$
by (*auto simp*: *is_blocked_alt sem_clause'_true_conv*)

lemma *is_blockedI1*: $[\![l{\in}C;\ sem\_lit'\ l\ A = Some\ True]\!] \Longrightarrow is\_blocked\ A\ C$
by (*auto simp*: *is_blocked_def sem_clause'_true_conv*)

lemma *is_blockedI2*: $[\![l{\in}C;\ neg\_lit\ l \in C]\!] \Longrightarrow is\_blocked\ A\ C$
by (*auto simp*: *is_blocked_def is_syn_taut_def*)

lemma *syn_taut_true*[*simp*]: $is\_syn\_taut\ C \Longrightarrow sem\_clause\ C\ \sigma = True$
apply (*auto simp*: *sem_clause_def is_syn_taut_def*)
using *sem_neg_lit* **by** *blast*

lemma *syn_taut_imp_blocked*: $is\_syn\_taut\ C \Longrightarrow is\_blocked\ A\ C$
unfolding *is_blocked_def* **by** *auto*

lemma *blocked_redundant*: $is\_blocked\ A\ C \Longrightarrow redundant\_clause\ F\ A\ C$
unfolding *is_blocked_alt*
using *implied_is_redundant implied_taut true_clause_implied* **by** *fastforce*

lemma *blocked_clause_true*:
$[\![is\_blocked\ A\ C;\ compat\_assignment\ A\ \sigma]\!] \Longrightarrow sem\_clause\ C\ \sigma$
proof −
assume *a1*: *compat_assignment A* σ
assume *is_blocked A C*
then have *f2*: $sem\_clause'\ C\ A = Some\ True \vee C \cap neg\_lit\ `\ C \neq \{\}$
by (*simp add*: *is_blocked_alt*)
have *f3*: $\forall\, l\ L\ p.\ ((l::'a\ literal) \notin L \vee neg\_lit\ l \notin L) \vee sem\_clause\ L\ p$
by (*simp add*: *tautology*)
have $sem\_clause'\ C\ A = Some\ True \longrightarrow sem\_clause\ C\ \sigma$
using *a1* **by** (*simp add*: *compat_clause*)
then show *?thesis*
using *f3 f2* **by** *fastforce*
qed

2.2 Unit Propagation

lemma *unit_propagation*:
assumes *C*∈*F*
assumes *UNIT*: *is_unit_lit A C l*
shows *equiv' F A* (*assign_lit A l*)
unfolding *equiv'_def models'_def*
proof *safe*
from *UNIT* **have** *l*∈*C*
and *UNDEC*: $sem\_lit'\ l\ A = None$
and *OTHER_FALSE'*: $sem\_clause'\ (C-\{l\})\ A = Some\ False$
unfolding *is_unit_lit_def* **by** *auto*

{
fix σ
assume *COMPAT*: *compat_assignment A* σ
have *OTHER_FALSE*: $sem\_clause\ (C-\{l\})\ \sigma = False$
using *compat_clause*[*OF COMPAT OTHER_FALSE'*] **.**

assume *sem_cnf F* σ
with ‹*C*∈*F*› ‹*l*∈*C*› *OTHER_FALSE* **have** *sem_lit l* σ
unfolding *sem_cnf_def sem_clause_def* **by** *auto*

with *COMPAT* **show** *compat_assignment* (*assign_lit A l*) σ
unfolding *compat_assignment_def*
by (*cases l*) *auto*
}
{
fix σ
assume *compat_assignment* (*assign_lit A l*) σ

```
    with UNDEC show compat_assignment A σ
      unfolding compat_assignment_def
      apply (cases l; simp)
      apply (metis option.distinct(1))+
      done
  }
qed


inductive-set prop_unit_R :: 'a cnf ⇒ (('a⇀bool) × ('a⇀bool)) set for F
  where
  step: ⟦ C∈F; is_unit_lit A C l ⟧ ⟹ (A,assign_lit A l)∈prop_unit_R F

lemma prop_unit_R_Domain[simp]:
  A ∈ Domain (prop_unit_R F) ⟷ (∃ C∈F. is_unit_clause A C)
  by (auto
      elim!: prop_unit_R.cases
      simp: is_unit_clause_def
      dest: prop_unit_R.intros)

lemma prop_unit_R_equiv:
  assumes (A,A')∈(prop_unit_R F)*
  shows equiv' F A A'
  using assms
  apply induction
  apply simp
  apply (erule prop_unit_R.cases)
  using equiv'_trans unit_propagation by blast


lemma wf_prop_unit_R: finite F ⟹ wf ((prop_unit_R F)⁻¹)
  apply (rule wf_subset[OF
          wf_measure[where f=λA. card { C∈F. sem_clause' C A = None }]])
  apply safe
  apply (erule prop_unit_R.cases)
  apply simp
  apply (rule psubset_card_mono)
  subgoal by auto []
  apply safe
  subgoal
    apply (auto simp: is_unit_lit_def)
    apply (metis assign_undec_pres_dec_clause boolopt_cases_aux.cases)
    done
  subgoal for _ _ C A l
    proof −
      assume a1: C ∈ F
      assume a2: is_unit_lit A C l
      assume a3: {C ∈ F. sem_clause' C (assign_lit A l) = None}
                 = {C ∈ F. sem_clause' C A = None}
      have sem_clause' C A = None
        using a2 by (metis unit_clause_sem')
      then show ?thesis
        using a3 a2 a1 unit_clause_assign_dec by force
    qed
  done
```

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∈C
```

```
  assumes NFALSE: sem_lit' l A ≠ Some False
  assumes CANDS: ∀ D∈F. neg_lit l ∈ D
            ⟶ implied_clause F A (C ∪ (D − {neg_lit l}))
  shows redundant_clause F A C
proof (cases is_blocked A C)
  case True thus ?thesis using blocked_redundant by blast
next
  case NBLOCKED: False
  show ?thesis
  proof
    fix σ
    assume COMPAT: compat_assignment A σ and MODELS: sem_cnf F σ
    show ∃σ'. compat_assignment A σ' ∧ sem_clause C σ' ∧ sem_cnf F σ'
    proof (cases sem_clause C σ)
      case True with COMPAT MODELS show ?thesis by blast
    next
      case False

      let ?σ' = σ(var_of_lit l := is_pos l)
      from NFALSE COMPAT have compat_assignment A ?σ'
        by (cases l) (auto simp: compat_assignment_def)
      moreover from LIC have sem_clause C ?σ'
        unfolding sem_clause_def by (cases l; force)
      moreover {
        fix E assume E∈F neg_lit l ∉ E
        with MODELS have sem_clause E ?σ'
          unfolding sem_cnf_def sem_clause_def
          apply (cases l; clarsimp)
          apply (metis sem_lit.simps(1) syn_indep_lit
                       upd_sigma_true var_of_lit.elims)
          by (metis sem_lit.simps(2) syn_indep_lit
                    upd_sigma_false var_of_lit.elims)
      } moreover {
        fix D assume D∈F neg_lit l ∈ D
        with CANDS have implied_clause F A (C ∪ (D − {neg_lit l})) by blast
        with MODELS COMPAT have sem_clause (C ∪ (D − {neg_lit l})) σ
          by (metis (no_types, lifting) implied_clause_def
                    mem_Collect_eq models'_def sem_cnf_insert)
        with False have sem_clause (D − {neg_lit l}) σ
          by (auto simp: sem_clause_def)
        hence sem_clause D ?σ' by (simp add: sem_clause_set)
      } ultimately show ?thesis unfolding sem_cnf_def by blast
    qed
  qed
qed

lemma abs_rat_criterion':
  assumes RAT: ∃ l∈C.
    sem_lit' l A ≠ Some False
  ∧ (∀ D∈F. neg_lit l ∈ D ⟶ implied_clause F A (C ∪ (D−{neg_lit l})))
  shows redundant_clause F A C
  using assms abs_rat_criterion by blast
```

Assign all literals of clause to false.

```
definition and_not_C A C ≡ λv.
  if Pos v ∈ C then Some False else if Neg v ∈ C then Some True else A v

lemma compat_and_not_C:
  assumes compat_assignment A σ
  assumes ¬sem_clause C σ
  shows compat_assignment (and_not_C A C) σ
  by (smt SAT_Basic.sem_neg_lit and_not_C_def assms(1) assms(2)
      compat_assignment_def neg_lit.simps(2) option.inject
```

```
    sem_clause_def sem_lit.simps(2))

lemma and_not_empty[simp]: and_not_C A {} = A
  unfolding and_not_C_def by auto

lemma and_not_insert_None: sem_lit' l (and_not_C A C) = None
  ⟹ and_not_C A (insert l C) = assign_lit (and_not_C A C) (neg_lit l)
  apply (cases l)
  apply (auto simp: and_not_C_def split: if_split_asm)
  done

lemma and_not_insert_False: sem_lit' l (and_not_C A C) = Some False
  ⟹ and_not_C A (insert l C) = and_not_C A C
  apply (cases l)
  apply (auto simp: and_not_C_def split: if_split_asm)
  done

lemma sem_lit_and_not_C_conv: sem_lit' l (and_not_C A C) = Some v ⟷ (
    (l∉C ∧ neg_lit l∉C ∧ sem_lit' l A = Some v)
  ∨ (l∈C ∧ neg_lit l∉C ∧ v=False)
  ∨ (l∉C ∧ neg_lit l∈C ∧ v=True)
  ∨ (l∈C ∧ neg_lit l∈C ∧ v=(¬is_pos l))
  )
  by (cases l) (auto simp: and_not_C_def)

lemma sem_lit_and_not_C_None_conv: sem_lit' l (and_not_C A C) = None ⟷
  sem_lit' l A = None ∧ l∉C ∧ neg_lit l∉C
  by (cases l) (auto simp: and_not_C_def)
```

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: ¬is_blocked A C ⟹
    ∃ A₁. equiv' F (and_not_C A C) A₁ ∧ (∃ E∈F. is_conflict_clause A₁ E)
  shows implied_clause F A C
proof
  fix σ
  assume COMPAT: compat_assignment A σ
  assume MODELS: sem_cnf F σ

  show sem_clause (C) σ
  proof (cases is_blocked A C)
    case True
    thus ?thesis using blocked_clause_true COMPAT by auto
  next
    case False
    from RC[OF False] obtain A₁ E where
        EQ: equiv' F (and_not_C A C) A₁
      and CONFL: E ∈ F sem_clause' E A₁ = Some False
      by auto
    show ?thesis
    proof (rule ccontr)
      assume ¬sem_clause C σ
      with compat_and_not_C[OF COMPAT]
      have compat_assignment (and_not_C A C) σ by auto
      with EQ have COMPAT1: compat_assignment A₁ σ
        by (metis (mono_tags, lifting) MODELS equiv'_def
              mem_Collect_eq models'_def)
      with MODELS CONFL show False using compat_clause sem_cnf_def by blast
    qed
  qed
qed
```

The unit-propagation steps of (¬ *is_blocked ?A ?C* ⟹ ∃ A_1. *equiv' ?F* (*and_not_C ?A ?C*) A_1 ∧ (∃ *E*∈*?F*.

sem_clause′ E A₁ = Some False)) ⟹ *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 ¬is_blocked A C
    ⟹ ∃ A₁. equiv′ F (and_not_C A C) A₁ ∧ (¬is_blocked A₁ D
    ⟶ (∃ A₂. equiv′ F (and_not_C A₁ D) A₂ ∧ (∃ E∈F. is_conflict_clause A₂ E)))

  shows implied_clause F A (C∪D)
proof
  fix σ
  assume COMPAT: compat_assignment A σ
  assume MODELS: sem_cnf F σ

  show sem_clause (C∪D) σ
  proof (cases is_blocked A C)
    case True
    thus ?thesis using blocked_clause_true COMPAT by auto
  next
    case False
    from assms[OF False] obtain A₁ where
        EQ1: equiv′ F (and_not_C A C) A₁
      and RC2: (¬is_blocked A₁ D
        ⟶ (∃ A₂. equiv′ F (and_not_C A₁ D) A₂
            ∧ (∃ E∈F. is_conflict_clause A₂ E)))
      by auto

    show ?thesis
    proof (rule ccontr; clarsimp)
      assume ¬sem_clause C σ ¬sem_clause D σ
      with compat_and_not_C[OF COMPAT]
      have compat_assignment (and_not_C A C) σ by auto
      with EQ1 have COMPAT1: compat_assignment A₁ σ
        by (metis (mono_tags, lifting) MODELS equiv′_def
              mem_Collect_eq models′_def)
      from compat_and_not_C[OF COMPAT1] ‹¬ sem_clause D σ› have
        1: compat_assignment (and_not_C A₁ D) σ by auto
      have ¬is_blocked A₁ D
        using COMPAT1 ‹¬ sem_clause D σ› blocked_clause_true by auto
      with RC2 obtain A₂ E where
          EQ2: equiv′ F (and_not_C A₁ D) A₂
        and CONFL: E∈F is_conflict_clause A₂ E
        by auto
      from EQ2 1 have COMPAT2: compat_assignment A₂ σ
        by (metis (mono_tags, lifting) MODELS equiv′_def
              mem_Collect_eq models′_def)
      with MODELS CONFL show False using compat_clause sem_cnf_def by blast
    qed
  qed
qed
```

2.4 Old *assign_all_negated* Formulation

```
definition assign_all_negated A C ≡ let UD = {l∈C. sem_lit′ l A = None} in
  A ++ (λl.      if Pos l∈UD then Some False
          else if Neg l ∈ UD then Some True
            else None)

lemma abs_rup_criterion:
  assumes models′ F (assign_all_negated A C) = {}
  shows implied_clause F A C
  using assms
  unfolding models′_def implied_clause_def
```

apply (*safe*; *simp*)
proof (*rule ccontr*)
fix σ
assume *COMPAT*: *compat_assignment A* σ
assume *S*: *sem_cnf F* σ
assume *CD*: $\forall\sigma$. *compat_assignment* (*assign_all_negated A C*) σ
$\longrightarrow \neg$ *sem_cnf F* σ
assume *NS*: $\neg$ *sem_clause C* σ

from *NS* **have** $\forall\, l{\in}C$. *sem_lit l* σ = *False* **by** (*auto simp*: *sem_clause_def*)

with *COMPAT* **have** *compat_assignment* (*assign_all_negated A C*) σ
by (*clarsimp simp*: *compat_assignment_def assign_all_negated_def*
split: *if_split_asm*) *auto*
with *S CD* **show** *False* **by** *blast*
qed

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*
| *sem_lit′ l A* = *None neg_lit l* ∈ *C v*=*True*
| *sem_lit′ l A* = *None l* ∈ *C v*=*False*
using *assms* **unfolding** *assign_all_negated_def*
apply (*cases l*)
apply (*auto simp*: *map_add_def split*: *if_split_asm*)
done

lemma *sem_lit_assign_all_negated_none_iff*:
sem_lit′ l (*assign_all_negated A C*) = *None*
$\longleftrightarrow$ (*sem_lit′ l A* = *None* ∧ *l*∉*C* ∧ *neg_lit l* ∉ *C*)
unfolding *assign_all_negated_def*
apply (*cases l*)
apply (*auto simp*: *map_add_def split*: *if_split_asm*)
done

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*
using *assms* **unfolding** *assign_all_negated_def*
apply (*cases l*)
apply (*fastforce simp*: *map_add_def split*: *if_split_asm*)+
done

lemma *sem_lit_assign_all_negated_assign*:
assumes $\forall\, l{\in}C$. *neg_lit l*∉*C l* ∈ *C sem_lit′ l A* = *None*
shows *sem_lit′ l* (*assign_all_negated A C*) = *Some False*
using *assms* **unfolding** *assign_all_negated_def*
apply (*cases l*)
apply (*auto simp*: *map_add_def split*: *if_split_asm*)
done

lemma *sem_lit_assign_all_negated_neqv*:
sem_lit′ l (*assign_all_negated A C*) ≠ *Some v* ⟹ *sem_lit′ l A* ≠ *Some v*
by (*auto simp*: *sem_lit_assign_all_negated_pres_decided*)

lemma *aan_idem*[*simp*]:
assign_all_negated (*assign_all_negated A C*) *C* = *assign_all_negated A C*
by (*auto intro*!: *ext simp*: *assign_all_negated_def map_add_def*)

lemma *aan_dbl*:
assumes $\forall\, l{\in}C{\cup}C'$. *neg_lit l* ∉ *C*∪*C′*
shows *assign_all_negated* (*assign_all_negated A C*) *C′*

```
      = assign_all_negated A (C∪C')
  using assms by (force intro!: ext simp: assign_all_negated_def map_add_def)

 lemma aan_mono2:
  ⟦C⊆C'; ∀ l∈C'. neg_lit l ∉ C'⟧
   ⟹ assign_all_negated A C ⊆m assign_all_negated A C'
  by (auto simp: assign_all_negated_def map_add_def map_le_def)


 lemma aan_empty[simp]: assign_all_negated A {} = A
  by (auto simp: assign_all_negated_def)

 lemma aan_restrict:
  assign_all_negated A C |` (- var_of_lit ` {l ∈ C. sem_lit' l A = None}) = A
  apply (rule ext)
  unfolding assign_all_negated_def
  apply (clarsimp simp: map_add_def restrict_map_def; safe)
  apply simp_all
  apply force
  apply force
  subgoal for l by (cases l) auto
  subgoal for l v by (cases l) auto
  subgoal for v l by (cases l) auto
  subgoal for v l by (cases l) auto
  done

 lemma aan_insert:
  assumes ∀ l'∈C. sem_lit' l' A ≠ Some True ∧ neg_lit l' ∉ C
  assumes sem_lit' l A ≠ Some True ∧ neg_lit l ∉ C
  shows assign_lit (assign_all_negated A C) (neg_lit l)
      = assign_all_negated A (insert l C)
  apply (rule ext)
  using assms
  apply (cases l)
  apply (auto simp: assign_all_negated_def map_add_def)
  done

 lemma aan_insert_set:
  assumes sem_lit' l A ≠ None
  shows assign_all_negated A (insert l C) = assign_all_negated A C
  apply (rule ext)
  using assms
  apply (cases l)
  apply (auto simp: assign_all_negated_def map_add_def)
  done

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*
by (*auto simp*: *rdomp_def list_set_assn_def elim*!: *finite_set_rel_transfer*)

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*)
apply (*clarsimp simp*: *is_pure_conv list_set_assn_def*
list_assn_pure_conv IS_PURE_def list_set_rel_compp)
apply (*rule sepref_gen_algo_rules*)
done

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*
unfolding *it_invar_def* **by** *auto*
lemma *it_invar_imp_ran*[*simp*, *intro*]: *it_invar it* ⟹ *itran it it_end*
unfolding *it_invar_def* **by** *auto*
lemma *itran_invarD*: *itran it it_end* ⟹ *it_invar it*
unfolding *it_invar_def* **by** *auto*
lemma *itran_invarI*: ⟦*itran it it'*; *it_invar it'*⟧ ⟹ *it_invar it*
unfolding *it_invar_def* **by** (*blast intro*: *itran_trans*)

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*
apply (*rule wf_subset*[*of measure* (λ*it*. *length* (*the_seg it it_end*))])
unfolding *it_invar_def WF_def*
by (*auto*)

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*)
unfolding *it_invar_def* **by** *auto*

lemma *it_next_wf*[*simp*, *intro*]:
⟦ *it_invar it*; *it* ≠ *it_end* ⟧ ⟹ (*it_next it*, *it*) ∈ *WF*
unfolding *WF_def* **by** *auto*

lemma *seg_wf*[*simp*, *intro*]: ⟦*seg it l it'*; *it_invar it'*⟧ ⟹ (*it'*,*it*)∈*WF**
apply (*induction l arbitrary*: *it*)
apply *auto*
by (*metis it_invar_def it_next_wf itran_antisym itran_def itran_next*
itran_trans rtrancl.intros(*1*) *rtrancl.intros*(*2*))

lemma *lz_string_wf*[*simp*, *intro*]:
⟦*lz_string 0 it l ita*; *it_invar ita*⟧ ⟹ (*ita*, *it*) ∈ *WF*$^+$
unfolding *lz_string_def*
apply *auto*
by (*metis input_pre.it_invar_def input_pre_axioms it_next_wf itran_def*
itran_next rtrancl_into_trancl2 seg_invar2 seg_no_cyc seg_wf)

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*)
unfolding *check_not_end_def* **by** (*refine_vcg*; *auto*)

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 it₀;
  CHECK (x≥0) (mk_errIit STR ''Invalid nat'' x it₀);
  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⁺)
  unfolding parse_literal_def
  by refine_vcg auto

lemma skip_spec[THEN ESPEC_trans,refine_vcg]:
  ⟦it_invar it⟧
  ⟹ skip it ≤ ESPEC (λ_. True) (λit'. it_invar it' ∧ (it',it)∈WF⁺)
  unfolding skip_def
  by refine_vcg auto

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⁺)
  unfolding parse_int_def
  by refine_vcg auto

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⁺)
  unfolding parse_nat_def
  by refine_vcg auto
```

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 (=)
  apply standard
  subgoal by (auto simp: lit_rel_def in_br_conv lit_invar_def)
  subgoal
    apply sepref_to_hoare
    apply (sep_auto simp: lit_rel_def lit_α_def in_br_conv)
    done
  applyS sep_auto
  done

lemma neg_lit_refine[sepref_import_param]:
  (uminus, neg_lit) ∈ lit_rel → lit_rel
  by (auto simp: lit_rel_def in_br_conv lit_α_def lit_invar_def)

lemma lit_α_refine[sepref_import_param]:
  (λx. x, lit_α) ∈ [λx. x≠0]ₓ int_rel → lit_rel
  by (auto simp: lit_rel_def lit_invar_def in_br_conv intro!: frefI)
```

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* .

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* .

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* .

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* .

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*
by *auto*

lemma *lit_eq_impl*[*sepref_import_param*]:
((=),(=)) ∈ *lit_rel* → *lit_rel* → *bool_rel*
by (*auto*
simp: *lit_rel_def in_br_conv lit_α_def lit_invar_def*
split: *if_split_asm*)

lemma *var_of_lit_refine*[*sepref_import_param*]:
(*nat o abs*,*var_of_lit*) ∈ *lit_rel* → *nat_rel*
by (*auto simp*: *lit_rel_def lit_α_def in_br_conv*)

lemma *is_pos_refine*[*sepref_import_param*]:
(λ*x*. *x*>*0*, *is_pos*) ∈ *lit_rel* → *bool_rel*
by (*auto*
simp: *lit_rel_def lit_α_def in_br_conv lit_invar_def*
split: *if_split_asm*)

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*)
)
apply (*cases l*)
by (*auto split*: *option.split simp*: *Let_def*)

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)
  )
  apply (cases l)
  by (auto split: option.split simp: Let_def)


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
  by simp_all

lemma vv_bool_eq_refine[sepref_import_param]:
  (vv_eq_bool, (=)) ∈ vv_rel → bool_rel → bool_rel
  by (auto simp: vv_rel_def)

sepref-definition op_lit_is_true_impl is uncurry (RETURN oo op_lit_is_true)
  :: (pure lit_rel)^k *_a assignment_assn^k →_a bool_assn
  unfolding op_lit_is_true_alt assignment_assn_def
  supply option.splits[split]
  by sepref

sepref-definition op_lit_is_false_impl is uncurry (RETURN oo op_lit_is_false)
  :: (pure lit_rel)^k *_a assignment_assn^k →_a bool_assn
  unfolding op_lit_is_false_alt assignment_assn_def
  supply option.splits[split]
  by sepref

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
  by (auto simp: vv_rel_def b2vv_conv_impl_def split: if_split_asm)

lemma vv_unused0[safe_constraint_rules]: (is_unused_elem 0) (pure vv_rel)
  by (auto simp: vv_rel_def)


sepref-definition assign_lit_impl
  is uncurry (RETURN oo assign_lit)
    :: assignment_assn^d *_a (pure lit_rel)^k →_a assignment_assn
  unfolding assign_lit_def assignment_assn_def
  apply (rewrite at is_pos _ b2vv_conv_def[symmetric])
  by sepref

term op_unset_lit
sepref-definition unset_lit_impl
  is uncurry (RETURN oo op_unset_lit)
    :: assignment_assn^d *_a (pure lit_rel)^k →_a assignment_assn
  unfolding op_unset_lit_def assignment_assn_def
  by sepref

sepref-definition unset_var_impl
  is uncurry (RETURN oo op_map_delete)
    :: (pure nat_rel)^k *_a assignment_assn^d →_a assignment_assn
  unfolding assignment_assn_def
  by sepref

sepref-definition assignment_empty_impl is uncurry0 (RETURN op_map_empty)
  :: unit_assn^k →_a assignment_assn
  unfolding assignment_assn_def
```

apply (*rewrite amd.fold_custom_empty*)
by *sepref*

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

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

definition [*simp*]: *op_assignment_empty* ≡ *op_map_empty*
interpretation *assignment*: *map_custom_empty op_assignment_empty*
by *unfold_locales simp*
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* .

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

abbreviation *error_assn*
≡ *id_assn* ×$_a$ *option_assn int_assn* ×$_a$ *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*
by *unfold_locales*

sublocale *input liti.I liti.next liti.peek liti.end*
apply *unfold_locales*
unfolding *it_invar_def liti.itran_alt*
apply (*auto simp*: *ait_begin_def ait_end_def*)
done

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 ≡ ∃$_A$*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*>
unfolding *creg_empty_def* **by** (*sep_auto simp*: *is_creg_def*)

definition [*simp*]: *op_creg_empty* ≡ *op_map_empty* :: *nat literal* ⇀ *nat list*
interpretation *creg*: *map_custom_empty op_creg_empty* **by** *unfold_locales simp*
lemma *creg_empty_hnr*[*sepref_fr_rules*]:
(*uncurry0 creg_empty*, *uncurry0* (*RETURN op_creg_empty*))
∈ *unit_assn*^{k} →$_a$ *is_creg*
apply *sepref_to_hoare*
apply *sep_auto*
done

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*>$_t$
unfolding *creg_initialize_def is_creg_def*
by (*sep_auto intro*!: *ext simp*: *lit_rel_def in_br_conv int_encode_eq*)

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*)>$_t$
unfolding *creg_register_def is_creg_def abs_cr_register_def*
by (*sep_auto intro*!: *ext simp*: *lit_rel_def in_br_conv int_encode_eq*)

lemma *creg_register_hnr*[*sepref_fr_rules*]:
(*uncurry2 creg_register*, *uncurry2* (*RETURN ooo abs_cr_register*))
∈ (*pure lit_rel*)^k ∗$_a$ *nat_assn*^{k} ∗$_a$ *is_creg*^{d} →$_a$ *is_creg*

unfolding *list_assn_pure_conv option_assn_pure_conv*
apply *sepref_to_hoare*
apply *sep_auto*
done

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*))
∈ (*pure lit_rel*)^k *∗$_a$ is_creg*^{d} →$_a$ *is_creg*
apply *sepref_to_hoare*
apply *sep_auto*
done

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*)>
unfolding *is_creg_def op_creg_lookup_def*
by (*sep_auto intro*!: *ext simp*: *lit_rel_def in_br_conv*)

lemma *creg_lookup_hnr*[*sepref_fr_rules*]:
(*uncurry op_creg_lookup*, *uncurry* (*RETURN oo op_map_lookup*))
∈ (*pure lit_rel*)^k *∗$_a$ is_creg*^{k} →$_a$ *option_assn* (*list_assn id_assn*)
unfolding *list_assn_pure_conv option_assn_pure_conv*
apply *sepref_to_hoare*
apply *sep_auto*
done

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* ×$_a$ *cm_assn* ×$_a$ *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*))
  ∈ *liti.a_assn*^{k} ∗$_a$ *liti.it_assn*^{k} →$_a$ *dflt_option_assn 0 liti.it_assn*
  unfolding *liti.it_assn_def liti.a_assn_def dflt_option_assn_def*
  apply (*simp add*: *b_assn_pure_conv*)
  apply (*sepref_to_hoare*)
  unfolding *item_next_impl_def*
  by (*sep_auto simp*: *liti.I_def item_next_def dflt_option_rel_aux_def*)

lemma *at_item_end_hnr*[*sepref_fr_rules*]:
  (*uncurry* (*return oo at_item_end*), *uncurry* (*RETURN oo at_item_end*))
  ∈ *nat_assn*^{k} ∗$_a$ *liti.it_assn*^{k} →$_a$ *bool_assn*
  unfolding *liti.it_assn_def liti.a_assn_def dflt_option_assn_def*
  apply (*simp add*: *b_assn_pure_conv*)
  apply (*sepref_to_hoare*)
  apply *sep_auto*
  done

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*))
unfolding *verify_sat_spec_def clause_DB_valid_def clause_DB_sat_def Let_def*
by *auto*

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*))
unfolding *verify_unsat_spec_def clause_DB_valid_def clause_DB_sat_def Let_def*
by *auto*

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*)
apply (*rule eq_reflection*)
unfolding *clause_DB_valid_def F_invar_def*
by *auto*

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*)
using *clause_DB_sat_def*
unfolding *direct_sat_iff_sat*[*symmetric*] *direct_sat_def parse_direct_def*
by *auto*

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*)))
unfolding *verify_sat_spec_def clause_DB_sat_concise clause_DB_valid_concise*
by (*simp add*: *Let_def*)

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*)))
unfolding *verify_unsat_spec_def clause_DB_sat_concise clause_DB_valid_concise*
apply (*rule eq_reflection*)
apply (*cases F_end* = *1*)
apply (*auto simp add*: *Let_def tl_take*)
done

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* **by** (*auto simp*: *in_br_conv*)
sepref-decl-impl (*no_register*) *ndls_empty*: *ndls_empty_hnr_aux*[*sepref_param*] .

lemma *ndls_is_empty_hnr_aux*: ((=) [], *op_set_is_empty*) ∈ *ndls_rel* → *bool_rel*
by (*auto simp*: *in_br_conv*)
sepref-decl-impl *ndls_is_empty*: *ndls_is_empty_hnr_aux*[*sepref_param*] .

lemma *ndls_insert_hnr_aux*: ((#), *op_set_insert*) ∈ *Id* → *ndls_rel* → *ndls_rel*
by (*auto simp*: *in_br_conv*)

sepref-decl-impl *ndls_insert*: *ndls_insert_hnr_aux*[*sepref_param*] .

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

sepref-decl-impl *op_ndls_ls_copy_hnr_aux*[*sepref_param*] .
end

definition [*simp*]: *op_ndls_empty* = *op_set_empty*
interpretation *ndls*: *set_custom_empty return* [] *op_ndls_empty*
by *unfold_locales simp*
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* **by** *simp*

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 reason 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?//////RAT literals.////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.

~~subsection ‹Input Format Specification› text ‹The ‹unsat_input› locale is an iterator over integers, overlaid with an iteration scheme over items.› locale unsat_input = input it_invar' for it_invar'::'it::linorder ⇒ _ + fixes WFitem :: ('it × 'it) set and item_is_last :: 'it ⇒ bool and item_next :: 'it → 'it assumes wf_WFitem[simp, intro!]: wf WFitem and invar_next: ⟦it_invar it; ¬item_is_last it; item_next it = Some it'⟧ ⟹ it_invar it' and wf_next: ⟦it_invar it; ¬item_is_last it; item_next it = Some it'⟧ ⟹ (it',it)∈WFitem begin lemma wf_WFitem_trancl[simp, intro!]: wf (WFitem⁺) using wf_trancl[OF wf_WFitem] . end~~

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 unfolding prfWF_def by simp
 lemma wf_prfWFtrcl[simp, intro!]: wf (prfWF⁺)
  by (simp add: wf_trancl)

 lemma parse_prf_spec[THEN ESPEC_trans,refine_vcg]:
  parse_prf prf ≤ ESPEC (λ_. True) (λ(_,prf'). (prf',prf)∈prfWF⁺)
  unfolding parse_prf_def
  by refine_vcg (auto simp: prfWF_def)

 lemma parse_id_spec[THEN ESPEC_trans,refine_vcg]:
  parse_id prf
    ≤ ESPEC (λ_. True) (λ(x,prf'). (prf',prf)∈prfWF⁺ ∧ x>0)
  unfolding parse_id_def
  by refine_vcg auto

 lemma parse_idZ_spec[THEN ESPEC_trans,refine_vcg]:
  parse_idZ prf
    ≤ ESPEC (λ_. True) (λ(x,prf'). (prf',prf)∈prfWF⁺)
  unfolding parse_idZ_def
  by refine_vcg auto

 lemma parse_type_spec[THEN ESPEC_trans,refine_vcg]:
  parse_type prf
    ≤ ESPEC (λ_. True) (λ(x,prf'). (prf',prf)∈prfWF⁺)
  unfolding parse_type_def
  by refine_vcg auto


 lemma parse_prf_literal_spec[THEN ESPEC_trans,refine_vcg]:
  parse_prf_literal prf
    ≤ ESPEC (λ_. True) (λ(_,prf'). (prf',prf)∈prfWF⁺)
  unfolding parse_prf_literal_def
  by refine_vcg auto

 lemma parse_prf_literalZ_spec[THEN ESPEC_trans,refine_vcg]:
  parse_prf_literalZ prf
    ≤ ESPEC (λ_. True) (λ(_,prf'). (prf',prf)∈prfWF⁺)
  unfolding parse_prf_literalZ_def
  by refine_vcg auto

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* ≡ λ*i* (*CM*,*RL*). *ERETURN* (*CM*(*i*:=*None*),*RL*)

definition *remove_ids CMRL*$_0$ *prf* ≡ *doE* {
  (*i*,*prf*) ← *parse_idZ prf*;
  (*CMRL*,*i*,*prf*) ← *EWHILEIT*
    (λ(*CMRL*,*i*,*it*). *cm_invar CMRL*
              ∧ *cm_F CMRL* ⊆ *cm_F CMRL*$_0$
              ∧ *cm_ids CMRL* ⊆ *cm_ids CMRL*$_0$)
    (λ(_,*i*,_). *i*≠*0*)
    (λ(*CMRL*,*i*,*prf*). *doE* {
      CMRL ← *remove_id i CMRL*;
      (*i*,*prf*) ← *parse_idZ prf*;
      ERETURN (*CMRL*,*i*,*prf*)
    }) (*CMRL*$_0$,*i*,*prf*);
  ERETURN (*CMRL*,*prf*)
}

definition *add_clause*
  :: *id* ⇒ *var clause* ⇒ *clausemap* ⇒ (_, *clausemap*) *enres*
  where *add_clause* ≡ λ*i C* (*CM*,*RL*). *doE* {
    EASSERT (¬*is_syn_taut C*);
    EASSERT (*i*∉*cm_ids* (*CM*,*RL*));
    let CM = *CM*(*i* ↦ *C*);
    let RL = (λ*l*. *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)
  unfolding resolve_id_def
  apply refine_vcg
  apply (auto simp: cm_F_def intro: ranI)
  done

lemma remove_id_correct[THEN ESPEC_trans,refine_vcg]:
  cm_invar CMRL
  ⟹ remove_id i CMRL
     ≤ ESPEC
         (λ_. False)
         (λCMRL'. cm_invar CMRL'
               ∧ cm_F CMRL' ⊆ cm_F CMRL
               ∧ cm_ids CMRL' ⊆ cm_ids CMRL)
  unfolding remove_id_def
  apply (refine_vcg)
  apply (auto
         simp: cm_F_def ran_def restrict_map_def cm_invar_def cm_ids_def
         split: if_split_asm)
  apply fastforce
  done

//TODO: Move to Misc//
lemma rtrancl_inv_image_ss: (inv_image R f)* ⊆ inv_image (R*) f
proof (clarify)
  fix a b
  assume (a,b) ∈ (inv_image R f)*
  thus (a,b) ∈ inv_image (R*) f
    by induction auto
qed

lemmas rtrancl_inv_image_ssI = rtrancl_inv_image_ss[THEN set_mp]

lemma remove_ids_correct[THEN ESPEC_trans,refine_vcg]:
  ⟦cm_invar CMRL⟧
  ⟹ remove_ids CMRL prf
     ≤ ESPEC
         (λ_. True)
         (λ(CMRL',prf'). cm_invar CMRL'
               ∧ cm_F CMRL' ⊆ cm_F CMRL
               ∧ cm_ids CMRL' ⊆ cm_ids CMRL
               ∧ (prf',prf)∈prfWF+
         )
  unfolding remove_ids_def
  apply (refine_vcg EWHILEIT_rule[where
         R=inv_image (prfWF+) (λ(_,_,prf). prf)
         ])
  by (auto dest: rtrancl_inv_image_ssI)


lemma add_clause_correct[THEN ESPEC_trans,refine_vcg]:
  ⟦cm_invar CM; i∉cm_ids CM; ¬is_syn_taut C⟧ ⟹
    add_clause i C CM ≤ ESPEC (λ_. False) (λCM'.
       cm_F CM' = insert C (cm_F CM)
     ∧ cm_invar CM'
     ∧ cm_ids CM' = insert i (cm_ids CM)
    )
  unfolding add_clause_def
  apply (refine_vcg)
  apply (vc_solve
         simp: cm_ids_def cm_F_def ran_def restrict_map_def cm_invar_def
         split: option.split
         solve: asm_rl)
```

```
  subgoal by fastforce
  ~~subgoal by (auto: metis (no_types, hide_lams) insertCI                              not_Some_eq option.inject)~~
  done

definition rat_candidates CM A reslit
  ≡ {i. ∃C. CM i = Some C
          ∧ neg_lit reslit ∈ C
          ∧ ¬is_blocked A (C−{neg_lit reslit})}

lemma is_syn_taut_mono_aux: is_syn_taut (C−X) ⟹ is_syn_taut C
  by (auto simp: is_syn_taut_def)

lemma get_rat_candidates_correct[THEN ESPEC_trans,refine_vcg]:
  ⟦ cm_invar CM ⟧
  ⟹ get_rat_candidates CM A reslit
      ≤ ESPEC (λ_. True) (λr. r=rat_candidates (fst CM) A reslit)
  unfolding get_rat_candidates_def
  apply refine_vcg
  unfolding cm_invar_def rat_candidates_def is_blocked_def
  apply (auto dest!: is_syn_taut_mono_aux simp: ranI)
  apply force
  done


definition check_unit_clause A C
  ≡ ESPEC (λ_. ¬is_unit_clause A C) (λl. is_unit_lit A C l)

definition apply_unit i CM A ≡ doE {
  C ← resolve_id CM i;
  l ← 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')
  unfolding apply_unit_def check_unit_clause_def
  apply (refine_vcg)
  apply (auto simp: unit_propagation)
  apply (auto simp: is_unit_lit_def)
  done

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⁺)
  unfolding apply_units_def
  apply (refine_vcg
```

```
    EWHILET_rule[where
        I=λ(A',_,_). equiv' (cm_F CM) A A'
      and R=inv_image (prfWF⁺) (λ(_,_,prf). prf)
    ]
    )
  apply (auto dest: equiv'_trans rtrancl_inv_image_ssI)
  done
```

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))}

//////////////////////////////////////////////////////////////////////////////

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⁺)
  unfolding parse_skip_listZ_def
  apply (refine_vcg EWHILET_rule[where R=inv_image (prfWF⁺) snd and I=λ_. True])
  apply (auto dest: rtrancl_inv_image_ssI)
  done
```

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⁺)
  supply check_correct[THEN ESPEC_trans, refine_vcg]
  unfolding check_candidates_def
  apply (refine_vcg
      EWHILET_rule[where
        I=λ(cleft,cand,prf).
          cleft ⊆ candidates
        ∧ (∀c∈candidates − cleft. Φ c)
      and R=inv_image (prfWF⁺) (λ(_,_,prf). prf)
      ])
  by (auto dest: rtrancl_inv_image_ssI)

~~id id* 0 id, advance iterator by one clause~~
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⁺
  )
  unfolding check_rup_proof_def parse_check_blocked_def
  apply refine_vcg
  using assms
  by (vc_solve
      simp:
      split!: if_split_asm
      intro: implied_is_redundant one_step_implied syn_taut_imp_blocked
      solve: asm_rl)


~~lit id lit* 0 id* 0 (id id* 0 id)* 0~~
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 (rule abs_rat_criterion[OF LIC NFALSE]; safe)
  fix D
  assume A: D∈cm_F (CM,RL) neg_lit reslit ∈ D

  show implied_clause (cm_F (CM, RL)) A (C ∪ (D − {neg_lit reslit}))
  proof (cases is_blocked A' (D − {neg_lit reslit}))
    case False
    with A obtain cand
      where D=the (CM cand) and cand∈rat_candidates CM A' reslit
      by (force simp: rat_candidates_def cm_F_def ran_def)
    thus ?thesis
      using CANDS by auto
  next
    case True
    thus ?thesis
      apply (rule_tac two_step_implied)
      using EQ1 by auto
  qed
qed


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⁺
)
 unfolding check_rat_proof_def parse_check_blocked_def
 apply refine_vcg
 subgoal using assms by auto
 subgoal using assms by auto
 using assms
 apply (cases CM)
 apply (elim conjE exE; simp; elim conjE)
 apply hypsubst apply simp
 subgoal premises prems for reslit prf1 i prf2 it' A' prf3 CM RL l
 proof −
  from prems have A:
       reslit ∈ clause_α l
    and CMI: cm_invar (CM, RL)
    and RESLIT_SEM: sem_lit' (reslit) A ≠ Some False
    and INID: i∉cm_ids (CM, RL)
    and NBLK: ¬ is_blocked A (clause_α l)
    and EQ1: equiv' (cm_F (CM, RL)) (and_not_C A (clause_α l)) A'
    and [simp]: it_invar it'
    and PRF: (prf1, prf) ∈ prfWF⁺ (prf2, prf1) ∈ prfWF⁺ (prf3, prf2) ∈ prfWF⁺
   by − assumption+

  from A have ARIC: reslit ∈ clause_α l by auto

  show ?thesis
   apply (refine_vcg check_candidates_rule[where
        Φ=λi. implied_clause
              (cm_F (CM,RL))
              A
              (clause_α l ∪ (the (CM i) − {neg_lit reslit}))])
   apply vc_solve
   applyS (auto simp: rat_candidates_def)
   subgoal
    thm two_step_implied
    apply (rule two_step_implied)
    apply (rule exI[where x=A'])
    using EQ1 apply auto
    done
   applyS auto []
   subgoal
    apply (rule rat_criterion[OF ARIC RESLIT_SEM EQ1])
    apply auto
    done
   applyS (rule CMI)
   subgoal using INID by simp
   subgoal using NBLK by (auto intro: syn_taut_imp_blocked)
   subgoal using PRF by auto
   done
 qed
 done


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
  )
  using assms(2,3)
  apply clarsimp
  unfolding check_item_def
  apply refine_vcg
  apply (split item_type.split; intro allI impI conjI)
  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)
  done


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)
  )
  using check_item_correct_pre[of s _ _ it prf] assms
  apply (cases s) by auto
```

definition *cm_empty* :: *clausemap* **where** *cm_empty* ≡ (*Map.empty*, *Map.empty*)
lemma *cm_empty_invar*[*simp*]: *cm_invar cm_empty*
by (*auto simp*: *cm_empty_def cm_invar_def*)
lemma *cm_F_empty*[*simp*]: *cm_F cm_empty* = {}
by (*auto simp*: *cm_empty_def cm_F_def*)
lemma *cm_ids_empty*[*simp*]: *cm_ids cm_empty* = {}
by (*auto simp*: *cm_empty_def cm_ids_def*)

lemma *cm_ids_empty_imp_F_empty*: *cm_ids CM* = {} ⟹ *cm_F CM* = {}
unfolding *cm_F_def cm_ids_def* **by** (*auto simp*: *ran_def*)

////cancel//TODO: Can we remove that?//////definition read_cnf :: "var clause list ⇒ clausemap ⇒ (_, clausemap × nat) enres" where "read_cnf L CM = doE {////(CM,next_id) ← enfold (λ_. True) (λC (CM,i). doE {////if (is_syn_taut C) then//////ERETURN (CM,i+1)//////else doE {//////CM ← add_clause i C CM;//////ERETURN (CM,i+1)////}////}) (CM,1);////ERETURN (CM,next_id-1)}"lemma read_cnf_correct[THEN ESPEC_trans, refine_vcg]:////[| cm_invar CM; cm_ids CM ≠ {} |]////⟹ read_cnf F CM ≤ ESPEC (λ_. True) (λ(CM,last_id). cm_invar CM ∧ sat (cm_F CM) = sat (set F) ∧ (∀i∈cm_ids CM. i≤last_id))"unfolding read_cnf_def apply (refine_vcg enfold_rule[where I=λ1 _ (CM,next_id). cm_invar CM ∧ SAT_Basic_models (cm_F CM) = SAT_Basic_models (set 1) ∧ (∀i∈cm_ids CM. i≤next_id))apply (auto simp: SAT_Basic_models_def sat_def cm_ids_empty_imp_F_empty)done

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*) (λ*x* (*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*) (λ*x 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*
apply (*clarsimp simp*: *is_syn_taut_conv*)
by (*metis map_option_eq_Some option.inject sem_neg_lit'*)

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*))))
unfolding *read_clause_check_taut_def*
apply (*refine_vcg*
parse_lz_rule[**where**
Φ=λ*lst* (*t*,*A*). *dom A* ⊆ *var_of_lit*`*clause_α lst*
∧ (*t* ⟶ *is_syn_taut* (*clause_α lst*))
∧ (¬*t* ⟶ (∀ *l*. *sem_lit' l A* = *Some True* ⟷ *l* ∈ *clause_α lst*))

```
        ]
      iterate_lz_rule[where Φ=λ_ l2 A. dom A ⊆ var_of_lit‘clause_α l2]
  )
  apply (vc_solve simp: not_Some_bool_if itran_invarI)
  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)
  done

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))))
  )
  unfolding read_cnf_new_def
  apply (refine_vcg tok_fold_rule[where
        Φ=λlst (CM,next_id,A).
            A = Map.empty
          ∧ cm_invar CM
          ∧ SAT_Basic.models (cm_F CM)
            = SAT_Basic.models (set (map clause_α lst))
          ∧ (∀ i∈cm_ids CM. i<next_id)
    and Z=litZ and l=lst
    ])
  apply (vc_solve)
  apply ((drule (1) lz_string_determ)?;
      fastforce
          simp: SAT_Basic.models_def sat_def
          simp: cm_ids_empty_imp_F_empty itran_invarI)+
  done

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' = {})
  unfolding cm_init_lit_def
  apply refine_vcg
  apply (auto simp: cm_invar_def cm_ids_def ran_def)
  done

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⁺)
  unfolding init_rat_counts_def
  apply (refine_vcg EWHILET_rule[where
      I=λ(CM,_,_). cm_invar CM
                 ∧ cm_ids CM = {}
    and R=inv_image (prfWF⁺) (λ(_,_,prf). prf)
    ])
  by (auto dest: rtrancl_inv_image_ssI)



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))
  unfolding verify_unsat_def
  apply (refine_vcg
    EWHILEIT_expinv_rule[where
        I=λ
          (None) ⇒ ¬sat (F_α lst)
        | (Some ((CM,A), it', prf')) ⇒ it_invar it'
            ∧ cm_invar CM
            ∧ (sat (F_α lst) ⟶ sat' (cm_F CM) A)
      and R=inv_image (less_than <*lex*> prfWF⁺) (λNone ⇒ (0::nat,undefined) | Some (_,_,prf) ⇒ (1,prf))
      ]
    )
  apply vc_solve
  apply assumption
  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)
  done


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
  unfolding backtrack_def by auto

definition is_backtrack A' T' A ≡ T'⊆dom A' ∧ A = backtrack A' T'
lemma is_backtrack_empty[simp]: is_backtrack A {} A
  unfolding is_backtrack_def by auto
lemma is_backtrack_not_undec:
  ⟦ is_backtrack A' T' A; var_of_lit l∈T' ⟧ ⟹ sem_lit' l A' ≠ None
  unfolding is_backtrack_def apply (cases l) by auto

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
  unfolding is_backtrack_def backtrack_def
  apply (cases l; simp; intro conjI)
  by (auto simp: restrict_map_def)


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 λe. 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)
```

unfolding *check_candidates'_def check_candidates_def*
apply *refine_rcg*
supply *RELATESI*[**where** *R*={((*c*,*A*,*prf*),(*c*,*prf*))| *c prf*. *True*}, *refine_dref_RELATES*]
supply *RELATESI*[**where** *R*={((*A*,*prf*),*prf*) | *prf*. *True* }, *refine_dref_RELATES*]
apply *refine_dref_type*
apply (*vc_solve simp*: *RELATES_def*)
done

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*×$_r$*Id*) (*check cand A prf*)
shows *check_candidates' candidatesi Ai prfi check'*
≤⇓$_E$ *ER* (*Id*×$_r$*Id*) (*check_candidates' candidates A prf check*)
unfolding *check_candidates'_def check_candidates_def*
apply *refine_rcg*
apply *refine_dref_type*
using *ERID*
apply (*vc_solve solve*: *asm_rl*)
done

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* (λ*cand_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_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)

lemma bt_rel_simps:
 ((Ai,T),A)∈bt_assign_rel A₀ ⟹ Ai=A ∧ backtrack A T = A₀ ∧ T⊆dom A
 ((Ai,T),A)∈bt_need_bt_rel A₀ ⟹ A=A₀ ∧ backtrack Ai T = A₀ ∧ T⊆dom Ai
 unfolding bt_assign_rel_def bt_need_bt_rel_def
 by (auto simp: backtrack_def in_br_conv)

lemma bt_in_bta_rel: T ⊆ dom A ⟹ ((A,T),A)∈bt_assign_rel (backtrack A T)
 by (auto simp: bt_assign_rel_def backtrack_def)


lemma and_not_C_bt_refine[refine]: ⟦ ¬is_blocked A C; (Ai,A)∈Id; (Ci,C)∈Id ⟧
 ⟹ and_not_C_bt Ai Ci ≤⇓_E UNIV (bt_assign_rel A) (ERETURN (and_not_C A C))
 apply (auto
    simp: pw_ele_iff refine_pw_simps
    simp: and_not_C_bt_def and_not_C_def bt_assign_rel_def restrict_map_def
    split!: if_splits intro!: ext)
 apply force
 apply force
 apply (metis var_of_lit.elims)
 apply force
 apply force
 apply (force simp: is_blocked_alt sem_clause'_true_conv)
 apply (force simp: is_blocked_alt sem_clause'_true_conv)
 done

lemma parse_check_blocked_bt_refine[refine]: ⟦ (Ai,A)∈Id; (iti,it)∈Id ⟧
 ⟹ parse_check_blocked_bt Ai iti
  ≤ ⇓_E UNIV (Id ×_r bt_assign_rel A ×_r Id) (parse_check_blocked A it)
 unfolding parse_check_blocked_bt_def parse_check_blocked_def
 apply clarsimp
 apply (refine_rcg)
 apply (clarsimp simp: econc_fun_ESPEC; rule ESPEC_rule)
 apply (clarsimp simp: bt_assign_rel_def; safe; simp?)

 subgoal for _ _ lit
  by (cases lit; auto simp: and_not_C_def; force)

 subgoal
  apply (
   clarsimp
    simp: and_not_C_def restrict_map_def is_blocked_def
    intro!: ext;
   safe)
  apply (force|force simp: sem_clause'_true_conv)+
  done
 subgoal by auto
 done
```

```
lemma assign_lit_bt_refine[refine]:
  ⟦ sem_lit' l A = None; ((Ai,Ti),A)∈bt_assign_rel A₀; (li,l)∈Id ⟧
  ⟹ assign_lit_bt Ai Ti li
     ≤⇓E UNIV (bt_assign_rel A₀) (ERETURN (assign_lit A l))
  unfolding assign_lit_bt_def assign_lit_def bt_assign_rel_def
  apply refine_vcg
  applyS simp
  subgoal by (cases l) auto
  subgoal by (cases l; auto simp: restrict_map_def intro!: ext)
  done

lemma apply_unit_bt_refine[refine]:
  ⟦ (ii,i)∈Id; (CMi,CM)∈Id; ((Ai,Ti),A)∈bt_assign_rel A₀ ⟧
  ⟹ apply_unit_bt ii CMi Ai Ti
     ≤⇓E UNIV (bt_assign_rel A₀) (apply_unit i CM A)
  unfolding apply_unit_bt_def apply_unit_def
  apply refine_rcg
  apply refine_dref_type
  apply (vc_solve dest!: bt_rel_simps)
  done

lemma apply_units_bt_refine[refine]:
  ⟦ (CMi,CM)∈Id; ((Ai,Ti),A)∈bt_assign_rel A₀; (iti,it)∈Id ⟧
  ⟹ apply_units_bt CMi Ai Ti iti
     ≤⇓E UNIV (bt_assign_rel A₀ ×r Id) (apply_units CM A it)
  unfolding apply_units_bt_def apply_units_def
  supply RELATESI[of bt_assign_rel A for A, refine_dref_RELATES]
  apply refine_rcg
  apply refine_dref_type
  apply auto
  done

term check_rup_proof
lemma check_rup_proof_bt_refine[refine]:
  ⟦ (si,s)∈Id; (iti,it)∈Id; (prfi,prf)∈Id ⟧
  ⟹ check_rup_proof_bt si iti prfi ≤⇓E UNIV Id (check_rup_proof s it prf)
  unfolding check_rup_proof_bt_def check_rup_proof_def
  apply refine_rcg
  apply refine_dref_type
  apply (auto simp: bt_in_bta_rel dest!: bt_rel_simps)
  done

lemma check_rat_proof_bt_refine[refine]:
  ⟦ (si,s)∈Id; (iti,it)∈Id; (prfi,prf)∈Id ⟧
  ⟹ check_rat_proof_bt si iti prfi ≤⇓E UNIV Id (check_rat_proof s it prf)
  unfolding check_rat_proof_bt_def check_rat_proof_def
  apply refine_rcg
  apply refine_dref_type
  apply (auto simp: bt_in_bta_rel dest!: bt_rel_simps) //TAKES LONG
  done


definition check_item_bt :: state ⇒ 'it ⇒ (nat × 'prf) ⇒ (_, (state × 'it × (nat × 'prf)) option) enres
  where check_item_bt ≡ 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_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)
  unfolding check_item_bt_def check_item_def
  apply refine_rcg
  apply refine_dref_type
  applyS simp

  subgoal
    apply (split item_type.split; intro impI conjI; simp)
    apply (refine_rcg; auto)
    apply (refine_rcg; auto)
    done
  done

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)
  unfolding verify_unsat_bt_def verify_unsat_def
  apply refine_rcg
  apply refine_dref_type
  apply vc_solve
  done

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: Not 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)
      )
    unfolding parse_check_clause_def
    apply (refine_vcg parse_lz_rule[where Φ=λl s. I (clause_α l) s])
    apply (vc_solve simp: F_RL)
    apply (auto simp: cref_rel_def next_it_rel_def dest!: itran_invarD)
    done

  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
  apply (rule order_trans)
  apply (rule parse_check_clause_rule_aux[of I, OF I0])
  apply (erule (1) F_RL)
  apply fact
  using assms(4,5)
  by (fastforce simp: ESPEC_rule_iff next_it_rel_def cref_rel_def)


//Iterate over already parsed clause//
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 −
  from CR obtain l it' where
        ISLZ: lz_string litZ cref l it'
    and INV: it_invar it'
    and [simp]: C = clause_α l
    by (auto simp: cref_rel_def)

  show ?thesis
    unfolding iterate_clause_def
    apply (refine_vcg
       iterate_lz_rule[OF ISLZ, where Φ=λl1 l2 s. I (clause_α l1) s])
    apply vc_solve
    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)
    done
qed

definition check_unit_clause1 A cref ≡ doE {
  ul ← iterate_clause cref (λul. True) (λl 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)
  unfolding check_unit_clause1_def check_unit_clause_def econc_fun_univ_id
```

apply *refine_vcg*
apply (*refine_vcg iterate_clause_rule*[*OF CR*, **where**
I=λ*C′ ul*. *case ul of*
None ⇒ *sem_clause′ C′ A* = *Some False*
| *Some l* ⇒ *is_unit_lit A C′ l*]
)
apply (*auto split*: *option.splits simp*: *is_unit_clause_def*)
subgoal by (*smt Diff_iff insert_iff is_unit_lit_def sem_clause′_false_conv*)
subgoal by (*smt Diff_empty Diff_insert0 boolopt_cases_aux.cases*
insertI1 insert_Diff1 is_unit_lit_def
sem_clause′_false_conv)
subgoal by (*simp add*: *is_unit_lit_def*)
subgoal apply (*drule* (*2*) *is_unit_lit_unique_ss*)
using *sem_not_false_the_unit_lit* **by** *blast*
subgoal using *is_unit_clauseI unit_contains_no_true* **by** *blast*
subgoal using *is_unit_clauseI unit_contains_no_true* **by** *blast*
subgoal by (*simp add*: *unit_clause_sem′*)
done

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*)
unfolding *resolve_id1_def resolve_id_def clausemap1_rel_def*
apply (*cases CM*; *cases CMi*)
apply (*clarsimp simp*: *pw_ele_iff refine_pw_simps*)
apply (*auto dest*!: *fun_relD*[**where** *x*=*i* **and** *x′*=*i*] *elim*: *option_relE*)
done

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*)
unfolding *apply_unit_bt_def apply_unit1_bt_def*
apply *refine_rcg*
apply (*vc_solve*)
done

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*)

unfolding *apply_units1_bt_def apply_units_bt_def*
apply *refine_rcg*
apply *refine_dref_type*
apply *vc_solve*
done

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* ≤ ⇓$_E$ *UNIV Id* (*apply_unit i CM A*)
unfolding *apply_unit_def apply_unit1_def*
apply *refine_rcg*
apply (*vc_solve*)
done

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* ≤ ⇓$_E$ *UNIV Id* (*apply_units CM A it*)
unfolding *apply_units1_def apply_units_def*
apply *refine_rcg*
apply *refine_dref_type*
apply *vc_solve*
done

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*}
apply (*auto simp*: *is_blocked_def sem_clause*′*_true_conv*)
apply (*auto simp*: *dom_def and_not_C_def split*: *if_split_asm*)
apply *force*
apply *force*
subgoal for *l* **by** (*cases l*) *auto*
done

lemma *dom_and_not_C_eq*: *dom* (*and_not_C A C*) = *dom A* ∪ *var_of_lit*‘*C*
apply (*safe*; *clarsimp?*)
apply (*force simp*: *and_not_C_def dom_def split*: *if_split_asm*) []
apply (*force simp*: *and_not_C_def*) []
subgoal for *l* **by** (*cases l*) (*auto simp*: *and_not_C_def*)
done

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*}
apply (*safe*; *clarsimp?*)
subgoal
apply (*clarsimp*

```
        simp: is_backtrack_def backtrack_def
        simp: dom_and_not_C_eq restrict_map_def)
   apply (frule (1) set_rev_mp; clarsimp)
   apply (metis option.distinct(1))
   done
 subgoal
   apply (clarsimp simp: is_backtrack_def backtrack_def restrict_map_def)
   by meson
 subgoal
   apply (clarsimp simp: is_backtrack_def backtrack_def restrict_map_def)
   by (metis sem_lit'_none_conv sem_lit_and_not_C_None_conv)
 done

definition parse_check_blocked1 A₀ cref ≡ doE {
 ((A,T),it') ← parse_check_clause cref (λ_. True) (λl (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)
   }
 }) (A₀,{});
 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
   ≤ ⇓E UNIV (cref_rel ×r Id ×r Id) (parse_check_blocked_bt A it)
 unfolding parse_check_blocked_bt_def
 apply refine_rcg
 unfolding econc_fun_ESPEC
 apply simp
 unfolding parse_check_blocked1_def
 apply (refine_vcg
    parse_check_clause_rule[where I=λC (A',T').
      is_backtrack A' T' A
    ∧ ¬is_blocked A C
    ∧ A' = and_not_C A C
  ]
 )
 apply (vc_solve
        simp: and_not_insert_False
        simp: is_backtrack_assignI is_backtrack_not_undec)

 subgoal by (auto
   simp: is_blocked_insert_iff sem_lit_and_not_C_conv
   intro: is_blockedI1 is_blockedI2) []
 subgoal by (auto simp: not_Some_bool_if) []
 subgoal by (auto simp: is_blocked_insert_iff sem_lit_and_not_C_None_conv) []
 subgoal by (auto simp: simp: and_not_insert_None) []
 subgoal
   apply (clarsimp simp: next_it_rel_def cref_rel_def)
   apply (drule (1) lz_string_determ)
   apply (intro exI conjI;
        assumption?;
        auto simp: backtrack_and_not_C_trail_eq; fail)
   done
 done

definition check_conflict_clause1 prf₀ A cref
 ≡ iterate_clause cref (λ_. True) (λl _. doE {
```

```
    CHECK (sem_lit' l A = Some False)
        (mkp_errprf STR ''Expected conflict clause'' prf₀)
  }) ()

lemma check_conflict_clause1_refine[refine]:
  assumes [simplified,simp]: (Ai,A)∈Id
  assumes CR: (cref,C)∈cref_rel
  shows check_conflict_clause1 it₀ Ai cref
      ≤⇓E UNIV Id (CHECK (is_conflict_clause A C) msg)
proof −
  have ES_REW: ⇓E UNIV Id (CHECK (is_conflict_clause A C) msg)
    = ESPEC (λ_. ¬is_conflict_clause A C) (λ_. is_conflict_clause A C)
    by (auto simp: pw_eeq_iff refine_pw_simps)

  show ?thesis
    unfolding check_conflict_clause1_def ES_REW
    apply (refine_vcg
      iterate_clause_rule[OF CR, where I=λC _. is_conflict_clause A C])
    by (auto simp: sem_clause'_false_conv)
qed

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)
  unfolding lit_in_clause1_def
  apply (refine_vcg iterate_clause_rule[OF CR, where I=λC r. r ⟷ lc∈C])
  by auto

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)
  unfolding lit_in_clause_and_not_true_def
  apply (refine_vcg iterate_clause_rule[OF CR, where I=λC f. f∈{0,1,2}
          ∧ (f=2 ⟷ sem_clause' (C−{lc}) A = Some True)
          ∧ (f=1 ⟶ lc∈C)
          ∧ (f=0 ⟶ lc∉C)])
  by (vc_solve simp: insert_minus_eq sem_clause'_true_conv solve: asm_rl)


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
        ≤⇓E UNIV (Id×rId) (and_not_C_bt A (C-{exl}))
  unfolding and_not_C_bt_def
  apply (rule EASSERT_bind_refine_right)
  apply (simp add: econc_fun_ERETURN)
  unfolding and_not_C_excl_def
  apply (refine_vcg iterate_clause_rule[OF CR,
    where I=λC' (A',T'). A' = and_not_C A (C' - {exl})
                       ∧ is_backtrack A' T' A])
  apply (vc_solve simp: insert_minus_eq)
  subgoal
    by (auto
        simp: sem_lit_and_not_C_conv sem_clause'_true_conv is_blocked_alt)
  subgoal
    by (meson boolopt_cases_aux.cases is_backtrack_not_undec)
  subgoal
    by (metis (full_types) and_not_insert_None boolopt_cases_aux.cases
          insert_minus_eq)
  subgoal
    by (metis (full_types) boolopt_cases_aux.cases is_backtrack_assignI
          sem_lit'_none_conv var_of_lit_neg_eq)
  subgoal by (simp add: and_not_insert_False)
  subgoal using backtrack_and_not_C_trail_eq by blast
  done


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
  }
```

```
//// TODO: Choice: We could 1) either remove duplicates after all candidates have been gathered, or 2) from RL list
before delete/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(n²) removing impl can be used. Moreover,
we do not expect massive duplicates. 2) In case of long candidate lists, removing 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)
    unfolding get_rat_candidates1_def get_rat_candidates_def
    apply (rewrite at Let (RL _) _ in case CMi of (CM,RL) ⇒ ⌑ Let_def)
    apply refine_rcg
    apply refine_dref_type
    apply vc_solve
    subgoal
      using CMR
      by (auto
          simp: clausemap1_rel_def cref_rel_def
          dest!: fun_relD[where x=neg_lit reslit and x'=neg_lit reslit]
          elim: option_relE
          )
    //subgoal for RL RLi using CMR apply (clarsimp simp: clausemap1_rel_def in_br_conv) apply
(drule fun_relD[where x=neg_lit reslit and x'=neg_lit reslit]) simp apply (auto simp: in_br_conv elim:
option_relE) done
    subgoal premises prems for CM RL CMi RLi cands_raw
    proof −
      from CMR prems have
        CM_ref: (CMi, CM) ∈ Id → ⟨cref_rel⟩option_rel and
        RL_ref: (RLi, RL) ∈ Id → ⟨br set (λ_. True)⟩option_rel
        by (auto simp: clausemap1_rel_def in_br_conv)

      define cands_rawi where cands_rawi = the (RLi (neg_lit reslit))
      from prems fun_relD[OF RL_ref IdI[of neg_lit reslit]]
      have [simp]: cands_raw = set cands_rawi //distinct cands_raw
        unfolding cands_rawi_def by (auto simp: in_br_conv elim: option_relE)
      note cands_rawi_def[symmetric,simp]

      show ?thesis
        apply (refine_vcg enfoldli_rule[where I=λl1 l2 s.
            //distinct (l1@l2) ∧
            s = { i∈set l1. ∃ C.
                  CM i = Some C
                ∧ neg_lit reslit∈C
                ∧ sem_clause' (C − {neg_lit reslit}) A ≠ Some True }])
        apply vc_solve

        subgoal for i l1 l2
          using fun_relD[OF CM_ref IdI[of i]]
          by (auto elim: option_relE simp: cref_rel_def in_br_conv)
        focus apply (rename_tac i l1 l2)
          apply (subgoal_tac (the (CMi i), the (CM i)) ∈ cref_rel, assumption)
          subgoal for i l1 l2
            using fun_relD[OF CM_ref IdI[of i]]
            by (force elim!: option_relE simp: cref_rel_def in_br_conv) //Takes some time//
          solved
          subgoal for i l1 l2
            using fun_relD[OF CM_ref IdI[of i]]
            by (auto elim!: option_relE simp: cref_rel_def in_br_conv)
          subgoal for i l1 l2
            using fun_relD[OF CM_ref IdI[of i]]
            by (auto elim!: option_relE simp: cref_rel_def in_br_conv)
```

```
    done
  qed
  done


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)
  unfolding backtrack1_def
  apply (refine_vcg FOREACH_rule[where I=λit A'. A' = backtrack A (T−it)])
  apply (vc_solve simp: backtrack_def)
  by (auto simp: it_step_insert_iff restrict_map_def intro!: ext)


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 nub in insert? We iterate over literals of clause, which should not contain
duplicates//


definition RL_upd cid C RL ≡ (λl. 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
  by (auto simp: RL_upd_def split: option.split)

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)
  by (auto simp: RL_upd_def split: option.split if_split_asm intro!: ext)

lemma RL_upd_insert_noeff:
  RL_upd cid C RL l = None ⟹ RL_upd cid (insert l C) RL = RL_upd cid C RL
  by (auto simp: RL_upd_def split: option.split if_split_asm intro!: ext)


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_cid: cid ∉ ⋃ran`RL//
  shows register_clause1 cid cref RLi
    ≤ ESPEC (λ_. False)
        (λRLi'. (RLi', RL_upd cid C RL) ∈ Id → ⟨br set (λ_. True)⟩option_rel)
proof −
  show ?thesis
    unfolding register_clause1_def abs_cr_register_ndj_def
    apply (refine_vcg
        iterate_clause_rule[OF CR, where
```

```
        I=λC RLi'. (RLi', RL_upd cid C RL)
                  ∈ Id → ⟨br set (λ_. True)⟩option_rel]
      )
    apply (vc_solve solve: asm_rl)

    subgoal for l
      using fun_relD[OF RL IdI[of l]] by simp
    subgoal for C l RL l'
      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) []
      done
    subgoal for RLi l'
      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)
      done
    done
qed

definition add_clause1
  :: id ⇒ 'it ⇒ ('it) clausemap1 ⇒ (_,('it) clausemap1) enres
  where add_clause1 ≡ λi 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 ≤⇓_E UNIV clausemap1_rel (add_clause i C CM)
  unfolding add_clause1_def add_clause_def
  apply (cases CMi; cases CM; simp only: split)
  subgoal for _ RLi _ RL
    apply refine_vcg
    supply RELATESI[of Id → _, refine_dref_RELATES]
    supply RELATESI[of br set (λ_. True), refine_dref_RELATES]
    apply refine_dref_type
    applyS assumption
    applyS (erule fun_relD[rotated, where f=RLi and f'=RL];
          auto simp: clausemap1_rel_def)
    applyS (auto simp: cm_upds_def)
    apply1 clarsimp subgoal for RLi' l
      apply (drule fun_relD[OF _ IdI[of l]])
      apply (cases RLi' l; cases RL l; simp)
      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)
      done
    subgoal
      apply (simp add: clausemap1_rel_def)
      apply parametricity
      by auto
    done
  done


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 −
  show ?thesis using assms
    apply (rule_tac IdD)
    unfolding clausemap1_rel_def cm_ids_def
    apply parametricity
    apply (force elim!: option_relE dest: fun_relD[OF _ IdI])
    done
qed

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 −
  have REW: ERETURN (i,CM, backtrack A T) = doE {
    let A = backtrack A T;
    ERETURN (i,CM,A)} for i CM A T
    by auto

  note [refine_dref_RELATES] = RELATESI[of clausemap1_rel]

  show ?thesis
    unfolding check_rup_proof1_def check_rup_proof_bt_def
    unfolding REW
    using SR
    apply refine_rcg
    apply refine_dref_type
    apply (vc_solve)
  subgoal by (intro refine_dref_RELATES) //TODO: refine_dref_type should always try to discharge RELATES-goals,
////////////////////////////////////////////////////////////even if they do not contain schematic vars//
    subgoal by refine_vcg auto
    done
qed

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 −
  have REW1: ERETURN (i,CM, backtrack A T) = doE {
    let A = backtrack A T;
    ERETURN (i,CM,A)} for i CM A T
    by auto

  have REW2: ERETURN (backtrack A T, it) = doE {
    let A = backtrack A T;
    ERETURN (A,it)} for A T it
    by auto

  show ?thesis
    unfolding check_rat_proof1_def check_rat_proof_bt_def
              check_rat_candidates_part1_def
    unfolding REW1 REW2
    using SR
    apply refine_vcg
    supply RELATESI[of Id → Id, refine_dref_RELATES]
    apply refine_dref_type
    supply [[goals_limit=1]]
    apply (vc_solve solve: asm_rl RELATESI) Takes its time
    done
qed


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)
```

```
  unfolding remove_id1_def remove_id_def
  by (auto
      simp: pw_ele_iff refine_pw_simps clausemap1_rel_def
      simp: in_br_conv restrict_map_def
      dest: fun_relD
      elim: option_relE
      split: prod.split
      )

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)
 unfolding remove_ids1_def remove_ids_def EWHILET_def //TODO: Register EWHILET as EWHILEIT to refine rule

  supply RELATESI[of clausemap1_rel, refine_dref_RELATES]
  apply refine_rcg
  apply refine_dref_type
  apply vc_solve
  done

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)
  unfolding check_item1_def check_item_bt_def
  apply refine_rcg
  using SR
  apply refine_dref_type
  applyS simp
  apply (split item_type.split; intro allI impI conjI; clarsimp)
  apply ((refine_rcg, refine_dref_type?); auto; fail)+
  done

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)
  })
  unfolding check_item1_def parse_type_def
  //////////////////////////////////////////////////////////////////
  apply (intro ext)
  apply (simp split: prod.split)
  apply (intro allI impI)
  apply (fo_rule fun_cong arg_cong)+
  apply (intro ext)
  apply (simp split: prod.split)
  done


definition (in −) cm_empty1 :: ('cref) clausemap1
  where cm_empty1 ≡ (Map.empty, Map.empty)
lemma cm_empty_refine[refine]: (cm_empty1, cm_empty) ∈ clausemap1_rel
  unfolding cm_empty1_def cm_empty_def clausemap1_rel_def
```

```
  by auto

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) //// Interval: Perhaps check for it? //
    else doE {
      EASSERT (sem_lit' l A = None);
      ERETURN (False,assign_lit A l)
    }
  }) (False,A);

  //// Again, check 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 −
  show ?thesis
    unfolding is_syn_taut1_def
    apply (refine_vcg
      iterate_clause_rule[OF CR, where I=λC (t,A).
          (t ⟶ is_syn_taut C)
        ∧ (¬t ⟶ (∀ l. sem_lit' l A = Some True ⟷ l∈C))
        ∧ dom A ⊆ var_of_lit`C]
      iterate_clause_rule[OF CR,
        where I=λC' A. (dom A ⊆ var_of_lit`(C − C'))]
     )
    apply (vc_solve simp: not_Some_bool_if)
    apply (auto)
    apply (auto simp: is_syn_taut_def) []
    apply (auto simp: sem_lit'_assign_conv split: if_splits) []
    apply (force simp: sem_lit'_assign_conv split: if_splits) []
    subgoal for _ l by (case_tac l; auto split: if_splits)
    subgoal premises prems for _ A
    proof −
      from prems have
        SL: ∀ l. sem_lit' l A = Some True ⟷ l∈C and
        TAUT: is_syn_taut C
        by auto

      from TAUT obtain l where l∈C neg_lit l ∈ C
        by (auto simp: is_syn_taut_conv)
      with SL
      have sem_lit' l A = Some True sem_lit' (neg_lit l) A = Some True
        by (auto simp del: sem_neg_lit')
      thus False by simp
    qed
    subgoal by fastforce
    subgoal by (fastforce simp: is_syn_taut_def)
    done
qed
```

```
definition read_cnf1 :: 'it list ⇒ ('it, clausemap1) ⌈⌈'it, clausemap1⌉⌉ ⇒ (_, 'it clausemap1) enres
where read_cnf1 L 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)
  unfolding read_cnf_new1_def read_cnf_new_def
  apply refine_rcg
  supply RELATESI[of clausemap1_rel, refine_dref_RELATES]
  apply refine_dref_type
  apply vc_solve
  applyS auto
  subgoal unfolding cref_rel_def by auto
  done

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 ≤⇓_E UNIV (clausemap1_rel ×_r Id) (init_rat_counts prf)
  unfolding init_rat_counts1_def init_rat_counts_def
        cm_init_lit_def cm_init_lit1_def
  apply refine_rcg
  supply RELATESI[of clausemap1_rel, refine_dref_RELATES]
  apply refine_dref_type
  apply (vc_solve simp: cm_empty_refine)
  subgoal by (auto simp: clausemap1_rel_def in_br_conv dest!: fun_relD)
  done

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)
}
  unfolding init_rat_counts1_def parse_type_def
  apply (simp split: prod.split)
  apply (fo_rule fun_cong arg_cong)+
  apply (intro ext)
  apply (simp split: prod.split)
  done


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
      $\leq\Downarrow_E$ UNIV Id (verify_unsat_bt F_begin F_end it prf)
  unfolding verify_unsat1_def verify_unsat_bt_def
  apply refine_rcg
  supply RELATESI[of state1_rel, refine_dref_RELATES]
  apply (auto elim: option_relE)
  done

end

4.4 Refinement 2

~~id → nat (already done for verify_unsat1) (id → 'a) → 'a option array, dynamic resizing (iam!?) literal → int ≠ 0 (literal → 'a) → 'a array, indexing: 2*|l| + sgn l. Array uses dynamic resizing. (based on iam!?) assignment: variable → bool, (iam) candidate list: stick to list, or use array-list (with reversed order) clausedb → array, which also contains proof items. size N cref → nat < N proof-item → nat < N (reference into array)~~

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 = ⌑
    apply (rule CNV_eqD)
    unfolding parse_prf_def
    apply opt_enres_unfold
    apply (rule CNV_I)
    done

  ~~synth-definition parse_type_bd is [enres_unfolds]: parse_type prf = ⌑ apply (rule CNV_eqD) unfolding parse_type_def apply opt_enres_unfold apply (rule CNV_I) done~~

  synth-definition check_unit_clause1_bd
    is [enres_unfolds]: check_unit_clause1 A cref = ⌑
    apply (rule CNV_eqD)
    unfolding check_unit_clause1_def iterate_clause_def ~~iterate_lz_def~~
    apply opt_enres_unfold
    apply (rule CNV_I)
    done

  — Optimization to reduce map lookups
  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)
    })
    unfolding resolve_id1_def
    apply (intro ext)
    apply (auto simp: pw_eeq_iff refine_pw_simps Let_def split: if_split_asm)
    done

  synth-definition resolve_id1_bd **is** [enres_unfolds]: resolve_id1 CM cid = ⌑
    apply (rule CNV_eqD)

unfolding *resolve_id1_alt*
apply *opt_enres_unfold*
apply (*rule CNV_I*)
done

synth-definition *apply_unit1_bt_bd*
is [*enres_unfolds*]: *apply_unit1_bt i CM A T* = ¤
apply (*rule CNV_eqD*)
unfolding *apply_unit1_bt_def assign_lit_bt_def*
apply *opt_enres_unfold*
apply (*rule CNV_I*)
done

synth-definition *apply_units1_bt_bd*
is [*enres_unfolds*]: *apply_units1_bt CM A T units* = ¤
apply (*rule CNV_eqD*)
unfolding *apply_units1_bt_def*
apply *opt_enres_unfold*
apply (*rule CNV_I*)
done

synth-definition *apply_unit1_bd* **is** [*enres_unfolds*]: *apply_unit1 i CM A* = ¤
apply (*rule CNV_eqD*)
unfolding *apply_unit1_def*
apply *opt_enres_unfold*
apply (*rule CNV_I*)
done

synth-definition *apply_units1_bd*
is [*enres_unfolds*]: *apply_units1 CM A units* = ¤
apply (*rule CNV_eqD*)
unfolding *apply_units1_def*
apply *opt_enres_unfold*
apply (*rule CNV_I*)
done

synth-definition *remove_ids1_bd*
is [*enres_unfolds*]: *remove_ids1 CM prf* = ¤
apply (*rule CNV_eqD*)
unfolding *remove_ids1_def remove_id1_def*
apply *opt_enres_unfold*
apply (*rule CNV_I*)
done

synth-definition *parse_check_blocked1_bd*
is [*enres_unfolds*]: *parse_check_blocked1 A cref* = ¤
apply (*rule CNV_eqD*)
unfolding *parse_check_blocked1_def parse_check_clause_def* ~~*parse_lit_def*~~
apply *opt_enres_unfold*
apply (*rule CNV_I*)
done

synth-definition *check_conflict_clause1_bd*
is [*enres_unfolds*]: *check_conflict_clause1 prf*$_0$ *A cref* = ¤
apply (*rule CNV_eqD*)
unfolding *check_conflict_clause1_def iterate_clause_def* ~~*it_step_def*~~
apply *opt_enres_unfold*
apply (*rule CNV_I*)
done

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

unfolding *and_not_C_excl_def iterate_clause_def* ////////////////
by *opt_enres_unfold*

synth-definition *lit_in_clause_and_not_true_bd*
is [*enres_breakdown*]: *lit_in_clause_and_not_true A cref lc = enres_lift* ¤
unfolding *lit_in_clause_and_not_true_def iterate_clause_def* ////////////////
by *opt_enres_unfold*

synth-definition *lit_in_clause_bd*
is [*enres_breakdown*]: *lit_in_clause1 cref lc = enres_lift* ¤
unfolding *lit_in_clause1_def iterate_clause_def* ////////////////
by *opt_enres_unfold*

synth-definition *get_rat_candidates1_bd*
is [*enres_unfolds*]: *get_rat_candidates1 CM A l* = ¤
apply (*rule CNV_eqD*)
unfolding *get_rat_candidates1_def*
apply *opt_enres_unfold*
apply (*rule CNV_I*)
done

synth-definition *add_clause1_bd*
is [*enres_breakdown*]: *add_clause1 i it CM = enres_lift* ¤
unfolding *add_clause1_def register_clause1_def iterate_clause_def* ////////////////
by *opt_enres_unfold*

synth-definition *check_rup_proof1_bd*
is [*enres_unfolds*]: *check_rup_proof1 s it prf* = ¤
apply (*rule CNV_eqD*)
unfolding *check_rup_proof1_def*
apply *opt_enres_unfold*
apply (*rule CNV_I*)
done

term *check_rat_candidates_part1*
synth-definition *check_rat_candidates_part1_bd*
is [*enres_unfolds*]:
check_rat_candidates_part1 CM reslit candidates A prf = ¤
apply (*rule CNV_eqD*)
unfolding *check_rat_candidates_part1_def*
check_candidates′_def parse_skip_listZ_def ////////////////
apply *opt_enres_unfold*
apply (*rule CNV_I*)
done

synth-definition *check_rat_proof1_bd*
is [*enres_unfolds*]: *check_rat_proof1 s it prf* = ¤
apply (*rule CNV_eqD*)
unfolding *check_rat_proof1_def*
apply *opt_enres_unfold*
apply (*rule CNV_I*)
done

synth-definition *check_item1_bd* **is** [*enres_unfolds*]: *check_item1 s it prf* = ¤
apply (*rule CNV_eqD*)
unfolding *check_item1_deforest*
apply *opt_enres_unfold*
apply (*rule CNV_I*)
done

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

unfolding *is_syn_taut1_def iterate_clause_def*
by *opt_enres_unfold*

synth-definition *read_clause_check_taut_bd*
is [*enres_unfolds*]: *read_clause_check_taut F_end F_begin A* = ⌑
apply (*rule CNV_eqD*)
unfolding *read_clause_check_taut_def*
apply *opt_enres_unfold*
apply (*rule CNV_I*)
done

synth-definition *read_cnf_new1_bd*
is [*enres_unfolds*]: *read_cnf_new1 F_begin F_end CM* = ⌑
apply (*rule CNV_eqD*)
unfolding *read_cnf_new1_def tok_fold_def*
apply *opt_enres_unfold*
apply (*rule CNV_I*)
done

synth-definition *init_rat_counts1_bd*
is [*enres_unfolds*]: *init_rat_counts1 prf* = ⌑
apply (*rule CNV_eqD*)
unfolding *init_rat_counts1_deforest cm_init_lit1_def*
apply *opt_enres_unfold*
apply (*rule CNV_I*)
done

synth-definition *goto_next_item_bd*
is [*enres_unfolds*]: *goto_next_item it* = ⌑
apply (*rule CNV_eqD*)
unfolding *goto_next_item_def*
apply *opt_enres_unfold*
apply (*rule CNV_I*)
done

synth-definition *verify_unsat1_bd*
is [*enres_unfolds*]: *verify_unsat1 F_begin F_end it prf* = ⌑
apply (*rule CNV_eqD*)
unfolding *verify_unsat1_def*
apply *opt_enres_unfold*
apply (*rule CNV_I*)
done

end

4.4.2 Instantiating Input Locale

locale *GRAT_def_loc* = *DB2_def_loc* +
fixes *prf_next* :: *'prf* ⇒ *int* × *'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*
apply *unfold_locales*
done

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_item2_loc_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_loc_def[unfolded 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 XXX2_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*)>$_t$
  unfolding *creg_register_ndj_def is_creg_def abs_cr_register_ndj_def*
  by (*sep_auto intro*!: *ext simp*: *lit_rel_def in_br_conv int_encode_eq*)

lemma *creg_register_hnr*[*sepref_fr_rules*]:
  (*uncurry2 creg_register_ndj*, *uncurry2* (*RETURN ooo abs_cr_register_ndj*))
    ∈ (*pure lit_rel*)^k *∗*$_a$ *nat_assn*^{k} *∗*$_a$ *is_creg*^{d} →$_a$ *is_creg*
  unfolding *list_assn_pure_conv option_assn_pure_conv*
  apply *sepref_to_hoare*
  apply *sep_auto*
  done

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*) ∈ *Id* → *Id* ×$_r$ *Id*
    by *auto*

  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* ⇒ _) ×$_a$ *option_assn int_assn* ×$_a$ *option_assn prfi_assn*

  lemma *prfi_assn_pure*[*safe_constraint_rules*]: *is_pure prfi_assn* **by** (*auto simp*: *prfi_assn_def*)

  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* **.**

  abbreviation *state_assn'* ≡ *cm_assn* ×$_a$ *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*))
      ∈ *id_assn*^{k} *$*$_a$ (*option_assn int_assn*)^k *$*$_a$ (*option_assn prfi_assn*)^k →$_a$ *errorp_assn*
    unfolding *prfi_assn_def option_assn_pure_conv*
    apply *sepref_to_hoare*
    by (*sep_auto simp*: *prod_assn_def split*: *prod.split*)

  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}

```
    →a errorp_assn +a int_assn ×a prfi_assn
  unfolding parse_prf_impl_def parse_prf2_def prfn_assn_def prfi_assn_def mkp_raw_err_def
  apply sepref_to_hoare
  by sep_auto
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
    →a errorp_assn +a liti.it_assn ×a bool_assn ×a assignment_assn
  unfolding read_clause_check_taut2_def
  supply [[goals_limit = 1]]
  supply liti.itran_ord[dest]
  supply sum.splits[split]
  supply liti.itran_antisym[simp]
  by sepref
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 →a cm_assn
  unfolding add_clause2_def
  supply [[goals_limit = 1]]
  by sepref
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
    →a errorp_assn +a cm_assn
  unfolding read_cnf_new2_def
  apply (rewrite at (_,_,1,¤) assignment.fold_custom_empty)
  supply [[id_debug, goals_limit=1]]
  by sepref
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
    →a errorp_assn +a
        liti.it_assn
      ×a (assignment_assn ×a list_set_assn id_assn)
      ×a liti.it_assn
  unfolding parse_check_blocked2_def
  apply (rewrite at (_, _, ¤) ls.fold_custom_empty)
  apply (rewrite in insert (var_of_lit _) _ fold_set_insert_dj)
  supply split[sepref_opt_simps]
  supply [[goals_limit = 1]]
  by sepref

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
$\to_a$ *sum_assn errorp_assn* (*pure lit_rel*)
unfolding *check_unit_clause2_def*
supply *option.split_asm*[*split*] *FIXME: Extra setup should not be necessary to translate if (x ≠ None) then ... the x*
by *sepref*
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} $\to_a$ *sum_assn errorp_assn liti.it_assn*
unfolding *resolve_id2_def*
supply *option.splits*[*split*]
by *sepref*
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}
$\to_a$ *errorp_assn* $+_a$ *assignment_assn* $\times_a$ *prfi_assn*
unfolding *apply_units2_def*
by *sepref*
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}
$\to_a$ *errorp_assn* $+_a$
(*assignment_assn* $\times_a$ *list_set_assn nat_assn*) $\times_a$ *prfi_assn*
unfolding *apply_units2_bt_def*
apply (*rewrite* **in** *insert* (*var_of_lit* _) _ *fold_set_insert_dj*)
supply [[*id_debug*, *goals_limit* = *1*]]
by *sepref*

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}
$\to_a$ *errorp_assn* $+_a$ *cm_assn* $\times_a$ *prfi_assn*
unfolding *remove_ids2_def*
supply [[*id_debug*, *goals_limit* = *1*]]
by *sepref*
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*
→$_a$ sum_assn errorp_assn unit_assn
unfolding *check_conflict_clause2_def*
supply [[*id_debug*, *goals_limit = 1*]]
by *sepref*
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*
→$_a$ prod_assn assignment_assn (list_set_assn nat_assn)
unfolding *and_not_C_excl2_def*
apply (*rewrite at* (_,_,⌑) *ls.fold_custom_empty*)
apply (*rewrite* **in** *insert* (*var_of_lit* _) _ *fold_set_insert_dj*)
by *sepref*
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*
→$_a$ bool_assn
unfolding *lit_in_clause_and_not_true2_def*
by *sepref*
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*
→$_a$ sum_assn errorp_assn (list_set_assn nat_assn)
unfolding *get_rat_candidates2_def*
supply *option.splits*[*split*]
apply (*rewrite ndls.fold_custom_empty*)
apply (*rewrite* **in** *RETURN* (*Inr* ⌑) *fold_ndls_ls_copy*)
by *sepref*

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 →$_a$ assignment_assn*
unfolding *backtrack2_def*
by *sepref*
sepref-register *backtrack2* :: *(nat⇀bool) ⇒ _*
:: *i_assignment ⇒ nat set ⇒ i_assignment nres*
lemmas [*sepref_fr_rules*] = *backtrack3.refine*

//TODO: Make this a proper definition in CNF!

lemma *not_in_cm_ids_unf*: $i \notin cm\_ids\ CM \longleftrightarrow (case\ CM\ of\ (CM,\_) \Rightarrow is\_None\ (CM\ i))$
unfolding *cm_ids_def* **by** (*cases CM*) (*auto split*: *option.splits*)

sepref-definition *check_rup_proof3* **is** *uncurry4 check_rup_proof2*
:: $liti.a\_assn^k *_a prfn\_assn^k *_a (state\_assn')^d *_a liti.it\_assn^k *_a prfi\_assn^d$
$\rightarrow_a errorp\_assn +_a state\_assn' \times_a liti.it\_assn \times_a prfi\_assn$
unfolding *check_rup_proof2_def*
apply (*rewrite not_in_cm_ids_unf*)
by *sepref*
sepref-register *check_rup_proof2*
:: *int list* $\Rightarrow$ *'prf i_prfn* $\Rightarrow$ *i_state'* $\Rightarrow$ *nat* $\Rightarrow$ *'prf i_prfi*
$\Rightarrow$ (*'prf i_prfi error* + *i_state'* $\times$ *nat* $\times$ *'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*
:: $liti.a\_assn^k *_a liti.it\_assn^k *_a lit\_assn^k \rightarrow_a bool\_assn$
unfolding *lit_in_clause2_def*
by *sepref*
sepref-register *lit_in_clause2* :: *int list* $\Rightarrow$ *nat* $\Rightarrow$ *nat literal* $\Rightarrow$ *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* ::
$liti.a\_assn^k$
$*_a\ prfn\_assn^k$
$*_a\ cm\_assn^k$
$*_a\ lit\_assn^k$
$*_a\ (list\_set\_assn\ nat\_assn)^d$
$*_a\ assignment\_assn^d$
$*_a\ prfi\_assn^d$
$\rightarrow_a errorp\_assn +_a (assignment\_assn \times_a prfi\_assn)$
unfolding *check_rat_candidates_part2_def*
supply [[*goals_limit* = *1*, *id_debug*]]
by *sepref* *//TODO//*
sepref-register *check_rat_candidates_part2* :: _ $\Rightarrow$ _ $\Rightarrow$ (*nat*) *clausemap1* $\Rightarrow$ _
:: *int list* $\Rightarrow$ *'prf i_prfn* $\Rightarrow$ *i_cm* $\Rightarrow$ *nat literal* $\Rightarrow$ *nat set* $\Rightarrow$ *i_assignment* $\Rightarrow$ *'prf i_prfi*
$\Rightarrow$ (*'prf i_prfi error* + *i_assignment* $\times$ *'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*
:: $liti.a\_assn^k *_a prfn\_assn^k *_a (state\_assn')^d *_a liti.it\_assn^k *_a prfi\_assn^d$
$\rightarrow_a errorp\_assn +_a state\_assn' \times_a liti.it\_assn \times_a prfi\_assn$
unfolding *check_rat_proof2_def short_circuit_conv*
supply [[*goals_limit* = *1*, *id_debug*]]
supply *if_splits*[*split*!]
apply (*rewrite not_in_cm_ids_unf*)
by *sepref* *//TODO//*
sepref-register *check_rat_proof2*
:: *int list* $\Rightarrow$ *'prf i_prfn* $\Rightarrow$ *i_state'* $\Rightarrow$ *nat* $\Rightarrow$ *'prf i_prfi*
$\Rightarrow$ (*'prf i_prfi error* + *i_state'* $\times$ *nat* $\times$ *'prf i_prfi*) *nres*
lemmas [*sepref_fr_rules*] = *check_rat_proof3.refine*

term *check_item2*
sepref-definition *check_item3* **is** *uncurry4 check_item2*
:: $liti.a\_assn^k *_a prfn\_assn^k *_a (state\_assn')^d *_a liti.it\_assn^k *_a prfi\_assn^d$
$\rightarrow_a errorp\_assn +_a option\_assn\ (state\_assn' \times_a liti.it\_assn \times_a prfi\_assn)$
unfolding *check_item2_def*
supply [[*goals_limit* = *1*, *id_debug*]]
by *sepref*
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
        →a bool_assn ×a assignment_assn
    unfolding is_syn_taut2_def
    by sepref
  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 →a cm_assn ×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 →a errorp_assn +a (cm_assn ×a prfi_assn)
    unfolding init_rat_counts2_def cm_empty1_def
    apply (rewrite at (¤,_) amd.fold_custom_empty)
    apply (rewrite at (_,¤) creg.fold_custom_empty)
    apply (rewrite at RETURN (_,¤) op_creg_initialize_def[symmetric])
    supply [[goals_limit = 1, id_debug]]
    by sepref
  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
      →a errorp_assn +a unit_assn
    unfolding verify_unsat2_def
    apply (rewrite at Let (_,¤) assignment.fold_custom_empty)
    apply (rewrite at do {let it0 = ¤; _ ← ASSERT (_ ≠ None); let s =
the _; _} fold_COPY)
    supply [[goals_limit = 1, id_debug]]
    supply option.splits[split] TODO: The should be translated without extra setup
    by sepref

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 rel (A,F) E (cref,rel) lst rel*~~
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_unsat3 DBi prf_next F_begin F_end it prf
<λr. *DBi* $\mapsto_a$ *DB* $*$ $\uparrow$($\neg$*isl r* $\longrightarrow$ *F_invar lst* $\wedge$ $\neg$*sat* (*F_*α *lst*))>$_t$
proof −
note *verify_unsat2.refine*[*OF GRAT_loc_axioms*, *symmetric*, *THEN meta_eq_to_obj_eq*]
also note *verify_unsat2_loc.refine*[*symmetric*,*THEN meta_eq_to_obj_eq*]
also note *verify_unsat1_bd.refine*[*symmetric*]
also note *verify_unsat1_refine*[*OF IdI IdI IdI*]
also note *verify_unsat_bt_refine*[*OF IdI IdI IdI*]
also note *verify_unsat_correct*[*OF SEG itI*]
finally have *C1*: *verify_unsat2 DB prf_next F_begin F_end it prf*
≤ *ESPEC* (λ_. *True*) (λ_. *F_invar lst* $\wedge$ $\neg$ *sat* ((*F_*α *lst*)))
by (*auto simp*: *pw_ele_iff refine_pw_simps*)

from *C1* **have** *NF*: *nofail* (*verify_unsat2 DB prf_next F_begin F_end it prf*)
by (*auto simp*: *pw_ele_iff refine_pw_simps*)

note *A* = *verify_unsat3.refine*[*of DB*, *to_hnr*]
note *A* = *A*[
of prf prf it it F_end F_end F_begin F_begin prf_next prf_next DB DBi,
unfolded autoref_tag_defs]
note *A* = *A*[*THEN hn_refineD*]
note *A* = *A*[*OF NF*]
note *A* = *A*[
unfolded hn_ctxt_def liti.it_assn_def liti.a_assn_def
b_assn_pure_conv list_assn_pure_conv sum_assn_pure_conv
option_assn_pure_conv prod_assn_pure_conv,
unfolded pure_def,
simplified, *rule_format*
]

from *C1* **have** *1*: *RETURN x* ≤ *verify_unsat2 DB prf_next F_begin F_end it prf*
$\Longrightarrow$ $\neg$*isl x* $\longrightarrow$ *F_invar lst* $\wedge$ $\neg$*sat* (*F_*α *lst*) **for** *x*
unfolding *enres_unfolds*
apply (*cases x*)
apply (*auto simp*: *pw_le_iff refine_pw_simps*)
done

from *SEG* **have** *I_begin*: *liti.I F_begin* **by** *auto*

show *?thesis*
apply (*rule cons_rule*[*OF _ _ A*])
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*)
done
qed
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 −
  {
    assume A: 1 ≤ F_end F_end ≤ length DB 0 < it it ≤ length DB

    then interpret GRAT_loc DB F_end
      apply unfold_locales by auto

    have SEG: liti.seg 1 (slice 1 F_end DB) F_end
      using ‹1 ≤ F_end› ‹F_end ≤ length DB›
      by (simp add: liti.seg_sliceI)

    have INV: it_invar F_end it_invar it
      subgoal
        by (meson SEG it_end_invar it_invar_imp_ran
            itran_invarD liti.itran_alt liti.itran_refl liti.seg_invar2)
      subgoal
        by (metis ‹0 < it› ‹it ≤ length DB› liti.seg_exists exists_leI
            it_invar_imp_ran
            itran_invarD it_end_invar liti.itran_alt liti.itran_refl
            liti.seg_invar1)
      done

    have U1: slice 1 F_end DB = tl (take F_end DB)
      unfolding Misc.slice_def
      by (metis One_nat_def drop_0 drop_Suc_Cons drop_take list.sel(3) tl_drop)

    have U2: F_invar (tl (take F_end DB)) ∧ ¬ sat (F_α (tl (take F_end DB)))
      ⟷ verify_unsat_spec DB F_end
      unfolding verify_unsat_spec_def clause_DB_valid_def clause_DB_sat_def
      using A by auto

    note verify_unsat3_correct_aux[OF SEG INV, unfolded U1 U2]
  } note [sep_heap_rules] = this


  show ?thesis
    unfolding verify_unsat_split_impl_wrapper_def by sep_auto
qed

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)
  unfolding read_assignment_def
  apply (refine_vcg EWHILEIT_rule[where R=inv_image (WF⁺) snd])
  apply vc_solve
  done

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))
  unfolding read_clause_check_sat_def
  apply (refine_vcg parse_lz_rule[
        where Φ = λl r. r ⟷ sem_clause' (clause_α l) A = Some True
      ] )
  apply (vc_solve simp: itran_invarI)
  subgoal by (auto simp: sem_clause'_true_conv)
  subgoal by auto
  done


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
  show compat_assignment A (λx. A x = Some True) unfolding compat_assignment_def by auto
qed

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))
```

unfolding *check_sat_def*
apply (*refine_vcg tok_fold_rule*[**where**
Φ = λ*lst* _. ∀ *C*∈*set* (*map clause_α lst*). *sem_clause*′ *C A* = *Some True* **and** *Z*=*litZ* **and** *l*=*lst*
]
)
apply (*vc_solve simp*: *F_invar_def*)
subgoal
apply (*rule obtain_compat_assignment*[*of A*])
apply (*auto simp*: *F_α_def sat_def sem_cnf_def dest*: *compat_clause*)
done
done

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*))
unfolding *verify_sat_def*
apply *refine_vcg*
apply *assumption*
by *auto*

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* = ⌑
apply (*rule CNV_eqD*)
unfolding *read_assignment_def*
apply *opt_enres_unfold*
apply (*rule CNV_I*)
done

synth-definition *read_clause_check_sat_bd* **is** [*enres_unfolds*]: *read_clause_check_sat itE it A* = ⌑
apply (*rule CNV_eqD*)
unfolding *read_clause_check_sat_def*
apply *opt_enres_unfold*
apply (*rule CNV_I*)
done

synth-definition *check_sat_bd* **is** [*enres_unfolds*]: *check_sat it itE* = ⌑
apply (*rule CNV_eqD*)
unfolding *check_sat_def*
apply *opt_enres_unfold*
apply (*rule CNV_I*)
done

synth-definition *verify_sat_bd* **is** [*enres_unfolds*]: *verify_sat F_begin F_end it* = ⌑
apply (*rule CNV_eqD*)
unfolding *verify_sat_def*
apply *opt_enres_unfold*
apply (*rule CNV_I*)
done

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*
by *unfold_locales*
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* .

term *read_assignment2*

sepref-definition *read_assignment3* **is** *uncurry read_assignment2*
:: $liti.a\_assn^k *_a liti.it\_assn^k \to_a error\_assn +_a assignment\_assn$
unfolding *read_assignment2_def*
apply (*rewrite* **in** *Let Map.empty assignment.fold_custom_empty*)
by *sepref*

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*
:: $liti.a\_assn^k *_a liti.it\_assn^k *_a liti.it\_assn^k *_a assignment\_assn^k \to_a error\_assn +_a liti.it\_assn \times_a bool\_assn$
unfolding *read_clause_check_sat2_def*
supply [[*goals_limit* = *1*]]
supply *liti.itran_antisym*[*simp*]
supply *sum.splits*[*split*]
by *sepref*
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 →_a error_assn +_a unit_assn
    unfolding check_sat2_def
    by sepref
  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 →_a error_assn +_a unit_assn
    unfolding verify_sat2_def
    by sepref
  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 ↦_a DB> verify_sat3 DBi F_begin F_end it <λr. DBi ↦_a DB * ↑(¬isl r ⟶ F_invar lst ∧ sat (F_α
lst)) >_t
  proof −
    note verify_sat2.refine[of DB F_begin F_end it, OF DB2_loc_axioms,symmetric,THEN meta_eq_to_obj_eq]
    also note verify_sat2_loc.refine[symmetric,THEN meta_eq_to_obj_eq]
    also note verify_sat_bd.refine[symmetric]
    also note verify_sat_correct[OF SEG itI order_refl]
    finally have C1: verify_sat2 DB F_begin F_end it ≤ ESPEC (λ_. True) (λ_. F_invar lst ∧ sat (F_α lst)) .

    from C1 have NF: nofail (verify_sat2 DB F_begin F_end it)
      by (auto simp: pw_ele_iff refine_pw_simps)

    note A = verify_sat3.refine[of DB, to_hnr, of it it F_end F_end F_begin F_begin, unfolded autoref_tag_defs]
    note A = A[THEN hn_refineD]
    note A = A[OF NF]
    note A = A[
        unfolded hn_ctxt_def liti.it_assn_def liti.a_assn_def
          b_assn_pure_conv list_assn_pure_conv sum_assn_pure_conv option_assn_pure_conv prod_assn_pure_conv,
        unfolded pure_def,
        simplified, rule_format
      ]

    from C1 have 1: RETURN x ≤ verify_sat2 DB F_begin F_end it ⟹ ¬isl x ⟶ F_invar lst ∧ sat (F_α lst) for
x
      unfolding enres_unfolds
      apply (cases x)
      apply (auto simp: pw_le_iff refine_pw_simps)
      done

    from SEG have I_begin: liti.I F_begin by auto
```

```
    show ?thesis
      apply (rule cons_rule[OF _ _ A])
      applyS sep_auto
      applyS (sep_auto dest!: 1 simp: sum.disc_eq_case split: sum.splits)
      applyS (simp add: I_begin)
      done
  qed

end

theorem verify_sat_impl_wrapper_correct[sep_heap_rules]:
  shows
    <DBi ↦a DB>
      verify_sat_impl_wrapper DBi F_end
    <λresult. DBi ↦a DB * ↑(¬isl result ⟶ verify_sat_spec DB F_end)>t
proof −
  {
    assume A: 1 ≤ F_end F_end ≤ length DB

    then interpret DB2_loc DB F_end
      apply unfold_locales by auto

    have SEG: liti.seg 1 (slice 1 F_end DB) F_end
      using ‹1 ≤ F_end› ‹F_end ≤ length DB›
      by (simp add: liti.seg_sliceI)

    have INV: it_invar F_end
      subgoal
        by (meson SEG it_end_invar it_invar_imp_ran
            itran_invarD liti.itran_alt liti.itran_refl liti.seg_invar2)
      done

    have U1: slice 1 F_end DB = tl (take F_end DB)
      unfolding slice_def
      by (metis Misc.slice_def One_nat_def drop_0 drop_Suc_Cons drop_take list.sel(3) tl_drop)

    have U2: F_invar (tl (take F_end DB)) ∧ sat (F_α (tl (take F_end DB)))
      ⟷ verify_sat_spec DB F_end
      unfolding verify_sat_spec_def clause_DB_valid_def clause_DB_sat_def using A
      by simp

    note verify_sat3_correct[OF SEG INV INV, unfolded U1 U2]
  } note [sep_heap_rules] = this

  show ?thesis
    unfolding verify_sat_impl_wrapper_def
    by sep_auto

qed

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 ↦a DB>
     verify_sat_impl_wrapper DBi F_end
   <λresult. DBi ↦a DB * ↑(¬isl result ⟶ verify_sat_spec DB F_end) >t
  by (rule verify_sat_impl_wrapper_correct)

theorem
  <DBi ↦a DB>
     verify_unsat_impl_wrapper DBi F_end it
   <λresult. DBi ↦a DB * ↑(¬isl result ⟶ verify_unsat_spec DB F_end) >t
  by (rule verify_unsat_impl_wrapper_correct)

theorem
  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
  by (rule verify_unsat_split_impl_wrapper_correct)
```

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 σ ⟷ (∀ l. l≠0 ⟶ σ l = (¬ σ (−l)))
  by (rule assn_consistent_def)
```

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 ≡ *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*)))
by (*rule verify_sat_spec_concise*)

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 ≡ *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*)))
by (*rule verify_unsat_spec_concise*)

end