Refinement to Imperative/HOL

Peter Lammich

March 21, 2015

Abstract

Many algorithms can be implemented most efficiently with imperative data structures that support destructive update. In this paper we present an approach to automatically generate verified imperative implementations from abstract specifications in Isabelle/HOL. It is based on the Isabelle Refinement Framework, for which a lot of abstract algorithms are already formalized.

Based on Imperative/HOL, which allows to generate verified imperative code, we develop a separation logic framework with automation to make it conveniently usable. On top of this, we develop an imperative collection framework, which provides standard implementations for sets and maps like hash tables and array lists. Finally, we define a refinement calculus to refine abstract (functional) algorithms to imperative ones.

Moreover, we have implemented a tool to automate the refinement process, replacing abstract data types by efficient imperative implementations from our collection framework. As a case study, we apply our tool to automatically generate verified imperative implementations of nested depth-first search and Dijkstra's shortest paths algorithm, which are considerably faster than the corresponding functional implementations. The nested DFS implementation is almost as fast as a C++ implementation of the same algorithm.

Contents

1 Introduction

We present a tool to automatically refine programs formalized in the Isabelle Refinement Framework to Imperative/HOL [1].

We define a notion of data refinement between an IRF program and an Imperative/HOL program, which supports mixing of imperative and functional data structures, and provide proof rules for the standard combinators (return, bind, recursion, while, foreach, etc.). We implement a tool which

automatically synthesizes an imperative program from an abstract functional one, selecting appropriate data structures from both our imperative collection framework and the (functional) Isabelle Collection Framework.

Moreover, we present some case studies. We can use already existing abstract formalizations for the IRF unchanged. In particular we use our tool to automatically synthesize an imperative implementation of a nested DFS algorithm from an existing abstract formalization, which is considerably faster than the original purely functional implementation, and almost as fast as a C++ implementation of the same algorithm.

2 Operation Identification Phase

```
theory Id_Op
imports
  Main
  "../Automatic_Refinement/Lib/Refine_Lib"
  "../Automatic_Refinement/Tool/Autoref_Tagging"
begin
```

The operation identification phase is adapted from the Autoref tool. The basic idea is to have a type system, which works on so called interface types (also called conceptual types). Each conceptual type denotes an abstract data type, e.g., set, map, priority queue.

Each abstract operation, which must be a constant applied to its arguments, is assigned a conceptual type. Additionally, there is a set of *p*attern rewrite rules, which are applied to subterms before type inference takes place, and which may be backtracked over. This way, encodings of abstract operations in Isabelle/HOL, like `"λ_. None"` for the empty map, or `"fun_upd m k (Some v)"` for map update, can be rewritten to abstract operations, and get properly typed.

2.1 Proper Protection of Term

The following constants are meant to encode abstraction and application as proper HOL-constants, and thus avoid strange effects with HOL's higher-order unification heuristics and automatic beta and eta-contraction.

The first step of operation identification is to protect the term by replacing all function applications and abstractions be the constants defined below.

```
definition [simp]: "PROTECT2 x (y::prop) ≡ x"
consts DUMMY :: "prop"

abbreviation PROTECT2_syn ("'(#_#')") where "PROTECT2_syn t ≡ PROTECT2
t DUMMY"
```

abbreviation *(input)ABS2 :: "('a⇒'b)⇒'a⇒'b"* (**binder** *"λ₂" 10*)
  where *"ABS2 f ≡ (λx. PROTECT2 (f x) DUMMY)"*

lemma *beta: "(λ₂x. f x)$x ≡ f x"* **by** *simp*

Another version of *op $*. Treated like *op $* by our tool. Required to avoid infinite pattern rewriting in some cases, e.g., map-lookup.

definition *APP'* (**infixl** *"$''" 900*) **where** *[simp, autoref_tag_defs]: "f$'a ≡ f a"*

Sometimes, whole terms should be protected from being processed by our tool. For example, our tool should not look into numerals. For this reason, the *PR_CONST* tag indicates terms that our tool shall handle as atomic constants, an never look into them.

The special form *UNPROTECT* can be used inside pattern rewrite rules. It has the effect to revert the protection from its argument, and then wrap it into a *PR_CONST*.

definition *[simp, autoref_tag_defs]: "PR_CONST x ≡ x"* — Tag to protect constant
definition *[simp, autoref_tag_defs]: "UNPROTECT x ≡ x"* — Gets converted to *PR_CONST*, after unprotecting its content

2.2 Operation Identification

Indicator predicate for conceptual typing of a constant

definition *intf_type :: "'a ⇒ 'b itself ⇒ bool"* (**infix** *"::ᵢ" 10*) **where**
  [simp]: "c::ᵢI ≡ True"

lemma *itypeI: "c::ᵢI"* **by** *simp*

Wrapper predicate for an conceptual type inference

definition *ID :: "'a ⇒ 'a ⇒ 'c itself ⇒ bool"*
  where *[simp]: "ID t t' T ≡ t=t'"*

2.2.1 Conceptual Typing Rules

lemma *ID_unfold_vars: "ID x y T ⟹ x≡y"* **by** *simp*
lemma *ID_PR_CONST_trigger: "ID (PR_CONST x) y T ⟹ ID (PR_CONST x) y T"* .

lemma *pat_rule:*
  "⟦ p≡p'; ID p' t' T ⟧ ⟹ ID p t' T" **by** *simp*

lemma *app_rule:*
  "⟦ ID f f' TYPE('a⇒'b); ID x x' TYPE('a)⟧ ⟹ ID (f$x) (f'$x') TYPE('b)"
  by *simp*

lemma *app'_rule:*
```
  "⟦ ID f f' TYPE('a⇒'b); ID x x' TYPE('a)⟧ ⟹ ID (f$'x) (f'$x') TYPE('b)"
```
by *simp*

lemma *abs_rule:*
```
  "⟦ ⋀x x'. ID x x' TYPE('a) ⟹ ID (t x) (t' x') TYPE('b) ⟧ ⟹
    ID (λ2x. t x) (λ2x'. t' x') TYPE('a⇒'b)"
```
by *simp*

lemma *id_rule:* `"c::ᵢI ⟹ ID c c I"` **by** *simp*

lemma *fallback_rule:*
```
  "ID (c::'a) c TYPE('c)"
```
by *simp*

lemma *unprotect_rl1:* `"ID (PR_CONST x) t T ⟹ ID (UNPROTECT x) t T"`
by *simp*

2.3 ML-Level code

ML *{**
```
infix 0 THEN_ELSE_COMB'

signature ID_OP_TACTICAL = sig
  val SOLVE_FWD: tactic' -> tactic'
  val DF_SOLVE_FWD: bool -> tactic' -> tactic'
end

structure Id_Op_Tactical :ID_OP_TACTICAL = struct

  fun SOLVE_FWD tac i st = SOLVED' (
    tac
    THEN_ALL_NEW_FWD (SOLVE_FWD tac)) i st


  (* Search for solution with DFS-strategy. If dbg-flag is given,
    return sequence of stuck states if no solution is found.
  *)
  fun DF_SOLVE_FWD dbg tac = let
    val stuck_list_ref = Unsynchronized.ref []

    fun stuck_tac _ st = if dbg then (
      stuck_list_ref := st :: !stuck_list_ref;
      Seq.empty
    ) else Seq.empty

    fun rec_tac i st = (
        (tac THEN_ALL_NEW_FWD (SOLVED' rec_tac))
        ORELSE' stuck_tac
```

```
    ) i st

  fun fail_tac _ _ = if dbg then
    Seq.of_list (rev (!stuck_list_ref))
  else Seq.empty
in
  rec_tac ORELSE' fail_tac
end

end
*}
```

ML *{**

```
  structure Id_Op = struct

    fun id_a_conv cnv ct = case term_of ct of
      @{mpat "ID _ _ _"} => Conv.fun_conv (Conv.fun_conv (Conv.arg_conv
cnv)) ct
    | _ => raise CTERM("id_a_conv",[ct])

    fun
      protect env (t1$t2) = let
        val t1 = protect env t1
        val t2 = protect env t2
      in
        @{mk_term env: "?t1.0 $ ?t2.0"}
      end
    | protect env (Abs (x,T,t)) = let
        val t = protect (T::env) t
      in
        @{mk_term env: "λv_x::?'v_T. PROTECT2 ?t DUMMY"}
      end
    | protect _ t = t

    fun protect_conv ctxt = Refine_Util.f_tac_conv ctxt
      (protect [])
      (simp_tac
        (put_simpset HOL_basic_ss ctxt addsimps @{thms PROTECT2_def APP_def})
1)

    fun unprotect_conv ctxt
      = Simplifier.rewrite (put_simpset HOL_basic_ss ctxt
        addsimps @{thms PROTECT2_def APP_def})

    fun do_unprotect_tac ctxt =
      rtac @{thm unprotect_rl1} THEN'
      CONVERSION (Refine_Util.HOL_concl_conv (fn ctxt => id_a_conv (unprotect_conv
```

```
ctxt)) ctxt)

    val cfg_id_debug =
      Attrib.setup_config_bool @{binding id_debug} (K false)

    val cfg_id_trace_fallback =
      Attrib.setup_config_bool @{binding id_trace_fallback} (K false)

    fun dest_id_rl thm = case concl_of thm of
      @{mpat (typs) "Trueprop (?c::ᵢTYPE(?'v_T))"} => (c,T)
    | _ => raise THM("dest_id_rl",~1,[thm])


    structure id_rules = Named_Thms (
      val name = @{binding id_rules};
      val description = "Operation identification rules"
    )

    structure pat_rules = Named_Thms (
      val name = @{binding pat_rules};
      val description = "Operation pattern rules"
    )

    structure def_pat_rules = Named_Thms (
      val name = @{binding def_pat_rules};
      val description = "Definite operation pattern rules (not backtracked
over)"
    )

    datatype id_tac_mode = Init | Step | Normal | Solve

    fun id_tac ss ctxt = let
      open Id_Op_Tactical
      val thy = Proof_Context.theory_of ctxt
      val certT = ctyp_of thy
      val cert = cterm_of thy

      val id_rules = id_rules.get ctxt
      val pat_rules = pat_rules.get ctxt
      val def_pat_rules = def_pat_rules.get ctxt

      val rl_net = Tactic.build_net (
        (pat_rules |> map (fn thm => thm RS @{thm pat_rule}))
        @ @{thms app_rule app'_rule abs_rule}
        @ (id_rules |> map (fn thm => thm RS @{thm id_rule}))
      )

      val def_rl_net = Tactic.build_net (
        (def_pat_rules |> map (fn thm => thm RS @{thm pat_rule}))
```

```
      )

      val id_pr_const_rename_tac =
          rtac @{thm ID_PR_CONST_trigger} THEN'
          Subgoal.FOCUS (fn { context=ctxt, prems, ... } =>
            let
              fun is_ID @{mpat "Trueprop (ID _ _ _)"} = true | is_ID _
= false
              val prems = filter (prop_of #> is_ID) prems
              val eqs = map (fn thm => thm RS @{thm ID_unfold_vars}) prems
              val conv = Conv.rewrs_conv eqs
              val conv = fn ctxt => (Conv.top_sweep_conv (K conv) ctxt)
              val conv = fn ctxt => Conv.fun2_conv (Conv.arg_conv (conv
ctxt))
              val conv = Refine_Util.HOL_concl_conv conv ctxt
            in CONVERSION conv 1 end
          ) ctxt THEN'
          rtac @{thm id_rule} THEN'
          resolve_tac id_rules

      val ityping = id_rules
        |> map dest_id_rl
        |> filter (is_Const o #1)
        |> map (apfst (#1 o dest_Const))
        |> Symtab.make_list

      val has_type = Symtab.defined ityping

      val fallback_tac = IF_EXGOAL (fn i => fn st =>
        case Logic.concl_of_goal (prop_of st) i of
          @{mpat "Trueprop (ID ?c _ _)"} => ( case c of
            Const (name,cT) => (
              case try (Sign.the_const_constraint thy) name of
                SOME T =>
                  if not (has_type name) then
                    let
                      val thm = @{thm fallback_rule}
                        |> Drule.instantiate'
                             [SOME (certT cT), SOME (certT T)]
                             [SOME (cert c)]
                      val _ = Config.get ctxt cfg_id_trace_fallback
                        andalso let
                          open Pretty
                          val p = block [str "ID_OP: Applying fallback
rule: ", Display.pretty_thm ctxt thm]
                        in
                          string_of p |> tracing;
                          false
                        end
```

```
                  in
                    rtac thm i st
                  end
                else Seq.empty
            | _ => Seq.empty
          )
        | _ => Seq.empty
        )
      | _ => Seq.empty
    )

    val init_tac = CONVERSION (
      Refine_Util.HOL_concl_conv (fn ctxt => (id_a_conv (protect_conv
ctxt)))
        ctxt
    )

    val step_tac = (FIRST' [
      atac,
      resolve_from_net_tac def_rl_net,
      resolve_from_net_tac rl_net,
      id_pr_const_rename_tac,
      do_unprotect_tac ctxt,
      fallback_tac])

    val solve_tac = DF_SOLVE_FWD (Config.get ctxt cfg_id_debug) step_tac


  in
    case ss of
      Init => init_tac
    | Step => step_tac
    | Normal => init_tac THEN' solve_tac
    | Solve => solve_tac

  end

  val setup = I
    #> id_rules.setup
    #> pat_rules.setup
    #> def_pat_rules.setup
end

*}

setup Id_Op.setup
```

2.4 Default Setup

2.4.1 Maps

typedecl *('k,'v) i_map*

definition *[simp]: "op_map_empty ≡ Map.empty"*
definition *[simp]: "op_map_is_empty m ≡ m = Map.empty"*
definition *[simp]: "op_map_update k v m ≡ m(k↦v)"*
definition *[simp]: "op_map_delete k m ≡ m|`(-{k})"*
definition *[simp]: "op_map_lookup k m ≡ m k::'a option"*

lemma *pat_map_empty[pat_rules]: "λ2_. None ≡ op_map_empty"* **by** *simp*

lemma *pat_map_is_empty[pat_rules]:*
"op =m(λ2_. None) ≡ op_map_is_empty$m"
"op =$(λ2_. None)$m ≡ op_map_is_empty$m"
"op =$(dom$m)${} ≡ op_map_is_empty$m"
"op =${}$(dom$m) ≡ op_map_is_empty$m"
unfolding *atomize_eq*
by *auto*

lemma *pat_map_update[pat_rules]:*
"fun_upd$m$k$(Some$v) ≡ op_map_update$'k$'v$'m"
by *simp*
lemma *pat_map_lookup[pat_rules]: "m$k ≡ op_map_lookup$'k$'m"*
by *simp*
lemma *op_map_delete_pat[pat_rules]:*
"op |` $ m $ (uminus $ (insert $ k $ {})) ≡ op_map_delete$'k$'m"
by *simp*

lemma *id_map_empty[id_rules]: "op_map_empty ::i TYPE(('k,'v) i_map)"*
by *simp*

lemma *id_map_is_empty[id_rules]: "op_map_is_empty ::i TYPE(('k,'v) i_map ⇒ bool)"*
by *simp*

lemma *id_map_update[id_rules]:*
"op_map_update ::i TYPE('k ⇒ 'v ⇒ ('k,'v) i_map ⇒ ('k,'v) i_map)"
by *simp*

lemma *id_map_lookup[id_rules]:*
"op_map_lookup ::i TYPE('k ⇒ ('k,'v) i_map ⇒ 'v option)"
by *simp*

lemma *id_map_delete[id_rules]:*
"op_map_delete ::i TYPE('k ⇒ ('k,'v) i_map ⇒ ('k,'v) i_map)"
by *simp*

2.4.2 Numerals

lemma *pat_numeral[def_pat_rules]: "numeral$x ≡ UNPROTECT (numeral$x)"*
by *simp*

lemma *id_nat_const[id_rules]: "(PR_CONST (a::nat)) ::$_i$ TYPE(nat)"* **by** *simp*
lemma *id_int_const[id_rules]: "(PR_CONST (a::int)) ::$_i$ TYPE(int)"* **by** *simp*

2.5 Example

schematic_lemma
"ID (λa b. (b(1::int↦2::nat) |`(-{3})) a, Map.empty, λa. case a of None ⇒ Some a | Some _ ⇒ None) (?c) (?T::?'d itself)"

using *[[id_debug]]*
by *(tactic {* Id_Op.id_tac Id_Op.Normal @{context} 1 *})*

end

3 Basic Definitions

theory *Sepref_Basic*
imports
"../Separation_Logic_Imperative_HOL/Sep_Main"
"../Collections/Refine_Dflt"
Id_Op
begin

In this theory, we define the basic concept of refinement from a nondeterministic program specified in the Isabelle Refinement Framework to an imperative deterministic one specified in Imperative/HOL.

3.1 Values on Heap

We tag every refinement assertion with the tag *hn_ctxt*, to avoid higher-order unification problems when the refinement assertion is schematic.

definition *hn_ctxt :: "('a⇒'c⇒assn) ⇒ 'a ⇒ 'c ⇒ assn"*
— Tag for refinement assertion
where
"hn_ctxt P a c ≡ P a c"

definition *pure :: "('b × 'a) set ⇒ 'a ⇒ 'b ⇒ assn"*
— Pure binding, not involving the heap
where *"pure R ≡ (λa c. ↑((c,a)∈R))"*

abbreviation *"hn_val R ≡ hn_ctxt (pure R)"*

lemma *hn_val_unfold: "hn_val R a b = ↑((b,a)∈R)"*
by *(simp add: hn_ctxt_def pure_def)*

abbreviation *hn_invalid*
— Vacuous refinement assertion for invalidated variables
where *"hn_invalid a c ≡ hn_ctxt (λ_ _. true) a c"*

lemma *fr_invalidate: "A⟹$_A$B ⟹ A⟹$_A$B*hn_invalid x x'"*
apply *(simp add: hn_ctxt_def)*
by *(rule ent_true_drop)*

3.2 Heap-Nres Refinement Calculus

Predicate that expresses refinement. Given a heap Γ, program `c` produces a heap Γ' and a concrete result that is related with predicate `R` to some abstract result from `m`

definition *"hn_refine Γ c Γ' R m ≡ nofail m ⟶*
*<Γ> c <λr. Γ' * (∃$_A$x. R x r * ↑(RETURN x ≤ m)) >$_t$"*

lemma *hn_refineI[intro?]:*
assumes *"nofail m*
*⟹ <Γ> c <λr. Γ' * (∃$_A$x. R x r * ↑(RETURN x ≤ m)) >$_t$"*
shows *"hn_refine Γ c Γ' R m"*
using *assms* **unfolding** *hn_refine_def* **by** *blast*

lemma *hn_refineD:*
assumes *"hn_refine Γ c Γ' R m"*
assumes *"nofail m"*
shows *"<Γ> c <λr. Γ' * (∃$_A$x. R x r * ↑(RETURN x ≤ m)) >$_t$"*
using *assms* **unfolding** *hn_refine_def* **by** *blast*

lemma *hn_refine_false[simp]: "hn_refine false c Γ' R m"*
by *rule auto*

lemma *hn_refine_fail[simp]: "hn_refine Γ c Γ' R FAIL"*
by *rule auto*

lemma *hn_refine_frame:*
assumes *"hn_refine P' c Q' R m"*
assumes *"P ⟹$_A$ F * P'"*
shows *"hn_refine P c (F * Q') R m"*
using *assms*
unfolding *hn_refine_def*
apply *clarsimp*
apply *(erule cons_pre_rule)*
apply *(rule cons_post_rule)*
apply *(erule frame_rule_left)*
apply *(simp only: star_aci)*

```
  apply simp
  done

lemma hn_refine_cons:
  assumes I: "P⟹_A P'"
  assumes R: "hn_refine P' c Q R m"
  assumes I': "Q⟹_A Q'"
  shows "hn_refine P c Q' R m"
  using R unfolding hn_refine_def
  apply clarsimp
  apply (rule cons_pre_rule[OF I])
  apply (erule cons_post_rule)
  apply (rule ent_star_mono ent_refl I' ent_ex_preI ent_ex_postI)+
  done

lemma hn_refine_cons_pre:
  assumes I: "P⟹_A P'"
  assumes R: "hn_refine P' c Q R m"
  shows "hn_refine P c Q R m"
  using assms
  by (rule hn_refine_cons[OF _ _ ent_refl])

lemma hn_refine_cons_post:
  assumes R: "hn_refine P c Q R m"
  assumes I: "Q⟹_A Q'"
  shows "hn_refine P c Q' R m"
  using assms
  by (rule hn_refine_cons[OF ent_refl])

lemma hn_refine_ref:
  assumes LE: "m≤m'"
  assumes R: "hn_refine P c Q R m"
  shows "hn_refine P c Q R m'"
  apply rule
  apply (rule cons_post_rule)
  apply (rule hn_refineD[OF R])
  using LE apply (simp add: pw_le_iff)
  apply (sep_auto intro: order_trans[OF _ LE])
  done

lemma hn_refine_cons_complete:
  assumes I: "P⟹_A P'"
  assumes R: "hn_refine P' c Q R m"
  assumes I': "Q⟹_A Q'"
  assumes LE: "m≤m'"
  shows "hn_refine P c Q' R m'"
  apply (rule hn_refine_ref[OF LE])
  apply (rule hn_refine_cons[OF I R I'])
  done
```

3.3 Convenience Lemmas

lemma *hn_refine_guessI:*
assumes *"hn_refine P f P' R f'"*
assumes *"f=f_conc"*
shows *"hn_refine P f_conc P' R f'"*
— To prove a refinement, first synthesize one, and then prove equality
using *assms* **by** *simp*

lemma *imp_correctI:*
assumes *R: "hn_refine* Γ *c* Γ*' R a"*
assumes *C: "a* $\leq$ *SPEC* Φ*"*
shows *"<*Γ*> c <*λ*r'.* $\exists_A$*r.* Γ*' * R r r' ** $\uparrow$*(*Φ *r)>*$_t$*"*
apply *(rule cons_post_rule)*
apply *(rule hn_refineD[OF R])*
apply *(rule le_RES_nofailI[OF C])*
apply *(sep_auto dest: order_trans[OF _ C])*
done

3.3.1 Return

lemma *hnr_RETURN_pass:*
"hn_refine (hn_ctxt R x p) (return p) (hn_invalid x p) R (RETURN x)"
— Pass on a value from the heap as return value
by *rule (sep_auto simp: hn_ctxt_def)*

lemma *hnr_RETURN_pure:*
assumes *"(c,a)*$\in$*R"*
shows *"hn_refine emp (return c) emp (pure R) (RETURN a)"*
— Return pure value
unfolding *hn_refine_def* **using** *assms*
by *(sep_auto simp: pure_def)*

3.3.2 Assertion

lemma *hnr_FAIL[simp, intro!]: "hn_refine* Γ *c* Γ*' R FAIL"*
unfolding *hn_refine_def*
by *simp*

lemma *hnr_ASSERT:*
assumes *"*Φ $\Longrightarrow$ *hn_refine* Γ *c* Γ*' R c'"*
shows *"hn_refine* Γ *c* Γ*' R (do { ASSERT* Φ*; c'})"*
using *assms*
apply *(cases* Φ*)*
by *auto*

3.3.3 Bind

```
lemma bind_det_aux: "⟦ RETURN x ≤ m; RETURN y ≤ f x ⟧ ⟹ RETURN y
≤ m ≫= f"
  apply (rule order_trans[rotated])
  apply (rule bind_mono)
  apply assumption
  apply (rule order_refl)
  apply simp
  done

lemma hnr_bind:
  assumes D1: "hn_refine Γ m' Γ1 Rh m"
  assumes D2:
    "⋀x x'. hn_refine (Γ1 * hn_ctxt Rh x x') (f' x') (Γ2 x x') R (f
x)"
  assumes IMP: "⋀x x'. Γ2 x x' ⟹A Γ' * hn_ctxt Rx x x'"
  shows "hn_refine Γ (m'≫=f') Γ' R (m≫=f)"
  using assms
  unfolding hn_refine_def
  apply (clarsimp simp add: pw_bind_nofail)
  apply (rule Hoare_Triple.bind_rule)
  apply assumption
  apply (clarsimp intro!: normalize_rules simp: hn_ctxt_def)
proof -
  fix x' x
  assume 1: "RETURN x ≤ m"
    and "nofail m" "∀x. inres m x ⟶ nofail (f x)"
  hence "nofail (f x)" by (auto simp: pw_le_iff)
  moreover assume "⋀x x'.
          nofail (f x) ⟶ <Γ1 * Rh x x'> f' x'
          <λr'. ∃Ar. Γ2 x x' * R r r' * true * ↑ (RETURN r ≤ f x)>"
  ultimately have "⋀x'. <Γ1 * Rh x x'> f' x'
          <λr'. ∃Ar. Γ2 x x' * R r r' * true * ↑ (RETURN r ≤ f x)>"
    by simp
  also have "⋀r'. ∃Ar. Γ2 x x' * R r r' * true * ↑ (RETURN r ≤ f x)
⟹A
    ∃Ar. Γ' * R r r' * true * ↑ (RETURN r ≤ f x)"
    apply sep_auto
    apply (rule ent_frame_fwd[OF IMP])
    apply frame_inference
    apply (solve_entails)
    done
  finally (cons_post_rule) have
    R: "<Γ1 * Rh x x'> f' x'
        <λr'. ∃Ar. Γ' * R r r' * true * ↑(RETURN r ≤ f x)>"
    .
  show "<Γ1 * Rh x x' * true> f' x'
          <λr'. ∃Ar. Γ' * R r r' * true * ↑ (RETURN r ≤ m ≫= f)>"
    by (sep_auto heap: R intro: bind_det_aux[OF 1])
```

qed

3.3.4 Recursion

definition *"hn_rel P m ≡ λr. ∃$_A$x. P x r * ↑(RETURN x ≤ m)"*

lemma *hn_refine_alt: "hn_refine Fpre c Fpost P m ≡ nofail m ⟶*
*<Fpre> c <λr. hn_rel P m r * Fpost>$_t$"*
apply *(rule eq_reflection)*
unfolding *hn_refine_def hn_rel_def*
apply *(simp add: hn_ctxt_def)*
apply *(simp only: star_aci)*
done

lemma *wit_swap_forall:*
assumes *W: "<P> c <λ_. true>"*
assumes *T: "(∀x. A x ⟶ <P> c <Q x>)"*
shows *"<P> c <λr. ¬$_A$ (∃$_A$x. ↑(A x) * ¬$_A$ Q x r)>"*

unfolding *hoare_triple_def Let_def*
apply *(intro conjI impI allI)*
apply *(elim conjE)*
apply *(rule hoare_tripleD[OF W], assumption+) []*
defer
apply *(elim conjE)*
apply *(rule hoare_tripleD[OF W], assumption+) []*
apply *(elim conjE)*
apply *(rule hoare_tripleD[OF W], assumption+) []*

apply *(clarsimp, intro conjI allI)*
apply *(rule models_in_range)*
apply *(rule hoare_tripleD[OF W], assumption+) []*

apply *(simp only: disj_not2, intro impI)*
apply *(drule spec[OF T, THEN mp])*
apply *(drule (2) hoare_tripleD(2))*
.

lemma *hn_admissible:*
assumes *PREC: "precise Ry"*
assumes *E: "∀f∈A. nofail (f x) ⟶ <P> c <λr. hn_rel Ry (f x) r * F>"*
assumes *NF: "nofail (INF f:A. f x)"*
shows *"<P> c <λr. hn_rel Ry (INF f:A. f x) r * F>"*
proof -
from *NF* **obtain** *f* **where** *"f∈A"* **and** *"nofail (f x)"*
by *(simp only: refine_pw_simps INF_def) blast*

with *E* **have** *"<P> c <λr. hn_rel Ry (f x) r * F>"* **by** *blast*

```
  hence W: "<P> c <λ_. true>" by (rule cons_post_rule, simp)

  from E have
    E': "∀f. f∈A ∧ nofail (f x) ⟶ <P> c <λr. hn_rel Ry (f x) r * F>"
    by blast
  from wit_swap_forall[OF W E'] have
    E'': "<P> c
     <λr. ¬_A (∃_A xa. ↑ (xa ∈ A ∧ nofail (xa x)) *
                ¬_A (hn_rel Ry (xa x) r * F))>" .

  thus ?thesis
    apply (rule cons_post_rule)
    unfolding entails_def hn_rel_def
    apply clarsimp
  proof -
    fix h as p
    assume A: "∀f. f∈A ⟶ (∃a.
      ((h, as) ⊨ Ry a p * F ∧ RETURN a ≤ f x)) ∨ ¬ nofail (f x)"
    with ‘f∈A‘ and ‘nofail (f x)‘ obtain a where
      1: "(h, as) ⊨ Ry a p * F" and "RETURN a ≤ f x"
      by blast
    have
      "∀f∈A. nofail (f x) ⟶ (h, as) ⊨ Ry a p * F ∧ RETURN a ≤ f x"
    proof clarsimp
      fix f'
      assume "f'∈A" and "nofail (f' x)"
      with A obtain a' where
        2: "(h, as) ⊨ Ry a' p * F" and "RETURN a' ≤ f' x"
        by blast

      moreover note preciseD'[OF PREC 1 2]
      ultimately show "(h, as) ⊨ Ry a p * F ∧ RETURN a ≤ f' x" by simp
    qed
    hence "RETURN a ≤ (INF f:A. f x)"
      by (metis (mono_tags) le_INF_iff le_nofailI)
    with 1 show "∃a. (h, as) ⊨ Ry a p * F ∧ RETURN a ≤ (INF f:A. f
x)"
      by blast
  qed
qed

lemma hn_admissible':
  assumes PREC: "precise Ry"
  assumes E: "∀f∈A. nofail (f x) ⟶ <P> c <λr. hn_rel Ry (f x) r *
F>_t"
  assumes NF: "nofail (INF f:A. f x)"
  shows "<P> c <λr. hn_rel Ry (INF f:A. f x) r * F>_t"
  apply (rule hn_admissible[OF PREC, where F="F*true", simplified])
  apply simp
```

```
  by fact+

lemma hnr_RECT:
  assumes S: "⋀cf af ax px. ⟦
    ⋀ax px. hn_refine (hn_ctxt Rx ax px * F) (cf px) (F' ax px) Ry (af
ax)⟧
    ⟹ hn_refine (hn_ctxt Rx ax px * F) (cB cf px) (F' ax px) Ry (aB
af ax)"
  assumes M: "(⋀x. mono_Heap (λf. cB f x))"
  assumes PREC: "precise Ry"
  shows "hn_refine
    (hn_ctxt Rx ax px * F) (heap.fixp_fun cB px) (F' ax px) Ry (RECT aB
ax)"
  unfolding RECT_gfp_def
proof (simp, intro conjI impI)
  assume "trimono aB"
  hence "mono aB" by (simp add: trimonoD)
  have "∀ax px.
    hn_refine (hn_ctxt Rx ax px * F) (heap.fixp_fun cB px) (F' ax px)
Ry
      (gfp aB ax)"
    apply (rule gfp_cadm_induct[OF _ _ `mono aB`])

    apply rule
    apply (auto simp: hn_refine_alt intro: hn_admissible'[OF PREC]) []

    apply (auto simp: hn_refine_alt) []

    apply clarsimp
    apply (subst heap.mono_body_fixp[of cB, OF M])
    apply (rule S)
    apply blast
    done
  thus "hn_refine (hn_ctxt Rx ax px * F)
     (ccpo.fixp (fun_lub Heap_lub) (fun_ord Heap_ord) cB px) (F' ax px)
Ry
     (gfp aB ax)" by simp
qed

lemma hnr_If:
  assumes P: "Γ ⟹A Γ1 * hn_val bool_rel a a'"
  assumes RT: "a ⟹ hn_refine (Γ1 * hn_val bool_rel a a') b' Γ2b R
b"
  assumes RE: "¬a ⟹ hn_refine (Γ1 * hn_val bool_rel a a') c' Γ2c R
c"
  assumes IMP: "Γ2b ∨A Γ2c ⟹A Γ'"
  shows "hn_refine Γ (if a' then b' else c') Γ' R (if a then b else c)"
  apply rule
  apply (cases a)
```

```
    apply (rule cons_pre_rule[OF P])
    apply vcg
    apply (frule RT[unfolded hn_refine_def])
    apply (simp add: pure_def)
    apply (erule cons_post_rule)
    apply (sep_auto intro: ent_star_mono ent_disjI1[OF IMP] ent_refl)
    apply (simp add: pure_def hn_ctxt_def)

    apply (rule cons_pre_rule[OF P])
    apply vcg
    apply (simp add: hn_ctxt_def pure_def)

    apply (frule RE[unfolded hn_refine_def])
    apply (simp add: hn_ctxt_def pure_def)
    apply (erule cons_post_rule)
    apply (sep_auto intro: ent_star_mono ent_disjI2[OF IMP] ent_refl)
  done
```

3.4 ML-Level Utilities

```
ML {*
  signature SEPREF_BASIC = sig
    (* Conversion for hn_refine - term*)
    val hn_refine_conv : conv -> conv -> conv -> conv -> conv -> conv

    (* Conversion on abstract value (last argument) of hn_refine - term
*)
    val hn_refine_conv_a : conv -> conv

    (* Conversion on abstract value of hn_refine term in conclusion of
theorem *)
    val hn_refine_concl_conv_a: (Proof.context -> conv) -> Proof.context
-> conv

    (* Make certified == *)
    val mk_cequals : cterm * cterm -> cterm
    (* Make ⟹_A *)
    val mk_entails : term * term -> term

    (* Make separation conjunction *)
    val mk_star : term * term -> term
    (* Make separation conjunction from list *)
    val list_star : term list -> term
    (* Decompose separation conjunction *)
    val strip_star : term -> term list

    (* Check if term is hn_ctxt-assertion *)
    val is_hn_ctxt : term -> bool
    (* Decompose hn_ctxt-assertion *)
```

```
    val dest_hn_ctxt : term -> term * term * term
    (* Decompose hn_ctxt-assertion, NONE if term has wrong format *)
    val dest_hn_ctxt_opt : term -> (term * term * term) option

    (* Decompose function application, return constructor to rebuild it
*)
    val dest_APPc : term -> (term * term) * (term * term -> term)
    (* Get argument of function application, return constructor to exchange
argument *)
    val dest_APP_argc : term -> term * (term -> term)
    (* Get arguments, return constructor to exchange arguments *)
    val strip_APP_argc : term -> term list * (term list -> term)
    (* Get function and arguments, return constructor to exchange *)
    val strip_APPc :
       term -> (term * term list) * (term * term list -> term)

  end

  structure Sepref_Basic : SEPREF_BASIC = struct
    local open Conv in
      fun hn_refine_conv c1 c2 c3 c4 c5 ct = case term_of ct of
        @{mpat "hn_refine _ _ _ _ _"} => let
          val cc = combination_conv
        in
          cc (cc (cc (cc (cc all_conv c1) c2) c3) c4) c5 ct
        end
      | _ => raise CTERM ("hn_refine_conv",[ct])

      val hn_refine_conv_a = hn_refine_conv all_conv all_conv all_conv
all_conv

      fun hn_refine_concl_conv_a conv ctxt = Refine_Util.HOL_concl_conv

        (fn ctxt => hn_refine_conv_a (conv ctxt)) ctxt

    end

    (* FIXME: Strange dependency! *)
    val mk_cequals = uncurry SMT_Utils.mk_cequals

    val mk_entails = HOLogic.mk_binrel @{const_name "entails"}

    val mk_star = HOLogic.mk_binop @{const_name "Groups.times_class.times"}

    fun list_star [] = @{term "emp::assn"}
      | list_star [a] = a
      | list_star (a::l) = mk_star (list_star l,a)

    fun strip_star @{mpat "?a*?b"} = strip_star a @ strip_star b
```

```
  | strip_star t = [t]


fun is_hn_ctxt @{mpat "hn_ctxt _ _ _"} = true | is_hn_ctxt _ = false
fun dest_hn_ctxt @{mpat "hn_ctxt ?R ?a ?p"} = (R,a,p)
  | dest_hn_ctxt t = raise TERM("dest_hn_ctxt",[t])

fun dest_hn_ctxt_opt @{mpat "hn_ctxt ?R ?a ?p"} = SOME (R,a,p)
  | dest_hn_ctxt_opt _ = NONE


fun
  dest_APPc (Const(@{const_name "APP"},T)$f$x) =
    ((f,x),fn (f,x) => Const(@{const_name "APP"},T)$f$x)
| dest_APPc t = raise TERM("dest_APPc",[t])

local
  fun
    strip_APPc_aux (Const(@{const_name "APP"},T)$f$x) = let
      val ((f',l), c) = strip_APPc_aux f
      val l' = x::l
      fun c' (f,x::l) = Const(@{const_name "APP"},T)$c (f,l)$x
        | c' _ = error "strip_APPc (constructor): Too few args"

    in
      ((f',l'),c')
    end
  | strip_APPc_aux t = ((t,[]),
      fn (t,[]) => t | _ => error "strip_APPc (constructor): Extra
#args")
in
  fun strip_APPc t = let
    val ((f,l),c) = strip_APPc_aux t
  in
    ((f,rev l), fn (f,l) => c (f, rev l))
  end

end

fun dest_APP_argc t = let
  val ((f,x),c) = dest_APPc t
in
  (x,curry c f)
end

fun strip_APP_argc t = let
  val ((f,l),c) = strip_APPc t
in
  (l,curry c f)
```

```
    end


  end
*}
```

end

4 Monadify

theory *Sepref_Monadify*
imports *Sepref_Basic Id_Op*
begin

In this phase, a monadic program is converted to complete monadic form, that is, computation of compound expressions are made visible as top-level operations in the monad.

The monadify process is separated into 2 steps.

1. In a first step, eta-expansion is used to add missing operands to operations and combinators. This way, operators and combinators always occur with the same arity, which simplifies further processing.

2. In a second step, computation of compound operands is flattened, introducing new bindings for the intermediate values.

definition *SP* — Tag to protect content from further application of arity and combinator equations
  where *[simp]: "SP x ≡ x"*
lemma *SP_cong[cong]: "SP x ≡ SP x"* **by** *simp*

definition *RCALL* — Tag that marks recursive call
  where *[simp]: "RCALL D ≡ D"*
definition *EVAL* — Tag that marks evaluation of plain expression for monadify phase
  where *[simp]: "EVAL x ≡ RETURN x"*

Internally, the package first applies rewriting rules from *sepref_monadify_arity*, which use eta-expansion to ensure that every combinator has enough actual parameters. Moreover, this phase will mark recursive calls by the tag *RCALL*.

Next, rewriting rules from *sepref_monadify_comb* are used to add *EVAL*-tags to plain expressions that should be evaluated in the monad.

Finally, the expressions inside the eval-tags are flattened. In this step, rewrite rules from *sepref_monadify_evalcomb* are applied, in conjunction with a default rule that evaluates the arguments of each function from left to right.

lemma *monadify_simps:*

```
  "bind$(RETURN$x)$(λ2x. f x) = f x"
  "EVAL$x ≡ RETURN$x"
  by simp_all
```

definition *[simp]: "PASS ≡ RETURN"*
— Pass on value, invalidating old one

lemma *remove_pass_simps:*
```
  "bind$(PASS$x)$(λ2x. f x) ≡ f x"
  "bind$m$(λ2x. PASS$x) ≡ m"
  by simp_all
```

ML *{**
```
  structure Sepref_Monadify = struct
    structure arity_eqs = Named_Thms (
      val name = @{binding sepref_monadify_arity}
      val description = "Sepref.Monadify: Arity alignment equations"
    )

    structure comb_eqs = Named_Thms (
      val name = @{binding sepref_monadify_comb}
      val description = "Sepref.Monadify: Combinator equations"
    )

    structure eval_comb_eqs = Named_Thms (
      val name = @{binding sepref_monadify_evalcomb}
      val description = "Sepref.Monadify: Eval-Combinator equations"
    )

    local
      fun cr_var (i,T) = ("v"^string_of_int i, Free ("__v"^string_of_int
i,T))

      fun lambda2_name n t = let
        val t = @{mk_term "PROTECT2 ?t DUMMY"}
      in
        Term.lambda_name n t
      end


      fun
        bind_args exp0 [] = exp0
      | bind_args exp0 ((x,m)::xms) = let
          val lr = bind_args exp0 xms
            |> incr_boundvars 1
            |> lambda2_name x
        in @{mk_term "bind$?m$?lr"} end
```

```
      fun monadify t = let
        val (f,args) = Autoref_Tagging.strip_app t
        val _ = not (is_Abs f) orelse
          raise TERM ("monadify: higher-order",[t])

        val argTs = map fastype_of args
        (*val args = map monadify args*)
        val args = map (fn a => @{mk_term "EVAL$?a"}) args

        (*val fT = fastype_of f
        val argTs = binder_types fT*)

        val argVs = tag_list 0 argTs
          |> map cr_var

        val res0 = let
          val x = Autoref_Tagging.list_APP (f,map #2 argVs)
        in
          @{mk_term "RETURN$?x"}
        end

        val res = bind_args res0 (argVs ~~ args)
      in
        res
      end

      fun monadify_conv_aux ctxt ct = case term_of ct of
        @{mpat "EVAL$_"} => let
          val ss = ctxt
          val ss = (ss addsimps @{thms monadify_simps})
          val tac = (simp_tac ss 1)
        in (*Refine_Util.monitor_conv "monadify"*) (
          Refine_Util.f_tac_conv ctxt (dest_comb #> #2 #> monadify) tac)
ct
        end
      | t => raise TERM ("monadify_conv",[t])

      fun extract_comb_conv ctxt = Conv.rewrs_conv (eval_comb_eqs.get
ctxt)
    in

      val monadify_conv = Conv.top_conv
        (fn ctxt =>
          Conv.try_conv (
            extract_comb_conv ctxt else_conv monadify_conv_aux ctxt
          )
        )

    end
```

```
    fun mark_params env @{mpat "RETURN$(?x AS_s mpaq_Bound _)"} =
          @{mk_term env: "PASS$?x"}
      | mark_params env (t1$t2) = mark_params env t1 $ mark_params env
t2
      | mark_params env (Abs (x,T,t)) = Abs (x,T,mark_params (T::env)
t)
      | mark_params _ t = t

    fun mark_params_conv ctxt = Refine_Util.f_tac_conv ctxt
      (mark_params [])
      (simp_tac (put_simpset HOL_basic_ss ctxt addsimps @{thms PASS_def})
1)

    fun monadify_tac ctxt = let
      val arity1_ss = put_simpset HOL_basic_ss ctxt
        addsimps arity_eqs.get ctxt
        |> Simplifier.add_cong @{thm SP_cong}

      val arity2_ss = put_simpset HOL_basic_ss ctxt
        addsimps @{thms beta SP_def}

      val arity_tac = simp_tac arity1_ss THEN' simp_tac arity2_ss

      val comb1_ss = put_simpset HOL_basic_ss ctxt
        addsimps comb_eqs.get ctxt
        addsimps eval_comb_eqs.get ctxt
        |> Simplifier.add_cong @{thm SP_cong}

      val comb2_ss = put_simpset HOL_basic_ss ctxt
        addsimps @{thms SP_def}

      val comb_tac = simp_tac comb1_ss THEN' simp_tac comb2_ss

      open Sepref_Basic
    in
      arity_tac
      THEN' comb_tac
      THEN' CONVERSION (hn_refine_concl_conv_a monadify_conv ctxt)
      THEN' CONVERSION (hn_refine_concl_conv_a (K (mark_params_conv ctxt))
ctxt)
      THEN' simp_tac
        (put_simpset HOL_basic_ss ctxt addsimps @{thms remove_pass_simps})
    end


    val setup = I
      #> arity_eqs.setup
      #> comb_eqs.setup
```

```
    #> eval_comb_eqs.setup
  end
*}

setup Sepref_Monadify.setup

lemma dflt_arity[sepref_monadify_arity]:
  "RECT ≡ λ2B x. SP RECT$(λ2D x. B$(λ2x. RCALL$D$x)$x)$x"
  "case_list ≡ λ2fn fc l. SP case_list$fn$(λ2x xs. fc$x$xs)$l"
  "case_prod ≡ λ2fp p. SP case_prod$(λ2a b. fp$a$b)$p"
  "If ≡ λ2b t e. SP If$b$t$e"
  "Let ≡ λ2x f. SP Let$x$(λ2x. f$x)"
  by (simp_all only: SP_def APP_def PROTECT2_def RCALL_def)


lemma dflt_comb[sepref_monadify_comb]:
  "⋀B x. RECT$B$x ≡ bind$(EVAL$x)$(λ2x. SP (RECT$B$x))"
  "⋀D x. RCALL$D$x ≡ bind$(EVAL$x)$(λ2x. SP (RCALL$D$x))"
  "⋀fn fc l. case_list$fn$fc$l ≡ bind$(EVAL$l)$(λ2l. (SP case_list$fn$fc$l))"
  "⋀fp p. case_prod$fp$p ≡ bind$(EVAL$p)$(λ2p. (SP case_prod$fp$p))"
  "⋀fn fs ov. case_option$fn$fs$ov
    ≡ bind$(EVAL$ov)$(λ2ov. (SP case_option$fn$fs$ov))"
  "⋀b t e. If$b$t$e ≡ bind$(EVAL$b)$(λ2b. (SP If$b$t$e))"
  "⋀x. RETURN$x ≡ bind$(EVAL$x)$(λ2x. SP (RETURN$x))"
  "⋀x f. Let$x$f ≡ bind$(EVAL$x)$(λ2x. (SP Let$x$f))"
  by (simp_all)


lemma dflt_plain_comb[sepref_monadify_comb]:
  "EVAL$(If$b$t$e) ≡ bind$(EVAL$b)$(λ2b. If$b$(EVAL$t)$(EVAL$e))"
  "EVAL$(case_list$fn$(λ2x xs. fc x xs)$l) ≡
    bind$(EVAL$l)$(λ2l. case_list$(EVAL$fn)$(λ2x xs. EVAL$(fc x xs))$l)"
  "EVAL$(case_prod$(λ2a b. fp a b)$p) ≡
    bind$(EVAL$p)$(λ2p. case_prod$(λ2a b. EVAL$(fp a b))$p)"
  "EVAL$(case_option$fn$(λ2x. fs x)$ov) ≡
    bind$(EVAL$ov)$(λ2ov. case_option$(EVAL$fn)$(λ2x. EVAL$(fs x))$ov)"
  apply (rule eq_reflection, simp split: list.split prod.split option.split)+
  done

lemma evalcomb_PR_CONST[sepref_monadify_evalcomb]:
  "EVAL$(PR_CONST x) ≡ RETURN$(PR_CONST x)"
  by simp


end
```

5 Linearity Analysis

theory *Sepref_Lin_Ana*

imports *Sepref_Monadify*
begin

The goal of this phase is to add to each occurrence of a bound variable a flag that indicates whether the value stored in this bound variable is accessed again (non-linear) or not (linear).

The intention is that, for linear references to bound variables, the content of the variable on the heap may be destroyed.

datatype *lin_type* — Type of linearity annotation
  = LINEAR | NON_LINEAR

definition *LIN_ANNOT* — Tag to annotate linearity
  :: "'a => lin_type => 'a"
  where *[simp]: "LIN_ANNOT x T == x"*

abbreviation *is_LINEAR ("_ ^{L}")* **where** *"x^L == LIN_ANNOT x LINEAR"*
abbreviation *is_NON_LINEAR ("_ ^{N}")* **where** *"x^N == LIN_ANNOT x NON_LINEAR"*

Internally, this linearity analysis works in two stages. First, a constraint system is generated from the program, which is solved in the second stage, to obtain the linearity annotations.

In the following, we define constants to represent the constraints

type_synonym *la_skel = unit*

consts
  la_seq :: "la_skel => la_skel => la_skel" — Sequential evaluation
  la_choice :: "la_skel => la_skel => la_skel" — Alternatives
  la_rec :: "(la_skel => la_skel) => la_skel" — Recursion
  la_rcall :: "la_skel => la_skel" — Recursive call
  la_op :: "'a => la_skel" — Primitive operand
  la_lambda :: "(la_skel ⇒ la_skel) ⇒ la_skel" — Lambda abstraction

  SKEL :: "'a => la_skel" — Tag to indicate progress of constraint system generation
  UNSKEL :: "la_skel ⇒ 'a" — Placed on arguments of recursion and abstraction

definition *lin_ana* — Tag to indicate linearity analysis
  where *[simp]: "lin_ana x ≡ True"*
lemma *lin_anaI: "lin_ana x"* **by** *simp*

lemma *lin_ana_init:*
  assumes *"lin_ana (SKEL a)"*
  assumes *"hn_refine Γ c Γ' R a"*
  shows *"hn_refine Γ c Γ' R a"*
  by *fact*

ML *{*
  structure Sepref_Lin_Ana = struct
    structure skel_eqs = Named_Thms (
      val name = @{binding sepref_la_skel}
      val description = "Sepref.Linearity-Analysis: Skeleton equations"
)

    local
      fun add_annot_vars t = let
        val prefix =
          let
            val context = Name.make_context (Term.add_var_names t [] |>
map #1)
          in (Name.variant "a" context |> #1) ^ "_" end

        fun f (e,i) (t1$t2) = let
              val (i,t1) = f (e,i) t1
              val (i,t2) = f (e,i) t2
            in (i,t1$t2) end
          | f (e,i) (Abs(x,T,t)) = let
              val (i,t) = f (T::e,i) t
            in (i,Abs (x,T,t)) end
          | f (e,i) (t as Bound _) = let
              val a = Var ((prefix^string_of_int i,0),@{typ lin_type})
              val t = @{mk_term e: "LIN_ANNOT ?t ?a"}
            in (i+1,t) end
          | f (_,i) t = (i,t)

      in
        f ([],0) t |> #2
      end
    in
      (* Add schematic linearity annotation to each bound variable *)
      fun add_annot_vars_conv ctxt = Refine_Util.f_tac_conv ctxt
        (add_annot_vars)
        (simp_tac
          (put_simpset HOL_basic_ss ctxt addsimps @{thms LIN_ANNOT_def})
1)

    end

    local
      fun fin_annot_vars (t as @{mpat "_ $^L"}) = t
        | fin_annot_vars (t as @{mpat "_ $^N"}) = t
        | fin_annot_vars (@{mpat "LIN_ANNOT ?x _"}) = x
        | fin_annot_vars (t1$t2) = fin_annot_vars t1 $ fin_annot_vars
t2
        | fin_annot_vars (Abs (x,T,t)) = Abs (x,T,fin_annot_vars t)*}*

```
      | fin_annot_vars t = t
  in
    (* Remove all unfinished linearity annotations *)
    fun fin_annot_vars_conv ctxt = Refine_Util.f_tac_conv ctxt
      (fin_annot_vars)
      (simp_tac
        (put_simpset HOL_basic_ss ctxt addsimps @{thms LIN_ANNOT_def})
1)
  end

  local
    datatype env = Val of bool | Rec of int list

    fun set_used (Val _) = Val true | set_used x = x

    fun
      merge_env [] [] = []
    | merge_env (Val b1::r1) (Val b2::r2)
      = Val (b1 orelse b2) :: merge_env r1 r2
    | merge_env (Rec l1::r1) (Rec _::r2) = Rec l1 :: merge_env r1 r2
    | merge_env _ _ = error "merge_env: Unequal length or rec/val mismatch"

    fun
      lin_ana (env : env list )
        @{mpat "la_seq ?s ?t"}
        : env list * (term * term) list
      =
      let
        val (env,s1) = lin_ana env t
        val (env,s2) = lin_ana env s
      in
        (env,s1@s2)
      end
    | lin_ana env @{mpat "la_choice ?s ?t"} = let
        val (env1,s1) = lin_ana env s
        val (env2,s2) = lin_ana env t
      in (merge_env env1 env2, s1@s2) end
    | lin_ana env @{mpat "la_rec (λ_. ?f)"} = let
        val f_used = add_loose_bnos (f,1,[]) |> map (curry op + 1)
        val env = Rec f_used :: env
        val (env,s) = lin_ana env f
      in (tl env,s) end
    | lin_ana env @{mpat "la_rcall (mpaq_STRUCT (mpaq_Bound ?i))"} =
let
        val used = case nth env i of Rec used => used
          | _ => raise TERM ("lin_ana: rcall rec/val mismatch",[Bound
i])
        val used = map (curry op + i) used
```

```
      val env = map_index
        (fn (i,e) => if member op= used i then set_used e else e)
        env

    in
      (env,[])
    end
  | lin_ana env @{mpat "la_lambda (λ_. ?f)"} = let
      val (env,s) = lin_ana (Val false::env) f
    in
      (tl env,s)
    end
  | lin_ana env @{mpat "la_op ?t"} = let
      (* Collect loose bound vars with their annotations *)
      fun collect n @{mpat "LIN_ANNOT (mpaq_STRUCT (mpaq_Bound ?i))
?a"} =
            if i>=n then [(i-n,a)] else []
        | collect n (t1$t2) = collect n t1 @ collect n t2
        | collect n (Abs (_,_,t)) = collect (n+1) t
        | collect _ _ = []

      val used = collect 0 t

      (* Check whether they are used in env ... add subst to result
*)
      val s = map (fn (i,a) => case nth env i of
          Val false => (a,@{const LINEAR})
        | Val true => (a,@{const NON_LINEAR})
        | _ =>
            raise TERM ("lin_ana: Invalid occurence of recursion var",[t])
      ) used

      (* Mark them as used in env *)
      val used = map #1 used
      val env = map_index (fn (i,e) =>
        if member op= used i then set_used e else e) env

    in
      (env,s)
    end
  | lin_ana _ t = raise TERM ("lin_ana: Invalid",[t])


  fun lin_ana_trans t = let
    val (_,s) = lin_ana [] t
    val res = subst_atomic s t
  in
    res
  end
```

```
    in
      (* Solve linearity constraint system: As conversion*)
      fun lin_ana_conv ctxt = Refine_Util.f_tac_conv ctxt
        (lin_ana_trans)
        (simp_tac (put_simpset HOL_basic_ss ctxt addsimps @{thms LIN_ANNOT_def})
1)

      (* Solve linearity constraint system: As tactic *)
      fun lin_ana_inst_tac i st = case Logic.concl_of_goal (prop_of st)
i of
        @{mpat "Trueprop (lin_ana ?t)"} => let
          val thy = theory_of_thm st
          val cert = cterm_of thy
          val (_,s) = lin_ana [] t
          val s = map (pairself cert) s
        in
          ( rtac @{thm lin_anaI} i
            THEN PRIMITIVE (Thm.instantiate ([],s))
          ) st
        end
      | _ => Seq.empty

    end

    (* TODO: Move *)
    fun ex_aterm P (t1$t2) = ex_aterm P t1 orelse ex_aterm P t2
      | ex_aterm P (Abs (_,_,t)) = ex_aterm P t
      | ex_aterm P t = P t

    val contains_skel = ex_aterm (fn @{mpat "SKEL"} => true | _ => false)

    (* Perform linearity analysis *)
    fun lin_ana_tac ctxt = let
      fun err_tac i st = let
        val g = Logic.get_goal (prop_of st) i
        val _ = Pretty.block [
          Pretty.str "Unresolved combinators remain:", Pretty.brk 1,
          Syntax.pretty_term ctxt g
        ] |> Pretty.string_of |> tracing
      in
        Seq.empty
      end

      open Sepref_Basic
    in
      (* Add schematic annotations *)
      CONVERSION (hn_refine_concl_conv_a (K (add_annot_vars_conv ctxt))
ctxt)
```

```
      (* Generate constraint system *)
      THEN' rtac @{thm lin_ana_init}
      THEN' simp_tac (put_simpset HOL_basic_ss ctxt addsimps skel_eqs.get
ctxt)
      THEN' (
        COND' contains_skel
        THEN_ELSE'
        ( err_tac, (* CS not fully generated*)
          lin_ana_inst_tac (* Solve CS*)
          THEN'
            CONVERSION (hn_refine_concl_conv_a (K (fin_annot_vars_conv
ctxt)) ctxt)
        ))
    end

    val setup = skel_eqs.setup
  end
*}

setup Sepref_Lin_Ana.setup

lemma dflt_skel_eqs[sepref_la_skel]:
  "⋀a b. SKEL (bind$a$b) ≡ la_seq (SKEL a) (SKEL b)"
  "⋀a. SKEL (RETURN$a) ≡ la_op a"
  "⋀a. SKEL (PASS$a) ≡ la_op a"
  "⋀f x. SKEL (RECT$(λ2D. f D)$x)
    ≡ la_seq (la_op x) (la_rec (λD. SKEL (f (UNSKEL D))))"
  "⋀D x. SKEL (RCALL$(UNSKEL D)$x) ≡ la_seq (la_op x) (la_rcall D)"
  "⋀D a. la_rcall (LIN_ANNOT D a) ≡ la_rcall D"
  "⋀f. SKEL (λ2x. f x) ≡ la_lambda (λx. SKEL (f (UNSKEL x)))"
  "⋀v a. LIN_ANNOT (UNSKEL v) a = UNSKEL (LIN_ANNOT v a)"
  "⋀x. la_op (UNSKEL x) = la_op x"
  "⋀f p. SKEL (case_prod$f$p) ≡ la_seq (la_op p) (SKEL f)"
  "⋀fn fc l. SKEL (case_list$fn$fc$l)
    ≡ la_seq (la_op l) (la_choice (SKEL fn) (SKEL fc))"
  "⋀fn fs ov. SKEL (case_option$fn$fs$ov)
    ≡ la_seq (la_op ov) (la_choice (SKEL fn) (SKEL fs))"
  "⋀v f. SKEL (Let$v$f) ≡ la_seq (la_op v) (SKEL f)"
  "⋀b t e. SKEL (If$b$t$e) ≡ la_seq (la_op b) (la_choice (SKEL t) (SKEL
e))"
  by simp_all

end
```

6 Depth-First Search Solver with Forward Constraints

theory *DF_Solver*
imports *"../Automatic_Refinement/Lib/Refine_Lib"*
begin

This solver tries to solve a subgoal by repeatedly applying a tactic, backtracking in a depth-first manner. Apart from normal subgoals, the tactic may also produce constraint subgoals, which pose constraints on terms. These constraints are solved recursively by a special set of rules, unless the term is a schematic variable. In this case, solving is delayed until the schematic variable is instantiated, or until all other constraints are solved.

definition *CONSTRAINT* **where** *[simp]: "CONSTRAINT P x ≡ P x"*
definition *SOLVED* **where** *[simp]: "SOLVED ≡ True"*

lemma *SOLVED_I_eq:*
  "PROP P == (SOLVED ==> PROP P)"
  unfolding *SOLVED_def* **by** *simp*

lemma *is_SOLVED: "SOLVED ==> SOLVED"* .

lemma *SOLVED_I: "SOLVED"* **by** *simp*

lemma *CONSTRAINT_D:*
  assumes *"CONSTRAINT (P::'a => bool) x"*
  shows *"P x"*
  using *assms* **unfolding** *CONSTRAINT_def* **by** *simp*

lemma *CONSTRAINT_I:*
  assumes *"P x"*
  shows *"CONSTRAINT (P::'a => bool) x"*
  using *assms* **unfolding** *CONSTRAINT_def* **by** *simp*

lemma *is_CONSTRAINT_rl: "CONSTRAINT P x ⟹ CONSTRAINT P x"* .

ML *{**
```
  signature DF_SOLVER = sig
    val add_constraint_rule: thm -> Context.generic -> Context.generic
    val del_constraint_rule: thm -> Context.generic -> Context.generic
    val get_constraint_rules: Proof.context -> thm list

    val add_forced_constraint_rule: thm -> Context.generic -> Context.generic
    val del_forced_constraint_rule: thm -> Context.generic -> Context.generic
    val get_forced_constraint_rules: Proof.context -> thm list

    val check_constraints_tac: Proof.context -> tactic
    val constraint_tac: Proof.context -> tactic'
```

```
  val force_constraints_tac: Proof.context -> tactic
  val force_constraint_tac: Proof.context -> tactic'

  val defer_constraints: tactic' -> tactic'

  val DF_SOLVE_FWD_C: bool -> tactic' -> Proof.context -> tactic'

  val is_constraint_tac: tactic'

  val setup: theory -> theory

end

structure DF_Solver :DF_SOLVER = struct
  local
    fun prepare_constraint_conv ctxt = let
      open Conv
      fun CONSTRAINT_conv ct = case term_of ct of
        @{mpat "Trueprop (_ _)"} =>
          HOLogic.Trueprop_conv
            (rewr_conv @{thm CONSTRAINT_def[symmetric]}) ct
      | _ => raise CTERM ("CONSTRAINT_conv", [ct])

      fun rec_conv ctxt ct = (
        CONSTRAINT_conv
        else_conv
        implies_conv (rec_conv ctxt) (rec_conv ctxt)
        else_conv
        forall_conv (rec_conv o #2) ctxt
      ) ct

      (*
      fun add_solved_conv ct = case term_of ct of
        @{mpat "_==>_"} => all_conv ct
      | _ => rewr_conv @{thm SOLVED_I_eq} ct
      *)

    in
      rec_conv ctxt (*then_conv add_solved_conv*)
    end
  in
    structure constraint_rules = Named_Sorted_Thms (
      val name = @{binding constraint_rules}
      val description = "Constraint rules"
      val sort = K I
      fun transform context thm = let
        open Conv
        val ctxt = Context.proof_of context
```

```
      in
        case try (fconv_rule (prepare_constraint_conv ctxt)) thm of
          NONE => raise THM ("Invalid constraint rule",~1,[thm])
        | SOME thm => [thm]
        end
      )

      structure forced_constraint_rules = Named_Sorted_Thms (
        val name = @{binding forced_constraint_rules}
        val description = "Forced Constraint rules"
        val sort = K I
        fun transform context thm = let
          open Conv
          val ctxt = Context.proof_of context
        in
          case try (fconv_rule (prepare_constraint_conv ctxt)) thm of
            NONE => raise THM ("Invalid constraint rule",~1,[thm])
          | SOME thm => [thm]
        end
      )
    end

    val add_constraint_rule = constraint_rules.add_thm
    val del_constraint_rule = constraint_rules.del_thm
    val get_constraint_rules = constraint_rules.get

    val add_forced_constraint_rule = forced_constraint_rules.add_thm
    val del_forced_constraint_rule = forced_constraint_rules.del_thm
    val get_forced_constraint_rules = forced_constraint_rules.get

    fun constraint_tac_aux thms no_ctac i st =
      case Logic.concl_of_goal (prop_of st) i |> Envir.beta_eta_contract
of
        @{mpat "Trueprop (CONSTRAINT _ ?t)"} =>
          if (is_Var (head_of t)) then
            Seq.single st
          else (
              resolve_tac thms THEN_ALL_NEW_FWD constraint_tac_aux thms
(K no_tac)
            ) i st
      | @{mpat "Trueprop SOLVED"} => Seq.single st
      | _ => no_ctac i st

    fun check_constraints_tac ctxt = DETERM (ALLGOALS (
      constraint_tac_aux (constraint_rules.get ctxt) (K all_tac)
    ))

    fun constraint_tac ctxt = DETERM o (
      constraint_tac_aux (constraint_rules.get ctxt) (K all_tac)
```

```
  )

  fun force_constraint_tac_aux ctxt no_ctac i st =
    case Logic.concl_of_goal (prop_of st) i |> Envir.beta_eta_contract
of
      @{mpat "Trueprop (CONSTRAINT _ ?t)"} =>
        (
          (
            if (is_Var (head_of t)) then
              resolve_tac (forced_constraint_rules.get ctxt)
            else
              constraint_tac_aux (constraint_rules.get ctxt) (K no_tac)
          ) THEN_ALL_NEW_FWD force_constraint_tac_aux ctxt (K no_tac)
        ) i st
    | @{mpat "Trueprop SOLVED"} => Seq.single st
    | _ => no_ctac i st

  fun force_constraint_tac ctxt =
    force_constraint_tac_aux ctxt (K all_tac)

  fun force_constraints_tac ctxt = ALLGOALS (force_constraint_tac ctxt)

  fun is_CONSTRAINT_goal t = case Logic.strip_assums_concl t of
    @{mpat "Trueprop (CONSTRAINT _ _)"} => true
  | _ => false

  (* Defer constraints produced by tac *)
  local
    fun dc_int l u st =
      if l>u then
        Seq.single st
      else if is_CONSTRAINT_goal (Logic.get_goal (prop_of st) l) then
        (defer_tac l THEN dc_int l (u-1)) st
      else
        dc_int (l+1) u st

  in
    fun defer_constraints tac i st = (
      tac i THEN
      (fn st' => dc_int i (i + nprems_of st' - nprems_of st) st')
    ) st

  end


  local
    fun c_nprems_of st = prems_of st
      |> filter (not o is_CONSTRAINT_goal)
      |> length
```

```
      (* Apply tactic to subgoals in interval, in a forward manner,
         skipping over emerging subgoals *)
      fun INTERVAL_FWD tac l u st =
        if l>u then all_tac st
        else (tac l THEN (fn st' => let
            val ofs = c_nprems_of st' - c_nprems_of st;
          in
            if ofs < ~1 then raise THM (
              "INTERVAL_FWD: Tac solved more than one goal",~1,[st,st'])
            else INTERVAL_FWD tac (l+1+ofs) (u+ofs) st'
          end)) st;

      (* Apply tac2 to all subgoals emerged from tac1, in forward manner.
*)
      fun (tac1 THEN_ALL_NEW_FWD tac2) i st =
        (tac1 i
          THEN (fn st' =>
            INTERVAL_FWD tac2 i (i+c_nprems_of st'-c_nprems_of st) st')
        ) st;

    in

      fun DF_SOLVE_FWD_C dbg tac ctxt = let
        val stuck_list_ref = Unsynchronized.ref []

        fun stuck_tac _ st = if dbg then (
          stuck_list_ref := st :: !stuck_list_ref;
          Seq.empty
        ) else Seq.empty

        fun tac' i =  defer_constraints tac i THEN check_constraints_tac
ctxt

        fun is_CONSTRAINT i st = case Logic.concl_of_goal (prop_of st)
i of
          @{mpat "Trueprop (CONSTRAINT _ _)"} => Seq.single st
        | _ => Seq.empty

        fun rec_tac i st = (
            rtac @{thm SOLVED_I} ORELSE'
            is_CONSTRAINT ORELSE'
            (tac' THEN_ELSE_COMB' (op THEN_ALL_NEW_FWD, rec_tac, stuck_tac))

            (* (tac' THEN_ALL_NEW_FWD (rec_tac))
            ORELSE' stuck_tac *)
          ) i st

        fun fail_tac _ _ = if dbg then
```

```
          Seq.of_list (rev (!stuck_list_ref))
        else Seq.empty
      in
        (rec_tac ORELSE' fail_tac) THEN' (K (ALLGOALS (TRY o (force_constraint_tac
ctxt))))
      end

    end

    val is_constraint_tac = rtac @{thm is_CONSTRAINT_rl}

    val setup =
         constraint_rules.setup
      #> forced_constraint_rules.setup
  end

*}

setup DF_Solver.setup

method_setup trace_constraints = {* Scan.succeed (fn ctxt => SIMPLE_METHOD
(
  fn st => let
    fun is_CONSTRAINT_goal t = case Logic.strip_assums_concl t of
      @{mpat "Trueprop (CONSTRAINT _ _)"} => true
    | _ => false

    val cgoals = prems_of st
      |> filter is_CONSTRAINT_goal


    val _ = case cgoals of
      [] => tracing "No constraints"
    | _ => ( cgoals
      |> map (Syntax.pretty_term ctxt)
      |> Pretty.fbreaks
      |> Pretty.block
      |> Pretty.string_of |> tracing )

  in
    Seq.single st
  end
  ))
  *}

end
```

7 Frame Inference

theory *Sepref_Frame*
imports *Sepref_Basic*
begin

In this theory, we provide a specific frame inference tactic for Sepref.

The first tactic, *frame_tac*, is a standard frame inference tactic, based on the assumption that only *hn_ctxt*-assertions need to be matched.

The second tactic, *merge_tac*, resolves entailments of the form *F1* $\vee_A$ *F2* $\Longrightarrow_A$ *?F* that occur during translation of if and case statements. It synthesizes a new frame ?F, where refinements of variables with equal refinements in *F1* and *F2* are preserved, and the others are set to *hn_invalid*.

lemma *frame_thms:*
  "P $\Longrightarrow_A$ *P"*
  "hn_ctxt R x y $\Longrightarrow_A$ *hn_invalid x y"*

  "P$\Longrightarrow_A$*P'* $\Longrightarrow$ *F*$\Longrightarrow_A$*F'* $\Longrightarrow$ *F*P* $\Longrightarrow_A$ *F'*P'"*

  "P$\Longrightarrow_A$*P'* $\Longrightarrow$ *emp*$\Longrightarrow_A$*F'* $\Longrightarrow$ *P*$\Longrightarrow_A$*F'*P'"*
  apply *(blast intro: ent_refl ent_star_mono)*
  apply *(simp add: hn_ctxt_def)*
  apply *(erule (1) ent_star_mono)*
  by *(metis assn_one_left ent_star_mono)*

lemma *frame_ctxt_dischargeI:*
  "R = R' $\Longrightarrow$ *hn_ctxt R x y* $\Longrightarrow_A$ *hn_ctxt R' x y"*
  by *simp*

lemma *hn_merge0:*
  "emp $\vee_A$ *emp* $\Longrightarrow_A$ *emp"*
  by *simp*

lemma *hn_merge1:*
  "hn_ctxt R x x' $\vee_A$ *hn_ctxt R x x'* $\Longrightarrow_A$ *hn_ctxt R x x'"*
  "⟦ Fl $\vee_A$ *Fr* $\Longrightarrow_A$ *F ⟧* $\Longrightarrow$
    *Fl * hn_ctxt R x x'* $\vee_A$ *Fr * hn_ctxt R x x'* $\Longrightarrow_A$ *F * hn_ctxt R x x'"*
  apply *(rule ent_disjE)*
  apply *(rule ent_refl)*
  apply *(rule ent_refl)*

  apply *(rule ent_disjE)*
  apply *(rule ent_star_mono[OF _ ent_refl])*
  apply *(erule ent_disjI1)*
  apply *(rule ent_star_mono[OF _ ent_refl])*
  apply *(erule ent_disjI2)*
  done

```
lemma hn_merge2:
  "hn_ctxt R1 x x' ∨_A hn_ctxt R2 x x' ⟹_A hn_invalid x x'"
  "⟦ Fl ∨_A Fr ⟹_A F ⟧ ⟹
    Fl * hn_ctxt R1 x x' ∨_A Fr * hn_ctxt R2 x x' ⟹_A F * hn_invalid
x x'"
  apply (rule ent_disjE)
  apply (simp add: hn_ctxt_def pure_def)
  apply (simp add: hn_ctxt_def pure_def)

  apply (rule ent_disjE)
  apply (rule ent_star_mono)
  apply (erule ent_disjI1)
  apply (simp add: hn_ctxt_def pure_def)
  apply (rule ent_star_mono)
  apply (erule ent_disjI2)
  apply (simp add: hn_ctxt_def pure_def)
  done

lemmas hn_merge = hn_merge0 hn_merge1 hn_merge2

lemma is_merge: "P1∨_AP2⟹_AP ⟹ P1∨_AP2⟹_AP" .
ML {*
signature SEPREF_FRAME = sig
  (* Check if subgoal is a frame obligation *)
  val is_frame : term -> bool
  (* Check if subgoal is a merge obligation *)
  val is_merge : term -> bool
  (* Perform frame inference *)
  val frame_tac : Proof.context -> int -> tactic
  (* Perform merging *)
  val merge_tac : Proof.context -> int -> tactic
  (* Reorder frame, used for debugging *)
  val prepare_frame_tac : Proof.context -> int -> tactic

  val add_normrel_eq : thm -> Context.generic -> Context.generic
  val del_normrel_eq : thm -> Context.generic -> Context.generic
  val get_normrel_eqs : Proof.context -> thm list

  val setup: theory -> theory
end


structure Sepref_Frame : SEPREF_FRAME = struct

  structure normrel_eqs = Named_Thms (
    val name = @{binding sepref_normrel_eqs}
    val description = "Equations to normalize relations before frame matching"
  )
```

```
val add_normrel_eq = normrel_eqs.add_thm
val del_normrel_eq = normrel_eqs.del_thm
val get_normrel_eqs = normrel_eqs.get

local
  open Sepref_Basic Refine_Util Conv

  fun assn_ord p = case pairself dest_hn_ctxt_opt p of
      (NONE,NONE) => EQUAL
    | (SOME _, NONE) => LESS
    | (NONE, SOME _) => GREATER
    | (SOME (_,a,_), SOME (_,a',_)) => Term_Ord.fast_term_ord (a,a')

in
  fun reorder_ctxt_conv ctxt ct = let
    val cert = cterm_of (theory_of_cterm ct)

    val new_ct = term_of ct
      |> strip_star
      |> sort assn_ord
      |> list_star
      |> cert

    val thm = Goal.prove_internal ctxt [] (mk_cequals (ct,new_ct))
      (fn _ => simp_tac
        (put_simpset HOL_basic_ss ctxt addsimps @{thms star_aci}) 1)

  in
    thm
  end

  fun prepare_fi_conv ctxt ct = case term_of ct of
    @{mpat "?P ⟹⇩A ?Q"} => let
      val cert = cterm_of (theory_of_cterm ct)

      (* Build table from abs-vars to ctxt *)
      val (Qm, Qum) = strip_star Q |> List.partition is_hn_ctxt
      val Qtab = (
        Qm |> map (fn x => (#2 (dest_hn_ctxt x),(NONE,x)))
        |> Termtab.make
      ) handle
          e as (Termtab.DUP _) => (
            tracing ("Dup heap: " ^ PolyML.makestring ct); reraise e)

      (* Go over entries in P and try to find a partner *)
      val (Qtab,Pum) = fold (fn a => fn (Qtab,Pum) =>
        case dest_hn_ctxt_opt a of
          NONE => (Qtab,a::Pum)
```

```
          | SOME (_,p,_) => ( case Termtab.lookup Qtab p of
              SOME (NONE,tg) => (Termtab.update (p,(SOME a,tg)) Qtab,
Pum)
            | _ => (Qtab,a::Pum)
            )
        ) (strip_star P) (Qtab,[])

        (* Read out information from Qtab *)
        val (pairs,Qum2) = Termtab.dest Qtab |> map #2
          |> List.partition (is_some o #1)
          |> apfst (map (apfst the))
          |> apsnd (map #2)

        (* Build reordered terms *)
        val P' = map fst pairs @ Pum |> list_star
        val Q' = map snd pairs @ Qum2 @ Qum |> list_star

        val new_ct = mk_entails (P',Q') |> cert

        (*
          val _ = pairs |> map (pairself cert) |> PolyML.makestring |>
tracing
        *)

        val thm = Goal.prove_internal ctxt [] (mk_cequals (ct,new_ct))

          (fn _ => simp_tac
            (put_simpset HOL_basic_ss ctxt addsimps @{thms star_aci})
1)

      in
        thm
      end
    | _ => no_conv ct

  end

  fun is_merge @{mpat "Trueprop (_ ∨A _ ⟹A _)"} = true | is_merge _
= false
  fun
    is_frame @{mpat "Trueprop (?P ⟹A _)"} = let
      open Sepref_Basic
      val Ps = strip_star P

      fun is_atomic (Const (_,@{typ "assn⇒assn⇒assn"})$_$_) = false
        | is_atomic _ = true

    in
      forall is_atomic Ps
```

```
    end
  | is_frame _ = false

  fun prepare_frame_tac ctxt = let
    open Refine_Util Conv
    val frame_ss = put_simpset HOL_basic_ss ctxt addsimps
      @{thms mult_1_right[where 'a=assn] mult_1_left[where 'a=assn]}
  in
    CONVERSION Thm.eta_conversion THEN'
    CONCL_COND' is_frame THEN'
    simp_tac frame_ss THEN'
    CONVERSION (HOL_concl_conv (fn _ => prepare_fi_conv ctxt) ctxt)
  end

  fun frame_ctxt_discharge_tac ctxt = let
    val ss = put_simpset HOL_basic_ss ctxt addsimps normrel_eqs.get ctxt
  in
    rtac @{thm frame_ctxt_dischargeI}
    THEN' SOLVED' (full_simp_tac ss)
  end

  fun frame_tac ctxt = let
    open Refine_Util Conv
    val frame_thms = @{thms frame_thms}
  in
    prepare_frame_tac ctxt THEN'
    TRY o REPEAT_ALL_NEW (DETERM o
      ( resolve_tac frame_thms
        ORELSE' frame_ctxt_discharge_tac ctxt))
  end

  fun merge_tac ctxt = let
    open Refine_Util Conv
    val merge_conv = arg1_conv (binop_conv (reorder_ctxt_conv ctxt))
    val merge_thms = @{thms hn_merge}
  in
    CONVERSION Thm.eta_conversion THEN'
    CONCL_COND' is_merge THEN'
    simp_tac (put_simpset HOL_basic_ss ctxt addsimps @{thms star_aci})
THEN'
    CONVERSION (HOL_concl_conv (fn _ => merge_conv) ctxt) THEN'
    TRY o REPEAT_ALL_NEW (resolve_tac merge_thms)
  end

  val setup = normrel_eqs.setup
end
*}

setup Sepref_Frame.setup
```

end

8 Hack: The monotonicity prover of the partial function package

theory *Pf_Mono_Prover*
imports *"../Separation_Logic_Imperative_HOL/Sep_Main"*
begin

The partial function package comes with a monotonicity prover. However, it does not export it. This has the unfortunate effect that it cannot be used from external packages, e.g., the automatic refinement tool.

This theory duplicates the monotonicity prover,

ML *{**

```
  structure Pf_Mono_Prover = struct
    structure Mono_Rules = Named_Thms
    (
      val name = @{binding partial_function_mono};
      val description = "monotonicity rules for partial function definitions";
    );

(*** Automated monotonicity proofs ***)

fun strip_cases ctac = ctac #> Seq.map snd;

(*rewrite conclusion with k-th assumtion*)
fun rewrite_with_asm_tac ctxt k =
  Subgoal.FOCUS (fn {context = ctxt', prems, ...} =>
    Local_Defs.unfold_tac ctxt' [nth prems k]) ctxt;

fun dest_case thy t =
  case strip_comb t of
    (Const (case_comb, _), args) =>
      (case Datatype.info_of_case thy case_comb of
         NONE => NONE
       | SOME {case_rewrites, ...} =>
           let
             val lhs = prop_of (hd case_rewrites)
               |> HOLogic.dest_Trueprop |> HOLogic.dest_eq |> fst;
             val arity = length (snd (strip_comb lhs));
             val conv = funpow (length args - arity) Conv.fun_conv
               (Conv.rewrs_conv (map mk_meta_eq case_rewrites));
           in
             SOME (nth args (arity - 1), conv)
           end)
  | _ => NONE;
```

```
(*split on case expressions*)
val split_cases_tac = Subgoal.FOCUS_PARAMS (fn {context=ctxt, ...} =>
  SUBGOAL (fn (t, i) => case t of
    _ $ (_ $ Abs (_, _, body)) =>
      (case dest_case (Proof_Context.theory_of ctxt) body of
         NONE => no_tac
       | SOME (arg, conv) =>
           let open Conv in
              if Term.is_open arg then no_tac
              else ((DETERM o strip_cases o Induct.cases_tac ctxt false
[[SOME arg]] NONE [])
                THEN_ALL_NEW (rewrite_with_asm_tac ctxt 0)
                THEN_ALL_NEW etac @{thm thin_rl}
                THEN_ALL_NEW (CONVERSION
                  (params_conv ~1 (fn ctxt' =>
                    arg_conv (arg_conv (abs_conv (K conv) ctxt'))) ctxt)))
i
           end)
  | _ => no_tac) 1);

(*monotonicity proof: apply rules + split case expressions*)
fun mono_tac ctxt =
  K (Local_Defs.unfold_tac ctxt [@{thm curry_def}])
  THEN' (TRY o REPEAT_ALL_NEW
   (resolve_tac (Mono_Rules.get ctxt)
     ORELSE' split_cases_tac ctxt));

  end
  *}


setup {* Pf_Mono_Prover.Mono_Rules.setup *}
declare Partial_Function.partial_function_mono[partial_function_mono]

end
```

9 Translation

```
theory Sepref_Translate
imports Sepref_Lin_Ana DF_Solver Sepref_Frame Pf_Mono_Prover
begin


lemma bind_ASSERT_eq_if: "do { ASSERT Φ; m } = (if Φ then m else FAIL)"
  by auto
```

This theory defines the translation phase.

The main functionality of the translation phase is to apply refinement rules. Thereby, the linearity information is exploited to create copies of parameters

that are still required, but would be destroyed by a synthesized operation. These *frame-based* rules are in the named theorem collection `sepref_fr_rules`, and the collection `sepref_copy_rules` contains rules to handle copying of parameters.

Apart from the frame-based rules described above, there is also a set of rules for combinators, in the collection `sepref_comb_rules`, where no automatic copying of parameters is applied.

Moreover, this theory contains

- A setup for the basic monad combinators and recursion.
- A tool to import parametricity theorems.
- Some setup to identify pure refinement relations, i.e., those not involving the heap.
- A preprocessor that identifies parameters in refinement goals, and flags them with a special tag, that allows their correct handling.

9.1 Basic Translation Tool

definition *COPY* — Copy operation
  where *[simp]: "COPY ≡ RETURN"*

lemma *tagged_nres_monad1: "bind$(RETURN$x)$(λ2x. f x) = f x"* **by** *simp*

The PREPARED-tag is used internally, to flag a refinement goal with the index of the refinement rule to be used

definition *PREPARED_TAG :: "'a => nat => 'a"*
  where *[simp]: "PREPARED_TAG x i == x"*
lemma *PREPARED_TAG_I:*
  "hn_refine Γ c Γ' R a ⟹ hn_refine Γ c Γ' R (PREPARED_TAG a i)"
  by *simp*

lemmas *prepare_refine_simps = tagged_nres_monad1 COPY_def LIN_ANNOT_def*

  PREPARED_TAG_def

ML *{**
structure Sepref_Translate = struct

  structure sepref_fr_rules = Named_Thms (
    val name = @{binding "sepref_fr_rules"}
    val description = "Sepref: Frame-based rules"
)

```
structure sepref_comb_rules = Named_Thms (
  val name = @{binding "sepref_comb_rules"}
  val description = "Sepref: Combinator rules"
)

structure sepref_copy_rules = Named_Thms (
  val name = @{binding "sepref_copy_rules"}
  val description = "Sepref: Copy rules"
)


local
  open Autoref_Tagging Sepref_Basic
  fun dest_arg (Var (x,_)) = x
    | dest_arg trm = raise TERM("Argument must be variable",[trm])

  fun is_valid_head (Const _) = true
    | is_valid_head (Free _) = true
    | is_valid_head @{mpat "PR_CONST _"} = true
    | is_valid_head _ = false

  fun
    dest_opr @{mpat "RETURN$?f"} = dest_opr f
  | dest_opr t = let
      val (f,args) = strip_app t
      val _ = is_valid_head f orelse
        raise TERM("get_args: Expected constant head",[t])

    in (f,map dest_arg args) end

  fun valid_pair @{mpat "hn_invalid (mpaq_STRUCT(mpaq_Var ?x _)) _"}
= (x,NONE)
    | valid_pair @{mpat "hn_ctxt ?R (mpaq_STRUCT(mpaq_Var ?x _)) _"}
= (x,SOME R)
    | valid_pair t = raise TERM("Invalid assertion in heap",[t])

  fun is_emp @{mpat emp} = true | is_emp _ = false

in
  (* Given a frame theorem, return the constant
    and a list of refinement relations for the arguments, NONE if the
    argument is not preserved *)
  fun
    analyze_args thm = (
      case concl_of thm of
        @{mpat "Trueprop (hn_refine ?G _ ?G' _ ?a)"} => let
          val in_args = (
               strip_star G
            |> filter (not o is_emp)
```

```
            |> map valid_pair
            |> Vartab.make
          ) handle
              Vartab.DUP _ => raise THM ("analyze_args: Dup in-args",~1,[thm])

          val out_args = (
               strip_star G'
            |> filter (not o is_emp)
            |> map valid_pair
            |> Vartab.make
          ) handle
              Vartab.DUP _ => raise THM ("analyze_args: Dup out-args",~1,[thm])

          val (f,formal_args) = dest_opr a

          (* Check that no parameters are dropped or invented *)
          val _ = Vartab.forall (fn (x,_) => Vartab.defined out_args
x) in_args
            orelse raise THM ("analyze_args: Dropped parameters",~1,[thm])
          val _ = Vartab.forall (fn (x,_) => Vartab.defined in_args
x) out_args
            orelse raise THM ("analyze_args: Invanted parameters",~1,[thm])

          (* Check that precisely the parameters are present *)
          val _ = forall (fn x => Vartab.defined in_args x) formal_args
            orelse raise THM ("analyze_args: Missing parameters",~1,[thm])
          val _ = Vartab.forall (fn (x,_) => member op= formal_args
x) in_args
            orelse raise THM ("analyze_args: Extra parameters",~1,[thm])

          val preserved = map (the o Vartab.lookup out_args) formal_args

        in
          (f,preserved)
        end
      | _ => raise THM("Invalid hn_refine theorem",~1,[thm])
    )

  end

  local
    open Autoref_Tagging Sepref_Basic

    fun is_valid_head (Const _) = true
      | is_valid_head (Free _) = true
      | is_valid_head @{mpat "PR_CONST _"} = true
      | is_valid_head _ = false

    fun dest_arg @{mpat "(?x AS$_s$ (mpaq_Free _ _))$^L$"} = (x,true)
```

```
    | dest_arg @{mpat "(?x AS_s (mpaq_Free _ _))^N"} = (x,false)
    | dest_arg t = raise TERM("Malformed argument",[t])

  fun
    dest_opr (t as @{mpat "RETURN$_"}) = let
      val (f,c) = dest_APP_argc t
      val (res,c') = dest_opr f
    in
      (res,c o c')
    end
  | dest_opr t = let
      val ((f,args),c) = strip_APPc t
      val _ = is_valid_head f orelse
        raise TERM("get_args: Expected constant head",[t])
    in ((f,map dest_arg args),c) end


in
  (* Analyze the arguments in the actual refinement goal.
    Return the head constant, a list of
      argument × linear × refinement-relation option
    and a function to reconstruct the term.
  *)
  fun
    analyze_actual_args @{mpat "hn_refine ?G _ _ _ ?a"} = let
      val ((f,args),c) = dest_opr a

      fun dest_hn_ctxt @{mpat "hn_ctxt ?R ?a _"} = SOME (a,R)
        | dest_hn_ctxt _ = NONE

      val on_heap =
        strip_star G
        |> map_filter dest_hn_ctxt
        |> Termtab.make

      val args = map (fn (x,l) => (x,l,Termtab.lookup on_heap x)) args
    in
      ((f,args),c)
    end
  | analyze_actual_args t = raise TERM("No hn-refine subgoal",[t])
end

local
  open Conv

  fun lambda2_name n t = Term.lambda_name n @{mk_term "PROTECT2 ?t DUMMY"}

  fun prepare_refine thy ((f,largs),mk_a) (i,(f',is_pres)) = let
    val _ = Pattern.matches thy (f',f) orelse raise TERM("No match",[f',f])
```

```
      (* Quick frame check *)
      fun check_rel (_,_,SOME R) (SOME R') = Term.could_unify (R,R')
        | check_rel _ NONE = true (* Strange case, should not happen!
*)
        | check_rel _ _  = false

      val _ = forall2 check_rel largs is_pres
        orelse raise TERM("No frame match",[f',f])

      val args_nc
        = map2 (fn (x,l,_) => fn r => (x,not l andalso is_none r))
          largs is_pres

      (* Aliasing check: Avoid aliasing due to duplicate arguments *)
      fun dups [] = ([],Termtab.empty)
        | dups ((x,_,_)::l) = let
            val (l,tab) = dups l
            val d = Termtab.defined tab x
            val tab = Termtab.update (x,()) tab
          in
            (d::l, tab)
          end

      val is_dup = #1 (dups largs)

      val args_nc = map2 (fn (x,nc) => fn d => (x,nc orelse d)) args_nc
is_dup

      fun prep ((x,c)::l) args i =
        if c then let
          val name = "v" ^ string_of_int i
          val fv = Free ("__prep__"^name,fastype_of x)
          val r = prep l (fv::args) (i+1)
            |> lambda2_name (name,fv)
        in
          @{mk_term "bind$(COPY$?x)$?r"}
        end
        else
          prep l (x::args) (i+1)
      | prep [] args _ = let
          val i = HOLogic.mk_number @{typ nat} i
          val a = mk_a (f,rev args)
        in @{mk_term "PREPARED_TAG ?a ?i"} end

    in
      prep args_nc [] 1
    end
  in
```

```
    (* Try to prepare refinement with the specified index×theorem pair
*)
    fun prepare_refine_conv (i,thm) ctxt ct = let
      val thy = Proof_Context.theory_of ctxt
      val aa = analyze_actual_args (term_of ct)
      val ta = analyze_args thm
      val a' = prepare_refine thy aa (i,ta)
      val tac =
        ALLGOALS (simp_tac
          (put_simpset HOL_basic_ss ctxt addsimps @{thms prepare_refine_simps}))
      open Sepref_Basic
    in
      hn_refine_conv_a (Refine_Util.f_tac_conv ctxt (K a') tac) ct
    end

  end

  (* Refine with the specified theorem *)
  fun try_refine_tac (i,thm) ctxt =
    CONVERSION Thm.eta_conversion THEN'
    CONVERSION (Refine_Util.HOL_concl_conv (prepare_refine_conv (i,thm))
ctxt)
    THEN'
    simp_tac (put_simpset HOL_basic_ss ctxt addsimps @{thms LIN_ANNOT_def})

    (*THEN'
    rtac @{thm hn_refine_frame} THEN'
    rtac thm THEN'
    SOLVED' (DETERM o Sepref_Frame.frame_tac ctxt)*)

  (* Refine with all matching theorems, allow backtracking *)
  fun refine_fr_tac ctxt = let
    val fr_rules = sepref_fr_rules.get ctxt
      |> tag_list 0
    val tacs = map (fn ixthm => try_refine_tac ixthm ctxt) fr_rules
  in
    APPEND_LIST' tacs
  end


  local
    (* Combine rule with frame-thm,
      move frame-premise before first non-PREFER premise *)
    fun prep_fr_rule thm = let
      val thm = thm RS @{thm hn_refine_frame}
      val prems = prems_of thm
      val fix = find_index (fn @{mpat "Trueprop (PREFER_tag _)"} => false
| _ => true) prems
    in
```

```
      Thm.permute_prems fix ~1 thm
    end

    (* Discharge premises *)
    fun discharge_rprem ctxt =
    DF_Solver.is_constraint_tac  (* Keep constraints *)
    ORELSE' SOLVED' (
    FIRST' [
      resolve_tac @{thms PREFER_tagI DEFER_tagI}
      THEN' CONVERSION (Refine_Util.HOL_concl_conv (K (Id_Op.unprotect_conv
ctxt)) ctxt)
      THEN' Tagged_Solver.solve_tac ctxt
    ,
      DETERM o Sepref_Frame.frame_tac ctxt
    ,
      CONVERSION (Refine_Util.HOL_concl_conv (K (Id_Op.unprotect_conv
ctxt)) ctxt)
      THEN' Tagged_Solver.solve_tac ctxt
    ,
      DETERM o Indep_Vars.indep_tac
    ])

  in
    (* Solve prepared frame *)
    fun prepared_tac ctxt i st =
      case Logic.concl_of_goal (prop_of st) i of
        @{mpat "Trueprop (hn_refine _ _ _ _ (PREPARED_TAG _ ?n))"} =>
let
          val n = #2 (HOLogic.dest_number n)
          val thm = nth (sepref_fr_rules.get ctxt) n |> prep_fr_rule
        in
          rtac @{thm PREPARED_TAG_I} THEN'
          (rtac thm THEN_ALL_NEW_FWD (discharge_rprem ctxt))
        end i st
      | _ => Seq.empty

  end

  (* Single translation step *)
  fun trans_step_tac ctxt = let
    val combinator_rules = sepref_comb_rules.get ctxt

    val rcall_tac =
      rtac @{thm hn_refine_frame[where m="RCALL$D$(x^L)" for D x]}
      THEN' DETERM o rprems_tac ctxt
      THEN' SOLVED' (DETERM o Sepref_Frame.frame_tac ctxt)

    val copy_tac =
      resolve_tac (sepref_copy_rules.get ctxt)
```

```
      THEN' SOLVED' (DETERM o Sepref_Frame.frame_tac ctxt)

    val misc_tac = FIRST' [
      SOLVED' (DETERM o Sepref_Frame.frame_tac ctxt),
      SOLVED' (DETERM o Indep_Vars.indep_tac),
      SOLVED' (DETERM o Sepref_Frame.merge_tac ctxt),
      SOLVED' (Pf_Mono_Prover.mono_tac ctxt),
      SOLVED' (
        CONVERSION (Refine_Util.HOL_concl_conv (K (Id_Op.unprotect_conv
ctxt)) ctxt)
        THEN' (TRY o resolve_tac @{thms PREFER_tagI DEFER_tagI})
        THEN' Tagged_Solver.solve_tac ctxt)
    ]

    val fr_tac = refine_fr_tac ctxt
  in
    (*(K (print_tac "trans_step_tac")) THEN'*)
    (
    FIRST' [
      resolve_tac combinator_rules,
      rcall_tac,
      copy_tac,
      prepared_tac ctxt,
      fr_tac,
      misc_tac
    ] (*THEN' (K (print_tac "yields"))*)
    (*ORELSE' (K (print_tac "FAILED"))*)
    )
  end

  (* Do translation *)
  fun trans_tac ctxt =
    DF_Solver.DF_SOLVE_FWD_C true (trans_step_tac ctxt) ctxt
    THEN_ALL_NEW (TRY o rtac @{thm SOLVED_I})

  (* Single translation step *)
  fun cstep_tac ctxt = IF_EXGOAL (
    DF_Solver.defer_constraints (trans_step_tac ctxt)
    THEN' K (DF_Solver.check_constraints_tac ctxt)
    THEN' (CONVERSION Thm.eta_conversion))

  val setup = I
    #> sepref_fr_rules.setup
    #> sepref_comb_rules.setup
    #> sepref_copy_rules.setup

end
*}
```

setup *Sepref_Translate.setup*

9.1.1 Basic Setup

lemma *hn_pass[sepref_fr_rules]:*
shows *"hn_refine (hn_ctxt P x x') (return x') (hn_invalid x x') P (PASS$x)"*
by *rule (sep_auto simp: hn_ctxt_def)*

lemma *hn_bind[sepref_comb_rules]:*
assumes *D1: "hn_refine Γ m' Γ1 Rh m"*
assumes *D2:*
*"⋀x x'. hn_refine (Γ1 * hn_ctxt Rh x x') (f' x') (Γ2 x x') R (f x)"*
assumes *IMP: "⋀x x'. Γ2 x x' ⟹$_A$ Γ' * hn_ctxt Rx x x'"*
shows *"hn_refine Γ (m'⤜f') Γ' R (bind$m$(λ$_2$x. f x))"*
using *assms*
unfolding *APP_def PROTECT2_def*
by *(rule hnr_bind)*

lemma *hn_RECT'[sepref_comb_rules]:*
assumes *"INDEP Ry" "INDEP Rx" "INDEP Rx'"*
assumes *FR: "P ⟹$_A$ hn_ctxt Rx ax px * F"*
assumes *S: "⋀cf af ax px. ⟦*
*⋀ax px. hn_refine (hn_ctxt Rx ax px * F) (cf px) (hn_ctxt Rx' ax px * F) Ry*
(RCALLaf(ax^L))⟧
*⟹ hn_refine (hn_ctxt Rx ax px * F) (cB cf px) (F' ax px) Ry*
(aB af ax)"
assumes *FR': "⋀ax px. F' ax px ⟹$_A$ hn_ctxt Rx' ax px * F"*
assumes *M: "(⋀x. mono_Heap (λf. cB f x))"*
assumes *PREC[unfolded CONSTRAINT_def]: "CONSTRAINT precise Ry"*
shows *"hn_refine*
*(P) (heap.fixp_fun cB px) (hn_ctxt Rx' ax px * F) Ry*
(RECT$(λ$_2$D x. aB D x)$(ax^L))"
unfolding *APP_def PROTECT2_def LIN_ANNOT_def*
apply *(rule hn_refine_cons_pre[OF FR])*
apply *(rule hnr_RECT)*

apply *(rule hn_refine_cons_post[OF _ FR'])*
apply *(rule S[unfolded RCALL_def APP_def LIN_ANNOT_def])*
apply *assumption*
apply *fact+*
done

lemma *hn_RECT_nl[sepref_comb_rules]:*
assumes *"hn_refine*
*(P) t (hn_ctxt Rx' ax px * F) Ry*
(bind$(COPY$ax)$(λ$_2$ax. RECT$(λ$_2$D x. aB D x)$(ax^L)))"
shows *"hn_refine*

```
    (P) t (hn_ctxt Rx' ax px * F) Ry
        (RECT$(λ2D x. aB D x)$(ax^N))"
  using assms by simp

lemma hn_RCALL_nl[sepref_comb_rules]:
  assumes "hn_refine Γ c Γ' R (bind$(COPY$x)$(λ2x. RCALL$D$(x^L)))"
  shows "hn_refine Γ c Γ' R (RCALL$D$(x^N))"
  using assms by simp

definition "monadic_WHILEIT I b f s ≡ do {
  RECT (λD s. do {
    ASSERT (I s);
    bv ← b s;
    if bv then do {
      s ← f s;
      D s
    } else do {RETURN s}
  }) s
}"

definition "heap_WHILET b f s ≡ do {
  heap.fixp_fun (λD s. do {
    bv ← b s;
    if bv then do {
      s ← f s;
      D s
    } else do {return s}
  }) s
}"

lemma heap_WHILET_unfold[code]: "heap_WHILET b f s =
  do {
    bv ← b s;
    if bv then do {
      s ← f s;
      heap_WHILET b f s
    } else
      return s
  }"
  unfolding heap_WHILET_def
  apply (subst heap.mono_body_fixp)
  apply (tactic {* Pf_Mono_Prover.mono_tac @{context} 1 *})
  apply simp
  done



lemma WHILEIT_to_monadic: "WHILEIT I b f s = monadic_WHILEIT I (λs.
RETURN (b s)) f s"
```

```
  unfolding WHILEIT_def monadic_WHILEIT_def
  unfolding WHILEI_body_def bind_ASSERT_eq_if
  by (simp cong: if_cong)

lemma WHILEIT_pat[def_pat_rules]:
  "WHILEIT$I ≡ UNPROTECT (WHILEIT I)"
  "WHILET ≡ PR_CONST (WHILEIT (λ_. True))"
  by (simp_all add: WHILET_def)

lemma id_WHILEIT[id_rules]:
  "PR_CONST (WHILEIT I) ::ᵢ TYPE(('a ⇒ bool) ⇒ ('a ⇒ 'a nres) ⇒ 'a
⇒ 'a nres)"
  by simp

lemma WHILE_arities[sepref_monadify_arity]:

  "PR_CONST (WHILEIT I) ≡ λ2b f s. SP (PR_CONST (WHILEIT I))$(λ2s. b$s)$(λ2s.
f$s)$s"
  by (simp_all add: WHILET_def)

lemma WHILEIT_comb[sepref_monadify_comb]:
  "PR_CONST (WHILEIT I)$(λ2x. b x)$f$s ≡
    bind$(EVAL$s)$(λ2s.
      SP (PR_CONST (monadic_WHILEIT I))$(λ2x. (EVAL$(b x)))$f$s
    )"
  by (simp_all add: WHILEIT_to_monadic)

lemma [sepref_la_skel]: "SKEL (PR_CONST (monadic_WHILEIT I)$b$f$x) ≡

  la_seq
    (la_op x)
    (la_rec (λD. la_seq (SKEL b) (la_seq (SKEL f) (la_rcall D))))"
  by simp


lemma merge4:
  "⟦Fl ∨A Fr ⟹A F⟧ ⟹ Fl * hn_val R x x' ∨A Fr ⟹A F"
  "⟦Fl ∨A Fr ⟹A F⟧ ⟹ Fl ∨A Fr * hn_val R x x' ⟹A F"
  apply (rule ent_disjE)
  apply (drule ent_disjI1)
  apply (sep_auto simp: hn_ctxt_def pure_def)
  apply (erule ent_disjI2)

  apply (rule ent_disjE)
  apply (erule ent_disjI1)
  apply (drule ent_disjI2)
  apply (sep_auto simp: hn_ctxt_def pure_def)
  done
```

```
lemma hn_monadic_WHILE_aux:
  assumes FR: "P ⟹A Γ * hn_ctxt Rs s' s"
  assumes b_ref: "⋀s s'. hn_refine
    (Γ * hn_ctxt Rs s' s)
    (b s)
    (Γb s' s)
    (pure bool_rel)
    (b' s')"
  assumes b_fr: "⋀s' s. Γb s' s ⟹A Γ * hn_ctxt Rs s' s"

  assumes f_ref: "⋀s' s. hn_refine
    (Γ * hn_ctxt Rs s' s)
    (f s)
    (Γf s' s)
    Rs
    (f' s')"
  assumes f_fr: "⋀s' s. Γf s' s ⟹A Γ * hn_invalid s' s"
  assumes PREC: "precise Rs"
  shows "hn_refine (P) (heap_WHILET b f s) (Γ * hn_invalid s' s) Rs (monadic_WHILEIT
I b' f' s')"
  unfolding monadic_WHILEIT_def heap_WHILET_def
  apply (rule hn_refine_cons_pre[OF FR])
  apply (rule hn_refine_cons_pre[OF _ hnr_RECT])
  apply (subst mult_ac(2)[of Γ]) apply (rule ent_refl)
  apply (rule hnr_ASSERT)
  apply (rule hnr_bind)
  apply (rule hn_refine_cons[OF _ b_ref b_fr])
  apply sep_auto []
  apply (rule hnr_If)
  apply sep_auto []
  apply (rule hnr_bind)
  apply (rule hn_refine_cons[OF _ f_ref f_fr])
  apply (sep_auto simp: hn_ctxt_def pure_def) []
  apply (rule hn_refine_frame)
  apply rprems
  apply (tactic {* Sepref_Frame.frame_tac @{context} 1*})
  apply sep_auto []
  apply (rule hn_refine_frame)
  apply (rule hnr_RETURN_pass)
  apply (tactic {* Sepref_Frame.frame_tac @{context} 1*})
  apply (tactic {* Sepref_Frame.merge_tac @{context} 1*})
  apply (rule hn_merge merge4)
  apply sep_auto []
  apply (rule fr_invalidate)
  apply simp
  apply (tactic {* Pf_Mono_Prover.mono_tac @{context} 1 *})
  apply (rule PREC)
  done
```

lemma *hn_monadic_WHILE_lin[sepref_comb_rules]:*
assumes *"INDEP Rs"*
assumes *FR: "P* $\Longrightarrow_A$ *Γ * hn_ctxt Rs s' s"*
assumes *b_ref: "*$\bigwedge$*s s'. hn_refine*
*(Γ * hn_ctxt Rs s' s)*
(b s)
(Γb s' s)
(pure bool_rel)
(b' s')"
assumes *b_fr: "*$\bigwedge$*s' s. TERM (monadic_WHILEIT,''cond'')* $\Longrightarrow$ *Γb s' s*
$\Longrightarrow_A$ *Γ * hn_ctxt Rs s' s"*

assumes *f_ref: "*$\bigwedge$*s' s. hn_refine*
*(Γ * hn_ctxt Rs s' s)*
(f s)
(Γf s' s)
Rs
(f' s')"
assumes *f_fr: "*$\bigwedge$*s' s. TERM (monadic_WHILEIT,''body'')* $\Longrightarrow$ *Γf s' s*
$\Longrightarrow_A$ *Γ * hn_invalid s' s"*
assumes *"CONSTRAINT precise Rs"*
shows *"hn_refine*
P
(heap_WHILET b f s)
*(Γ * hn_invalid s' s)*
Rs
(PR_CONST (monadic_WHILEIT I)$(*$\lambda_2$*s'. b' s')$(λ_2*s'. f' s')$(s'*^{L}*))"*
using *assms(2-)*
unfolding *APP_def PROTECT2_def LIN_ANNOT_def CONSTRAINT_def PR_CONST_def*
by *(rule hn_monadic_WHILE_aux)*

definition *[simp]: "op_ASSERT_bind I m* $\equiv$ *bind (ASSERT I) (*λ*_. m)"*
lemma *pat_ASSERT_bind[def_pat_rules]:*
*"bind$(ASSERT$I)$(*$\lambda_2$*_. m)* $\equiv$ *UNPROTECT (op_ASSERT_bind I)$m"*
by *simp*

term *"PR_CONST (op_ASSERT_bind I)"*
lemma *id_op_ASSERT_bind[id_rules]:*
"PR_CONST (op_ASSERT_bind I) $::_i$ *TYPE('a nres* $\Rightarrow$ *'a nres)"*
by *simp*

lemma *arity_ASSERT_bind[sepref_monadify_arity]:*
"PR_CONST (op_ASSERT_bind I) $\equiv$ λ_2*m. SP (PR_CONST (op_ASSERT_bind I))$m"*
apply *(rule eq_reflection)*
by *auto*

lemma *skel_ASSERT_bind[sepref_la_skel]:*
"SKEL (PR_CONST (op_ASSERT_bind I)$m) = SKEL m"

```
  by simp

lemma hn_ASSERT_bind[sepref_comb_rules]:
  assumes "I ⟹ hn_refine Γ c Γ' R m"
  shows "hn_refine Γ c Γ' R (PR_CONST (op_ASSERT_bind I)$m)"
  using assms
  apply (cases I)
  apply auto
  done
```

9.2 Import of Parametricity Theorems

```
lemma pure_hn_refineI:
  assumes "Q ⟶ (c,a)∈R"
  shows "hn_refine (↑Q) (return c) (↑Q) (pure R) (RETURN a)"
  unfolding hn_refine_def using assms
  by (sep_auto simp: pure_def)

lemma pure_hn_refineI_no_asm:
  assumes "(c,a)∈R"
  shows "hn_refine emp (return c) emp (pure R) (RETURN a)"
  unfolding hn_refine_def using assms
  by (sep_auto simp: pure_def)

lemma import_param_1:
  "(P⟹Q) ≡ Trueprop (P⟶Q)"
  "(P⟶Q⟶R) ⟷ (P∧Q ⟶ R)"
  "(a,c)∈Rel ∧ PREFER_tag P ⟷ PREFER_tag P ∧ (a,c)∈Rel"
  apply (rule, simp+)+
  done

lemma import_param_2:
  "Trueprop (PREFER_tag P ∧ Q ⟶ R) ≡ (PREFER_tag P ⟹ Q⟶R)"
  "Trueprop (DEFER_tag P ∧ Q ⟶ R) ≡ (DEFER_tag P ⟹ Q⟶R)"
  apply (rule, simp+)+
  done

lemma import_param_3:
  "↑(P ∧ Q) = ↑P*↑Q"
  "↑((c,a)∈R) = hn_val R a c"
  by (simp_all add: hn_ctxt_def pure_def)

ML {*
structure Sepref_Import_Param = struct

  structure sepref_import_rewrite = Named_Thms (
    val name = @{binding "sepref_import_rewrite"}
    val description = "Rewrite rules on importing parametricity theorems"
  )
```

```
  fun import ctxt thm = let
    open Sepref_Basic
    val thm = Parametricity.fo_rule thm
      |> Local_Defs.unfold ctxt @{thms import_param_1}
      |> Local_Defs.unfold ctxt @{thms import_param_2}

    val thm = case concl_of thm of
      @{mpat "Trueprop (_ ⟶ _)"} => thm RS @{thm pure_hn_refineI}
    | _ => thm RS @{thm pure_hn_refineI_no_asm}

    val thm = Local_Defs.unfold ctxt @{thms import_param_3} thm
      |> Conv.fconv_rule (hn_refine_concl_conv_a (K (Id_Op.protect_conv
ctxt)) ctxt)

    val thm = Local_Defs.unfold ctxt (sepref_import_rewrite.get ctxt)
thm
  in
    thm
  end

  val import_attr = Scan.succeed (Thm.mixed_attribute (fn (context,thm)
=>
    let
      val thm = import (Context.proof_of context) thm
      val context = Sepref_Translate.sepref_fr_rules.add_thm thm context
    in (context,thm) end
  ))

  val setup = I
    #> sepref_import_rewrite.setup
    #> Attrib.setup @{binding sepref_import_param} import_attr
        "Sepref: Import parametricity rule"

end
*}

setup Sepref_Import_Param.setup
```

9.3 Purity

definition *"is_pure P ≡ ∃P'. ∀x x'. P x x'=↑(P' x x')"*
lemma *is_pureI[intro?]:*
  assumes *"⋀x x'. P x x' = ↑(P' x x')"*
  shows *"is_pure P"*
  using *assms* **unfolding** *is_pure_def* **by** *blast*

lemma *is_pureE:*
  assumes *"is_pure P"*

```
  obtains P' where "⋀x x'. P x x' = ↑(P' x x')"
  using assms unfolding is_pure_def by blast

lemma pure_pure[constraint_rules, simp]: "is_pure (pure P)"
  unfolding pure_def by rule blast
lemma pure_hn_ctxt[constraint_rules, intro!]: "is_pure P ⟹ is_pure
(hn_ctxt P)"
  unfolding hn_ctxt_def[abs_def] .


definition "the_pure P ≡ THE P'. ∀x x'. P x x'=↑((x',x)∈P')"


lemma assn_basic_inequalities[simp, intro!]:
  "true ≠ emp" "emp ≠ true"
  "false ≠ emp" "emp ≠ false"
  "true ≠ false" "false ≠ true"
proof -
  def neh ≡ "(⦇ arrays = undefined, refs=undefined, lim = 1 ⦈, {0::nat})"
  have [simp]: "in_range neh" unfolding neh_def
    by (simp add: in_range.simps)

  have "neh ⊨ true" by simp
  moreover have "¬(neh ⊨ false)" by simp
  moreover have "¬(neh ⊨ emp)" by (simp add: mod_emp neh_def)
  moreover have "h⊥ ⊨ emp" by simp
  moreover have "¬(h⊥ ⊨ false)" by simp
  ultimately show
    "true ≠ emp" "emp ≠ true"
    "false ≠ emp" "emp ≠ false"
    "true ≠ false" "false ≠ true"
    by metis+
qed

lemma pure_assn_eq_conv[simp]: "↑P = ↑Q ⟷ P=Q"
  apply (cases P, simp_all)
  apply (cases Q, simp_all)
  apply (cases Q, simp_all)
  done


lemma the_pure_pure[simp]: "the_pure (pure R) = R"
  unfolding pure_def the_pure_def
  by (rule theI2[where a=R]) auto

lemma is_pure_alt_def: "is_pure R ⟷ (∃Ri. ∀x y. R x y = ↑((y,x)∈Ri))"
  unfolding is_pure_def
  apply auto
  apply (rename_tac P')
```

```
  apply (rule_tac x="{(x,y). P' y x}" in exI)
  apply auto
  done

lemma pure_the_pure[simp]: "is_pure R ⟹ pure (the_pure R) = R"
  unfolding is_pure_alt_def pure_def the_pure_def
  apply (intro ext)
  apply clarsimp
  apply (rename_tac a c Ri)
  apply (rule_tac a=Ri in theI2)
  apply auto
  done

definition "import_rel1 R ≡ λA c ci. ↑(is_pure A ∧ (ci,c)∈⟨the_pure A⟩R)"
definition "import_rel2 R ≡ λA B c ci. ↑(is_pure A ∧ is_pure B ∧ (ci,c)∈⟨the_pure
A, the_pure B⟩R)"

lemma import_rel1_pure_conv: "import_rel1 R (pure A) = pure (⟨A⟩R)"
  unfolding import_rel1_def
  apply simp
  apply (simp add: pure_def)
  done

lemma import_rel2_pure_conv: "import_rel2 R (pure A) (pure B) = pure
(⟨A,B⟩R)"
  unfolding import_rel2_def
  apply simp
  apply (simp add: pure_def)
  done

lemma hn_pure_copy_complete:
  assumes F: "Γ ⟹_A F * hn_ctxt P x x'"
  assumes P: "CONSTRAINT is_pure P"
  shows "hn_refine Γ (return x') (F * hn_ctxt P x x') (λxc xc'. P xc
xc' * ↑(xc = x))
    (COPY$x)"
  apply (rule is_pureE[OF P[simplified]])
  apply (rule hn_refine_frame[OF _ F])
  apply rule
  apply (sep_auto simp: hn_ctxt_def)
  done


lemma hn_pure_copy[sepref_copy_rules]:
  assumes F: "Γ ⟹_A F * hn_ctxt P x x'"
  assumes P: "CONSTRAINT is_pure P"
  shows "hn_refine Γ (return x') (F * hn_ctxt P x x') P
    (COPY$x)"
  apply (rule is_pureE[OF P[simplified]])
```

```
  apply (rule hn_refine_frame[OF _ F])
  apply rule
  apply (sep_auto simp: hn_ctxt_def)
  done


lemma precise_pure[constraint_rules]: "single_valued R ⟹ precise (pure
R)"
  unfolding precise_def pure_def
  by (auto dest: single_valuedD)


lemmas [constraint_rules] = single_valued_Id list_set_rel_sv
```

9.4 Parameters

```
definition PARAM :: "'a ⇒ ('a ⇒ 'b nres) ⇒ 'b nres"
  where [simp]: "PARAM s f == f s"
lemma skel_param[sepref_la_skel]:
  "SKEL (PARAM$s$f) = la_seq (la_op s) (SKEL f)"
  by simp

lemma hn_param_bind[sepref_comb_rules]:
  fixes f :: "'a ⇒ 'b nres"
  assumes F1: "Γ ⟹_A Γ1 * hn_ctxt R x x'"
  assumes R: "⋀x x'. hn_refine
    (Γ1 * hn_ctxt R x x')
    (c x') (Γ2 x x') S (f x)"
  assumes F2: "⋀x x'. Γ2 x x' ⟹_A Γ' * hn_ctxt R' x x'"
  shows "hn_refine Γ (c x') (Γ' * hn_ctxt R' x x') S (PARAM$x$(λ2x. f
x))"
  apply (simp)
  apply (rule hn_refine_cons_pre[OF F1])
  apply (rule hn_refine_cons_post[OF _ F2])
  apply (rule R)
  done


ML {*
  structure Sepref_Param = struct
    fun
      id_param @{mpat "hn_refine ?P ?c ?P' ?R ?a"} = let
        open Sepref_Basic
        val used_params = Term.add_frees a [] |> map Free
          |> Termtab.make_set

        val params = strip_star P
          |> map_filter (dest_hn_ctxt_opt) |> map #2
          |> filter (Termtab.defined used_params)
```

```
          |> rev

        fun abs_p (p as (Free (x,_))) a = let
            val b = Term.lambda_name (x,p) a
            val res = @{mk_term "PARAM ?p ?b"}
          in res end
        | abs_p _ _ =
            error ("id_param: Internal error: expected only frees in params")

        val new_a = fold abs_p params a
      in
       @{mk_term "hn_refine ?P ?c ?P' ?R ?new_a"}
      end
    | id_param t = raise TERM ("id_param",[t])

    fun id_param_conv ctxt = Refine_Util.f_tac_conv ctxt
      (id_param)
      (simp_tac (put_simpset HOL_basic_ss ctxt addsimps @{thms PARAM_def})
1)

  end
*}

end
```

10 Sepref Tool

theory *Sepref_Tool*
imports *Sepref_Translate*
begin

In this theory, we set up the sepref tool.

definition *[simp]: "CNV x y ≡ x=y"*

lemma *ID_init: "⟦ID a a' TYPE('T); hn_refine Γ c Γ' R a'⟧*
⟹ hn_refine Γ c Γ' R a" **by** *simp*

lemma *TRANS_init: "⟦ hn_refine Γ c Γ' R a; CNV c c' ⟧*
⟹ hn_refine Γ c' Γ' R a"
by *simp*

lemma *CNV_I: "CNV x x"* **by** *simp*

ML *{**
```
  structure Sepref = struct
    structure sepref_opt_simps = Named_Thms (
      val name = @{binding sepref_opt_simps}
      val description = "Sepref: Post-Translation optimizations, phase
1"
```

```
  )

  structure sepref_opt_simps2 = Named_Thms (
    val name = @{binding sepref_opt_simps2}
    val description = "Sepref: Post-Translation optimizations, phase
2"
  )

  fun id_tac ctxt =
    rtac @{thm ID_init}
    THEN' CONVERSION Thm.eta_conversion
    THEN' Id_Op.id_tac Id_Op.Normal ctxt

  fun id_param_tac ctxt = CONVERSION (Refine_Util.HOL_concl_conv
    (K (Sepref_Param.id_param_conv ctxt)) ctxt)

  fun monadify_tac ctxt = Sepref_Monadify.monadify_tac ctxt

  fun lin_ana_tac ctxt = Sepref_Lin_Ana.lin_ana_tac ctxt

  fun trans_tac ctxt =
    rtac @{thm TRANS_init}
    THEN' Sepref_Translate.trans_tac ctxt

  fun opt_tac ctxt = let
    val opt1_ss = put_simpset HOL_basic_ss ctxt
      addsimps sepref_opt_simps.get ctxt
      addsimprocs [@{simproc "HOL.let_simp"}]
    |> Simplifier.add_cong @{thm SP_cong}
    val unsp_ss = put_simpset HOL_basic_ss ctxt addsimps @{thms SP_def}

    val opt2_ss = put_simpset HOL_basic_ss ctxt
      addsimps sepref_opt_simps2.get ctxt
      addsimprocs [@{simproc "HOL.let_simp"}]

  in
    simp_tac opt1_ss THEN' simp_tac unsp_ss THEN'
    simp_tac opt2_ss THEN' simp_tac unsp_ss
  end

  fun PHASES [] _ = K all_tac
    | PHASES (tac::tacs) ctxt =
        IF_EXGOAL (tac ctxt)
        THEN_ELSE' (PHASES tacs ctxt, K all_tac)

  fun sepref_tac ctxt =
    PHASES
      [ id_param_tac,
        id_tac,
```

```
          monadify_tac,
          lin_ana_tac,
          trans_tac,
          opt_tac,
          K (CONVERSION Thm.eta_conversion),
          K (rtac @{thm CNV_I})
        ]
        ctxt

    val setup = sepref_opt_simps.setup #> sepref_opt_simps2.setup
  end
*}

setup Sepref.setup

method_setup sepref = ‹Scan.succeed (fn ctxt =>
  SIMPLE_METHOD (DETERM (SOLVED' (IF_EXGOAL (
      Sepref.sepref_tac ctxt
    )) 1)))›
  ‹Automatic refinement to Imperative/HOL›

method_setup sepref_keep = ‹Scan.succeed (fn ctxt =>
  SIMPLE_METHOD (IF_EXGOAL (Sepref.sepref_tac ctxt) 1))›
  ‹Automatic refinement to Imperative/HOL›
```

10.0.1 Debugging Methods

```
ML ‹
  fun SIMPLE_METHOD_NOPARAM' tac = Scan.succeed (fn ctxt => SIMPLE_METHOD'
(tac ctxt))
  fun SIMPLE_METHOD_NOPARAM tac = Scan.succeed (fn ctxt => SIMPLE_METHOD
(tac ctxt))
›
method_setup sepref_dbg_id_param = ‹SIMPLE_METHOD_NOPARAM' Sepref.id_param_tac›
  ‹Sepref debug: Identify parameters phase›
method_setup sepref_dbg_id = ‹SIMPLE_METHOD_NOPARAM' Sepref.id_tac›
  ‹Sepref debug: Identify operations phase›
method_setup sepref_dbg_monadify = ‹SIMPLE_METHOD_NOPARAM' Sepref.monadify_tac›
  ‹Sepref debug: Monadify phase›
method_setup sepref_dbg_lin_ana = ‹SIMPLE_METHOD_NOPARAM' Sepref.lin_ana_tac›
  ‹Sepref debug: Linearity analysis phase›
method_setup sepref_dbg_trans = ‹SIMPLE_METHOD_NOPARAM' Sepref.trans_tac›
  ‹Sepref debug: Translation phase›
method_setup sepref_dbg_opt = ‹SIMPLE_METHOD_NOPARAM' Sepref.opt_tac›
  ‹Sepref debug: Optimization phase›

method_setup sepref_dbg_trans_step = ‹SIMPLE_METHOD_NOPARAM' (Sepref_Translate.cstep_tac)›
  ‹Sepref debug: Translation phase single step›
```

method_setup *sepref_dbg_prepare_frame = ⟨SIMPLE_METHOD_NOPARAM' Sepref_Frame.prepare_frame*
⟨Sepref debug: Prepare frame inference⟩

method_setup *sepref_dbg_frame = ⟨SIMPLE_METHOD_NOPARAM' Sepref_Frame.frame_tac⟩*
⟨Sepref debug: Frame inference⟩

lemmas *[sepref_opt_simps] = return_bind bind_return bind_bind id_def*

We allow the synthesized function to contain tagged function applications. This is important to avoid higher-order unification problems when synthesizing generic algorithms, for example the to-list algorithm for foreach-loops.

lemmas *[sepref_opt_simps] = Autoref_Tagging.APP_def*

Revert case-pulling done by monadify

lemma *case_prod_return_opt[sepref_opt_simps]:*
"case_prod (λa b. return (f a b)) p = return (case_prod f p)"
by *(simp split: prod.split)*

lemma *case_option_return_opt[sepref_opt_simps]:*
"case_option (return fn) (λs. return (fs s)) v = return (case_option fn fs v)"
by *(simp split: option.split)*

lemma *case_list_return[sepref_opt_simps]:*
"case_list (return fn) (λx xs. return (fc x xs)) l = return (case_list fn fc l)"
by *(simp split: list.split)*

lemma *if_return[sepref_opt_simps]:*
"If b (return t) (return e) = return (If b t e)" **by** *simp*

In some cases, pushing in the returns is more convenient

lemma *case_prod_opt2[sepref_opt_simps2]:*
"(λx. return (case x of (a,b) ⇒ f a b))
= (λ(a,b). return (f a b))"
by *auto*

10.1 Setup of Extraction-Tools

declare *[[cd_patterns "hn_refine _ ?f _ _ _"]]*

definition *[simp, code_unfold]: "TRIV_EXTRACTION x ≡ x"*

lemma *TRIV_EXTRACTION_codegen:*
assumes *DEF: "f ≡ TRIV_EXTRACTION B"*
shows *"f = B"*
using *assms* **by** *simp*

```
  lemma TRIV_extraction_cong:
    "TRIV_EXTRACTION x ≡ TRIV_EXTRACTION x"
    by simp

  setup {*
    Refine_Automation.add_extraction "trivial" {
      pattern = term_of @{cpat "TRIV_EXTRACTION _"},
      gen_thm = @{thm TRIV_EXTRACTION_codegen},
      gen_tac = (K (K all_tac))
    }
  *}

  lemma heap_fixp_codegen:
    assumes DEF: "f ≡ heap.fixp_fun cB"
    assumes M: "(⋀x. mono_Heap (λf. cB f x))"
    shows "f x = cB f x"
    unfolding DEF
    apply (rule fun_cong[of _ _ x])
    apply (rule heap.mono_body_fixp)
    apply fact
    done

  setup {*
    Refine_Automation.add_extraction "heap" {
      pattern = term_of @{cpat "heap.fixp_fun _"},
      gen_thm = @{thm heap_fixp_codegen},
      gen_tac = (fn ctxt =>
        (* TODO: This is a bit hacky: We should better handle those stuff
in
          the mono-prover *)
        simp_tac
          (put_simpset HOL_basic_ss ctxt addsimps @{thms TRIV_EXTRACTION_def})

        THEN'
        Pf_Mono_Prover.mono_tac ctxt
      )
    }
  *}

end
theory Sepref_HOL_Bindings
imports Sepref_Tool
begin


lemma (in -) param_map_option[param]: "(map_option, map_option) ∈ (A
→ B) → ⟨A⟩option_rel → ⟨B⟩option_rel"
  apply (intro fun_relI)
```

```
  apply (auto elim!: option_relE dest: fun_relD)
  done


lemma return_refine_prop_return:
  assumes "nofail m"
  assumes "RETURN x ≤ ⇓R m"
  obtains x' where "(x,x')∈R" "RETURN x' ≤ m"
  using assms
  by (auto simp: refine_pw_simps pw_le_iff)
```

10.2 Identity Relations

```
definition "IS_PURE_ID R ≡ R = (λx y. ↑(y=x))"
definition [simp]: "FORCE_PURE_ID R ≡ IS_PURE_ID R"

lemma IS_PURE_ID_I: "(⋀x y. R x y = ↑(y=x)) ⟹ IS_PURE_ID R"
  unfolding IS_PURE_ID_def
  by (auto intro!: ext)

lemma IS_PURE_ID_D: "IS_PURE_ID R ⟹ R=(λx y. ↑(y=x))"
  unfolding IS_PURE_ID_def
  by (auto intro!: ext)

lemma IS_PURE_ID_trigger: "IS_PURE_ID R ⟹ IS_PURE_ID R" by simp
lemma FORCE_PURE_ID_trigger: "IS_PURE_ID R ⟹ FORCE_PURE_ID R" by simp

declaration {* Tagged_Solver.add_triggers
  "Relators.relator_props_solver" @{thms IS_PURE_ID_trigger} *}

declaration {* Tagged_Solver.add_triggers
  "Relators.force_relator_props_solver" @{thms FORCE_PURE_ID_trigger}
*}

lemma IS_PURE_ID_alt_def: "IS_PURE_ID R ⟷ R = pure Id"
  by (auto simp: IS_PURE_ID_def pure_def)

lemma IS_PURE_ID_pure[relator_props]: "R=Id ⟹ IS_PURE_ID (pure R)"
  by (auto simp add: IS_PURE_ID_alt_def)

lemma IS_PURE_ID_pure_Id[solve_relator_props]:
  "IS_PURE_ID (pure Id)"
  by (auto simp add: IS_PURE_ID_def pure_def)



lemma [constraint_rules, forced_constraint_rules]:
  "REL_IS_ID Id"
  "REL_IS_ID R ⟹ IS_PURE_ID (pure R)"
```

by *(simp_all add: IS_PURE_ID_alt_def)*

lemma *[constraint_rules]:*
"REL_IS_ID A ⟹ REL_IS_ID B ⟹ REL_IS_ID (A → B)"
"REL_IS_ID A ⟹ REL_IS_ID B ⟹ REL_IS_ID (A ×$_r$ B)"
"REL_IS_ID A ⟹ REL_IS_ID (⟨A⟩option_rel)"
"REL_IS_ID A ⟹ REL_IS_ID (⟨A⟩list_rel)"
"REL_IS_ID A ⟹ REL_IS_ID B ⟹ REL_IS_ID (⟨A,B⟩sum_rel)"
by *simp_all*

10.3 HOL Combinators

lemma *hn_if[sepref_comb_rules]:*
assumes *P: "Γ ⟹$_A$ Γ1 * hn_val bool_rel a a'"*
assumes *RT: "a ⟹ hn_refine (Γ1 * hn_val bool_rel a a') b' Γ2b R b"*
assumes *RE: "¬a ⟹ hn_refine (Γ1 * hn_val bool_rel a a') c' Γ2c R c"*
assumes *IMP: "TERM If ⟹ Γ2b ∨$_A$ Γ2c ⟹$_A$ Γ'"*
shows *"hn_refine Γ (if a' then b' else c') Γ' R (If$(LIN_ANNOT a A)$b$c)"*
using *P RT RE IMP[OF TERMI]*
unfolding *APP_def PROTECT2_def LIN_ANNOT_def*
by *(rule hnr_If)*

lemma *hn_let[sepref_comb_rules]:*
assumes *P: "Γ ⟹$_A$ Γ1 * hn_ctxt R v v'"*
assumes *R: "⋀x x'. hn_refine (Γ1 * hn_ctxt R x x') (f' x') (Γ' x x') R2 (f x)"*
assumes *F: "⋀x x'. Γ' x x' ⟹$_A$ Γ2 * hn_ctxt R' x x'"*
shows
*"hn_refine Γ (Let v' f') (Γ2 * hn_ctxt R' v v') R2 (Let(v^L)(λ$_2$x. f x))"*
apply *rule*
apply *(rule cons_pre_rule[OF P])*
apply *(vcg)*
apply *(rule cons_post_rule)*
apply *(rule hn_refineD[OF R])*
apply *simp*
apply *(rule ent_frame_fwd[OF F])*
apply *(fr_rot 2)*
apply *(fr_rot_rhs 1)*
apply *(rule fr_refl)*
apply *(rule ent_refl)*
apply *sep_auto*
done

lemma *hn_let_nl[sepref_comb_rules]:*
assumes *"hn_refine Γ t' Γ' R2 (bind$(COPY$v)$(λ$_2$v. Let$(v^L)$f))"*

shows *"hn_refine Γ t' Γ' R2 (Let(v^N)f)"*
using *assms*
by *simp*

10.4 Basic HOL types

lemmas *[sepref_import_param] =*
param_bool
param_nat1
param_int

lemma *hn_nat[sepref_fr_rules]:*
"hn_refine emp (return x) emp (pure Id) (RETURN$(PR_CONST (x::nat)))"
by *rule (sep_auto simp: pure_def hn_ctxt_def)*

lemma *hn_int[sepref_fr_rules]:*
"hn_refine emp (return x) emp (pure Id) (RETURN$(PR_CONST (x::int)))"
by *rule (sep_auto simp: pure_def hn_ctxt_def)*

10.5 Product

definition *hn_prod_aux :: "('a1⇒'c1⇒assn) ⇒ ('a2⇒'c2⇒assn)*
⇒ 'a1'a2 ⇒ 'c1*'c2 ⇒ assn"* **where**
"hn_prod_aux P1 P2 a c ≡ case (a,c) of ((a1,a2),(c1,c2)) ⇒
*P1 a1 c1 * P2 a2 c2"*

lemma *hn_prod_pure_conv: "hn_prod_aux (pure R1) (pure R2) = pure (R1*
×$_r$ R2)"
by *(auto simp: pure_def hn_prod_aux_def intro!: ext)*

lemmas *[sepref_import_rewrite, sepref_normrel_eqs] = hn_prod_pure_conv[symmetric]*

abbreviation *"hn_prod P1 P2 ≡ hn_ctxt (hn_prod_aux P1 P2)"*

lemma *hn_prod_precise[constraint_rules]:*
"precise P1 ⟹ precise P2 ⟹ precise (hn_prod_aux P1 P2)"
apply *rule*
apply *(clarsimp simp: hn_prod_aux_def)*
proof *(rule conjI)*
fix *F F' h as a b a' b' ap bp*
assume *P1: "precise P1"* **and** *P2: "precise P2"*
assume *F: "(h, as) ⊨ P1 a ap * P2 b bp * F ∧$_A$ P1 a' ap * P2 b' bp*
** F'"*

from *F* **have** *"(h, as) ⊨ P1 a ap * (P2 b bp * F) ∧$_A$ P1 a' ap * (P2*
*b' bp * F')"*
by *(simp add: star_aci)*
with *preciseD[OF P1]* **show** *"a=a'"* .
from *F* **have** *"(h, as) ⊨ P2 b bp * (P1 a ap * F) ∧$_A$ P2 b' bp * (P1*
*a' ap * F')"*

```
    by (simp add: star_aci)
  with preciseD[OF P2] show "b=b'" .
qed

lemma pure_prod[constraint_rules]:
  assumes P1: "is_pure P1" and P2: "is_pure P2"
  shows "is_pure (hn_prod_aux P1 P2)"
proof -
  from P1 obtain P1' where P1': "⋀x x'. P1 x x' = ↑(P1' x x')"
    using is_pureE by blast
  from P2 obtain P2' where P2': "⋀x x'. P2 x x' = ↑(P2' x x')"
    using is_pureE by blast

  show ?thesis proof
    fix x x'
    show "hn_prod_aux P1 P2 x x' =
        ↑ (case (x, x') of ((a1, a2), c1, c2) ⇒ P1' a1 c1 ∧ P2' a2 c2)"
      unfolding hn_prod_aux_def
      apply (simp add: P1' P2' split: prod.split)
      done
  qed
qed


lemma hn_case_prod[sepref_comb_rules]:
  assumes P: "Γ⟹ₐΓ1 * hn_prod P1 P2 p' p"
  assumes R: "⋀a1 a2 a1' a2'. ⟦p'=(a1',a2')⟧
    ⟹ hn_refine (Γ1 * hn_ctxt P1 a1' a1 * hn_ctxt P2 a2' a2) (f a1 a2)

          (Γh a1 a1' a2 a2') R (f' a1' a2')"
  assumes M: "⋀a1 a1' a2 a2'. Γh a1 a1' a2 a2'
    ⟹ₐ Γ' * hn_ctxt P1' a1' a1 * hn_ctxt P2' a2' a2"
  shows "hn_refine Γ (case_prod f p) (Γ' * hn_prod P1' P2' p' p)
    R (case_prod$(λ2a b. f' a b)$(p'^L))"
  apply rule
  apply (rule cons_pre_rule[OF P])
  unfolding hn_prod_aux_def hn_ctxt_def
  apply vcg
  apply (simp split: prod.splits)
  apply (rule cons_post_rule)
  apply (simp only: star_assoc[symmetric])
  apply (rule hn_refineD[OF R[unfolded hn_ctxt_def]])
  apply simp_all
  apply solve_entails
  apply clarsimp
  apply assumption

  apply (rule ent_frame_fwd[OF M])
  apply frame_inference
```

```
  apply (simp only: hn_ctxt_def)
  apply solve_entails
  done

lemma hn_case_prod_nl[sepref_comb_rules]:
  assumes "hn_refine Γ t (Γ' * hn_prod P1' P2' p' p)
    R (bind$(COPY$p')$(λ2p'. case_prod$(λ2a b. f' a b)$(p'^L)))"
  shows "hn_refine Γ t (Γ' * hn_prod P1' P2' p' p)
    R (case_prod$(λ2a b. f' a b)$(p'^N))"
  using assms by simp


lemma hn_Pair[sepref_fr_rules]: "hn_refine
  (hn_ctxt P1 x1 x1' * hn_ctxt P2 x2 x2')
  (return (x1',x2'))
  (hn_invalid x1 x1' * hn_invalid x2 x2')
  (hn_prod_aux P1 P2)
  (RETURN$(Pair$x1$x2))"
  unfolding hn_refine_def
  apply (sep_auto simp: hn_ctxt_def hn_prod_aux_def)
  done

lemma IS_PURE_ID_prod[relator_props]: "⟦IS_PURE_ID R1; IS_PURE_ID R2⟧

  ⟹ IS_PURE_ID (hn_prod_aux R1 R2)"
  by (auto simp: hn_ctxt_def hn_prod_aux_def IS_PURE_ID_alt_def pure_def
intro!: ext)
lemmas [constraint_rules] = IS_PURE_ID_prod
```

10.6 Option

```
fun hn_option_aux :: "('a ⇒ 'c ⇒ assn) ⇒ 'a option ⇒ 'c option ⇒
assn" where
  "hn_option_aux P None None = emp"
| "hn_option_aux P (Some a) (Some c) = P a c"
| "hn_option_aux _ _ _ = false"

lemma hn_option_aux_simps[simp]:
  "hn_option_aux P None v' = ↑(v'=None)"
  "hn_option_aux P v None = ↑(v=None)"
  apply (cases v', simp_all)
  apply (cases v, simp_all)
  done

lemma hn_option_pure_conv: "hn_option_aux (pure R) = pure (⟨R⟩option_rel)"
  apply (intro ext)
  apply (rename_tac a c)
  apply (case_tac "(pure R,a,c)" rule: hn_option_aux.cases)
  by (auto simp: pure_def)
```

```
lemmas [sepref_import_rewrite, sepref_normrel_eqs] = hn_option_pure_conv[symmetric]


abbreviation "hn_option P ≡ hn_ctxt (hn_option_aux P)"

lemma hn_option_precise[constraint_rules]:
  assumes "precise P"
  shows "precise (hn_option_aux P)"
proof
  fix a a' p h F F'
  assume A: "h ⊨ hn_option_aux P a p * F ∧_A hn_option_aux P a' p * F'"
  thus "a=a'" proof (cases "(P,a,p)" rule: hn_option_aux.cases)
    case (2 _ av pv) hence [simp]: "a=Some av" "p=Some pv" by simp_all

    from A obtain av' where [simp]: "a'=Some av'" by (cases a', simp_all)

    from A have "h ⊨ P av pv * F ∧_A P av' pv * F'" by simp
    with ‘precise P‘ have "av=av'" by (rule preciseD)
    thus ?thesis by simp
  qed simp_all
qed

lemma pure_option[constraint_rules]:
  assumes P: "is_pure P"
  shows "is_pure (hn_option_aux P)"
proof -
  from P obtain P' where P': "⋀x x'. P x x' = ↑(P' x x')"
    using is_pureE by blast

  show ?thesis proof
    fix x x'
    show "hn_option_aux P x x' =
         ↑ (case (x, x') of
             (None,None) ⇒ True | (Some v, Some v') ⇒ P' v v' | _ ⇒
False
           )"
      apply (simp add: P' split: prod.split option.split)
      done
  qed
qed

lemma hn_case_option[sepref_comb_rules]:
  assumes P: "Γ⟹_AΓ1 * hn_option P p p'"
  assumes Rn: "p=None ⟹ hn_refine Γ1 fn' Γ2a R fn"
  assumes Rs: "⋀x x'. ⟦ p=Some x ⟧ ⟹
    hn_refine (Γ1 * hn_ctxt P x x') (fs' x') (Γ2b x x') R (fs x)"
  assumes F2: "!!x x'. Γ2b x x' ⟹_A Γ2b' * hn_ctxt Px' x x'"
  assumes MERGE2: "TERM case_list ⟹ Γ2a ∨_A Γ2b' ⟹_A Γ'"
```

```
  shows "hn_refine Γ (case_option fn' fs' p') (Γ' * hn_option Px' p p')
R
    (case_option$fn$(λ2x. fs x)$(p^L))"
  apply rule
  apply (rule cons_pre_rule[OF P])
  unfolding hn_ctxt_def
  apply vcg
  apply (simp, intro impI)
  apply (rule cons_post_rule)
  apply (rule hn_refineD[OF Rn[unfolded hn_ctxt_def]])
  apply simp_all
  apply solve_entails
  apply clarsimp
  apply assumption
  apply (auto
    simp: hn_ctxt_def
    intro!: ent_star_mono ent_disjI1[OF MERGE2[OF TERMI]]) []

  apply (simp split: option.split, intro impI allI)
  apply (rule cons_post_rule)
  apply (rule hn_refineD[OF Rs[unfolded hn_ctxt_def]])
  apply assumption
  apply simp


  apply (simp only: assn_aci)
  apply (rule ent_frame_fwd[OF F2])
  apply (fr_rot 2)
  apply (fr_rot_rhs 1)
  apply (rule fr_refl)
  apply (rule ent_refl)

  apply (rule ent_frame_fwd[OF ent_disjI2[OF MERGE2[OF TERMI]]])
  apply (fr_rot 2)
  apply (fr_rot_rhs 1)
  apply (rule fr_refl)
  apply (rule ent_refl)
  apply (simp add: hn_ctxt_def assn_aci)
  done

lemma hn_case_option_nl[sepref_comb_rules]:
  assumes "hn_refine Γ c Γ' R
    (bind$(COPY$p)$(λ2p. case_option$fn$fs$(p^L)))"
  shows "hn_refine Γ c Γ' R (case_option$fn$fs$(p^N))"
  using assms by simp

lemma hn_None[sepref_fr_rules]:
  "hn_refine emp (return None) emp (hn_option_aux P) (RETURN$None)"
  by rule sep_auto
```

```
lemma hn_Some[sepref_fr_rules]: "hn_refine
  (hn_ctxt P v v')
  (return (Some v'))
  (hn_invalid v v')
  (hn_option_aux P)
  (RETURN$(Some$v))"
  by rule (sep_auto simp: hn_ctxt_def)

definition "imp_option_eq eq a b ≡ case (a,b) of
  (None,None) ⇒ return True
| (Some a, Some b) ⇒ eq a b
| _ ⇒ return False"

lemma hn_option_eq[sepref_comb_rules]:
  fixes a b :: "'a option"
  assumes F1: "Γ ⟹A hn_option P a a' * hn_option P b b' * Γ1"
  assumes EQ: "⋀va va' vb vb'. hn_refine
    (hn_ctxt P va va' * hn_ctxt P vb vb' * Γ1)
    (eq' va' vb')
    (Γ' va va' vb vb')
    (pure bool_rel)
    (RETURN$(op=$LIN_ANNOT va Aa$LIN_ANNOT vb Ab))"
  assumes F2:
    "⋀va va' vb vb'.
      Γ' va va' vb vb' ⟹A hn_ctxt P va va' * hn_ctxt P vb vb' * Γ1"
  shows "hn_refine
    Γ
    (imp_option_eq eq' a' b')
    (hn_option P a a' * hn_option P b b' * Γ1)
    (pure bool_rel)
    (RETURN$(op=$LIN_ANNOT a Aa$LIN_ANNOT b Ab))"
  apply (rule hn_refine_cons_pre[OF F1])
  unfolding imp_option_eq_def
  apply rule
  apply (simp split: option.split add: hn_ctxt_def, intro impI conjI)

  apply (sep_auto split: option.split simp: hn_ctxt_def pure_def)
  apply (cases a, (sep_auto split: option.split simp: hn_ctxt_def pure_def)+)[]
  apply (cases a, (sep_auto split: option.split simp: hn_ctxt_def pure_def)+)[]
  apply (cases b, (sep_auto split: option.split simp: hn_ctxt_def pure_def)+)[]
  apply (rule cons_post_rule)
  apply (rule hn_refineD[OF EQ[unfolded hn_ctxt_def]])
  apply simp
  apply (rule ent_frame_fwd[OF F2[unfolded hn_ctxt_def]])
  apply (fr_rot 2)
  apply (fr_rot_rhs 1)
  apply (rule fr_refl)
  apply (rule ent_refl)
```

```
  apply (sep_auto simp: pure_def)
  done

lemma [pat_rules]:
  "op=$a$None ≡ is_None$a"
  "op=$None$a ≡ is_None$a"
  apply (rule eq_reflection, simp split: option.split)+
  done

lemma hn_is_None[sepref_fr_rules]: "hn_refine
  (hn_option P a a')
  (return (is_None a'))
  (hn_option P a a')
  (pure bool_rel)
  (RETURN$(is_None$a))"
  apply rule
  apply (sep_auto split: option.split simp: hn_ctxt_def pure_def)
  done

lemma (in -) sepref_the_complete:
  assumes "SIDE_PRECOND (x≠None)"
  shows "hn_refine
    (hn_option R x xi)
    (return (the xi))
    (hn_invalid x xi)
    (R)
    (RETURN$(the$x))"
    using assms
    apply (cases x)
    apply simp
    apply (cases xi)
    apply (simp add: hn_ctxt_def)
    apply rule
    apply (sep_auto simp: hn_ctxt_def)
    done


lemma (in -) sepref_the_id[sepref_fr_rules]:
  assumes "CONSTRAINT IS_PURE_ID R"
  shows "hn_refine
    (hn_option R x xi)
    (return (the xi))
    (hn_invalid x xi)
    (R)
    (RETURN$(the$x))"
    using assms
    apply (simp add: IS_PURE_ID_alt_def hn_ctxt_def)
    apply (cases x)
    apply simp
```

```
    apply (cases xi)
    apply (simp add: hn_ctxt_def)
    apply rule apply (sep_auto simp: pure_def)
    apply rule apply (sep_auto)
    apply (simp add: hn_option_pure_conv)
    apply rule apply (sep_auto simp: pure_def)
    done

lemma IS_PURE_ID_option[relator_props]: "⟦IS_PURE_ID R⟧
  ⟹ IS_PURE_ID (hn_option_aux R)"
  unfolding IS_PURE_ID_alt_def
  apply (intro ext)
  apply (rename_tac x y)
  apply (case_tac "(R,x,y)" rule: hn_option_aux.cases)
  apply (auto simp: hn_ctxt_def pure_def)
  done

lemmas [constraint_rules] = IS_PURE_ID_option
```

10.7 Lists

```
fun hn_list_aux :: "('a ⇒ 'c ⇒ assn) ⇒ 'a list ⇒ 'c list ⇒ assn"
where
  "hn_list_aux P [] [] = emp"
| "hn_list_aux P (a#as) (c#cs) = P a c * hn_list_aux P as cs"
| "hn_list_aux _ _ _ = false"

lemma hn_list_aux_simps[simp]:
  "hn_list_aux P [] l' = (↑(l'=[]))"
  "hn_list_aux P l [] = (↑(l=[]))"
  unfolding hn_ctxt_def
  apply (cases l')
  apply simp
  apply simp
  apply (cases l)
  apply simp
  apply simp
  done

lemma hn_list_aux_append[simp]:
  "length l1=length l1' ⟹
    hn_list_aux P (l1@l2) (l1'@l2')
    = hn_list_aux P l1 l1' * hn_list_aux P l2 l2'"
  apply (induct rule: list_induct2)
  apply simp
  apply (simp add: star_assoc)
  done

lemma hn_list_pure_conv: "hn_list_aux (pure R) = pure (⟨R⟩list_rel)"
```

```
proof (intro ext)
  fix l li
  show "hn_list_aux (pure R) l li = pure (⟨R⟩list_rel) l li"
    apply (induction "pure R" l li rule: hn_list_aux.induct)
    by (auto simp: pure_def)
qed

lemmas [sepref_import_rewrite, sepref_normrel_eqs] = hn_list_pure_conv[symmetric]


abbreviation "hn_list P ≡ hn_ctxt (hn_list_aux P)"

lemma hn_list_simps[simp]:
  "hn_list P [] l' = (↑(l'=[]))"
  "hn_list P l [] = (↑(l=[]))"
  "hn_list P [] [] = emp"
  "hn_list P (a#as) (c#cs) = hn_ctxt P a c * hn_list P as cs"
  "hn_list P (a#as) [] = false"
  "hn_list P [] (c#cs) = false"
  unfolding hn_ctxt_def
  apply (cases l')
  apply simp
  apply simp
  apply (cases l)
  apply simp
  apply simp
  apply simp_all
  done

lemma hn_list_precise[constraint_rules]: "precise P ⟹ precise (hn_list_aux
P)"
proof
  fix l1 l2 l h F1 F2
  assume P: "precise P"
  assume "h⊨hn_list_aux P l1 l * F1 ∧_A hn_list_aux P l2 l * F2"
  thus "l1=l2"
  proof (induct l arbitrary: l1 l2 F1 F2)
    case Nil thus ?case by simp
  next
    case (Cons a ls)
    from Cons obtain a1 ls1 where [simp]: "l1=a1#ls1"
      by (cases l1, simp)
    from Cons obtain a2 ls2 where [simp]: "l2=a2#ls2"
      by (cases l2, simp)

    from Cons.prems have M:
      "h ⊨ P a1 a * hn_list_aux P ls1 ls * F1
        ∧_A P a2 a * hn_list_aux P ls2 ls * F2" by simp
    have "a1=a2"
```

```
      apply (rule preciseD[OF P, where a=a1 and a'=a2 and p=a
        and F= "hn_list_aux P ls1 ls * F1"
        and F'="hn_list_aux P ls2 ls * F2"
        ])
      using M
      by (simp add: star_assoc)

    moreover have "ls1=ls2"
      apply (rule Cons.hyps[where ?F1.0="P a1 a * F1" and ?F2.0="P a2
a * F2"])
      using M
      by (simp only: star_aci)
    ultimately show ?case by simp
  qed
qed
lemma hn_list_pure[constraint_rules]:
  assumes P: "is_pure P"
  shows "is_pure (hn_list_aux P)"
proof -
  from P obtain P' where P_eq: "⋀x x'. P x x' = ↑(P' x x')"
    by (rule is_pureE) blast

  {
    fix l l'
    have "hn_list_aux P l l' = ↑(list_all2 P' l l')"
      by (induct P≡P l l' rule: hn_list_aux.induct)
         (simp_all add: P_eq)
  } thus ?thesis by rule
qed

lemma hn_list_mono:
  "⟦⋀x x'. P x x'⟹_A P' x x'⟧ ⟹ hn_list P l l' ⟹_A hn_list P' l l'"
  unfolding hn_ctxt_def
  apply (induct P l l' rule: hn_list_aux.induct)
  by (auto intro: ent_star_mono)

lemma hn_case_list[sepref_comb_rules]:
  assumes P: "Γ ⟹_A Γ1 * hn_list P l l'"
  assumes Rn: "l=[] ⟹ hn_refine Γ1 fn' Γ2a R fn"
  assumes Rc: "⋀x xs x' xs'. ⟦ l=x#xs ⟧ ⟹
    hn_refine (Γ1 * hn_ctxt P x x' * hn_list P xs xs') (fc' x' xs')
      (Γ2b x xs x' xs') R (fc x xs)"
  assumes F2: "!!x xs x' xs'. Γ2b x xs x' xs'
    ⟹_A Γ2b' * hn_ctxt Px' x x' * hn_ctxt Pxs' xs xs'"
  assumes MERGE2: "TERM case_list ⟹ Γ2a ∨_A Γ2b' ⟹_A Γ'"
  shows "hn_refine Γ (case_list fn' fc' l') (Γ' * hn_invalid l l') R

    (case_list$fn$(λ_2x xs. fc x xs)$(l^L))"
proof (cases l)
```

```
  case Nil[simp]

  from Rn have Rn':
    "nofail fn ⟹
      <Γ1> fn' <λr. ∃A x. Γ2a * hn_ctxt R x r * true * ↑ (RETURN x ≤
fn)>"
    by (simp add: hn_refine_def hn_ctxt_def)

  show ?thesis
    apply (rule hn_refine_cons_pre[OF P])
    apply simp
    apply rule
    apply clarsimp
    apply (rule cons_post_rule)
    apply (erule Rn')
    apply (rule ent_ex_preI ent_ex_postI)+
    apply (auto
      simp: hn_ctxt_def
      intro!: ent_star_mono ent_disjI1[OF MERGE2[OF TERMI]])
    done
next
  case (Cons x xs)[simp]
  show ?thesis
  proof (cases l')
    case Nil thus ?thesis
      apply (rule_tac hn_refine_cons_pre[OF P])
      apply rule
      apply (simp)
      done
  next
    case (Cons x' xs')[simp]

    from Rc[OF `l=x#xs`] have Rc': "⋀x' xs'.
      nofail (fc x xs) ⟹ <Γ1 * hn_ctxt P x x' * hn_list P xs xs'> fc'
x' xs'
        <λr. ∃A xa. Γ2b x xs x' xs' * hn_ctxt R xa r * true *
                  ↑ (RETURN xa ≤ fc x xs)>"
      by (clarsimp simp: hn_refine_def hn_ctxt_def)

    show ?thesis
      apply (rule hn_refine_cons_pre[OF P])
      apply rule
      apply (clarsimp)
      apply (simp only: star_assoc[symmetric])
      apply (rule cons_post_rule)
      apply (erule Rc')
      apply (rule ent_ex_preI ent_ex_postI ent_star_mono)+
      apply (rule ent_trans)
      apply (rule F2)
```

```
    apply (simp add: hn_ctxt_def)
    apply (intro ent_true_drop)
    apply (rule ent_disjI2[OF MERGE2[OF TERMI]])
    unfolding hn_ctxt_def apply (rule ent_refl)+
    done
  qed
qed

lemma hn_case_list_nl[sepref_comb_rules]:
  assumes "hn_refine Γ t' (Γ' * hn_invalid l l') R
    (bind$(COPY$l)$(λ2l. case_list$fn$(λ2x xs. fc x xs)$(l^L)))"
  shows "hn_refine Γ t' (Γ' * hn_invalid l l') R
    (case_list$fn$(λ2x xs. fc x xs)$(l^N))"
  using assms by simp

lemma hn_Nil[sepref_fr_rules]:
  "hn_refine emp (return []) emp (hn_list_aux P) (RETURN$[])"
  unfolding hn_refine_def
  by sep_auto

lemma hn_Cons[sepref_fr_rules]: "hn_refine (hn_ctxt P x x' * hn_list
P xs xs')
  (return (x'#xs')) (hn_invalid x x' * hn_invalid xs xs') (hn_list_aux
P)
  (RETURN$(op #$x$xs))"
  unfolding hn_refine_def
  by (sep_auto simp: hn_ctxt_def)

lemma hn_list_aux_len:
  "hn_list_aux P l l' = hn_list_aux P l l' * ↑(length l = length l')"
  apply (induct P≡P l l' rule: hn_list_aux.induct)
  apply simp_all
  apply (erule_tac t="hn_list_aux P as cs" in subst[OF sym])
  apply simp
  done

lemma hn_append[sepref_fr_rules]: "hn_refine (hn_list P l1 l1' * hn_list
P l2 l2')
  (return (l1'@l2')) (hn_invalid l1 l1' * hn_invalid l2 l2') (hn_list_aux
P)
  (RETURN$(op @$l1$l2))"
  apply rule
  apply (sep_auto simp: hn_ctxt_def)
  apply (subst hn_list_aux_len)
  apply (sep_auto simp: hn_list_aux_append)
  done


lemma pure_hn_list_eq_list_rel:
```

```
  "hn_val (⟨R⟩list_rel) = hn_list (pure R)"
proof (intro ext)
  fix a c
  show "hn_val (⟨R⟩list_rel) a c = hn_list (pure R) a c"
    apply (induct "pure R" a c rule: hn_list_aux.induct)
    apply (auto simp: hn_ctxt_def pure_def)
    apply (drule sym)
    apply (simp add: conj_commute)
    done
qed

lemma IS_PURE_ID_list[relator_props]: "⟦IS_PURE_ID R⟧
  ⟹ IS_PURE_ID (hn_list_aux R)"
  unfolding IS_PURE_ID_alt_def
proof (simp,intro ext)
  fix x :: "'a list" and y :: "'a list"
  show "hn_list_aux (pure Id) x y = pure Id x y"
    apply (induction R x y rule: hn_list_aux.induct)
    apply (auto simp: pure_def)
    done
qed

lemmas [constraint_rules] = IS_PURE_ID_list



end
```

11 Graphs defined by Successor Functions

```
theory Succ_Graph
imports
  "../../Refine_Dflt"
begin
```

This code is used in various examples

```
type_synonym 'a slg = "'a ⇒ 'a list"
definition slg_rel_def_internal: "slg_rel R ≡
  {(succs,G). ∀v. (succs v, G``{v}) ∈ ⟨R⟩list_set_rel}"

lemma slg_rel_def: "⟨R⟩slg_rel ≡
  {(succs,G). ∀v. (succs v, G``{v}) ∈ ⟨R⟩list_set_rel}"
  by (auto simp: slg_rel_def_internal relAPP_def)

lemma [relator_props]: "single_valued R ⟹ single_valued (⟨R⟩slg_rel)"
  unfolding slg_rel_def
  apply (rule single_valuedI)
  apply (auto dest: single_valuedD[OF list_set_rel_sv])
  done
```

```
consts i_slg :: "interface ⇒ interface"

lemmas [autoref_rel_intf] =
  REL_INTFI[of slg_rel i_slg]

definition [simp]: "slg_succs E v ≡ E``{v}"

lemma [autoref_itype]: "slg_succs ::ᵢ ⟨I⟩ᵢi_slg →ᵢ I →ᵢ ⟨I⟩ᵢi_set" by
simp

context begin interpretation autoref_syn .
lemma [autoref_op_pat]: "E``{v} ≡ slg_succs$E$v" by simp
end

lemma refine_slg_succs[autoref_rules_raw]:
  "(λsuccs v. succs v,slg_succs)∈⟨Id⟩slg_rel→Id→⟨Id⟩list_set_rel"
  apply (intro fun_relI)
  apply (simp add: slg_succs_def slg_rel_def)
  done
```

11.1 Graph Representations

```
definition succ_of_list :: "(nat×nat) list ⇒ nat ⇒ nat set"
  where
  "succ_of_list l ≡ let
    m = fold (λ(u,v) g.
          case g u of
            None ⇒ g(u↦{v})
          | Some s ⇒ g(u↦insert v s)
        ) l Map.empty
  in
    (λu. case m u of None ⇒ {} | Some s ⇒ s)

"

schematic_lemma succ_of_list_impl:
  notes [autoref_tyrel] =
    ty_REL[where 'a="nat→nat set" and R="⟨nat_rel,R⟩iam_map_rel" for
R]
    ty_REL[where 'a="nat set" and R="⟨nat_rel⟩list_set_rel"]

  shows "(?f::?'c,succ_of_list) ∈ ?R"
  unfolding succ_of_list_def[abs_def]
  apply (autoref (keep_goal))
  done

concrete_definition succ_of_list_impl uses succ_of_list_impl
export_code succ_of_list_impl in SML
```

```
definition acc_of_list :: "nat list ⇒ nat set"
  where "acc_of_list l ≡ fold insert l {}"

schematic_lemma acc_of_list_impl:
  notes [autoref_tyrel] =
    ty_REL[where 'a="nat set" and R="⟨nat_rel⟩iam_set_rel" for R]

  shows "(?f::?'c,acc_of_list) ∈ ?R"
  unfolding acc_of_list_def[abs_def]
  apply (autoref (keep_goal))
  done

concrete_definition acc_of_list_impl uses acc_of_list_impl
export_code acc_of_list_impl in SML

end
theory Array_Map_Impl
imports
  "../Sep_Main" Imp_Map_Spec Array_Blit
  "~~/src/HOL/Library/Code_Target_Numeral"
begin
```

11.2 Array Map

```
  type_synonym 'v array_map = "'v option array"
  definition "iam_initial_size ≡ 8::nat"

  definition "iam_of_list l i ≡ if i<length l then l!i else None"

  definition is_iam :: "(nat⇀'a) ⇒ ('a::heap) array_map ⇒ assn" where
    "is_iam m a ≡ ∃_A l. a↦_a l * ↑(m=iam_of_list l)"

  definition iam_new_sz :: "nat ⇒ ('v::heap) array_map Heap"
    where "iam_new_sz sz ≡ Array.new sz None"

  definition iam_new :: "('v::heap) array_map Heap"
    where "iam_new ≡ iam_new_sz iam_initial_size"

  definition iam_lookup
    :: "nat ⇒ ('v::heap) array_map ⇒ 'v option Heap"
    where "iam_lookup k a = do {
      l←Array.len a;
      if k < l then Array.nth a k else return None
    }"

  lemma [code]: "iam_lookup k a ≡ nth_oo None a k"
    unfolding nth_oo_def iam_lookup_def .
```

```
definition iam_delete
  :: "nat ⇒ ('v::heap) array_map ⇒ ('v::heap) array_map Heap"
where "iam_delete k a = do {
    l←Array.len a;
    if k < l then Array.upd k None a else return a
  }"

lemma [code]: "iam_delete k a ≡ upd_oo (return a) k None a"
  unfolding upd_oo_def iam_delete_def .

definition iam_update
  :: "nat ⇒ 'v::heap ⇒ 'v array_map ⇒ 'v array_map Heap"
  where "iam_update k v a = do {
    l←Array.len a;
    a←if k>=l then do {
        let newsz = max (k+1) (2 * l + 3);
        array_grow a newsz None
      } else return a;

    Array.upd k (Some v) a
  }"

lemma [code]: "iam_update k v a = upd_oo
  (do {
    l←Array.len a;
    let newsz = max (k+1) (2 * l + 3);
    a←array_grow a newsz None;
    Array.upd k (Some v) a
  })
  k (Some v) a"
proof -
  have [simp]:
    "⋀x t e. do {
      l←Array.len a;
      if x l then
        t l
      else do {
        l'←Array.len a;
        e l l'
      }
    }
    =
    do {
      l←Array.len a;
      if x l then t l else e l l
    }"
    apply (auto
      simp: bind_def execute_len
      split: option.split
```

```
      intro!: ext
    )
    done

  show ?thesis
    unfolding upd_oo_def iam_update_def
    apply simp
    apply (rule cong[OF arg_cong, where f1=bind])
    apply simp
    apply (rule ext)
    apply auto
    done
qed

lemma precise_iam: "precise is_iam"
  apply rule
  by (auto simp add: is_iam_def dest: preciseD[OF snga_prec])

lemma iam_new_abs: "iam_of_list (replicate n None) = Map.empty"
  unfolding iam_of_list_def[abs_def]
  by auto

lemma iam_new_sz_rule: "<emp> iam_new_sz n < is_iam Map.empty >"
  unfolding iam_new_sz_def is_iam_def[abs_def]
  by (sep_auto simp: iam_new_abs)

lemma iam_new_rule: "<emp> iam_new < is_iam Map.empty >"
  unfolding iam_new_def by (sep_auto heap: iam_new_sz_rule)

lemma iam_lookup_abs1: "k<length l ⟹ iam_of_list l k = l!k"
  by (simp add: iam_of_list_def)
lemma iam_lookup_abs2: "¬k<length l ⟹ iam_of_list l k = None"
  by (simp add: iam_of_list_def)

lemma iam_lookup_rule: "< is_iam m p >
  iam_lookup k p
  <λr. is_iam m p * ↑(r=m k) >"
  unfolding iam_lookup_def is_iam_def
  by (sep_auto simp: iam_lookup_abs1 iam_lookup_abs2)

lemma iam_delete_abs1: "k<length l
  ⟹ iam_of_list (l[k := None]) = iam_of_list l |` (- {k})"
  unfolding iam_of_list_def[abs_def]
  by (auto intro!: ext simp: restrict_map_def)

lemma iam_delete_abs2: "¬k<length l
  ⟹ iam_of_list l |` (- {k}) = iam_of_list l"
  unfolding iam_of_list_def[abs_def]
  by (auto intro!: ext simp: restrict_map_def)
```

```
  lemma iam_delete_rule: "< is_iam m p >
    iam_delete k p
    <λr. is_iam (m|`(-{k})) r>"
    unfolding is_iam_def iam_delete_def
    by (sep_auto simp: iam_delete_abs1 iam_delete_abs2)


  lemma iam_update_abs1: "iam_of_list (l@replicate n None) = iam_of_list
l"
    unfolding iam_of_list_def[abs_def]
    by (auto intro!: ext simp: nth_append)

  lemma iam_update_abs2: "¬ length l ≤ k
    ⟹ iam_of_list (l[k := Some v]) = iam_of_list l(k ↦ v)"
    unfolding iam_of_list_def[abs_def]
    by auto

  lemma iam_update_rule:
    "< is_iam m p > iam_update k v p <λr. is_iam (m(k↦v)) r>ₜ"
    unfolding is_iam_def iam_update_def
    by (sep_auto
      decon: decon_split_if
      simp: iam_update_abs1 iam_update_abs2)

  interpretation iam: imp_map is_iam
    apply unfold_locales
    by (rule precise_iam)
  interpretation iam: imp_map_empty is_iam iam_new
    apply unfold_locales
    by (sep_auto heap: iam_new_rule)
  interpretation iam_sz: imp_map_empty is_iam "iam_new_sz sz"
    apply unfold_locales
    by (sep_auto heap: iam_new_sz_rule)

  interpretation iam: imp_map_lookup is_iam iam_lookup
    apply unfold_locales
    by (sep_auto heap: iam_lookup_rule)
  interpretation iam: imp_map_delete is_iam iam_delete
    apply unfold_locales
    by (sep_auto heap: iam_delete_rule)
  interpretation iam: imp_map_update is_iam iam_update
    apply unfold_locales
    by (sep_auto heap: iam_update_rule)

end
theory Sepref_IICF_Bindings
imports Sepref_Tool
  "Sepref_HOL_Bindings"
```

```
  "../Collections/Examples/Autoref/Succ_Graph"
  "../Separation_Logic_Imperative_HOL/Sep_Examples"
  "../Collections/Refine_Dflt_ICF"
begin
```

11.2.1 List-Set

```
lemmas [sepref_import_param] =
  list_set_autoref_empty
  list_set_autoref_member
  list_set_autoref_insert
  list_set_autoref_delete
```

11.2.2 Imperative Set

```
lemma (in imp_set_empty) hn_set_empty:
  shows "hn_refine emp (empty) emp (is_set) (RETURN${})"
  unfolding hn_refine_def
  by (sep_auto simp: hn_ctxt_def)

lemma (in imp_set_ins) hn_set_ins:
  shows
  "hn_refine (hn_ctxt is_set s s' * hn_val Id x x') (ins x' s')
    (hn_invalid s s' * hn_val Id x x') is_set (RETURN$(insert$x$s))"
  unfolding hn_refine_def hn_ctxt_def pure_def
  by sep_auto

lemma (in imp_set_memb) hn_set_memb:
  "hn_refine (hn_ctxt is_set s s' * hn_val Id x x') (memb x' s')
    (hn_ctxt is_set s s' * hn_val Id x x') (pure bool_rel)
    (RETURN$(op ∈$x$s))"
  unfolding hn_refine_def hn_ctxt_def pure_def
  by sep_auto

lemma (in imp_set_delete) hn_set_delete:
  shows
  "hn_refine (hn_ctxt is_set s s' * hn_val Id x x') (delete x' s')
    (hn_invalid s s' * hn_val Id x x') is_set (RETURN$(op_set_delete$x$s))"
  unfolding hn_refine_def hn_ctxt_def pure_def
  by sep_auto

lemmas (in imp_set) [constraint_rules] = precise
```

Empty set with initial size hint

```
definition empty_set_sz :: "nat => 'a set"
  where [simp]: "empty_set_sz n == {}"

lemma hn_ias_empty_sz[sepref_fr_rules]: "hn_refine
  (hn_val nat_rel s s') (ias_new_sz s') (hn_val nat_rel s s') is_ias
  (RETURN$(empty_set_sz$s))"
```

```
  by rule sep_auto


lemmas [sepref_fr_rules] =
  hs.hn_set_empty hs.hn_set_ins hs.hn_set_memb hs.hn_set_delete

lemmas [sepref_fr_rules] =
  ias.hn_set_empty ias.hn_set_ins ias.hn_set_memb ias.hn_set_delete

lemma pat_set_delete[pat_rules]: "op-$s$(insert$x${}) ≡ op_set_delete$x$s"
  by simp
```

11.2.3 Imperative Map

```
context imp_map begin
  definition "is_map_rel Rk Rv m mi ≡ is_map m mi * ↑(Rk=pure Id ∧ Rv=pure
Id)"
  lemma is_map_rel_eq[simp]:
    "IS_PURE_ID Rk ⟹ IS_PURE_ID Rv ⟹ is_map_rel Rk Rv = is_map"
    apply (intro ext)
    unfolding is_map_rel_def pure_def IS_PURE_ID_alt_def
    by auto

  lemma precise_rel: "precise (is_map_rel Rk Rv)"
    using precise
    by (simp add: is_map_rel_def[abs_def] precise_def)

end

lemma (in imp_map_empty) hn_map_empty:
  assumes "INDEP Rk" "INDEP Rv"
  assumes "CONSTRAINT IS_PURE_ID Rk"
  assumes "CONSTRAINT IS_PURE_ID Rv"
  shows "hn_refine emp (empty) emp (is_map_rel Rk Rv) (RETURN$op_map_empty)"
  unfolding hn_refine_def
  using assms
  by (sep_auto simp: hn_ctxt_def)

lemma (in imp_map_is_empty) hn_map_is_empty:
  shows "hn_refine (hn_ctxt (is_map_rel Rk Rv) m mi) (is_empty mi) (hn_ctxt
(is_map_rel Rk Rv) m mi) (pure bool_rel) (RETURN$(op_map_is_empty$m))"
  apply rule
  unfolding hn_ctxt_def pure_def is_map_rel_def
  by sep_auto

lemma (in imp_map_lookup) hn_map_lookup:
  shows "hn_refine
    (hn_ctxt (is_map_rel Rk Rv) m mi * hn_ctxt Rk k ki)
    (lookup ki mi)
```

```
    (hn_ctxt (is_map_rel Rk Rv) m mi * hn_ctxt Rk k ki)
    (hn_option_aux Rv)
    (RETURN$(op_map_lookup$k$m))"
  apply rule
  using assms
  unfolding hn_ctxt_def is_map_rel_def
  apply (simp add: hn_option_pure_conv)
  by (sep_auto simp: pure_def)

lemma (in imp_map_update) hn_map_update:
  shows "hn_refine
    (hn_ctxt (is_map_rel Rk Rv) m mi * hn_ctxt Rk k ki * hn_ctxt Rv v
vi)
    (update ki vi mi)
    (hn_invalid m mi * hn_ctxt Rk k ki * hn_ctxt Rv v vi)
    (is_map_rel Rk Rv)
    (RETURN$(op_map_update$k$v$m))"
  apply rule
  using assms
  unfolding hn_ctxt_def pure_def is_map_rel_def
  by sep_auto

lemma (in imp_map_delete) hn_map_delete:
  shows "hn_refine
    (hn_ctxt (is_map_rel Rk Rv) m mi * hn_ctxt Rk k ki)
    (delete k mi)
    (hn_invalid m mi * hn_ctxt Rk k ki)
    (is_map_rel Rk Rv)
    (RETURN$(op_map_delete$k$m))"
  apply rule
  using assms
  unfolding hn_ctxt_def pure_def is_map_rel_def
  by sep_auto

lemma (in imp_map_add) hn_map_add:
  "hn_refine
    (hn_ctxt (is_map_rel Rk Rv) m1 mi1 * hn_ctxt (is_map_rel Rk Rv) m2
mi2)
    (add mi1 mi2)
    (hn_ctxt (is_map_rel Rk Rv) m1 mi1 * hn_ctxt (is_map_rel Rk Rv) m2
mi2)
    (is_map_rel Rk Rv)
    (RETURN$(op++$m1$m2))"
  apply rule
  unfolding hn_ctxt_def pure_def is_map_rel_def
  by sep_auto


lemmas [sepref_fr_rules] =
```

hm.hn_map_empty hm.hn_map_is_empty hm.hn_map_lookup hm.hn_map_delete hm.hn_map_update
iam.hn_map_empty iam.hn_map_lookup iam.hn_map_delete iam.hn_map_update

lemmas *(in imp_map) [constraint_rules] = precise precise_rel*

11.3 Graphs

typedecl *'v i_slg*

lemma *pat_slg_succs[pat_rules]:*
"op‘‘E(insert$v${}) == slg_succs$E$v" **by** *simp*

lemma *id_slg_succs[id_rules]:*
"slg_succs ::$_i$ TYPE('v i_slg ⇒ 'v ⇒ 'v set)" **by** *simp*

lemmas *[sepref_import_param] = refine_slg_succs*

11.4 Unique priority queues (from functional ICF)

typedecl *('e,'a) i_uprio*

lemma *[sepref_la_skel]:*
"SKEL (prio_pop_min$a) = la_op a" **by** *simp*

lemmas *[id_rules] =*
itypeI[of op_uprio_empty "TYPE(('a,'b)i_uprio)"]
itypeI[of op_uprio_insert "TYPE(('a,'b)i_uprio ⇒ 'a ⇒ 'b ⇒ ('a,'b)i_uprio)"]
itypeI[of prio_pop_min "TYPE(('a,'b)i_uprio ⇒ ('a×'b×('a,'b)i_uprio) nres)"]
itypeI[of op_uprio_is_empty "TYPE(('a,'b)i_uprio ⇒ bool)"]
itypeI[of op_uprio_prio "TYPE(('a,'b)i_uprio ⇒ 'a ⇒ 'b option)"]

lemma *pat_uprio_empty[pat_rules]:*
"(λ$_2$_. None)::('e ⇀ 'a::linorder) ≡ op_uprio_empty"
by *simp*

lemma *pat_uprio_insert[pat_rules]:*
"fun_upd$m$k$(Some$v) ≡ op_uprio_insert$'m$'k$'v"
by *simp*

lemma *pat_uprio_lookup[pat_rules]:*
"m$k ≡ op_uprio_prio$'m$'k"
by *simp*

lemma *pat_uprio_is_empty[pat_rules]:*
"op =m(λ$_2$_. None) ≡ op_uprio_is_empty$m"
"op =$(λ$_2$_. None)$m ≡ op_uprio_is_empty$m"
"op =$(dom$m)${} ≡ op_uprio_is_empty$m"

```
  "op =${}$(dom$m) ≡ op_uprio_is_empty$m"
  unfolding atomize_eq
  by auto

context uprio begin
  definition "is_uprio Rk Rv c ci ≡ ↑(Rk=pure Id ∧ Rv=pure Id ∧ c=α ci
∧ invar ci)"

  lemma is_uprio_alt_def: "is_uprio Rk Rv c ci
    = ↑(Rk=pure Id ∧ Rv=pure Id) * pure (br α invar) c ci"
    unfolding is_uprio_def pure_def
    apply (auto simp: br_def)
    done


  lemma [constraint_rules]: "precise (is_uprio Rk Rv)"
    unfolding is_uprio_alt_def[abs_def]
    by (simp add: precise_pure)


end

lemma (in uprio_empty) sepref_empty[sepref_fr_rules]:
  assumes "CONSTRAINT IS_PURE_ID Rk"
  assumes "CONSTRAINT IS_PURE_ID Rv"
  shows
  "hn_refine emp (return (empty ())) emp (is_uprio Rk Rv) (RETURN$op_uprio_empty)"
  apply rule
  using assms
  apply (simp add: IS_PURE_ID_alt_def is_uprio_def)
  apply (sep_auto simp: empty_correct)
  done

lemma (in uprio_insert) sepref_insert[sepref_fr_rules]:
  shows
  "hn_refine
    (hn_ctxt (is_uprio Rk Rv) m mi * hn_ctxt Rk k ki * hn_ctxt Rv v vi)
    (return (insert mi ki vi))
    (hn_ctxt (is_uprio Rk Rv) m mi * hn_ctxt Rk k ki * hn_ctxt Rv v vi)
    (is_uprio Rk Rv)
    (RETURN$(op_uprio_insert$m$k$v))"
  apply rule
  using assms
  apply (simp add: hn_ctxt_def is_uprio_def pure_def)
  by (sep_auto simp: insert_correct)

lemma (in uprio_isEmpty) sepref_is_empty[sepref_fr_rules]:
  "hn_refine
    (hn_ctxt (is_uprio Rk Rv) m mi)
```

```
    (return (isEmpty mi))
    (hn_ctxt (is_uprio Rk Rv) m mi)
    (pure bool_rel)
    (RETURN$(op_uprio_is_empty$m))"
  apply rule
  using assms
  apply (simp add: hn_ctxt_def is_uprio_def pure_def)
  by (sep_auto simp: isEmpty_correct)

lemma (in uprio_prio) sepref_prio[sepref_fr_rules]:
  shows
  "hn_refine
    (hn_ctxt (is_uprio Rk Rv) m mi * hn_ctxt Rk k ki)
    (return (prio mi ki))
    (hn_ctxt (is_uprio Rk Rv) m mi * hn_ctxt Rk k ki)
    (hn_option_aux Rv)
    (RETURN$(op_uprio_prio$m$k))"
  apply rule
  apply (simp add: hn_ctxt_def is_uprio_def)
  apply (simp add: hn_option_pure_conv)
  by (sep_auto simp: prio_correct pure_def)


lemma (in uprio_pop) sepref_prio_pop_min[sepref_fr_rules]:
  "hn_refine
    (hn_ctxt (is_uprio Rk Rv) m mi)
    (return (pop mi))
    (hn_ctxt (is_uprio Rk Rv) m mi)
    (hn_prod_aux Rk (hn_prod_aux Rv (is_uprio Rk Rv)))
    (prio_pop_min$m)"
  apply (rule)
  apply (simp add: hn_ctxt_def is_uprio_alt_def[abs_def])
  apply (simp add: hn_prod_pure_conv)
  apply (clarsimp simp: pure_def)
  apply (erule (1) return_refine_prop_return[OF _ prio_pop_min_refine])
  apply (sep_auto)
  done




end
theory Pf_Add
imports "../Automatic_Refinement/Lib/Misc" "~~/src/HOL/Library/Monad_Syntax"
begin

lemma fun_ordI:
  assumes "⋀x. ord (f x) (g x)"
```

```
  shows "fun_ord ord f g"
  using assms unfolding fun_ord_def by auto

lemma fun_ordD:
  assumes "fun_ord ord f g"
  shows "ord (f x) (g x)"
  using assms unfolding fun_ord_def by auto

lemma mono_fun_fun_cnv:
  assumes "⋀d. monotone (fun_ord ordA) ordB (λx. F x d)"
  shows "monotone (fun_ord ordA) (fun_ord ordB) F"
  apply rule
  apply (rule fun_ordI)
  using assms
  by (blast dest: monotoneD)

lemma fun_lub_Sup[simp]: "fun_lub Sup = Sup"
  unfolding fun_lub_def[abs_def]
  by (auto intro!: ext simp: SUP_def image_def)

lemma fun_ord_le[simp]: "fun_ord op ≤ = op ≤"
  unfolding fun_ord_def[abs_def]
  by (auto intro!: ext simp: le_fun_def)

end
```

12 Setup for Foreach Combinator

```
theory Sepref_Foreach
imports Sepref_HOL_Bindings Sepref_IICF_Bindings Pf_Add
begin
```

12.1 Foreach Loops

12.1.1 Monadic Version of Foreach

In a first step, we define a version of foreach where the continuation condition is also monadic, and show that it is equal to the standard version for continuation conditions of the form `λx. RETURN (c x)`

```
definition "FOREACH_inv xs Φ s ≡
  case s of (it, σ) ⇒ ∃xs'. xs = xs' @ it ∧ Φ (set it) σ"

definition "monadic_FOREACH R Φ S c f σ0 ≡ do {
  ASSERT (finite S);
  xs0 ← it_to_sorted_list R S;
  (_,σ) ← RECT (λW (xs,σ). do {
    ASSERT (FOREACH_inv xs0 Φ (xs,σ));
    if xs≠[] then do {
```

```
      b ← c σ;
      if b then
        FOREACH_body f (xs,σ) ≫= W
      else
        RETURN (xs,σ)
    } else RETURN (xs,σ)
  }) (xs0,σ0);
  RETURN σ
}"
```

lemma *FOREACH_oci_to_monadic:*

```
  "FOREACHoci R Φ S c f σ0 = monadic_FOREACH R Φ S (λσ. RETURN (c σ))
f σ0"
```

unfolding *FOREACHoci_def monadic_FOREACH_def WHILEIT_def WHILEI_body_def*
unfolding *it_to_sorted_list_def FOREACH_cond_def FOREACH_inv_def*
apply *simp*
apply *(fo_rule arg_cong[THEN cong] | rule refl ext)+*
apply *(simp split: prod.split)*
apply *(rule refl)+*
done

Next, we define a characterization w.r.t. *nfoldli*

definition *"monadic_nfoldli l c f s ≡ RECT (λD (l,s). case l of*

```
    [] ⇒ RETURN s
  | x#ls ⇒ do {
      b ← c s;
      if b then do { s'←f x s; D (ls,s')} else RETURN s
    }
  ) (l,s)"
```

lemma *monadic_nfoldli_eq:*

```
  "monadic_nfoldli l c f s = (
    case l of
      [] ⇒ RETURN s
    | x#ls ⇒ do {
        b←c s;
        if b then f x s ≫= monadic_nfoldli ls c f else RETURN s
      }
  )"
```

apply *(subst monadic_nfoldli_def)*
apply *(subst RECT_unfold)*
apply *(tagged_solver)*
apply *(subst monadic_nfoldli_def[symmetric])*
apply *simp*
done

lemma *monadic_nfoldli_simp[simp]:*

```
  "monadic_nfoldli [] c f s = RETURN s"
  "monadic_nfoldli (x#ls) c f s = do {
```

```
    b←c s;
    if b then f x s ≫= monadic_nfoldli ls c f else RETURN s
  }"
  apply (subst monadic_nfoldli_eq, simp)
  apply (subst monadic_nfoldli_eq, simp)
  done

lemma nfoldli_to_monadic:
  "nfoldli l c f = monadic_nfoldli l (λx. RETURN (c x)) f"
  apply (induct l)
  apply auto
  done

definition "nfoldli_alt l c f s ≡ RECT (λD (l,s). case l of
    [] ⇒ RETURN s
  | x#ls ⇒ do {
      let b = c s;
      if b then do { s'←f x s; D (ls,s')} else RETURN s
    }
  ) (l,s)"

lemma nfoldli_alt_eq:
  "nfoldli_alt l c f s = (
    case l of
      [] ⇒ RETURN s
    | x#ls ⇒ do {let b=c s; if b then f x s ≫= nfoldli_alt ls c f else
RETURN s}
  )"
  apply (subst nfoldli_alt_def)
  apply (subst RECT_unfold)
  apply (tagged_solver)
  apply (subst nfoldli_alt_def[symmetric])
  apply simp
  done

lemma nfoldli_alt_simp[simp]:
  "nfoldli_alt [] c f s = RETURN s"
  "nfoldli_alt (x#ls) c f s = do {
    let b = c s;
    if b then f x s ≫= nfoldli_alt ls c f else RETURN s
  }"
  apply (subst nfoldli_alt_eq, simp)
  apply (subst nfoldli_alt_eq, simp)
  done


lemma nfoldli_alt:
  "(nfoldli::'a list ⇒ ('b ⇒ bool) ⇒ ('a ⇒ 'b ⇒ 'b nres) ⇒ 'b ⇒
'b nres)
```

```
  = nfoldli_alt"
proof (intro ext)
  fix l::"'a list" and c::"'b ⇒ bool" and f::"'a ⇒ 'b ⇒ 'b nres" and
s :: 'b
  have "nfoldli l c f = nfoldli_alt l c f"
    by (induct l) auto
  thus "nfoldli l c f s = nfoldli_alt l c f s" by simp
qed

lemma monadic_nfoldli_rec:
  "monadic_nfoldli x' c f σ
         ≤⇓Id (REC_T
            (λW (xs, σ).
                ASSERT (FOREACH_inv xs0 I (xs, σ)) ≫=
                (λ_. if xs = [] then RETURN (xs, σ)
                     else c σ ≫=
                          (λb. if b then FOREACH_body f (xs, σ) ≫= W
                               else RETURN (xs, σ))))
            (x', σ) ≫=
           (λ(_, y). RETURN y))"
  apply (induct x' arbitrary: σ)

  apply (subst RECT_unfold, refine_mono)
  apply (simp)
  apply (rule le_ASSERTI)
  apply simp

  apply (subst RECT_unfold, refine_mono)
  apply (subst monadic_nfoldli_simp)
  apply (simp del: conc_Id)
  apply refine_rcg
  apply (clarsimp simp add: FOREACH_body_def)
  apply (rule bind_mono(1)[OF order_refl])
  apply assumption
  done


lemma monadic_FOREACH_itsl:
  fixes R I tsl
  shows
    "do { l ← it_to_sorted_list R s; monadic_nfoldli l c f σ }
     ≤ monadic_FOREACH R I s c f σ"
    apply (rule refine_IdD)
    unfolding monadic_FOREACH_def it_to_sorted_list_def
    apply (refine_rcg)
    apply simp
    apply (rule monadic_nfoldli_rec[simplified])
    done
```

```
lemma FOREACHoci_itsl:
  fixes R I tsl
  shows
    "do { l ← it_to_sorted_list R s; nfoldli l c f σ }
     ≤ FOREACHoci R I s c f σ"
    apply (rule refine_IdD)
    unfolding FOREACHoci_def it_to_sorted_list_def
    apply refine_rcg
    apply simp
    apply (rule nfoldli_while)
    done

lemma [def_pat_rules]:
  "FOREACHc ≡ PR_CONST (FOREACHoci (λ_ _. True) (λ_ _. True))"
  "FOREACHci$I ≡ PR_CONST (FOREACHoci (λ_ _. True) I)"
  "FOREACHi$I ≡ λ2s. PR_CONST (FOREACHoci (λ_ _. True) I)$s$(λ2x. True)"
  "FOREACH ≡ FOREACHi$(λ2_ _. True)"
  by (simp_all add:
    FOREACHci_def FOREACHi_def[abs_def] FOREACHc_def FOREACH_def[abs_def])

term "FOREACHoci R I"
lemma id_FOREACHoci[id_rules]: "PR_CONST (FOREACHoci R I) ::i
  TYPE('c set ⇒ ('d ⇒ bool) ⇒ ('c ⇒ 'd ⇒ 'd nres) ⇒ 'd ⇒ 'd nres)"
  by simp
```

We set up the monadify-phase such that all FOREACH-loops get rewritten to the monadic version of FOREACH

```
lemma FOREACH_arities[sepref_monadify_arity]:

  "PR_CONST (FOREACHoci R I) ≡ λ2s c f σ. SP (PR_CONST (FOREACHoci R
I))$s$(λ2x. c$x)$(λ2x σ. f$x$σ)$σ"
  by (simp_all)

lemma FOREACHoci_comb[sepref_monadify_comb]:
  "⋀s c f σ. (PR_CONST (FOREACHoci R I))$s$(λ2x. c x)$f$σ ≡
    bind$(EVAL$s)$(λ2s. bind$(EVAL$σ)$(λ2σ.
      SP (PR_CONST (monadic_FOREACH R I))$s$(λ2x. (EVAL$(c x)))$f$σ
    ))"
  by (simp_all add: FOREACH_oci_to_monadic)
```

Setup for linearity analysis.

```
lemma monadic_FOREACH_skel[sepref_la_skel]:
  "⋀s c f σ. SKEL ((PR_CONST (monadic_FOREACH R I))$s$c$f$σ) =
    la_seq
      (la_op (s,σ))
      (la_rec (λD. la_seq (SKEL c) (la_seq (SKEL f) (la_rcall D)))
      )" by simp
```

12.1.2 Imperative Version of nfoldli

We define an imperative version of *nfoldli*. It is the equivalent to the monadic version in the nres-monad

definition *"imp_nfoldli l c f s ≡ heap.fixp_fun (λD (l,s). case l of*
[] ⇒ return s
| x#ls ⇒ do {
b←c s;
if b then do { s'←f x s; D (ls,s')} else return s
}
) (l,s)"

declare *imp_nfoldli_def[code del]*

lemma *imp_nfoldli_simps[simp,code]:*
"imp_nfoldli [] c f s = return s"
"imp_nfoldli (x#ls) c f s = (do {
b ← c s;
if b then do {
s'←f x s;
imp_nfoldli ls c f s'
} else return s
})"
apply -
unfolding *imp_nfoldli_def*
apply *(subst heap.mono_body_fixp)*
apply *(tactic {* Pf_Mono_Prover.mono_tac @{context} 1 *})*
apply *simp*
apply *(subst heap.mono_body_fixp)*
apply *(tactic {* Pf_Mono_Prover.mono_tac @{context} 1 *})*
apply *simp*
done

lemma *monadic_nfoldli_refine_aux:*
assumes *c_ref: "⋀s s'. hn_refine*
*(Γ * hn_ctxt Rs s' s)*
(c s)
*(Γ * hn_ctxt Rs s' s)*
(pure bool_rel)
(c' s')"
assumes *f_ref: "⋀x x' s s'. hn_refine*
*(Γ * hn_ctxt Rl x' x * hn_ctxt Rs s' s)*
(f x s)
*(Γ * hn_invalid x' x * hn_invalid s' s) Rs*
(f' x' s')"

shows *"hn_refine*
*(Γ * hn_list Rl l' l * hn_ctxt Rs s' s)*
(imp_nfoldli l c f s)

```
  (Γ * hn_invalid l' l * hn_invalid s' s) Rs
  (monadic_nfoldli l' c' f' s')"

apply (induct p≡Rl l' l
  arbitrary: s s'
  rule: hn_list_aux.induct)

apply simp
apply (rule hn_refine_cons_post)
apply (rule hn_refine_frame[OF hnr_RETURN_pass])
apply (tactic {* Sepref_Frame.frame_tac @{context} 1 *})
apply (simp add: hn_ctxt_def ent_true_drop)

apply (simp only: imp_nfoldli_simps monadic_nfoldli_simp)
apply (rule hnr_bind)
apply (rule hn_refine_frame[OF c_ref])
apply (tactic {* Sepref_Frame.frame_tac @{context} 1 *})

apply (rule hnr_If)
apply (tactic {* Sepref_Frame.frame_tac @{context} 1 *})
apply (rule hnr_bind)
apply (rule hn_refine_frame[OF f_ref])
apply (simp add: assn_aci)
apply (fr_rot_rhs 1)
apply (fr_rot 2)
apply (rule fr_refl)
apply (rule fr_refl)
apply (rule fr_refl)
apply (rule ent_refl)

apply (rule hn_refine_frame)
apply rprems

apply (simp add: assn_aci)
apply (fr_rot_rhs 1)
apply (fr_rot 2)
apply (rule fr_refl)
apply (fr_rot 1)
apply (rule fr_refl)
apply (rule fr_refl)
apply (rule ent_refl)

apply (tactic {* Sepref_Frame.frame_tac @{context} 1 *})

apply (rule hn_refine_frame[OF hnr_RETURN_pass])
apply (tactic {* Sepref_Frame.frame_tac @{context} 1 *})

apply (simp add: assn_assoc)
apply (tactic {* Sepref_Frame.merge_tac @{context} 1 *})
```

```
  apply (simp only: sup.idem, rule ent_refl)
  apply (simp add: hn_ctxt_def, (rule fr_refl ent_refl)+) []

  apply (rule, sep_auto)
  apply (rule, sep_auto)
  done

lemma hn_monadic_nfoldli:
  assumes FR: "P ⟹_A Γ * hn_list Rl l' l * hn_ctxt Rs s' s"
  assumes c_ref: "⋀s s'. hn_refine
    (Γ * hn_ctxt Rs s' s)
    (c s)
    (Γ * hn_ctxt Rs s' s)
    (pure bool_rel)
    (c'$s')"
  assumes f_ref: "⋀x x' s s'. hn_refine
    (Γ * hn_ctxt Rl x' x * hn_ctxt Rs s' s)
    (f x s)
    (Γ * hn_invalid x' x * hn_invalid s' s) Rs
    (f'$x'$s')"
  shows "hn_refine
    P
    (imp_nfoldli l c f s)
    (Γ * hn_invalid l' l * hn_invalid s' s)
    Rs
    (monadic_nfoldli$l'$c'$f'$s')
    "
  apply (rule hn_refine_cons_pre[OF FR])
  unfolding APP_def
  apply (rule monadic_nfoldli_refine_aux)
  apply (rule c_ref[unfolded APP_def])
  apply (rule f_ref[unfolded APP_def])
  done

lemma hn_itsl:
  assumes ITSL: "is_set_to_sorted_list ordR Rk Rs tsl"
  shows "hn_refine
    (hn_val (⟨Rk⟩Rs) s' s)
    (return (tsl s))
    (hn_val (⟨Rk⟩Rs) s' s)
    (pure (⟨Rk⟩list_rel))
    (it_to_sorted_list ordR s')"
  apply rule
  unfolding hn_ctxt_def pure_def
  apply vcg
  apply clarsimp
  apply (erule is_set_to_sorted_listE[OF ITSL])
  apply sep_auto
  done
```

lemma *hn_monadic_FOREACH[sepref_comb_rules]:*
assumes *"INDEP Rk" "INDEP Rs" "INDEP Rσ"*
assumes *FR: "P ⟹$_A$ Γ * hn_val (⟨Rk⟩Rs) s' s * hn_ctxt Rσ σ' σ"*
assumes *STL: "GEN_ALGO_tag (is_set_to_sorted_list ordR Rk Rs tsl)"*
assumes *c_ref: "⋀σ σ'. hn_refine*
*(Γ * hn_val (⟨Rk⟩Rs) s' s * hn_ctxt Rσ σ' σ)*
(c σ)
(Γc σ' σ)
(pure bool_rel)
(c' σ')"
assumes *C_FR:*
"⋀σ' σ. TERM monadic_FOREACH ⟹
*Γc σ' σ ⟹$_A$ Γ * hn_val (⟨Rk⟩Rs) s' s * hn_ctxt Rσ σ' σ"*

assumes *f_ref: "⋀x' x σ' σ. hn_refine*
*(Γ * hn_val (⟨Rk⟩Rs) s' s * hn_val Rk x' x * hn_ctxt Rσ σ' σ)*
(f x σ)
(Γf x' x σ' σ) Rσ
(f' x' σ')"
assumes *F_FR: "⋀x' x σ' σ. TERM monadic_FOREACH ⟹ Γf x' x σ' σ*
⟹$_A$
*Γ * hn_val (⟨Rk⟩Rs) s' s * hn_ctxt Pfx x' x * hn_ctxt Pfσ σ' σ"*

shows *"hn_refine*
P
(imp_nfoldli (tsl$s) c f σ) (Important: Using tagged application*
*to avoid ho-unifier problems *)*
*(Γ * hn_val (⟨Rk⟩Rs) s' s * hn_invalid σ' σ)*
Rσ
((PR_CONST (monadic_FOREACH ordR I))
$(LIN_ANNOT s' a)$(λ$_2$σ'. c' σ')$(λ$_2$x' σ'. f' x' σ')$(σ'^{L})
)"
unfolding *APP_def PROTECT2_def LIN_ANNOT_def PR_CONST_def*
apply *(rule hn_refine_ref[OF monadic_FOREACH_itsl])*
apply *(rule hn_refine_guessI)*
apply *(rule hnr_bind)*
apply *(rule hn_refine_cons_pre[OF FR])*
apply *(rule hn_refine_frame)*
apply *(rule hn_itsl[OF STL[unfolded GEN_ALGO_tag_def]])*
apply *(tactic {* Sepref_Frame.frame_tac @{context} 1*})*
apply *(rule hn_monadic_nfoldli[unfolded APP_def])*
apply *(simp add: pure_hn_list_eq_list_rel)*
apply *(tactic {* Sepref_Frame.frame_tac @{context} 1*})*
apply *(rule hn_refine_cons_post)*
apply *(rule c_ref[unfolded APP_def])*
apply *(rule C_FR)*
apply *(rule TERMI)*
apply *(rule hn_refine_cons_post)*

```
  apply (rule f_ref[unfolded APP_def])
  apply (rule ent_trans[OF F_FR])
  apply (rule TERMI)
  apply (tactic {* Sepref_Frame.frame_tac @{context} 1*})
  apply (tactic {* Sepref_Frame.frame_tac @{context} 1*})
  apply simp
  done



lemma heap_fixp_mono[partial_function_mono]:
  assumes [partial_function_mono]:
    "⋀x d. mono_Heap (λxa. B x xa d)"
    "⋀Z xa. mono_Heap (λa. B a Z xa)"
  shows "mono_Heap (λx. heap.fixp_fun (λD σ. B x D σ) σ)"
  apply rule
  apply (rule ccpo.fixp_mono[OF heap.ccpo, THEN fun_ordD])
  apply (rule mono_fun_fun_cnv,
    erule thin_rl, tactic {* Pf_Mono_Prover.mono_tac @{context} 1 *})+
  apply (rule fun_ordI)
  apply (erule monotoneD[of "fun_ord Heap_ord" Heap_ord, rotated])
  apply (tactic {* Pf_Mono_Prover.mono_tac @{context} 1 *})
  done

lemma imp_nfoldli_mono[partial_function_mono]:
  assumes [partial_function_mono]: "⋀x σ. mono_Heap (λfa. f fa x σ)"
  shows "mono_Heap (λx. imp_nfoldli l c (f x) σ)"
  unfolding imp_nfoldli_def
  by (tactic {* Pf_Mono_Prover.mono_tac @{context} 1 *})


declare imp_nfoldli_def[sepref_opt_simps]

end
theory Sepref
imports
  Sepref_Tool
  Sepref_HOL_Bindings
  Sepref_IICF_Bindings
  Sepref_Foreach
begin


end

theory DFS_Framework_Misc
imports Misc
begin
```

abbreviation *comp2* (**infixl** *"oo" 55*) **where** *"f oo g ≡ λx. f o (g x)"*
abbreviation *comp3* (**infixl** *"ooo" 55*) **where** *"f ooo g ≡ λx. f oo (g x)"*

notation (*xsymbols*)
  comp2 (**infixl** *"∘∘" 55*) **and**
  comp3 (**infixl** *"∘∘∘" 55*)

notation (*HTML* **output**)
  comp2 (**infixl** *"∘∘" 55*) **and**
  comp3 (**infixl** *"∘∘∘" 55*)

lemma *tri_caseE:*
  "⟦⟦¬P;¬Q⟧ ⟹ R; P ⟹ R; ⟦¬P; Q⟧ ⟹ R⟧ ⟹ R"
by *auto*

definition *"opt_tag x y ≡ x=y"*
lemma *opt_tagI: "opt_tag x x"* **unfolding** *opt_tag_def* **by** *simp*
lemma *opt_tagD: "opt_tag x y ⟹ x=y"* **unfolding** *opt_tag_def* **by** *simp*

lemma *map_update_eta_repair[simp]:*

  "dom (λx. if x=k then Some v else m x) = insert k (dom m)"
  "m k = None ⟹ ran (λx. if x=k then Some v else m x) = insert v (ran m)"
  apply *auto []*
  apply (*force simp: ran_def*)
  done

  lemma *subset_Collect_conv: "S ⊆ Collect P ⟷ (∀x∈S. P x)"*
    by *auto*

  lemma *memb_imp_not_empty: "x∈S ⟹ S≠{}"*
    by *auto*

definition *"bijective R ≡*
  (∀x y z. (x,y)∈R ∧ (x,z)∈R ⟶ y=z) ∧

```
  (∀x y z. (x,z)∈R ∧ (y,z)∈R ⟶ x=y)"

lemma bijective_alt: "bijective R ⟷ single_valued R ∧ single_valued
(R⁻¹)"
  unfolding bijective_def single_valued_def by blast

lemma bijective_Id[simp, intro!]: "bijective Id"
  by (auto simp: bijective_def)

lemma bijective_Empty[simp, intro!]: "bijective {}"
  by (auto simp: bijective_def)


definition "fun_of_rel R x ≡ SOME y. (x,y)∈R"

lemma for_in_RI: "x∈Domain R ⟹ (x,fun_of_rel R x)∈R"
  unfolding fun_of_rel_def
  by (auto intro: someI)

lemma finite_Field_eq_finite[simp]: "finite (Field R) ⟷ finite R"
  by (metis finite_cartesian_product finite_subset R_subset_Field finite_Field)


lemma trancl_Image_unfold_left: "E⁺``S = E*``E``S"
  by (auto simp: trancl_unfold_left)

lemma trancl_Image_unfold_right: "E⁺``S = E``E*``S"
  by (auto simp: trancl_unfold_right)

lemma trancl_Image_advance_ss: "(u,v)∈E ⟹ E⁺``{v} ⊆ E⁺``{u}"
  by auto

lemma rtrancl_Image_advance_ss: "(u,v)∈E ⟹ E*``{v} ⊆ E*``{u}"
  by auto



lemma rtrancl_restrictI_aux:
  assumes "(u,v)∈(E-UNIV×R)*"
  assumes "u∉R"
  shows "(u,v)∈(rel_restrict E R)* ∧ v∉R"
  using assms
  by (induction) (auto simp: rel_restrict_def intro: rtrancl.intros)

corollary rtrancl_restrictI:
  assumes "(u,v)∈(E-UNIV×R)*"
  assumes "u∉R"
  shows "(u,v)∈(rel_restrict E R)*"
  using rtrancl_restrictI_aux[OF assms] ..
```

```
lemma E_closed_restr_reach_cases:
  assumes P: "(u,v)∈E*"
  assumes CL: "E``R ⊆ R"
  obtains "v∈R" | "u∉R" "(u,v)∈(rel_restrict E R)*"
  using P
proof (cases rule: rtrancl_last_visit[where S=R])
  case no_visit
  show ?thesis proof (cases "u∈R")
    case True with P have "v∈R"
      using rtrancl_reachable_induct[OF _ CL, where I="{u}"]
      by auto
    thus ?thesis ..
  next
    case False with no_visit have "(u,v)∈(rel_restrict E R)*"
      by (rule rtrancl_restrictI)
    with False show ?thesis ..
  qed
next
  case (last_visit_point x)
  from ‹(x, v) ∈ (E - UNIV × R)*› have "(x,v)∈E*"
    by (rule rtrancl_mono_mp[rotated]) auto
  with ‹x∈R› have "v∈R"
    using rtrancl_reachable_induct[OF _ CL, where I="{x}"]
    by auto
  thus ?thesis ..
qed

lemma rel_restrict_trancl_notR:
  assumes "(v,w) ∈ (rel_restrict E R)+"
  shows "v ∉ R" and "w ∉ R"
  using assms
  by (metis rel_restrict_trancl_mem rel_restrict_notR)+

lemma rel_restrict_tranclI:
  assumes "(x,y) ∈ E+"
  and "x ∉ R" "y ∉ R"
  and "E `` R ⊆ R"
  shows "(x,y) ∈ (rel_restrict E R)+"
  using assms
  proof (induct)
    case base thus ?case by (metis r_into_trancl rel_restrictI)
  next
    case (step y z) hence "y ∉ R" by auto
    with step show ?case by (metis trancl_into_trancl rel_restrictI)
  qed

lemma trancl_sub:
  "R ⊆ R+"
```

```
by auto

lemma trancl_single[simp]:
  "{(a,b)}⁺ = {(a,b)}"
proof -
  {
    fix x y
    assume "(x,y) ∈ {(a,b)}⁺"
    hence "(x,y) ∈ {(a,b)}"
      by (induct rule: trancl.induct) auto
  }
  with trancl_sub show ?thesis by auto
qed


lemma trancl_union_outside:
  assumes "(v,w) ∈ (E∪U)⁺"
  and "(v,w) ∉ E⁺"
  shows "∃x y. (v,x) ∈ (E∪U)* ∧ (x,y) ∈ U ∧ (y,w) ∈ (E∪U)*"
using assms
proof (induction)
  case base thus ?case by auto
next
  case (step w x)
  show ?case
  proof (cases "(v,w)∈E⁺")
    case True
    from step have "(v,w)∈(E∪U)*" by simp
    moreover from True step have "(w,x) ∈ U" by (metis Un_iff trancl.simps)
    moreover have "(x,x) ∈ (E∪U)*" by simp
    ultimately show ?thesis by blast
  next
    case False with step.IH obtain a b where "(v,a) ∈ (E∪U)*" "(a,b)
∈ U" "(b,w) ∈ (E∪U)*" by blast
    moreover with step have "(b,x) ∈ (E∪U)*" by (metis rtrancl_into_rtrancl)
    ultimately show ?thesis by blast
  qed
qed

lemma trancl_restrict_reachable:
  assumes "(u,v) ∈ E⁺"
  assumes "E``S ⊆ S"
  assumes "u∈S"
  shows "(u,v) ∈ (E∩S×S)⁺"
  using assms
  by (induction rule: converse_trancl_induct)
     (auto intro: trancl_into_trancl2)

lemma rtrancl_image_unfold_right: "E``E*``V ⊆ E*``V"
```

```
  by (auto intro: rtrancl_into_rtrancl)

lemma Field_not_elem:
  "v ∉ Field R ⟹ ∀(x,y) ∈ R. x ≠ v ∧ y ≠ v"
unfolding Field_def by auto

lemma trancl_Image_in_Range:
  "R⁺ `` V ⊆ Range R"
by (auto elim: trancl.induct)

lemma rtrancl_Image_in_Field:
  "R* `` V ⊆ Field R ∪ V"
proof -
  from trancl_Image_in_Range have "R⁺ `` V ⊆ Field R"
    unfolding Field_def by fast
  hence "R⁺ `` V ∪ V ⊆ Field R ∪ V" by blast
  with trancl_image_by_rtrancl show ?thesis by metis
qed

lemma rtrancl_sub_insert_rtrancl:
  "R* ⊆ (insert x R)*"
by (auto elim: rtrancl.induct rtrancl_into_rtrancl)

lemma trancl_sub_insert_trancl:
  "R⁺ ⊆ (insert x R)⁺"
by (auto elim: trancl.induct trancl_into_trancl)

lemma Restr_rtrancl_mono:
  "(v,w) ∈ (Restr E U)* ⟹ (v,w) ∈ E*"
  by (metis inf_le1 rtrancl_mono subsetCE)

lemma Restr_trancl_mono:
  "(v,w) ∈ (Restr E U)⁺ ⟹ (v,w) ∈ E⁺"
  by (metis inf_le1 trancl_mono)

lemma Sigma_UNIV_cancel[simp]: "(A × X - A × UNIV) = {}" by auto

lemma cyclic_subset:
  "⟦ ¬ acyclic R; R ⊆ S ⟧ ⟹ ¬ acyclic S"
  unfolding acyclic_def
  by (blast intro: trancl_mono)


lemma in_hd_or_tl_conv[simp]: "l≠[] ⟹ x=hd l ∨ x∈set (tl l) ⟷
x∈set l"
  by (cases l) auto

lemma rev_split_conv[simp]:
  "l ≠ [] ⟹ rev (tl l) @ [hd l] = rev l"
```

```
by (induct l) simp_all

lemma neq_Nil_rev_conv: "l≠[] ⟷ (∃xs x. l = xs@[x])"
  by (cases l rule: rev_cases) auto

lemma rev_butlast_is_tl_rev: "rev (butlast l) = tl (rev l)"
  by (induct l) auto

lemma hd_last_singletonI:
  "⟦xs ≠ []; hd xs = last xs; distinct xs⟧ ⟹ xs = [hd xs]"
  by (induct xs rule: list_nonempty_induct) auto

lemma last_filter:
  "⟦xs ≠ []; P (last xs)⟧ ⟹ last (filter P xs) = last xs"
  by (induct xs rule: rev_nonempty_induct) simp_all

lemma length_dropWhile_takeWhile:
  assumes "x < length (dropWhile P xs)"
  shows "x + length (takeWhile P xs) < length xs"
  using assms
  by (induct xs) auto


lemma rev_induct2' [case_names empty snocl snocr snoclr]:
  assumes empty: "P [] []"
  and snocl: "⋀x xs. P (xs@[x]) []"
  and snocr: "⋀y ys. P [] (ys@[y])"
  and snoclr: "⋀x xs y ys.  P xs ys  ⟹ P (xs@[x]) (ys@[y])"
  shows "P xs ys"
proof (induct xs arbitrary: ys rule: rev_induct)
  case Nil thus ?case using empty snocr
    by (cases ys rule: rev_exhaust) simp_all
next
  case snoc thus ?case using snocl snoclr
    by (cases ys rule: rev_exhaust) simp_all
qed

lemma rev_nonempty_induct2' [case_names single snocl snocr snoclr, consumes
2]:
  assumes "xs ≠ []" "ys ≠ []"
  assumes single': "⋀x y. P [x] [y]"
  and snocl: "⋀x xs y. xs ≠ [] ⟹ P (xs@[x]) [y]"
  and snocr: "⋀x y ys. ys ≠ [] ⟹ P [x] (ys@[y])"
  and snoclr: "⋀x xs y ys. ⟦P xs ys; xs ≠ []; ys≠[]⟧  ⟹ P (xs@[x])
(ys@[y])"
  shows "P xs ys"
  using assms(1,2)
proof (induct xs arbitrary: ys rule: rev_nonempty_induct)
  case single then obtain z zs where "ys = zs@[z]" by (metis rev_exhaust)
```

```
  thus ?case using single' snocr
    by (cases "zs = []") simp_all
next
  case (snoc x xs) then obtain z zs where zs: "ys = zs@[z]" by (metis
rev_exhaust)
  thus ?case using snocl snoclr snoc
    by (cases "zs = []") simp_all
qed

end

theory DFS_Framework_Refine_Aux
imports "../../Libs/CAVA_Automata/CAVA_Base/CAVA_Base" DFS_Framework_Misc
begin
```

General casting-tag, that allows type-casting on concrete level, while being identity on abstract level.

```
definition [simp]: "CAST ≡ id"
lemma [autoref_itype]: "CAST ::ᵢ I →ᵢ I" by simp

attribute_setup zero_var_indexes = {*
  Scan.succeed (Thm.rule_attribute (K Drule.zero_var_indexes))
*} "Set variable indexes to zero, renaming to avoid clashes"

lemma mk_record_simp_thm:
  fixes f :: "'a ⇒ 'b"
  assumes "f s = x"
  assumes "r ≡ s"
  shows "f r = x"
  using assms by simp

ML {*
  fun mk_record_simp thm = let
    val thy = theory_of_thm thm
    val cert = cterm_of thy
  in
    case concl_of thm of
      @{mpat "Trueprop (?f _=_)"} =>
        let
```

```
          val cf = cert f
          val r = cterm_instantiate
            [(@{cpat "?f :: ?'a ⇒ ?'b"},cf)]
            @{thm mk_record_simp_thm}
          val r = r OF [thm]
        in r end
    | _ => raise THM("",~1,[thm])

  end
*}

attribute_setup mk_record_simp =
  {* Scan.succeed (Thm.rule_attribute (K mk_record_simp)) *}
  "Make simplification rule for record definition"
```

lemma *flat_ge_sup_mono[refine_mono]: "⋀a a'::'a::complete_lattice.*
flat_ge a a' ⟹ flat_ge b b' ⟹ flat_ge (sup a b) (sup a' b')"
by *(auto simp: flat_ord_def)*

declare *sup_mono[refine_mono]*

lemma *nofail_RES_conv: "nofail m ⟷ (∃M. m=RES M)"* **by** *(cases m) auto*

lemma *nofail_SPEC: "nofail m ⟹ m ≤ SPEC (λ_. True)"*
by *(simp add: pw_le_iff)*

lemma *nofail_SPEC_iff: "nofail m ⟷ m ≤ SPEC (λ_. True)"*
by *(simp add: pw_le_iff)*

lemma *nofail_SPEC_triv_refine: "⟦ nofail m; ⋀x. Φ x ⟧ ⟹ m ≤ SPEC Φ"*
by *(simp add: pw_le_iff)*

lemma *bind_cong:*
assumes *"m=m'"*
assumes *"⋀x. RETURN x ≤ m' ⟹ f x = f' x"*
shows *"bind m f = bind m' f'"*
using *assms*
by *(auto simp: refine_pw_simps pw_eq_iff pw_le_iff)*

primrec *the_RES* **where** *"the_RES (RES X) = X"*
lemma *the_RES_inv[simp]: "nofail m ⟹ RES (the_RES m) = m"*
by *(cases m) auto*

lemma *le_SPEC_UNIV_rule [refine_vcg]:*
"m ≤ SPEC (λ_. True) ⟹ m ≤ RES UNIV" **by** *auto*

definition *[refine_pw_simps]: "nf_inres m x ≡ nofail m ∧ inres m x"*

lemma *nf_inres_RES[simp]: "nf_inres (RES X) x ⟷ x∈X"*
by *(simp add: refine_pw_simps)*

lemma *nf_inres_SPEC[simp]: "nf_inres (SPEC Φ) x ⟷ Φ x"*
by *(simp add: refine_pw_simps)*

lemma *bind_refine_abs':*
fixes *R' :: "('a×'b) set"* **and** *R::"('c×'d) set"*
assumes *R1: "M ≤ ⇓ R' M'"*
assumes *R2: "⋀x x'. ⟦ (x,x')∈R'; nf_inres M' x'*
⟧ ⟹ f x ≤ ⇓ R (f' x')"
shows *"bind M (λx. f x) ≤ ⇓ R (bind M' (λx'. f' x'))"*
using *assms*
apply *(simp add: pw_le_iff refine_pw_simps)*
apply *blast*
done

lemma *Let_refine':*
assumes *"(m,m')∈R"*
assumes *"(m,m')∈R ⟹ f m ≤⇓S (f' m')"*
shows *"Let m f ≤ ⇓S (Let m' f')"*
using *assms* **by** *simp*

lemma *RETURN_refine_iff[simp]: "RETURN x ≤⇓R (RETURN y) ⟷ (x,y)∈R"*
by *(auto simp: pw_le_iff refine_pw_simps)*

lemma *RETURN_RES_refine_iff:*
"RETURN x ≤⇓R (RES Y) ⟷ (∃y∈Y. (x,y)∈R)"
by *(auto simp: pw_le_iff refine_pw_simps)*

lemma *in_nres_rel_iff: "(a,b)∈⟨R⟩nres_rel ⟷ a ≤⇓R b"*
by *(auto simp: nres_rel_def)*

lemma *inf_RETURN_RES:*

```
  "inf (RETURN x) (RES X) = (if x∈X then RETURN x else SUCCEED)"
  "inf (RES X) (RETURN x) = (if x∈X then RETURN x else SUCCEED)"
  by (auto simp: pw_eq_iff refine_pw_simps)


lemma inf_RETURN_SPEC[simp]:
  "inf (RETURN x) (SPEC (λy. Φ y)) = SPEC (λy. y=x ∧ Φ x)"
  "inf (SPEC (λy. Φ y)) (RETURN x) = SPEC (λy. y=x ∧ Φ x)"
  by (auto simp: pw_eq_iff refine_pw_simps)

lemma RES_sng_eq_RETURN: "RES {x} = RETURN x"
  by simp

lemma nofail_inf_serialize:
  "⟦nofail a; nofail b⟧ ⟹ inf a b = do {x←a; ASSUME (inres b x); RETURN
x}"
  by (auto simp: pw_eq_iff refine_pw_simps)

lemma conc_fun_SPEC:
  "⇓R (SPEC (λx. Φ x)) = SPEC (λy. ∃x. (y,x)∈R ∧ Φ x)"
  by (auto simp: pw_eq_iff refine_pw_simps)

lemma conc_fun_RETURN:
  "⇓R (RETURN x) = SPEC (λy. (y,x)∈R)"
  by (auto simp: pw_eq_iff refine_pw_simps)



definition lift_assn :: "('a × 'b) set ⇒ ('b ⇒ bool) ⇒ ('a ⇒ bool)"
  — Lift assertion over refinement relation
  where "lift_assn R Φ s ≡ ∃s'. (s,s')∈R ∧ Φ s'"
lemma lift_assnI: "⟦(s,s')∈R; Φ s'⟧ ⟹ lift_assn R Φ s"
  unfolding lift_assn_def by auto


lemma case_option_refine[refine]:
  assumes "(x,x')∈Id"
  assumes "x=None ⟹ fn ≤ ⇓R fn'"
  assumes "⋀v v'. ⟦x=Some v; (v,v')∈Id⟧ ⟹ fs v ≤ ⇓R (fs' v')"
  shows "case_option fn (λv. fs v) x ≤ ⇓R (case_option fn' (λv'. fs'
v') x')"
  using assms by (auto split: option.split)


definition GHOST :: "'a ⇒ 'a"
  — Ghost tag to mark ghost variables in let-expressions
  where [simp]: "GHOST ≡ λx. x"
lemma GHOST_elim_Let: — Unfold rule to inline GHOST-Lets
  shows "(let x=GHOST m in f x) = f m" by simp
```

The following set of rules executes a step on the LHS or RHS of a refinement proof obligation, without changing the other side. These kind of rules is useful for performing refinements with invisible steps.

lemma *lhs_step_If:*
```
"⟦ b ⟹ t ≤ m; ¬b ⟹ e ≤ m ⟧ ⟹ If b t e ≤ m" by simp
```

lemma *lhs_step_RES:*
```
"⟦ ⋀x. x∈X ⟹ RETURN x ≤ m ⟧ ⟹ RES X ≤ m"
by (simp add: pw_le_iff)
```

lemma *lhs_step_SPEC:*
```
"⟦ ⋀x. Φ x ⟹ RETURN x ≤ m ⟧ ⟹ SPEC (λx. Φ x) ≤ m"
by (simp add: pw_le_iff)
```

lemma *lhs_step_bind:*
fixes `m :: "'a nres"` **and** `f :: "'a ⇒ 'b nres"`
assumes `"nofail m' ⟹ nofail m"`
assumes `"⋀x. nf_inres m x ⟹ f x ≤ m'"`
shows `"do {x←m; f x} ≤ m'"`
using *assms*
by *(simp add: pw_le_iff refine_pw_simps) blast*

lemma *rhs_step_bind:*
assumes `"m ≤ ⇓R m'" "inres m x" "⋀x'. (x,x')∈R ⟹ lhs ≤⇓S (f' x')"`
shows `"lhs ≤ ⇓S (m' ≫= f')"`
using *assms*
by *(simp add: pw_le_iff refine_pw_simps) blast*

lemma *rhs_step_bind_RES:*
assumes `"x'∈X'"`
assumes `"m ≤ ⇓R (f' x')"`
shows `"m ≤ ⇓R (RES X' ≫= f')"`
using *assms* **by** *(simp add: pw_le_iff refine_pw_simps) blast*

lemma *rhs_step_bind_SPEC:*
assumes `"Φ x'"`
assumes `"m ≤ ⇓R (f' x')"`
shows `"m ≤ ⇓R (SPEC Φ ≫= f')"`
using *assms* **by** *(simp add: pw_le_iff refine_pw_simps) blast*

lemma *RES_bind_choose:*
assumes `"x∈X"`
assumes `"m ≤ f x"`
shows `"m ≤ RES X ≫= f"`
using *assms* **by** *(auto simp: pw_le_iff refine_pw_simps)*

lemma *pw_RES_bind_choose:*
```
"nofail (RES X ≫= f) ⟷ (∀x∈X. nofail (f x))"
"inres (RES X ≫= f) y ⟷ (∃x∈X. inres (f x) y)"
```

by *(auto simp: refine_pw_simps)*

lemma *use_spec_rule:*
assumes *"m ≤ SPEC Ψ"*
assumes *"m ≤ SPEC (λs. Ψ s ⟶ Φ s)"*
shows *"m ≤ SPEC Φ"*
using *assms*
by *(auto simp: pw_le_iff refine_pw_simps)*

lemma *strengthen_SPEC: "m ≤ SPEC Φ ⟹ m ≤ SPEC(λs. inres m s ∧ Φ s)"*
— Strengthen SPEC by adding trivial upper bound for result
by *(auto simp: pw_le_iff refine_pw_simps)*

lemma *weaken_SPEC:*
"m ≤ SPEC Φ ⟹ (⋀x. Φ x ⟹ Ψ x) ⟹ m ≤ SPEC Ψ"
by *(force elim!: order_trans)*

lemma *ife_FAIL_to_ASSERT_cnv:*
"(if Φ then m else FAIL) = op_nres_ASSERT_bnd Φ m"
by *(cases Φ, auto)*

lemma *param_op_nres_ASSERT_bnd[param]:*
assumes *"Φ' ⟹ Φ"*
assumes *"⟦Φ'; Φ⟧ ⟹ (m,m')∈⟨R⟩nres_rel"*
shows *"(op_nres_ASSERT_bnd Φ m, op_nres_ASSERT_bnd Φ' m') ∈ ⟨R⟩nres_rel"*
using *assms*
by *(auto simp: pw_le_iff refine_pw_simps nres_rel_def)*

declare *autoref_FAIL[param]*

method_setup *refine_vcg =*
{ Attrib.thms >> (fn add_thms => fn ctxt => SIMPLE_METHOD' (*
Refine.rcg_tac (add_thms @ Refine.vcg.get ctxt) ctxt THEN_ALL_NEW (TRY o Refine.post_tac ctxt)
*)) *}*
"Refinement framework: Generate refinement and verification conditions"

```
lemma (in transfer) transfer_sum[refine_transfer]:
  assumes "⋀l. α (fl l) ≤ Fl l"
  assumes "⋀r. α (fr r) ≤ Fr r"
  shows "α (case_sum fl fr x) ≤ (case_sum Fl Fr x)"
  using assms by (auto split: sum.split)


lemma nres_of_transfer[refine_transfer]: "nres_of x ≤ nres_of x" by
simp




declare FOREACH_patterns[autoref_op_pat_def]



theorem param_RECT[param]:
  assumes "(B, B') ∈ (Ra → ⟨Rr⟩nres_rel) → Ra → ⟨Rr⟩nres_rel"
    and "trimono B"
  shows "(REC_T B, REC_T B')∈ Ra → ⟨Rr⟩nres_rel"
  using autoref_RECT assms by simp

definition "REC_annot pre post body x ≡
  REC (λD x. do {ASSERT (pre x); r←body D x; ASSERT (post x r); RETURN
r}) x"

theorem REC_annot_rule[refine_vcg]:
  assumes M: "trimono body"
  and P: "pre x"
  and S: "⋀f x. ⟦⋀x. pre x ⟹ f x ≤ SPEC (post x); pre x⟧
          ⟹ body f x ≤ SPEC (post x)"
  and C: "⋀r. post x r ⟹ Φ r"
  shows "REC_annot pre post body x ≤ SPEC Φ"
proof -
  from ‘trimono body‘ have [refine_mono]:
    "⋀f g x xa. (⋀x. flat_ge (f x) (g x)) ⟹ flat_ge (body f x) (body
g x)"
    "⋀f g x xa. (⋀x. f x ≤ g x) ⟹ body f x ≤ body g x"
    apply -
    unfolding trimono_def monotone_def fun_ord_def mono_def le_fun_def
    apply (auto)
    done

  show ?thesis
    unfolding REC_annot_def
```

```
    apply (rule order_trans[where y="SPEC (post x)"])
    apply (refine_rcg
      refine_vcg
      REC_rule[where pre=pre and M="λx. SPEC (post x)"]
      order_trans[OF S]
    )
    apply fact
    apply simp
    using C apply (auto) []
    done
qed
```

```
context begin interpretation autoref_syn .

lemma [autoref_op_pat_def]:
  "WHILEIT I ≡ OP (WHILEIT I)"
  "WHILEI I ≡ OP (WHILEI I)"
  by auto
end

context begin interpretation autoref_syn .


lemma autoref_WHILE'[autoref_rules]:
  assumes "⋀x x'. ⟦ (x,x')∈R⟧ ⟹ (c x,c'$x') ∈ Id"
  assumes "⋀x x'. ⟦ REMOVE_INTERNAL c' x'; (x,x')∈R⟧
    ⟹ (f x,f'$x') ∈ ⟨R⟩nres_rel"
  shows "(WHILE c f,
      (OP WHILE ::: (R→Id) → (R→⟨R⟩nres_rel) → R → ⟨R⟩nres_rel)$c'$f'
    )∈R → ⟨R⟩nres_rel"
  using assms
  by (auto simp add: nres_rel_def fun_rel_def intro!: WHILE_refine)


lemma autoref_WHILEI[autoref_rules]:
  assumes "⋀x x'. ⟦I x'; (x,x')∈R⟧ ⟹ (c x,c'$x') ∈ Id"
  assumes "⋀x x'. ⟦REMOVE_INTERNAL c' x'; I x'; (x,x')∈R⟧ ⟹(f x,f'$x')
∈ ⟨R⟩nres_rel"
  assumes "I s' ⟹ (s,s')∈R"
  shows "(WHILE c f s,
      (OP (WHILEI I) ::: (R→Id) → (R→⟨R⟩nres_rel) → R → ⟨R⟩nres_rel)$c'$f'$s'
    )∈⟨R⟩nres_rel"
  using assms
  by (auto simp add: nres_rel_def fun_rel_def intro!: WHILE_refine')
```

lemma *autoref_WHILEI'[autoref_rules]:*
  assumes *"⋀x x'. ⟦ (x,x')∈R; I x'⟧ ⟹ (c x,c'$x') ∈ Id"*
  assumes *"⋀x x'. ⟦ REMOVE_INTERNAL c' x'; (x,x')∈R; I x'⟧*
    ⟹ (f x,f'$x') ∈ ⟨R⟩nres_rel"
  shows *"(WHILE c f,*
      (OP (WHILEI I) ::: (R→Id) → (R→⟨R⟩nres_rel) → R → ⟨R⟩nres_rel)c'f'
)∈R → ⟨R⟩nres_rel"
    unfolding *autoref_tag_defs*
    by *(parametricity*
      add: autoref_WHILEI[unfolded autoref_tag_defs]
      assms[unfolded autoref_tag_defs])

lemma *autoref_WHILEIT[autoref_rules]:*
  assumes *"⋀x x'. ⟦I x'; (x,x')∈R⟧ ⟹ (c x,c'$x') ∈ Id"*
  assumes *"⋀x x'. ⟦REMOVE_INTERNAL c' x'; I x'; (x,x')∈R⟧ ⟹(f x,f'$x')*
∈ ⟨R⟩nres_rel"
  assumes *"I s' ⟹ (s,s')∈R"*
  shows *"(WHILET c f s,*
      (OP (WHILEIT I) ::: (R→Id) → (R→⟨R⟩nres_rel) → R → ⟨R⟩nres_rel)c'f'$s'
)∈⟨R⟩nres_rel"
  using *assms*
  by *(auto simp add: nres_rel_def fun_rel_def intro!: WHILET_refine')*

lemma *autoref_WHILEIT'[autoref_rules]:*
  assumes *"⋀x x'. ⟦ (x,x')∈R; I x'⟧ ⟹ (c x,c'$x') ∈ Id"*
  assumes *"⋀x x'. ⟦ REMOVE_INTERNAL c' x'; (x,x')∈R; I x'⟧*
    ⟹ (f x,f'$x') ∈ ⟨R⟩nres_rel"
  shows *"(WHILET c f,*
      (OP (WHILEIT I) ::: (R→Id) → (R→⟨R⟩nres_rel) → R → ⟨R⟩nres_rel)c'f'
)∈R → ⟨R⟩nres_rel"
    unfolding *autoref_tag_defs*
    by *(parametricity*
      add: autoref_WHILEIT[unfolded autoref_tag_defs]
      assms[unfolded autoref_tag_defs])

end

lemma *set_relD: "(s,s')∈⟨R⟩set_rel ⟹ x∈s ⟹ ∃x'∈s'. (x,x')∈R"*

unfolding *set_rel_def* **by** *blast*

lemma *set_relE[consumes 2]:*
assumes *"(s,s')∈⟨R⟩set_rel" "x∈s"*
obtains *x'* **where** *"x'∈s'" "(x,x')∈R"*
using *set_relD[OF assms]* **..**

lemma *param_prod': "⟦*
⋀a b a' b'. ⟦p=(a,b); p'=(a',b')⟧ ⟹ (f a b,f' a' b')∈R
⟧ ⟹ (case_prod f p, case_prod f' p')∈R"
by *(auto split: prod.split)*

lemma *list_rel_append1: "(as @ bs, l) ∈ ⟨R⟩list_rel*
⟷ (∃cs ds. l = cs@ds ∧ (as,cs)∈⟨R⟩list_rel ∧ (bs,ds)∈⟨R⟩list_rel)"
apply *(simp add: list_rel_def list_all2_append1)*
apply *auto*
apply *(metis list_all2_lengthD)*
done

lemma *list_rel_append2: "(l,as @ bs) ∈ ⟨R⟩list_rel*
⟷ (∃cs ds. l = cs@ds ∧ (cs,as)∈⟨R⟩list_rel ∧ (ds,bs)∈⟨R⟩list_rel)"
apply *(simp add: list_rel_def list_all2_append2)*
apply *auto*
apply *(metis list_all2_lengthD)*
done

ML *{**

```
(* Prefix p_ or wrong type supresses generation of relAPP *)

fun cnv_relAPP t = let
  fun consider (Var ((name,_),T)) =
    if String.isPrefix "p_" name then false
    else (
      case T of
        Type(@{type_name set},[Type(@{type_name prod},_)]) => true
      | _ => false)
  | consider _ = true

  fun strip_rcomb u : term * term list =
    let
      fun stripc (x as (f$t, ts)) =
        if consider t then stripc (f, t::ts) else x
      | stripc  x =  x
    in  stripc(u,[])  end;
```

```
    val (f,a) = strip_rcomb t
  in
    Relators.list_relAPP a f
  end

  fun to_relAPP_conv ctxt = Refine_Util.f_tac_conv ctxt
    cnv_relAPP
    (ALLGOALS (simp_tac
      (put_simpset HOL_basic_ss ctxt addsimps @{thms relAPP_def})))


  val to_relAPP_attr = Thm.rule_attribute (fn context => let
    val ctxt = Context.proof_of context
  in
    Conv.fconv_rule (Conv.arg1_conv (to_relAPP_conv ctxt))
  end)
*}

attribute_setup to_relAPP = {* Scan.succeed (to_relAPP_attr) *}
  "Convert relator definition to prefix-form"
```

```
lemma dropWhile_param[param]:
  "(dropWhile, dropWhile) ∈ (A → bool_rel) → ⟨A⟩list_rel → ⟨A⟩list_rel"
  unfolding dropWhile_def by parametricity

term takeWhile
lemma takeWhile_param[param]:
  "(takeWhile, takeWhile) ∈ (A → bool_rel) → ⟨A⟩list_rel → ⟨A⟩list_rel"
  unfolding takeWhile_def by parametricity


lemma [relator_props]:
  "bijective R ⟹ single_valued R"
  "bijective R ⟹ single_valued (R⁻¹)"
  by (simp_all add: bijective_alt)


declare bijective_Id[relator_props]
declare bijective_Empty[relator_props]

lemma param_subseteq[param]:
  "⟦single_valued (R⁻¹)⟧ ⟹ (op ⊆, op ⊆) ∈ ⟨R⟩set_rel → ⟨R⟩set_rel
→ bool_rel"
```

```
  by (clarsimp simp: set_rel_def single_valued_def) blast

lemma param_subset[param]:
  "⟦single_valued (R⁻¹)⟧ ⟹ (op ⊂, op ⊂) ∈ ⟨R⟩set_rel → ⟨R⟩set_rel
→ bool_rel"

  apply (simp add: set_rel_def single_valued_def)
  apply safe
  apply blast+
  done

lemma bij_set_rel_for_inj:
  fixes R
  defines "α ≡ fun_of_rel R"
  assumes "bijective R" "(s,s')∈⟨R⟩set_rel"
  shows "inj_on α s" "s' = α`s"
  using assms
  unfolding bijective_def set_rel_def α_def fun_of_rel_def[abs_def]
  apply (auto intro!: inj_onI ImageI simp: image_def)
  apply (metis (mono_tags) Domain.simps contra_subsetD tfl_some)
  apply (metis (mono_tags) someI)
  apply (metis DomainE contra_subsetD tfl_some)
  done




lemmas [autoref_rules] = dropWhile_param takeWhile_param




method_setup autoref_solve_id_op = {*
  Scan.succeed (fn ctxt => SIMPLE_METHOD' (
    Autoref_Id_Ops.id_tac (Config.put Autoref_Id_Ops.cfg_ss_id_op false
ctxt)
  ))
*}
```

Default setup of the autoref-tool for the monadic framework.

```
lemma autoref_monadicI1:
  assumes "(b,a)∈⟨R⟩nres_rel"
  assumes "RETURN c ≤ b"
  shows "(RETURN c, a)∈⟨R⟩nres_rel" "RETURN c ≤⇓R a"
  using assms
  unfolding nres_rel_def
  by simp_all
```

```
lemma autoref_monadicI2:
  assumes "(b,a)∈⟨R⟩nres_rel"
  assumes "nres_of c ≤ b"
  shows "(nres_of c, a)∈⟨R⟩nres_rel" "nres_of c ≤ ⇓R a"
  using assms
  unfolding nres_rel_def
  by simp_all

lemmas autoref_monadicI = autoref_monadicI1 autoref_monadicI2

ML {*
  structure Autoref_Monadic = struct

    val cfg_plain = Attrib.setup_config_bool @{binding autoref_plain}
(K false)

    fun autoref_monadic_tac ctxt = let
      open Autoref_Tacticals
      val ctxt = Autoref_Phases.init_data ctxt
      val plain = Config.get ctxt cfg_plain
      val trans_thms = if plain then [] else @{thms the_resI}

    in
      resolve_tac @{thms autoref_monadicI}
      THEN'
      IF_SOLVED (Autoref_Phases.all_phases_tac ctxt)
        (RefineG_Transfer.post_transfer_tac trans_thms ctxt)
        (K all_tac) (* Autoref failed *)

    end
  end
*}

method_setup autoref_monadic = {* let
    open Refine_Util Autoref_Monadic
    val autoref_flags =
          parse_bool_config "trace" Autoref_Phases.cfg_trace
      ||  parse_bool_config "debug" Autoref_Phases.cfg_debug
      ||  parse_bool_config "plain" Autoref_Monadic.cfg_plain

  in
    parse_paren_lists autoref_flags
    >>
    ( fn _ => fn ctxt => SIMPLE_METHOD' (
      let
        val ctxt = Config.put Autoref_Phases.cfg_keep_goal true ctxt
      in autoref_monadic_tac ctxt end
    ))
```

```
  end

 *}
 "Automatic Refinement and Determinization for the Monadic Refinement
Framework"


lemma dres_unit_simps[refine_transfer_post_simp]:
  "dbind (dRETURN (u::unit)) f = f ()"
  by auto

lemma Let_dRETURN_simp[refine_transfer_post_simp]:
  "Let m dRETURN = dRETURN m" by auto


lemmas [refine_transfer_post_simp] = dres_monad_laws




lemma le_ASSERT_defI1:
  assumes "c ≡ do {ASSERT Φ; m}"
  assumes "Φ ⟹ m' ≤ c"
  shows "m' ≤ c"
  using assms
  by (simp add: le_ASSERTI)

lemma refine_ASSERT_defI1:
  assumes "c ≡ do {ASSERT Φ; m}"
  assumes "Φ ⟹ m' ≤ ⇓R c"
  shows "m' ≤ ⇓R c"
  using assms
  by (simp, refine_vcg)

lemma le_ASSERT_defI2:
  assumes "c ≡ do {ASSERT Φ; ASSERT Ψ; m}"
  assumes "⟦Φ; Ψ⟧ ⟹ m' ≤ c"
  shows "m' ≤ c"
  using assms
  by (simp add: le_ASSERTI)

lemma refine_ASSERT_defI2:
  assumes "c ≡ do {ASSERT Φ; ASSERT Ψ; m}"
  assumes "⟦Φ; Ψ⟧ ⟹ m' ≤ ⇓R c"
  shows "m' ≤ ⇓R c"
  using assms
  by (simp, refine_vcg)

lemma ASSERT_le_defI:
```

assumes *"c ≡ do { ASSERT Φ; m'}"*
assumes *"Φ"*
assumes *"Φ ⟹ m' ≤ m"*
shows *"c ≤ m"*
using *assms* **by** *(auto)*

lemma *select_correct:*
"select X ≤ SPEC (λr. case r of None ⇒ X={} | Some x ⇒ x∈X)"
unfolding *select_def*
apply *(refine_rcg refine_vcg)*
by *auto*

definition *"prod_bhc bhc1 bhc2 ≡ λn (a,b). (bhc1 n a * 33 + bhc2 n b) mod n"*

lemma *prod_bhc_ga[autoref_ga_rules]:*
"⟦ GEN_ALGO_tag (is_bounded_hashcode R eq1 bhc1);
GEN_ALGO_tag (is_bounded_hashcode S eq2 bhc2)⟧
⟹ is_bounded_hashcode (R ×$_r$ S) (prod_eq eq1 eq2) (prod_bhc bhc1 bhc2)"
unfolding *is_bounded_hashcode_def prod_bhc_def prod_eq_def[abs_def]*
apply *safe*
apply *(auto dest: fun_relD)*
done

lemma *prod_dhs_ga[autoref_ga_rules]:*
"⟦ GEN_ALGO_tag (is_valid_def_hm_size TYPE('a) n1);
GEN_ALGO_tag (is_valid_def_hm_size TYPE('b) n2) ⟧
⟹ is_valid_def_hm_size TYPE('a'b) (n1+n2)"*
unfolding *is_valid_def_hm_size_def* **by** *auto*

abbreviation *ahs_rel* **where** *"ahs_rel bhc ≡ (map2set_rel (ahm_rel bhc))"*

end

13 Imperative Graph Representation

theory *Sepref_Graph*
imports
"../Sepref"
"../../../DFS_Framework/Misc/DFS_Framework_Refine_Aux"
begin

Nodes are identified by numbers from $0..<n$, graphs are represented as adjacency lists

type_synonym *graph_impl = "(nat list) Heap.array"*

definition *[to_relAPP]: "is_graph R G Gi ≡*
*∃A l. Gi ↦a l * ↑(*
R = nat_rel ∧
Domain G ⊆ {0..<length l} ∧
(∀ v<length l. (l!v,G‘‘{v})∈⟨nat_rel⟩list_set_rel)
)"

definition *succi :: "graph_impl ⇒ nat ⇒ nat list Heap"*
where *"succi G v = do {*
l ← Array.len G;
if v<l then do {
r ← Array.nth G v;
return r
} else return []
}"

lemma *succi_rule[sep_heap_rules]: "*
< is_graph R G Gi >
succi Gi v
*< λr. is_graph R G Gi * ↑((r,G‘‘{v})∈⟨nat_rel⟩list_set_rel) >"*
unfolding *is_graph_def succi_def*
apply *(sep_auto)*
apply *(subst ne_dom_imp_succs_empty)*
apply *(auto simp: list_set_autoref_empty)*
done

term *slg_succs*

lemma *hnr_op_succi[sepref_fr_rules]: "*
hn_refine
*(hn_ctxt (is_graph R) G Gi * hn_val nat_rel v vi)*
(succi Gi vi)
*(hn_ctxt (is_graph R) G Gi * hn_val nat_rel v vi)*
(pure (⟨nat_rel⟩list_set_rel))

```
    (RETURN$(slg_succs$G$v))"
  apply rule
  unfolding hn_ctxt_def pure_def
  by (sep_auto simp: list_set_autoref_empty)

definition cr_graph
  :: "nat ⇒ (nat × nat) list ⇒ graph_impl Heap"
where
  "cr_graph numV Es ≡ do {
    a ← Array.new numV [];
    a ← imp_nfoldli Es (λ_. return True) (λ(u,v) a. do {
      l ← Array.nth a u;
      let l = v#l;
      a ← Array.upd u l a;
      return a
    }) a;
    return a
  }"



export_code cr_graph checking SML_imp


end
```

14 Simple DFS Algorithm

```
theory Sepref_DFS
imports
  "../../CAVA_Automata/CAVA_Base/CAVA_Base"
  "../../../DFS_Framework/Misc/DFS_Framework_Refine_Aux"
  "../Sepref"
  Sepref_Graph
begin
```

We define a simple DFS-algorithm, prove a simple correctness property, and do data refinement to an efficient implementation.

14.1 Definition

Recursive DFS-Algorithm. `E` is the edge relation of the graph, `vd` the node to search for, and `v0` the start node. Already explored nodes are stored in `V`.

```
context
  fixes E :: "'v rel" and v0 :: 'v and tgt :: 'v
begin
```

```
definition dfs :: "bool nres" where
  "dfs ≡ do {
    (_,r) ← RECT (λdfs (V,v).
      if v∈V then RETURN (V,False)
      else do {
        let V=insert v V;
        if v = tgt then
          RETURN (V,True)
        else
          FOREACH_C (E``{v}) (λ(_,b). ¬b) (λv' (V,_). dfs (V,v')) (V,False)
      }
    ) ({},v0);
    RETURN r
  }"

definition "reachable ≡ {v. (v0,v)∈E*}"

lemma dfs_correct:
  assumes fr: "finite reachable"
  shows "dfs ≤ SPEC (λr. r ⟷ (v0,tgt)∈E*)"
proof -
  have F: "⋀v. v∈reachable ⟹ finite (E``{v})"
    using fr
    apply (auto simp: reachable_def)
    by (metis (mono_tags) Image_singleton Image_singleton_iff
      finite_subset rtrancl.rtrancl_into_rtrancl subsetI)


  def rpre ≡ "λS (V,v).
      v∈reachable
    ∧ V⊆reachable
    ∧ S⊆V
    ∧ tgt∉V
    ∧ E``(V-S) ⊆ V"

  def rpost ≡ "λS (V,v) (V',r).
        (r⟷tgt∈V')
      ∧ V⊆V'
      ∧ v∈V'
      ∧ V'⊆reachable
      ∧ (¬r ⟶ (E``(V'-S) ⊆ V'))
    "

  def fe_inv ≡ "λS V v it (V',r).
      (r⟷tgt∈V')
    ∧ insert v V⊆V'
    ∧ E``{v} - it ⊆ V'
    ∧ V'⊆reachable
    ∧ S⊆insert v V
```

```
  ∧ (¬r ⟶ E``(V'-S) ⊆ V' ∪ it ∧ E``(V'-insert v S) ⊆ V')"

have vc_pre_initial: "rpre {} ({}, v0)"
  by (auto simp: rpre_def reachable_def)

{

  fix S V v
  assume "rpre S (V,v)"
    and "v∈V"
  hence "rpost S (V,v) (V,False)"
    unfolding rpre_def rpost_def
    by auto
} note vc_node_visited = this

{

  fix S V
  assume "rpre S (V,tgt)"
  hence "rpost S (V,tgt) (insert tgt V, True)"
    unfolding rpre_def rpost_def
    by auto
} note vc_node_found = this

{
  fix S V v
  assume "rpre S (V, v)"
  hence "finite (E``{v})"
    unfolding rpre_def using F by (auto)
} note vc_foreach_finite = this

{

  fix S V v
  assume A: "v ∉ V" "v ≠ tgt"
    and PRE: "rpre S (V, v)"
  hence "fe_inv S V v (E``{v}) (insert v V, False)"
    unfolding fe_inv_def rpre_def by (auto)
} note vc_enter_foreach = this

{

  fix S V v v' it V'
  assume A: "v ∉ V" "v ≠ tgt" "v' ∈ it" "it ⊆ E``{v}"
    and FEI: "fe_inv S V v it (V', False)"
    and PRE: "rpre S (V, v)"

  from A have "v' ∈ E``{v}" by auto
  moreover from PRE have "v ∈ reachable" by (auto simp: rpre_def)
```

```
  hence "E``{v} ⊆ reachable" by (auto simp: reachable_def)
  ultimately have [simp]: "v'∈reachable" by blast

  have "rpre (insert v S) (V', v')"
    unfolding rpre_def
    using FEI PRE by (auto simp: fe_inv_def rpre_def) []
} note vc_rec_pre = this

{

  fix S V V' v v' it Vr''
  assume "fe_inv S V v it (V', False)"
    and "rpost (insert v S) (V', v') Vr''"
  hence "fe_inv S V v (it - {v'}) Vr''"
    unfolding rpre_def rpost_def fe_inv_def
    by clarsimp blast
} note vc_iterate_foreach = this

{

  fix S V v V'
  assume PRE: "rpre S (V, v)"
  assume A: "v ∉ V" "v ≠ tgt"
  assume FEI: "fe_inv S V v {} (V', False)"
  have "rpost S (V, v) (V', False)"
    unfolding rpost_def
    using FEI by (auto simp: fe_inv_def) []
} note vc_foreach_completed_imp_post = this

{

  fix S V v V' it
  assume PRE: "rpre S (V, v)"
    and A: "v ∉ V" "v ≠ tgt" "it ⊆ E``{v}"
    and FEI: "fe_inv S V v it (V', True)"
  hence "rpost S (V, v) (V', True)"
    by (auto simp add: rpre_def rpost_def fe_inv_def) []
} note vc_foreach_interrupted_imp_post = this

{
  fix V r
  assume "rpost {} ({}, v0) (V, r)"
  hence "r = ((v0, tgt) ∈ E*)"
    by (auto
      simp: rpost_def reachable_def
      dest: Image_closed_trancl
      intro: rev_ImageI)
} note vc_rpost_imp_spec = this
```

```
    show ?thesis
      unfolding dfs_def
      apply (refine_rcg refine_vcg)
      apply (rule order_trans)

      apply(rule RECT_rule_arb'[where
          pre=rpre
          and M="λa x. SPEC (rpost a x)"
          and V="finite_psupset reachable <*lex*> {}"
          ])
      apply refine_mono
      apply (blast intro: fr)
      apply (rule vc_pre_initial)

      apply (refine_rcg refine_vcg
        FOREACHc_rule'[where I="fe_inv S v s" for S v s]
        )
      apply (simp_all add: vc_node_visited vc_node_found)

      apply (simp add: vc_foreach_finite)

      apply (auto intro: vc_enter_foreach) []

      apply (rule order_trans)
      apply (rprems)
      apply (erule (5) vc_rec_pre)
        apply (auto simp add: fe_inv_def finite_psupset_def) []
        apply (refine_rcg refine_vcg)
        apply (simp add: vc_iterate_foreach)

      apply (auto simp add: vc_foreach_completed_imp_post) []

      apply (auto simp add: vc_foreach_interrupted_imp_post) []

      apply (auto simp add: vc_rpost_imp_spec) []
      done
    qed
end
```

14.2 Refinement to Imperative/HOL

We set up a schematic proof goal, and use the sepref-tool to synthesize the implementation.

```
schematic_lemma dfs_impl:
  fixes s t :: nat
  notes [sepref_opt_simps del] = imp_nfoldli_def
```

— Prevent the foreach-loop to be unfolded to a fixed-point, to produce more readable code for presentation purposes.

```
  notes [id_rules] =
```

```
    itypeI[of s "TYPE(nat)"]
    itypeI[of t "TYPE(nat)"]
    itypeI[of E "TYPE(nat i_slg)"]
    — Declare parameters to operation identification
  shows "hn_refine (
    hn_ctxt (is_graph nat_rel) E Ei
  * hn_val nat_rel s si
  * hn_val nat_rel t ti) (?c::?'c Heap) ?Γ' ?R (dfs E s t)"
  unfolding dfs_def — Unfold definition of DFS
  apply sepref — Invoke sepref-tool
  done

concrete_definition dfs_impl uses dfs_impl
  — Extract generated implementation into constant
prepare_code_thms dfs_impl_def
  — Set up code equations for recursion combinators
export_code dfs_impl checking SML_imp
  — Generate SML code with Imperative/HOL
```

Finally, correctness is shown by combining the generated refinement theorem with the abstract correctness theorem.

```
corollary dfs_impl_correct:
  "finite (reachable E s) ⟹
  <is_graph nat_rel E Ei>
    dfs_impl Ei s t
  < λr. is_graph nat_rel E Ei * ↑(r ⟷ (s,t)∈E* ) >t"
proof -
  assume FIN: "finite (reachable E s)"
  note aux = hn_refine_ref[OF dfs_correct dfs_impl.refine,
    unfolded hn_refine_def, simplified, OF FIN]
  show ?thesis
    apply (rule cons_rule[OF _ _ aux])
    apply (sep_auto simp: hn_ctxt_def pure_def)+
    done
qed

end
```

15 Performance Test

```
theory Test
  imports Dijkstra_Impl_Adet
begin
```

In this theory, we test our implementation of Dijkstra's algorithm for larger, randomly generated graphs.

Simple linear congruence generator for (low-quality) random numbers:

```
definition "lcg_next s = ((81::nat)*s + 173) mod 268435456"
```

Generate a complete graph over the given number of vertices, with random weights:

```
definition ran_graph :: "nat ⇒ nat ⇒ (nat list×(nat × nat × nat) list)"
where
  "ran_graph vertices seed ==
    ([0::nat..<vertices],fst
     (while (λ (g,v,s). v < vertices)
     (λ (g,v,s).
     let (g'',v'',s'') = (while (λ (g',v',s'). v' < vertices)
      (λ (g',v',s'). ((v,s',v')#g',v'+1,lcg_next s'))
      (g,0,s))
     in (g'',v+1,s''))
     ([],0,lcg_next seed)))"
```

To experiment with the exported code, we fix the node type to natural numbers, and add a from-list conversion:

```
type_synonym nat_res = "(nat,((nat,nat) path × nat)) rm"
type_synonym nat_list_res = "(nat × (nat,nat) path × nat) list"

definition nat_dijkstra :: "(nat,nat) hlg ⇒ nat ⇒ nat_res" where
  "nat_dijkstra ≡ hrfn_dijkstra"

definition hlg_from_list_nat :: "(nat,nat) adj_list ⇒(nat,nat) hlg" where

  "hlg_from_list_nat ≡ hlg.from_list"

definition
  nat_res_to_list :: "nat_res ⇒ nat_list_res"
  where "nat_res_to_list ≡ rm.to_list"

value "nat_res_to_list (nat_dijkstra (hlg_from_list_nat (ran_graph 4 8912))
0)"

ML_val {*
let
  (* Configuration of test: *)
  val vertices = @{code nat_of_integer} 1000; (* Number of vertices *)
  val seed = @{code nat_of_integer} 123454; (* Seed for random number
generator *)
  val cfg_print_paths = true; (* Whether to output complete paths *)
  val cfg_print_res = true; (* Whether to output result at all *)

  (* Internals *)
  fun string_of_edge (u,(w,v)) = let
    val u = @{code integer_of_nat} u;
    val w = @{code integer_of_nat} w;
    val v = @{code integer_of_nat} v;
  in
```

```
    "(" ^ string_of_int u ^ "," ^ string_of_int w ^ "," ^ string_of_int
v ^ ")"
  end

  fun print_entry (dest,(path,weight)) = let
    val dest = @{code integer_of_nat} dest;
    val weight = @{code integer_of_nat} weight;
  in
    writeln (string_of_int dest ^ ": " ^ string_of_int weight ^
      ( if cfg_print_paths then
          " via [" ^ commas (map string_of_edge (rev path)) ^ "]"
        else ""
      )
    )
  end

  fun print_res [] = ()
    | print_res (a::l) = let val _ = print_entry a in print_res l end;

  val start = Time.now();
  val graph = @{code hlg_from_list_nat} (@{code ran_graph} vertices seed);
  val rt1 = Time.toMilliseconds (Time.now() - start);

  val start = Time.now();
  val res = @{code nat_dijkstra} graph (@{code nat_of_integer} 0);
  val rt2 = Time.toMilliseconds (Time.now() - start);
in
  writeln (string_of_int (@{code integer_of_nat} vertices) ^ " vertices:
"
  ^ string_of_int rt2 ^ " ms + "
  ^ string_of_int rt1 ^ " ms to create graph = "
  ^ string_of_int (rt1+rt2) ^ " ms");

  if cfg_print_res then
    print_res (@{code nat_res_to_list} res)
  else ()
end;
*}
```

end

16 Imperative Weighted Graphs

theory *Sepref_WGraph*
imports
  "../Sepref"
  "../../Dijkstra_Shortest_Path/Graph"
begin

```
  type_synonym 'w graph_impl = "(('w×nat) list) Heap.array"

  definition "is_graph R G Gi ≡
      ∃_A l. Gi ↦_a l * ↑(
        valid_graph G ∧
        nodes G = {0..<length l} ∧
        (∀ v∈nodes G. (l!v, succ G v) ∈ ⟨R ×_r nat_rel⟩list_set_rel)
      )"

  definition succi :: "'w::heap graph_impl ⇒ nat ⇒ ('w×nat) list Heap"
  where "succi G v = do {
    l ← Array.len G;
    if v<l then do { (* TODO: Alternatively, require v to be a valid node
as precondition! *)
      r ← Array.nth G v;
      return r
    } else return []
  }"

  lemma "
    < is_graph R G Gi * ↑(v∈nodes G) >
      succi Gi v
    < λr. is_graph R G Gi * ↑((r,succ G v)∈⟨R ×_r nat_rel⟩list_set_rel)
>"
    unfolding is_graph_def succi_def
    by sep_auto

  lemma (in valid_graph) succ_no_node_empty: "v∉V ⟹ succ G v = {}"
    unfolding succ_def using E_valid
    by auto

  lemma [sepref_fr_rules]: "
    hn_refine
      (hn_ctxt (is_graph R) G Gi * hn_val nat_rel v vi)
      (succi Gi vi)
      (hn_ctxt (is_graph R) G Gi * hn_val nat_rel v vi)
      (pure (⟨R ×_r nat_rel⟩list_set_rel))
      (RETURN$(succ$G$v))"
    apply rule
    unfolding hn_ctxt_def pure_def is_graph_def succi_def
    by (sep_auto simp: valid_graph.succ_no_node_empty list_set_autoref_empty)

  definition nodes_impl :: "'w::heap graph_impl ⇒ nat list Heap"
    where "nodes_impl Gi ≡ do {
    l ← Array.len Gi;
    return [0..<l]
  }"

  lemma [sepref_fr_rules]: "hn_refine
```

```
  (hn_ctxt (is_graph R) G Gi)
  (nodes_impl Gi)
  (hn_ctxt (is_graph R) G Gi)
  (pure (⟨nat_rel⟩list_set_rel))
  (RETURN$(nodes$G))"
  apply rule
  unfolding hn_ctxt_def pure_def is_graph_def nodes_impl_def
  by (sep_auto simp: list_set_rel_def br_def)

typedecl ('w)i_wgraph
typedecl i_weight


lemmas [id_rules] =
  itypeI[of succ "TYPE('w i_wgraph ⇒ nat ⇒ ('w×nat)set)"]
  itypeI[of nodes "TYPE('w i_wgraph ⇒ (nat)set)"]


definition cr_graph
  :: "nat ⇒ (nat × nat × 'w) list ⇒ 'w::heap graph_impl Heap"
where
  "cr_graph numV Es ≡ do {
    a ← Array.new numV [];
    a ← imp_nfoldli Es (λ_. return True) (λ(u,v,w) a. do {
      l ← Array.nth a u;
      let l = (w,v)#l;
      a ← Array.upd u l a;
      return a
    }) a;
    return a
  }"

export_code cr_graph checking SML_imp


end
```

17 Imperative Implementation of Dijkstra's Shortest Paths Algorithm

```
theory Sepref_Dijkstra
imports
  "../Sepref"
  "../../Dijkstra_Shortest_Path/Dijkstra"
  "../../Dijkstra_Shortest_Path/Test"
  "~~/src/HOL/Library/Code_Target_Numeral"
  "../../../DFS_Framework/Misc/DFS_Framework_Refine_Aux"
  "Sepref_WGraph"
```

```
begin



instantiation infty :: (heap) heap
begin
  instance
    apply default
    apply (rule_tac x="λInfty ⇒ 0 | Num a ⇒ to_nat a + 1" in exI)
    apply (rule injI)
    apply (auto split: infty.splits)
    done
end

lemma [sepref_import_param]:
  "(Infty,Infty)∈⟨R⟩infty_rel"
  "(Num,Num) ∈ R → ⟨R⟩infty_rel"
  "(Weight.val,Weight.val) ∈ ⟨Id⟩infty_rel → Id"
  by (auto simp: infty_rel_def)

lemma [constraint_rules]: "REL_IS_ID R ⟹ REL_IS_ID (⟨R⟩infty_rel)"
  apply (auto simp: infty_rel_def)
  apply (rename_tac x, case_tac x, auto)+
  done

definition "infty_plusi plusi a b ≡
  case (a,b) of
    (Infty, _) ⇒ Infty
  | (_, Infty) ⇒ Infty
  | (Num a, Num b) ⇒ Num (plusi a b)"

lemma [sepref_import_param]:
  assumes "GEN_OP plusi (op +) (R → R → R)"
  shows "(infty_plusi plusi, op +) ∈ ⟨R⟩infty_rel → ⟨R⟩infty_rel →
⟨R⟩infty_rel"
  using assms
  by (fastforce simp: infty_plusi_def infty_rel_def split: infty.splits
dest: fun_relD)

definition "infty_lessi lessi a b ≡
  case (a,b) of
    (Infty, _) ⇒ False
  | (Num _, Infty) ⇒ True
  | (Num a, Num b) ⇒ lessi a b"

lemma [sepref_import_param]:
  assumes "GEN_OP lessi (op <) (R → R → bool_rel)"
  shows "(infty_lessi lessi, op <) ∈ ⟨R⟩infty_rel → ⟨R⟩infty_rel →
bool_rel"
```

```
  using assms
  by (fastforce simp: infty_lessi_def infty_rel_def split: infty.splits
dest: fun_relD)



lemma aux_prod: "(λ(a,b) (c,d). f a b c d)
  = (λp1 p2. let (a,b) = p1; (c,d) = p2 in f a b c d)"
  by auto


lemma param_mpath': "(mpath',mpath')
  ∈ ⟨⟨A ×_r B ×_r A⟩list_rel ×_r B⟩option_rel → ⟨⟨A ×_r B ×_r A⟩list_rel⟩option_rel"
proof -
  have 1: "mpath' = map_option fst"
    apply (intro ext, rename_tac x)
    apply (case_tac x)
    apply simp
    apply (rename_tac a)
    apply (case_tac a)
    apply simp
    done
  show ?thesis
    unfolding 1
    by parametricity
qed
lemmas (in -) [sepref_import_param] = param_mpath'

lemma [sepref_import_param]:
  "(mpath_weight', mpath_weight') ∈ ⟨⟨A ×_r B ×_r A⟩list_rel ×_r B⟩option_rel
→ ⟨B⟩infty_rel"
  by (auto elim!: option_relE simp: infty_rel_def top_infty_def)


locale Dijkstra_Impl = Dijkstra G
  for G :: "(nat,'W::{weight,heap}) graph"
begin

  lemmas [id_rules] =
    itypeI[of G "TYPE('W i_wgraph)"]
    itypeI[of v0 "TYPE(nat)"]

  lemmas [sepref_import_param] =
    IdI[of v0]
    IdI[of "0::'W"]

  lemma w_plus_param[autoref_rules]: "(op +, op +::'W⇒_) ∈ Id → Id
→ Id" by simp
  lemma w_less_param[autoref_rules]: "(op <, op <::'W⇒_) ∈ Id → Id
```

→ *Id"* **by** *simp*
  lemmas *[sepref_import_param] = w_plus_param w_less_param*

  schematic_lemma *dijkstra_imp:*
    shows *"hn_refine (hn_ctxt (is_graph Id) G Gi) (?c1::?'c1 Heap) ?Γ1' ?R1 mdijkstra"*
    unfolding *mdijkstra_def mdinit_def mpop_min_def mupdate_def*
    unfolding *aux_prod*
    by *sepref*

end

concrete_definition *dijkstra_imp* **uses** *Dijkstra_Impl.dijkstra_imp*
prepare_code_thms *dijkstra_imp_def*
export_code *dijkstra_imp* **checking** *SML_imp*

end
theory *Buchi_Graph_Basic*
imports *Main "../../../Automatic_Refinement/Lib/Misc"*
begin

Specification of a reachable accepting cycle:

definition *"has_acc_cycle E A v0 ≡ ∃v∈A. (v0,v)∈E* ∧ (v,v)∈E⁺"*

17.0.1 Paths

inductive *path :: "('v×'v) set ⇒ 'v ⇒ 'v list ⇒ 'v ⇒ bool"* **for** *E*
where
  path0: "path E u [] u"
| *path_prepend: "⟦ (u,v)∈E; path E v l w ⟧ ⟹ path E u (u#l) w"*

lemma *path1: "(u,v)∈E ⟹ path E u [u] v"*
  by *(auto intro: path.intros)*

lemma *path_simps[simp]:*
  "path E u [] v ⟷ u=v"
  by *(auto intro: path0 elim: path.cases)*

lemma *path_conc:*
  assumes *P1: "path E u la v"*
  assumes *P2: "path E v lb w"*
  shows *"path E u (la@lb) w"*
  using *P1 P2* **apply** *induct*
  by *(auto intro: path.intros)*

lemma *path_append:*
  "⟦ path E u l v; (v,w)∈E ⟧ ⟹ path E u (l@[v]) w"
  using *path_conc[OF _ path1]* **.**

lemma *path_unconc:*
assumes *"path E u (la@lb) w"*
obtains *v* **where** *"path E u la v"* **and** *"path E v lb w"*
using *assms*
thm *path.induct*
apply *(induct u "la@lb" w arbitrary: la lb rule: path.induct)*
apply *(auto intro: path.intros elim!: list_Cons_eq_append_cases)*
done

lemma *path_uncons:*
assumes *"path E u (u'#l) w"*
obtains *v* **where** *"u'=u"* **and** *"(u,v)∈E"* **and** *"path E v l w"*
apply *(rule path_unconc[of E u "[u']" l w, simplified, OF assms])*
apply *(auto elim: path.cases)*
done

lemma *path_is_rtrancl:*
assumes *"path E u l v"*
shows *"(u,v)∈E***"*
using *assms*
by *induct auto*

lemma *rtrancl_is_path:*
assumes *"(u,v)∈E***"*
obtains *l* **where** *"path E u l v"*
using *assms*
by *induct (auto intro: path0 path_append)*

lemma *path_is_trancl:*
assumes *"path E u l v"*
and *"l≠[]"*
shows *"(u,v)∈E*$^+$*"*
using *assms*
apply *induct*
apply *auto []*
apply *(case_tac l)*
apply *auto*
done

lemma *trancl_is_path:*
assumes *"(u,v)∈E*$^+$*"*
obtains *l* **where** *"l≠[]"* **and** *"path E u l v"*
using *assms*
by *induct (auto intro: path0 path_append)*

Specification of witness for accepting cycle

definition *"is_acc_cycle E A v0 v r c*
≡ v∈A ∧ path E v0 r v ∧ path E v c v ∧ c≠[]"

Specification is compatible with existence of accepting cycle

lemma *is_acc_cycle_eq:*
  "has_acc_cycle E A v0 ⟷ (∃ v r c. is_acc_cycle E A v0 v r c)"
  unfolding *has_acc_cycle_def is_acc_cycle_def*
  by *(auto elim!: rtrancl_is_path trancl_is_path*
    intro: path_is_rtrancl path_is_trancl)

end

18 Nested DFS (HPY improvement)

theory *Nested_DFS*
imports
  "../../Refine_Dflt"
  Buchi_Graph_Basic
  Succ_Graph
  "../../Lib/Code_Target_ICF"
begin

Implementation of a nested DFS algorithm for accepting cycle detection using the refinement framework. The algorithm uses the improvement of [HPY96], i.e., it reports a cycle if the inner DFS finds a path back to the stack of the outer DFS.

The algorithm returns a witness in case that an accepting cycle is detected.

18.1 Tools for DFS Algorithms

18.1.1 Invariants

definition *"gen_dfs_pre E U S V u0 ≡*
  E``U ⊆ U   (Upper bound is closed under transitions *)*
  ∧ finite U (Upper bound is finite *)*
  ∧ V ⊆ U   (Visited set below upper bound *)*
  ∧ u0 ∈ U   (Start node in upper bound *)*
  ∧ E``(V-S) ⊆ V (Visited nodes closed under reachability, or on stack *)*
  ∧ u0∉V   (Start node not yet visited *)*
  ∧ S ⊆ V   (Stack is visited *)*
  ∧ (∀ v∈S. (v,u0)∈(E∩S×UNIV)⁺) (u0 reachable from stack, only over stack *)*
  "

definition *"gen_dfs_var U ≡ finite_psupset U"*

definition *"gen_dfs_fe_inv E U S V0 u0 it V brk ≡*
  (¬brk ⟶ E``(V-S) ⊆ V) (Visited set closed under reachability *)*
  ∧ E``{u0} - it ⊆ V   (Successors of u0 visited *)*
  ∧ V0 ⊆ V   (Visited set increasing *)*
  ∧ V ⊆ V0 ∪ (E - UNIV×S)⁺ `` (E``{u0} - it - S) (All visited*

```
                                                    nodes reachable *)
"

definition "gen_dfs_post E U S V0 u0 V brk ≡
  (¬brk ⟶ E‘‘(V-S) ⊆ V) (* Visited set closed under reachability *)
  ∧ u0 ∈ V               (* u0 visited *)
  ∧ V0 ⊆ V               (* Visited set increasing *)
  ∧ V ⊆ V0 ∪ (E - UNIV×S)* ‘‘ {u0} (* All visited nodes reachable *)
"
```

18.1.2 Invariant Preservation

```
lemma gen_dfs_pre_initial:
  assumes "finite (E*‘‘{v0})"
  assumes "v0∈U"
  shows "gen_dfs_pre E (E*‘‘{v0}) {} {} v0"
  using assms unfolding gen_dfs_pre_def
  apply auto
  done

lemma gen_dfs_pre_imp_wf:
  assumes "gen_dfs_pre E U S V u0"
  shows "wf (gen_dfs_var U)"
  using assms unfolding gen_dfs_pre_def gen_dfs_var_def by auto

lemma gen_dfs_pre_imp_fin:
  assumes "gen_dfs_pre E U S V u0"
  shows "finite (E ‘‘ {u0})"
  apply (rule finite_subset[where B="U"])
  using assms unfolding gen_dfs_pre_def
  by auto
```

Inserted *u0* on stack and to visited set

```
lemma gen_dfs_pre_imp_fe:
  assumes "gen_dfs_pre E U S V u0"
  shows "gen_dfs_fe_inv E U (insert u0 S) (insert u0 V) u0
    (E‘‘{u0}) (insert u0 V) False"
  using assms unfolding gen_dfs_pre_def gen_dfs_fe_inv_def
  apply auto
  done

lemma gen_dfs_fe_inv_pres_visited:
  assumes "gen_dfs_pre E U S V u0"
  assumes "gen_dfs_fe_inv E U (insert u0 S) (insert u0 V) u0 it V' False"
  assumes "t∈it" "it⊆E‘‘{u0}" "t∈V'"
  shows "gen_dfs_fe_inv E U (insert u0 S) (insert u0 V) u0 (it-{t}) V'
False"
  using assms unfolding gen_dfs_fe_inv_def
  apply auto
```

```
  done

lemma gen_dfs_upper_aux:
  assumes "(x,y)∈E'*"
  assumes "(u0,x)∈E"
  assumes "u0∈U"
  assumes "E'⊆E"
  assumes "E``U ⊆ U"
  shows "y∈U"
  using assms
  by induct auto


lemma gen_dfs_fe_inv_imp_var:
  assumes "gen_dfs_pre E U S V u0"
  assumes "gen_dfs_fe_inv E U (insert u0 S) (insert u0 V) u0 it V' False"
  assumes "t∈it" "it⊆E``{u0}" "t∉V'"
  shows "(V',V) ∈ gen_dfs_var U"
  using assms unfolding gen_dfs_fe_inv_def gen_dfs_pre_def gen_dfs_var_def
  apply (clarsimp simp add: finite_psupset_def)
  apply (blast dest: gen_dfs_upper_aux)
  done

lemma gen_dfs_fe_inv_imp_pre:
  assumes "gen_dfs_pre E U S V u0"
  assumes "gen_dfs_fe_inv E U (insert u0 S) (insert u0 V) u0 it V' False"
  assumes "t∈it" "it⊆E``{u0}" "t∉V'"
  shows "gen_dfs_pre E U (insert u0 S) V' t"
  using assms unfolding gen_dfs_fe_inv_def gen_dfs_pre_def
  apply clarsimp
  apply (intro conjI)
  apply (blast dest: gen_dfs_upper_aux)
  apply blast
  apply blast
  apply blast
  apply clarsimp
  apply (rule rtrancl_into_rtrancl[where b=u0])
  apply (auto intro: set_rev_mp[OF _ rtrancl_mono[where r="E ∩ S × UNIV"]])
[]
  apply blast
  done

lemma gen_dfs_post_imp_fe_inv:
  assumes "gen_dfs_pre E U S V u0"
  assumes "gen_dfs_fe_inv E U (insert u0 S) (insert u0 V) u0 it V' False"
  assumes "t∈it" "it⊆E``{u0}" "t∉V'"
  assumes "gen_dfs_post E U (insert u0 S) V' t V'' cyc"
  shows "gen_dfs_fe_inv E U (insert u0 S) (insert u0 V) u0 (it-{t}) V''
cyc"
```

```
  using assms unfolding gen_dfs_fe_inv_def gen_dfs_post_def gen_dfs_pre_def
  apply clarsimp
  apply (intro conjI)
  apply blast
  apply blast
  apply blast
  apply (erule order_trans)
  apply simp
  apply (rule conjI)
    apply (erule order_trans[
      where y="insert u0 (V ∪ (E - UNIV × insert u0 S)*
        `` (E `` {u0} - it - insert u0 S))"])
    apply blast

    apply (cases cyc)
      apply simp
      apply blast

      apply simp
      apply blast
  done

lemma gen_dfs_post_aux:
  assumes 1: "(u0,x)∈E"
  assumes 2: "(x,y)∈(E - UNIV × insert u0 S)*"
  assumes 3: "S⊆V" "x∉V"
  shows "(u0, y) ∈ (E - UNIV × S)*"
proof -
  from 1 3 have "(u0,x)∈(E - UNIV × S)" by blast
  also have "(x,y)∈(E - UNIV × S)*"
    apply (rule_tac set_rev_mp[OF 2 rtrancl_mono])
    by auto
  finally show ?thesis .
qed

lemma gen_dfs_fe_imp_post_brk:
  assumes "gen_dfs_pre E U S V u0"
  assumes "gen_dfs_fe_inv E U (insert u0 S) (insert u0 V) u0 it V' True"
  assumes "it⊆E``{u0}"
  shows "gen_dfs_post E U S V u0 V' True"
  using assms unfolding gen_dfs_pre_def gen_dfs_fe_inv_def gen_dfs_post_def
  apply clarify
  apply (intro conjI)
  apply simp
  apply simp
  apply simp
  apply clarsimp
  apply (blast intro: gen_dfs_post_aux)
  done
```

lemma *gen_dfs_fe_inv_imp_post:*
assumes *"gen_dfs_pre E U S V u0"*
assumes *"gen_dfs_fe_inv E U (insert u0 S) (insert u0 V) u0 {} V' cyc"*
assumes *"cyc⟶cyc'"*
shows *"gen_dfs_post E U S V u0 V' cyc'"*
using *assms* **unfolding** *gen_dfs_pre_def gen_dfs_fe_inv_def gen_dfs_post_def*
apply *clarsimp*
apply *(intro conjI)*
apply *blast*
apply *(auto intro: gen_dfs_post_aux) []*
done

lemma *gen_dfs_pre_imp_post_brk:*
assumes *"gen_dfs_pre E U S V u0"*
shows *"gen_dfs_post E U S V u0 (insert u0 V) True"*
using *assms* **unfolding** *gen_dfs_pre_def gen_dfs_post_def*
apply *auto*
done

18.1.3 Consequences of Postcondition

lemma *gen_dfs_post_imp_reachable:*
assumes *"gen_dfs_pre E U S V0 u0"*
assumes *"gen_dfs_post E U S V0 u0 V brk"*
shows *"V ⊆ V0 ∪ E*``{u0}"*
using *assms* **unfolding** *gen_dfs_post_def gen_dfs_pre_def*
apply *clarsimp*
apply *(blast intro: set_rev_mp[OF _ rtrancl_mono])*
done

lemma *gen_dfs_post_imp_complete:*
assumes *"gen_dfs_pre E U {} V0 u0"*
assumes *"gen_dfs_post E U {} V0 u0 V False"*
shows *"V0 ∪ E*``{u0} ⊆ V"*
using *assms* **unfolding** *gen_dfs_post_def gen_dfs_pre_def*
apply *clarsimp*
apply *(blast dest: Image_closed_trancl)*
done

lemma *gen_dfs_post_imp_eq:*
assumes *"gen_dfs_pre E U {} V0 u0"*
assumes *"gen_dfs_post E U {} V0 u0 V False"*
shows *"V = V0 ∪ E*``{u0}"*
using *gen_dfs_post_imp_reachable[OF assms] gen_dfs_post_imp_complete[OF assms]*
by *blast*

```
lemma gen_dfs_post_imp_below_U:
  assumes "gen_dfs_pre E U S V0 u0"
  assumes "gen_dfs_post E U S V0 u0 V False"
  shows "V ⊆ U"
  using assms unfolding gen_dfs_pre_def gen_dfs_post_def
  apply clarsimp
  apply (blast intro: set_rev_mp[OF _ rtrancl_mono] dest: Image_closed_trancl)
  done
```

18.2 Abstract Algorithm

18.2.1 Inner (red) DFS

A witness of the red algorithm is a node on the stack and a path to this node

```
type_synonym 'v red_witness = "('v list × 'v) option"
```

Prepend node to red witness

```
fun prep_wit_red :: "'v ⇒ 'v red_witness ⇒ 'v red_witness" where
  "prep_wit_red v None = None"
| "prep_wit_red v (Some (p,u)) = Some (v#p,u)"
```

Initial witness for node `u` with onstack successor `v`

```
definition red_init_witness :: "'v ⇒ 'v ⇒ 'v red_witness" where
  "red_init_witness u v = Some ([u],v)"

definition red_dfs where
  "red_dfs E onstack V u ≡
    REC_T (λD (V,u). do {
      let V=insert u V;

      (* Check whether we have a successor on stack *)
      brk ← FOREACH_C (E``{u}) (λbrk. brk=None)
        (λt _. if t∈onstack then RETURN (red_init_witness u t) else RETURN
None)
        None;

      (* Recurse for successors *)
      case brk of
        None ⇒
          FOREACH_C (E``{u}) (λ(V,brk). brk=None)
            (λt (V,_).
              if t∉V then do {
                (V,brk) ← D (V,t);
                RETURN (V,prep_wit_red u brk)
              } else RETURN (V,None))
            (V,None)
      | _ ⇒ RETURN (V,brk)
```

```
    }) (V,u)
  "
```

A witness of the blue DFS may be in two different phases, the *REACH* phase is before the node on the stack has actually been popped, and the *CIRC* phase is after the node on the stack has been popped.

REACH v p u p':

v accepting node

p path from *v* to *u*

u node on stack

p' path from current node to *v*

CIRC v pc pr:

v accepting node

pc path from *v* to *v*

pr path from current node to *v*

```
datatype 'v blue_witness =
  NO_CYC
| REACH "'v" "'v list" "'v" "'v list"
| CIRC "'v" "'v list" "'v list"
```

Prepend node to witness

```
primrec prep_wit_blue :: "'v ⇒ 'v blue_witness ⇒ 'v blue_witness" where
  "prep_wit_blue u0 NO_CYC = NO_CYC"
| "prep_wit_blue u0 (REACH v p u p') = (
  if u0=u then
    CIRC v (p@u#p') (u0#p')
  else
    REACH v p u (u0#p')
  )"
| "prep_wit_blue u0 (CIRC v pc pr) = CIRC v pc (u0#pr)"
```

Initialize blue witness

```
fun init_wit_blue :: "'v ⇒ 'v red_witness ⇒ 'v blue_witness" where
  "init_wit_blue u0 None = NO_CYC"
| "init_wit_blue u0 (Some (p,u)) = (
  if u=u0 then
    CIRC u0 p []
  else REACH u0 p u [])"
```

Extract result from witness

```
definition "extract_res cyc
  ≡ (case cyc of CIRC v pc pr ⇒ Some (v,pc,pr) | _ ⇒ None)"
```

18.2.2 Outer (Blue) DFS

definition *blue_dfs*
```
  :: "('a×'a) set ⇒ 'a set ⇒ 'a ⇒ ('a×'a list×'a list) option nres"

  where
  "blue_dfs E A s ≡ do {
    (_,_,_,cyc) ← REC_T (λD (blues,reds,onstack,s). do {
      let blues=insert s blues;
      let onstack=insert s onstack;
      (blues,reds,onstack,cyc) ←
      FOREACH_C (E``{s}) (λ(_,_,_,cyc). cyc=NO_CYC)
        (λt (blues,reds,onstack,cyc).
          if t∉blues then do {
            (blues,reds,onstack,cyc) ← D (blues,reds,onstack,t);
            RETURN (blues,reds,onstack,(prep_wit_blue s cyc))
          } else RETURN (blues,reds,onstack,cyc))
        (blues,reds,onstack,NO_CYC);

      (reds,cyc) ←
      if cyc=NO_CYC ∧ s∈A then do {
        (reds,rcyc) ← red_dfs E onstack reds s;
        RETURN (reds, init_wit_blue s rcyc)
      } else RETURN (reds,cyc);

      let onstack=onstack - {s};
      RETURN (blues,reds,onstack,cyc)
    }) ({},{},{},s);
    RETURN (extract_res cyc)
  }
  "
```

18.3 Correctness

Additional invariant to be maintained between calls of red dfs

definition *"red_dfs_inv E U reds onstack ≡*
```
  E``U ⊆ U           (* Upper bound is closed under transitions *)
  ∧ finite U         (* Upper bound is finite *)
  ∧ reds ⊆ U         (* Red set below upper bound *)
  ∧ E``reds ⊆ reds   (* Red nodes closed under reachability *)
  ∧ E``reds ∩ onstack = {} (* No red node with edge to stack *)
"
```

lemma *red_dfs_inv_initial:*
assumes *"finite (E*``{v0})"*
shows *"red_dfs_inv E (E*``{v0}) {} {}"*
using *assms* **unfolding** *red_dfs_inv_def*
apply *auto*
done

Correctness of the red DFS.

```
theorem red_dfs_correct:
  fixes v0 u0 :: 'v
  assumes PRE:
    "red_dfs_inv E U reds onstack"
    "u0∈U"
    "u0∉reds"
  shows "red_dfs E onstack reds u0
    ≤ SPEC (λ(reds',cyc). case cyc of
      Some (p,v) ⇒ v∈onstack ∧ p≠[] ∧ path E u0 p v
    | None ⇒
        red_dfs_inv E U reds' onstack
        ∧ u0∈reds'
        ∧ reds' ⊆ reds ∪ E* `` {u0}
  )"
proof -
  let ?dfs_red = "
    REC_T (λD (V,u). do {
      let V=insert u V;

      (* Check whether we have a successor on stack *)
      brk ← FOREACH_C (E``{u}) (λbrk. brk=None)
        (λt _. if t∈onstack then
            RETURN (red_init_witness u t)
          else RETURN None)
        None;

      (* Recurse for successors *)
      case brk of
        None ⇒
          FOREACH_C (E``{u}) (λ(V,brk). brk=None)
            (λt (V,_).
              if t∉V then do {
                (V,brk) ← D (V,t);
                RETURN (V,prep_wit_red u brk)
              } else RETURN (V,None))
            (V,None)
      | _ ⇒ RETURN (V,brk)
    }) (V,u)
"
  let "REC_T ?body ?init" = "?dfs_red"

  def pre ≡ "λS (V,u0). gen_dfs_pre E U S V u0 ∧ E``V ∩ onstack = {}"
  def post ≡ "λS (V0,u0) (V,cyc). gen_dfs_post E U S V0 u0 V (cyc≠None)
    ∧ (case cyc of None ⇒ E``V ∩ onstack = {}
      | Some (p,v) ⇒ v∈onstack ∧ p≠[] ∧ path E u0 p v)
    "

  def fe_inv ≡ "λS V0 u0 it (V,cyc).
```

```
  gen_dfs_fe_inv E U S V0 u0 it V (cyc≠None)
  ∧ (case cyc of None ⇒ E‘‘V ∩ onstack = {}
    | Some (p,v) ⇒ v∈onstack ∧ p≠[] ∧ path E u0 p v)
  "


from PRE have GENPRE: "gen_dfs_pre E U {} reds u0"
  unfolding red_dfs_inv_def gen_dfs_pre_def
  by auto
with PRE have PRE': "pre {} (reds,u0)"
  unfolding pre_def red_dfs_inv_def
  by auto

have IMP_POST: "SPEC (post {} (reds,u0))
  ≤ SPEC (λ(reds',cyc). case cyc of
    Some (p,v) ⇒ v∈onstack ∧ p≠[] ∧ path E u0 p v
  | None ⇒
      red_dfs_inv E U reds' onstack
      ∧ u0∈reds'
      ∧ reds' ⊆ reds ∪ E* ‘‘ {u0})"
  apply (clarsimp split: option.split)
  apply (intro impI conjI allI)
  apply simp_all
proof -
  fix reds' p v
  assume "post {} (reds,u0) (reds',Some (p,v))"
  thus "v∈onstack" and "p≠[]" and "path E u0 p v"
    unfolding post_def by auto
next
  fix reds'
  assume "post {} (reds, u0) (reds', None)"
  hence GPOST: "gen_dfs_post E U {} reds u0 reds' False"
    and NS: "E‘‘reds' ∩ onstack = {}"
    unfolding post_def by auto

  from GPOST show "u0∈reds'" unfolding gen_dfs_post_def by auto

  show "red_dfs_inv E U reds' onstack"
    unfolding red_dfs_inv_def
    apply (intro conjI)
    using GENPRE[unfolded gen_dfs_pre_def]
    apply (simp_all) [2]
    apply (rule gen_dfs_post_imp_below_U[OF GENPRE GPOST])
    using GPOST[unfolded gen_dfs_post_def] apply simp
    apply fact
    done

  from GPOST show "reds' ⊆ reds ∪ E* ‘‘ {u0}"
    unfolding gen_dfs_post_def by auto
```

```
  qed

  {
    fix σ S
    assume INV0: "pre S σ"
    have "REC_T ?body σ
      ≤ SPEC (post S σ)"

    apply (rule RECT_rule_arb[where
      pre="pre" and
      V="gen_dfs_var U <*lex*> {}" and
      arb="S"
      ])

    apply refine_mono

    using INV0[unfolded pre_def] apply (auto intro: gen_dfs_pre_imp_wf)
[]

    apply fact

    apply (rename_tac D S u)
    apply (intro refine_vcg)


    apply (rule_tac I="λit cyc.
      (case cyc of None ⇒ (E‘‘{b} - it) ∩ onstack = {}
        | Some (p,v) ⇒ (v∈onstack ∧ p≠[] ∧ path E b p v))"
      in FOREACHc_rule)
    apply (auto simp add: pre_def gen_dfs_pre_imp_fin) []
    apply auto []
    apply (auto
      split: option.split
      simp: red_init_witness_def intro: path1) []

    apply (intro refine_vcg)


    apply (rule_tac I="fe_inv (insert b S) (insert b a) b" in
      FOREACHc_rule
    )
    apply (auto simp add: pre_def gen_dfs_pre_imp_fin) []

    apply (auto simp add: pre_def fe_inv_def gen_dfs_pre_imp_fe) []

    apply (intro refine_vcg)


    apply (rule order_trans)
```

```
      apply (rprems)
      apply (clarsimp simp add: pre_def fe_inv_def)
      apply (rule gen_dfs_fe_inv_imp_pre, assumption+) []
      apply (auto simp add: pre_def fe_inv_def intro: gen_dfs_fe_inv_imp_var)
[]

      apply (clarsimp simp add: pre_def post_def fe_inv_def
        split: option.split_asm prod.split_asm
        ) []
        apply (blast intro: gen_dfs_post_imp_fe_inv)
        apply (blast intro: gen_dfs_post_imp_fe_inv path_prepend)

      apply (auto simp add: pre_def post_def fe_inv_def
        intro: gen_dfs_fe_inv_pres_visited) []

      apply (auto simp add: pre_def post_def fe_inv_def
        intro: gen_dfs_fe_inv_imp_post) []

      apply (auto simp add: pre_def post_def fe_inv_def
        intro: gen_dfs_fe_imp_post_brk) []

      apply (auto simp add: pre_def post_def fe_inv_def
        intro: gen_dfs_pre_imp_post_brk) []

      apply (auto simp add: pre_def post_def fe_inv_def
        intro: gen_dfs_pre_imp_post_brk) []

      done
  } note GEN=this

  note GEN[OF PRE']
  also note IMP_POST
  finally show ?thesis
    unfolding red_dfs_def .
qed
```

Main theorem: Correctness of the blue DFS

```
theorem blue_dfs_correct:
  fixes v0 :: 'v
  assumes FIN[simp,intro!]: "finite (E*``{v0})"
  shows "blue_dfs E A v0 ≤ SPEC (λr.
    case r of None ⇒ ¬has_acc_cycle E A v0
  | Some (v,pc,pv) ⇒ is_acc_cycle E A v0 v pv pc)"
proof -
  let ?ndfs = "
  do {
    (_,_,_,cyc) ← REC_T (λD (blues,reds,onstack,s). do {
      let blues=insert s blues;
      let onstack=insert s onstack;
```

```
    (blues,reds,onstack,cyc) ←
    FOREACH_C (E``{s}) (λ(_,_,_,cyc). cyc=NO_CYC)
      (λt (blues,reds,onstack,cyc).
        if t∉blues then do {
          (blues,reds,onstack,cyc) ← D (blues,reds,onstack,t);
          RETURN (blues,reds,onstack,(prep_wit_blue s cyc))
        } else RETURN (blues,reds,onstack,cyc))
      (blues,reds,onstack,NO_CYC);

    (reds,cyc) ←
    if cyc=NO_CYC ∧ s∈A then do {
      (reds,rcyc) ← red_dfs E onstack reds s;
      RETURN (reds, init_wit_blue s rcyc)
    } else RETURN (reds,cyc);

    let onstack=onstack - {s};
    RETURN (blues,reds,onstack,cyc)
  }) ({},{},{},s);
  RETURN (case cyc of NO_CYC ⇒ None | CIRC v pc pr ⇒ Some (v,pc,pr))
}"
let "do {_ ← REC_T ?body ?init; _}" = "?ndfs"

let ?U = "E*``{v0}"

def add_inv ≡ "λblues reds onstack.
  ¬(∃v∈(blues-onstack)∩A. (v,v)∈E^+)  (* No cycles over finished,
                                         accepting states *)
  ∧ reds ⊆ blues                     (* Red nodes are also blue *)
  ∧ reds ∩ onstack = {}              (* No red nodes on stack *)
  ∧ red_dfs_inv E ?U reds onstack"

def cyc_post ≡ "λblues reds onstack u0 cyc. (case cyc of
    NO_CYC ⇒ add_inv blues reds onstack
  | REACH v p u p' ⇒ v∈A ∧ u∈onstack-{u0} ∧ p≠[]
    ∧ path E v p u ∧ path E u0 p' v
  | CIRC v pc pr ⇒ v∈A ∧ pc≠[] ∧ path E v pc v ∧ path E u0 pr v
  )"

def pre ≡ "λ(blues,reds,onstack,u).
  gen_dfs_pre E ?U onstack blues u ∧ add_inv blues reds onstack"

def post ≡ "λ(blues0,reds0::'v set,onstack0,u0) (blues,reds,onstack,cyc).

  onstack = onstack0
  ∧ gen_dfs_post E ?U onstack0 blues0 u0 blues (cyc≠NO_CYC)
  ∧ cyc_post blues reds onstack u0 cyc"

def fe_inv ≡ "λblues0 u0 onstack0 it (blues,reds,onstack,cyc).
  onstack=onstack0
```

```
    ∧ gen_dfs_fe_inv E ?U onstack0 blues0 u0 it blues (cyc≠NO_CYC)
    ∧ cyc_post blues reds onstack u0 cyc"

  have GENPRE: "gen_dfs_pre E ?U {} {} v0"
    apply (auto intro: gen_dfs_pre_initial)
    done
  hence PRE': "pre ({},{},{},v0)"
    unfolding pre_def add_inv_def
    apply (auto intro: red_dfs_inv_initial)
    done

  {
    fix blues reds onstack cyc
    assume "post ({},{},{},v0) (blues,reds,onstack,cyc)"
    hence "case cyc of NO_CYC ⇒ ¬has_acc_cycle E A v0
      | REACH _ _ _ _ ⇒ False
      | CIRC v pc pr ⇒ is_acc_cycle E A v0 v pr pc"
      unfolding post_def cyc_post_def
      apply (cases cyc)
      using gen_dfs_post_imp_eq[OF GENPRE, of blues]
      apply (auto simp: add_inv_def has_acc_cycle_def) []
      apply auto []
      apply (auto simp: is_acc_cycle_def) []
      done
  } note IMP_POST = this

  {
    fix onstack blues u0 reds
    assume "pre (blues,reds,onstack,u0)"
    hence "fe_inv (insert u0 blues) u0 (insert u0 onstack) (E``{u0})
      (insert u0 blues,reds,insert u0 onstack,NO_CYC)"
      unfolding fe_inv_def add_inv_def cyc_post_def
      apply clarsimp
      apply (intro conjI)
      apply (simp add: pre_def gen_dfs_pre_imp_fe)
      apply (auto simp: pre_def add_inv_def) []
      apply (auto simp: pre_def add_inv_def) []
      defer
      apply (auto simp: pre_def add_inv_def) []
      apply (unfold pre_def add_inv_def red_dfs_inv_def gen_dfs_pre_def)
[]
      apply clarsimp
      apply blast

      apply (auto simp: pre_def add_inv_def gen_dfs_pre_def) []
      done
  } note PRE_IMP_FE = this

  have [simp]: "⋀u cyc. prep_wit_blue u cyc = NO_CYC ⟷ cyc=NO_CYC"
```

```
  by (case_tac cyc) auto

{
  fix blues0 reds0 onstack0 u0
    blues reds onstack blues' reds' onstack'
    cyc it t
  assume PRE: "pre (blues0,reds0,onstack0,u0)"
  assume FEI: "fe_inv (insert u0 blues0) u0 (insert u0 onstack0)
      it (blues,reds,onstack,NO_CYC)"
  assume IT: "t∈it" "it⊆E``{u0}" "t∉blues"
  assume POST: "post (blues,reds,onstack, t) (blues',reds',onstack',cyc)"
  note [simp del] = path_simps
  have "fe_inv (insert u0 blues0) u0 (insert u0 onstack0) (it-{t})
    (blues',reds',onstack',prep_wit_blue u0 cyc)"
    unfolding fe_inv_def
    using PRE FEI IT POST
    unfolding fe_inv_def post_def pre_def
    apply (clarsimp)
    apply (intro allI impI conjI)
    apply (blast intro: gen_dfs_post_imp_fe_inv)
    unfolding cyc_post_def
    apply (auto split: blue_witness.split_asm intro: path_conc path_prepend)
    done
} note FE_INV_PRES=this

{
  fix blues reds onstack u0
  assume "pre (blues,reds,onstack,u0)"
  hence "(v0,u0)∈E*"
    unfolding pre_def gen_dfs_pre_def by auto
} note PRE_IMP_REACH = this

{
  fix blues0 reds0 onstack0 u0 blues reds onstack
  assume A: "pre (blues0,reds0,onstack0,u0)"
     "fe_inv (insert u0 blues0) u0 (insert u0 onstack0)
      {} (blues,reds,onstack,NO_CYC)"
     "u0∈A"
  have "u0∉reds" using A
    unfolding fe_inv_def add_inv_def pre_def cyc_post_def
    apply auto
    done
} note FE_IMP_RED_PRE = this

{
  fix blues0 reds0 onstack0 u0 blues reds onstack rcyc reds'
  assume PRE: "pre (blues0,reds0,onstack0,u0)"
  assume FEI: "fe_inv (insert u0 blues0) u0 (insert u0 onstack0)
      {} (blues,reds,onstack,NO_CYC)"
```

```
    assume ACC: "u0∈A"
    assume SPECR: "case rcyc of
      Some (p,v) ⇒ v∈onstack ∧ p≠[] ∧ path E u0 p v
    | None ⇒
        red_dfs_inv E ?U reds' onstack
        ∧ u0∈reds'
        ∧ reds' ⊆ reds ∪ E* `` {u0}"
    have "post (blues0,reds0,onstack0,u0)
      (blues,reds',onstack - {u0},init_wit_blue u0 rcyc)"
      unfolding post_def add_inv_def cyc_post_def
      apply (clarsimp)
      apply (intro conjI)
    proof -
      from PRE FEI show OS0[symmetric]: "onstack - {u0} = onstack0"
        by (auto simp: pre_def fe_inv_def add_inv_def gen_dfs_pre_def)
[]

      from PRE FEI have "u0∈onstack"
        unfolding pre_def gen_dfs_pre_def fe_inv_def gen_dfs_fe_inv_def
        by auto

      from PRE FEI
      show POST: "gen_dfs_post E (E* `` {v0}) onstack0 blues0 u0 blues

        (init_wit_blue u0 rcyc ≠ NO_CYC)"
        by (auto simp: pre_def fe_inv_def intro: gen_dfs_fe_inv_imp_post)

      from FEI have [simp]: "onstack=insert u0 onstack0"
        unfolding fe_inv_def by auto
      from FEI have "u0∈blues" unfolding fe_inv_def gen_dfs_fe_inv_def
by auto

      case goal3 show ?case
        apply (cases rcyc)
        apply (simp_all add: split_paired_all)
      proof -
        assume [simp]: "rcyc=None"
        show "(∀v∈(blues - (onstack0 - {u0})) ∩ A. (v, v) ∉ E+) ∧
          reds' ⊆ blues ∧
          reds' ∩ (onstack0 - {u0}) = {} ∧
          red_dfs_inv E (E* `` {v0}) reds' (onstack0 - {u0})"
        proof (intro conjI)
          from SPECR have RINV: "red_dfs_inv E ?U reds' onstack"
            and "u0∈reds'"
            and REDS'R: "reds' ⊆ reds ∪ E*``{u0}"
            by auto

          from RINV show
            RINV': "red_dfs_inv E (E* `` {v0}) reds' (onstack0 - {u0})"
```

```
    unfolding red_dfs_inv_def by auto

  from RINV'[unfolded red_dfs_inv_def] have
    REDS'CL: "E``reds' ⊆ reds'"
    and DJ': "E `` reds' ∩ (onstack0 - {u0}) = {}" by auto

  from RINV[unfolded red_dfs_inv_def] have
    DJ: "E `` reds' ∩ (onstack) = {}" by auto

  show "reds' ⊆ blues"
  proof
    fix v assume "v∈reds'"
    with REDS'R have "v∈reds ∨ (u0,v)∈E*" by blast
    thus "v∈blues" proof
      assume "v∈reds"
      moreover with FEI have "reds⊆blues"
        unfolding fe_inv_def add_inv_def cyc_post_def by auto
      ultimately show ?thesis ..
    next
      from POST[unfolded gen_dfs_post_def OS0] have
        CL: "E `` (blues - (onstack0 - {u0})) ⊆ blues" and "u0∈blues"

        by auto
      from PRE FEI have "onstack0 ⊆ blues"
        unfolding pre_def fe_inv_def gen_dfs_pre_def gen_dfs_fe_inv_def
        by auto

      assume "(u0,v)∈E*"
      thus "v∈blues"
      proof (cases rule: rtrancl_last_visit[where S="onstack
- {u0}"])
        case no_visit
        thus "v∈blues" using `u0∈blues` CL
          by induct (auto elim: rtranclE)
      next
        case (last_visit_point u)
        then obtain uh where "(u0,uh)∈E*" and "(uh,u)∈E"
          by (metis tranclD2)
        with REDS'CL DJ' `u0∈reds'` have "uh∈reds'"
          by (auto dest: Image_closed_trancl)
        with DJ' `(uh,u)∈E` `u ∈ onstack - {u0}` have False

          by simp blast
        thus ?thesis ..
      qed
    qed
  qed

  show "∀v∈(blues - (onstack0 - {u0})) ∩ A. (v, v) ∉ E+"
```

```
          proof
            fix v
            assume A: "v ∈ (blues - (onstack0 - {u0})) ∩ A"
            show "(v,v)∉E⁺" proof (cases "v=u0")
              assume "v≠u0"
              with A have "v∈(blues - (insert u0 onstack)) ∩ A" by auto
              with FEI show ?thesis
                unfolding fe_inv_def add_inv_def cyc_post_def by auto
            next
              assume [simp]: "v=u0"
              show ?thesis proof
                assume "(v,v)∈E⁺"
                then obtain uh where "(u0,uh)∈E*" and "(uh,u0)∈E"
                  by (auto dest: tranclD2)
                with REDS'CL DJ ‘u0∈reds'‘ have "uh∈reds'"
                  by (auto dest: Image_closed_trancl)
                with DJ ‘(uh,u0)∈E‘ ‘u0 ∈ onstack‘ show False by blast
              qed
            qed
          qed

          show "reds' ∩ (onstack0 - {u0}) = {}"
          proof (rule ccontr)
            assume "reds' ∩ (onstack0 - {u0}) ≠ {}"
            then obtain v where "v∈reds'" and "v∈onstack0" and "v≠u0"
by auto

            from ‘v∈reds'‘ REDS'R have "v∈reds ∨ (u0,v)∈E*"
              by auto
            thus False proof
              assume "v∈reds"
              with FEI[unfolded fe_inv_def add_inv_def cyc_post_def]
                ‘v∈onstack0‘
              show False by auto
            next
              assume "(u0,v)∈E*"
              with ‘v≠u0‘ obtain uh where "(u0,uh)∈E*" and "(uh,v)∈E"
                by (auto elim: rtranclE)
              with REDS'CL DJ ‘u0∈reds'‘ have "uh∈reds'"
                by (auto dest: Image_closed_trancl)
              with DJ ‘(uh,v)∈E‘ ‘v ∈ onstack0‘ show False by simp
blast
            qed
          qed
        qed
      next
        fix u p
        assume [simp]: "rcyc = Some (p,u)"
        show "(u = u0 ⟶ u0 ∈ A ∧ p ≠ [] ∧ path E u0 p u0) ∧
```

```
        (u ≠ u0 ⟶ u0 ∈ A ∧ u ∈ onstack0 ∧ p ≠ [] ∧ path E u0 p
u)"
      proof (intro conjI impI)
        show "u0∈A" by fact
        show "u0∈A" by fact
        from SPECR show
          "u≠u0 ⟹ u∈onstack0"
          "p≠[]"
          "p≠[]"
          "path E u0 p u"
          "u=u0 ⟹ path E u0 p u0"
          by auto
      qed
    qed
  qed
} note RED_IMP_POST = this

{
  fix blues0 reds0 onstack0 u0 blues reds onstack and cyc :: "'v blue_witness"
  assume PRE: "pre (blues0,reds0,onstack0,u0)"
  and FEI: "fe_inv (insert u0 blues0) u0 (insert u0 onstack0)
      {} (blues,reds,onstack,NO_CYC)"
  and FC[simp]: "cyc=NO_CYC"
  and NCOND: "u0∉A"

  from PRE FEI have OS0: "onstack0 = onstack - {u0}"
    by (auto simp: pre_def fe_inv_def add_inv_def gen_dfs_pre_def) []

  from PRE FEI have "u0∈onstack"
    unfolding pre_def gen_dfs_pre_def fe_inv_def gen_dfs_fe_inv_def
    by auto
  with OS0 have OS1: "onstack = insert u0 onstack0" by auto

  have "post (blues0,reds0,onstack0,u0) (blues,reds,onstack - {u0},NO_CYC)"
    apply (clarsimp simp: post_def cyc_post_def) []
    apply (intro conjI impI)
    apply (simp add: OS0)
    using PRE FEI apply (auto
      simp: pre_def fe_inv_def intro: gen_dfs_fe_inv_imp_post) []

    using FEI[unfolded fe_inv_def cyc_post_def] unfolding add_inv_def
    apply clarsimp
    apply (intro conjI)
    using NCOND apply auto []
    apply auto []
    apply (clarsimp simp: red_dfs_inv_def, blast) []
    done
} note NCOND_IMP_POST=this
```

```
{
  fix blues0 reds0 onstack0 u0 blues reds onstack it
    and cyc :: "'v blue_witness"
  assume PRE: "pre (blues0,reds0,onstack0,u0)"
  and FEI: "fe_inv (insert u0 blues0) u0 (insert u0 onstack0)
      it (blues,reds,onstack,cyc)"
  and NC: "cyc≠NO_CYC"
  and IT: "it⊆E``{u0}"
  from PRE FEI have OS0: "onstack0 = onstack - {u0}"
    by (auto simp: pre_def fe_inv_def add_inv_def gen_dfs_pre_def) []

  from PRE FEI have "u0∈onstack"
    unfolding pre_def gen_dfs_pre_def fe_inv_def gen_dfs_fe_inv_def
    by auto
  with OS0 have OS1: "onstack = insert u0 onstack0" by auto

  have "post (blues0,reds0,onstack0,u0) (blues,reds,onstack - {u0},cyc)"
    apply (clarsimp simp: post_def) []
    apply (intro conjI impI)
    apply (simp add: OS0)
    using PRE FEI IT NC apply (auto
      simp: pre_def fe_inv_def intro: gen_dfs_fe_imp_post_brk) []
    using FEI[unfolded fe_inv_def] NC
    unfolding cyc_post_def
    apply (auto split: blue_witness.split simp: OS1) []
    done
} note BREAK_IMP_POST = this

{
  fix σ
  assume INV0: "pre σ"
  have "REC_T ?body σ
    ≤ SPEC (post σ)"

    apply (intro refine_vcg
      RECT_rule[where pre="pre"
      and V="gen_dfs_var ?U <*lex*> {}"]
    )
    apply refine_mono
    apply (blast intro!: gen_dfs_pre_imp_wf[OF GENPRE])
    apply (rule INV0)


    apply (rule_tac
      I="fe_inv (insert bb a) bb (insert bb ab)"
      in FOREACHc_rule')

    apply (auto simp add: pre_def gen_dfs_pre_imp_fin) []
```

```
      apply (blast intro: PRE_IMP_FE)

      apply (intro refine_vcg)


      apply (rule order_trans)
      apply (rprems)
      apply (clarsimp simp add: pre_def fe_inv_def cyc_post_def)
      apply (rule gen_dfs_fe_inv_imp_pre, assumption+) []
      apply (auto simp add: pre_def fe_inv_def intro: gen_dfs_fe_inv_imp_var)
[]

      apply (auto intro: FE_INV_PRES) []

      apply (auto simp add: pre_def post_def fe_inv_def
        intro: gen_dfs_fe_inv_pres_visited) []

      apply (intro refine_vcg)


      apply (rule order_trans)
      apply (rule red_dfs_correct[where U="E* `` {v0}"])
      apply (auto simp add: fe_inv_def add_inv_def cyc_post_def) []
      apply (auto intro: PRE_IMP_REACH) []
      apply (auto dest: FE_IMP_RED_PRE) []

      apply (intro refine_vcg)
      apply clarsimp
      apply (rule RED_IMP_POST, assumption+) []

      apply (clarsimp, blast intro: NCOND_IMP_POST) []

      apply (intro refine_vcg)
      apply simp

      apply (clarsimp, blast intro: BREAK_IMP_POST) []
      done
  } note GEN=this

  show ?thesis
    unfolding blue_dfs_def extract_res_def
    apply (intro refine_vcg)
    apply (rule order_trans)
    apply (rule GEN)
    apply fact
    apply (intro refine_vcg)
    apply clarsimp
    apply (drule IMP_POST)
    apply (simp split: blue_witness.split_asm)
```

done
qed

18.4 Refinement

18.4.1 Setup for Custom Datatypes

This effort can be automated, but currently, such an automation is not yet implemented

abbreviation *"red_wit_rel ≡ ⟨⟨⟨nat_rel⟩list_rel,nat_rel⟩prod_rel⟩option_rel"*
abbreviation *"wit_res_rel ≡*
⟨⟨nat_rel,⟨⟨nat_rel⟩list_rel,⟨nat_rel⟩list_rel⟩prod_rel⟩prod_rel⟩option_rel"
abbreviation *"i_red_wit ≡ ⟨⟨⟨i_nat⟩$_i$i_list,i_nat⟩$_i$i_prod⟩$_i$i_option"*
abbreviation *"i_res ≡*
⟨⟨i_nat,⟨⟨i_nat⟩$_i$i_list,⟨i_nat⟩$_i$i_list⟩$_i$i_prod⟩$_i$i_prod⟩$_i$i_option"

abbreviation *"blue_wit_rel ≡ (Id::(nat blue_witness × _) set)"*
consts *i_blue_wit :: interface*

term *extract_res*

lemma *[autoref_itype]:*
"NO_CYC ::$_i$ i_blue_wit"
"op = ::$_i$ i_blue_wit →$_i$ i_blue_wit →$_i$ i_bool"
"init_wit_blue ::$_i$ i_nat →$_i$ i_red_wit →$_i$ i_blue_wit"
"prep_wit_blue ::$_i$ i_nat →$_i$ i_blue_wit →$_i$ i_blue_wit"
"red_init_witness ::$_i$ i_nat →$_i$ i_nat →$_i$ i_red_wit"
"prep_wit_red ::$_i$ i_nat →$_i$ i_red_wit →$_i$ i_red_wit"
"extract_res ::$_i$ i_blue_wit →$_i$ i_res"
by *auto*

context begin interpretation *autoref_syn .*
lemma *[autoref_op_pat]: "NO_CYC ≡ OP NO_CYC :::$_i$ i_blue_wit"* **by** *simp*
end

lemma *autoref_wit[autoref_rules_raw]:*
"(NO_CYC,NO_CYC)∈blue_wit_rel"
"(op =, op =) ∈ blue_wit_rel → blue_wit_rel → bool_rel"
"(init_wit_blue, init_wit_blue) ∈ nat_rel → red_wit_rel → blue_wit_rel"
"(prep_wit_blue,prep_wit_blue)∈nat_rel → blue_wit_rel → blue_wit_rel"
"(red_init_witness, red_init_witness) ∈ nat_rel→nat_rel→red_wit_rel"
"(prep_wit_red,prep_wit_red) ∈ nat_rel → red_wit_rel → red_wit_rel"
"(extract_res,extract_res) ∈ blue_wit_rel → wit_res_rel"
by *simp_all*

18.4.2 Actual Refinement

schematic_lemma *red_dfs_impl_refine_aux:*

```
  fixes u'::"nat" and V'::"nat set"
  notes [autoref_tyrel] =
    ty_REL[where 'a="nat set" and R="⟨nat_rel⟩iam_set_rel"]
  assumes [autoref_rules]:
    "(u,u')∈nat_rel"
    "(V,V')∈⟨nat_rel⟩iam_set_rel"
    "(onstack,onstack')∈⟨nat_rel⟩iam_set_rel"
    "(E,E')∈⟨nat_rel⟩slg_rel"
  shows "(RETURN (?f::?'c), red_dfs E' onstack' V' u') ∈ ?R"
  apply -
  unfolding red_dfs_def
  apply (autoref_monadic)
  done

concrete_definition red_dfs_impl uses red_dfs_impl_refine_aux
prepare_code_thms red_dfs_impl_def
declare red_dfs_impl.refine[autoref_higher_order_rule, autoref_rules]

schematic_lemma ndfs_impl_refine_aux:
  fixes s::"nat"
  notes [autoref_tyrel] =
    ty_REL[where 'a="nat set" and R="⟨nat_rel⟩iam_set_rel"]
  assumes [autoref_rules]:
    "(succi,E)∈⟨nat_rel⟩slg_rel"
    "(Ai,A)∈⟨nat_rel⟩iam_set_rel"
  notes [autoref_rules] = IdI[of s]
  shows "(RETURN (?f::?'c), blue_dfs E A s) ∈ ⟨?R⟩nres_rel"
  unfolding blue_dfs_def
  apply (autoref_monadic (trace))
  done

concrete_definition ndfs_impl for succi Ai s uses ndfs_impl_refine_aux

prepare_code_thms ndfs_impl_def
export_code ndfs_impl in SML

term "λE A v0. ndfs_impl (succ_of_list_impl E) (acc_of_list_impl A) v0"

term nat_of_integer

definition "succ_of_list_impl_int ≡
  succ_of_list_impl o map (λ(u,v). (nat_of_integer u, nat_of_integer v))"

definition "acc_of_list_impl_int ≡
  acc_of_list_impl o map nat_of_integer"

export_code
  ndfs_impl
  succ_of_list_impl_int
```

```
  acc_of_list_impl_int
  nat_of_integer
  integer_of_nat
  in SML module_name HPY_new
  file "nested_dfs.sml"

ML_val {*
  @{code ndfs_impl} (@{code succ_of_list_impl_int} [(1,2),(2,3),(2,7),(7,1)])
    (@{code acc_of_list_impl_int} [7]) (@{code nat_of_integer} 1)

*}


schematic_lemma ndfs_impl_refine_aux_old:
  fixes s::"nat"
  assumes [autoref_rules]:
    "(succi,E)∈⟨nat_rel⟩slg_rel"
    "(Ai,A)∈⟨nat_rel⟩dflt_rs_rel"
  notes [autoref_rules] = IdI[of s]
  shows "(RETURN (?f::?'c), blue_dfs E A s) ∈ ⟨?R⟩nres_rel"
  unfolding blue_dfs_def red_dfs_def
  using [[autoref_trace]]
  apply (autoref_monadic)
  done

end
```

19 Imperative Implementation of of Nested DFS (HPY-Improvement)

```
theory Sepref_NDFS
imports
  "../Sepref"
  "../../Collections/Examples/Autoref/Nested_DFS"
  "Sepref_Graph"
  "~~/src/HOL/Library/Code_Target_Numeral"
begin

typedecl 'v i_red_witness

lemma id_red_witness[id_rules]:
  "red_init_witness ::ᵢ TYPE('v ⇒ 'v ⇒ 'v i_red_witness option)"
  "prep_wit_red ::ᵢ TYPE('v ⇒ 'v i_red_witness option ⇒ 'v i_red_witness
option)"
  by simp_all

definition
  red_witness_rel_def_internal: "red_witness_rel R ≡ ⟨⟨R⟩list_rel,R⟩prod_rel"
```

```
lemma red_witness_rel_def: "⟨R⟩red_witness_rel ≡ ⟨⟨R⟩list_rel,R⟩prod_rel"
  unfolding red_witness_rel_def_internal[abs_def] by (simp add: relAPP_def)

lemma red_witness_rel_sv[constraint_rules]:
  "single_valued R ⟹ single_valued (⟨R⟩red_witness_rel)"
  unfolding red_witness_rel_def
  by (blast intro: relator_props)

lemma [sepref_fr_rules]: "hn_refine
  (hn_val R u u' * hn_val R v v')
  (return (red_init_witness u' v'))
  (hn_val R u u' * hn_val R v v')
  (hn_option_aux (pure (⟨R⟩red_witness_rel)))
  (RETURN$(red_init_witness$u$v))"
  apply simp
  unfolding red_init_witness_def
  apply rule
  apply (sep_auto simp: hn_ctxt_def pure_def red_witness_rel_def)
  done

lemma [sepref_fr_rules]: "hn_refine
  (hn_val R u u' * hn_option (pure (⟨R⟩red_witness_rel)) w w')
  (return (prep_wit_red u' w'))
  (hn_val R u u' * hn_option (pure (⟨R⟩red_witness_rel)) w w')
  (hn_option_aux (pure (⟨R⟩red_witness_rel)))
  (RETURN$(prep_wit_red$u$w))"
  apply rule
  apply (cases w)
  apply (sep_auto simp: hn_ctxt_def pure_def red_witness_rel_def)
  apply (cases w')
  apply (sep_auto simp: hn_ctxt_def pure_def red_witness_rel_def)
  apply (sep_auto simp: hn_ctxt_def pure_def red_witness_rel_def)
  done

schematic_lemma testsuite_red_dfs:
  fixes succ succ'
  notes [id_rules] =
    itypeI[of E "TYPE(nat i_slg)"]
    itypeI[of onstack "TYPE(nat set)"]
    itypeI[of V "TYPE(nat set)"]
    itypeI[of u "TYPE(nat)"]
  shows "hn_refine
    (   hn_ctxt (is_graph nat_rel) E succ'
      * hn_val nat_rel u u'
      * hn_ctxt is_ias V V'
      * hn_ctxt is_ias onstack onstack'
    )
    (?c::?'c Heap) ?Γ' ?R (red_dfs E onstack V u)"
```

```
  unfolding red_dfs_def
  by sepref

concrete_definition red_dfs_impl uses testsuite_red_dfs
prepare_code_thms red_dfs_impl_def
export_code red_dfs_impl checking SML_imp

lemma id_red_dfs[id_rules]:
  "red_dfs ::ᵢ TYPE(
    'a i_slg ⇒ 'a set ⇒ 'a set ⇒ 'a
    ⇒ ('a set * 'a i_red_witness option) nres)"
  by simp

lemma skel_red_dfs[sepref_la_skel]: "SKEL (red_dfs$E$os$V$s) = la_op
(E,os,V,s)"
  by simp

lemma id_init_wit_blue[id_rules]:
  "init_wit_blue ::ᵢ TYPE('a ⇒ 'a i_red_witness option ⇒ 'a blue_witness)"

  by simp

lemma hn_blue_wit[sepref_import_param]:
  "(NO_CYC,NO_CYC)∈blue_wit_rel"
  "(prep_wit_blue,prep_wit_blue)∈nat_rel→blue_wit_rel→blue_wit_rel"
  "(op=,op=)∈blue_wit_rel→blue_wit_rel→bool_rel"
  by simp_all

lemma hn_init_wit_blue[sepref_fr_rules]: "hn_refine
  (hn_val nat_rel v v' * hn_option (pure (⟨nat_rel⟩red_witness_rel)) w
w')
  (return (init_wit_blue v' w'))
  (hn_val nat_rel v v' * hn_option (pure (⟨nat_rel⟩red_witness_rel)) w
w')
  (pure blue_wit_rel)
  (RETURN$(init_wit_blue$v$w))"
  apply rule
  apply (sep_auto simp: hn_ctxt_def pure_def)
  apply (case_tac w, sep_auto)
  apply (case_tac w', sep_auto, sep_auto simp: red_witness_rel_def)
  done

lemma hn_extract_res[sepref_import_param]:
  "(extract_res, extract_res) ∈ blue_wit_rel → Id"
  by simp

lemma hn_red_dfs[sepref_fr_rules]:
  shows "hn_refine
   (  hn_ctxt (is_graph nat_rel) E succ'
```

```
    * hn_val nat_rel u u'
    * hn_ctxt is_ias V V'
    * hn_ctxt is_ias onstack onstack')
   (red_dfs_impl succ' u' V' onstack')
   (  hn_ctxt (is_graph nat_rel) E succ'
    * hn_ctxt is_ias onstack onstack'
    * hn_invalid V V'
    * hn_invalid u u')
   (hn_prod_aux is_ias (hn_option_aux (pure (⟨nat_rel⟩red_witness_rel))))
   (red_dfs$E$onstack$V$u)"
  using red_dfs_impl.refine by (simp add: star_aci)

schematic_lemma testsuite_blue_dfs:
  fixes succ' E A s
  notes [id_rules] =
    itypeI[of E "TYPE(nat i_slg)"]
    itypeI[of A "TYPE(nat set)"]
    itypeI[of s "TYPE(nat)"]
  shows "hn_refine
    (
        hn_ctxt (is_graph nat_rel) E succ'
      * hn_val nat_rel s s'
      * hn_ctxt is_ias A A'
    )
    (?c::?'c Heap) ?Γ' ?R (blue_dfs E A s)"
  unfolding blue_dfs_def
  by sepref

concrete_definition blue_dfs_impl for succ' A' s' uses testsuite_blue_dfs
prepare_code_thms blue_dfs_impl_def
export_code blue_dfs_impl checking SML_imp

theorem blue_dfs_impl_correct:
  fixes E
  assumes "finite (E*``{v0})"
  shows "<is_ias A A_impl * is_graph nat_rel E succ_impl>
      blue_dfs_impl succ_impl A_impl v0
    <λr. is_ias A A_impl * is_graph nat_rel E succ_impl
      * ↑(
        case r of None ⇒ ¬has_acc_cycle E A v0
      | Some (v,pc,pv) ⇒ is_acc_cycle E A v0 v pv pc
    )>t"
  using assms
  apply vcg
  apply (rule cons_post_rule)
  apply (rule imp_correctI)
  apply (rule hn_refine_cons_pre[rotated])
  apply (rule blue_dfs_impl.refine)
  apply (sep_auto simp: hn_ctxt_def pure_def)
```

```
  apply (rule blue_dfs_correct)
  apply simp
  apply (sep_auto simp: hn_ctxt_def pure_def)
  done
```

We tweak the initialization vector of the outer DFS, to allow pre-initialization of the size of the array-lists. When set to the number of nodes, array-lists will never be resized during the run, which saves some time.

```
lemma testsuite_blue_dfs_modify:
  "({}::nat set, {}::nat set, {}::nat set, s)
  = (empty_set_sz n, empty_set_sz n, empty_set_sz n, s)"
  by simp

schematic_lemma testsuite_blue_dfs_sz:
  fixes succ' E A s and n::nat
  notes [id_rules] =
    itypeI[of E "TYPE(nat i_slg)"]
    itypeI[of A "TYPE(nat set)"]
    itypeI[of s "TYPE(nat)"]
    itypeI[of n "TYPE(nat)"]
  shows "hn_refine
    (
        hn_ctxt (is_graph nat_rel) E succ'
      * hn_val nat_rel s s'
      * hn_val nat_rel n n'
      * hn_ctxt is_ias A A'
    )
    (?c::?'c Heap) ?Γ' ?R (blue_dfs E A s)"
  unfolding blue_dfs_def
  unfolding testsuite_blue_dfs_modify[where n=n]
  by sepref


concrete_definition blue_dfs_sz_impl
  for n' succ' A' s' uses testsuite_blue_dfs_sz
prepare_code_thms blue_dfs_sz_impl_def
export_code blue_dfs_sz_impl checking SML_imp

theorem blue_dfs_sz_impl_correct:
  fixes E
  assumes "finite (E*``{v0})"
  shows "<is_ias A A_impl * is_graph nat_rel E succ_impl>
      blue_dfs_sz_impl n succ_impl A_impl v0
    <λr. is_ias A A_impl * is_graph nat_rel E succ_impl
      * ↑(
        case r of None ⇒ ¬has_acc_cycle E A v0
      | Some (v,pc,pv) ⇒ is_acc_cycle E A v0 v pv pc
    )>t"
  using assms
```

```
  apply vcg
  apply (rule cons_post_rule)
  apply (rule imp_correctI)
  apply (rule hn_refine_cons_pre[rotated])
  apply (rule blue_dfs_sz_impl.refine)
  apply (sep_auto simp: hn_ctxt_def pure_def)
  apply (rule blue_dfs_correct)
  apply simp
  apply (sep_auto simp: hn_ctxt_def pure_def)
  done

end
theory Dijkstra_Benchmark
imports "../../../Examples/Sepref_Dijkstra"
  "../../../../Dijkstra_Shortest_Path/Test"
begin

definition nat_cr_graph_imp
  :: "nat ⇒ (nat × nat × nat) list ⇒ nat graph_impl Heap"
  where "nat_cr_graph_imp ≡ cr_graph"

definition nat_dijkstra_imp
  :: "nat ⇒ nat graph_impl ⇒ ((nat × nat × nat) list × nat) option
Heap.array Heap"
  where
  "nat_dijkstra_imp ≡ dijkstra_imp"

definition "nat_cr_graph_fun nn es ≡ hlg_from_list_nat ([0..<nn], es)"

export_code
  integer_of_nat nat_of_integer

  ran_graph

  nat_cr_graph_fun nat_dijkstra

  nat_cr_graph_imp nat_dijkstra_imp
  in SML_imp module_name Dijkstra
  file "dijkstra_export.sml"


end
theory NDFS_Benchmark
imports
  "../../../../Collections/Examples/Autoref/Nested_DFS"
  "../../../Examples/Sepref_NDFS"
begin
```

```
locale bm_fun begin

  schematic_lemma succ_of_list_impl:
    notes [autoref_tyrel] =
      ty_REL[where 'a="nat→nat set" and R="⟨nat_rel,R⟩dflt_rm_rel"
for R]
      ty_REL[where 'a="nat set" and R="⟨nat_rel⟩list_set_rel"]

    shows "(?f::?'c,succ_of_list) ∈ ?R"
    unfolding succ_of_list_def[abs_def]
    apply (autoref (keep_goal))
    done

  concrete_definition succ_of_list_impl uses succ_of_list_impl

  schematic_lemma acc_of_list_impl:
    notes [autoref_tyrel] =
      ty_REL[where 'a="nat set" and R="⟨nat_rel⟩dflt_rs_rel" for R]

    shows "(?f::?'c,acc_of_list) ∈ ?R"
    unfolding acc_of_list_def[abs_def]
    apply (autoref (keep_goal))
    done

  concrete_definition acc_of_list_impl uses acc_of_list_impl

  schematic_lemma red_dfs_impl_refine_aux:

    fixes u'::"nat" and V'::"nat set"
    notes [autoref_tyrel] =
      ty_REL[where 'a="nat set" and R="⟨nat_rel⟩dflt_rs_rel"]
    assumes [autoref_rules]:
      "(u,u')∈nat_rel"
      "(V,V')∈⟨nat_rel⟩dflt_rs_rel"
      "(onstack,onstack')∈⟨nat_rel⟩dflt_rs_rel"
      "(E,E')∈⟨nat_rel⟩slg_rel"
    shows "(RETURN (?f::?'c), red_dfs E' onstack' V' u') ∈ ?R"
    apply -
    unfolding red_dfs_def
    apply (autoref_monadic)
    done

  concrete_definition red_dfs_impl uses red_dfs_impl_refine_aux
  prepare_code_thms red_dfs_impl_def
  declare red_dfs_impl.refine[autoref_higher_order_rule, autoref_rules]

  schematic_lemma ndfs_impl_refine_aux:
```

```
    fixes s::"nat" and succi
    notes [autoref_tyrel] =
      ty_REL[where 'a="nat set" and R="⟨nat_rel⟩dflt_rs_rel"]
    assumes [autoref_rules]:
      "(succi,E)∈⟨nat_rel⟩slg_rel"
      "(Ai,A)∈⟨nat_rel⟩dflt_rs_rel"
    notes [autoref_rules] = IdI[of s]
    shows "(RETURN (?f::?'c), blue_dfs E A s) ∈ ⟨?R⟩nres_rel"
    unfolding blue_dfs_def
    apply (autoref_monadic (trace))
    done

  concrete_definition fun_ndfs_impl for succi Ai s uses ndfs_impl_refine_aux

  prepare_code_thms fun_ndfs_impl_def

  definition "fun_succ_of_list ≡
    succ_of_list_impl o map (λ(u,v). (nat_of_integer u, nat_of_integer
v))"

  definition "fun_acc_of_list ≡
    acc_of_list_impl o map nat_of_integer"

end

interpretation "fun"!: bm_fun .



locale bm_funs begin

  schematic_lemma succ_of_list_impl:
    notes [autoref_tyrel] =
      ty_REL[where 'a="nat→nat set" and R="⟨nat_rel,R⟩iam_map_rel"
for R]
      ty_REL[where 'a="nat set" and R="⟨nat_rel⟩list_set_rel"]

    shows "(?f::?'c,succ_of_list) ∈ ?R"
    unfolding succ_of_list_def[abs_def]
    apply (autoref (keep_goal))
    done

  concrete_definition succ_of_list_impl uses succ_of_list_impl

  schematic_lemma acc_of_list_impl:
    notes [autoref_tyrel] =
      ty_REL[where 'a="nat set" and R="⟨nat_rel⟩iam_set_rel" for R]

    shows "(?f::?'c,acc_of_list) ∈ ?R"
```

```
    unfolding acc_of_list_def[abs_def]
    apply (autoref (keep_goal))
    done

  concrete_definition acc_of_list_impl uses acc_of_list_impl

  schematic_lemma red_dfs_impl_refine_aux:

    fixes u'::"nat" and V'::"nat set"
    notes [autoref_tyrel] =
      ty_REL[where 'a="nat set" and R="⟨nat_rel⟩iam_set_rel"]
    assumes [autoref_rules]:
      "(u,u')∈nat_rel"
      "(V,V')∈⟨nat_rel⟩iam_set_rel"
      "(onstack,onstack')∈⟨nat_rel⟩iam_set_rel"
      "(E,E')∈⟨nat_rel⟩slg_rel"
    shows "(RETURN (?f::?'c), red_dfs E' onstack' V' u') ∈ ?R"
    apply -
    unfolding red_dfs_def
    apply (autoref_monadic)
    done

  concrete_definition red_dfs_impl uses red_dfs_impl_refine_aux
  prepare_code_thms red_dfs_impl_def
  declare red_dfs_impl.refine[autoref_higher_order_rule, autoref_rules]

  schematic_lemma ndfs_impl_refine_aux:
    fixes s::"nat" and succi
    notes [autoref_tyrel] =
      ty_REL[where 'a="nat set" and R="⟨nat_rel⟩iam_set_rel"]
    assumes [autoref_rules]:
      "(succi,E)∈⟨nat_rel⟩slg_rel"
      "(Ai,A)∈⟨nat_rel⟩iam_set_rel"
    notes [autoref_rules] = IdI[of s]
    shows "(RETURN (?f::?'c), blue_dfs E A s) ∈ ⟨?R⟩nres_rel"
    unfolding blue_dfs_def
    apply (autoref_monadic (trace))
    done

  concrete_definition funs_ndfs_impl for succi Ai s uses ndfs_impl_refine_aux

  prepare_code_thms funs_ndfs_impl_def

  definition "funs_succ_of_list ≡
    succ_of_list_impl o map (λ(u,v). (nat_of_integer u, nat_of_integer
v))"

  definition "funs_acc_of_list ≡
    acc_of_list_impl o map nat_of_integer"
```

```
end

interpretation "funs"!: bm_funs .

definition "imp_ndfs_impl ≡ blue_dfs_impl"
definition "imp_ndfs_sz_impl ≡ blue_dfs_sz_impl"
definition "imp_acc_of_list l ≡ From_List_GA.ias_from_list (map nat_of_integer
l)"
definition "imp_graph_of_list n l ≡ cr_graph (nat_of_integer n) (map (pairself
nat_of_integer) l)"

export_code
  nat_of_integer integer_of_nat
  fun.fun_ndfs_impl fun.fun_succ_of_list fun.fun_acc_of_list
  funs.funs_ndfs_impl funs.funs_succ_of_list funs.funs_acc_of_list
  imp_ndfs_impl imp_ndfs_sz_impl imp_acc_of_list imp_graph_of_list
in SML_imp module_name NDFS_Benchmark file "NDFS_Benchmark_export.sml"

ML_val ‹open Time›
end
```

References

[1] L. Bulwahn, A. Krauss, F. Haftmann, L. Erkök, and J. Matthews. Imperative functional programming with Isabelle/HOL. In *TPHOL*, volume 5170 of *LNCS*, pages 134–149. Springer, 2008.