Formalization of a near-linear time algorithm for solving the unification problem

Kasper F. Brandt <s152156@student.dtu.dk>

Master Thesis
Project title: Proof Assistants and Formal Verification
Supervisor: Jørgen Villadsen
Technical University of Denmark

January 28, 2018
Abstract

This thesis deals with formal verification of an imperatively formulated algorithm for solving first-order syntactic unification based on sharing of terms by storing them in a DAG. A theory for working with the relevant data structures is developed and a part of the algorithm is shown equivalent with a functional formulation.
Preface

This thesis is submitted as part of the requirements for acquiring a M.Sc. in Engineering (Computer Science and Engineering) at the Technical University of Denmark.

- Kasper Fabæch Brandt, January 2018
1 Introduction

1.1 Aim & Scope

The aim of this project is to formalize an imperative algorithm for solving first-order syntactic unification with better time complexity than the simpler functional formulation. The goal is not to show anything about the functional definition but rather to show equivalence between the imperative and functional definition.

The algorithm in question is given in part 4.8 of Term Rewriting and All That\cite{1}, henceforth known as TRaAT. The algorithm has a time complexity that is practically linear for all practical problem sizes.

A 'classical' functionally formulated algorithm (i.e. Martelli, Montanari\cite{3} derived) is already contained in the Isabelle distribution in the HOL-ex. Unification theory, so showing theory about the functional formulation is not considered necessary. The almost-linear algorithm however has so far not been formalized in Isabelle.

1.2 Overview

This report will first go into some of the theory behind unification, then it will discuss proving in Isabelle in relation to imperative algorithms. Then there will be taken a look at how the algorithm is formalized and how the equivalence is shown. Finally there will be some discussion about lessons learned and further work.

2 Theoretical background on unification

Unification is the problem of solving equations between symbolic expressions. Specifically this thesis focuses on what is known as first-order unification. An example of an instance of a problem we would like to solve could be:

\[
\begin{align*}
  f(g(x), x) & \equiv f(z, a) \\
  z & \equiv y
\end{align*}
\]

What we are given here is a set of equations \( S = f(g(x), x) = f(z, a), z = y \) with the variables \( x, y, z \), constant \( a \) and functions \( f, g \). The constituent parts of each of the expressions are called terms.

Definition 1 (Term). A term is defined recursively as:

- A variable, an unbound value. The set of variables occurring in a term \( t \) is denoted as \( \text{Var}(t) \). In this treatment the lowercase letters \( x, y \) and \( z \) to are used to denote variables.
• A function. A function consists of a function symbol and a list of terms. The arity of the function is given by the length of the list. In this treatment all occurrences of a function symbol are required to have the same arity for the problem instance to be well-formed. Functions are in this treatment denoted by the lowercase letters $f$ and $g$.

• A constant. Constants are bound values that cannot be changed. Constants can be represented as functions of arity zero, which simplifies analysis and data structures, so this representation will be used here. Constants are denoted by the lowercase letters $a, b, c$ in this treatment.

The lowercase letter $t$ is used to denote terms.

The goal is now to put the equations into solved form.

**Definition 2 (Solved form).** A unification problem $S = x_1 = t_1, \ldots, x_n = t_n$ is in solved form if all the variables $x_i$ are pairwise distinct and none of them occurs in any of the terms $t_i$.

### 2.1 Martelli, Montanari / functional version in TRaAT

The following algorithm is presented in TRaAT and is based on the algorithm given by Martelli and Montanari[3]

\begin{align*}
\{t ? = t\} \cup S & \implies S & \text{(Delete)} \\
\{f(T_n) ? = f(U_n)\} \cup S & \implies \{t_1 ? = u_1, \ldots, t_n ? = u_n\} \cup S & \text{(Decompose)} \\
\{t ? = x\} \cup S & \implies \{x ? = t\} \cup S \text{ if } t \notin V & \text{(Orient)} \\
\{x ? = t\} \cup S & \implies \{x ? = t\} \cup \{x \mapsto t\}(S) & \text{(Eliminate)}
\end{align*}

if $x \in \text{Var}(S) - \text{Var}(t)$

TRaAT gives a formulation in ML. Besides minor syntactical differences and raising an exception rather than returning None it is identical to the formulation in appendix A.2.

One interesting thing to note here is the pattern match of function in solve is given as

\[
\text{solve}((T(f, ts), T(g, us)) :: S, s) = \\
\text{if } f = g \text{ then solve(zip(ts, us) @ S, s) else raise UNIFY}
\]

Since zip truncate additional elements this will cause erroneous unification if the arity of the functions differ, so presumably this excepts the arity of the same function to always match.
3 Formal verification with Isabelle

3.1 Formalization of imperative algorithms

The language for writing functions in Isabelle is a pure function language. This means that imperative algorithms generally cannot be written directly. Anything with side effects must be modeled as changes in a value representing the environment instead. Isabelle has some theories for working with imperative algorithms in the standard library, namely in the session HOL-Imperative_HOL which is based on [1].

3.1.1 The heap and references

One of the primary causes for side-effects in imperative programs is the usage of a heap. A heap can formally be described as a mapping from addresses to values, this is also how it is defined in the theory HOL-Imperative_HOL.

\begin{verbatim}
class heap = typerep + countable

type-synonym addr = nat  — untyped heap references
type-synonym heap-rep = nat  — representable values

record heap =
  arrays :: typerep ⇒ addr ⇒ heap-rep list
  refs :: typerep ⇒ addr ⇒ heap-rep
  lim :: addr

datatype 'a ref = Ref addr  — note the phantom type 'a
\end{verbatim}

Arrays and references are treated separately by the theory for simplicity, however this thesis makes no usage of arrays so they can be ignored here. refs is the map of addresses to values. A uniform treatment of all types is made possible by representing values as natural numbers. This is necessary since a function cannot directly be polymorphic in Isabelle. For this to work the types on the heap must be countable. This requirement is ensured by the functions for dereferencing and manipulating references requiring the type parameter 'a to be of typeclass heap, which requires it to be countable.

We should note the other requirement of a type representation. A type-rep is an identifier associated with types that uniquely identifies the type. For types defined the usual way such as with \texttt{datatype} these are automatically defined. The reason for this requirement is that it is necessary to know that the type stored in refs is the same as the one read to show anything about the value.
The limit value of the heap is the highest address currently allocated. While the typerep is part of the key for the map, the address does uniquely determine a value as long as only the provided functions for manipulating the heap are used rather than manipulating the fields of the record directly. Heap.alloc is used for allocating a new reference, and this function returns a new heap with an increased limit.

To illustrate how references can be used in practice we consider a very simple example

```ml
datatype ilist = INil | ICons nat × ilist ref

instantiation ilist :: heap begin
  instance by countable-datatype
end

function length :: heap ⇒ ilist ⇒ nat where
  length h INil = 0
  | length h (ICons(_, lsr)) = 1 + length h (Ref.get h lsr)
by (pat-completeness, auto)
```

This defines a singly linked list and a function for getting the length of one. It should be noted that length here is a partial function because it does not terminate if given a circular list.

A bit more complicated example could be reversing a list,

```ml
fun cons :: heap ⇒ nat ⇒ ilist ref ⇒ (ilist ref × heap) where
  cons h v ls = Ref.alloc (ICons(v, ls)) h

function rev0 :: ilist ref ⇒ heap ⇒ ilist ⇒ (ilist ref × heap) where
  rev0 _2 h INil = (_2, h)
  | rev0 _2 h (ICons(v, lsr)) = (let ls = Ref.get h lsr in
    let (_2′, h′) = cons h v _2 in
    rev0 _2′ h′ ls)
by (pat-completeness, auto)

definition rev where
  rev h = (let (nilr, h′) = Ref.alloc INil h in rev0 nilr h)
```

It quickly becomes clear that this is very cumbersome to write when we have to explicitly move the modified heap along. It should also be noted
that Imperative-HOL does not support code generation when used this way if we wanted to use that.

The theory HOL-Imperative_HOL.Heap_Monad defines a monad over the raw heap which makes it easier and clearer to use and also supports code generation, using this the code becomes

fun cons:: nat ⇒ ilist ref ⇒ ilist ref Heap where
cons v ls = ref (ICons(v, ls))

function rev0:: ilist ref ⇒ ilist ⇒ ilist ref Heap where
rev0 l2 INil = return l2
| rev0 l2 (ICons(v, lsr)) = do { 
ls ← !lsr;
ls′ ← cons v ls;
rev0 ls′ ls}
by (pat-completeness, auto)

definition rev where rev l = do { nilr ← ref INil; rev0 nilr l }

This is much clearer, however it still does not work so well. For one code generation still does not work, but a bigger problem is that the generated theorems for evaluation of the function are useless.

3.1.2 Partial functions and induction on them

As stated earlier we cannot guarantee that an ilist does not link back to itself. This is an inherent problem in using structures with references since we cannot directly in the type definition state that it cannot contain cyclic reference since that would require parameterization over the heap value.

So that means that we are stuck with working with partial functions. All functions in Isabelle are actually total[2]. What happens when a function is declared in Isabelle without a termination proof is that all the theorems for evaluation, usually named as (function name).simps, becomes guarded with an assertion that the value is in the domain of the function. The same is true for the inductions rules. For example for the rev0 function given above gets the following theorem statement for rev0.psimps:

rev0-dom (?!2.0, INil) \(\Rightarrow\) rev0 ?!2.0 INil = return ?!2.0
rev0-dom (?!2.0, ICons (?v, ?lsr)) \(\Rightarrow\) rev0 ?!2.0 (ICons (?v, ?lsr)) = !?lsr \(\Rightarrow\) (λls. cons ?v ?!2.0 \(\Rightarrow\) (λl2′. rev0 l2′ ls))

The \(\Rightarrow\) operator indicates monadic binding, this is what the do notation expands to. More importantly the theorems are guarded by the rev0-dom predicate. Now we should have been able to show which values are in the domain. This is done by adding the (domintros) attribute to the function
which generate introduction theorems for the domain predicate. However in
turns out that the function package has some limitations to this functional-
ity. In this case the two theorems generated are

\[
\text{rev0-dom (?l2.0, INil)}
\]

and

\[
(\lambda x. \text{rev0-dom (xa, x)}) \Rightarrow \text{rev0-dom (?l2.0, ICons (?v, ?lsr))}
\]

The first theorem is trivial. However the second one is useless, we
can only show that the predicate holds for a value if it holds for every
value. What we need to do here is to instead use the \textbf{partial-function}
command\cite{4}. Unfortunately this does not support writing functions with
pattern matching as well as mutual recursion. The lack of pattern matching
directly in the definition is easily worked around by using an explicit case
statement, however it does make the definition somewhat more unwieldy as
well as making it harder for automated tools to work with it. To implement
mutually recursive functions it becomes necessary to explicitly use a sum
type instead.

The definition of the \texttt{rev0} using this becomes

\begin{verbatim}
fun cons:: nat ⇒ ilist ref ⇒ ilist ref Heap where
cons v ls = ref (ICons(v, ls))

partial-function (heap) rev0:: ilist ref ⇒ ilist ⇒ ilist ref Heap where
[code]:
rev0 l2 l = (case l of
  INil ⇒ return l2
  | ICons(v, lsr) ⇒ do { l2' ← cons v l2;
                          rev0 l2' ls })

definition rev where rev l = do { nilr ← ref INil; rev0 nilr l }
\end{verbatim}

This generates some more useful theorems, \texttt{rev0.simps} becomes:

\[
\text{rev0 ?l2.0 ?l = (case ?l of INil ⇒ return ?l2.0 | ICons (v, ls) ⇒ !lsr ⇒ (λs. cons v ?l2.0 ⇒ (λt2'. rev0 t2' ls))}}
\]

Note that there is no guard this time. Induction rules for fixpoint in-
duction are also introduced, however for the concrete problems solved here
structural induction over the datatypes are used instead.
3.2 Working with the Heap monad

When it comes to working with functions defined using the Heap monad a way to talk about the result is needed. The function execute :: 'a Heap ⇒ heap ⇒ ('a × heap) option. The result of execute is an option with a tuple consisting of the result of the function and the updated heap. The reason it is wrapped in option is that the heap supports exceptions. This feature is not used anywhere in the theory developed but it does make it a bit more cumbersome to use the heap monad.

The Heap-Monad theory also contains the predicate effect :: 'a Heap ⇒ heap ⇒ heap ⇒ 'a ⇒ bool. effect x h h' r asserts that the result of x on the heap h is r with the modified heap h'.

The lemmas and definitions related to the value of the heap are not added to the simp method, which means that evaluating a function using the Heap monad becomes a somewhat standard step of using (simp add: lookup-def tap-def bind-def return-def execute-heap).

4 Formalization of the algorithms

4.1 The functional version

The functional algorithm is a completely straightforward translation of the one given in ML in TRaAT. Besides syntactical difference the only difference is that this version has the result of wrapped in an option and returns None rather than raises an exception if the problem is not unifyable.

4.2 The imperative version

The imperative version is given as Pascal code in TRaAt so it needs some adaption.

```
type  termP = ^term
    termsP = ^terms

term = record
    stamp: integer;
    is : termP;
    case isvar : boolean of
    true: ();
    false: (fn: string; args: termsP)
end;

terms = record t:termP; next: termsP end;
```

The most direct translation would be to also define terms as records in Isabelle, however the record command does not currently support mutual recursion so we have to do with a regular datatype definition. The definition is given as
The references do not have a "null-pointer". They can be invalid by pointing to addresses higher than limit, but that is not really helpful since they would become valid once new references are allocated. So pointers are instead modeled by ref options where None represents a null pointer.

Since Isabelle does not have the sort of tagged union with explicit tag as Pascal do the isvar field is not directly present, rather it is implicit in whether the data part (i-term-d) is IVarD or ITermD.

The definition of union in TRaAT merely updates the is pointer, however since the the values are immutable we must replace the whole record on update, so for simplicity sake a term that points to another term is always marked as IVarD, this does not matter to the algorithm since the function list is never read from terms with non-null is pointer. In fact in the theory about the imperative terms we consider a term with non-null is pointer and ITermD as data part as invalid.

The functions from TRaAT are translated as outlined outlined in section 3.1.

4.3 Theory about the imperative datastructures

The terms need to represent an acyclic graph for the algorithm to terminate, this is asserted by the mutually recursively defined predicates i-term-acyclic and i-terms-acyclic:

\[
\begin{align*}
\textbf{inductive } & \text{i-term-acyclic} :: \text{heap } \Rightarrow \text{i-termP } \Rightarrow \text{bool and } \\
\text{i-terms-acyclic} :: \text{heap } \Rightarrow \text{i-termsP } \Rightarrow \text{bool where } \\
\text{t-acyclic-nil} & : \text{i-term-acyclic - None } | \\
\text{t-acyclic-step-link} & : \text{i-term-acyclic h t } \implies \\
& \text{Ref.get h tref } = \text{ITerm(\_ , t, IVarD) } \implies \\
\text{t-acyclic-step-ITerm} & : \text{i-terms-acyclic h tsref } \implies \\
& \text{Ref.get h tref } = \text{ITerm(\_ , None, ITermD(\_ , tsref)) } \implies
\end{align*}
\]
i-term-acyclic h (Some tref) | 

ts-acyclic-nil: i-terms-acyclic - None | 

ts-acyclic-step-ITerms: 

i-terms-acyclic h ts2ref ⇒ 

i-term-acyclic h (Some tref) ⇒ 

Ref.get h tsref = ITerms (tref, ts2ref) ⇒ 

i-terms-acyclic h (Some tsref)

As noted earlier terms representing a function (with ITermD) are only considered valid if the is pointer is null (i.e. None).

A form of total induction is required where we can take as induction hypothesis that a predicate is true for every term 'further down' in the DAG. The base of this is the i-term-closure set. This is to be understood as the transitive closure of referenced terms.

\[
\text{inductive-set \ i-term-closure \ for \ h:: \ heap \ and \ tp:: \ i-termP \ where}
\]

\[
\begin{align*}
\text{Some tr = tp} & \implies \text{tr} \in \text{i-term-closure \ h \ tp} \\
\text{tr} & \in \text{i-term-closure \ h \ tp} \implies \\
\text{Ref.get h tr = ITerm(-, Some is, -)} & \implies \\
\text{is} & \in \text{i-term-closure \ h \ tp} \\
\text{tr} & \in \text{i-term-closure \ h \ tp} \implies \\
\text{Ref.get h tr = ITerm(-, None, ITermD(-, tsp))} & \implies \\
\text{tr2} & \in \text{i-terms-set \ h \ tsp} \implies \\
\text{tr2} & \in \text{i-term-closure \ h \ tp}
\end{align*}
\]

Related to this are the i-terms-sublists and i-term-chain. The former gives the set of i-terms referenced from a i-terms, i.e. the sublists of the list represented by the i-terms. The latter gives the set of terms traversed through the is pointers from a given term. Derived from i-terms-sublists is also define i-terms-set which is the set of terms referenced by the list. Closure and sublists over i-terms are also defined

\[
\text{abbreviation \ i-term-closures \ where}
\]

\[
\text{i-term-closures \ h \ trs} \equiv (\ast \cup \text{(i-term-closure \ h \ ' \ Some \ ' \ trs)*}) \\
\text{UNION (Some \ ' \ trs) \ (i-term-closure \ h)}
\]

\[
\text{abbreviation \ i-terms-closure \ where}
\]

\[
\text{i-terms-closure \ h \ tsp} \equiv \text{i-term-closures \ h \ (i-terms-set \ h \ tsp)}
\]

\[
\text{abbreviation \ i-term-sublists \ where}
\]

\[
\text{i-term-sublists \ h \ tr} \equiv \text{i-terms-sublists \ h \ (get-ITerm-args \ (Ref.get \ h \ tr)}
\]

\[
\text{abbreviation \ i-term-closure-sublists \ where}
\]

12
i-term-closure-sublists h tp \equiv (\ast \cup (\text{i-term-sublists h \ ' \ i-term-closure h tr})
\cup \text{tr}' \in \text{i-term-closure h tp. i-term-sublists h tr'}
\bigcup \text{tr}'

\text{abbreviation} \ i-terms-closure-sublists \ \text{where}
\text{i-terms-closure-sublists h tsp} \equiv (\ast \cup (\text{i-term-sublists h tsp} \cup \text{i-term-closure h tsp})
\cup \text{i-term-sublists h tsp} \cup (\bigcup \text{tr}' \in \text{i-terms-closure h tsp. i-term-sublists h tsp. i-term-sublists h tsp})

To meaningfully work with changes to the heap we need a predicate asserting that the structure of a term graph is unchanged, this is captured by heap-only-stamp-changed.

\text{abbreviation} \ i-term-closures \ \text{where}
\text{i-term-closures h trs} \equiv \text{UNION} (\text{Some} \ ' \ \text{trs}) (\text{i-term-closure h})

\text{abbreviation} \ i-terms-closure \ \text{where}
\text{i-terms-closure h tsp} \equiv \text{i-term-closures h (i-terms-set h tsp)}

\text{abbreviation} \ i-term-sublists \ \text{where}
\text{i-term-sublists h tr} \equiv \text{i-terms-sublists h (get-ITerm-args (Ref.get h tr))}

\text{abbreviation} \ i-term-closure-sublists \ \text{where}
\text{i-term-closure-sublists h tp} \equiv (\bigcup \text{tr}' \in \text{i-term-closure h tp. i-term-sublists h tr'})

\text{abbreviation} \ i-terms-closure-sublists \ \text{where}
\text{i-terms-closure-sublists h tsp} \equiv \text{i-terms-closure-sublists h tsp} \cup (\bigcup \text{tr}' \in \text{i-terms-closure h tsp. i-term-sublists h tsp. i-term-sublists h tr'})

More specifically it asserts that only changes to terms in the set trs are made, and the only the stamp value is changed, and no changes are made to any i-terms and nats. This is used as basis for a total induction rule where the induction hypothesis asserts that the predicate is true for every term further down the graph and every heap where that the closure of that term is unchanged.

\text{lemma} \ \text{acyclic-closure-ch-stamp-inductc}' \ [\text{consumes} \ 1, \ \text{case-names} \ \text{var link args terms-nil terms}]:
\text{fixes} \ h:: \ \text{heap}
\text{and} \ \text{tr:: i-term ref}
\textbf{and} P1:: heap ⇒ i-term ref set ⇒ i-term ref ⇒ bool
\textbf{and} P2:: heap ⇒ i-term ref set ⇒ i-termsP ⇒ bool
\textbf{assumes} acyclic: i-term-acyclic h (Some tr)
\textbf{and} var-case: \(\bigwedge h \ trs \ tr \ s\).
\hspace{1em} Ref.get h tr = ITerm(s, None, IVarD) \implies P1 h trs tr
\textbf{and} link-case: \(\bigwedge h \ trs \ tr \ isr \ s\).
\hspace{1em} (\(\bigwedge t2r \ h \ trs\).
\hspace{2em} trs \subseteq trs' \implies
\hspace{3em} heap-only-stamp-changed trs' h h' \implies
\hspace{4em} t2r \in i-term-closure h (Some isr) \implies
\hspace{5em} P1 h' trs' t2r) \implies
\hspace{1em} Ref.get h tr = ITerm(s, Some isr, IVarD) \implies P1 h trs tr
\textbf{and} args-case: \(\bigwedge h \ trs \ tr \ tsp \ s \ f\).
\hspace{1em} (\(\bigwedge h' \ trs\).
\hspace{2em} trs \subseteq trs' \implies
\hspace{3em} heap-only-stamp-changed trs' h h' \implies
\hspace{4em} P2 h' trs' tsp) \implies
\hspace{1em} (\(\bigwedge h' \ trs' t2r0 \ t2r\).
\hspace{2em} trs \subseteq trs' \implies
\hspace{3em} heap-only-stamp-changed trs' h h' \implies
\hspace{4em} t2r \in i-term-closure h (Some t2r0) \implies
\hspace{5em} t2r0 \in i-terms-set h tsp \implies
\hspace{6em} P1 h' trs' t2r) \implies
\hspace{1em} Ref.get h tr = ITerm(s, Some isr, ITermD(f, tsp)) \implies P1 h trs tr
\textbf{and} terms-nil-case: \(\bigwedge h \ trs \ P2 h \ trs \ None\)
\textbf{and} terms-case: \(\bigwedge h \ trs \ trsr \ tsnextp\).
\hspace{1em} (\(\bigwedge h' \ trs\).
\hspace{2em} trs \subseteq trs' \implies
\hspace{3em} heap-only-stamp-changed trs' h h' \implies
\hspace{4em} P2 h' trs' tsnextp) \implies
\hspace{1em} (\(\bigwedge h' \ trs' t2r\).
\hspace{2em} trs \subseteq trs' \implies
\hspace{3em} heap-only-stamp-changed trs' h h' \implies
\hspace{4em} t2r \in i-term-closure h (Some tr) \implies
\hspace{5em} P1 h' trs' t2r) \implies
\hspace{1em} Ref.get h trsr = ITerms (tr, tsnextp) \implies P2 h trs (Some tsr)
\textbf{shows} P1 h trs tr
5 Soundness of the imperative version

It is shown that the imperative version of the occurs is equivalent to the functional version. More specifically it is shown that given a wellformed i-term then the imperative version of occurs gives the same result as the functional version on the terms converted into their "functional" version. It is not shown that functional terms converted into imperative still gives the same result so it only shows soundness (relative to the functional formulation).

5.1 Conversion of imperative terms to functional terms

The function i-term-to-term-p converts i-term and i-terms into term and term list. The imperative terms does not contains names for the variables so we have to invent names for them. This is done by naming them as x followed by the heap address of the term.

i-term-to-term-p needs to be defined as a single function taking a sum type of i-term and i-terms because of the limitation of partial_function not allowing mutually recursive definitions. Separate i-term-to-term and i-terms-to-terms functions are defined and simpler evaluation rules are shown.

It is also shown that the term conversion functions are unaffected by changing the stamp of terms which is necessary in the proof for soundness of the imperative occurs.

5.2 Soundness of imperative occurs

The i-occ-p of a term is shown to equivalent to the occurs function on the term converted to its functional version given the following is satisfied:

1. The term, tr, must be acyclic
2. The variable to check for occurring, vr, must indeed be a variable
3. The stamp of all terms must be less than the current time

1. is asserted by i-term-acyclic and 3. is asserted by predicate stamp-current-not-occurs.

To show this it was necessary to identify which invariants holds. On entering with a term (representing occ) the above holds. When returning it holds that

1. The result is the same as occurs on the converted term.
2. Only changes are made to terms in the closure of tr and only the stamp is changed.
3. Either the stamp of all terms in the closure of tr are less than the current time, or vr did occur in tr.

On entering with a term list, tsp, (this corresponds to the occs function) the following holds

1. vr must be a variable under the current heap
2. For all terms in the list tsp the current stamp (i.e. time) does not occur.
3. tsp is acyclic

When returning it holds that

1. The result is whether vr occurs in any of the terms in tsp converted to functional terms.
2. Either the stamp of all terms in the closure of tsp are less than the current time, or vr did occur in one of the terms of tsp.

Since this proofs demonstrates that the algorithm always have a value when the requirements are fulfilled it also implicitly shows termination.

The final thing shown is that i-occurs is equivalent to the occurs function on the term converted to its functional version. The requirements are the same except that all stamps must be less than or equal to the time - since the function is adding one to time before calling i-occ-p. Besides the equivalence it is also shown that the resulting heap has 1 added to time and otherwise the only changes are to the stamp of terms in the closure of tr, and that the current time (after increment) either not occurs in the new heap or the occurs check is true.

6 Conclusion

6.1 Discussion

It was originally the goal to show full equivalence between the imperative DAG based algorithm and the functional algorithm. However it turned out to be incredibly difficult to work with imperative algorithms this way. A development of no less than 3300 lines of Isabelle code was necessary just to be able to reasonably work with the data structures. To add to that the lack of natural induction rules because of the partiality makes the function definitions harder to work with, as well as the function definitions being unwieldy because of the lack of support for mutual recursion and pattern matching directly in the function definition for the partial_function method. The
fact that any updates to references changes the heap also makes it very difficult to work with because it must be shown for every function whether they are affected by those specific changes to the heap.

Other approaches that might be worth looking into for working with imperative algorithms are the support for Hoare triples and using refinement frameworks. Hoare triples can often be more natural to work with for imperative algorithms. Refinement frameworks allows defining an algorithm in an abstract way and refining into equivalent concrete algorithms that may be harder to work with directly. The latter was attempted in the development of this thesis, however it was eventually dropped due to a large amount of background knowledge necessary combined with a lack of good documentation and examples.

6.2 Future work

Completeness of the occurs check relative to the functional definition as well as an equivalence proof of solve and unify would be obvious targets for future work. It may also be worth to look into the feasibility of other approaches.

7 References


Appendices

A Isabelle theory

A.1 Miscellaneous theory
t

theory Unification-Misc
imports Main
begin

zipping two lists and retrieving one of them back by mapping \texttt{fst} or \texttt{snd} results in the original list, possibly truncated

\textbf{lemma sublist-map-fst-zip:}

\begin{align*}
\text{fixes } & \text{xs: 'a list} \\
\text{and } & \text{ys: 'a list} \\
\text{obtains } & \text{xss} \\
\text{where } & \text{(map \texttt{fst} (zip xs ys)) @ xss = xs} \\
\text{by (induct xs ys rule: list-induct2', auto)}
\end{align*}

\textbf{lemma sublist-map-snd-zip:}

\begin{align*}
\text{fixes } & \text{xs: 'a list} \\
\text{and } & \text{ys: 'a list} \\
\text{obtains } & \text{yss} \\
\text{where } & \text{(map \texttt{snd} (zip xs ys)) @ yss = ys} \\
\text{by (induct xs ys rule: list-induct2', auto)}
\end{align*}

end

\section*{A.2 Functional version of the algorithm}

\textbf{theory Unification-Functional}

\textbf{imports Main}

\begin{flushleft}
Unification-Misc
\end{flushleft}

\textbf{begin}

datatype \texttt{term} =
\begin{align*}
V \ vname \\
T \ & \text{string } \times \text{term list}
\end{align*}

type-synonym \texttt{vname = string } \times \text{int}

datatype \texttt{subst} = (\texttt{vname } \times \texttt{term}) \text{list}

definition \texttt{indom :: vname } \Rightarrow \texttt{subst } \Rightarrow \texttt{bool where}
\begin{align*}
\text{indom } x \ s &= \text{list-ex } (\lambda(y, -). \ x = y) \ s
\end{align*}

fun \texttt{app :: subst } \Rightarrow \texttt{vname } \Rightarrow \texttt{term where}
\begin{align*}
\text{app } ((y,t)#s) \ & \text{x} = (\text{if } x = y \ \text{then } t \ \text{else } \text{app } s \ x) \\
\text{app } [] \ = \text{undefined}
\end{align*}

fun \texttt{lift :: subst } \Rightarrow \texttt{term } \Rightarrow \texttt{term where}
\begin{align*}
\text{lift } & \text{s } (V \ x) = (\text{if } \text{indom } x \ s \ \text{then } \text{app } s \ x \ \text{else } V \ x) \\
\text{lift } & \text{s } (T(f,ts)) = T(f, \text{map } \texttt{lift} \ s \ ts)
\end{align*}

fun \texttt{occurs :: vname } \Rightarrow \texttt{term } \Rightarrow \texttt{bool where}
\begin{align*}
\text{occurs } x \ (V \ y) & = (x = y) \\
\text{occurs } x \ (T(-,ts)) & = \text{list-ex } (\text{occurs } x) \ ts
\end{align*}
context
begin

private definition vars :: term list ⇒ vname set where
vars S = {x. ∃ t ∈ set S. occurs x t}

private lemma vars-nest-eq:
fixes ts :: term list
and S :: term list
and vn :: string
shows vars (ts @ S) = vars (T(vn, ts) # S)
unfolding vars-def
by (induction ts, auto)

private lemma vars-concat:
fixes ts:: term list
and S:: term list
shows vars (ts @ S) = vars ts ∪ vars S
unfolding vars-def
by (induction ts, auto)

private definition vars-eqs :: (term × term) list ⇒ vname set where
vars-eqs l = vars (map fst l) ∪ vars (map snd l)

lemma vars-eqs-zip:
fixes ts:: term list
and us:: term list
and S:: term list
shows vars-eqs (zip ts us) ⊆ vars ts ∪ vars us
using vars-concat sublist-map-fst-zip sublist-map-snd-zip vars-eqs-def
by (metis (no-types, hide-lams) Un-subset-iff sup.cobounded1 sup.coboundedI2)

private lemma vars-eqs-concat:
fixes ts:: (term × term) list
and S:: (term × term) list
shows vars-eqs (ts @ S) = vars-eqs ts ∪ vars-eqs S
using vars-concat vars-eqs-def by auto

private lemma vars-eqs-nest-subset:
fixes ts :: term list
and us :: term list
and S :: (term × term) list
and vn :: string
and wn :: string
shows vars-eqs (zip ts us @ S) ⊆ vars-eqs ((T(vn, ts), T(wn, us)) # S)
proof
have vars-eqs ((T(vn, ts), T(wn, us)) # S) = vars ts ∪ vars us ∪ vars-eqs S
using vars-concat vars-eqs-def vars-nest-eq by auto
then show ?thesis

19
using vars-eqs-concat vars-eqs-zip by fastforce
qed

private definition n-var :: (term × term) list ⇒ nat where
n-var l = card (vars-eqs l)

private lemma var-eqs-finite:
fixes ts
shows finite (vars-eqs ts)
proof -
{
  fix t
  have finite ({x. occurs x t})
  proof (induction t rule: occurs.induct)
    case (1 x y)
    then show ?case by simp
  next
    case (2 x fn ts)
    have {x. occurs x (T (fn, ts))} = vars ts
      using vars-def Bex-set-list-ex
    by fastforce
    then show ?case using vars-def 2.HI by simp
  qed
}
then show ?thesis
  using vars-def vars-eqs-def by simp
qed

private lemma vars-eqs-subset-n-var-le:
fixes S1 :: (term × term) list
and S2 :: (term × term) list
assumes vars-eqs S1 ⊆ vars-eqs S2
shows n-var S1 ≤ n-var S2
using assms var-eqs-finite n-var-def
by (simp add: card-mono)

private lemma vars-eqs-psubset-n-var-lt:
fixes S1 :: (term × term) list
and S2 :: (term × term) list
assumes vars-eqs S1 ⊂ vars-eqs S2
shows n-var S1 < n-var S2
using assms var-eqs-finite n-var-def
by (simp add: psubset-card_mono)

private fun fun-count :: term list ⇒ nat where
fun-count [] = 0
| fun-count ((V -)#S) = fun-count S
| fun-count (T(-, ts)#S) = 1 + fun-count ts + fun-count S
private lemma fun-count-concat:
  fixes ts :: term list
  and us :: term list
  shows fun-count (ts @ us) = fun-count ts + fun-count us
proof (induction ts)
case Nil
  then show ?case
  by force
next
case (Cons a ts)
  show ?case
  proof (cases a)
    case (V -)
    then have fun-count ((a # ts) @ us) = fun-count (ts @ us)
      by simp
    then show ?thesis
      by (simp add: Cons.IH V)
  next
    case (T x)
    then obtain fn ts’ where ts’-def: x=(fn, ts’)
      by fastforce
    then have fun-count ((a # ts) @ us) = 1 + fun-count (ts @ us) + fun-count ts’
      by (simp add: T)
    then show ?thesis
      by (simp add: Cons.IH T ts’-def)
  qed
qed

private definition n-fun :: (term × term) list ⇒ nat where
  n-fun l = fun-count (map fst l) + fun-count (map snd l)

private lemma n-fun-concat:
  fixes ts us
  shows n-fun (ts @ us) = n-fun ts + n-fun us
unfolding n-fun-def using fun-count-concat
  by simp

private lemma n-fun-nest-head:
  fixes ts g us S
  shows n-fun (zip ts us @ S) + 2 ≤ n-fun ((T (g, ts), T (g, us)) # S)
proof –
  let ?trunc-ts = (map fst (zip ts us))
  let ?trunc-us = (map snd (zip ts us))
  have trunc-sum: n-fun ((T (g, ?trunc-ts), T (g, ?trunc-us)) # S) = 2 + n-fun
    (zip ts us @ S)
    using n-fun-concat n-fun-def by auto

obtain tsp where ts-rest: (map fst (zip ts us)) @ tsp = ts by (fact sublist-map-fst-zip)
obtain usp where us-rest: \((\text{map \ snd \ (\text{zip \ ts \ us})}) \circ \text{usp} = \text{us}\) by (\text{fact sublist-map-snd-zip})

have fun-count \([T(g, \ ?\text{trunc-ts})] + \text{fun-count \ tsp} = \text{fun-count \ [T(g, \ ts)]}\)
  using ts-rest fun-count-concat
  by (metis add.assoc add.right-neutral fun-count.simps(1) fun-count.simps(3))

moreover have fun-count \([T(g, \ ?\text{trunc-us})] + \text{fun-count \ usp} = \text{fun-count \ [T(g, \ us)]}\)
  using us-rest fun-count-concat
  by (metis add.assoc add.right-neutral fun-count.simps(1) fun-count.simps(3))

ultimately have \(\text{n-fun \ ([T(g, \ ?\text{trunc-ts})] + \text{fun-count \ tsp} + \text{fun-count \ usp}) = \text{fun-count \ [T(g, \ ts)] + \text{fun-count \ [T(g, \ us)]}}\)
  by simp add: n-fun-def fun-count.simps(1) fun-count.simps(3)

then have \(\text{n-fun \ ([T(g, \ ?\text{trunc-ts})] + \text{fun-count \ tsp} + \text{fun-count \ usp}) = \text{fun-count \ [T(g, \ us)]}}\)
  using n-fun-def n-fun-concat by simp

private abbreviation (noprint) liftmap v t S' ≡ 
  map (\(\lambda (t1, t2). (\text{lift \ [(v, \ t)] \ t1, \ lift \ [(v, \ t)] \ t2)})\) S'

private lemma lift-elims:
  fixes x ::ename
  and t :: term
  and t0 :: term
  assumes ¬ occurs x t
  shows ¬ occurs x (lift [(x, t)] t0)
  proof (induction [(x, t)] t0 rule: lift.induct)
    case \(1 \ x\)
    then show ?case
      by (simp add: assms indom-def vars-def)
  next
case \(2 \ f \ ts\)
  { 
    fix v
    assume occurs v (lift [(x, t)] (T (f, ts)))
    then have list-ex (occurs v) (map (lift [(x, t)]) ts)
      by simp
    then obtain t1 where t1-def: t1 ∈ set (map (lift [(x, t)]) ts) ∧ occurs v t1
      by (meson Bex-set-list-ex)
    then obtain t1' where t1 = lift [(x,t)] t1' ∧ t1' ∈ set ts by auto
    then have \(\exists t1 \in \text{set ts}. \text{occurs \ v \ (lift \ [(x,t)] \ t1)}\)
      using t1-def by blast
  }
then show ?case
  using 2.hyps by blast
qed
private lemma lift-inv-occurs:
  fixes x :: vname
  and v :: vname
  and st :: term 
  and t :: term 
  assumes occurs v (lift [(x, st)] t)
  and ∼ occurs v st 
  and v ≠ x 
  shows occurs v t 
using assms proof (induction t rule: occurs.induct)
  case (1 v y)
  have lift [(x, st)] (V y) = V y 
   using 1.prems indom-def by auto
  then show ?case
   using 1.prems(1) by auto
next
  case (2 x t s)
  then show ?case
   by (metis (mono-tags, lifting) Bex-set-ex listE lift.simps(2) occurs.simps(2) set-map)
qed

private lemma vars-elim:
  fixes x st S
  assumes ∼ occurs x st 
  shows vars (map (lift [(x, st)]) S) ⊆ vars [st] ∪ vars S ∧
        x ∉ vars (map (lift [(x, st)]) S)
proof (induction S)
  case Nil
  then show ?case
   by (simp add: vars-def)
next
  case (Cons tx S)
    moreover have vars (map (lift [(x, st)]) (tx ≠ S)) =
       vars [lift [(x, st)] tx] ∪ vars (map (lift [(x, st)]) S)
    using vars-concat
    by (metis append.left-neutral append-Cons list.simps(9))
  moreover have vars [st] ∪ vars (tx ≠ S) = vars [st] ∪ vars S ∪ vars [tx]
    using vars-concat
    by (metis append.left-neutral append-Cons sup-commute)
moreover {
  fix v 
  assume v-mem-vars-lift: v ∈ vars [lift [(x, st)] tx]
  have v-neq-x: v ≠ x using lift-elims assms v-mem-vars-lift vars-def 
   by fastforce
  moreover have v ∈ vars [st] ∪ vars [tx]
  proof (cases)
    assume occurs v st
  qed
then show \( ?\text{thesis} \) unfolding \( \text{vars-def} \) by simp
next
assume not-occurs-v-st: \( \neg \text{occurs } v \text{ st} \)
have occurs v (lift [(x, st)] tx)
  using v-mem-vars-lift vars-def by force
then have occurs v tx using lift-inv-occurs
  using v-neq-x not-occurs-v-st by blast
then show \( ?\text{thesis} \)
  by (simp add: vars-def)
qed
ultimately have \( v \in \text{vars} [st] \cup \text{vars} [tx] \land v \neq x \) by simp
}
ultimately show \( ?\text{case} \) by blast
qed

private lemma \( n\text{-var-elim} \):
fixes \( x \text{ st } S \)
assumes \( \neg \text{occurs } x \text{ st} \)
shows \( n\text{-var} ((\text{liftmap } x \text{ st } S) < n\text{-var} ((V x, st) \# S) \)
proof –
have \( \lambda f. \text{map} \text{ fst} \text{ (map} (\lambda (t1, t2) \text{ (map} \text{ fst} (\text{map} \text{ lst}))) \text{ by} \text{ (simp add: case-prod-unfold)} \)
moreover have \( \lambda f. \text{map} \text{ snd} \text{ (map} (\lambda (t1, t2) \text{ (map} \text{ lst}))) \text{ by} \text{ (simp add: case-prod-unfold)} \)
ultimately have \( \text{lhs-split} \): \( \text{vars-eqs} ((\text{liftmap } x \text{ st } S) = \text{vars} (\text{map} (\text{lift} [(x, st)])) \text{ by} \text{ (simp add: case-prod-unfold)} \)
ultimately have \( \text{rhs-eq1} \): \( \text{vars-eqs} ((V x, st) \# S) = \{x\} \cup \text{vars} [st] \cup \text{vars-eqs} S \)
by presburger
then have \( \text{rhs-eq2} \):
  \( \text{vars-eqs} ((V x, st) \# S) = \{x\} \cup \text{vars} [st] \cup \text{vars} (\text{map} \text{ fst} S) \cup \text{vars} (\text{map} \text{ snd} S) \)
  unfolding \( \text{vars-eqs-def} \) by (simp add: sup.assoc)
from this \( \text{lhs-split vars-elim assms} \)
have \( \text{vars-eqs} (\text{liftmap } x \text{ st } S) \subseteq \text{vars} [st] \cup \text{vars-eqs} S \land \)
  \( x \notin \text{vars-eqs} (\text{liftmap } x \text{ st } S) \)
  using \( \text{vars-concat vars-eqs-def} \) by (metis map-append)
moreover have \( x \in \text{vars-eqs} ((V x, st) \# S) \)
  by (simp add: rhs-eq2)
ultimately have \( \text{vars-eqs} \ (\text{liftmap} \ x \ S) \subseteq \text{vars-eqs} \ ((V \ x, \ st) \neq S) \)
using \( \text{rhs-eq1} \) by blast
then show \( \)thesis using \( \text{vars-eqs-psubset-n-var-lt} \) by blast
qed

function \( (\text{sequential}) \) solve :: \( (\text{term} \times \text{term}) \ \text{list} \times \text{subst} \Rightarrow \text{subst} \ \text{option} \)
and elim :: \( \text{vname} \times \text{term} \times (\text{term} \times \text{term}) \ \text{list} \times \text{subst} \Rightarrow \text{subst} \ \text{option} \) where
solve([], s) = Some s
| solve((V x, t) \neq S, s) = (if V x = t then solve(S, s) else elim(x, t, S, s))
| solve((t, V x) \neq S, s) = elim(x, t, S, s)
| solve((T(f, ts), T(g, us)) \neq S, s) = (if f = g then solve((zip ts us) @ S, s) else None)

| elim(x, t, S, s) = (if occurs x t then None else let xt = lift [(x, t)]
in solve(map \( \lambda \ (t1, t2). \ (xt \ t1, xt \ t2) \) \ S,
\( (x, t) \neq (\map \ (y, u). \ (y, xt \ s)) \))
by pat-completeness auto

termination proof (relation
\( (\lambda \ X. \ \text{case} \ XX \ \text{of} \ \text{Inl}(l, -) \Rightarrow \text{n-var} \ l \ | \ \text{Inr}(x, t, S, -) \Rightarrow \text{n-var} \ ((V x, t)\#S)) \)
\( <\#\text{mlex}>\)
\( (\lambda \ X. \ \text{case} \ XX \ \text{of} \ \text{Inl}(l, -) \Rightarrow \text{n-fun} \ l \ | \ \text{Inr}(x, t, S, -) \Rightarrow \text{n-fun} \ ((V x, t)\#S)) \)
\( <\#\text{mlex}>\)
\( (\lambda \ X. \ \text{case} \ XX \ \text{of} \ \text{Inl}(l, -) \Rightarrow \text{size} \ l \ | \ \text{Inr}(x, t, S, -) \Rightarrow \text{size} \ ((V x, t)\#S)) \)
\( <\#\text{mlex}>\)

\( (\lambda \ X. \ \text{case} \ XX \ \text{of} \ \text{Inl}(l, -) \Rightarrow 1 \ | \ \text{Inr}(x, t, S, -) \Rightarrow 0) <\#\text{mlex}>\)
\( \{ \}
\)


auto simp add: uw-mlex mlex-less mlex-prod-def
have \( \bigwedge v \ S. \ \text{vars-eqs} \ S \subseteq \text{vars-eqs} \ ((V v, V v)\#S) \)
using \( \text{vars-eqs-def} \) vars-def by force
then show \( \bigwedge a b \ S. \ \neg \text{n-var} \ S < \text{n-var} \ ((V (a, b), V (a, b)) \# S) \ \Rightarrow \text{n-var} \ S = \text{n-var} \ ((V (a, b), V (a, b)) \# S) \)
using \( \text{vars-eqs-subset-n-var-le} \) by (simp add: nat-less-le)

show \( \bigwedge a b \ S. \ \neg \text{n-var} \ S < \text{n-var} \ ((V (a, b), V (a, b)) \# S) \ \Rightarrow \text{n-fun} \ S \neq \text{n-fun} \ ((V (a, b), V (a, b)) \# S) \ \Rightarrow \text{n-fun} \ S < \text{n-fun} \ ((V (a, b), V (a, b)) \# S) \)
using \( \text{n-fun-def} \) by simp

have \( \bigwedge t x v. \ \text{vars-eqs} \ [(V v, T t x)] = \text{vars-eqs} \ [(T t x, V v)] \)
using \( \text{vars-eqs-def} \)
by (simp add: sup-commute)
then have \( \bigwedge t x v. \ \text{vars-eqs} \ ((V v, T t x) \# S) = \text{vars-eqs} \ ((T t x, V v) \# S) \)
using \( \text{vars-eqs-concat} \)
by (metis append-Cons self-append-conv2)
then have \( \bigwedge a b v \ S. \ \text{n-var} \ ((V v, T (a, b)) \# S) = \text{n-var} \ ((T (a, b), V v) \# S) \)
then show \( \forall a \ b \ aa \ ba \ S \).

\( \neg \ n\text{-var} \ ((V \ (aa, \ ba), \ T \ (a, \ b)) \ # \ S) < \ n\text{-var} \ ((T \ (a, \ b), \ V \ (aa, \ ba)) \ # \ S) \)

\[ \equiv \]

\( n\text{-var} \ ((V \ (aa, \ ba), \ T \ (a, \ b)) \ # \ S) = n\text{-var} \ ((T \ (a, \ b), \ V \ (aa, \ ba)) \ # \ S) \)

by simp

show \( \forall a \ b \ aa \ ba \ S \).

\( \neg \ n\text{-var} \ ((V \ (aa, \ ba), \ T \ (a, \ b)) \ # \ S) < \ n\text{-var} \ ((T \ (a, \ b), \ V \ (aa, \ ba)) \ # \ S) \)

\[ \Rightarrow \]

\( n\text{-fun} \ ((V \ (aa, \ ba), \ T \ (a, \ b)) \ # \ S) \neq n\text{-fun} \ ((T \ (a, \ b), \ V \ (aa, \ ba)) \ # \ S) \)

\[ \Rightarrow \]

\( n\text{-fun} \ ((V \ (aa, \ ba), \ T \ (a, \ b)) \ # \ S) < n\text{-fun} \ ((T \ (a, \ b), \ V \ (aa, \ ba)) \ # \ S) \)

by simp add: n-fun-def

show \( \forall ts \ g \ us \ S. \neg \ n\text{-var} \ (\text{zip} \ ts \ us \ @ \ S) < \ n\text{-var} \ ((T \ (g, \ ts), \ T \ (g, \ us)) \ # \ S) \)

\[ \Rightarrow \]

\( n\text{-var} \ (\text{zip} \ ts \ us \ @ \ S) = n\text{-var} \ ((T \ (g, \ ts), \ T \ (g, \ us)) \ # \ S) \)

using vars-eqs-subset vars-eqs-subset-n-var-le-neg-implies-less by meson

have n-fun-nested-gt: \( \forall ts \ g \ us \ S. \ n\text{-fun} \ (\text{zip} \ ts \ us \ @ \ S) < \ n\text{-fun} \ ((T \ (g, \ ts), \ T \ (g, \ us)) \ # \ S) \)

using n-fun-nested-head

by (metis add-leD1 le-neg-implies-less add2-eq-Suc' leD less-Suc-eq)

show \( \forall ts \ g \ us \ S. \)

\( \neg \ n\text{-var} \ (\text{zip} \ ts \ us \ @ \ S) < \ n\text{-var} \ ((T \ (g, \ ts), \ T \ (g, \ us)) \ # \ S) \)

\( \Rightarrow \)

\( \neg \ n\text{-fun} \ (\text{zip} \ ts \ us \ @ \ S) < \ n\text{-fun} \ ((T \ (g, \ ts), \ T \ (g, \ us)) \ # \ S) \)

\[ \Rightarrow \]

\( n\text{-fun} \ (\text{zip} \ ts \ us \ @ \ S) = n\text{-fun} \ ((T \ (g, \ ts), \ T \ (g, \ us)) \ # \ S) \)

using n-fun-nested-gt by meson

show \( \forall ts \ g \ us \ S. \)

\( \neg \ n\text{-var} \ (\text{zip} \ ts \ us \ @ \ S) < \ n\text{-var} \ ((T \ (g, \ ts), \ T \ (g, \ us)) \ # \ S) \)

\( \Rightarrow \)

\( \neg \ n\text{-fun} \ (\text{zip} \ ts \ us \ @ \ S) < \ n\text{-fun} \ ((T \ (g, \ ts), \ T \ (g, \ us)) \ # \ S) \)

\[ \Rightarrow \]

\( \min \ (\text{length} \ ts) \ (\text{length} \ us) = 0 \)

using n-fun-nested-gt by blast

show \( \forall x \ t \ S. \)

\( \neg \ \text{occurs} \ x \ t \)

\[ \Rightarrow \]

\( \neg \ n\text{-var} \ (\text{liftmap} \ x \ t \ S) < \ n\text{-var} \ ((V \ x, \ t) \ # \ S) \)

\[ \Rightarrow \]

\( n\text{-var} \ (\text{liftmap} \ x \ t \ S) = n\text{-var} \ ((V \ x, \ t) \ # \ S) \)

using n-var-elim leD linorder-negE-nat by blast

show \( \forall x \ t \ S. \)

\( \neg \ \text{occurs} \ x \ t \)

\[ \Rightarrow \]

\( \neg \ n\text{-var} \ (\text{liftmap} \ x \ t \ S) < \ n\text{-var} \ ((V \ x, \ t) \ # \ S) \)

\[ \Rightarrow \]

\( n\text{-fun} \ (\text{liftmap} \ x \ t \ S) \neq n\text{-fun} \ ((V \ x, \ t) \ # \ S) \)

\[ \Rightarrow \]

\( n\text{-fun} \ (\text{liftmap} \ x \ t \ S) < n\text{-fun} \ ((V \ x, \ t) \ # \ S) \)

using n-var-elim by simp
A.3 Theory about datastructures for imperative version

theory ITerm
  imports Main
  HOL-Imperative-HOL.Ref
  HOL-Imperative-HOL.Heap-Monad
begin

datatype i-term-d =
  IVarD
| ITermD (string × i-terms ref option)
and i-terms = ITerms (i-term ref × i-terms ref option)
and i-term = ITerm (nat × i-term ref option × i-term-d)

instantiation i-term :: heap begin
  instance by countable-datatype
end

instantiation i-terms :: heap begin
  instance by countable-datatype
end

lemma typerep-term-neq-terms: TYPEREPS(i-term) ≠ TYPEREPS(i-terms)
using typerep-i-terms-def typerep-i-term-def by fastforce

lemma typerep-term-neq-nat: TYPEREPS(i-term) ≠ TYPEREPS(nat)
using typerep-i-term-def typerep-nat-def by fastforce

lemma typerep-terms-neq-nat: TYPEREPS(i-terms) ≠ TYPEREPS(nat)
using typerep-i-terms-def typerep-nat-def by fastforce

definition is-IVar where is-IVar t =
  (case t of ITerm(_, _, IVarD) ⇒ True | _ ⇒ False)

definition get-ITerm-args where get-ITerm-args t =
  (case t of ITerm(_, _, ITermD (_, tn)) ⇒ tn | _ ⇒ None)

fun get-is where
  get-is-def: get-is t (ITerm(_, is, _)) = is

fun get-stamp where
  get-stamp-def: get-stamp (ITerm(s, _, _)) = s

lemma get-stamp-iff-ex:
  fixes t s shows (get-stamp t = s) ↔ (∃ is d. t = ITerm(s, is, d))
by (cases t, cases, blast, force)

27
lemma \textbf{get-ITerm-args-iff-ex}:
\begin{align*}
\text{shows} \quad & (\exists s \text{ is } d. \ t = \text{ITerm}(s, \text{is}, \text{d}) \land (tsp = \text{None} \land d = \text{IVarD}) \lor \\
& (\exists f. \ d = \text{ITermD}(f, tsp)))) \\
\text{proof} & - \\
\end{align*}
\begin{itemize}
\item obtain \(s \text{ is } d \) where \(t = \text{ITerm}(s, \text{is}, \text{d})\) \\
\hspace{1em} by (metis i-term.exhaust surj-pair) \\
\item then show \(\exists \text{thesis unfolding get-ITerm-args-def}\) \\
\hspace{1em} by (cases d rule: i-term-d.exhaust; force) \\
\end{itemize}
qed

\textbf{type-synonym} \(\text{i-termP} = \text{i-term ref option}\)

\textbf{type-synonym} \(\text{i-termsP} = \text{i-terms ref option}\)

\textbf{inductive} \(\text{i-term-acyclic}:: \text{heap} \Rightarrow \text{i-termP} \Rightarrow \text{bool}\) and \(\text{i-terms-acyclic}:: \text{heap} \Rightarrow \text{i-termsP} \Rightarrow \text{bool}\) where
\begin{enumerate}
\item \(\text{t-acyclic-nil}:: \text{i-term-acyclic - None}\)
\item \(\text{t-acyclic-step-link}:: \text{i-term-acyclic h t} \Rightarrow \text{Ref.} \cdot \text{get h tref} = \text{ITerm}(-, t, \text{IVarD}) \Rightarrow \text{i-term-acyclic h (Some tref)}\)
\item \(\text{t-acyclic-step-ITerm}:: \text{i-terms-acyclic h tsref} \Rightarrow \text{Ref.} \cdot \text{get h tref} = \text{ITerm}(-, \text{None}, \text{ITermD}(-, \text{tsref})) \Rightarrow \text{i-term-acyclic h (Some tref)}\)
\item \(\text{ts-acyclic-nil}:: \text{i-terms-acyclic - None}\)
\item \(\text{ts-acyclic-step-ITerms}:: \text{i-terms-acyclic h ts2ref} \Rightarrow \text{i-term-acyclic h (Some tref)} \Rightarrow \text{Ref.} \cdot \text{get h tsref} = \text{ITerms} (\text{tref}, \text{ts2ref}) \Rightarrow \text{i-terms-acyclic h (Some tsref)}\)
\end{enumerate}

\textbf{lemma} \textbf{acyclic-terms-term-simp [simp]}:
\begin{itemize}
\item \textbf{fixes} \(\text{tr}:: \text{i-term ref}\) \\
\hspace{1em} and \(\text{termsp}\) \\
\hspace{1em} and \(\text{terms}\) \\
\hspace{1em} and \(\text{s}:: \text{nat}\) \\
\hspace{1em} and \(\text{h}:: \text{heap}\) \\
\item \textbf{assumes} \(\text{acyclic}:: \text{i-term-acyclic h (Some tr)}\) \\
\hspace{1em} and \(\text{get-tr}:: \text{Ref.} \cdot \text{get h tr} = \text{ITerm} (s, \text{None}, \text{ITermD}(f, \text{termsp}))\) \\
\item \textbf{shows} \(\text{i-terms-acyclic h termsp}\)
\end{itemize}

\textbf{proof} — \\
\begin{itemize}
\item \textbf{consider} \\
\hspace{2em} (a) \(h'\) where \(h = h' \land (\text{Some tr}) = \text{None}\) \\
\hspace{2em} (b) \(h' \land \text{tref' s'}\) where \\
\hspace{3em} \(h' = h \land (\text{Some tr}) = \text{Some tref} \land \text{i-term-acyclic h t}\) \\
\end{itemize}
Ref\_get\_h\_tref = ITerm\ (s', t, IVarD) \ |
(c) h' tsref tref s' f' where
h' = h \land (Some\ tr) = Some\ tref \land
\text{i-terms-acyclic\ h\ tsref} \land
Ref\_get\_h\_tref = ITerm\ (s', None, ITermD\ (f', tsref))
\text{using\ i-term-acyclic\ \text{simps\ acyclic\ by\ fast}
then\ show\ \text{?thesis\ using\ get-tr}
by\ (cases,\ fastforce+)}
qed

\text{lemma\ acyclic-terms-terms-simp\ [simp]:}
\text{fixes\ tsr::\ i-terms\ ref}
\text{and\ tthis::\ i-term\ ref}
\text{and\ tnext::\ i-termsP}
\text{and\ h::\ heap}
\text{assumes\ acyclic: i-term-acyclic\ h\ (Some\ tsr)}
\text{and\ get-termsr: Ref\_get\_h\_tsr = ITerms\ (tthis, tnext)}
\text{shows\ i-terms-acyclic\ h\ tnext}
proof –
consider\ (a)\ (Some\ tsr) = None\ |
(b)\ tref\ where
\text{i-term-acyclic\ h\ (Some\ tref)} \land
Ref\_get\_h\_tref = ITerms\ (tref, None) \mid
(c)\ ts2ref\ tref\ where
\text{i-terms-acyclic\ h\ ts2ref} \land
\text{i-term-acyclic\ h\ (Some\ tref)} \land
Ref\_get\_h\_tref = ITerms\ (tref, ts2ref)
\text{using\ acyclic\ i-term-acyclic\ \text{simps[of\ h\ Some\ tsr]}\ by\ fast}
then\ show\ \text{?thesis\ using\ get-termsr\ ts-acyclic-nil}
by\ (cases,\ fastforce+)
qed

\text{lemma\ acyclic-term-link-simp:}
\text{fixes\ tr::\ i-term\ ref}
\text{and\ tr'::\ i-term\ ref}
\text{and\ d::\ i-term-d}
\text{and\ s::\ nat}
\text{and\ h::\ heap}
\text{assumes\ acyclic: i-term-acyclic\ h\ (Some\ tr)}
\text{and\ get-tr: Ref\_get\_h\_tr = ITerm\ (s, Some\ tr', d)}
\text{shows\ i-term-acyclic\ h\ (Some\ tr')}
proof –
consider\ (a)\ (Some\ tr) = None\ |
(b)\ t\ s'\ where
\text{i-term-acyclic\ h\ t} \land
Ref\_get\_h\_tr = ITerm\ (s', t, IVarD) \mid
(c)\ tsref\ s'\ f\ where
\text{i-terms-acyclic\ h\ tsref} \land
Ref\_get\_h\_tr = ITerm\ (s', None, ITermD\ (f, tsref))
using acyclic i-term-acyclic.simps[of h Some tr] by blast
then show ?thesis using get-tr
  by cases (fastforce+)
qed

lemma acyclic-args-nil-is:
  assumes i-term-acyclic h (Some tr)
  and Ref.get h tr = ITerm(s, is, ITermD(f, tsp))
  shows is = None
using assms by (cases h Some tr rule: i-term-acyclic.cases; fastforce)

lemma acyclic-heap-change-nt:
  fixes tr :: i-term ref
  and r :: 'a::heap ref
  and v :: 'a::heap
  and h :: heap
  assumes acyclic: i-term-acyclic h (Some tr)
  and ne-iterm: TYPEREQ('a) ≠ TYPEREQ(i-term)
  and ne-iterms: TYPEREQ('a) ≠ TYPEREQ(i-terms)
  shows i-term-acyclic (Ref.set r v h) (Some tr)
using acyclic
proof (induction h Some tr
  arbitrary: tr
  taking: λ h tsp. i-terms-acyclic (Ref.set r v h) tsp
  rule: ITerm.i-term-acyclic-i-terms-acyclic.inducts(1))
  case (t-acyclic-step-link h is tr s)
  show ?case proof (cases is)
  case None
  then have Ref.get (Ref.set r v h) tr = ITerm (s, None, IVarD)
    using ne-iterm Ref.get-set-neq Ref.noteq-def t-acyclic-step-link.hyps(3) by metis
  then show ?thesis
    using i-term-acyclic-i-terms-acyclic.t-acyclic-step-link t-acyclic-nil by blast
  next
  case (Some isr)
  then have Ref.get (Ref.set r v h) tr = ITerm (s, Some isr, IVarD)
    using ne-iterm Ref.get-set-neq Ref.noteq-def t-acyclic-step-link.hyps(3) by metis
  then show ?thesis
    using Some i-term-acyclic-i-terms-acyclic.t-acyclic-step-link
    t-acyclic-step-link.hyps(2) by blast
  qed
next
  case (t-acyclic-step-ITerm h tsref tr s f)
  then have Ref.get (Ref.set r v h) tr = ITerm (s, None, ITermD (f, tsref))
    using ne-iterm Ref.get-set-neq Ref.noteq-def by metis
  then show ?case
    using i-term-acyclic-i-terms-acyclic.t-acyclic-step-ITerm
    t-acyclic-step-ITerm.hyps(2) by blast

30
next
case (ts-acyclic-nil h)
then show ?case
  using i-term-acyclic-i-terms-acyclic.ts-acyclic-nil by blast
next
case (ts-acyclic-step-ITerms h ts2ref tref tsref)
then have Ref.get h r = ITerm(s, is, IVarD)
  using iterm-get ne-iterms Ref.get-neq Ref.neq-def by metis
then show ?case
  using i-term-acyclic-i-terms-acyclic.ts-acyclic-step-ITerms ts-acyclic-step-ITerms.hyps(2)
  ts-acyclic-step-ITerms.hyps(4) by blast
qed

lemma acyclic-heap-change-is-uc:
fixes tr :: i-term ref
  and r :: i-term ref
  and v :: i-term
  and h :: heap
assumes acyclic: i-term-acyclic h (Some tr)
  and get-r: Ref.get h r = ITerm(s, is, IVarD)
  and v-val: v = ITerm(s', is, IVarD)
sows i-term-acyclic (Ref.set r v h) (Some tr)
using acyclic get-r
proof (induction h Some tr arbitrary: tr
  taking: λ tsp. Ref.get h r = ITerm(s, is, IVarD) → i-term-acyclic (Ref.set r v h) tsp
  rule: ITerm.i-term-acyclic-i-terms-acyclic.inducts(1))
case (t-acyclic-step-link h tr-is tr s1)
then have case-get-r: Ref.get h r = ITerm(s, is, IVarD)
  and get-tr: Ref.get h tr = ITerm(s1, tr-is, IVarD)
  and IH: ∀ tr. tr-is = Some tr ⇒
  i-term-acyclic (Ref.set r v h) (Some tr)
  by blast+
have ∃ s0. Ref.get (Ref.set r v h) tr = ITerm(s0, tr-is, IVarD)
proof (rule case-split)
  assume r = tr
  then show ?thesis using get-tr case-get-r v-val by simp
next
  assume r ≠ tr
  then show ?thesis using get-tr Ref.get-set-neq Ref.uneq-def by metis
qed
moreover have i-term-acyclic (Ref.set r v h) tr-is
  using t-acyclic-nil IH case-get-r
  by (metis option.exhaust-sel)
ultimately show ?case
  using i-term-acyclic-i-terms-acyclic.t-acyclic-step-link by blast
next
case \((t\text{-acyclic-step}-\text{ITerm} \ h \ tsref \ tr \ s1 \ f)\)
then have \(\text{case-get-r: Ref.get \ h \ r = \text{ITerm} (s, \ is, IVarD)}\)
and \(\text{get-tr: Ref.get \ h \ tr = \text{ITerm} (s1, \ None, \text{ITermD} (f, tsref))}\)
and \(\text{get-tsref: i-terms-acyclic (Ref.set \ r \ v \ h) \ tsref}\)
by blast+
have \(tr \neq r\)
using \(\text{get-tr case-get-r by force}\)
then have \(\exists s0. \text{Ref.get (Ref.set \ r \ v \ h) \ tr} = \text{ITerm} (s0, \ None, \text{ITermD} (f, tsref))\)
using \(\text{get-tr by simp}\)
then show \(\exists \) case
using \(\text{i-term-acyclic-i-terms-acyclic.2}\)
by blast
next
case \((ts\text{-acyclic-nil} \ h)\)
then show ?case
by \((\text{simp add: i-term-acyclic-i-terms-acyclic.ts-acyclic-nil})\)
next
case \((ts\text{-acyclic-step-ITerms} \ h \ ts2ref \ tref \ tsref)\)
then have \(\text{get-tsref: Ref.get \ h \ tsref} = \text{ITerms} (tref, ts2ref)\)
and \(IH1: \text{Ref.get \ h \ r} = \text{ITerm} (s, \ is, IVarD) \Rightarrow i\text{-terms-acyclic (Ref.set \ r \ v \ h)} \ ts2ref\)
and \(IH2: \text{Ref.get \ h \ r} = \text{ITerm} (s, \ is, IVarD) \Rightarrow i\text{-term-acyclic (Ref.set \ r \ v \ h)} (\text{Some} tref)\)
by blast+
have \(\text{Ref.get (Ref.set \ r \ v \ h) \ tsref} = \text{ITerms} (tref, ts2ref)\)
using \(\text{get-tsref typerep-term-neq-terms Ref.get-set-neq Ref.noteq-def by metis}\)
then show ?case
using \(\text{i-term-acyclic-i-terms-acyclic.ts-acyclic-step-ITerms}\)
IH1 IH2 by blast
qed

lemma \(i\text{-terms-acyclic-induct [consumes 1.}\)

case-names ts-acyclic-nil ts-acyclic-step:\)
fixes \(h::\text{heap}\)
and \(tsp::i\text{-terms ref option}\)
and \(P::\text{heap} \Rightarrow i\text{-terms ref option} \Rightarrow \text{bool}\)
assumes \(\text{acyclic: i-term-acyclic h tsp}\)
and \(\lambda h. \ P \ h \ None\)
and \(\lambda h \ ts2ref \ tref \ tsref.\)
\(\text{i-terms-acyclic h ts2ref} \Rightarrow\)
\(P \ h \ ts2ref \Rightarrow \text{i-term-acyclic h (Some tref)} \Rightarrow\)
\(\text{Ref.get \ h \ tsref} = \text{ITerms} (tref, ts2ref) \Rightarrow\)
\(P \ h \ (\text{Some tsref})\)
shows \(P \ h \ tsp\)
using \(\text{assms ts-acyclic-nil}\)
by \((\text{induction taking: } \lambda h \ tp. \ True \ rule: \text{i-term-acyclic-i-terms-acyclic.inducts}(2),\)
blast+)
inductive-set \textit{i-terms-sublists} for \( h :: \text{heap} \) and \( tsp :: \text{i-termsP} \) where
\begin{align*}
\text{next:} & \\
(Some \ tsp') & \in \text{i-terms-sublists} \ h \ tsp \implies \\
\text{Ref. get } h \ tsp' & = \text{ITerms}(\cdot, \tnext) \implies \\
\tnext & \in \text{i-terms-sublists} \ h \ tsp \\
\text{self:} & \ tsp \in \text{i-terms-sublists} \ h \ tsp
\end{align*}

\textbf{lemma} \textit{i-terms-sublists-mNone:}
\begin{itemize}
\item \textbf{fixes} \( h :: \text{heap} \)
\item \textbf{and} \( tsp :: \text{i-termsP} \)
\item \textbf{assumes} \( \text{i-terms-acyclic} \ h \ tsp \)
\item \textbf{shows} \( None \in \text{i-terms-sublists} \ h \ tsp \)
\item \textbf{using} \assms
\end{itemize}
\textbf{proof} (\textit{induction rule: i-terms-acyclic-induct})
\begin{itemize}
\item \textbf{case} \( \text{(ts-acyclic-nil uy)} \)
\item then show \( ?\text{case} \)
\begin{itemize}
\item by \( (\text{simp add: i-terms-sublists}. \text{intros}(2)) \)
\end{itemize}
\textbf{next}
\item \textbf{case} \( \text{(ts-acyclic-step} \ h \ \tnext \ \tref \ \tsref) \)
\item have \( \text{i-terms-acyclic} \ h \ \tnext \implies \\
\text{Ref. get } h \ \tsref = \text{ITerms} \ (\tref, \ \tnext) \implies \\
None \in \text{i-terms-sublists} \ h \ (\text{Some} \ \tsref) \\
\text{using} \text{ts-acyclic-step}.\text{IH} \ \text{i-terms-sublists}.\text{intros} \\
\text{by} \ (\text{induction rule: i-terms-sublists}.\text{induct}, \text{blast+})
\item then show \( ?\text{case} \text{ using ts-acyclic-step by blast} \)
\end{itemize}
\textbf{qed}

\textbf{lemma} \textit{i-terms-sublists-None-om:}
\begin{itemize}
\item \textbf{fixes} \( h :: \text{heap} \)
\item \textbf{shows} \( \text{i-terms-sublists} \ h \ None = \{None\} \)
\item \textbf{proof} – \begin{itemize}
\item \textbf{fix} \( tsp \ ts2p \)
\item \textbf{have} \( ts2p \in \text{i-terms-sublists} \ h \ tsp \implies \exists \tr. \ (\text{Some} \ \tr) = ts2p \implies tsp \neq None \\
\text{by} \ (\text{induction rule: i-terms-sublists}.\text{induct}, \text{blast+}) \)
\end{itemize}
\item then show \( ?\text{thesis} \)
\begin{itemize}
\item using \( \text{i-terms-sublists}.\text{intros}(2) \text{ these-empty-eq by fastforce} \)
\end{itemize}
\textbf{qed}

\textbf{lemma} \textit{i-terms-sublists-subset:}
\begin{itemize}
\item \textbf{fixes} \( h :: \text{heap} \)
\item \textbf{and} \( tsp \ \text{and} \ \tr \)
\item \textbf{assumes} \( \text{Ref. get } h \ \tsr = \text{ITerms} \ (\tr, \ tsp) \)
\item \textbf{shows} \( \text{i-terms-sublists} \ h \ tsp \subseteq \text{i-terms-sublists} \ h \ (\text{Some} \ \tsr) \)
\item \textbf{proof} – \begin{itemize}
\item \textbf{fix} \( ts2p \)
\item \textbf{have} \( ts2p \in \text{i-terms-sublists} \ h \ tsp \implies ts2p \in \text{i-terms-sublists} \ h \ (\text{Some} \ \tsr) \)
\end{itemize}
\end{itemize}
proof (induction rule: i-terms-sublists.inducts)
case (next tsr' uu tnext)
  then show ?case using assms
    using i-terms-sublists.intros(1) by blast
next
case self
  then show ?case using assms
    using i-terms-sublists.intros(2) by blast
qed
}
then show ?thesis by fast
qed

lemma i-terms-sublists-insert:
  fixes h :: heap
and tsr and tr
assumes Ref. get h tsr = ITerms (tr, tsp)
shows i-terms-sublists h (Some tsr) = insert (Some tsr) (i-terms-sublists h tsp)
proof –
  \{ 
  fix ts2p
  have ts2p ∈ i-terms-sublists h (Some tsr) ⇒ ts2p = Some tsr ∨ ts2p ∈ i-terms-sublists h tsp
  proof (induction rule: i-terms-sublists.inducts)
  case (next tsr' this tnext)
  then consider (a) Some tsr' = Some tsr | (b) Some tsr' ∈ i-terms-sublists h tsp
  by fast
  then show ?case
  proof (cases)
  case a
  then show ?thesis using next assms i-terms-sublists.self by force
  next
  case b
  then show ?thesis using next assms i-terms-sublists.next by blast
  qed
  next
  case self
  then show ?case using assms by blast
  qed
  \}
moreover have Some tsr ∈ i-terms-sublists h (Some tsr)
  by (simp add: i-terms-sublists.intros(2))
ultimately show ?thesis
  using assms i-terms-sublists.intros i-terms-sublists-subset by blast
qed

lemma i-terms-sublists-finite:
  fixes h :: heap

and tsp:: i-termsP
assumes i-terms-acyclic h tsp
shows finite (i-terms-sublists h tsp)
using assms proof (induction rule:i-terms-acyclic-induct)
case (ts-acyclic-nil h)
then show ?case using i-terms-sublists-None-om by fastforce
next
case (ts-acyclic-step h ts2ref tref tsref)
then show ?case using i-terms-sublists-insert by fastforce
qed

lemma i-terms-sublists-acyclic:
fixes ts2p:: i-termsP
    and tsp:: i-termsP
    and h:: heap
assumes acyclic: i-terms-acyclic h tsp
    and ts2p-mem: ts2p ∈ i-terms-sublists h tsp
shows i-terms-acyclic h ts2p
using ts2p-mem acyclic acyclic-terms-terms-simp
by (induction rule: i-terms-sublists.inducts, blast)

inductive-set i-terms-set for h:: heap and tsp:: i-termsP where
(Some tsr') ∈ i-terms-sublists h tsp ⟹
Ref.get h tsr' = ITerms(tp, -) ⟹
.tp ∈ i-terms-set h tsp

lemma i-terms-set-def2:
fixes h:: heap and tsp:: i-termsP
shows
i-terms-set h tsp = \{tp.
    \exists tsr' tnext. (Some tsr') ∈ i-terms-sublists h tsp ∧ Ref.get h tsr' = ITerms(tp, tnext)\}
using i-terms-set-def i-terms-setp.simps i-terms-sublistsp-i-terms-sublists-eq by presburger

lemma i-terms-set-None-empty:
fixes h:: heap
shows i-terms-set h None = {}
using i-terms-sublists-None-om i-terms-set-def2
by auto

lemma i-terms-set-empty-iff:
fixes tsp:: i-termsP
    and h:: heap
shows (i-terms-set h tsp = {}) = (tsp = None)
proof
{
assume tsp ≠ None
then obtain tsr tthisr tsnextp

35
where Some tsr = tsp
and Ref.get h tsr = ITerms(thisr, tsnextp)
by (metis i-terms.exhaust old.prod.exhaust option.exhaust)
then have thisr ∈ i-terms-set h tsp
using i-terms-set.simps i-terms-sublists.self by blast
then have i-terms-set h tsp ≠ {} by blast
}
then show ?thesis
using i-terms-set-None-empty by blast
qed

lemma i-terms-set-insert:
fixes h:: heap
and tsr and tr
assumes Ref.get h tsr = ITerms(tr, tsp)
shows i-terms-set h (Some tsr) = insert tr (i-terms-set h tsp)
using assms i-terms-sublists-insert i-terms-set-def2 by auto

lemma i-terms-set-single:
fixes h:: heap
and tsr and tr
assumes Ref.get h tsr = ITerms(tr, None)
shows i-terms-set h (Some tsr) = {tr}
using assms i-terms-set-insert i-terms-set-None-empty by simp

lemma i-terms-set-finite:
fixes h:: heap
and tsp:: i-termsP
assumes i-terms-acyclic h tsp
shows finite (i-terms-set h tsp)
using assms proof (induction rule:i-terms-acyclic-induct)
case (ts-acyclic-nil h)
then show ?case
using i-terms-set-None-empty by simp
next
case (ts-acyclic-step h ts2ref tref tsref)
show ?case
qed

lemma i-term-acyclic-induct [consumes 1, case-names nil var link args]:
fixes h:: heap
and tp:: i-term ref option
and P:: heap ⇒ i-term ref option ⇒ bool
assumes acyclic: i-term-acyclic h tp
and nil-case: ∃ h. P h None
and var-case: ∃ h tr s.
Ref.get h tr = ITerm(s, None, IVarD) ⇒
P h (Some tr)
and link-case: \( \wedge h \ tr \ isr \ s. \)
\[
P h (\text{Some} \ isr) \quad \Rightarrow \quad \text{Ref.get} \ h \ tr = \text{ITerm}(s, \text{Some} \ isr, \text{IVarD}) \quad \Rightarrow \quad P h (\text{Some} \ tr)
\]
and args-case: \( \wedge h \ tr \ tsp \ s \ f. \)
\[
(\forall \ tr2 \in \text{i-terms-set} \ h \ tsp. \ P h (\text{Some} \ tr2)) \quad \Rightarrow \quad \text{i-terms-acyclic} \ h \ tsp \quad \Rightarrow \quad \text{Ref.get} \ h \ tr = \text{ITerm}(s, \text{None}, \text{ITermD}(f, \ tsp)) \quad \Rightarrow \quad P h (\text{Some} \ tr)
\]
shows \( P h \ tp \)
using acyclic
proof (induction \( h \ tp \)
  taking: \( \lambda h \ tp. \ \forall \ tr2 \in \text{i-terms-set} \ h \ tsp. \ P h (\text{Some} \ tr2) \)
  rule: \( \text{i-term-acyclic-i-terms-acyclic.inducts}(1) \)
  case (\( t\text{-acyclic-nil} \ h \))
  then show \( \)\( ?\)case by (\( \text{fact nil-case} \))
next
case (\( t\text{-acyclic-step-link} \ h \ t \ tref \ uv \))
  then show \( ?\)case using var-case link-case
    by (metis not-None-eq)
next
case (\( t\text{-acyclic-step-ITerm} \ h \ tsref \ tref \ uv \))
  then show \( ?\)case using args-case by blast
next
case (\( ts\text{-acyclic-nil} \ h \))
  then show \( ?\)case using \( \text{i-terms-set-None-empty} \) by blast
next
case (\( ts\text{-acyclic-step-ITerms} \ h \ ts2ref \ tref \ uv \))
  then show \( ?\)case using \( \text{i-terms-set-insert} \) by fast
qed

lemma \( \text{i-term-acyclic-induct}' \) [consumes 1, case-names var link args]:
fixes \ h::\ heap
  and \ tr::\ \text{i-term ref}
and \ P::\ \text{heap} \Rightarrow \ \text{i-term ref} \Rightarrow \ \text{bool}
assumes acyclic: \( \text{i-term-acyclic} \ h (\text{Some} \ tr) \)
and var-case: \( \wedge h \ tr \ s. \)
  \( \text{Ref.get} \ h \ tr = \text{ITerm}(s, \text{None}, \text{IVarD}) \quad \Rightarrow \quad P h \ tr \)
and link-case: \( \wedge h \ tr \ isr \ s. \)
  \( P h \ isr \quad \Rightarrow \quad \text{Ref.get} \ h \ tr = \text{ITerm}(s, \text{Some} \ isr, \text{IVarD}) \quad \Rightarrow \quad P h \ tr \)
and args-case: \( \wedge h \ tr \ tsp \ s \ f. \)
\[
(\forall \ tr2 \in \text{i-terms-set} \ h \ tsp. \ P h \ tr2) \quad \Rightarrow \quad \text{i-terms-acyclic} \ h \ tsp \quad \Rightarrow \quad \text{Ref.get} \ h \ tr = \text{ITerm}(s, \text{None}, \text{ITermD}(f, \ tsp)) \quad \Rightarrow \quad P h \ tr
\]
shows \( P h \ tr \)

37
proof -
{
    fix tp
    have i-term-acyclic h tp \implies tp = Some tr \implies P h tr
    proof (induction h tp arbitrary; tr rule: i-term-acyclic-induct)
    case (nil h)
    then show ?case by fast
    next
    case (var h tr s)
    then show ?case using var-case by blast
    next
    case (link h tr s isr d)
    then show ?case using link-case by fast
    next
    case (args h tr tsp s f)
    then show ?case using args-case by fast
    qed
}
then show ?thesis by (simp add: acyclic)
qed

lemma i-terms-set-acyclic:
fixes tr:: i-term ref
    and tsp:: i-termsP
    and s:: nat
    and h:: heap
assumes acyclic: i-terms-acyclic h tsp
    and tr-mem: tr \in i-terms-set h tsp
shows i-term-acyclic h (Some tr)
using tr-mem proof (cases rule: i-terms-set.cases)
case (1 tsr' tsnext)
then have *: Some tsr' \in i-terms-sublists h tsp
    and **: Ref.get h tsr' = ITerms (tr, tsnext)
    by blast+
from * have i-terms-acyclic h (Some tsr')
    using acyclic i-terms-sublists-acyclic by blast
then consider
(a) Some tsr' = None |
(b) tref tsref where
    Some tsr' = Some tsref and
    i-term-acyclic h (Some tref) and
    Ref.get h tsref = ITerms (tref, None) |
(c) ts2ref tref tsref where
    Some tsr' = Some tsref and
    i-terms-acyclic h ts2ref and
    i-term-acyclic h (Some tref) and
    Ref.get h tsref = ITerms (tref, ts2ref)
using i-terms-acyclic.simps[of h Some tsr'] by blast

38
then show \( \text{thesis} \)

proof (cases)
  case a
  then show \( \text{thesis by simp} \)

next
  case b
  then show \( \text{thesis using ** by simp} \)

next
  case c
  then have Some tsref' \( = \) Some tsref and i-term-acyclic h (Some tref) and Ref.get h tsref = ITerms (tref, ts2ref)
    by blast
  then show \( \text{thesis using ** by simp} \)

qed

inductive-set i-term-closure for h :: heap and tp :: i-termP where
  Some tr = tp \( \Rightarrow \) tr \( \in \) i-term-closure h tp |
  tr \( \in \) i-term-closure h tp \( \Rightarrow \)
    Ref.get h tr = ITerm(-, Some is, -) \( \Rightarrow \)
    is \( \in \) i-term-closure h tp |
  tr \( \in \) i-term-closure h tp \( \Rightarrow \)
    Ref.get h tr = ITerm(-, None, ITermD(-, tsp)) \( \Rightarrow \)
    tr2 \( \in \) i-terms-set h tsp \( \Rightarrow \)
    tr2 \( \in \) i-term-closure h tp

abbreviation i-term-closures where
i-term-closures h trs \( \equiv \) UNION (Some ' trs) (i-term-closure h)

abbreviation i-terms-closure where
i-terms-closure h tsp \( \equiv \) i-term-closures h (i-terms-set h tsp)

abbreviation i-term-sublists where
i-term-sublists h tr \( \equiv \) i-terms-sublists h (get-ITerm-args (Ref.get h tr))

abbreviation i-term-closure-sublists where
i-term-closure-sublists h tp \( \equiv \) (\( \bigcup \) tr' \( \in \) i-term-closure h tp. i-term-sublists h tr')

abbreviation i-terms-closure-sublists where
i-terms-closure-sublists h tsp \( \equiv \) i-terms-sublists h tsp \( \cup \) (\( \bigcup \) tr\( \in \)i-term-closure h tsp. i-term-sublists h tr)

lemma i-term-closure-None:
  fixes h :: heap
  shows i-term-closure h None = {}
proof
  {tp tr

39
have \( tr \in i\text{-}term\text{-}closure \ h \ tp \implies tp = None \implies False \)
by (cases rule: i-term-closure.induct, blast+)

} then show \(?thesis\) by auto
qed

lemma i-term-closure-var:
fixes \( tr::i-term\ \ref \)
and \( s::\nat \)
and \( h::\heap \)
assumes Ref.get \( h \ tr = ITerm \ (s, None, IVarD) \)
shows \( i\text{-}term\text{-}closure \ h \ (Some \ tr) = \{ tr \} \)
proof –

\{  
fix \( tp \ tr \ x \)
have \( x \in i\text{-}term\text{-}closure \ h \ tp \implies  
    tp = Some \ tr \implies Ref.get \ h \ tr = ITerm \ (s, None, IVarD) \implies x = tr \)
by (induction rule: i-term-closure.induct, fastforce+)
\}

then show \(?thesis\) using \( \text{assms} \)
using i-term-closure.intros(1) by blast
qed

lemma i-term-closure-link:
fixes \( tr::i-term\ \ref \)
and \( isr::i-term\ \ref \)
and \( d::i-term\text{-}d \)
and \( s::\nat \)
and \( h::\heap \)
assumes Ref.get \( h \ tr = ITerm \ (s, Some \ isr, d) \)
shows \( i\text{-}term\text{-}closure \ h \ (Some \ tr) = insert \ tr \ (i\text{-}term\text{-}closure \ h \ (Some \ isr)) \)
proof –

\{  
fix \( tp \ x \)
have \( x \in i\text{-}term\text{-}closure \ h \ tp \implies  
    tp = Some \ tr \implies  
    Ref.get \ h \ tr = ITerm \ (s, Some \ isr, d) \implies x = tr \lor x \in i\text{-}term\text{-}closure \ h \ (Some \ isr) \)
proof (induction rule: i-term-closure.induct)
  case (1 \( tr \))
  then show \(?case\) by blast
next
  case (2 \( tr' s' \) is \( uv \))
  then show \(?case\)
proof (cases \( tr' = tr \))
  case True
  then show \(?thesis\) using 2 by (simp add: i-term-closure.intros(1))
next
  case False

40
then show \( \textit{thesis} \) using \( 2 \) \textit{i-term-closure.intro}(2) by \textbf{blast}

\textbf{qed}

next

case \( 3 \) \( \textit{tr}' \ s' f \textit{tsp} \textit{tr2} \)
then have \( \textit{tr} \neq \textit{tr}' \) by \textbf{fastforce}
then show \( \textit{?case} \) using \( 3 \) \textit{i-term-closure.intro}(3) by \textbf{blast}

\textbf{qed}

\}

moreover {

d \textbf{fix} \( \textit{x} \)
assume \( \textit{x} \in \textbf{insert} \textit{tr} \) \( \textbf{(i-term-closure} \textit{h (Some} \textit{isr)}) \)
then consider \( (a) \textit{x} = \textit{tr} \mid (b) \textit{x} \in \textbf{i-term-closure} \textit{h (Some} \textit{isr)}) \)
by \textbf{blast}
then have \( \textit{x} \in \textbf{i-term-closure} \textit{h (Some} \textit{tr}) \)

\textbf{proof} (cases)

case \textit{a}
then show \( \textit{?thesis} \) using \( \textbf{i-term-closure.intro}(1) \) by \textbf{blast}

\textbf{next}

case \textit{b}
then show \( \textit{?thesis} \) \textbf{proof} (\textbf{induction rule: i-term-closure.induct})

case \( 1 \) \( \textit{x} \)
then show \( \textit{?case} \)
using \( \textbf{assms i-term-closure.intro}(1) \) \( \textbf{i-term-closure.intro}(2) \) by \textbf{blast}

\textbf{next}

case \( 2 \) \( \textit{x} \) \textit{s' is d} \)
then show \( \textit{?case} \)
using \( \textbf{i-term-closure.intro}(2) \) by \textbf{blast}

\textbf{next}

case \( 3 \) \( \textit{x} \) \textit{s' f tsp} \textit{tr2} \)
then show \( \textit{?case} \)
using \( \textbf{i-term-closure.intro}(3) \) by \textbf{blast}

\textbf{qed}

\textbf{qed}

\}

ultimately show \( \textit{?thesis} \) using \( \textbf{assms} \) by \textbf{fast}

\textbf{qed}

\textbf{lemma} \textit{i-term-closure-args}:
\textbf{fixes} \( \textit{tr}:: \textbf{i-term ref} \)
\textbf{and} \( \textit{tsp}:: \textit{i-termsP} \)
\textbf{and} \( \textit{isr}:: \textbf{i-term ref} \)
\textbf{and} \( \textit{f}:: \textbf{string} \)
\textbf{and} \( \textit{s}:: \textbf{nat} \)
\textbf{and} \( \textit{h}:: \textbf{heap} \)
\textbf{assumes} \( \textbf{Ref.get} \textit{h} \textit{tr} = \textbf{ITerm}(\textit{s}, \textbf{None}, \textbf{ITermD}(\textit{f}, \textit{tsp})) \)
\textbf{shows} \( \textbf{i-term-closure} \textit{h (Some} \textit{tr}) = \textbf{insert} \textit{tr (i-terms-closure} \textit{h tsp}) \)
\textbf{proof} –
\{
\textbf{fix} \( \textit{tp} \) \( \textit{x} \)

\}
have $x \in \text{i-term-closure } h \text{ tp} \implies$

$tp = \text{Some } tr \implies$

Ref.get $h \text{ tr} = I\text{Term } (s, \text{None, I}\text{TermD } f, \text{ tsp}) \implies$

$x = tr \lor (\exists t2r \in \text{i-terms-set } h \text{ tsp}. \ x \in \text{i-term-closure } h \text{ (Some } t2r))$

proof (induction rule: i-term-closure.induct)

case (1 tr)
then show ?case by blast

next

case (2 tr' s' is uv)
then show ?case
proof (cases tr' = tr)

case True
then show ?thesis using 2 by (simp add: i-term-closure.intros(1))

next

case False
then show ?thesis using 2 i-term-closure.intros(2) by blast

qed

next

case (3 tr' s' f tsp tr2)
then have $\land tr', tr2 \in \text{i-term-closure } h \text{ tr'' } \lor \text{ tr' } \notin \text{i-term-closure } h \text{ tr''}$
using i-term-closure.intros(3) by blast
then show ?case using 3 i-term-closure.intros(1) by fastforce

qed

} then have i-term-closure $h \text{ (Some } tr) \subseteq \text{ insert } tr \text{ (i-terms-closure } h \text{ tsp)}$
by (simp add: assms subsetI)

moreover {
fix $x$
assume $x \in \text{ insert } tr \text{ (i-terms-closure } h \text{ tsp)}$
then consider (a) $x = tr \mid (b) \exists t2r \in \text{i-terms-set } h \text{ tsp}. \ x \in \text{i-term-closure } h$ (Some t2r)
by blast

then have $x \in \text{i-term-closure } h $ (Some tr)
proof (cases)
case a
then show ?thesis using i-term-closure.intros(1) by blast

next

case b
then obtain t2r where t2r \in i-terms-set h tsp \land x \in i-term-closure h (Some t2r)
by blast

moreover have $x \in \text{i-term-closure } h $ (Some t2r) \implies

t2r \in i-terms-set h tsp \implies
x \in i-term-closure h (Some tr)
proof (induction rule: i-term-closure.induct)
case (1 x)
then show ?case
using assms i-term-closure.intros(1) i-term-closure.intros(3) by fast

next

42
case (2 x s' is d)
  then show ?case
    using i-term-closure.intros(2) by blast
next
  case (3 x s' f tsp tr2)
  then show ?case
    using i-term-closure.intros(3) by blast
qed
ultimately show ?thesis by blast
qed

then have insert tr (i-terms-closure h tsp) ⊆ i-term-closure h (Some tr) by blast
ultimately show ?thesis by blast
qed

lemma i-terms-closure-terms:
assumes Ref.get h tsr = ITerms(tthisr, tsnextp)
shows i-terms-closure h (Some tsr) =
  (i-term-closure h (Some tthisr)) ∪ (i-terms-closure h tsnextp)
by (simp add: assms i-terms-set-insert)

lemma i-term-closure-sublists-terms:
assumes Ref.get h tsr = ITerms(tthisr, tsnextp)
shows i-terms-closure-sublists h (Some tsr) =
  insert (Some tsr) (i-term-closure-sublists h (Some tthisr) ∪
  i-terms-closure-sublists h tsnextp)
proof (intro Set.equalityI subsetI)
  fix tsp'
  assume tsp' ∈ i-terms-closure-sublists h (Some tsr)
  then consider (a) tsp' ∈ i-terms-sublists h (Some tsr) |
    (b) tsp' ∈ (⋃ tr∈i-terms-closure h (Some tsr). i-term-sublists h tr)
    by blast
  then show tsp' ∈ insert (Some tsr) (i-term-closure-sublists h (Some tthisr) ∪
  i-terms-closure-sublists h tsnextp)
proof (cases)
  case a
  then show ?thesis
    using assms i-terms-sublists-insert by fast
next
  case b
  then show ?thesis
    using assms i-terms-closure-terms by fastforce
qed
next
  show ∀ x. x ∈ insert (Some tsr) (i-term-closure-sublists h (Some tthisr) ∪
        i-terms-closure-sublists h tsnextp) ℄ x ∈ i-terms-closure-sublists h (Some tsr)
using assms i-terms-closure-terms i-terms-sublists-insert by force

qed

lemma i-terms-sublists-someE[elim]:
assumes tsr-sublist-tr: Some tsr ∈ i-term-sublists h tr
obtains s f is tsp0
where Ref.get h tr = ITerm (s, is, ITermD (f, tsp0))
and Some tsr ∈ i-terms-sublists h tsp0

proof –
obtain s is d where
  t1: Ref.get h tr = ITerm(s, is, d)
  using get-stamp.cases by blast
have t2: get-ITerm-args (Ref.get h tr) ≠ None
  using i-terms-sublists-None-om tsr-sublist-tr by force
with t1 obtain f tsp0 where t3: d = ITermD(f, tsp0)
  using tsr-sublist-tr get-ITerm-args-iff-ex by force
have get-ITerm-args (Ref.get h tr) = tsp0
  by (simp add: get-ITerm-args-iff-ex t1 t3)
then have Some tsr ∈ i-terms-sublists h tsp0
  using tsr-sublist-tr by blast
with t1 t3 that show ?thesis by presburger

qed

lemma i-term-closure-finite:
fixes tp:: i-termP
  and h:: heap
assumes i-term-acyclic h tp
shows finite (i-term-closure h tp)
using assms proof (induction rule: i-term-acyclic-induct)
  case (nil h)
then show ?case using i-term-closure-None by simp
next
  case (var h tr s)
then show ?case using i-term-closure-var by force
next
  case (link h tr s isr)
then show ?case using i-term-closure-link by force
next
  case (args h tr tsp s f)
then show ?case using i-term-closure-args i-terms-set-finite by force
qed

lemma i-term-closure-acyclic:
fixes tp:: i-termP
  and t2r:: i-term ref
  and h:: heap
assumes acyclic: i-term-acyclic h tp
  and t2r-mem: t2r ∈ i-term-closure h tp
shows i-term-acyclic h (Some t2r)
using acyclic t2r-mem acyclic
proof (induction rule: i-term-acyclic-induct)
case (nil h)
then show ?case using i-term-closure-None by simp
next
case (var h tr s)
then show ?case using i-term-closure-var t-acyclic-nil t-acyclic-step-link by fast
next
case (link h tr s isr)
then show ?case using i-term-closure-link acyclic-term-link-simp by fast
next
case (args h tr tsp s f)
then consider
  (a) t2r = tr |
  (b) t2r0 where t2r0 ∈ i-terms-set h tsp ∧ t2r ∈ i-term-closure h (Some t2r0)
using i-term-closure-args by blast
then show ?case proof (cases)
case a
then show ?thesis
  by (simp add: args.prems(2))
next
case b
then have i-term-acyclic h (Some t2r0)
  using i-terms-set-acyclic args.hyps(1) by blast
then show ?thesis
  using args.IH b by blast
qed
qed

lemma i-term-acyclic-closure-induct [consumes 1, case-names in-closure]:
  fixes h :: heap
  and tp :: i-termP
  and P :: heap ⇒ i-termP ⇒ bool
  assumes acyclic: i-term-acyclic h tp
  and step:
    ∩ h tp. ( t2r ∈ i-term-closure h tp ➝
    Some t2r ≠ tp ➝
    P h (Some t2r)) ➝
    P h tp
  shows P h tp
proof
  have ∩ t2r. t2r ∈ i-term-closure h tp ➝ P h (Some t2r)
  using acyclic proof (induction h tp rule: i-term-acyclic-induct)
case (nil h)
then show ?case
using i-term-closure-None by simp
next
case (var h tr s)
  then show ?case
  using i-term-closure-var step by fastforce
next
case (link h tr isr s)
  then consider (a) t2r = tr | (b) t2r ∈ i-term-closure h (Some isr)
  using i-term-closure-link by blast
  then show ?case proof (cases)
    case a
    then show ?thesis using i-term-closure-link step link.IH link.hyps
      by (metis insertE)
    next
    case b
    then show ?thesis
      using link.IH by blast
  qed
next
case (args h tr tsp s f)
  then consider
    (a) t2r = tr |
    (b) t2r0 where t2r0 ∈ i-terms-set h tsp ∧ t2r ∈ i-term-closure h (Some t2r0)
  using i-term-closure-args by blast
  then show ?case proof (cases)
    case a
    then have \( \forall t2r. \) 
      \( t2r \in \text{i-term-closure } h (\text{Some } tr) \implies \) 
      \( \text{Some } t2r \neq \text{Some } tr \implies \) 
      \( P h (\text{Some } t2r) \) 
    using args.IH args.hyps(2) i-term-closure-args by fast
    then show ?thesis
      using a step by blast
  next
  case b
  then show ?thesis
    using args.IH by blast
  qed
  qed
  then show ?thesis
    using step by blast
  qed

lemma i-term-acyclic-closure-induct [consumes 1, case-names nil var link args]:
  fixes h:: heap
  and tp:: i-termP
  and P:: heap ⇒ i-termP ⇒ bool
  assumes acyclic: i-term-acyclic h tp
  and nil-case: \( h. \) P h None
and \textit{var-case}: \( \bigwedge h \ tr \ s. \)
\[
\text{Ref.get} \ h \ tr = \text{ITerm}(s, \text{None}, \text{IVarD}) \implies \\
P h (\text{Some tr})
\]
and \textit{link-case}: \( \bigwedge h \ tr \ isr \ s. \)
\[
(\bigwedge t2r. \ t2r \in \text{i-term-closure} \ h (\text{Some isr}) \implies \\
P h (\text{Some t2r})) \implies \\
\text{Ref.get} \ h \ tr = \text{ITerm}(s, \text{Some isr}, \text{IVarD}) \implies \\
P h (\text{Some tr})
\]
and \textit{args-case}: \( \bigwedge h \ tr \ tsp \ s \ f. \)
\[
(\bigwedge t2r0 \ t2r. \\
t2r \in \text{i-term-closure} \ h (\text{Some t2r0}) \implies \\
t2r0 \in \text{i-terms-set} \ h \ tsp \implies \\
P h (\text{Some t2r})) \implies \\
\text{Ref.get} \ h \ tr = \text{ITerm}(s, \text{None}, \text{ITermD}(f, \ tsp)) \implies \\
P h (\text{Some tr})
\]
shows \( P h \ tp \)

\textbf{proof} –

have \( \bigwedge t2r. \ t2r \in \text{i-term-closure} \ h \ tp \implies \\
P h (\text{Some t2r}) \)

using \textit{acyclic proof} (induction \( h \ tp \) rule: \textit{i-term-acyclic-induct})

case (\text{nil} \( h \))

then show \( \text{?case using i-term-closure-None by simp} \)

next

case (\text{var} \( h \ tr \ s \))

then show \( \text{?case using i-term-closure-var \ var-case by fast} \)

next

case (\text{link} \( h \ tr \ isr \ s \))

then consider (a) \( t2r = \text{tr} \mid (b) \ t2r \in \text{i-term-closure} \ h (\text{Some isr}) \)

using \textit{i-term-closure-link by blast}

then show \( \text{?case proof (cases)} \)

case a

then show \( \text{?thesis} \)

using \textit{link.IH link.hyps link-case by blast}

next

case b

then show \( \text{?thesis} \)

using \textit{link.IH by blast}

qed

next

case (\text{args} \( h \ tr \ tsp \ s \ f \))

then consider

(a) \( t2r = \text{tr} \mid (b) \ t2r0 \in \text{i-terms-set} \ h \ tsp \wedge t2r \in \text{i-term-closure} \ h (\text{Some t2r0}) \)

using \textit{i-term-closure-args by blast}

then show \( \text{?case proof (cases)} \)

case a

then have \( \bigwedge t2r. \\
t2r \in \text{i-term-closure} \ h (\text{Some tr}) \implies \\
\text{Some t2r} \neq \text{Some tr} \implies \\
P h (\text{Some t2r}) \)

using \textit{args.IH args.hyps(2) i-term-closure-args by fast}
then show thesis
    using a args.IH args.hyps(2) args-case by blast
next
  case b
then show thesis
    using args.IH by blast
qed
qed
then show thesis using acyclic nil-case i-term-closure.intros(1)
  by (metis not-None-eq)
qed

lemma i-term-acyclic-closure-inductc' [consumes 1, case-names var link args]:
fixes h:: heap
  and tr:: i-term ref
  and P:: heap ⇒ i-term ref ⇒ bool
assumes acyclic: i-term-acyclic h (Some tr)
  and var-case: ∃h tr s.
    Ref.get h tr = ITerm(s, None, IVarD) ⇒
      P h tr
  and link-case: ∃h tr isr s.
    (∀t2r. t2r ∈ i-term-closure h (Some isr) ⇒ P h t2r) ⇒
    Ref.get h tr = ITerm(s, Some isr, IVarD) ⇒
      P h tr
  and args-case: ∃h tr tsp s f.
    (∀t2r0 t2r. t2r ∈ i-term-closure h (Some t2r0) ⇒
      t2r0 ∈ i-terms-set h tsp ⇒
        P h t2r) ⇒
    Ref.get h tr = ITerm(s, None, ITermD(f, tsp)) ⇒
      P h tr
shows P h tr
using assms
by (induction h (Some tr) arbitrary: tr rule: i-term-acyclic-closure-inductc) blast+

lemma i-term-closure-link-same-cyclic:
fixes tr :: i-term ref
  and isr :: i-term ref
  and d :: i-term-d
  and s :: nat
  and h :: heap
assumes Ref.get h tr = ITerm(s, Some isr, d)
  and tr ∈ i-term-closure h (Some isr)
sows ¬i-term-acyclic h (Some tr)
proof
  have i-term-acyclic h (Some tr) ⇒
    Ref.get h tr = ITerm(s, Some isr, d) ⇒
    tr ∈ i-term-closure h (Some isr) ⇒
    False
by (induction rule: i-term-acyclic-closure-inductc')
(simp, fastforce, force)
then show ?thesis using assms by blast
qed

lemma i-term-closure-args-same-cyclic:
fixes tr :: i-term ref
and tsp :: i-terms ref option
and f :: string
and s :: nat
and h :: heap
assumes Ref.get h tr = ITerm(s, None, ITermD(f, tsp))
and ∃t2r ∈ i-terms-set h tsp. tr ∈ i-term-closure h (Some t2r)
shows ¬i-term-acyclic h (Some tr)
proof −
have i-term-acyclic h (Some tr) ⇒
Ref.get h tr = ITerm(s, None, ITermD(f, tsp)) ⇒
∃t2r ∈ i-terms-set h tsp. tr ∈ i-term-closure h (Some t2r) ⇒ False
by (induction rule: i-term-acyclic-closure-inductc')
(simp, force, auto)
then show ?thesis using assms by blast
qed

lemma i-term-closure-trans:
fixes tr0 :: i-term ref
and tr1 :: i-term ref
and tr2 :: i-term ref
and h :: heap
assumes tr1-mem: tr1 ∈ i-term-closure h (Some tr0)
and tr2-mem: tr2 ∈ i-term-closure h (Some tr1)
shows tr2 ∈ i-term-closure h (Some tr0)
using tr1-mem tr2-mem proof (induction tr1 rule: i-term-closure.induct)
case (1 tr)
then show ?case by simp
next
case (2 tr uu is uv)
then show ?case
using i-term-closure-link by blast
next
case (3 tr uu ux tsp tr2)
then show ?case
using i-term-closure-args by fast
qed

definition is-closed where
is-closed h trs = (i-term-closures h trs = trs)

lemma i-term-closures-idem:
\[ i\text{-term-closures } h (i\text{-term-closures } h \text{ trs}) = i\text{-term-closures } h \text{ trs} \]

**proof**

- have \( i\text{-term-closures } h (i\text{-term-closures } h \text{ trs}) \supseteq i\text{-term-closures } h \text{ trs} \)
  - using \( i\text{-term-closure}.\text{intros}(1) \) by \( \text{fastforce} \)

moreover {
  
  fix \( tr \)
  
  assume \( tr \in i\text{-term-closures } h (i\text{-term-closures } h \text{ trs}) \)
  
  then obtain \( tr0 \)
  
  where \( tr \in i\text{-term-closure } h (\text{Some } tr0) \)
  
  and \( tr0\text{-mem: } tr0 \in i\text{-term-closures } h \text{ trs} \)
  
  by \( \text{fast} \)
  
  then have \( tr \in i\text{-term-closures } h \text{ trs} \)
  
  **proof** (induction \( tr \) rule: \( i\text{-term-closure}.\text{induct} \))
  
  case \( (1 \ tr) \)
  
  then show \( ?\text{case} \)
  
  by \( \text{blast} \)
  
  next
  
  case \( (2 \ tr \ uu \ is \ uv) \)
  
  then show \( ?\text{case} \)
  
  by \( \text{metis UN-iff } i\text{-term-closure.\text{intros}(2)} \)
  
  next
  
  case \( (3 \ tr \ uu \ wx \ tsp \ tr2) \)
  
  then show \( ?\text{case} \)
  
  by \( \text{metis (full-types) UN-iff } tr0\text{-mem } i\text{-term-closure.\text{intros}(3)} \)
  
  qed

} ultimately show \( ?\text{thesis} \) by \( \text{fastforce} \)

qed

**lemma** \( i\text{-terms-closure-is-closed} : \)

shows \( \text{is-closed } h (i\text{-terms-closure } h \text{ tsp}) \)

by \( (\text{meson } i\text{-term-closures-idem } \text{is-closed-def}) \)

**lemma** \( i\text{-term-closure-is-closed} : \)

shows \( \text{is-closed } h (i\text{-term-closure } h \text{ tp}) \)

**proof** (cases \( tp \))

 case \( \text{None} \)

 then show \( ?\text{thesis unfolding } \text{is-closed-def} \)

 by \( (\text{simp add: } i\text{-term-closure-None}) \)

 next

 case \( (\text{Some } tr) \)

 have \( i\text{-term-closure } h (\text{Some } tr) = i\text{-term-closures } h \{ tr \} \)

 by \( \text{simp} \)

 then show \( ?\text{thesis unfolding } \text{is-closed-def} \)

 using \( i\text{-term-closures-idem } \text{Some by } \text{presburger} \)

qed

definition \( i\text{-term-closure-v } \text{where} \)

\( i\text{-term-closure-v } h \text{ tp} = \text{Ref.\,get } h \vdash i\text{-term-closure } h \text{ tp} \)
inductive-set

\( i\text{-term-chain} \) for \( h:: \text{heap} \) and \( tr:: \text{i-term ref} \) where

link:

\( tr' \in i\text{-term-chain} h \ tr \implies \)  
\( \text{Ref}.\text{get} h \ tr' = \text{ITerm}(s, \text{Some} \ tnextr, d) \implies \)  
\( \text{tnextr} \in i\text{-term-chain} h \ tr \mid \)  
self: \( tr \in i\text{-term-chain} h \ tr \)

lemma \( i\text{-term-chain-dest}: \)

fixes \( tr:: \text{i-term ref} \)
and \( d:: \text{i-term-d} \)
and \( s:: \text{nat} \)
and \( h:: \text{heap} \)

assumes \( \text{Ref}.\text{get} h \ tr = \text{ITerm}(s, \text{None}, d) \)
shows \( i\text{-term-chain} h \ tr = \{tr\} \)

proof –

\{  
fix \( x \)
assume \( x \in i\text{-term-chain} h \ tr \)
then have \( x = tr \)
using assms by (induction rule: \( i\text{-term-chain}.\text{induct} \), simp+)
\}
then show \( \text{thesis} \)
using \( i\text{-term-chain}.\text{self} \) by blast
qed

lemma \( i\text{-term-chain-link}: \)

fixes \( tr:: \text{i-term ref} \)
and \( tr0:: \text{i-term ref} \)
and \( s:: \text{nat} \)
and \( d:: \text{i-term-d} \)
and \( h:: \text{heap} \)

assumes \( \text{Ref}.\text{get} h \ tr = \text{ITerm}(s, \text{Some} \ tr0, d) \)
shows \( i\text{-term-chain} h \ tr = \text{insert} \ tr \ (i\text{-term-chain} h \ tr0) \)

proof –

\{  
fix \( x \)
assume \( x \in i\text{-term-chain} h \ tr \)
then have \( x \in \text{insert} \ tr \ (i\text{-term-chain} h \ tr0) \)
proof (induction rule: \( i\text{-term-chain}.\text{induct} \))
\( \text{case} \ (\text{link} \ tr' \ s \ tnextr \ d) \)
show \( \text{thesis} \)
proof (cases \( tr' = tr \))
\( \text{case} \ True \)
then show \( \text{thesis} \)
using \( i\text{-term-chain}.\text{self} \) \( \text{assms} \ \text{link}.\text{hyps}(2) \) by auto
next
\( \text{case} \ False \)
then show \( \text{thesis} \)
using \( i\text{-term-chain}.\text{link} \) \( \text{IH} \) \( \text{link}.\text{hyps}(2) \) by blast
\}
qed
next
case self
  then show ?case by simp
qed
}
moreover
{
  fix x
  assume x ∈ insert tr (i-term-chain h tr₀)
  then consider (a) x = tr | (b) x ∈ i-term-chain h tr₀ by blast
  then have x ∈ i-term-chain h tr
  proof (cases)
    case a
    then show ?thesis
      by (simp add: i-term-chain.self)
  next
    case b
    then show ?thesis
  proof (induction rule: i-term-chain.induct)
    case (link tr' s trnext d)
    then show ?case
      using i-term-chain.link by blast
  next
    case self
    then show ?case
      using assms i-term-chain.link i-term-chain.self by blast
  qed
}
ultimately show ?thesis by blast
qed

lemma i-term-chain-acyclic:
  fixes tr:: i-term ref
  and tr':: i-term ref
  and h:: heap
  assumes acyclic: i-term-acyclic h (Some tr)
  and tr'-mem: tr' ∈ i-term-chain h tr
  shows i-term-acyclic h (Some tr')
  using acyclic tr'-mem acyclic
  proof (induction rule: i-term-acyclic-induct')
    case (var h tr s)
    then show ?case
      using i-term-chain-dest t-acyclic-nil t-acyclic-step-link by fast
  next
    case (link h tr isr s)
    then consider (a) tr' = tr | (b) tr' ∈ i-term-chain h isr
    using i-term-chain-link by blast
then show \( \text{thesis} \) using \( \text{link.prems}(2) \) by simp

next
  case \( b \)
  moreover have \( \text{i-term-acyclic} \ h \ (\text{Some} \ \text{isr}) \)
    using \( \text{link.hyps} \ \text{link.prems}(2) \) \( \text{acyclic-term-link-simp} \)
    by blast
  ultimately show \( \text{thesis} \) using \( \text{link.IH} \) by blast
qed

next
  case \( \text{args} \ h \ \text{tsp} \ s \ f \)
  then show \( \text{thesis} \)
    using \( \text{i-term-chain-dest} \ \text{t-acyclic-step-ITerm} \) by fast
qed

lemma \( \text{i-term-chain-subset-closure} \):
  fixes \( \text{tr} \) :: \( \text{i-term ref} \)
  and \( \text{h} \) :: \( \text{heap} \)
  shows \( \text{i-term-chain} \ h \ \text{tr} \subseteq \text{i-term-closure} \ h \ (\text{Some} \ \text{tr}) \)
proof (intro subsetI)
  fix \( \text{tr}' \)
  assume \( \text{tr}' \in \text{i-term-chain} \ h \ \text{tr} \)
  then show \( \text{tr}' \in \text{i-term-closure} \ h \ (\text{Some} \ \text{tr}) \)
    proof (induction \( \text{tr}' \)
      rule: \( \text{i-term-chain} \).inducts)
      case \( \text{link} \ \text{tr}' \ s \ \text{tnextr} \ d \)
      then show \( \text{thesis} \)
        using \( \text{i-term-closure}.\text{intros}(2) \) by blast
    next
    case \( \text{self} \)
    then show \( \text{thesis} \)
      using \( \text{i-term-closure}.\text{intros}(1) \) by blast
    qed
qed

lemma \( \text{i-term-chain-linkE} \): where \( \text{Ref.get} \ h \ \text{tr} = \text{ITerm}(s, \ \text{Some} \ \text{tnextr}, \ d) \)
  and \( \text{tr}' \in \text{i-term-chain} \ h \ \text{tnextr} \)
using \( \text{assms} \)
proof (atomize-calc, induction rule: \( \text{i-term-chain}.\text{induct} \))
case \( \text{link} \ \text{tr}' \ s \ \text{tnextr} \ d \)
  show \( \text{thesis} \)
    using \( \text{i-term-chain}.\text{link} \ \text{i-term-chain}.\text{self} \) \( \text{link.IH} \) \( \text{link.hyps}(1) \) \( \text{link.hyps}(2) \) by blast
next
  case \( \text{self} \)
  then show \( \text{thesis} \) by blast
fun i-maxstamp:: heap ⇒ i-termP ⇒ nat where
i-maxstamp h None = 0
| i-maxstamp h tp = Max (get-stamp ' i-term-closure-v h tp)

lemma i-maxstamp-is-max:
fixes t1p:: i-termP
and t2r:: i-term ref
and is:: i-termP
and d:: i-term-d
and h:: heap
assumes acyclic: i-term-acyclic h t1p
and t2r-get: Ref.get h t2r = ITerm(s, is, d)
and t2r-mem: t2r ∈ i-term-closure h t1p
shows s ≤ i-maxstamp h t1p
proof (cases t1p)
case None
then show thesis using t2r-mem i-term-closure-None by simp
next
case (Some t1r)
have ITerm(s, is, d) ∈ i-term-closure-v h t1p
  unfolding i-term-closure-v-def
  using t2r-get t2r-mem image-iff by fastforce
then have s ∈ get-stamp ' i-term-closure-v h t1p
  by force
moreover have finite (get-stamp ' i-term-closure-v h t1p)
  by (simp add: acyclic i-term-closure-finite i-term-closure-v-def)
ultimately show thesis
  by (simp add: Some)
qed

lemma i-maxstamp-closure-trans:
fixes t1p:: i-termP
and t2r:: i-term ref
and is:: i-termP
and d:: i-term-d
and h:: heap
assumes acyclic: i-term-acyclic h t1p
and t2r-mem: t2r ∈ i-term-closure h t1p
shows i-maxstamp h (Some t2r) ≤ i-maxstamp h t1p
proof (cases t1p)
case None
then show thesis using t2r-mem i-term-closure-None by simp
next
case (Some t1r)
  fix s assume s ∈ get-stamp ' i-term-closure-v h (Some t2r)
  then have s ∈ get-stamp ' i-term-closure-v h t1p

qed
unfolding \textit{i-term-closure-v-def}

using \textit{i-term-closure-trans Some t2r-mem} by blast

} then have \(*\): get-stamp \(\cdot\) i-term-closure-v \(\cdot\) (Some t2r) \(\subseteq\) get-stamp \(\cdot\) i-term-closure-v \(\cdot\) t1p

by blast

moreover have \(\text{finite (get-stamp \(\cdot\) i-term-closure-v \(\cdot\) t1p)}\)

by (simp add: acyclic i-term-closure-finite i-term-closure-v-def)

ultimately show \(\text{thesis}\)

by (simp add: Some)

(metis Max.antimono empty-iff i-term-closure.intros(1)

i-term-closure-v-def image-is-empty)

qed

definition heap-only-stamp-changed:: i-term ref set \(\Rightarrow\) heap \(\Rightarrow\) heap \(\Rightarrow\) bool where

heap-only-stamp-changed \(\text{trs h h'} = (\forall \text{ typ x. heaprefs h typ x \neq heaprefs h' typ x \rightarrow (typ \neq \text{TYPEREP(i-term)} \land typ \neq \text{TYPEREP(i-terms)} \land typ \neq \text{TYPEREP(nat)}) \lor (\exists s s' is d. typ = \text{TYPEREP(i-term)} \land Ref x \in \text{trs} \land from-nat (heaprefs h typ x) = \text{ITerm}(s, is, d) \land from-nat (heaprefs h' typ x) = \text{ITerm}(s', is, d)))\)

abbreviation heap-only-stamp-changed-tr where

heap-only-stamp-changed-tr \(\text{trs h} \equiv\) heap-only-stamp-changed \(\text{(i-term-closure h (Some tr)) h}\)

abbreviation heap-only-stamp-changed-ts where

heap-only-stamp-changed-ts \(\text{tsp h} \equiv\) heap-only-stamp-changed \(\text{(i-terms-closure h tsp) h}\)

lemma heap-only-stamp-ch-nt:

fixes \(\text{trs:: i-term ref set}\)

and \(\text{r::'a::heap ref}\)

and \(\text{v::'a::heap}\)

and \(\text{h::heap}\)

assumes \(\text{TYPEREP('a) \neq TYPEREP(i-term)}\)

and \(\text{TYPEREP('a) \neq TYPEREP(i-terms)}\)

and \(\text{TYPEREP('a) \neq TYPEREP(nat)}\)

shows \(\text{heap-only-stamp-changed \(\text{trs h (Ref.set r v h)}\)}\)

unfolding heap-only-stamp-changed-def Ref.set-def using \text{assms} by simp

lemma heap-only-stamp-ch-term:

fixes \(\text{trs:: i-term ref set}\)

and \(\text{r:: i-term ref}\)

and \(\text{is:: i-termP}\)

and \(\text{d:: i-term-d}\)

and \(\text{s:: nat}\)
and \( s' \): nat
and \( h \): heap
assumes \( \text{Ref.get \( h \) \( r = ITerm(s, is, d) \)} \)
and \( r \in \text{trs} \)
shows heap-only-stamp-changed \( \text{trs \( h \)} (\text{Ref.set \( r \) (\( ITerm(s', is, d) \)) \( h \)}) \)
unfolding heap-only-stamp-changed-def \( \text{Ref.set-def using assms} \)
by (simp add: \( \text{Ref.get-def} \))
(metis addr-of-ref.simps addr-of-ref-inj)

lemma heap-only-stamp-ch-get-term:
fixes \( \text{trs} \):: i-term ref set
and \( \text{tr} \):: i-term ref
and \( h \):: heap
and \( h' \):: heap
assumes heap-only-stamp-changed \( \text{trs} \( h \) \( h' \))
and \( \text{Ref.get \( h' \) \( tr = ITerm(s, is, d) \)} \)
shows \( \exists s'. \text{Ref.get \( h' \) \( tr = ITerm(s', is, d) \)} \)
proof (rule case-split)
assume \( \text{heap.refs \( h \) TYPEREP(i-term) (addr-of-ref \( tr \)) = heap.refs \( h' \) TYPEREP(i-term) (addr-of-ref \( tr \))} \)
then show \( \exists \text{thesis} \)
using assms[unfolded heap-only-stamp-changed-def]
by (simp add: \( \text{Ref.get-def} \))
next
assume \( \text{heap.refs \( h \) TYPEREP(i-term) (addr-of-ref \( tr \)) } \neq \text{heap.refs \( h' \) TYPEREP(i-term) (addr-of-ref \( tr \))} \)
then show \( \exists \text{thesis} \)
using assms[unfolded heap-only-stamp-changed-def]
by (simp add: \( \text{Ref.get-def, fastforce} \))
qed

lemma heap-only-stamp-ch-get-term':
fixes \( \text{trs} \):: i-term ref set
and \( \text{tr} \):: i-term ref
and \( h \):: heap
and \( h' \):: heap
assumes heap-only-stamp-changed \( \text{trs} \( h \) \( h' \))
and \( \text{Ref.get \( h' \) \( tr = ITerm(s, is, d) \)} \)
shows \( \exists s'. \text{Ref.get \( h \) \( tr = ITerm(s', is, d) \)} \)
proof (rule case-split)
assume \( \text{heap.refs \( h \) TYPEREP(i-term) (addr-of-ref \( tr \)) = heap.refs \( h' \) TYPEREP(i-term) (addr-of-ref \( tr \))} \)
then show \( \exists \text{thesis} \)
using assms[unfolded heap-only-stamp-changed-def]
by (simp add: \( \text{Ref.get-def} \))
next
assume \( \text{heap.refs \( h \) TYPEREP(i-term) (addr-of-ref \( tr \)) } \neq \text{heap.refs \( h' \) TYPEREP(i-term) (addr-of-ref \( tr \))} \)
then show \( \exists \text{thesis} \)
using assms[unfolded heap-only-stamp-changed-def]
by (simp add: Ref.get-def, fastforce)
qed

lemma heap-only-stamp-ch-get-term-nclos:
  fixes trs:: i-term ref set
  and tr:: i-term ref
  and h:: heap
  and h':: heap
  assumes heap-only-stamp-changed trs h h'
  and tr ∉ trs
  shows Ref.get h' tr = Ref.get h tr
proof
  { 
    assume heap.refs h TYPEREPS(i-term) (addr-of-ref tr) ≠
    heap.refs h' TYPEREPS(i-term) (addr-of-ref tr)
    then have tr ∈ trs
      using assms[unfolded heap-only-stamp-changed-def]
      by (metis addr-of-ref.simps addr-of-ref-inj)
  }
  then show ?thesis
    by (metis Ref.get-def assms(2) comp-apply)
qed

lemma heap-only-stamp-ch-get-terms:
  fixes trs:: i-term ref set
  and tsr:: i-terms ref
  and h:: heap
  and h':: heap
  assumes heap-only-stamp-changed trs h h'
  shows Ref.get h tsr = Ref.get h' tsr
proof (rule case-split)
  assume heap.refs h TYPEREPS(i-terms) (addr-of-ref tsr) ≠
  heap.refs h' TYPEREPS(i-terms) (addr-of-ref tsr)
  then show ?thesis
    using assms[unfolded heap-only-stamp-changed-def]
    by (simp add: Ref.get-def)
next
  assume heap.refs h TYPEREPS(i-terms) (addr-of-ref tsr) ≠
  heap.refs h' TYPEREPS(i-terms) (addr-of-ref tsr)
  then show ?thesis
    using assms[unfolded heap-only-stamp-changed-def] typerep-term-neq-terms by
    fastforce
qed

lemma heap-only-stamp-ch-get-nat:
  fixes ir:: nat ref
  assumes heap-only-stamp-changed trs h h'
  shows Ref.get h ir = Ref.get h' ir
using assms[unfolded heap-only-stamp-changed-def]
by (simp add: Ref.get-def Ref.set-def, metis typerep-term-neq-nat)

lemma heap-only-stamp-ch-sublists:
  fixes trs:: i-term ref set
  and tr:: i-term ref
  and tsp:: i-termsP
  and f:: string
  and s:: nat
  and h:: heap
  and h': heap
assumes heap-only-stamp-changed trs h h'
shows i-terms-sublists h tsp = i-terms-sublists h' tsp
proof –
  { fix x
    have x ∈ i-terms-sublists h tsp → x ∈ i-terms-sublists h tsp
      proof (induction rule: i-terms-sublists.induct)
        case (next tsr' tthis tnext)
        then have Ref.get h tsr' = ITerms (tthis, tnext)
          using heap-only-stamp-ch-get-terms assms
          by presburger
        then show ?case
          using i-terms-sublists.next next.IH next.prems by blast
        next
        case self
        then show ?case
          using i-terms-sublists.self by blast
      qed
    }
  moreover
  { fix x
    have x ∈ i-terms-sublists h tsp → x ∈ i-terms-sublists h tsp
      proof (induction rule: i-terms-sublists.induct)
        case (next tsr' tthis tnext)
        then have Ref.get h tsr' = ITerms (tthis, tnext)
          using heap-only-stamp-ch-get-terms assms
          by simp
        then show ?case
          using i-terms-sublists.next next.IH next.prems by blast
        next
        case self
        then show ?case
          using i-terms-sublists.self by blast
      qed
    }
  ultimately show ?thesis
    by auto
lemma heap-only-stamp-ch-terms-set:
  fixes trs :: i-term ref set
  and tr :: i-term ref
  and tsp :: i-termsP
  and f :: string
  and s :: nat
  and h :: heap
  and h' :: heap
assumes heap-only-stamp-changed trs h h'
shows i-terms-set h tsp = i-terms-set h' tsp
using assms heap-only-stamp-ch-sublists i-terms-set-def2
heap-only-stamp-ch-get-terms by auto

qed

lemma heap-only-stamp-ch-diff-in-clos:
  fixes tr0 :: i-term ref
  and tr1 :: i-term ref
  and h0 :: heap
  and h1 :: heap
assumes hosc: heap-only-stamp-changed-tr tr0 h h'
  and get-tr1: Ref.get h tr1 ≠ Ref.get h' tr1
shows tr1 ∈ i-term-closure h (Some tr0)
using heap-only-stamp-changed-def
proof –
  have heap.refs h TYPEREP(i-term) (addr-of-ref tr1) ≠
    heap.refs h' TYPEREP(i-term) (addr-of-ref tr1)
    using get-tr1
    by (metis Ref.get-def comp-apply)
  then have Ref (addr-of-ref tr1) ∈ i-term-closure h (Some tr0)
    using hosc\[unfolded heap-only-stamp-changed-def\] by blast
  then show ?thesis
    by (metis addr-of-ref.simps addr-of-ref-inj)
qed

lemma heap-only-stamp-ch-antimono:
  assumes heap-only-stamp-changed trs' h h'
  and trs' ⊆ trs
shows heap-only-stamp-changed trs h h'
proof –
  { 
    fix typ x
    assume heap.refs h typ x ≠ heap.refs h' typ x
    then consider
      (a) (typ ≠ TYPEREP(i-term) ∧ typ ≠ TYPEREP(i-terms) ∧ typ ≠ TYPEREP(nat))
      | (b) s s' is d where
        typ = TYPEREP(i-term) ∧
        Ref x ∈ trs' ∧
from-nat (heap.refs h typ x) = ITerm(s, is, d) ∧
from-nat (heap.refs h' typ x) = ITerm(s', is, d)

using assms[unfolded heap-only-stamp-changed-def]
by blast

then have (typ ≠ TYPEREPT(i-term) ∧ typ ≠ TYPEREPT(i-terms) ∧ typ ≠ TYPEREPT(nat)) ∨
(∃s s' is d. typ = TYPEREPT(i-term) ∧
Ref x ∈ trs ∧
from-nat (heap.refs h typ x) = ITerm(s, is, d) ∧
from-nat (heap.refs h' typ x) = ITerm(s', is, d))

proof (cases)
case a
then show ?thesis by blast
next
case b
then show ?thesis
using assms(2) i-term-closure-trans by blast
qed

lemma heap-only-stamp-ch-closantimono:
assumes heap-only-stamp-changed-tr tr' h h'
and tr' ∈ i-term-closure h (Some tr)
shows heap-only-stamp-changed-tr h h'
using assms heap-only-stamp-ch-antimono i-term-closure-trans by blast

lemma heap-only-stamp-ch-closure:
assumes heap-only-stamp-changed trs h h'
shows i-term-closure h' (Some tr) = i-term-closure h (Some tr)
proof –
{ fix x
have x ∈ i-term-closure h' (Some tr) ⇒ x ∈ i-term-closure h (Some tr)
proof (induction rule: i-term-closure.induct)
case (1 tr')
then show ?case
by (simp add: i-term-closure.intros(1))
next
case (2 tr' s is uv)
then obtain s' where Ref.get h tr' = ITerm (s', Some is, uv)
using assms heap-only-stamp-ch-get-term' by blast
then show ?case
using 2.IH i-term-closure.intros(2) by blast
next
case (3 tr' s f tsp tr2)
obtain s' where **: Ref.get h tr' = ITerm (s', None, ITermD (f, tsp))
using 3.IH 3.hyps(2) assms heap-only-stamp-ch-get-term' by blast
have tr2 ∈ i-terms-set h tsp
  using heap-only-stamp-ch-terms-set[OF assms] 3.hyps(3) by simp
then show ?case
  using ** 3.IH i-term-closure.intros(3) by blast
qed
}
moreover {
  fix x
  have x ∈ i-term-closure h (Some tr) ⇒ x ∈ i-term-closure h' (Some tr)
proof (induction rule: i-term-closure.induct)
  case (1 tr')
  then show ?case
    by (simp add: i-term-closure.intros(1))
next
case (2 tr' s is uv)
  then obtain s' where Ref.get h' tr'= ITerm (s', Some is, uv)
    using assms heap-only-stamp-ch-get-term by blast
  then show ?case
    using 2.IH i-term-closure.intros(2) by blast
next
case (3 tr' s f tsp tr2)
  obtain s' where **: Ref.get h' tr' = ITerm (s', None, ITermD (f, tsp))
    using 3.IH 3.hyps(2) assms heap-only-stamp-ch-get-term by blast
  have tr2 ∈ i-terms-set h' tsp
    using heap-only-stamp-ch-terms-set[OF assms] 3.hyps(3) by simp
  then show ?case
    using ** 3.IH i-term-closure.intros(3) by blast
qed
}
ultimately show ?thesis by blast
qed

lemma heap-only-stamp-ch-terms-closure:
  assumes heap-only-stamp-changed trs h h'
  shows i-terms-closure h' tsp = i-terms-closure h tsp
using assms heap-only-stamp-ch-closure heap-only-stamp-ch-terms-set by auto

lemma heap-only-stamp-ch-sym [sym]:
  assumes heap-only-stamp-changed trs h h'
  shows heap-only-stamp-changed trs h' h
using assms unfolding heap-only-stamp-changed-def
by (subst eq-sym-conv, blast)

lemma heap-only-stamp-ch-trans [trans]:
  assumes heap-only-stamp-changed trs h0 h1
  and heap-only-stamp-changed trs h1 h2
  shows heap-only-stamp-changed trs h0 h2
unfolding heap-only-stamp-changed-def
proof (intro allI impI)
  fix ty: typerep
  and x :: nat
  assume *: heap.refs h0 typ x ≠ heap.refs h2 typ x
  show (ty ≠ TYPEREP(i-term) ∧ ty ≠ TYPEREP(i-terms) ∧ ty ≠ TYPE-REP(nat)) ∨
    (∃ s s′ is d.
      tpy = TYPEREP(i-term) ∧
      Ref x ∈ trs ∧
      from-nat (heap.refs h0 typ x) = ITerm(s, is, d) ∧
      from-nat (heap.refs h2 typ x) = ITerm(s′, is, d))
proof (rule case-split)
  assume ty ≠ TYPEREP(i-term) ∧ ty ≠ TYPEREP(i-terms) ∧ ty ≠ TYPE-REP(nat)
  then show *thesis by simp
next
  assume **: ¬ (ty ≠ TYPEREP(i-term) ∧ ty ≠ TYPEREP(i-terms) ∧ ty ≠ TYPE-REP(nat))
  from * consider
    (a) heap.refs h0 typ x ≠ heap.refs h1 typ x ∨
    (b) heap.refs h0 typ x = heap.refs h1 typ x and
        heap.refs h1 typ x ≠ heap.refs h2 typ x
  by fastforce
  then show *thesis
proof (cases)
case a
  then obtain s0 s1 is d where
    from-nat (heap.refs h0 typ x) = ITerm(s0, is, d) and
    from-nat (heap.refs h1 typ x) = ITerm(s1, is, d)
  using ** assms(1)[unfolded heap-only-stamp-changed-def] by blast
  moreover from this a obtain s2 where
    from-nat (heap.refs h1 typ x) = ITerm(s1, is, d) and
    from-nat (heap.refs h2 typ x) = ITerm(s2, is, d)
  using ** assms(2)[unfolded heap-only-stamp-changed-def]
  by (cases heap.refs h1 typ x = heap.refs h2 typ x) fastforce+
  ultimately show *thesis
  using a assms(1) heap-only-stamp-changed-def by blast
next
case b
  then obtain s1 s2 is d where
    from-nat (heap.refs h1 typ x) = ITerm(s1, is, d) and
    from-nat (heap.refs h2 typ x) = ITerm(s2, is, d)
  using ** assms(2)[unfolded heap-only-stamp-changed-def] by blast
  moreover from this b obtain s0 where
    from-nat (heap.refs h0 typ x) = ITerm(s0, is, d) and
    from-nat (heap.refs h1 typ x) = ITerm(s1, is, d)
  using ** assms(2)[unfolded heap-only-stamp-changed-def] by fastforce+
  ultimately show *thesis
  using assms(1) assms(2) b(2) heap-only-stamp-ch-closure heap-only-stamp-changed-def
by blast
qed
qed

lemma heap-only-stamp-ch-refl:
  shows heap-only-stamp-changed trs h h
by (simp add: heap-only-stamp-changed-def)

lemma heap-only-stamp-ch-term-terms-acyclic:
  assumes heap-only-stamp-changed trs h h'
  shows ((i-term-acyclic h tp) → i-term-acyclic h' tp) ∧
           ((i-terms-acyclic h tsp) → i-terms-acyclic h' tsp)
proof –
  have ((i-term-acyclic h tp) → heap-only-stamp-changed trs h h' → i-term-acyclic h' tp) ∧
           ((i-terms-acyclic h tsp) → heap-only-stamp-changed trs h h' → i-terms-acyclic h' tsp)
  proof (induction rule: i-term-acyclic-i-terms-acyclic.induct)
    case (t-acyclic-nil h)
    then show ?case
      by (simp add: i-term-acyclic-i-terms-acyclic.t-acyclic-nil)
  next
    case (t-acyclic-step-link h t tref s)
    show ?case
    proof (intro impI)
      assume hosc: heap-only-stamp-changed trs h h'
      then obtain s' where Ref.get h' tref = ITerm (s', t, IVarD)
        using heap-only-stamp-ch-get-term t-acyclic-step-link.hyps(2) by blast
      then show i-term-acyclic h' (Some tref)
        using hosc i-term-acyclic-i-terms-acyclic.t-acyclic-step-link t-acyclic-step-link.IH
          by blast
      qed
    next
    case (t-acyclic-step-ITerm h tsref tref s f)
    show ?case
    proof (intro impI)
      assume hosc: heap-only-stamp-changed trs h h'
      then obtain s' where Ref.get h' tref = ITerm (s', None, ITermD (f, tsref))
        using heap-only-stamp-ch-get-term t-acyclic-step-ITerm.hyps(2) by blast
      then show i-term-acyclic h' (Some tref)
        using hosc i-term-acyclic-i-terms-acyclic.t-acyclic-step-ITerm t-acyclic-step-ITerm.IH
          by blast
      qed
    next
    case (ts-acyclic-nil uy)
    then show ?case
      by (simp add: i-term-acyclic-i-terms-acyclic.ts-acyclic-nil)
  next
case (ts-acyclic-step-ITerms h ts2ref tref tsref)
show ?case
proof (intro \(\text{impl}\))
  assume hosc: heap-only-stamp-changed trs h h'
  have Ref.get h' tsref = ITerms (tref, ts2ref)
    using heap-only-stamp-ch-get-terms hosc ts-acyclic-step-ITerms.hyps(3) by auto
then show \(i\text{-terms-acyclic} h'\) (Some tsref)
  using hosc \(i\text{-term-acyclic-i-terms-acyclic}.\) ts-acyclic-step-ITerms.IH by blast
qed
then show \(?\text{thesis}\) using assms by blast
qed

lemma heap-only-stamp-ch-term-acyclic:
assumes \(i\text{-term-acyclic} h\) tp
  and heap-only-stamp-changed trs h h'
shows \(i\text{-term-acyclic} h'\) tp
using assms heap-only-stamp-ch-term-terms-acyclic by blast

lemma heap-only-stamp-ch-terms-acyclic:
assumes \(i\text{-terms-acyclic} h\) tsp
  and heap-only-stamp-changed trs h h'
shows \(i\text{-terms-acyclic} h'\) tsp
using assms heap-only-stamp-ch-terms-acyclic by blast

lemma heap-only-stamp-ch-terms-set-antimono:
assumes hosc: heap-only-stamp-changed-tr tr' h h'
  and Ref.get h tr = ITerm(s, None, ITermD(f, tsp))
  and \(\text{tr'} \in \text{i-terms-set} h\) tsp
shows heap-only-stamp-changed-tr tr h h'
unfolding heap-only-stamp-changed-def
proof (intro allI \(\text{impl}\))
fix \(\text{typ}\) x
assume refs h typ x \(\neq\) refs h' typ x
then consider
  (a) typ \(\neq\) TYPEREQ(i-term) \(\land\) typ \(\neq\) TYPEREQ(i-terms) \(\land\) typ \(\neq\) TYPEREQ(nat) \(\lor\)
  (b) s s' is d where typ = TYPEREQ(i-term) \(\land\)
    Ref x \(\in\) i-term-closure h (Some tr') \(\land\)
    from-nat (refs h typ x) = ITerm (s, is, d) \(\land\)
    from-nat (refs h' typ x) = ITerm (s', is, d)
    using hosc[unfolded heap-only-stamp-changed-def] by blast
then show typ \(\neq\) TYPEREQ(i-term) \(\land\) typ \(\neq\) TYPEREQ(i-terms) \(\land\) typ \(\neq\) TYPEREQ(nat) \(\lor\)
  (\(\exists\) s s' is d)
    typ = TYPEREQ(i-term) \(\land\)
    Ref x \(\in\) i-term-closure h (Some tr) \(\land\)

64
from-nat (refs h typ x) = ITerm (s, is, d) ∧
from-nat (refs h' typ x) = ITerm (s', is, d))

proof (cases)
  case a
  then show thesis by simp

next
  case b
  then show thesis using assms [unfolded heap-only-stamp-changed-def]
  by (meson i-term-closure.intros(1) i-term-closure.intros(3) i-term-closure-trans)

qed

lemma heap-only-stamp-ch-tr-sym [sym]:
  assumes heap-only-stamp-changed-tr tr h h'
  shows heap-only-stamp-changed-tr tr h' h
  using assms heap-only-stamp-ch-closure heap-only-stamp-ch-sym
  by presburger

lemma heap-only-stamp-ch-ts-sym [sym]:
  assumes heap-only-stamp-changed-ts tsp h h'
  shows heap-only-stamp-changed-ts tsp h' h
  using assms heap-only-stamp-ch-sym heap-only-stamp-ch-terms-closure
  by presburger

lemma heap-only-stamp-ch-tr-trans [trans]:
  assumes heap-only-stamp-changed-tr tr h0 h1
  and heap-only-stamp-changed-tr tr h1 h2
  shows heap-only-stamp-changed-tr tr h0 h2
  by (metis (no-types, lifting) assms heap-only-stamp-ch-terms-closure heap-only-stamp-ch-tr-trans)

lemma heap-only-stamp-ch-ts-trans [trans]:
  assumes heap-only-stamp-changed-ts tsp h0 h1
  and heap-only-stamp-changed-ts tsp h1 h2
  shows heap-only-stamp-changed-ts tsp h0 h2
  by (metis (no-types, lifting) assms heap-only-stamp-ch-terms-closure
  heap-only-stamp-ch-ts-trans)

definition heap-only-nonterm-changed where
  heap-only-nonterm-changed h h' = (∀ typ x.
    heap.refs h typ x ≠ heap.refs h' typ x →
    (typ ≠ TYPEREP(i-term) ∧ typ ≠ TYPEREP(i-terms)))

lemma heap-only-nonterm-chI:
  fixes r :: 'a::heap ref
  assumes TYPEREP('a) ≠ TYPEREP(i-term) ∧ TYPEREP('a) ≠ TYPEREP(i-terms)
  shows heap-only-nonterm-changed h (Ref.set r v h)
  unfolding heap-only-nonterm-changed-def using assms
  by (simp add: Ref.set-def)

lemma heap-only-nonterm-ch-get:
fixes $r :: 'a :: heap ref$
assumes hosc: heap-only-nonterm-changed $h h'$
and $nt :: TYPEREP('a) = TYPEREP(i-term) \lor TYPEREP('a) = TYPE-REP(i-terms)$
shows $\text{Ref}.\text{get} h r = \text{Ref}.\text{get} h' r$
unfolding $\text{Ref}.\text{get-def comp-def}$
using hosc\[unfolded heap-only-nonterm-changed-def, rule-format,
 of $TYPEREP('a) addr-of-ref r\] $nt$ by fastforce

lemmas
heap-only-nonterm-ch-get-term =
  heap-only-nonterm-ch-get\[of - - tr, OF - refl[THEN disjI1]\] and
heap-only-nonterm-ch-get-terms =
  heap-only-nonterm-ch-get\[of - - tsr, OF - refl[THEN disjI2]\]
for $tr$ $tsr$

lemma heap-only-nonterm-ch-sym[sym]:
assumes heap-only-nonterm-changed $h h'$
shows heap-only-nonterm-changed $h' h$
using $\text{assms unfolding}$ heap-only-nonterm-changed-def
by (subst eq-sym-conv)

lemma
assumes heap-only-nonterm-changed $h h'$
shows heap-only-nonterm-ch-term-acyclic:
  $i\text{-term-acyclic} h tr \implies \text{i-term-acyclic} h' tr$
and heap-only-nonterm-ch-terms-acyclic:
  $i\text{-terms-acyclic} h tsp \implies \text{i-terms-acyclic} h' tsp$
unfolding conjunction-def
proof (atomize, unfold atomize-conj\[unfolded conjunction-def], goal-cases)
case 1
have $(\text{i-term-acyclic} h tr \implies \text{heap-only-nonterm-changed} h h' \implies \text{i-term-acyclic} h' tr) \land$
  $(\text{i-terms-acyclic} h tsp \implies \text{heap-only-nonterm-changed} h h' \implies \text{i-terms-acyclic} h' tsp)$
proof ((
  induction
  rule: i-term-acyclic-i-terms-acyclic.induct;
  intro impl),
  goal-cases nil link args terms-nil terms-next)
case (nil $h$)
then show $\text{?case}$
by (simp add: i-term-acyclic-i-terms-acyclic.t-acyclic-nil)
next
case (link $h$ $t$ $tref$ $s$)
have $\text{Ref}.\text{get} h' tref = \text{ITerm} (s, t, IVarD)$
using heap-only-nonterm-ch-get i-term-acyclic-i-terms-acyclic.t-acyclic-step-link
by (metis link(3) link(4))
then show ?case
  using link(2)[rule-format, OF link(4), THEN t-acyclic-step-link]
  by blast

next
  case (args h tsref tref s f)
  have Ref.get h' tref = ITerm (s, None, ITermD (f, tsref))
    using heap-only-nonterm-ch-get[OF args(4)] args(3) by metis
  then show ?case
    using args(2)[rule-format, OF args(4), THEN t-acyclic-step-ITerm]
    by blast

next
  case (terms-nil h)
  then show ?case
    by (simp add: ts-acyclic-nil)

next
  case (terms-next h ts2ref tref tsref)
  then have Ref.get h' tsref = ITerms (tref, ts2ref)
    using heap-only-nonterm-ch-get by metis
  then show ?case using terms-next ts-acyclic-step-ITerms by blast
qed

then show ?case
  using assms by fast
qed

lemma heap-only-nonterm-ch-sublists:
  assumes heap-only-nonterm-changed h h'
  shows i-terms-sublists h tsp = i-terms-sublists h' tsp
proof -
  { fix tsp' and
    h:: heap and
    h':: heap
    assume tsp' ∈ i-terms-sublists h tsp
    and heap-only-nonterm-changed h h'
    then have tsp' ∈ i-terms-sublists h' tsp
    proof (induction rule: i-terms-sublists.induct)
    case (next tsr' uu tnext)
    have Some tsr' ∈ i-terms-sublists h' tsp
      by (metis next.IH next.prems)
    then show ?case
      by (metis (no-types) heap-only-nonterm-ch-get-terms i-terms-sublists.next
        next.hyps(2) next.prems)
  next
    case self
    then show ?case
      using i-terms-sublists.self by auto
    qed
  }
then show ?thesis using assms assms[ symmetric] by blast

67
lemma heap-only-nonterm-ch-terms-set:  
assumes heap-only-nonterm-changed h h'  
shows i-terms-set h tsp = i-terms-set h' tsp  
unfolding i-terms-set-def2  
using assms heap-only-nonterm-ch-get TERMS heap-only-nonterm-ch-sublists by auto

lemma heap-only-nonterm-ch-closure:  
assumes heap-only-nonterm-changed h h'  
shows i-term-closure h tp = i-term-closure h' tp  
proof -  
{  
  fix tr  
  and h :: heap  
  and h' :: heap  
  assume tr ∈ i-term-closure h tp  
  and heap-only-nonterm-changed h h'  
  then have tr ∈ i-term-closure h' tp  
  proof (induction rule: i-term-closure.induct)  
  case (1 tr)  
  then show ?case  
  by (simp add: i-term-closure.intros(1))  
  next  
  case (2 tr uu is uv)  
  have tr ∈ i-term-closure h' tp  
  by (metis 2.IH 2.prems)  
  then show ?case  
  by (metis (no-types) 2.hyps(2) 2.prems  
      heap-only-nonterm-ch-get-term i-term-closure.intros(2))  
  next  
  case (3 tr uw ux tsp tr2)  
  show ?case  
  using 3.IH 3.hyps(2) 3.hyps(3) 3.prems heap-only-nonterm-ch-get-term  
      heap-only-nonterm-ch-terms-set i-term-closure.intros(3)  
  by fastforce  
  qed  
}  
then show ?thesis using assms assms[symmetric] by blast  
qed

lemma acyclic-closure-ch-stamp-inductc' [consumes 1,  
  case-names var link arys terms-nil terms]:  
fixes h:: heap  
  and tr:: i-term ref  
  and P1:: heap ⇒ i-term ref set ⇒ i-term ref ⇒ bool  
  and P2:: heap ⇒ i-term ref set ⇒ i-termsP ⇒ bool  
assumes acyclic: i-term-acyclic h (Some tr)
and var-case: $\bigwedge h\ trs\ tr\ s$.

\[
\text{Ref.get}\ h\ tr = \text{ITerm}(s, \text{None}, \text{IVarD}) \implies P1\ h\ trs\ tr
\]

and link-case: $\bigwedge h\ trs\ tr\ isr\ s$.

\[
(\bigwedge t2r\ h'\ trs'.
\begin{align*}
\text{trs} \subseteq \text{trs}' & \implies \text{heap-only-stamp-changed}\ \text{trs}'\ h' \implies P1\ h'\ trs'\ t2r) \implies \\
\text{Ref.get}\ h\ tr = \text{ITerm}(s, \text{Some}\ isr, \text{IVarD}) & \implies P1\ h\ trs\ tr
\end{align*}
\]

and args-case: $\bigwedge h\ trs\ tr\ tsp\ s\ f$.

\[
(\bigwedge h'\ trs'.
\begin{align*}
\text{trs} \subseteq \text{trs}' & \implies \text{heap-only-stamp-changed}\ \text{trs}'\ h' \implies P2\ h'\ trs'\ tsp) \implies \\
(\bigwedge h'\ trs'\ t2r0\ t2r.
\begin{align*}
\text{trs} \subseteq \text{trs}' & \implies \text{heap-only-stamp-changed}\ \text{trs}'\ h' \implies t2r \in \text{i-term-closure}\ h (\text{Some}\ t2r0) \implies \\
t2r0 & \in \text{i-terms-set}\ h\ tsp \implies P1\ h'\ trs'\ t2r) \implies \\
\text{Ref.get}\ h\ tr = \text{ITerm}(s, \text{None}, \text{ITermD}(f,\ tsp)) & \implies P1\ h\ trs\ tr
\end{align*}
\]

and terms-nil-case: $\bigwedge h\ trs.\ P2\ h\ trs\ \text{None}$

and terms-case: $\bigwedge h\ trs\ tr\ tsr\ tsnextp$.

\[
(\bigwedge h'\ trs'.
\begin{align*}
\text{trs} \subseteq \text{trs}' & \implies \text{heap-only-stamp-changed}\ \text{trs}'\ h' \implies P2\ h'\ trs'\ tsnextp) \implies \\
(\bigwedge h'\ trs'\ t2r.
\begin{align*}
\text{trs} \subseteq \text{trs}' & \implies \text{heap-only-stamp-changed}\ \text{trs}'\ h' \implies t2r \in \text{i-term-closure}\ h (\text{Some}\ tr) \implies \\
P1\ h'\ trs'\ t2r) \implies \\
\text{Ref.get}\ h\ tsr = \text{ITerms}\ (tr,\ tsnextp) & \implies P2\ h\ trs\ (\text{Some}\ tsr)
\end{align*}
\]

shows $P1\ h\ trs\ tr$

proof –

\[
\{
\text{fix tp}
\begin{align*}
\text{have i-term-acyclic}\ h\ tp & \implies \\
(\bigwedge tr\ h'\ trs'.
\begin{align*}
tr \in \text{i-term-closure}\ h\ tp & \implies \text{heap-only-stamp-changed}\ \text{trs}'\ h' \implies P1\ h'\ trs'\ tr\ \land \\
(\forall s\ f\ tsp0\ tsp.
\begin{align*}
\text{Ref.get}\ h\ tr = \text{ITerm}(s, \text{None}, \text{ITermD}(f,\ tsp0)) & \implies \\
tsp \in \text{i-terms-sublists}\ h\ tsp0 & \implies
\end{align*}
\end{align*}
\}
\end{align*}
\]
\]
proof (induction taking: λh tsp. (∀ trs’ h’.

heap-only-stamp-changed trs’ h h’ →

(∀ tsp’.

tsp’ ∈ i-terms-closure-sablists h tsp →
P2 h’ trs’ tsp’) ∧

(∀ tr.

tr ∈ i-terms-closure h tsp →
P1 h’ trs’ tr))

rule: i-term-acyclic-i-terms-acyclic.inducts(1))

case (t-acyclic-nil uu)

then show ?case

by (simp add: i-term-closure-None)

next
case (t-acyclic-step-link h t tref s tr h’ trs)

consider (a) t = None |

(b) tr ∈ i-term-closure h t |

(c) isr where t = Some isr and tr = tref

using i-term-closure-link t-acyclic-step-link.hyps(2)

t-acyclic-step-link.prems(1) by blast

then show ?case

proof (cases)

case (a)

then show ?thesis

using heap-only-stamp-ch-get-term i-term-closure-var t-acyclic-step-link.hyps(2)

t-acyclic-step-link.prems(1) t-acyclic-step-link.prems(2) var-case by fastforce

next
case (b)

then show ?thesis using t-acyclic-step-link by blast

next
case (c)

have ∃t2r h’a trs'. trs' ⊆ trs' ⊃ heap-only-stamp-changed trs' h' h'a →
t2r ∈ i-term-closure h'(Some isr) ⊃ P1 h’a trs’ t2r

proof —

fix t2r h’a trs'

assume trs-subset-trs': trs' ⊆ trs'

and hosch-h’-h’a: heap-only-stamp-changed trs’ h’ h’a

and tr2-clos’-isr: t2r ∈ i-term-closure h’ (Some isr)

have ∗: t2r ∈ i-term-closure h t

using c(1) heap-only-stamp-ch-closure

t-acyclic-step-link.prems(2) tr2-clos’-isr by blast

have heap-only-stamp-changed trs’ h h'

using t-acyclic-step-link(5) trs-subset-trs'

heap-only-stamp-ch-antimono by blast

then have ∗∗: heap-only-stamp-changed trs’ h h’a

using hosch-h’-h’a heap-only-stamp-ch-trans by blast

show P1 h’a trs’ t2r using t-acyclic-step-link.IH[OF ∗ ∗]
by simp

qed

moreover obtain \( s' \) where \( \text{Ref.get } h' \text{ tr } = \text{ITerm} (s', \text{Some } \text{isr}, \text{IVarD}) \)

using heap-only-stamp-ch-get-term[OF t-acyclic-step-link(5) t-acyclic-step-link(2)]

c by blast

ultimately have \( P1 \, h' \, \text{trs tr} \) using link-case by meson

then show \( \text{thesis} \)

using \( c(2) \) t-acyclic-step-link.hyps(2) by auto

qed

next

case (t-acyclic-step-ITerm h tsref tref s f tr h' \( \text{trs} \))

then have get-tref: \( \text{Ref.get } h \text{ tref } = \text{ITerm} (s, \text{None}, \text{ITermD} (f, \text{tsref})) \)

and tr-clos-tref: \( tr \in \text{i-term-closure h (Some tref)} \)

and hose-h-h': heap-only-stamp-changed \( \text{trs} \) \( h' \)

and \( IH1: \bigwedge \text{trs}' \, h' \, \text{tsp}' \).

heap-only-stamp-changed \( \text{trs}' \) \( h' \) \( \text{h' } \)

\( \text{tsp}' \in \text{i-terms-closure-sublists h tsref } \Rightarrow \)

\( P2 \, h' \, \text{trs}' \, \text{tsp'} \)

and \( IH2: \bigwedge \text{trs}' \, h' \, \text{tr} \).

heap-only-stamp-changed \( \text{trs}' \) \( h' \) \( \text{h' } \)

\( tr \in \text{i-terms-closure h tsref } \Rightarrow \)

\( P1 \, h' \, \text{trs}' \, \text{tr} \) by blast+

have tr-clos-tref: \( tr \in \text{i-term-closure h' (Some tref)} \)

using hose-h-h': heap-only-stamp-ch-closure tr-clos-tref by auto

have *: \( \bigwedge h'' \, \text{trs}'' \, \text{trs}'' \subseteq \text{trs}' \Rightarrow \text{heap-only-stamp-changed \( \text{trs}' \) \( h' \) \( h'' \)} \Rightarrow \)

\( P2 \, h'' \, \text{trs}'' \) tsref

proof –

fix \( h'' \, \text{trs}' \)

assume \( \text{trs}'' \subseteq \text{trs}' \)

and heap-only-stamp-changed \( \text{trs}' \) \( h'' \)

then have heap-only-stamp-changed \( \text{trs}' \) \( h'' \)

using heap-only-stamp-ch-antimono heap-only-stamp-ch-trans

\( \text{t-acyclic-step-ITerm.prems(2)} \) by blast

then show \( P2 \, h'' \, \text{trs}'' \) tsref

using \( IH1 \) i-terms-sublists.self by fast

qed

have **: \( \bigwedge h'' \, \text{trs}'' \, \text{t}2r0 \, \text{t}2r. \)

\( \text{trs}'' \subseteq \text{trs}' \Rightarrow \)

heap-only-stamp-changed \( \text{trs}' \) \( h' \) \( h'' \)

\( \text{t}2r \in \text{i-term-closure h' (Some t}2r0) \Rightarrow \)

\( \text{t}2r0 \in \text{i-terms-set h' tsref } \Rightarrow \)

\( P1 \, h'' \, \text{trs}' \, \text{t}2r \)

proof –

fix \( h'' \, \text{trs}' \, \text{t}2r0 \, \text{t}2r \)

assume \( \text{trs}'' \subseteq \text{trs}' \)

and hose-h-h''': heap-only-stamp-changed \( \text{trs}' \) \( h' \) \( h'' \)

and \( \text{t}2r\text{-clos}'-\text{t}2r0: \text{t}2r \in \text{i-term-closure h' (Some t}2r0) \)

and \( \text{t}2r0\text{-terms}: \text{t}2r0 \in \text{i-terms-set h' tsref} \)

then have hose-h-h''': heap-only-stamp-ch-changed \( \text{trs}' \) \( h' \) \( h'' \)

using heap-only-stamp-ch-antimono heap-only-stamp-ch-trans
\textit{t-acyclic-step-ITerm.prems(2)} by blast

have \(t2\mathit{r0-terms-set-tsref}: t2\mathit{r0} \in \mathit{i-terms-set \ h \ tsref}\)
using \(t2\mathit{r0-terms'} \mathit{hosc-h-h}'[\mathit{symmetric}] \mathit{heap-only-stamp-ch-terms-set}\) by blast

have \(t2\mathit{r-clos-tsref}: t2\mathit{r} \in \mathit{i-terms-closure \ h \ tsref}\)
using \(UN-I \ t2\mathit{r-clos'-t2\mathit{r0} \ t2\mathit{r0-terms-set-tsref}}\)
heap-only-stamp-ch-closure \(\mathit{hosc-h-h}'\) by fast
then show \(P1 \ h'' \ trs' \ t2\mathit{r} \ \text{using} \ IH2[OF \ \mathit{hosc-h-h}'' \ t2\mathit{r-clos-tsref}]\) by blast
qed

consider \((a) \ tr \in \mathit{i-terms-closure \ h \ tsref} \ | \ (b) \ tr = \mathit{tref}\)
using \(get-tref \ \mathit{i-term-closure-args} \ tr-clos-tref\) by fastforce
then show \(?case\)
proof (cases)
  case \(a\)
  then have \(t1: P1 \ h' \ trs \ tr\) using \(IH2 \ \mathit{hosc-h-h}'\) by blast
  show \(?thesis\)
  proof (intro conjI, simp add: \(t1\), intro allI impI)
    fix \(s \ f \ tsp \ tsp0\)
    assume \(get-tr: \mathit{Ref.get \ h \ tr = ITerm} \ (s, \ \mathit{None}, \ \mathit{ITermD} \ (f, \ tsp0))\)
    and \(tsp-sublist-tsp0: tsp \in \mathit{i-terms-sublists \ h \ tsp0}\)
    have \(tsp \in (\bigcup tr \in \mathit{i-terms-closure \ h \ tsref} \ \mathit{i-term-sublists} \ h \ tr)\)
    by (metis (no-types) \mathit{UN-iff \ a \ get-ITerm-args-iff-ex \ get-tr} \ tsp-sublist-tsp0)
    then have \(tsp \in \mathit{i-terms-closure-sublists} \ h \ tsref\)
    by blast
    then show \(P2 \ h' \ trs \ tsp\) using \(IH1 \ \mathit{hosc-h-h}'\) by presburger
  qed
next
  case \(b\)
  then obtain \(s' \ where \ 
  \text{Ref.get} \ h' \ tr = \mathit{ITerm} \ (s', \ \mathit{None}, \ \mathit{ITermD} \ (f, \ tsp0)) \)
  using \(get-tref \ \mathit{heap-only-stamp-ch-get-term} \ \mathit{hosc-h-h}'\) by blast
from \(* \ \text{ars-case[OF - - this]}\)
  have \(t1: P1 \ h' \ trs \ tr\)
  by force
  then show \(?thesis\)
  proof (intro conjI, simp add: \(t1\), intro allI impI)
    fix \(s \ f \ tsp \ tsp0\)
    assume \(get-tr: \mathit{Ref.get} \ h \ tr = \mathit{ITerm} \ (s, \ \mathit{None}, \ \mathit{ITermD} \ (f, \ tsp0))\)
    and \(tsp-sublist-tsp0: tsp \in \mathit{i-terms-sublists} \ h \ tsp0\)
    then have \(tsp \in \mathit{i-terms-closure-sublists} \ h \ tsref\)
    using \(get-tref \ b\) by fastforce
    then show \(P2 \ h' \ trs \ tsp\)
    using \(IH1 \ \mathit{hosc-h-h}'\) by blast
  qed
  qed
next
  case \((ts-acyclic-nil \ uy)\)
  then show \(?case\)
  using \(\text{terms-nil-case}\)
by (simp add: i-terms-set-None-empty i-terms-sublists-None-om)

next
case (ts-acyclic-step-ITerms h ts2ref tref tsref)
then have IH1a: \( \forall \; tr \; ts' \; h' \). 
  \( tr \in \text{i-terms-closure} \; h \; ts2ref \Rightarrow \) 
  \( \text{heap-only-stamp-changed} \; ts' \; h \; h' \Rightarrow \) 
  \( P_1 \; h' \; ts' \; tr \)
and IH1b: \( \forall \; tr \; ts' \; tsp' \; h' \). 
  \( tsp' \in \text{i-terms-closure-sublists} \; h \; ts2ref \Rightarrow \) 
  \( \text{heap-only-stamp-changed} \; ts' \; h \; h' \Rightarrow \) 
  \( P_2 \; h' \; ts' \; tsp' \)
and get-tsref: \( \text{Ref} \cdot \text{get} \; h \; tsref = \text{ITerms} \; (\text{tref}, \; ts2ref) \)
and tref-acyclic: \( \text{i-term-acyclic} \; h \; (\text{Some} \; \text{tref}) \)
by blast+

have IH2a: \( \forall \; tr \; ts' \; tr \; h' \). 
  \( tr \in \text{i-term-closure} \; h \; (\text{Some} \; \text{tref}) \Rightarrow \) 
  \( \text{heap-only-stamp-changed} \; ts' \; h \; h' \Rightarrow \) 
  \( P_1 \; h' \; ts' \; tr \)
and IH2b: \( \forall \; tr \; ts' \; tr \; h' \; s \; f \; tsp0 \; tsp. \; tr \in \text{i-term-closure} \; h \; (\text{Some} \; \text{tref}) \Rightarrow \) 
  \( \text{heap-only-stamp-changed} \; ts' \; h \; h' \Rightarrow \) 
  \( \text{Ref} \cdot \text{get} \; h \; tr = \text{ITerm} \; (s, \; \text{None}, \; \text{ITermD} \; (f, \; tsp0)) \Rightarrow \) 
  \( tsp \in \text{i-terms-sublists} \; h \; tsp0 \Rightarrow \) 
  \( P_2 \; h' \; ts' \; tsp \)
by (simp add: ts-acyclic-step-ITerms.IH)+

show ?case
proof (intro allI impI conjI, goal-cases terms term)
case (term trs' tr h)
then have hosc-h-h': heap-only-stamp-changed trs' h h'
  and tr-clos-tsref: \( tr \in \text{i-terms-closure} \; h \; (\text{Some} \; \text{tsref}) \)
  by blast+
consider (a) \( tr \in \text{i-terms-closure} \; h \; ts2ref \mid \)
  (b) \( tr \in \text{i-terms-closure} \; h \; (\text{Some} \; \text{tref}) \)
  using get-tsref
  by (metis tr-clos-tsref UnE i-terms-closure-terms)
then show ?case
proof (cases)
case a
  then show \( ?\text{thesis} \) using IH1a hosc-h-h' by presburger
next
case b
  then show \( ?\text{thesis} \) using IH2a hosc-h-h' by presburger
qed
next
case (terms trs' h' tsp')
then have tsp'-csl-tsref: tsp' \( \in \text{i-terms-closure-sublists} \; h \; (\text{Some} \; \text{tsref}) \)
  and hosc-h-h': heap-only-stamp-changed trs' h h'
  by blast+
have get'-tsref: \( \text{Ref} \cdot \text{get} \; h' \; tsref = \text{ITerms} \; (\text{tref}, \; ts2ref) \)

using get-tsref hosec-h-h' heap-only-stamp-ch-get-terms by simp

consider (a) tsp' = None |
(b) tsr'

where tsp' = Some tsr'
  and tsp' ∈ i-term-closure-subsists h (Some tref) |
(c) tsr'

where tsp' = Some tsr'
  and tsp' ∈ i-terms-closure-subsists h ts2ref |
(d) tsp' = Some tsref

using i-term-closure-subsists-terms[OF get-tsref]
tsp'-clsl-tsref by (atomize-elim, force)

then show ?case
proof (cases)
  case a
  then show ?thesis by (simp add: terms-nil-case)

next
  case b
  then obtain tr where tr-clos-tref: tr ∈ i-term-closure h (Some tref)
    and tsr'-sublist-tr: Some tsr' ∈ i-term-sublists h tr
    by blast

  have i-term-acyclic h (Some tr)

  using i-term-closure-acyclic tr-clos-tref tref-acyclic by blast

  with tsr'-sublist-tr obtain s f tsp0

  where get-tr: Ref.get h tr = ITerm (s, None, ITermD (f, tsp0))
      and tsr'-sublist-tsp0: Some tsr' ∈ i-terms-sublists h tsp0

  using i-terms-sublists-someE acyclic-args-nil-is by auto

  show ?thesis using IH2b(OF tr-clos-tref hosec-h-h' get-tr tsr'-sublist-tsp0]

  by fast

next
  case c
  then show ?thesis using IH1b hosec-h-h' by presburger

next
  case d

  have *: ∃ h'' trs''.
    trs' ⊆ trs'' ⇒
    heap-only-stamp-changed trs'' h' h'' ⇒
    P2 h'' trs'' ts2ref

  by (meson IH1b UnCI heap-only-stamp-ch-antimono heap-only-stamp-ch-trans
       hosec-h-h' i-terms-sublists.self)

  have **: ∃ h'' trs'' t2r.
    trs' ⊆ trs'' ⇒
    heap-only-stamp-changed trs'' h' h'' ⇒
    t2r ∈ i-term-closure h' (Some tref) ⇒
    P1 h'' trs'' t2r

  proof –
    fix h'' trs'' t2r
    assume trs'-subset-trs'': trs' ⊆ trs''
and hosc-trs″-h′-h″: heap-only-stamp-changed trs″ h′ h″
and t2r-clos′-tref: t2r ∈ i-term-closure h′ (Some tref)

have t2r ∈ i-term-closure h (Some tref)
  using heap-only-stamp-ch-closure hosc-h-h t2r-clos′-tref by blast
moreover have heap-only-stamp-changed trs″ h′ h″
  by (metis heap-only-stamp-ch-antimono heap-only-stamp-ch-trs
hosc-h-h'
  hosc-trs″-h′-h″ trs′-subset-trs″)
ultimately show P1 h″ trs″ t2r
  using IH2a[where h'=h″ and tra=t2r and trs'=trs″]
  by blast
qed
from terms-case[where h=h′ and trs=trs′ and tsr=tsref, OF - - get'-tsref]
  show ?thesis using d * * by force
qed
qed
}
then show ?thesis using acyclic
  heap-only-stamp-ch-refl i-term-closure.intros(1) by auto
qed
end

A.4 Imperative version of algorithm

theory Unification-Imperative
imports Main
  ITerm
  HOL‐Imperative-HOL.
  HOL‐Imperative-HOL.Heap-Monad
begin

fun i-union where
i-union (Some v, t:: i-termP) = (v := ITerm (0, t, IVarD)) |
i-union (None, -) = return ()

partial-function (heap) i-find:: i-termP ⇒ i-termP Heap
  where [code]:
i-find tp = (case tp of
    (Some tr) ⇒ do |
      t ← !tr;
      case t of
        ITerm (-, Some is, -) ⇒ i-find (Some is)
      | ITerm (-, None, -) ⇒ return (Some tr)
      | None ⇒ return None)

context
fixes \textit{time}:: \texttt{nat ref} \\
\textbf{and} \textit{v}:: \texttt{i-termP} \\
begin

\textbf{partial-function} (heap) \textit{i-occ-p}:: \texttt{i-termP + i-termsP} ⇒ \texttt{bool Heap} \hspace{1em} \textbf{where} [\textit{code}]: \\
\textit{i-occ-p} \hspace{1em} \textit{XX} = ( \\
\hspace{1em} \textbf{case} \hspace{1em} \textit{XX} \hspace{1em} \textbf{of} \hspace{1em} \\
\hspace{2em} \texttt{(Inl (\textit{Some \hspace{1em} t}) \hspace{1em})} \hspace{1em} ⇒ \hspace{1em} \textbf{do} \hspace{1em} \\
\hspace{3em} \textit{tv} \leftarrow \texttt{!t}; \\
\hspace{3em} \textbf{case} \hspace{1em} \textit{tv} \hspace{1em} \textbf{of} \hspace{1em} \\
\hspace{4em} \texttt{ITerm (\_ \hspace{1em}, \_ \hspace{1em}, \_VarD)} \hspace{1em} ⇒ \hspace{1em} \textbf{return} \hspace{1em} (\textit{v} \hspace{1em} = \texttt{Some \hspace{1em} t}) \\
\hspace{4em} | \hspace{1em} \texttt{ITerm (\textit{stamp} \hspace{1em}, \_None \hspace{1em}, \_TermD (\_f \hspace{1em}, \_args) \hspace{1em})} \hspace{1em} ⇒ \hspace{1em} \textbf{do} \hspace{1em} \\
\hspace{5em} \textit{timev} \leftarrow \texttt{!time}; \hspace{1em} \\
\hspace{5em} \textbf{if} \hspace{1em} (\textit{stamp} \hspace{1em} = \textit{timev}) \hspace{1em} \textbf{then} \hspace{1em} \textbf{return} \hspace{1em} \texttt{False} \hspace{1em} \\
\hspace{5em} \textbf{else} \hspace{1em} \textbf{do} \hspace{1em} \\
\hspace{6em} \textit{t} ::= \texttt{ITerm(\textit{timev} \hspace{1em}, \_None \hspace{1em}, \_TermD (\_f \hspace{1em}, \_args))}; \hspace{1em} \\
\hspace{6em} \textit{i-occ-p} \hspace{1em} \texttt{(Inr \hspace{1em} \_args)} \\
\hspace{4em} \} \\
\hspace{2em} | \hspace{1em} \texttt{(Inr \hspace{1em} \_None)} \hspace{1em} ⇒ \hspace{1em} \textbf{return} \hspace{1em} \texttt{False} \\
\hspace{2em} | \hspace{1em} \texttt{(Inr (\textit{Some \hspace{1em} ts}) \hspace{1em})} \hspace{1em} ⇒ \hspace{1em} \textbf{do} \hspace{1em} \\
\hspace{3em} \textit{tsv} \leftarrow \texttt{!ts}; \\
\hspace{3em} \textbf{case} \hspace{1em} \textit{tsv} \hspace{1em} \textbf{of} \hspace{1em} \\
\hspace{4em} \texttt{ITerms (\_t \hspace{1em}, \_next)} \hspace{1em} ⇒ \hspace{1em} \textbf{do} \hspace{1em} \\
\hspace{5em} \textbf{find-res} \leftarrow \textbf{i-find} \hspace{1em} (\texttt{Some \hspace{1em} t}); \\
\hspace{5em} \textbf{occ-res} \leftarrow \textit{i-occ-p} \hspace{1em} \texttt{(Inl \hspace{1em} \textbf{find-res})}; \\
\hspace{5em} \textbf{if} \hspace{1em} \textbf{occ-res} \hspace{1em} \textbf{then} \hspace{1em} \textbf{return} \hspace{1em} \texttt{True} \hspace{1em} \\
\hspace{5em} \textbf{else} \hspace{1em} \textit{i-occ-p} \hspace{1em} \texttt{(Inr \hspace{1em} \_next)} \hspace{1em} \} \\
\} \\
\textbf{definition} \textit{i-occurs}:: \texttt{i-termP} ⇒ \texttt{bool Heap} \hspace{1em} \textbf{where} \\
i-occurs \hspace{1em} \textit{t} = \hspace{1em} \textbf{do} \hspace{1em} \\
\hspace{1em} \textit{timev} \leftarrow \texttt{!time}; \hspace{1em} \\
\hspace{1em} \texttt{time := timev} \hspace{1em} + \hspace{1em} \texttt{1}; \hspace{1em} \\
\hspace{1em} \textit{i-occ-p} \hspace{1em} \texttt{(Inl \hspace{1em} \_t)} \\
\) \\
\end

\textbf{end}

A.5 Equivalence of imperative and functional formulation

theory \textit{ImpEqFunc} \\
\textbf{imports} \textit{Main} \\
\hspace{1em} \textit{Unification-Functional} \\
\hspace{1em} \textit{Unification-Imperative}

76
HOL–Imperative-HOL. Ref
HOL–Imperative-HOL. Heap-Monad

begin

Variables are called \((x, h)\) where \(h\) is the heap address of the variable term.

**partial-function** (heap)
\[\text{i-term-to-term-p :: i-term ref + i-termsP} \Rightarrow (\text{term + term list}) \text{Heap}\]

**where** \[\text{code}]:
\[\text{i-term-to-term-p XX = (case XX of}\]
\[\text{(Inl tr) ⇒ do}\]
\[\text{t ← !tr; }\]
\[\text{case t of}\]
\[\text{ITerm (., None, IVarD) ⇒ return (Inl(V("x", int (addr-of-ref tr))))}\]
\[\text{ITerm (., (Some \text{t2p}), -) ⇒ i-term-to-term-p (Inl \text{t2p})}\]
\[\text{ITerm (., None, \text{ITermD}(f, \text{termsp})) ⇒ do}\]
\[\text{v ← i-term-to-term-p (Inr \text{termsp}); }\]
\[\text{case v of}\]
\[\text{Inr (terms) ⇒ return (Inl(T(f, terms)))}\]
\[\text{| (Inr None) ⇒ return (Inr([]))}\]
\[\text{| (Inr (Some \text{termsr})} \Rightarrow do}\]
\[\text{termsv ← !termsr; }\]
\[\text{case termsv of}\]
\[\text{(ITerms(tthis, tnext)) ⇒ do}\]
\[\text{v\text{tthis} ← i-term-to-term-p (Inl \text{tthis}); }\]
\[\text{v\text{tnext} ← i-term-to-term-p (Inr \text{tnext}); }\]
\[\text{case (v\text{tthis}, v\text{tnext}) of}\]
\[\text{(Inl(term), Inr(terms)) ⇒ return (Inr(term\#terms))}\]
\[\text{| )}\]

**lemma** \[\text{i-term-to-term-p-mr}:\]
\[\text{fixes h :: heap}\]
\[\text{and XX :: i-term ref + i-termsP}\]
\[\text{assumes term-acyclic: \(\forall tr. XX = \text{Inl tr} \Rightarrow \text{i-term-acyclic h (Some tr)}\) and}\]
\[\text{terms-acyclic: \(\forall tp. XX = \text{Inr tp} \Rightarrow \text{i-terms-acyclic h tp}\)\]
\[\text{shows} \exists r. (\text{Some(r, h)} = \text{execute (i-term-to-term-p XX)} h \land \text{isl r = isl XX})\]

**proof** –
\[\text{fix tp trp}\]
\[\text{let \?cond \text{XX0} h0 = \exists r. (\text{Some(r, h)} = \text{execute (i-term-to-term-p XX0)} h \land \text{isl r = isl XX0})}\]
\[\text{have (i-term-acyclic h trp \Rightarrow}\]
\[\text{trp \neq \text{None} \Rightarrow \?cond (\text{Inl (case trp of Some tr ⇒ tr)}) h \land}\]
\[\text{(i-terms-acyclic h tp \Rightarrow \?cond (Inr tp) h)}\]

**proof** (induction rule: i-term-acyclic-i-terms-acyclic.induct)

**case** (t-acyclic-nil h)
then show \( \text{?case by simp} \)

next

case (\( t\text{-acyclic-step-link} \ h \ t \ tref \ stamp \))
then consider (a) \( \text{Ref.get} \ h \ tref = \text{ITerm(stamp, None, IVarD)} \) | (b) \( tn \text{ iv where } \text{Ref.get} \ h \ tref = \text{ITerm(stamp, (Some } tn), iv} \)
by auto
then show \( \text{?case using } t\text{-acyclic-step-link.IH} \)

proof (cases)

case a
then show \( \text{?thesis} \)
by (subst \( \text{i-term-to-term-p.simps} \), simp add: \( \text{lookup-def tap-def bind-def return-def} \) \( \text{execute-heap isl-def} \))

next

case b
then show \( \text{?thesis using } t\text{-acyclic-step-link.IH} \)
by (subst \( \text{i-term-to-term-p.simps} \), simp add: \( \text{lookup-def tap-def bind-def return-def} \) \( \text{execute-heap t\text{-acyclic-step-link.hyps(2)} } \))

qed

next

case (\( t\text{-acyclic-step-ITerm} \ h \ tsref \ tref \ stamp \ f \))
then obtain \( r0 \) where \( r0\text{-def} \): \( \text{Some } (r0, h) = \text{execute } (\text{i-term-to-term-p} \ (\text{Inr } tsref)) \ h \land \neg \text{isl } r0 \)
by auto
then have \( \ast \): \( \text{Some } (r0, h) = \text{execute } (\text{i-term-to-term-p} \ (\text{Inr } tsref)) \ h \) by simp
obtain \( r0v \) where \( \ast \ast \): \( \text{Inr } r0v = r0 \)
using \( r0\text{-def} \) \( \text{sum.collapse(2) by blast} \)

show \( \text{?case using } t\text{-acyclic-step-ITerm} \)
apply (subst \( \text{i-term-to-term-p.simps} \), simp add: \( \text{lookup-def tap-def bind-def return-def} \) \( \text{execute-heap} \) \( \text{t\text{-acyclic-step-link.hyps(2)} } \))
apply (fold \( \ast \ast \))
by (simp add: \( \text{execute-heap} \) )

next

case (\( t\text{-acyclic-nil} \ h \))
then show \( \text{?case by } (\text{subst i-term-to-term-p.simps, simp add: return-def execute-heap}) \)

next

case (\( t\text{-acyclic-step-ITerms} \ h \ ts2ref \ tref \ tsref \))
then obtain \( r0 \) where \( r0\text{-def} \): \( \text{Some } (r0, h) = \text{execute } (\text{i-term-to-term-p} \ (\text{Inl } tref)) \ h \land \text{isl } r0 \)
by auto
then have \( a1 \): \( \text{Some } (r0, h) = \text{execute } (\text{i-term-to-term-p} \ (\text{Inl } tref)) \ h \) by simp
obtain \( r0v \) where \( a2 \): \( \text{Inl } r0v = r0 \) using \( r0\text{-def[unfolded isl-def]} \)
by auto
obtain \( r_1 \) where \( r_1\text{-def: } Some(r_1, h) = \text{execute}(\text{i-term-to-term-p}(\text{Inr}\ ts_2\text{ref}))) h \land \neg\text{isl} r_1 \)

using \text{ts-acyclic-step-ITerms} by auto
then have \( b_1: Some(r_1, h) = \text{execute}(\text{i-term-to-term-p}(\text{Inr}\ ts_2\text{ref}))) h \) by simp
obtain \( r_1v \) where \( b_2: \text{Inr} r_1v = r_1 \)
using \( r_1\text{-def} \) sum.collapse(2) by blast

from \text{ts-acyclic-step-ITerms} show \( \text{case} \)
apply (\text{subst} \text{i-term-to-term-p}.\text{simps},
\text{simp} add: \text{lookup-def} \text{tap-def} \text{bind-def} \text{return-def} \text{execute-heap})
by (fold a1 a2, simp, fold b1 b2, simp add: \text{return-def} \text{execute-heap})
qed


definition \text{i-term-to-term:: i-term ref \Rightarrow term Heap where}
\text{i-term-to-term tr = do \{ r \leftarrow \text{i-term-to-term-p}(\text{Inl} tr); case r of (\text{Inl} v) \Rightarrow \text{return} v \}} \}

abbreviation \text{i-term-to-term-e:: heap \Rightarrow i-term ref \Rightarrow term where}
\text{i-term-to-term-e h tr \equiv (case (execute(\text{i-term-to-term tr} h) of Some(r, -) \Rightarrow r)}

lemma \text{i-term-to-term-value-iff:}
fixes \( tr:: \text{i-term ref} \)
and \( r:: \text{term} \)
and \( h:: \text{heap} \)
assumes \text{i-term-acyclic h (Some tr)}
shows \((r = \text{i-term-to-term-e h tr}) = (\text{Some}(r, h) = \text{execute}(\text{i-term-to-term tr}) h))\)
proof –
\{
obtain \( XX \) where \(*: Some(XX, h) = \text{execute}(\text{i-term-to-term-p}(\text{Inl}\ tr))) h \)
and \( isi \ XX \)
using \text{i-term-to-term-p-mp} \text{assms} isi-def by fast
then obtain \( r' \) where \(**: \text{Inl} r' = XX \)
using isi-def by metis
assume \( r = \text{i-term-to-term-e} \ h \ tr \)
then have \( \text{Some}(r, h) = \text{execute} (\text{i-term-to-term} \ tr) \ h \)
by (simp add: i-term-to-term-def bind-def, fold * **, simp add: return-def execute-heap)
\}
then show \#thesis
by (metis case-prod-conv option.simps(5))
qed

lemma i-term-to-term-value:
fixes \( tr :: \text{i-term ref} \)
and \( h :: \text{heap} \)
assumes \( \text{i-term-acyclic} \ h \ (\text{Some} \ tr) \)
shows \( \text{execute} (\text{i-term-to-term} \ tr) \ h = \text{Some}(\text{i-term-to-term-e} \ h \ tr, h) \)
using assms i-term-to-term-value-iff by metis

definition i-terms-to-terms :: \( \text{i-termsP} \Rightarrow \text{term list Heap} \)
where
\( \text{i-terms-to-terms} \ tp = \text{do} \{ \ r \leftarrow \text{i-term-to-term-p} (\text{Inr} \ tp); \ \text{case} \ r \ of \ (\text{Inr} \ v) \Rightarrow \text{return} \ v \} \)

abbreviation i-terms-to-terms-e :: \( \text{heap} \Rightarrow \text{i-termsP} \Rightarrow \text{term list} \)
where
\( \text{i-terms-to-terms-e} \ h \ tr \equiv (\text{case} (\text{execute} (\text{i-terms-to-terms} \ tr) \ h) \ of \ \text{Some}(r, -) \Rightarrow r) \)

lemma i-terms-to-terms-value-iff:
fixes \( tsp :: \text{i-termsP} \)
and \( r :: \text{term list} \)
and \( h :: \text{heap} \)
assumes \( \text{i-terms-acyclic} \ h \ tsp \)
shows \( (r = \text{i-terms-to-terms-e} \ h \ tsp) = (\text{Some}(r, h) = \text{execute} (\text{i-terms-to-terms} \ tsp) \ h) \)
proof −
{ 
  obtain \( XX \) where \( \ast :: \text{Some}(XX, h) = \text{execute} (\text{i-term-to-term-p} (\text{Inr} \ tsp)) \ h \)
and \( \neg\text{isl} \ XX \)
  using i-term-to-term-p-mr assms isl-def by fast
  then obtain \( r' \) where \( \ast :: \text{Inr} \ r' = XX \)
  using sum.collapse(2) by blast
  assume \( r = \text{i-terms-to-terms-e} \ h \ tsp \)
  then have \( \text{Some}(r, h) = \text{execute} (\text{i-terms-to-terms} \ tsp) \ h \)
  by (simp add: i-terms-to-terms-def bind-def, fold * **, simp add: return-def execute-heap)
  
  } then show \#thesis
  by (metis case-prod-conv option.simps(5))
qed

lemma i-terms-to-terms-value:
fixes \( tsp :: \text{i-termsP} \)

80
and $h$:: heap
assumes $i$-terms-acyclic $h$ tsp
shows execute ($i$-terms-to-terms tsp) $h$ = Some ($i$-terms-to-terms-e $h$ tsp, $h$)
by (metis assms $i$-terms-to-terms-value-iff)

lemma $i$-term-to-term-var-none:
fixes $tr$:: $i$-term ref
   and $s$:: nat
   and $h$:: heap
assumes Ref.get $h$ $tr$ = ITerm($s$, None, IVarD)
shows execute ($i$-term-to-term $tr$) $h$ = Some (($V"x"$, int (addr-of-ref $tr$)), $h$)
unfolding $i$-term-to-term-def
by (subst $i$-term-to-term-p.simps,
    simp add: assms lookup-def tap-def bind-def return-def execute-heap)

lemma $i$-term-to-term-var-some:
fixes $tr$:: $i$-term ref
   and $t2p$:: $i$-term ref
   and $s$:: nat
   and $h$:: heap
assumes Ref.get $h$ $tr$ = ITerm($s$, Some $t2p$, IVarD)
shows execute ($i$-term-to-term $tr$) $h$ = execute ($i$-term-to-term $t2p$) $h$
unfolding $i$-term-to-term-def
by (subst $i$-term-to-term-p.simps,
    simp add: assms lookup-def tap-def bind-def return-def execute-heap)

lemma $i$-term-to-term-terms:
fixes $tr$:: $i$-term ref
   and $termsp$ and $terms$
   and $s$:: nat
   and $h$:: heap
assumes acyclic: $i$-term-acyclic $h$ (Some $tr$)
   and get-tr: Ref.get $h$ $tr$ = ITerm ($s$, None, ITermD($f$, $termsp$))
   and $termsp$-res: execute ($i$-terms-to-terms $termsp$) $h$ = Some ($terms$, $h$)
shows execute ($i$-term-to-term $tr$) $h$ = Some ($T$($f$, $terms$), $h$)
proof –
  have $i$-terms-acyclic $h$ $termsp$ using acyclic get-tr by (fact $acyclic$-terms-term-simp)
  then obtain $r$ where $r$-def: Some($r$, $h$) = execute ($i$-term-to-term-p (Inr $termsp$))
  $h$ ∧ ¬isl $r$
  using $i$-term-to-term-p-mr[where XX=Inr $termsp$] by auto
  then have *: Some($r$, $h$) = execute ($i$-term-to-term-p (Inr $termsp$)) $h$ by simp
  obtain $rv$ where **: Inr $rv$ = $r$
  using $r$-def sum.collapse(2) by fast
  have ***: Some(Inr $rv$, $h$) = Some(Inr terms, $h$)
    using * ** $termsp$-res[unfolded $i$-terms-to-terms-def]
    by (simp add: bind-def return-def execute-heap)
    (fold ***, simp add: execute-heap return-def)
  show ?thesis unfolding $i$-term-to-term-def
apply (subst i-term-to-term-p.simps)
apply (simp add: get-tr lookup-def tap-def bind-def return-def execute-heap)
by (fold * **, simp add: execute-heap return-def ***)
qed

lemma i-term-to-term-e-terms:
  fixes tr :: i-term ref
  and termsp
  and s :: nat
  and h :: heap
  assumes acyclic: i-term-acyclic h (Some tr)
  and get-tr: Ref.get h tr = ITerm (s, None, ITermD(f, termsp))
  shows i-term-to-term-e h tr = T(f, i-terms-to-terms-e h termsp)
  proof
    have i-terms-acyclic h termsp
      using acyclic acyclic-terms-term-simp get-tr
      by blast
    then have execute (i-terms-to-terms termsp) h = Some (i-terms-to-terms-e h termsp, h)
      using i-terms-to-terms-value
      by blast
    then show ?thesis
      using acyclic get-tr i-term-to-term-terms
      by force
  qed

lemma i-terms-to-terms-nil:
  fixes h :: heap
  shows execute (i-terms-to-terms None) h = Some([], h)
  unfolding i-terms-to-terms-def
  by (subst i-term-to-term-p.simps, simp add: return-def bind-def execute-heap)

lemma i-terms-to-terms-step:
  fixes termsr:: i-terms ref
  and tthis:: i-term ref
  and tnext:: i-termsP
  and term:: term
  and terms:: term list
  and h:: heap
  assumes acyclic: i-terms-acyclic h (Some termsr)
  and get-termsr: Ref.get h termsr = ITerms (tthis, tnext)
  and tthis-res: execute (i-term-to-term tthis) h = Some(term, h)
  and tnext-res: execute (i-terms-to-terms tnext) h = Some(terms, h)
  shows execute (i-terms-to-terms (Some termsr)) h = Some(term#terms, h)
  proof
    have tthis-acyclic: i-term-acyclic h (Some tthis)
      using acyclic get-termsr
      by (cases h Some termsr rule: i-terms-acyclic_cases, fastforce)
    have tnext-acyclic: i-terms-acyclic h tnext
      using acyclic get-termsr by (fact acyclic-terms-terms-simp)
    obtain r0 where r0-def: Some(r0, h) = execute (i-term-to-term-p (Inl tthis)) h

∧ isl r₀

using iverse m-iverse-p-mr tthis-isyclus
by (metis Inr-not-Inl sum_disc(1) sum.sel(1))
then have a₁: Some(r₀, h) = execute (iverse m-iverse-p (Inl tthis)) h by simp
obtain r₀v where a₂: Inl r₀v = r₀
using r₀-def sum.collapse(1) by blast
have a₃: Some(Inl r₀v, h) = Some(Inl term, h)
using tthis-isyclus[unfolded iverse m-iverse-p]
by (simp add: bind-def return-def execute-heap)
(fold a₁ a₂, simp add: execute-heap return-def)

obtain r₁ where r₁-def: Some(r₁, h) = execute (iverse m-iverse-p (Inr tnext)) h ∧ ¬isl r₁
using iverse m-iverse-p-mr[where XX=Inr tnext] tnext-aisyclus
by auto
then have b₁: Some(r₁, h) = execute (iverse m-iverse-p (Inr tnext)) h by simp
obtain r₁v where b₂: Inr r₁v = r₁
using r₁-def sum.collapse(2) by blast
have b₃: Some(Inr r₁v, h) = Some(Inl terms, h)
using tnext-isyclus[unfolded iverse m-iverse-p]
by (simp add: bind-def return-def execute-heap)
(fold b₁ b₂, simp add: execute-heap return-def)

show ?thesis unfolding iverse m-iverse-p
aply (subst iverse m-iverse-p.simps,
simp add: lookup-def tap-def bind-def return-def execute-heap get-termsr)
aply (fold a₁ a₂ b₁ b₂, simp, fold b₁ b₂, simp add: bind-def return-def execute-heap)
using a₃ b₃ by simp
qed

lemma iverse m-iverse-p-isyclus-step:
fixes termsr:: iverse m-iverse-p
and tthis:: iverse
and tnext:: iverse m-iverse-p
and h:: heap
assumes acyclic: iverse m-iverse-p-isyclus (Some termsr)
and get-termsr: Ref get h termsr = ITerms (tthis, tnext)
shows iverse m-iverse-p-step (Some termsr) = (iverse m-iverse-p h tthis)#(iverse m-iverse-p h tnext)
proof –
have iverse m-iverse-p h (Some tthis)
by (meson acyclic get-termsr iverse m-iverse-p-set-isyclus iverse m-iverse-p-setEq iverse m-iverse-p-sublistsp_self)
moreover have iverse m-iverse-p h tnext
using acyclic acyclic-iverse m-iverse-p-set get-termsr by blast
ultimately have execute (iverse m-iverse-p (Some termsr)) h = Some((iverse m-iverse-p h tthis)#(iverse m-iverse-p h tnext), h)
using acyclic get-termsr iverse m-iverse-p-isyclus-value iverse m-iverse-p-step iverse m-iverse-p-value

83
by blast
then show ?thesis by simp
qed

abbreviation i-term-structure-presv where
i-term-structure-presv h0 h1 ≡ ( ∀ tr′ s is d. Ref.get h0 tr′ = ITerm(s, is, d) → (∃ s′. Ref.get h1 tr′ = ITerm(s′, is, d))) ∧ (∀ (isr :: i-terms ref). Ref.get h0 tsr = Ref.get h1 tsr)

lemma i-term-to-term-get-presv:
assumes acyclic: i-term-acyclic h (Some tr)
and get-presv: i-term-structure-presv h h'
shows i-term-to-term-e h tr = i-term-to-term-e h' tr
proof
have i-term-to-term-e h tr = i-term-to-term-e h' tr ∧ i-term-acyclic h' (Some tr)
using assms proof (induction h Some tr arbitrary: tr
taking: λ tsp. i-term-structure-presv h h' →
i-terms-to-terms-e h tsp = i-terms-to-terms-e h' tsp ∧ i-terms-acyclic h' tsp
rule: i-term-acyclic-i-terms-acyclic.inducts(1))
case (t-acyclic-step-link h is tr s)
show ?case
proof (cases is)
case None
then obtain s' where Ref.get h' tr = ITerm (s', None, IVarD)
using typerep-term-neq-nat get-presv heap-only-stamp-ch-get-term
 t-acyclic-step-link
by presburger
moreover from this have i-term-acyclic h' (Some tr)
using i-term-acyclic-i-terms-acyclic.t-acyclic-step-link t-acyclic-nil by blast
ultimately show ?thesis using t-acyclic-step-link None
by (subst (1 2) i-term-to-term-var-none, simp-all)
next
case (Some isr)
then obtain s' where s'-def: Ref.get h' tr = ITerm (s', Some isr, IVarD)
using heap-only-stamp-ch-get-term t-acyclic-step-link by blast
have ttt: i-term-to-term-e h isr = i-term-to-term-e h' isr
using acyclic-term-link-simp i-term-closure.intros(1)
 t-acyclic-step-link Some by blast
moreover have tr-acyclic': i-term-acyclic h' (Some tr) using s'-def
using Some i-term-acyclic-i-terms-acyclic.t-acyclic-step-link t-acyclic-step-link.hyps(2)
 t-acyclic-step-link.prems by blast
show ?thesis
by (simp add: tr-acyclic', subst (1 2) i-term-to-term-var-some, simp-all add: s'-def t-acyclic-step-link
Some)
(fact ttt)
qed

next

case (t-acyclic-step-ITerm h tsref tref s f)
then obtain s' where s'-def: Ref.get h' tref = ITerm (s', None, ITermD (f, tsref))
  using heap-only-stamp-ch-get-term by blast
have acyclic'-tref: i-term-acyclic h' (Some tref)
using i-term-acyclic-i-terms-acyclic.t-acyclic-step-ITerm local.t-acyclic-step-ITerm(4)
  s'-def t-acyclic-step-ITerm.hyps(2) by blast
have acyclic-tref: i-term-acyclic h (Some tref)
using i-term-acyclic-i-terms-acyclic.t-acyclic-step-ITerm local.t-acyclic-step-ITerm(3)
  t-acyclic-step-ITerm.hyps(1) by blast
have tt-step: i-term-to-term-e h tref = T (f, i-terms-to-terms-e h' tsref)
  by (simp add: acyclic-tref i-term-to-term-e-terms t-acyclic-step-ITerm.hyps(2)
    t-acyclic-step-ITTerm.hyps(3) t-acyclic-step-ITerm.prems)
then show ?case
  by (simp add: acyclic'-tref i-term-to-term-e-terms s'-def)

next

case (ts-acyclic-nil uy)
then show ?case
  using i-terms-to-terms-nil
  by (simp add: i-term-acyclic-i-terms-acyclic.ts-acyclic-nil)

next

case (ts-acyclic-step-ITerms h ts2ref tref tsref)
show ?case
  proof (intro impI, goal-cases)
case 1
  then have get-presv: i-term-structure-presv h h' by blast
  then have get-tsref': Ref.get h' tref = ITerms (tref, ts2ref)
    using typerep-term-neq-terms heap-only-stamp-ch-get-terms
    ts-acyclic-step-ITerms.hyps(5) by presburger
  have tsref-acyclic: i-terms-acyclic h (Some tsref)
  using i-term-acyclic-i-terms-acyclic.ts-acyclic-step-ITerms
    ts-acyclic-step-ITerms.hyps(1) ts-acyclic-step-ITerms.hyps(3)
    ts-acyclic-step-ITerms.hyps(5) by blast
  then have tsref-acyclic': i-terms-acyclic h' (Some tsref)
    using heap-only-stamp-ch-terms-get-presv get-tsref'
    i-term-acyclic-i-terms-acyclic.ts-acyclic-step-ITerms
    ts-acyclic-step-ITerms.hyps(2) ts-acyclic-step-ITerms.hyps(4) by fast
  moreover from this
  have i-terms-to-terms-e h (Some tsref) = i-terms-to-terms-e h' (Some tsref)
  apply (subst i-terms-to-terms-e-step[OF tsref-acyclic ts-acyclic-step-ITerms.hyps(5)]))
  apply (subst i-terms-to-terms-e-step[OF tsref-acyclic' get-tsref'])
  using ts-acyclic-step-ITerms.hyps(2) ts-acyclic-step-ITerms.hyps(3)
    ts-acyclic-step-ITerms.hyps(4) get-presv by blast
  ultimately show ?case by blast
qed

qed

then show ?thesis using assms by blast
lemma i-term-to-term-only-stamp-changed:
assumes acyclic: i-term-acyclic h (Some tr)
  and only-stamp-changed: heap-only-stamp-changed trs h h'
shows i-term-to-term-e h tr = i-term-to-term-e h' tr
using assms i-term-to-term-get-presv
using heap-only-stamp-ch-get-term heap-only-stamp-ch-get-terms by auto

lemma i-terms-to-terms-only-stamp-changed:
assumes acyclic: i-terms-acyclic h tsp0
  and only-stamp-changed: heap-only-stamp-changed trs h h'
  and tsp-sublist: tsp ∈ i-terms-sublists h tsp0
shows i-terms-to-terms-e h tsp = i-terms-to-terms-e h' tsp
proof –
  have tsp-acyclic: i-terms-acyclic h tsp
    using acyclic tsp-sublist i-terms-sublists-acyclic by blast
  then show ?thesis using assms tsp-acyclic
    proof (induction h tsp rule: i-terms-acyclic-induct)
      case (ts-acyclic-nil h)
      then show ?case
        by (simp add: i-terms-to-terms-nil)
    next
      case (ts-acyclic-step h ts2ref tref tsref)
      have get'-tsref: Ref.get h tsref = Ref.get h' tsref
        by (metis (lifting) heap-only-stamp-ch-get-terms ts-acyclic-step.prems(2))
      have i-terms-to-terms-e h (Some tsref) =
        (i-term-to-term-e h tref) # (i-terms-to-terms-e h ts2ref)
        using i-terms-to-terms-e-step ts-acyclic-step.hyps(1) ts-acyclic-step.hyps(2)
          ts-acyclic-step.hyps(3) ts-acyclic-step-ITerms by blast
      moreover have i-terms-acyclic h' (Some tsref)
        using heap-only-stamp-ch-terms-acyclic
          ts-acyclic-step.prems(2) ts-acyclic-step.prems(4) by blast
      then have i-terms-to-terms-e h' (Some tsref) =
        (i-term-to-term-e h' tref) # (i-terms-to-terms-e h' ts2ref)
        by (metis (no-types) get'-tsref i-terms-to-terms-e-step ts-acyclic-step.hyps(3))
      moreover have i-term-to-term-e h tref = i-term-to-term-e h' tref
        using i-term-to-term-only-stamp-changed
          ts-acyclic-step.hyps(2) ts-acyclic-step.prems(2) by blast
    ultimately show ?case
    using i-terms-sublists.next ts-acyclic-step.IH ts-acyclic-step.hyps(1)
      ts-acyclic-step.hyps(3) ts-acyclic-step.prems(1) ts-acyclic-step.prems(2)
        ts-acyclic-step.prems(3) ts-acyclic-step.prems(4) by presburger
  qed
qed
lemma i-terms-to-terms-only-stamp-changed':
assumes acyclic: i-terms-acyclic h tsp
  and get-tr: Ref.get h tr = ITerm(s, None, ITermD(f, tsp))
  and only-stamp-changed: heap-only-stamp-changed trs h h'
shows i-terms-to-terms-e h tsp = i-terms-to-terms-e h' tsp
using assms i-terms-to-terms-only-stamp-changed i-terms-sublists self by blast

lemma i-term-to-term-chain:
assumes acyclic: i-term-acyclic h (Some tr)
and chain: tr' ∈ i-term-chain h tr
shows i-term-to-term-e h tr' = i-term-to-term-e h tr
using assms proof (induction h tr rule: i-term-acyclic-induct')
case (var h tr s)
  then have tr' = tr
  using i-term-chain-dest by blast
  then show ?case by simp
next
case (link h tr isr s)
  then show ?case
    using i-term-chain-link i-term-to-term-var-some by force
next
case (args h tr tsp s f)
  then have tr' = tr
  using i-term-chain-dest by blast
  then show ?case by simp
qed

lemma i-find-heap-change-nt:
fixes tr:: i-term ref
  and tdestp:: i-termP
  and r:: i-term ref
  and is:: i-termP
  and v:: i-term
  and h:: heap
assumes acyclic: i-term-acyclic h (Some tr)
  and TYPEREP(a') ≠ TYPEREP(i-term)
shows ∃ tdestp. (execute (i-find (Some tr)) (Ref.set r v h) = Some (tdestp, Ref.set r v h)) ∧
  execute (i-find (Some tr)) h = Some (tdestp, h))
using assms by
  (induction rule: i-term-acyclic-induct')
  (subst (1 2) i-find.simps,
    simp add: lookup-def bind-def tap-def return-def execute-heap Ref.get-def
Ref.set-def)

lemma i-find-heap-change-is-ae:
fixes tr:: i-term ref
  and tdestp:: i-termP
  and r:: i-term ref
  and is:: i-termP
  and v:: i-term

87
and $h :: heap$

assumes acyclic: $i$-term-acyclic $h$ (Some $tr$)
and $Ref.get h r = ITerm(s, is, d)$
and $v = ITerm(s', is, d')$

shows
\[(\text{execute } (i\text{-find (Some $tr$))) } (Ref.set r v h) = \text{Some } (tdestp, Ref.set r v h))\]

using assms proof
(induction rule: $i$-term-acyclic-induct')
case (var $h$ $tr$ $s$)
then show ?case
  by (subst (1 2) $i\text{-find.simps}$)
  (auto simp add: lookup-def bind-def tap-def return-def execute-heap
                   $Ref.get-def$ $Ref.set-def$)
next
case (link $h$ $tr$ $isr$ $s$)
then show ?case
  by (subst (1 2) $i\text{-find.simps}$)
  (auto simp add: lookup-def bind-def tap-def return-def execute-heap
                   $Ref.get-def$ $Ref.set-def$)
next
case (args $h$ $tr$ $tsp$ $s$ $f$)
then show ?case
  apply (subst (1 2) $i\text{-find.simps}$)
  apply (simp add: lookup-def bind-def tap-def return-def execute-heap
                   $Ref.get-def$ $Ref.set-def$)
  by (auto simp add: return-def execute-heap)
qed

lemma $i$-find-some:
fixes $tr :: i$-term ref
and $tdestr :: i$-term ref
and $h :: heap$
assumes $i$-term-acyclic $h$ (Some $tr$
shows $\exists tdestr s d.\ (\text{execute } (i\text{-find (Some $tr$))) } h = \text{Some } (\text{Some } tdestr, h) \land$
\hspace{1cm} $tdestr \in i$-term-chain $h$ $tr \land$
\hspace{1cm} $Ref.get h tdestr = ITerm(s, None, d)$
using assms proof (induction rule: $i$-term-acyclic-induct')
case (var $h$ $tr$ $s$)
then show ?case
  by (subst $i\text{-find.simps}$, simp add: bind-def lookup-def tap-def return-def execute-heap
                           $i$-term-chain.self)
next
case (link $h$ $tr$ $isr$ $s$)
then have $*:\ (\text{execute } (i\text{-find (Some $tr$))) } h = \text{execute } (i\text{-find (Some $isr$))) h$
  by (subst $i\text{-find.simps}$, simp add: bind-def lookup-def tap-def)
from $link$ obtain $tdestr s'$ $d'$ where
execute (i-find (Some isr)) h = Some (Some tdestr, h) ∧ 
tdestr ∈ i-term-chain h isr ∧ Ref.get h tdestr = ITerm (s', None, d') 
by blast 
then have tdestr ∈ i-term-chain h tr using i-term-chain-link link.hyps by blast 
then show ?case using * ** by simp 
next 
case (args h tr tsp s f) 
then show ?case 
by (subst i-find.simps, 
simp add: bind-def lookup-def tap-def return-def execute-heap i-term-chain.self) 
qed 
definition stamp-current-not-occurs where 
stamp-current-not-occurs time vr tr h = 
(∀ tr' s' is d. 
tr' ∈ i-term-closure h (Some tr) →
Ref.get h tr' = ITerm (s', is, d) →
s' = Ref.get h time →
¬occurs ("x", int (addr-of-ref vr)) (i-term-to-term-e h tr')) 
abbreviation stamp-current-not-occurs' where 
stamp-current-not-occurs' time vr tr h ≡ 
¬occurs ("x", int (addr-of-ref vr)) (i-term-to-term-e h tr) →
stamp-current-not-occurs time vr tr h 
abbreviation stamp-current-not-occurs'-'ts where 
stamp-current-not-occurs'-'ts time vr tsp h ≡ 
¬list-ex (occurs ("x", int (addr-of-ref vr))) (i-terms-to-terms-e h tsp) →
(∀ tr ∈ i-terms-set h tsp. stamp-current-not-occurs time vr tr h) 
lemma i-terms-to-terms-list-set: 
assumes i-terms-acyclic h tsp 
shows set (i-terms-to-terms-e h tsp) = i-term-to-term-e h ' i-terms-set h tsp 
using assms proof (induction h tsp rule: i-terms-acyclic-induct) 
case (ts-acyclic-nil h) 
show ?case using i-terms-to-terms-nil i-terms-set-None-empty by force 
next 
case (ts-acyclic-step h ts2ref tref tsref) 
then have i-terms-to-terms-e h (Some tsref) = 
i-term-to-term-e h tref ≠ i-terms-to-terms-e h ts2ref 
using i-terms-to-terms-e-step ts-acyclic-step-ITerms by presburger 
then show ?case 
by (simp add: i-terms-set-insert ts-acyclic-step.1H ts-acyclic-step.hyps(3)) 
qed 
lemma stamp-current-not-occurs'-'terms-set: 
assumes terms-scno: ∃ tr. tr ∈ i-terms-set h tsp → stamp-current-not-occurs' 
time vr tr h' 
and terms-hosc: heap-only-stamp-changed-ts tsp h h'
\textbf{and} acyclic: \textit{i-term-acyclic} \(h \ (\text{Some} \ tr0)\)

\textbf{and} get-tr0: \textit{Ref}.\textit{get} \(h \ tr0 = \text{ITerm}(s, \text{None}, \text{ITermD}(f, \ tsp))\)

\textbf{shows} stamp-current-not-occurs' \(\text{time} \ vr \ tr0 \ h'\)

\textbf{unfolding} stamp-current-not-occurs-def

\textbf{proof} (intro allI impI)

\begin{itemize}
  \item \textbf{fix} \(tr'\), \(s'\) is \(d\)
  \item \textbf{assume} not-occurs: \(\sim\) occurs \("x', \ \text{int} (\text{addr-of-ref} \ vr)\) \(\text{i-term-to-term-e} \ h' \ tr0\)
  \item \textbf{and} \(tr'-clos\): \(tr' \in \text{i-term-closure} \ h' \ (\text{Some} \ tr0)\)
  \item \textbf{and} get'\(\text{tr'}\): \textit{Ref}.\textit{get} \(h' \ tr' = \text{ITerm}(s', \text{is}, \ d)\)
  \item \textbf{and} \(s'-\text{time}': s' = \text{Ref}.\textit{get} \ h' \text{ time}\)
\end{itemize}

\textbf{obtain} \(\exists s2\) \textbf{where} get'\(\text{tr0}\): \textit{Ref}.\textit{get} \(h' \ tr0 = \text{ITerm}(s2, \text{None}, \text{ITermD}(f, \ tsp))\)

\textbf{using} get-tr0 heap-only-stamp-ch-get-term terms-hosc by blast

\textbf{have} \(\text{ttt}\): \textit{i-term-to-term-e} \(h \ tr0 = \text{i-term-to-term-e} \ h' \ tr0\)

\textbf{using} acyclic \textit{i-term-to-term-only-stamp-changed} terms-hosc by fastforce

\textbf{have} \(\text{tr0-acyclic}'\): \textit{i-term-acyclic} \(h' \ (\text{Some} \ tr0)\)

\textbf{using} acyclic heap-only-stamp-ch-term-acyclic terms-hosc by blast

\textbf{then have} tsp-acyclic': \textit{i-term-acyclic} \(h' \ tsp\)

\textbf{using} acyclic \textit{ terms-hosc} by blast

\textbf{have} \(\exists t\): \textit{i-term-to-term-e} \(h \ tr0 = T(f, \text{i-term-to-term-e} \ h' \ tsp)\)

\textbf{by} (simp add: tr0-acyclic' get'\(\text{tr0}\) i-term-to-term-e-terms)

\begin{itemize}
  \item \textbf{fix} \(tr\)
  \item \textbf{assume} \(\text{tr-tsp-set}\): \(tr \in \text{i-term-set} \ h \ tsp\)
  \item \textbf{assume} \(\text{occ-tr}:\) occurs \("x'', \ \text{int} (\text{addr-of-ref} \ vr)\) \(\text{i-term-to-term-e} \ h' \ tr0\)
  \item \textbf{have} \(\text{tr-tsp-set}'\): \(tr \in \text{i-term-set} \ h' \ tsp\)
  \item \textbf{using} \(\text{tr-tsp-set get-tr0 heap-only-stamp-ch-term-set} \ \text{terms-hosc by blast}\)
  \item \textbf{have} \(\text{list-ex occ-tr}:\) \textit{occurs} \("x'', \ \text{int} (\text{addr-of-ref} \ vr)\) \(\text{i-term-to-term-e} \ h' \ tsp\)

  \begin{itemize}
    \item \(\exists t\): \textit{i-term-to-term-e} \(h' \ tsp\)
    \item \textbf{using} \(\text{list-ex occ-tr}:\) \textit{occurs} \("x'', \ \text{int} (\text{addr-of-ref} \ vr)\)
    \item \textbf{by} (simp add: \(\text{ttt}\))
    \item \textbf{then have} \(\text{False}\)
    \item \textbf{using} not-occurs by simp
  \end{itemize}

\end{itemize}

\textbf{then have} \(\text{terms-scn0'}: \bigwedge tr. \ tr \in \text{i-term-set} \ h \ tsp \implies \text{stamp-current-not-occurs} \ \text{time} \ vr \ tr' \ h'\)

\textbf{using} \(\text{terms-scn0 by auto}\)

\begin{itemize}
  \item \textbf{consider} (a) \(tr' = tr0\) | \(b\) \(tr'0\) \ where \(tr'0 \in \text{i-term-set} \ h' \ tsp\) \textbf{and} \(tr' \in \text{i-term-closure} \ h' \ (\text{Some} \ tr'0)\)
  \item \textbf{using} \(\text{tr'-clos i-term-closure-args[OF get'\(\text{tr0}\)] by blast}\)
  \item \textbf{then show} \(\sim\) occurs \("x'', \ \text{int} (\text{addr-of-ref} \ vr)\) \(\text{i-term-to-term-e} \ h' \ tr'\)
\end{itemize}

\textbf{proof} (cases)
case a
  then show \texttt{thesis using ttt not-occurs by presburger}
next
  case b
  then have stamp-current-not-occurs time vr tr'0 h'
    using terms-scno\' get-tr0 heap-only-stamp-ch-terms-set terms-hosc by blast
  then have \(\ast\): stamp-current-not-occurs time vr tr' h'
    unfolding stamp-current-not-occurs-def using b(2)
  then show \texttt{thesis using ttt \([\text{unfolded stamp-current-not-occurs-def}]\) by blast}
qed

lemma stamp-current-not-occurs-terms-set:
  assumes terms-scno: \(\bigwedge tr. tr \in \text{i-terms-set } h'\) tsp \(\Rightarrow\) stamp-current-not-occurs time vr tr h'
  and terms-hosc: heap-only-stamp-changed-ts tsp h h'
  and acyclic: i-term-acyclic h (Some tr0)
  and get-tr0: Ref.get h tr0 = ITerm(s, None, ITermD(f, tsp))
  and not-occurs: \(\neg\)occurs ("x", int (addr-of-ref vr)) (i-term-to-term-e h tr0)
shows stamp-current-not-occurs time vr tr0 h'
unfolding stamp-current-not-occurs-def
proof (intro allI impI)
  fix tr' s' is d
  assume tr'-'clos: tr' \(\in\) i-term-closure h' (Some tr0)
  and get'-'tr': Ref.get h' tr' = ITerm (s', is, d)
  and s'-time': s' = Ref.get h' time
  obtain s2 where get'-'tr0: Ref.get h' tr0 = ITerm(s2, None, ITermD(f, tsp))
  using get-tr0 heap-only-stamp-ch-get-term terms-hosc by blast
  have ttt: i-term-to-term-e h tr0 = i-term-to-term-e h' tr0
    using acyclic i-term-to-term-only-stamp-changed terms-hosc by fastforce
  have i-term-acyclic h' (Some tr0)
    using acyclic heap-only-stamp-ch-term-terms-acyclic terms-hosc by blast
  consider (a) tr' = tr0 |
    (b) tr'0 where
      tr'0 \(\in\) i-terms-set h' tsp and
      tr' \(\in\) i-term-closure h' (Some tr'0)
    using tr'-'clos i-term-closure-args\([OF get'-'tr0]\) by blast
  then show \(\neg\)occurs ("x", int (addr-of-ref vr)) (i-term-to-term-e h' tr')
proof (cases)
  case a
    then show \texttt{thesis using ttt not-occurs by presburger}
next
  case b
  then have stamp-current-not-occurs time vr tr'0 h'
    by (simp add: terms-scno)
  then have \(\ast\): stamp-current-not-occurs time vr tr' h'
    unfolding stamp-current-not-occurs-def using b(2)
using i-term-closure-trans by blast
then show \textit{thesis using ttt }\textit{[unfolded stamp-current-not-occurs-def]}
using get\textsuperscript{\prime}\textit{-time }i-term-closure.intros(1) by blast
qed

lemma stamp-current-not-occurs-terms-set-None:
assumes hosc: heap-only-stamp-changed-tr tr h h'
and get-tr: \textit{Ref.get h tr = ITerm(s, None, ITermD(f, None))}
shows stamp-current-not-occurs time vr tr h'
unfolding stamp-current-not-occurs-def
proof (intro allI impI)
fix tr' s' is d
assume tr'-clos: tr' \in i-term-closure h' (Some tr)
and Ref.get h' tr' = ITerm (s', is, d)
and s' = Ref.get h' time
obtain s2 where get'-tr: \textit{Ref.get h' tr = ITerm(s2, None, ITermD(f, None))}
using hosc get-tr heap-only-stamp-ch-get-term by blast
then have i-term-closure h' (Some tr) = \{tr\}
using i-term-closure-args i-terms-set-None-empty by force
then have tr'-eq-tr: tr' = tr using tr'-clos by blast
have i-term-acyclic h' (Some tr')
using get'-tr i-acyclic-step ITerm tr'-eq-tr ts-acyclic-nil by blast
then have i-term-to-term-e h' tr' = T(f, [])
by (simp add: get'-tr i-term-to-term-e-terms i-terms-to-terms-nil tr'-eq-tr)
then show \exists h. execute (i-occ-p time (Some vr) (Inl(Some tr))) (i-term-to-term-e h' tr')
by simp
qed

lemma i-occ-p-sound:
fixes vr:: i-term ref
and tr:: i-term ref
and time:: nat ref
and td:: i-term-d
and h:: heap
and fun-term:: term
and s1:: nat
and s2:: nat
assumes acyclic: i-term-acyclic h (Some tr)
and Ref.get h tr = ITerm (s1, None, td)
and Ref.get h vr = ITerm (s2, None, IVarD)
and Some(fun-term, h) = execute (i-term-to-term tr) h
and stamp-current-not-occurs time vr tr h
and r-val: r = occurs ("x", int (addr-of-ref vr)) fun-term
shows \exists h'. execute (i-occ-p time (Some vr) (Inl(Some tr))) h = Some(r, h') \land
heap-only-stamp-changed-tr tr h h' \land
stamp-current-not-occurs' time vr tr h'
proof –
let ?occ-vr = occurs ("x", int (addr-of-ref vr))

92
let \( ?\text{occ} \ h \ tr = ?\text{occ-vr} \ (i\text{-term-to-term-e} \ h \ tr) \)
let \( ?\text{occ-ts} \ h \ tsp = \text{list-ex} \ ?\text{occ-vr} \ (i\text{-terms-to-terms-e} \ h \ tsp) \)
let \( ?\text{upd-s} \ h \ tr \ f \ tsp = \text{Ref.set} \ tr \ (ITerm \ (\text{Ref.get} \ h \ \text{time}, \ \text{None}, \ ITermD \ (f, \ tsp))) \)

let \( ?\text{cond} \ h \ tr \)

\[
\begin{align*}
\exists \ h' . \ \text{execute} \ (i\text{-occ-p} \ \text{time} \ \text{Some} \ vr \ (\text{Inl}(\text{Some} \ tr))) \ h = \\
\text{Some} \ (\text{occ-vr} \ \text{fun-term}, \ h') \land \\
\text{heap-only-stamp-changed-tr} \ tr \ h' \land \\
\text{stamp-current-not-occurs} \ \text{time} \ vr \ tr \ h'
\end{align*}
\]

\{ fix \( \text{trs} :: \ i\text{-term \ ref \ set} \)
have \( \text{trs} = \text{UNIV} \iff ?\text{cond} \ tr \)
using acyclic assms(2) assms(3) assms(4) assms(5) acyclic
proof (induction \( h \ \text{trs} \ tr \)
arbitrary: \( \text{fun-term} \ s1 \ s2 \ td \)
taking:
\( \lambda h \ \text{trs} \ tsp. \)
\( \forall \ s2 . \)
\( \text{Ref.get} \ h \ vr = \text{ITerm} (s2, \ \text{None}, \ \text{IVarD}) \longrightarrow \)
\( \text{trs} = \text{UNIV} \longrightarrow \)
\( (\forall \ tr \in \ i\text{-terms-set} \ h \ tsp. \ \text{stamp-current-not-occurs} \ \text{time} \ vr \ tr \ h) \longrightarrow \)
\( \text{i-terms-acyclic} \ h \ tsp \longrightarrow \)
\( (\exists \ h'. \ \text{execute} \ (i\text{-occ-p} \ \text{time} \ \text{Some} \ vr \ (\text{Inr} \ tsp)) \ h = \\
\text{Some} \ (\text{occ-ts} \ h \ tsp, \ h') \land \\
\text{heap-only-stamp-changed-ts} \ tsp \ h' \land \\
\text{stamp-current-not-occurs} \ \text{-ts} \ \text{time} \ vr \ tr \ h') \)
rule: acyclic-closure-ch-stamp-inductc
\}
case \( \text{var} \ h \ \text{trs} \ tr \ s \)
then have \( \text{get-tr} : \ \text{Ref.get} \ h \ tr = \text{ITerm} (s, \ \text{None}, \ \text{IVarD}) \)
and \( \text{get-tr'} : \ \text{Ref.get} \ h \ tr = \text{ITerm} (s1, \ \text{None}, \ td) \)
and \( \text{scno}: \ \text{stamp-current-not-occurs} \ \text{time} \ vr \ tr \ h \)
and acyclic: \( i\text{-term-acyclic} \ h \ (\text{Some} \ tr) \)
and \( \text{fun-term}: \ \text{Some} \ \text{fun-term} \ (\text{h}) = \text{execute} \ (i\text{-term-to-term} \ tr) \ h \)
by simp-all
from \( \text{fun-term acyclic} \)
have \( \text{fun-term} = i\text{-term-to-term-e} \ h \ tr \)
using \( \text{i-term-to-term-value-iff} \)
by simp
then have \( ** : (vr = tr) = ?\text{occ-vr} \ \text{fun-term} \)
using \( \text{var} \ \text{i-term-to-term-var-none} \)
by force
show \( ?\text{case} \)
using \( \text{var} \)
apply (subst \( i\text{-occ-p-simps} , \)
    simp add: lookup-def update-def tap-def bind-def return-def execute-heap
** )
using heap-only-stamp-changed-def
by blast
next
case \( \text{link} \ h \ tr \ isr \ s \)
then show \( ?\text{case} \)
by force
next
case (args h trs tr tsp s f fun-term s1 s2 trs')
then have get-tr: Ref.get h tr = ITerm (s, None, ITermD (f, tsp))
and get-vr: Ref.get h vr = ITerm (s2, None, IVarD)
and acyclic: i-term-acyclic h (Some tr)
and fun-term-val: Some (fun-term, h) = execute (i-term-to-term tr) h
and scno: stamp-current-not-occurs time vr tr h
and trs-val: trs = UNIV by blast+
have fun-term-e: fun-term = i-term-to-term e h tr
  by (metis acyclic fun-term-val i-term-to-term-value-iff)
show ?case
proof (rule case-split)
assume s-eq-time: s = Ref.get h time
then have *: ¬ ?occ-vr fun-term
  using scno [unfolded stamp-current-not-occurs-def] fun-term-e
proof (rule case-split)
assume s-neq-time: s ≠ Ref.get h time
let ?h' = Ref.set tr (ITerm (Ref.get h time, None, ITermD (f, tsp))) h
have hosc-h-h': heap-only-stamp-ch-closure tr h ?h'
  unfolding heap-only-stamp-ch-term OF get-tr
  by simp
have tsp-acyclic: i-terms-acyclic h tsp
  using acyclic acyclic-terms-term-simp get-tr by blast
have get'-tr: Ref.get ?h' tr = ITerm (Ref.get h time, None, ITermD (f, tsp))
  by simp
have tsp-scno: ∀ tr ∈ i-terms-set ?h' tsp. stamp-current-not-occurs time vr tr
  unfolding stamp-current-not-occurs-def
proof (intro ballI allI impl)
fix tr0 tr' s' is d
assume tr0-tsp-set': tr0 ∈ i-terms-set ?h' tsp
and tr'-clos': tr' ∈ i-term-closure ?h' (Some tr0)
and get'-tr': Ref.get ?h' tr' = ITerm (s', is, d)
and s'-time': s' = Ref.get ?h' time
then have tr' ∈ i-term-closure ?h' (Some tr)
  by (meson get'-tr i-term-closure-trans)
then have tr-clos-tr: tr' ∈ i-term-closure h (Some tr)
  using hosc-h-h' heap-only-stamp-ch-closure by blast
have get-tr': Ref.get h tr' = ITerm (s', is, d)
proof (rule case-split)
assume tr' = tr
then show ?thesis
using acyclic get\'-tr heap-only-stamp-ch-term-acyclic hosc-h-h'  
i-term-closure-args-same-cyclic tr\'-clos' tr0-tsp-set' by blast

next
  assume tr' \neq tr
  then show \emph{thesis}
    using get\'-tr' by auto
  qed

have s\'-time: s' = Ref.get h time  
by (metis (no-types, lifting) heap-only-stamp-ch-get-nat hosc-h-h' s\'-time')

have tr0-acyclic': i-term-acyclic \emph{?h'} (Some tr0)  
using heap-only-stamp-ch-term-terms-acyclic hosc-h-h' i-terms-set-acyclic  
tr0-tsp-set'
  tsp-acyclic by blast
  have \neg occurs ("x", int (addr-of-ref vr)) (i-term-to-term-e h tr')  
  using scno\{unfolded stamp-current-not-occurs-def\} tr-clos-tr get-tr' s\'-time
  by fast
  then show \neg occurs ("x", int (addr-of-ref vr)) (i-term-to-term-e \emph{?h'} tr')
    using tr0-acyclic' heap-only-stamp-ch-sym hosc-h-h' i-term-closure-acyclic  
i-term-to-term-only-stamp-changed tr\'-clos' by metis
  qed

have get\'-vr: Ref.get \emph{?h'} vr = ITerm (s2, None, IVarD)  
  by (metis (no-types, hide-lams) Ref.get-set-neq Ref.unequal get-tr get-vr  
i-term.inject i-term-d.distinct(1) snd-cone)

have tsp-acyclic': i-terms-acyclic \emph{?h'} tsp
  using heap-only-stamp-ch-term-terms-acyclic hosc-h-h' tsp-acyclic by blast

have hosc-h-h'\'-trs: heap-only-stamp-ch-term-acyclic tsp \emph{?h'} tsp'  
  using hosc-h-h' trs-val  
  get-tr heap-only-stamp-ch-term by auto

have i-terms-closure \emph{?h'} tsp = i-terms-closure h tsp
  using heap-only-stamp-ch-term-terms-closure hosc-h-h' by presburger

obtain h'' where
  IH-exec: execute (i-occ-p time (Some vr) (Inr tsp')) \emph{?h'} = Some(?occ-ts ?h'  
tsp, h'') and
  IH-hosc: heap-only-stamp-ch-term-acyclic tsp \emph{?h'} h'' and
  IH-concl: stamp-current-not-occurs-ts time vr tsp h''
  using args.hyps(1) hosc-h-h'\'-trs get\'-vr  
  trs-val tsp-scno tsp-acyclic' by blast

show \emph{case}

proof (rule case-split)
  assume i-terms-set \emph{?h'} tsp = \{
  then have tsp-none: tsp = None
    using i-terms-set-empty-iff by simp
  then have fun-term = T([f, []])
    by (simp add: acyclic fun-term-e get-tr i-term-to-term-terms i-terms-to-term-nil)
then have $\ast$: $\neg \text{ooc-\text{vr} fun-term}$
  by simp

have $\ast\ast$: heap-only-stamp-changed-tr $tr$ $h$
  $(\text{Ref.set} tr (\text{ITerm} (\text{Ref} \text{get} \ h \ \text{time}, \text{None}, \text{ITermD} \ (f, \text{None}))) \ h)$
  using hosec-$h$-$h''$ tsp-none by auto

have $\ast\ast\ast$: stamp-current-not-occurs $'$ time vr $tr$
  $(\text{Ref.set} tr (\text{ITerm} (\text{Ref} \text{get} \ h \ \text{time}, \text{None}, \text{ITermD} \ (f, \text{None}))) \ h)$
  using stamp-current-not-occurs-terms-set-None
  get-tr hosec-$h$-$h'$ tsp-none by blast

show $'$thesis
  by (subst i-occ-p.simps, subst i-occ-p.simps,
  simp add: lookup-def update-def tap-def bind-def return-def execute-heap
  args s-neq-time tsp-none $\ast\ast\ast$)

next

assume tsp-set-not-empty: $i\text{-terms-set }h' \ tsp \neq \{\}$

have $i\text{-terms-closure }h' \ tsp \subseteq i\text{-term-closure }h' \ ('\text{Some } tr)$
  using get$'\text{-tr }i\text{-term-closure-args}$ by blast

then have hosec: heap-only-stamp-changed-tr $tr$ $h'$ $h''$ using IH-hosc
  get$'\text{-tr heap-only-stamp-ch-antimono}$ by meson

have fun-term$: $i\text{-term-to-term-e }h'$ $tr$ $=$ fun-term
  using acyclic fun-term-e hosec-$h$-$h'$ $i\text{-term-to-term-only-stamp-changed}$ by auto

have occ-tr-eq-occ-tsp: $\text{ooc-\text{vr} fun-term }= \text{ooc-ts } h \ tsp$
  by (simp add: acyclic fun-term-e get-tr $i\text{-term-to-term-e-terms}$)

also have occ-tr-eq-occ$'$-tsp: $... = \text{ooc-ts }h' \ tsp$
  using i-terms-to-terms-only-stamp-changed$'\text{[OF tsp-acyclic get-tr hosec}$-$h$-$h'[]
  by presburger

have occ$'$-tr-eq-occ$'$-tsp: $\text{ooc }h' \ tr = \text{ooc-ts } h' \ tsp$
  by (simp add: fun-term$'$ occ-tr-eq-occ$'$-tsp occ-tr-eq-occ-tsp)

have hosec-$h$-$h''$: heap-only-stamp-changed-tr $tr$ $h \ h''$
  using heap-only-stamp-ch-trans hosec hosec-$h$-$h'$ heap-only-stamp-ch-closure
  by (metis (no-types, lifting))

have $i\text{-terms-closure }h'' \ tsp \subseteq i\text{-term-closure }h'' \ ('\text{Some } tr)$
  using i-term-closure-args IH-hosc get$'\text{-tr heap-only-stamp-ch-get-term}$ by blast

have fun-term$: $i\text{-term-to-term-e }h'' \ tr = \text{fun-term}$
  using acyclic fun-term-e hosec-$h$-$h''$ $i\text{-term-to-term-only-stamp-changed}$
  by auto

have itt-tsp$: $i\text{-terms-to-terms-e } h \ tsp = i\text{-terms-to-terms-e } h'' \ tsp$
  using get-tr hosec-$h$-$h''$ $i\text{-terms-to-terms-only-stamp-changed}$ tsp-acyclic
  by blast

have tr-acyclic$: i\text{-term-acyclic }h' \ ('\text{Some } tr)$
  using get$'\text{-tr l-acyclic-step-ITerm tsp-acyclic}$ by blast

have terms-set$'\text{-tsp-to}$$: $i\text{-terms-set }h' \ tsp = i\text{-terms-set } h'' \ tsp$
  using IH-hosc get$'\text{-tr heap-only-stamp-ch-terms-set}$ by blast

then have scno-h$: $\text{stamp-current-not-occurs'} \ tsp \ tr \ h''$
  using stamp-current-not-occurs$'$-terms-set IH-concl IH-hosc fun-term$''$

get$'\text{-tr}$
have \(?occ\)-ts \(?h\)' tsp = \(?occ\)-vr fun-term
using fun-term' \(occ\)-tr-eq-occ'-tsp by blast
then show \(?case\)
by (subst i-occ-p.simps,
simp add: lookup-def update-def tap-def bind-def return-def execute-heap
args s-neq-time IH-exec hosc-h-h'' scno-h'' )
qed

next

case (terms-nil \(h\))
then show \(?case\)
proof (intro allI impI, goal-cases)
case 1
then show \(?case\)
by (subst i-occ-p.simps,
simp add: lookup-def update-def tap-def bind-def return-def execute-heap,
simp add: heap-only-stamp-ch-refl i-terms-to-terms-nil)
qed

next

case (terms \(h\) \(trs\) tthisr tsr tsnextp)
then have get-tsr: Ref.get \(h\) tsr = \(ITerm\) (tthisr, tsnextp) by blast
show \(?case\)
proof (intro impI allI, goal-cases)
case (1 \(s2\))
then have get-vr: Ref.get \(h\) vr = \(ITerm\) (s2, None, IVarD)
and terms-scno:
\(\forall\) tr. tr \(\in\) i-terms-set \(h\) (Some tsr) \(\implies\)
stamp-current-not-occurs time vr tr \(h\)
and terms-acyclic: \(i\)-terms-acyclic \(h\) (Some tsr)
and trs-val: \(trs\) = UNIV
by blast+

from terms-acyclic obtain tdestr \(d'\) \(s'\) where
exec-ifind: execute (i-find (Some tthisr)) \(h\) = Some(Some tdestr, \(h\)) and
tdestr-mem: tdestr \(\in\) i-term-chain \(h\) tthisr and
get-tdestr: Ref.get \(h\) tdestr = \(ITerm\)(s', None, \(d'\))
proof (cases \(h\) Some tsr rule: i-terms-acyclic.cases,
goal-cases step-ITerms)
case (step-ITerms ts2ref tref)
have tref = tthisr
using get-tsr step-ITerms(4) by simp
then show \(?case\)
using i-find-some step-ITerms(1) step-ITerms(3) by blast
qed

have tthisr-acyclic: \(i\)-term-acyclic \(h\) (Some tthisr)
using terms-acyclic get-tsr i-terms-set.intros i-terms-set-acyclic
have exec-ifind': execute (i-find (Some thisor)) h = Some (Some tdestr, h)
    using exec-ifind thisor-acyclic
    i-find-heap-change-is-uc by blast

have tdestr-thisor-closure: tdestr ∈ i-term-closure h (Some thisor)
    using tdestr-mem i-term-chain-subset-closure by blast

have thisor-terms-set-tsp: thisor ∈ i-terms-set h (Some tsr)
    using get-tsr i-terms-set.intro i-terms-sublists.self get-tsr by blast

have tdestr-ttt: Some (i-term-to-term-e h tdestr, h) = execute (i-term-to-term tdestr) h
    using i-term-closure-acyclic i-term-to-term-value tdestr-thisor-closure
    thisor-acyclic
    by presburger

have stamp-current-not-occurs time vr thisor h
    using terms-scno
    by (simp add: thisor-terms-set-tsp)
moreover have tdestr-clos-subset-thisor-clos:
    i-term-closure h (Some tdestr) ⊆ i-term-closure h (Some thisor)
    using i-term-closure-trans tdestr-thisor-closure by blast
ultimately have scno-tdestr: stamp-current-not-occurs time vr tdestr h
    using stamp-current-not-occurs-def by blast
have tdestr-acyclic: i-term-acyclic h (Some tdestr)
    using i-term-closure-acyclic tdestr-thisor-closure thisor-acyclic by auto
obtain h' where
    IH-exec:
    execute (i-occ-p time (Some vr) (Inl (Some tdestr))) h = Some (?occ h tdestr, h')
    and
    IH-scno: stamp-current-not-occurs' time vr tdestr h'
    using terms.IH[OF - heap-only-stamp-ch-refl tdestr-thisor-closure trs-val get-tdestr get-vr tdestr-ttt scno-tdestr tdestr-acyclic]
    by blast

have tdestr-clos-subset-tsr-clos:
    i-term-closure h (Some tdestr) ⊆ i-terms-closure h (Some tsr)
    using tdestr-clos-subset-thisor-clos thisor-terms-set-tsp by auto

have hesr-tsr: heap-only-stamp-ch-antimono IH-hesr tdestr-clos-subset-tsr-clos by blast
have scno-tsnextp: ∀ tr ∈ i-terms-set h tsnextp. stamp-current-not-occurs time vr tr h
    by (simp add: get-tsr i-terms-set-insert terms-scno)
have tsnextp-acyclic: i-terms-acyclic h tsnextp
    using acyclic-terms-terms-simp get-tsr terms-acyclic by blast
have \( \text{tsr-acyclic}' \): \( \text{i-terms-acyclic} h' \) (Some tsr)

by (meson heap-only-stamp-ch-terms-acyclic hosc-tsr terms-acyclic)

have get'-tsr: \( \text{Ref} \cdot \text{get} h' \) tsr = ITerms \( (tthisr, tsnextp) \)

using get-tsr heap-only-stamp-ch-get-terms hosc-tsr by auto

have ttt-tdestr: \( \text{i-term-to-term-e} h tthisr = \text{i-term-to-term-e} h tdestr \)

using i-term-to-term-chain tdestr-mem thisr-acyclic by presburger

then have ttt-ts: \( \text{i-terms-to-terms-e} h \) \( (\text{Some tsr}) \) = i-term-to-term-e h tdestr # i-terms-to-terms-e h tsnextp

by (simp add: get-tsr i-terms-to-terms-e-step terms-acyclic)

then have ttt-ts': \( \text{i-terms-to-terms-e} h' \) (Some tsr) = i-term-to-term-e h' tdestr # i-terms-to-terms-e h' tsnextp

using get'-tsr tsr-acyclic' hosc-tsr i-term-to-term-only-stamp-changed

i-terms-to-terms-e-step tdestr-acyclic thisr-acyclic ttt-tdestr by presburger

have scno-ts: \( \text{stamp-current-not-occurs}'-ts \) time vr \( (\text{Some tsr}) \) h'

unfolding stamp-current-not-occurs-def

proof (intro impI allI ballI)

fix \( tr \) \( tr' \) \( s' \) is d

assume not-occ-ts: \( \neg \) occ-ts h' (Some tsr)

and tr-lsr-term-set': \( tr \in \text{i-terms-set} h' \) (Some tsr)

and tr-clos-tr': \( tr' \in \text{i-term-closure} h' \) (Some tr)

and get'-tr': \( \text{Ref} \cdot \text{get} h' \) \( tr' = \text{ITerm} (s', is, d) \)

and s'-time': \( s' = \text{Ref} \cdot \text{get} h' \) time

have not-occ'-tdestr: \( \neg \) occ h' tdestr

using ttt-ts' not-occ-tsr by auto

show \( \neg \) occ h' \( tr' \)

proof (rule case-split)

assume tr' \( \in \) i-term-closure h' (Some tdestr)

then show ?thesis

using IH-scno{unfolded stamp-current-not-occurs-def} not-occ'-tdestr

s'-time' get'-tr' by blast

next

assume tr' \( \notin \) i-term-closure h' (Some tdestr)

then have get-tr': \( \text{Ref} \cdot \text{get} h \) \( tr' = \text{ITerm} (s', is, d) \)

using IH-hosc get'-tr'

heap-only-stamp-ch-closure heap-only-stamp-ch-get-term-nclos by force

have tr-lsr-term-set: \( tr \in \text{i-terms-set} h \) (Some tsr)

using heap-only-stamp-ch-terms-set IH-hosc tr-lsr-term-set' by auto

have tr-clos-tr: \( tr' \in \text{i-term-closure} h \) (Some tr)

using IH-hosc heap-only-stamp-ch-closure tr-clos'-tr by auto

have s'-time': \( s' = \text{Ref} \cdot \text{get} h \) time

using IH-hosc heap-only-stamp-ch-get-nat s'-time' by auto

have \( \neg \) occ h \( tr' \)

using terms-scno{unfolded stamp-current-not-occurs-def}

tr-lsr-term-set tr-clos-tr get-tr' s'-time' by fast

moreover have i-term-to-term-e h \( tr' = \) i-term-to-term-e h' \( tr' \)

using IH-hosc i-term-closure-acyclic i-term-to-term-only-stamp-changed

99
ultimately show \( \text{thesis} \) by fastforce

qed

show \( \text{thesis} \)

proof (rule case-split)

assume \( \text{occ-tdestr}: \text{occ} \ h \ tdestr \)
then have \( \ast: \text{occ-ts} \ h \ (\text{Some} \ tsr) \) using ttt-tsr by simp

show \( \text{thesis} \)

apply (subst i-occ-p, simp add: lookup-def tap-def bind-def return-def execute-heap get-tsr exec-ifind IH-exec occ-tdestr)

using scno-tsr hosc-tsr by auto

next

assume \( \text{not-occ-tdestr}: \neg \text{occ} \ h \ tdestr \)

obtain \( s2' \) where get'-vr: \( \text{Ref.get} \ h' \ vr = \text{ITerm}(s2', \text{None}, \text{IVarD}) \)

using get-vr IH-hosc heap-only-stamp-ch-get-term by blast

have tsnextp-acyclic': \( \text{i-terms-acyclic} \ h' \ tsnextp \)

using IH-hosc heap-only-stamp-ch-terms-acyclic tsnextp-acyclic by blast

have scno-tsnextp': \( \bigwedge \text{tr} \in \text{i-terms-set} \ h' \ tsnextp \Rightarrow \text{stamp-current-not-occurs} \ time \ vr \ tr \ h' \)

unfolding stamp-current-not-occurs-def

proof (intro allI impI)

fix \( tr \ tr' \ s' \) is \( d \)

assume tr-terms'-tsnextp: \( tr \in \text{i-terms-set} \ h' \ tsnextp \)

and tr-clos'-tr: \( tr' \in \text{i-term-closure} \ h' \ (\text{Some} \ tr) \)

and get'-tr': \( \text{Ref.get} \ h' \ tr' = \text{ITerm}(s', \text{is}, \text{d}) \)

and s-eq'-time: \( s' = \text{Ref.get} \ h' \ time \)

have not-occurs'-tdestr: \( \neg \text{occ} \ h' \ tdestr \)

using hose-h'-h'-trs i-term-to-term-only-stamp-changed not-occ-tdestr tdestr-acyclic by auto

show \( \neg \text{occurs} \ (''(v''', \text{int}(\text{addr-of-ref} \ vr)) \ (\text{i-term-to-term-e} \ h' \ tr') \)

proof (rule case-split)

assume \( tr' \in \text{i-term-closure} \ h' \ (\text{Some} \ tdestr) \)

then show \( \text{thesis} \) using IH-scno[unfolded stamp-current-not-occurs-def]

not-occurs'-tdestr get'-tr' s-eq'-time by fast

next

assume tr'-not-clos'-tdestr: \( tr' \notin \text{i-term-closure} \ h' \ (\text{Some} \ tdestr) \)

then have get-tr': \( \text{Ref.get} \ h \ tr' = \text{ITerm}(s', \text{is}, \text{d}) \)

using IH-hosc get'-tr' heap-only-stamp-ch-diff-in-clos heap-only-stamp-ch-tr-sym by metis

moreover have tr-terms-tsnextp: \( tr \in \text{i-terms-set} \ h \ (\text{Some} \ tsr) \)

using IH-hosc tr-terms'-tsnextp get'-tsr heap-only-stamp-ch-terms-set i-terms-set-insert by blast
moreover have $tr' \cdot \text{clos-tr} \in i-term-closure h$ (Some $tr$)

using $IH$-hosc

heap-only-stamp-ch-closure $tr' \cdot \text{clos-tr}$ by blast

moreover have $s' = \text{Ref}.get h$ time

using heap-only-stamp-ch-get-nat hosc-h-h' - trs s-eq' - time by presburger

ultimately have $\sim \text{occ} h$ $tr'$

using terms-scno[unfolded stamp-current-not-occurs-def] by blast

then show ?thesis

by \{metis heap-only-stamp-ch-sym hosc-h-h' - trs i-term-closure-acyclic

i-term-to-term-only-stamp-changed i-terms-set-acyclic tr-clos'-tr

tr-terms'-tsnextp tsnextp-acyclic\}

qed

obtain $h''$ where

$IHn$-exec:

execute $(i-occ-p$ time (Some $vr$) $(\text{Inr} tsnextp))$ $h' = \text{Some}(\text{occ-ts} h'$

$tsnextp, h'')$ and

$IHn$-hosc: heap-only-stamp-changed $(i-terms-closure h' tsnextp)$ $h' h''$

and $IHN$-scno: stamp-current-not-occurs'-ts time $vr tsnextp h''$

using terms.hyps(1)[rule-format,

$OF - hosc-h-h' - trs get'\cdot vr trs-val scno-tsnextp' tsnextp-acyclic\}] by fast

have $i$-terms-closure $h$ $tsnextp \subseteq i$-terms-closure $h$ (Some $tsr$

by $(\text{simp add: get-ts}$ $i$-terms-set-insert)

then have $i$-terms-closure $h'$ $tsnextp \subseteq i$-terms-closure $h'$ (Some $tsr$

using heap-only-stamp-ch-terms-closure hosc-tsr by auto

then have heap-only-stamp-changed-ts $(Some$ $tsr)$ $h' h''$

using $IHn$-hosc

by $(\text{simp add: heap-only-stamp-ch-antimono}$

then have $\ast$: heap-only-stamp-changed-ts $(Some$ $tsr)$ $h h''$

using heap-only-stamp-ch-ts-trans hosc-tsr by blast

have get''-tsr: Ref.get $h''$ $tsr = \text{ITerms} (tthisr, tsnextp)$

using $IHn$-hosc get'-tsr heap-only-stamp-ch-get-terms by force

have $tsr$-acyclic$''$: $i$-terms-acyclic $h''$ (Some $tsr$

using $IHn$-hosc heap-only-stamp-ch-terms-acyclic using tsr-acyclic$' by$

blast

have $tdestr$-acyclic$'$: $i$-term-acyclic $h' (Some$ $tdestr$

using $IH$-hosc heap-only-stamp-ch-term-terms-acyclic tdestr-acyclic

by blast

have thistr-acyclic$'$: $i$-term-acyclic $h' (Some$ tthistr

using $IH$-hosc heap-only-stamp-ch-term-acyclic thistr-acyclic by blast

have ttt-tdestr$''$: $i$-term-to-term-e $h''$ thistr $= i$-term-to-term-e $h''$ tdestr

using $\ast$ $IH$-hosc $i$-term-to-term-only-stamp-changed tdestr-acyclic

$tdestr$-acyclic$'$

thistr-acyclic thistr-acyclic$' ttt-tdestr$ by auto

have ttt-tsdr$''$: $i$-terms-to-terms-e $h''$ (Some $tsr$) = $i$-term-to-term-e $h''$ tdestr $\# i$-terms-to-terms-e $h''$ $tsnextp

101
using get''-tsr i-terms-to-terms-e-step tsr-acyclic'' ttt-tdestr'' by presburger

have scno''-tsr: stamp-current-not-occurs'-ts time vr (Some tsr) h''
  unfolding stamp-current-not-occurs-def
proof (intro impI ballI allI)
  fix tr tr' s' d
  assume not-occ''-tsr: ¬?occ-ts h'' (Some tsr)
  and tr-tsr-term-set'': tr ∈ i-terms-set h'' (Some tsr)
  and tr'-clos''-tr: tr' ∈ i-term-closure h'' (Some tr)
  and get''-tr: Ref.get h'' tr' = ITerm (s', is, d)
  and s'-time'': s' = Ref.get h'' time
  have not-occ''-tdestr: ¬?occ-ts tsnextp
    using ttt-tsr'' not-occ''-tsr by force
  have not-occ''-tdestr: ¬?occ h' tdestr
    using not-occ''-tsr IHn-hosc i-term-to-term-only-stamp-changed
    tdestr-acyclic' ttt-tsr'' by simp

have tr'-acyclic'': i-term-acyclic h'' (Some tr')
using i-term-closure-acyclic i-terms-set-acyclic tr'-clos''-tr tr-tsr-term-set''
  tsr-acyclic'' by blast
then have tr'-acyclic': i-term-acyclic h' (Some tr')
using IHn-hosc heap-only-stamp-ch-sym heap-only-stamp-ch-term-acyclic
by blast
have ttt-h''-tr': i-term-to-term-e h' tr' = i-term-to-term-e h'' tr'
  using IHn-hosc i-term-to-term-only-stamp-changed tr'-acyclic' by
  presburger

have ttt-h''-tr': i-term-to-term-e h tr' = i-term-to-term-e h'' tr'
using * IH-hosc heap-only-stamp-ch-term-acyclic heap-only-stamp-ch-tr-sym
  i-term-to-term-only-stamp-changed tr'-acyclic' by blast

consider (a) tr' ∈ i-terms-closure h'' tsnextp |
  (b) tr' ∉ i-terms-closure h'' tsnextp and
    tr' ∈ i-term-closure h'' (Some tdestr) |
  (c) tr' ∉ i-terms-closure h'' tsnextp and
    tr' ∉ i-term-closure h'' (Some tdestr)
by fast
then show ¬?occ h'' tr'
proof (cases)
  case (a)
  then show ¬thesis
    using IHn-scn[unfolded stamp-current-not-occurs-def] not-occ''-tdestr
    s'-time'' get''-tr' by blast
next
  case (b)
  then have Ref.get h' tr' = ITerm (s', is, d)
    using IH-hosc get''-tr'
    heap-only-stamp-ch-closure heap-only-stamp-ch-get-term-nclos
    IHn-hosc heap-only-stamp-ch-terms-set by fastforce
moreover have \( tr' \in \text{i-term-closure } h' (\text{Some } \text{tdestr}) \)
using \( IHn-hosc b(2) \) heap-only-stamp-ch-closure by auto
moreover have \( s' = \text{Ref.get } h' \) time
using \( IHn-hosc \) heap-only-stamp-ch-get-nat \( s' \cdot \text{time}' \) by auto
ultimately have \( \neg \text{occurs } ("x", \text{int (addr-of-ref } vr)) \) (i-term-to-term-closure \( h' tr' \))
using \( IH-\text{scno\{unfolded stamp-current-not-occurs-def\}} \)
not-occ-tdestr by blast
then show \( ^?\text{thesis} \) using \( ttt-h'h''-\text{tr}'\) by simp
next
case \((c)\)
then have \( \text{Ref.get } h' tr' = \text{ITerm } (s', \text{is, d}) \)
using \( IHn-hosc \) get"-tr' heap-only-stamp-ch-get-term-nclos
heap-only-stamp-ch-terms-closure by fastforce
then have \( \text{Ref.get } h \text{ tr }' = \text{ITerm } (s', \text{is, d}) \)
using \( c(2) \)
* \( IH-\text{hosc} \) heap-only-stamp-ch-closure heap-only-stamp-ch-get-term-nclos
by force
moreover have \( tr \in \text{i-terms-set } h \) (Some \( \text{tsr} \))
using * heap-only-stamp-ch-terms-set tr-terms-set" by blast
moreover have \( tr' \in \text{i-term-closure } h \) (Some \( \text{tr} \))
using * heap-only-stamp-ch-closure tr'-clos"-tr by blast
moreover have \( s' = \text{Ref.get } h \) time
using * heap-only-stamp-ch-get-nat \( s' \cdot \text{time}' \) by presburger
ultimately have \( \neg ?\text{occ } h \text{ tr}' \)
using terms-scno\{unfolded stamp-current-not-occurs-def\}
by blast
then show \( ^?\text{thesis} \)
using \( ttt-h-h''-\text{tr}'\) by argo
qed
qed
have \( ?\text{occ-ts } h' \) tsnextp = \( ?\text{occ-ts } h' \) (Some \( \text{tsr} \))
using hosc-h-h'-trs i-term-to-term-only-stamp-changed not-occ-tdestr tdestr-acyclic ttt-ts' by auto
then have **: \( ?\text{occ-ts } h' \) tsnextp = \( ?\text{occ-ts } h \) (Some \( \text{tsr} \))
using i-terms-to-terms-only-stamp-changed
hosc-h-h'-trs i-terms-sublists.self terms-acyclic by presburger
show \( ^?\text{thesis} \)
apply (subst i-occ-p.simps,
  simp add: lookup-def tap-def bind-def return-def execute-heap
  get-ts exec-ifind IH-exec \( IHn\)-exec not-occ-tdestr ** scno"-tsr)
using * by simp
qed
qed
}
then show \( ^?\text{thesis} \)
using assms by presburger
qed
lemma i-occurs-sound:

fixes vr:: i-term ref
and tr:: i-term ref
and time:: nat ref
and td :: i-term-d
and h:: heap
and fun-term:: term
and s1:: nat
and s2:: nat

assumes acyclic: i-term-acyclic h (Some tr)
and get-tr: Ref.get h tr = I Term (s1, None, td)
and get-vr: Ref.get h vr = I Term (s2, None, IVarD)
and fun-term-val: Some (fun-term, h) = execute (i-term-to-term tr) h
and time-consistent: Ref.get h time ≥ i-maxstamp h (Some tr)

shows ∃ h'. execute (i-occurs time (Some vr) (Some tr)) h =
Some (occurs ("s"', int (addr-of-ref vr)) fun-term, h') ∧
heap-only-stamp-changed-tr tr (Ref.set time ((Ref.get h time) + 1) h) h' ∧
stamp-current-not-occurs' time vr tr h'

proof –

let ?h' = Ref.set time (Suc (Ref.get h time)) h
have honc: heap-only-nonterm-changed h ?h'
  using heap-only-nonterm-chI typerep-term-neq-nat typerep-terms-neq-nat
  by force
then have tr-acyclic': i-term-acyclic ?h' (Some tr)
  by (simp add: heap-only-nonterm-ch-term-acyclic[OF honc acyclic])
have tr-h': ∀ (x::i-term ref) y. Ref.get h x = y ⇒ Ref.get ?h' x = y
  using heap-only-nonterm-ch-get-term[OF honc] by fastforce
have tr-h: ∀ (x::i-term ref) y. Ref.get ?h' x = y ⇒ Ref.get h x = y
  using heap-only-nonterm-ch-get-term[OF honc[symmetric]] by fastforce
obtain fun-term' where
  fun-term-val': Some (fun-term', ?h') = execute (i-term-to-term tr) ?h'
  using i-term-to-term-value[OF tr-acyclic'] by metis
define r where
  r-val: r = occurs ("s"', int (addr-of-ref vr)) fun-term'
have sone: stamp-current-not-occurs time vr tr ?h'
  unfolding stamp-current-not-occurs-def
proof (intro allI impI, rule FalseE)

fix tr' s' is d
assume tr'-clos': tr' ∈ i-term-closure ?h' (Some tr)
and get-tr': Ref.get ?h' tr' = I Term (s', is, d)
and s'-time': s' = Ref.get ?h' time
have i-term-acyclic ?h' (Some tr')
  by (fact i-term-closure-acyclic[OF tr-acyclic' tr'-clos'])
then have tr'-acyclic: i-term-acyclic h (Some tr')
  using heap-only-nonterm-ch-term-acyclic[OF honc[symmetric]] by blast
have tr'-clos: tr' ∈ i-term-closure h (Some tr)
  using heap-only-nonterm-ch-closure honc tr'-clos' by auto
have mazstamp-tr': i-mazstamp h (Some tr') ≤ i-mazstamp h (Some tr)
using acyclic i-maxstamp-closure-trans tr'-clos by blast
have s' = Suc(Ref.get h time) using s'-time unfolding Ref.get-def Ref.set-def
by simp
moreover have s' ≤ i-maxstamp h (Some tr)
  using time-consistent maxstamp-tr' tr-h[OF get'-tr'] i-maxstamp-is-max
  acyclic tr'-clos by blast
then have s' ≤ Ref.get h time using time-consistent by fastforce
ultimately show False by force
qed
obtain h'' where
  res-exec: execute (i-occ-p time (Some vr) (Inl (Some tr))) ?h' = Some (r, h'')
and
  res-hosc: heap-only-stamp-changed-tr tr ?h' h'' and
  res-scno: stamp-current-not-occurs' time vr tr h''
using i-occ-p-sound[OF tr-acyclic' tr-h[OF get-tr] tr-h'][OF get-vr] fun-term-val'
    scno r-val by blast
have presv: i-term-structure-presv h ?h'
  by (simp add: heap-only-nonterm-ch-get-terms honc tr-h)
have i-term-to-term-e h tr = i-term-to-term-e ?h' tr
  using i-term-to-term-get-presv[OF acyclic presv] by blast
then have fun-term-eq: fun-term = fun-term'
  by (metis case-prod-conv fun-term-val fun-term-val' option.simps(5))
show ?thesis unfolding i-occurs-def
  by (simp add: bind-def lookup-def tap-def update-def execute-heap
    res-exec res-hosc res-scno r-val fun-term-eq)
qed
end