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

Contents

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[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[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:

$$f(g(x), x) \stackrel{?}{=} f(z, a)$$
$$z \stackrel{?}{=} y$$

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 occuring in a term t is denoted as $Var(t)$. 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 datastructures, 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]

$$
\begin{aligned}
\{t \stackrel{?}{=} t\} \uplus S &\implies S && \text{(Delete)} \\
\{f(\overline{t_n}) \stackrel{?}{=} f(\overline{u_n})\} \uplus S &\implies \{t_1 \stackrel{?}{=} u_1, \ldots, t_n \stackrel{?}{=} u_n\} \cup S && \text{(Decompose)} \\
\{t \stackrel{?}{=} x\} \uplus S &\implies \{x \stackrel{?}{=} t\} \cup S \text{ if } t \notin V && \text{(Orient)} \\
\{x \stackrel{?}{=} t\} \uplus S &\implies \{x \stackrel{?}{=} t\} \cup \{x \mapsto t\}(S) && \text{(Eliminate)} \\
& \quad\ \text{if } x \in Var(S) - Var(t)
\end{aligned}
$$

TRaAT gives a formulation in ML. Besides minor syntactical differences and raising an exception rather than returning None is it 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

```
solve((T(f,ts),T(g,us)) :: S, s)=
  if f = g 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.Heap:

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

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 typerep is an identifier associated with types that uniquely identifies the type. For types defined the usual way such as with **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

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,

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 l2 h INil = (l2, h)
| rev0 l2 h (ICons(v, lsr)) = (
    let ls = Ref.get h lsr in
    let (l2′, h′) = cons h v l2 in
    rev0 l2′ 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 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;
    l2′ ← cons v l2;
    rev0 l2′ 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 (?l2.0, INil) ⟹ rev0 ?l2.0 INil = return ?l2.0

rev0-dom (?l2.0, ICons (?v, ?lsr)) ⟹ rev0 ?l2.0 (ICons (?v, ?lsr)) = !?lsr ⤜ (λls. cons ?v ?l2.0 ⤜ (λl2′. rev0 l2′ ls))

The ⤜ 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 functionality. In this case the two theorems generated are

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

and

$$(\bigwedge x\ xa.\ \text{rev0-dom } (xa, x)) \Longrightarrow \text{rev0-dom } (?l2.0, \text{ ICons } (?v, ?lsr))$$

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 **partial_function** command[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 rev0 using this becomes

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 {
        ls ← !lsr;
        l2′ ← cons v l2;
        rev0 l2′ ls})

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

This generates some more useful theorems, rev0.simps becomes:
rev0 ?l2.0 ?l = (case ?l of INil ⇒ return ?l2.0 | ICons (v, lsr) ⇒ !lsr ⪢= (λls. cons v ?l2.0 ⪢= (λl2′. rev0 l2′ ls)))

Note that there is no guard this time. Induction rules for fixpoint induction 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 $\Rightarrow$ heap $\Rightarrow$ ($'a \times$ 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 $\Rightarrow$ heap $\Rightarrow$ heap $\Rightarrow$ $'a \Rightarrow$ 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

```
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)

type-synonym i-termP = i-term ref option
type-synonym i-termsP = i-terms ref option
```

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 needs 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:

```
inductive i-term-acyclic:: heap ⇒ i-termP ⇒ bool and
  i-terms-acyclic:: heap ⇒ i-termsP ⇒ bool where
t-acyclic-nil: i-term-acyclic - None |
t-acyclic-step-link:
  i-term-acyclic h t ⟹
  Ref.get h tref = ITerm(-, t, IVarD) ⟹
  i-term-acyclic h (Some tref) |
t-acyclic-step-ITerm:
  i-terms-acyclic h tsref ⟹
  Ref.get h tref = ITerm(-, None, ITermD(-, tsref)) ⟹
```

i-term-acyclic h (Some tref) |
ts-acyclic-nil: i-terms-acyclic - None |
ts-acyclic-step-ITerms:
i-terms-acyclic h ts2ref $\Longrightarrow$
i-term-acyclic h (Some tref) $\Longrightarrow$
Ref.get h tsref = ITerms (tref, ts2ref) $\Longrightarrow$
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.

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

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

abbreviation i-term-closures **where**
i-term-closures h trs $\equiv$ (*$\bigcup$ (i-term-closure h ‘ Some ‘ trs)*)
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 ≡ (∗⋃(i-term-sublists h ‘ i-term-closure h tr)∗)
(⋃tr′∈i-term-closure h tp. i-term-sublists h tr′)

abbreviation i-terms-closure-sublists **where**
i-terms-closure-sublists h tsp ≡ (∗⋃(i-term-sublists h ‘ i-terms-closure h tsp)∗)
i-terms-sublists h tsp ∪ (⋃tr∈i-terms-closure h tsp. i-term-sublists h tr)

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.

abbreviation i-term-closures **where**
i-term-closures h trs ≡ UNION (Some ‘ trs) (i-term-closure h)

abbreviation i-terms-closure **where**
i-terms-closure h tsp ≡ i-term-closures h (i-terms-set h tsp)

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

abbreviation i-term-closure-sublists **where**
i-term-closure-sublists h tp ≡ (⋃tr′∈i-term-closure h tp. i-term-sublists h tr′)

abbreviation i-terms-closure-sublists **where**
i-terms-closure-sublists h tsp ≡ i-terms-sublists h tsp ∪ (⋃tr∈i-terms-closure 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.

lemma acyclic-closure-ch-stamp-inductc′ [consumes 1,
case-names var link args 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.
Ref.get h tr = ITerm(s, None, IVarD) $\Longrightarrow$
P1 h trs tr
and link-case: $\bigwedge$h trs tr isr s.
($\bigwedge$t2r h′ trs′.
trs ⊆ trs′ $\Longrightarrow$
heap-only-stamp-changed trs′ h h′ $\Longrightarrow$
t2r ∈ i-term-closure h (Some isr) $\Longrightarrow$
P1 h′ trs′ t2r) $\Longrightarrow$
Ref.get h tr = ITerm(s, Some isr, IVarD) $\Longrightarrow$
P1 h trs tr
and args-case: $\bigwedge$h trs tr tsp s f.
($\bigwedge$h′ trs′.
trs ⊆ trs′ $\Longrightarrow$
heap-only-stamp-changed trs′ h h′ $\Longrightarrow$
P2 h′ trs′ tsp) $\Longrightarrow$
($\bigwedge$h′ trs′ t2r0 t2r.
trs ⊆ trs′ $\Longrightarrow$
heap-only-stamp-changed trs′ h h′ $\Longrightarrow$
t2r ∈ i-term-closure h (Some t2r0) $\Longrightarrow$
t2r0 ∈ i-terms-set h tsp $\Longrightarrow$
P1 h′ trs′ t2r) $\Longrightarrow$
Ref.get h tr = ITerm(s, None, ITermD(f, tsp)) $\Longrightarrow$
P1 h trs tr
and terms-nil-case: $\bigwedge$h trs. P2 h trs None
and terms-case: $\bigwedge$h trs tr tsr tsnextp.
($\bigwedge$h′ trs′.
trs ⊆ trs′ $\Longrightarrow$
heap-only-stamp-changed trs′ h h′ $\Longrightarrow$
P2 h′ trs′ tsnextp) $\Longrightarrow$
($\bigwedge$h′ trs′ t2r.
trs ⊆ trs′ $\Longrightarrow$
heap-only-stamp-changed trs′ h h′ $\Longrightarrow$
t2r ∈ i-term-closure h (Some tr) $\Longrightarrow$
P1 h′ trs′ t2r) $\Longrightarrow$
Ref.get h tsr = ITerms (tr, tsnextp) $\Longrightarrow$
P2 h trs (Some tsr)
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 occuring, 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

[1] L. Bulwahn, A. Krauss, F. Haftmann, L. Erkök, and J. Matthews. Imperative Functional Programming with Isabelle/HOL. In *Proceedings of the 21st International Conference on Theorem Proving in Higher Order Logics*, TPHOLs '08, pages 134–149, Berlin, Heidelberg, 2008. Springer-Verlag.

[2] A. Krauss. Defining Recursive Functions in Isabelle/HOL. http://isabelle.in.tum.de/dist/Isabelle2017/doc/functions.pdf, 08 2017.

[3] A. Martelli and U. Montanari. An efficient unification algorithm. *ACM Trans. Program. Lang. Syst.*, 4(2):258–282, Apr. 1982.

[4] M. Wenzel and T. Munchen. The Isabelle/Isar Reference Manual. http://isabelle.in.tum.de/dist/Isabelle2017/doc/isar-ref.pdf, 08 2017.

Appendices

A Isabelle theory

A.1 Miscellaneous theory

theory *Unification-Misc*
  imports *Main*

begin

*zip*ping two lists and retrieving one of them back by mapping *fst* or *snd* results in the original list, possibly truncated

```
lemma sublist-map-fst-zip:
  fixes xs:: 'a list
    and ys:: 'a list
  obtains xss
    where (map fst (zip xs ys)) @ xss = xs
  by (induct xs ys rule:list-induct2', auto)

lemma sublist-map-snd-zip:
  fixes xs:: 'a list
    and ys:: 'a list
  obtains yss
    where (map snd (zip xs ys)) @ yss = ys
  by (induct xs ys rule:list-induct2', auto)

end
```

A.2 Functional version of algorithm

```
theory Unification-Functional
  imports Main
     Unification-Misc
begin

type-synonym vname = string × int

datatype term =
    V vname
  | T string × term list

type-synonym subst = (vname × term) list

definition indom :: vname ⇒ subst ⇒ bool where
  indom x s = list-ex (λ(y, -). x = y) s

fun app :: subst ⇒ vname ⇒ term where
  app ((y,t)#s) x = (if x = y then t else app s x) |
  app [] - = undefined

fun lift :: subst ⇒ term ⇒ term where
  lift s (V x) = (if indom x s then app s x else V x)
| lift s (T(f,ts)) = T(f, map (lift s) ts)

fun occurs :: vname ⇒ term ⇒ bool where
  occurs x (V y) = (x = y)
| occurs x (T(-,ts)) = list-ex (occurs x) ts
```

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*

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.IH* **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: (map snd (zip ts us)) @ usp = us by (fact sublist-map-snd-zip)
  have fun-count [T(g, ?trunc-ts)] + fun-count tsp = 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, ?trunc-us)] + fun-count usp = 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 n-fun [(T (g, ?trunc-ts), T (g, ?trunc-us))] + fun-count tsp +
fun-count usp =
      fun-count [T(g, ts)] + fun-count [T(g, us)]
    by (simp add: n-fun-def)
  then have n-fun ((T (g, ?trunc-ts), T (g, ?trunc-us)) # S) + fun-count tsp +
fun-count usp =
      n-fun ((T (g, ts), T (g, us)) # S)
    using n-fun-def n-fun-concat by simp
  from this and trunc-sum show ?thesis by simp
qed

private abbreviation (noprint) liftmap v t S′ ≡
  map (λ(t1, t2). (lift [(v, t)] t1, lift [(v, t)] t2)) S′

private lemma lift-elims:
  fixes x :: vname
    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 ∃ t1∈set ts. 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 f ts)
  then show ?case
   by (metis (mono-tags, lifting) Bex-set-list-ex imageE 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
```

then show *?thesis* **unfolding** *vars-def* **by** *simp*
next
assume *not-occurs-v-st*: $\neg$*occurs v 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 *?thesis*
by (*simp add*: *vars-def*)
qed
ultimately have $v \in$ *vars* [*st*] $\cup$ *vars* [*tx*] $\wedge$ $v \neq x$ **by** *simp*
}
ultimately show *?case* **by** *blast*
qed

private lemma *n-var-elim*:
fixes *x st S*
assumes $\neg$ *occurs x st*
shows *n-var* (*liftmap x st S*) < *n-var* ((*V x*, *st*) # *S*)
proof −
have $\bigwedge$*l f*. *map fst* (*map* (λ(*t1*, *t2*). (*f t1*, *f t2*)) *l*) = *map f* (*map fst l*)
by (*simp add*: *case-prod-unfold*)
moreover have $\bigwedge$*l f*. *map snd* (*map* (λ(*t1*, *t2*). (*f t1*, *f t2*)) *l*) = *map f* (*map snd l*)
by (*simp add*: *case-prod-unfold*)
ultimately have *lhs-split*: *vars-eqs* (*liftmap x st S*) =
vars (*map* (*lift* [(*x*,*st*)]) (*map fst S*)) $\cup$ *vars* (*map* (*lift* [(*x*,*st*)]) (*map snd S*))
unfolding *vars-eqs-def* **by** *metis*

have *vars-eqs* ((*V x*, *st*) # *S*) = *vars-eqs* [(*V x*, *st*)] $\cup$ *vars-eqs S*
using *vars-eqs-concat*
by (*metis append-Cons self-append-conv2*)
moreover have *vars-eqs* [(*V x*, *st*)] = {*x*} $\cup$ *vars* [*st*]
unfolding *vars-eqs-def* **using** *vars-def occurs.simps*(*1*) **by** *force*
ultimately have *rhs-eq1*: *vars-eqs* ((*V x*, *st*) # *S*) = {*x*} $\cup$ *vars* [*st*] $\cup$ *vars-eqs S*
by *presburger*
then have *rhs-eq2*:
vars-eqs ((*V x*, *st*) # *S*) = {*x*} $\cup$ *vars* [*st*] $\cup$ *vars* (*map fst S*) $\cup$ *vars* (*map snd S*)
unfolding *vars-eqs-def*
by (*simp add*: *sup.assoc*)

from *this lhs-split vars-elim assms*
have *vars-eqs* (*liftmap x st S*) $\subseteq$ *vars* [*st*] $\cup$ *vars-eqs S* $\wedge$
$x \notin$ *vars-eqs* (*liftmap x st S*)
using *vars-concat vars-eqs-def* **by** (*metis map-append*)
moreover have $x \in$ *vars-eqs* ((*V x*, *st*) # *S*)
by (*simp add*: *rhs-eq2*)

```
  ultimately have vars-eqs (liftmap x st S) ⊂ vars-eqs ((V x, st) # S)
    using rhs-eq1 by blast
  then show ?thesis using vars-eqs-psubset-n-var-lt by blast
qed

function (sequential) solve :: (term × term) list × subst ⇒ subst option
  and elim :: vname × term × (term × term) list × subst ⇒ subst option where
  solve([], s) = Some s
| solve((V x, t) # S, s) = (
    if V x = t then solve(S,s) else elim(x,t,S,s))
| solve((t, V x) # S, s) = elim(x,t,S,s)
| solve((T(f,ts),T(g,us)) # 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 (λ (t1,t2). (xt t1, xt t2)) S,
        (x,t) # (map (λ (y,u). (y, xt u)) s)))
  by pat-completeness auto
termination proof (
    relation
       (λXX. case XX of Inl(l,-) ⇒ n-var l | Inr(x,t,S,-) ⇒ n-var ((V x, t)#S))
<*mlex*>
       (λXX. case XX of Inl(l,-) ⇒ n-fun l | Inr(x,t,S,-) ⇒ n-fun ((V x, t)#S))
<*mlex*>
        (λXX. case XX of Inl(l,-) ⇒ size l | Inr(x,t,S,-) ⇒ size ((V x, t)#S))
<*mlex*>
      (λXX. case XX of Inl(l,-) ⇒ 1 | Inr(x,t,S,-) ⇒ 0) <*mlex*>
      {},
    auto simp add: wf-mlex mlex-less mlex-prod-def)
  have ⋀v S. vars-eqs S ⊆ vars-eqs ((V v, V v)#S)
    using vars-eqs-def vars-def by force
  then show ⋀a b S. ¬ n-var S < n-var ((V (a, b), V (a, b)) # S) ⟹
      n-var S = n-var ((V (a, b), V (a, b)) # S)
    using vars-eqs-subset-n-var-le by (simp add: nat-less-le)

  show ⋀a b S. ¬ n-var S < n-var ((V (a, b), V (a, b)) # S) ⟹
           n-fun S ≠ n-fun ((V (a, b), V (a, b)) # S) ⟹
           n-fun S < n-fun ((V (a, b), V (a, b)) # S)
    using n-fun-def by simp

  have ⋀tx v. vars-eqs [(V v, T tx)] = vars-eqs [(T tx, V v)]
    using vars-eqs-def
    by (simp add: sup-commute)
  then have ⋀tx v S. vars-eqs ((V v, T tx) # S) = vars-eqs ((T tx, V v) # S)
    using vars-eqs-concat
    by (metis append-Cons self-append-conv2)
  then have ⋀a b v S. n-var ((V v, T (a, b)) # S) = n-var ((T (a, b), V v) #
```

$S)$
unfolding *n-var-def vars-eqs-def*
by *presburger*
then show $\bigwedge a\ b\ aa\ ba\ S.$
$\neg$ *n-var* $((V\ (aa,\ ba),\ T\ (a,\ b))\ \#\ S) <$ *n-var* $((T\ (a,\ b),\ V\ (aa,\ ba))\ \#\ S)$
$\Longrightarrow$
n-var $((V\ (aa,\ ba),\ T\ (a,\ b))\ \#\ S) =$ *n-var* $((T\ (a,\ b),\ V\ (aa,\ ba))\ \#\ S)$
by *simp*

show $\bigwedge a\ b\ aa\ ba\ S.$
$\neg$ *n-var* $((V\ (aa,\ ba),\ T\ (a,\ b))\ \#\ S) <$ *n-var* $((T\ (a,\ b),\ V\ (aa,\ ba))\ \#\ S)$
$\Longrightarrow$
n-fun $((V\ (aa,\ ba),\ T\ (a,\ b))\ \#\ S) \neq$ *n-fun* $((T\ (a,\ b),\ V\ (aa,\ ba))\ \#\ S) \Longrightarrow$
n-fun $((V\ (aa,\ ba),\ T\ (a,\ b))\ \#\ S) <$ *n-fun* $((T\ (a,\ b),\ V\ (aa,\ ba))\ \#\ S)$
by (*simp add*: *n-fun-def*)

show $\bigwedge ts\ g\ us\ S.$ $\neg$ *n-var* (*zip ts us* @ S) $<$ *n-var* $((T\ (g,\ ts),\ T\ (g,\ us))\ \#\ S)$
$\Longrightarrow$
n-var (*zip ts us* @ S) $=$ *n-var* $((T\ (g,\ ts),\ T\ (g,\ us))\ \#\ S)$
using *vars-eqs-nest-subset vars-eqs-subset-n-var-le le-neq-implies-less* **by** *meson*

have *n-fun-nested-gt*: $\bigwedge ts\ g\ us\ S.$ *n-fun* (*zip ts us* @ S) $<$ *n-fun* $((T\ (g,\ ts),\ T\ (g,\ us))\ \#\ S)$
using *n-fun-nest-head*
by (*metis add-leD1 le-neq-implies-less add-2-eq-Suc' leD less-Suc-eq*)
show $\bigwedge ts\ g\ us\ S.$
$\neg$ *n-var* (*zip ts us* @ S) $<$ *n-var* $((T\ (g,\ ts),\ T\ (g,\ us))\ \#\ S) \Longrightarrow$
$\neg$ *n-fun* (*zip ts us* @ S) $<$ *n-fun* $((T\ (g,\ ts),\ T\ (g,\ us))\ \#\ S) \Longrightarrow$
n-fun (*zip ts us* @ S) $=$ *n-fun* $((T\ (g,\ ts),\ T\ (g,\ us))\ \#\ S)$
using *n-fun-nested-gt* **by** *meson*

show $\bigwedge ts\ g\ us\ S.$
$\neg$ *n-var* (*zip ts us* @ S) $<$ *n-var* $((T\ (g,\ ts),\ T\ (g,\ us))\ \#\ S) \Longrightarrow$
$\neg$ *n-fun* (*zip ts us* @ S) $<$ *n-fun* $((T\ (g,\ ts),\ T\ (g,\ us))\ \#\ S) \Longrightarrow$
min (*length ts*) (*length us*) $=$ *0*
using *n-fun-nested-gt* **by** *blast*

show $\bigwedge x\ t\ S.$
$\neg$ *occurs* $x\ t \Longrightarrow$
$\neg$ *n-var* (*liftmap* $x\ t\ S$) $<$ *n-var* $((V\ x,\ t)\ \#\ S) \Longrightarrow$
n-var (*liftmap* $x\ t\ S$) $=$ *n-var* $((V\ x,\ t)\ \#\ S)$
using *n-var-elim leD linorder-neqE-nat* **by** *blast*

show $\bigwedge x\ t\ S.$
$\neg$ *occurs* $x\ t \Longrightarrow$
$\neg$ *n-var* (*liftmap* $x\ t\ S$) $<$ *n-var* $((V\ x,\ t)\ \#\ S) \Longrightarrow$
n-fun (*liftmap* $x\ t\ S$) $\neq$ *n-fun* $((V\ x,\ t)\ \#\ S) \Longrightarrow$
n-fun (*liftmap* $x\ t\ S$) $<$ *n-fun* $((V\ x,\ t)\ \#\ S)$
using *n-var-elim* **by** *simp*

qed

end

end

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*: *TYPEREP*(*i-term*) ≠ *TYPEREP*(*i-terms*)
  using *typerep-i-terms-def typerep-i-term-def* **by** *fastforce*

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

lemma *typerep-terms-neq-nat*: *TYPEREP*(*i-terms*) ≠ *TYPEREP*(*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*)

lemma *get-ITerm-args-iff-ex*:
shows (*get-ITerm-args t* = *tsp*) $\longleftrightarrow$
(∃ *s is d*. *t* = *ITerm*(*s*, *is*, *d*) ∧ (
(*tsp* = *None* ∧ *d* = *IVarD*) ∨
(∃ *f*. *d* = *ITermD*(*f*, *tsp*))))
proof –
obtain *s is d* **where** *t* = *ITerm*(*s*, *is*, *d*)
by (*metis i-term.exhaust surj-pair*)
then show *?thesis* **unfolding** *get-ITerm-args-def*
by (*cases d rule*: *i-term-d.exhaust*; *force*)
qed

type-synonym *i-termP* = *i-term ref option*
type-synonym *i-termsP* = *i-terms ref option*

inductive *i-term-acyclic*:: *heap* ⇒ *i-termP* ⇒ *bool* **and**
i-terms-acyclic:: *heap* ⇒ *i-termsP* ⇒ *bool* **where**
t-acyclic-nil: *i-term-acyclic* - *None* |
t-acyclic-step-link:
i-term-acyclic h t $\Longrightarrow$
Ref.get h tref = *ITerm*(-, *t*, *IVarD*) $\Longrightarrow$
i-term-acyclic h (*Some tref*) |
t-acyclic-step-ITerm:
i-terms-acyclic h tsref $\Longrightarrow$
Ref.get h tref = *ITerm*(-, *None*, *ITermD*(-, *tsref*)) $\Longrightarrow$
i-term-acyclic h (*Some tref*) |
ts-acyclic-nil: *i-terms-acyclic* - *None* |
ts-acyclic-step-ITerms:
i-terms-acyclic h ts2ref $\Longrightarrow$
i-term-acyclic h (*Some tref*) $\Longrightarrow$
Ref.get h tsref = *ITerms* (*tref*, *ts2ref*) $\Longrightarrow$
i-terms-acyclic h (*Some tsref*)

lemma *acyclic-terms-term-simp* [*simp*]:
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*))
shows *i-terms-acyclic h termsp*
proof –
consider
(*a*) *h'* **where** *h* = *h'* ∧ (*Some tr*) = *None* |
(*b*) *h' t tref s'* **where**
h' = *h* ∧ (*Some tr*) = *Some tref* ∧
i-term-acyclic h t ∧

```
      Ref.get h tref = ITerm (s', t, IVarD) |
    (c) h' tsref tref s' f' where
      h' = h ∧ (Some tr) = Some tref ∧
        i-terms-acyclic h tsref ∧
        Ref.get h tref = ITerm (s', None, ITermD (f', tsref))
    using i-term-acyclic.simps acyclic by fast
  then show ?thesis using get-tr
    by (cases, fastforce+)
qed

lemma acyclic-terms-terms-simp [simp]:
  fixes tsr:: i-terms ref
    and tthis:: i-term ref
    and tnext:: i-termsP
    and h:: heap
  assumes acyclic: i-terms-acyclic h (Some tsr)
    and get-termsr: Ref.get h tsr = ITerms (tthis, tnext)
  shows i-terms-acyclic h tnext
proof −
  consider (a) (Some tsr) = None |
          (b) tref where
             i-term-acyclic h (Some tref) ∧
             Ref.get h tsr = ITerms (tref, None) |
          (c) ts2ref tref where
             i-terms-acyclic h ts2ref ∧
             i-term-acyclic h (Some tref) ∧
             Ref.get h tsr = ITerms (tref, ts2ref)
    using acyclic i-terms-acyclic.simps[of h Some tsr] by fast
  then show ?thesis using get-termsr ts-acyclic-nil
    by (cases, fastforce+)
qed

lemma acyclic-term-link-simp:
  fixes tr:: i-term ref
    and tr':: i-term ref
    and d:: i-term-d
    and s:: nat
    and h:: heap
  assumes acyclic: i-term-acyclic h (Some tr)
    and get-tr: Ref.get h tr = ITerm (s, Some tr', d)
  shows i-term-acyclic h (Some tr')
proof −
  consider (a) (Some tr) = None |
          (b) t s' where
             i-term-acyclic h t ∧
             Ref.get h tr = ITerm (s', t, IVarD) |
          (c) tsref s' f where
             i-terms-acyclic h tsref ∧
             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: TYPEREP('a) ≠ TYPEREP(i-term)
    and ne-iterms: TYPEREP('a) ≠ TYPEREP(i-terms)
  shows i-term-acyclic (Ref.set r v h) (Some tr)
  using acyclic
proof (induction h Some tr
    arbitrary: tr
    taking: λh 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
```

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* (*Ref.set r v h*) *tsref* = *ITerms* (*tref*, *ts2ref*)
    using *ne-iterms Ref.get-set-neq Ref.noteq-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*)
  shows *i-term-acyclic* (*Ref.set r v h*) (*Some tr*)
  using *acyclic get-r*
proof (*induction h Some tr*
    arbitrary: *tr*
    taking: *λh tsp. Ref.get h r* = *ITerm* (*s*, *is*, *IVarD*) ⟶ *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 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* ⟹
      Ref.get h r = *ITerm* (*s*, *is*, *IVarD*) ⟹
      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.unequal* **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-acyclic-step-ITerm h tsref tr s1 f*)
then have *case-get-r*: *Ref.get h r = ITerm (s, is, IVarD)*
and *get-tr*: *Ref.get h tr = ITerm (s1, None, ITermD (f, tsref))*
and *get-tsref*: *i-terms-acyclic (Ref.set r v h) tsref*
by *blast+*
have $tr \neq r$
using *get-tr case-get-r* **by** *force*
then have $\exists$ *s0. Ref.get (Ref.set r v h) tr = ITerm (s0, None, ITermD (f, tsref))*
using *get-tr* **by** *simp*
then show *?case*
using *i-term-acyclic-i-terms-acyclic.t-acyclic-step-ITerm*
get-tsref **by** *blast*
next
case (*ts-acyclic-nil h*)
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*)
then have *get-tsref*: *Ref.get h tsref = ITerms (tref, ts2ref)*
and *IH1*: *Ref.get h r = ITerm (s, is, IVarD)* $\Longrightarrow$ *i-terms-acyclic (Ref.set r v h) ts2ref*
and *IH2*: *Ref.get h r = ITerm (s, is, IVarD)* $\Longrightarrow$ *i-term-acyclic (Ref.set r v h) (Some tref)*
by *blast+*
have *Ref.get (Ref.set r v h) tsref = ITerms (tref, ts2ref)*
using *get-tsref typerep-term-neq-terms Ref.get-set-neq Ref.noteq-def* **by** *metis*
then show *?case*
using *i-term-acyclic-i-terms-acyclic.ts-acyclic-step-ITerms*
IH1 IH2 **by** *blast*
qed

lemma *i-terms-acyclic-induct* [*consumes 1*,
case-names ts-acyclic-nil ts-acyclic-step]:
fixes *h* :: *heap*
and *tsp* :: *i-terms ref option*
and *P* :: *heap* $\Rightarrow$ *i-terms ref option* $\Rightarrow$ *bool*
assumes *acyclic*: *i-terms-acyclic h tsp*
and $\bigwedge$*h. P h None*
and $\bigwedge$*h ts2ref tref tsref.*
i-terms-acyclic h ts2ref $\Longrightarrow$
P h ts2ref $\Longrightarrow$ *i-term-acyclic h (Some tref)* $\Longrightarrow$
Ref.get h tsref = ITerms (tref, ts2ref) $\Longrightarrow$
P h (Some tsref)
shows *P h tsp*
using *assms ts-acyclic-nil*
by (*induction taking*: λ*h tp. True rule*: *i-term-acyclic-i-terms-acyclic.inducts(2)*, *blast+*)

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

lemma *i-terms-sublists-mNone*:
  fixes *h*:: *heap*
    and *tsp*:: *i-termsP*
  assumes *i-terms-acyclic h tsp*
  shows *None* ∈ *i-terms-sublists h tsp*
  using *assms*
proof (*induction rule*: *i-terms-acyclic-induct*)
  case (*ts-acyclic-nil uy*)
  then show *?case*
    by (*simp add*: *i-terms-sublists.intros*(*2*))
next
  case (*ts-acyclic-step h tnext tref tsref*)
  have *i-terms-acyclic h tnext* ⟹
    Ref.get h tsref = *ITerms* (*tref*, *tnext*) ⟹
     None ∈ *i-terms-sublists h* (*Some tsref*)
    using *ts-acyclic-step.IH i-terms-sublists.intros*
    by (*induction rule*: *i-terms-sublists.induct*, *blast+*)
  then show *?case* **using** *ts-acyclic-step* **by** *blast*
qed

lemma *i-terms-sublists-None-om*:
  fixes *h*:: *heap*
  shows *i-terms-sublists h None* = {*None*}
proof −
  {
    fix *tsp ts2p*
    have *ts2p* ∈ *i-terms-sublists h tsp* ⟹ ∃ *tr*. (*Some tr*) = *ts2p* ⟹ *tsp* ≠ *None*
      by (*induction rule*: *i-terms-sublists.induct*, *blast+*)
  }
  then show *?thesis*
    using *i-terms-sublists.intros*(*2*) *these-empty-eq* **by** *fastforce*
qed

lemma *i-terms-sublists-subset*:
  fixes *h*:: *heap*
    and *tsr* **and** *tr*
  assumes *Ref.get h tsr* = *ITerms* (*tr*, *tsp*)
  shows *i-terms-sublists h tsp* ⊆ *i-terms-sublists h* (*Some tsr*)
proof −
  {
    fix *ts2p*
    have *ts2p* ∈ *i-terms-sublists h tsp* ⟹ *ts2p* ∈ *i-terms-sublists h* (*Some tsr*)

```
    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(1) 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' tthis 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*.
∃ *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*

where *Some tsr* = *tsp*
and *Ref.get h tsr* = *ITerms*(*tthisr*, *tsnextp*)
by (*metis i-terms.exhaust old.prod.exhaust option.exhaust*)
then have *tthisr* ∈ *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*
by (*simp add*: *i-terms-set-insert ts-acyclic-step.IH ts-acyclic-step.hyps*(*3*))
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*: $\bigwedge h\ tr\ isr\ s.$
$P\ h\ (Some\ isr) \Longrightarrow$
$Ref.get\ h\ tr = ITerm(s,\ Some\ isr,\ IVarD) \Longrightarrow$
$P\ h\ (Some\ tr)$
and *args-case*: $\bigwedge h\ tr\ tsp\ s\ f.$
$(\forall\ tr2 \in i\text{-}terms\text{-}set\ h\ tsp.\ P\ h\ (Some\ tr2)) \Longrightarrow$
$i\text{-}terms\text{-}acyclic\ h\ tsp \Longrightarrow$
$Ref.get\ h\ tr = ITerm(s,\ None,\ ITermD(f,\ tsp)) \Longrightarrow$
$P\ h\ (Some\ tr)$
shows *P h tp*
using *acyclic*
proof (*induction h tp*
taking: $\lambda h\ tp.\ \forall\ tr2 \in i\text{-}terms\text{-}set\ h\ tp.\ P\ h\ (Some\ tr2)$
rule: *i-term-acyclic-i-terms-acyclic.inducts*(*1*))
case (*t-acyclic-nil h*)
then show *?case* **by** (*fact nil-case*)
next
case (*t-acyclic-step-link h t tref uv*)
then show *?case* **using** *var-case link-case*
by (*metis not-None-eq*)
next
case (*t-acyclic-step-ITerm h tsref tref uw ux*)
then show *?case* **using** *args-case* **by** *blast*
next
case (*ts-acyclic-nil h*)
then show *?case* **using** *i-terms-set-None-empty* **by** *blast*
next
case (*ts-acyclic-step-ITerms h ts2ref tref tsref*)
then show *?case* **using** *i-terms-set-insert* **by** *fast*
qed

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

```
proof -
  {
    fix tp
    have i-term-acyclic h tp ⟹ tp = Some tr ⟹ 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 ∈ 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' ∈ 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
```

then show *?thesis*
proof (*cases*)
case *a*
then show *?thesis* **by** *simp*
next
case *b*
then show *?thesis* **using** ∗∗ **by** *simp*
next
case *c*
then have $Some\ tsr' = Some\ tsref$
and *i-term-acyclic h* (*Some tref*)
and *Ref.get h tsref* = *ITerms* (*tref*, *ts2ref*)
by *blast+*
then show *?thesis* **using** ∗∗ **by** *simp*
qed
qed

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

abbreviation *i-term-closures* **where**
i-term-closures h trs ≡ *UNION* (*Some* ‘ *trs*) (*i-term-closure h*)

abbreviation *i-terms-closure* **where**
i-terms-closure h tsp ≡ *i-term-closures h* (*i-terms-set h tsp*)

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

abbreviation *i-term-closure-sublists* **where**
i-term-closure-sublists h tp ≡ ($\bigcup$ *tr'*∈*i-term-closure h tp*. *i-term-sublists h tr'*)

abbreviation *i-terms-closure-sublists* **where**
i-terms-closure-sublists h tsp ≡ *i-terms-sublists h tsp* ∪ ($\bigcup$ *tr*∈*i-terms-closure h tsp*. *i-term-sublists h tr*)

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

```
    have tr ∈ i-term-closure h tp ⟹ tp = None ⟹ 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-term-closure h (Some tr) = {tr}
proof −
  {
    fix tp tr x
    have x ∈ i-term-closure h tp ⟹
      tp = Some tr ⟹ Ref.get h tr = ITerm (s, None, IVarD) ⟹ x = tr
      by (induction rule: i-term-closure.induct, fastforce+)
  }
  then show ?thesis using 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-d
    and s:: nat
    and h:: heap
  assumes Ref.get h tr = ITerm (s, Some isr, d)
  shows i-term-closure h (Some tr) = insert tr (i-term-closure h (Some isr))
proof −
  {
    fix tp x
    have x ∈ i-term-closure h tp ⟹
      tp = Some tr ⟹
      Ref.get h tr = ITerm (s, Some isr, d) ⟹
      x = tr ∨ x ∈ i-term-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
```

```
      then show ?thesis using 2 i-term-closure.intros(2) by blast
    qed
  next
    case (3 tr′ s′ f tsp tr2)
    then have tr ≠ tr′ by fastforce
    then show ?case using 3 i-term-closure.intros(3) by blast
  qed
 }
 moreover {
  fix x
  assume x ∈ insert tr (i-term-closure h (Some isr))
  then consider (a) x = tr | (b) x ∈ i-term-closure h (Some isr)
    by blast
  then have x ∈ 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 show ?thesis proof (induction rule: i-term-closure.induct)
      case (1 x)
      then show ?case
        using assms i-term-closure.intros(1) i-term-closure.intros(2) by blast
    next
      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
  qed
 }
 ultimately show ?thesis using assms by fast
qed

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

```
    have x ∈ i-term-closure h tp ⟹
      tp = Some tr ⟹
      Ref.get h tr = ITerm (s, None, ITermD(f, tsp)) ⟹
      x = tr ∨ (∃ t2r ∈ i-terms-set h tsp. x ∈ i-term-closure h (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 ⋀tr''. tr2 ∈ i-term-closure h tr'' ∨ tr' ∉ i-term-closure h 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 (Some tr) ⊆ insert tr (i-terms-closure h tsp)
    by (simp add: assms subsetI)
  moreover {
    fix x
    assume x ∈ insert tr (i-terms-closure h tsp)
    then consider (a) x = tr | (b) ∃ t2r ∈ i-terms-set h tsp. x ∈ i-term-closure h
(Some t2r)
      by blast
    then have x ∈ 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 ∈ i-terms-set h tsp ∧ x ∈ i-term-closure h (Some
t2r)
        by blast
      moreover have x ∈ i-term-closure h (Some t2r) ⟹
        t2r ∈ i-terms-set h tsp ⟹
        x ∈ 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
```

```
      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.
          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 ⋀ *t2r*.
          t2r ∈ *i-term-closure h* (*Some tr*) ⟹
          Some t2r ≠ *Some tr* ⟹
          P h (*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-inductc* [*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 *var-case*: $\bigwedge h\ tr\ s.$
    Ref.get h tr = *ITerm*(*s*, *None*, *IVarD*) $\Longrightarrow$
    P h (*Some tr*)
and *link-case*: $\bigwedge h\ tr\ isr\ s.$
    ($\bigwedge t2r.$ *t2r* $\in$ *i-term-closure h* (*Some isr*) $\Longrightarrow$ *P h* (*Some t2r*)) $\Longrightarrow$
    Ref.get h tr = *ITerm*(*s*, *Some isr*, *IVarD*) $\Longrightarrow$
    P h (*Some tr*)
and *args-case*: $\bigwedge h\ tr\ tsp\ s\ f.$
    ($\bigwedge t2r0\ t2r.$
      t2r $\in$ *i-term-closure h* (*Some t2r0*) $\Longrightarrow$
      t2r0 $\in$ *i-terms-set h tsp* $\Longrightarrow$
      P h (*Some t2r*)) $\Longrightarrow$
    Ref.get h tr = *ITerm*(*s*, *None*, *ITermD*(*f*, *tsp*)) $\Longrightarrow$
    P h (*Some tr*)
shows *P h tp*
proof –
have $\bigwedge$ *t2r*. *t2r* $\in$ *i-term-closure h tp* $\Longrightarrow$ *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 var-case* **by** *fast*
next
  case (*link h tr isr s*)
   then consider (*a*) *t2r* = *tr* | (*b*) *t2r* $\in$ *i-term-closure h* (*Some isr*)
    using *i-term-closure-link* **by** *blast*
  then show *?case* **proof** (*cases*)
    case *a*
    then show *?thesis*
      using *link.IH link.hyps link-case* **by** *blast*
  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* $\in$ *i-terms-set h tsp* $\wedge$ *t2r* $\in$ *i-term-closure h* (*Some t2r0*)
  using *i-term-closure-args* **by** *blast*
  then show *?case* **proof** (*cases*)
    case *a*
    then have $\bigwedge$ *t2r*.
        t2r $\in$ *i-term-closure h* (*Some tr*) $\Longrightarrow$
        Some t2r $\neq$ *Some tr* $\Longrightarrow$
        P h (*Some t2r*)
      using *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)
  shows ¬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 uw 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-term-closures h (*i-term-closures h trs*) = *i-term-closures h trs*
proof −
  have *i-term-closures h* (*i-term-closures h trs*) ⊇ *i-term-closures h trs*
    using *i-term-closure.intros*(*1*) **by** *fastforce*
  moreover {
    fix *tr*
    assume *tr* ∈ *i-term-closures h* (*i-term-closures h trs*)
    then obtain *tr0*
      where *tr* ∈ *i-term-closure h* (*Some tr0*)
        and *tr0-mem*: *tr0* ∈ *i-term-closures h trs*
      by *fast*
    then have *tr* ∈ *i-term-closures h trs*
    proof (*induction tr rule*: *i-term-closure.induct*)
      case (*1 tr*)
      then show *?case*
        by *blast*
    next
      case (*2 tr uu is uv*)
      then show *?case*
        by (*metis UN-iff i-term-closure.intros*(*2*))
    next
      case (*3 tr uw ux tsp tr2*)
      then show *?case*
        by (*metis* (*full-types*) *UN-iff tr0-mem i-term-closure.intros*(*3*))
    qed
  }
  ultimately show *?thesis* **by** *fastforce*
qed

lemma *i-terms-closure-is-closed*:
  shows *is-closed h* (*i-terms-closure h tsp*)
  by (*meson i-term-closures-idem is-closed-def*)

lemma *i-term-closure-is-closed*:
  shows *is-closed h* (*i-term-closure h tp*)
proof (*cases tp*)
  case *None*
  then show *?thesis* **unfolding** *is-closed-def*
    by (*simp add*: *i-term-closure-None*)
next
  case (*Some tr*)
  have *i-term-closure h* (*Some tr*) = *i-term-closures h* {*tr*}
    by *simp*
  then show *?thesis* **unfolding** *is-closed-def*
    using *i-term-closures-idem Some* **by** *presburger*
qed

definition *i-term-closure-v* **where**
  i-term-closure-v h tp = *Ref.get h* ‘ *i-term-closure h tp*

inductive-set
  i-term-chain **for** *h*:: *heap* **and** *tr*:: *i-term ref* **where**
  link:
    $tr' \in$ *i-term-chain h tr* $\Longrightarrow$
    Ref.get h tr' = *ITerm*(*s*, *Some tnextr*, *d*) $\Longrightarrow$
    tnextr $\in$ *i-term-chain h tr* |
  self: *tr* $\in$ *i-term-chain h tr*

lemma *i-term-chain-dest*:
  fixes *tr*:: *i-term ref*
    and *d*:: *i-term-d*
    and *s*:: *nat*
    and *h*:: *heap*
  assumes *Ref.get h tr* = *ITerm*(*s*, *None*, *d*)
  shows *i-term-chain h tr* = {*tr*}
proof −
  {
    fix *x* **assume** *x* $\in$ *i-term-chain h tr*
    then have *x* = *tr*
    using *assms* **by** (*induction rule*: *i-term-chain.induct*, *simp*+)
  }
  then show *?thesis*
    using *i-term-chain.self* **by** *blast*
qed

lemma *i-term-chain-link*:
  fixes *tr*:: *i-term ref*
    and *tr0*:: *i-term ref*
    and *s*:: *nat*
    and *d*:: *i-term-d*
    and *h*:: *heap*
  assumes *Ref.get h tr* = *ITerm*(*s*, *Some tr0*, *d*)
  shows *i-term-chain h tr* = *insert tr* (*i-term-chain h tr0*)
proof −
  {
    fix *x*
    assume *x* $\in$ *i-term-chain h tr*
    then have *x* $\in$ *insert tr* (*i-term-chain h tr0*)
    proof (*induction rule*: *i-term-chain.induct*)
      case (*link* tr' *s tnextr d*)
      show *?case* **proof** (*cases* tr'=*tr*)
        case *True*
        then show *?thesis*
          using *i-term-chain.self assms link.hyps*(*2*) **by** *auto*
      next
        case *False*
        then show *?thesis*
          using *i-term-chain.link link.IH link.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 tnextr 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
    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 ?case
  proof (cases)
    case a
    then show ?thesis using link.prems(2) by simp
  next
    case b
    moreover have i-term-acyclic h (Some isr)
      using link.hyps link.prems(2) acyclic-term-link-simp
      by blast
    ultimately show ?thesis using link.IH by blast
  qed
next
  case (args h tr tsp s f)
  then show ?case
    using i-term-chain-dest t-acyclic-step-ITerm by fast
qed

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

lemma i-term-chain-linkE:
  assumes chain: tr' ∈ i-term-chain h tr
    and diff: tr' ≠ tr
  obtains s tnextr d
  where Ref.get h tr = ITerm(s, Some tnextr, d)
    and tr' ∈ i-term-chain h tnextr
using assms proof (atomize-elim, induction rule: i-term-chain.induct)
case (link tr' s tnextr d)
  show ?case
    using i-term-chain.link i-term-chain.self link.IH link.hyps(1) link.hyps(2) by
blast
next
  case self
  then show ?case by blast
```

qed

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*

unfolding *i-term-closure-v-def*
using *i-term-closure-trans Some t2r-mem* **by** *blast*
}
then have $*$: *get-stamp ‘ i-term-closure-v h* (*Some t2r*) $\subseteq$ *get-stamp ‘ i-term-closure-v h t1p*
by *blast*
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*)
(*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 trs h h' = ($\forall$ *typ x*.
heap.refs h typ x $\neq$ *heap.refs h' typ x* $\longrightarrow$
(*typ* $\neq$ *TYPEREP*(*i-term*) $\wedge$ *typ* $\neq$ *TYPEREP*(*i-terms*) $\wedge$ *typ* $\neq$ *TYPEREP*(*nat*)) $\vee$
($\exists$ *s s' is d*. *typ* = *TYPEREP*(*i-term*) $\wedge$
Ref x $\in$ *trs* $\wedge$
from-nat (*heap.refs h typ x*) = *ITerm*(*s*, *is*, *d*) $\wedge$
from-nat (*heap.refs h' typ x*) = *ITerm*(*s'*, *is*, *d*)))

abbreviation *heap-only-stamp-changed-tr* **where**
heap-only-stamp-changed-tr tr h $\equiv$ *heap-only-stamp-changed* (*i-term-closure h* (*Some tr*)) *h*

abbreviation *heap-only-stamp-changed-ts* **where**
heap-only-stamp-changed-ts tsp h $\equiv$
heap-only-stamp-changed (*i-terms-closure h tsp*) *h*

lemma *heap-only-stamp-ch-nt*:
fixes *trs*:: *i-term ref set*
and *r*:: *'a*::*heap ref*
and *v*:: *'a*::*heap*
and *h*:: *heap*
assumes *TYPEREP*(*'a*) $\neq$ *TYPEREP*(*i-term*)
and *TYPEREP*(*'a*) $\neq$ *TYPEREP*(*i-terms*)
and *TYPEREP*(*'a*) $\neq$ *TYPEREP*(*nat*)
shows *heap-only-stamp-changed trs h* (*Ref.set r v h*)
unfolding *heap-only-stamp-changed-def Ref.set-def* **using** *assms* **by** *simp*

lemma *heap-only-stamp-ch-term*:
fixes *trs*:: *i-term ref set*
and *r*:: *i-term ref*
and *is*:: *i-termP*
and *d*:: *i-term-d*
and *s*:: *nat*

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

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

lemma *heap-only-stamp-ch-get-term'*:
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 *Ref.get* h' *tr* = *ITerm*(*s*, *is*, *d*)
shows $\exists s'$. *Ref.get h tr* = *ITerm*(s', *is*, *d*)
proof (*rule case-split*)
assume *heap.refs h TYPEREP*(*i-term*) (*addr-of-ref tr*) =
heap.refs h' *TYPEREP*(*i-term*) (*addr-of-ref tr*)
then show *?thesis*
using *assms*[*unfolded heap-only-stamp-changed-def*]
by (*simp add*: *Ref.get-def*)
next
assume *heap.refs h TYPEREP*(*i-term*) (*addr-of-ref tr*) $\neq$
heap.refs h' *TYPEREP*(*i-term*) (*addr-of-ref tr*)
then show *?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 $tr \notin trs$
shows *Ref.get* h' *tr* = *Ref.get h tr*
proof −
{
assume *heap.refs h TYPEREP*(*i-term*) (*addr-of-ref tr*) ≠
heap.refs h' *TYPEREP*(*i-term*) (*addr-of-ref tr*)
then have $tr \in trs$
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 TYPEREP*(*i-terms*) (*addr-of-ref tsr*) =
heap.refs h' *TYPEREP*(*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 TYPEREP*(*i-terms*) (*addr-of-ref tsr*) ≠
heap.refs h' *TYPEREP*(*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
```

qed

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*

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* ≠ *TYPE-REP*(*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* ≠ *TYPEREP*(*i-term*) ∧ *typ* ≠ *TYPEREP*(*i-terms*) ∧ *typ* ≠ *TYPEREP*(*nat*)) ∨
(∃ *s s′ is d*. *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*))
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
}
then show *?thesis*
using *heap-only-stamp-changed-def* **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 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 *typ* :: *typerep*
and x :: *nat*
assume ∗: *heap.refs h0 typ x* ≠ *heap.refs h2 typ x*
show (*typ* ≠ *TYPEREP*(*i-term*) ∧ *typ* ≠ *TYPEREP*(*i-terms*) ∧ *typ* ≠ *TYPE-REP*(*nat*)) ∨
(∃ *s s′ is d*.
typ = *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 *typ* ≠ *TYPEREP*(*i-term*) ∧ *typ* ≠ *TYPEREP*(*i-terms*) ∧ *typ* ≠ *TYPE-REP*(*nat*)
then show *?thesis* **by** *simp*
next
assume ∗∗: ¬(*typ* ≠ *TYPEREP*(*i-term*) ∧ *typ* ≠ *TYPEREP*(*i-terms*) ∧ *typ* ≠ *TYPEREP*(*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
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* $\longrightarrow$ *i-term-acyclic h' tp*) $\wedge$
(*i-terms-acyclic h tsp* $\longrightarrow$ *i-terms-acyclic h' tsp*)
proof −
have (*i-term-acyclic h tp* $\longrightarrow$ *heap-only-stamp-changed trs h h'* $\longrightarrow$ *i-term-acyclic h' tp*) $\wedge$
(*i-terms-acyclic h tsp* $\longrightarrow$ *heap-only-stamp-changed trs h h'* $\longrightarrow$ *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 impI*)
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-terms-acyclic h′* (*Some tsref*)
using *hosc i-term-acyclic-i-terms-acyclic.ts-acyclic-step-ITerms ts-acyclic-step-ITerms.IH* **by** *blast*
qed
qed
then show *?thesis* **using** *assms* **by** *blast*
qed

lemma *heap-only-stamp-ch-term-acyclic*:
assumes *i-term-acyclic h tp*
and *heap-only-stamp-changed trs h h′*
shows *i-term-acyclic h′ tp*
using *assms heap-only-stamp-ch-term-terms-acyclic* **by** *blast*

lemma *heap-only-stamp-ch-terms-acyclic*:
assumes *i-terms-acyclic h tsp*
and *heap-only-stamp-changed trs h h′*
shows *i-terms-acyclic h′ tsp*
using *assms heap-only-stamp-ch-term-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 *tr′* ∈ *i-terms-set h tsp*
shows *heap-only-stamp-changed-tr tr h h′*
unfolding *heap-only-stamp-changed-def*
proof (*intro allI impI*)
fix *typ x*
assume *refs h typ x* ≠ *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*) **and**
Ref x ∈ *i-term-closure h* (*Some tr′*) **and**
from-nat (*refs h typ x*) = *ITerm* (*s*, *is*, *d*) **and**
from-nat (*refs h′ typ x*) = *ITerm* (*s′*, *is*, *d*)
using *hosc*[*unfolded heap-only-stamp-changed-def*] **by** *blast*
then show *typ* ≠ *TYPEREP*(*i-term*) ∧ *typ* ≠ *TYPEREP*(*i-terms*) ∧ *typ* ≠ *TYPEREP*(*nat*) ∨
(∃ *s s′ is d*.
typ = *TYPEREP*(*i-term*) ∧
Ref x ∈ *i-term-closure h* (*Some tr*) ∧

$from\text{-}nat\ (refs\ h\ typ\ x) = ITerm\ (s,\ is,\ d) \wedge$
$from\text{-}nat\ (refs\ h'\ typ\ x) = ITerm\ (s',\ is,\ d))$
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
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*) *assms heap-only-stamp-ch-closure heap-only-stamp-ch-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-trans)

definition *heap-only-nonterm-changed* **where**
$heap\text{-}only\text{-}nonterm\text{-}changed\ h\ h' = (\forall\ typ\ x.$
$heap.refs\ h\ typ\ x \neq heap.refs\ h'\ typ\ x \longrightarrow$
$(typ \neq TYPEREP(i\text{-}term) \wedge typ \neq TYPEREP(i\text{-}terms)))$

lemma *heap-only-nonterm-chI*:
fixes r ::'a::*heap ref*
assumes $TYPEREP('a) \neq TYPEREP(i\text{-}term) \wedge TYPEREP('a) \neq TYPEREP(i\text{-}terms)$
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*) ∨ *TYPEREP*(*'a*) = *TYPE-REP*(*i-terms*)
shows *Ref.get h r* = *Ref.get h' r*
unfolding *Ref.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 *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-term-acyclic h tr ⟹ *i-term-acyclic h' tr*
and *heap-only-nonterm-ch-terms-acyclic*:
i-terms-acyclic h tsp ⟹ *i-terms-acyclic h' tsp*
unfolding *conjunction-def*
proof (*atomize*, *unfold atomize-conj*[*unfolded conjunction-def*], *goal-cases*)
case *1*
have (*i-term-acyclic h tr* ⟶ *heap-only-nonterm-changed h h'* ⟶ *i-term-acyclic h' tr*) ∧
(*i-terms-acyclic h tsp* ⟶ *heap-only-nonterm-changed h h'* ⟶ *i-terms-acyclic h' tsp*)
proof ((
induction
rule: *i-term-acyclic-i-terms-acyclic.induct*;
intro impI),
goal-cases nil link args terms-nil terms-next)

case (*nil h*)
then show *?case*
by (*simp add*: *i-term-acyclic-i-terms-acyclic.t-acyclic-nil*)
next
case (*link h t tref s*)
have*Ref.get h' tref* = *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*

qed

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 args 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*: $\bigwedge h\ trs\ tr\ s.$
$\quad$ *Ref.get h tr = ITerm(s, None, IVarD)* $\Longrightarrow$
$\quad$ *P1 h trs tr*
and *link-case*: $\bigwedge h\ trs\ tr\ isr\ s.$
$\quad$ ($\bigwedge t2r\ h'\ trs'.$
$\quad\quad$ $trs \subseteq trs' \Longrightarrow$
$\quad\quad$ *heap-only-stamp-changed* $trs'\ h\ h' \Longrightarrow$
$\quad\quad$ $t2r \in$ *i-term-closure h* (*Some isr*) $\Longrightarrow$
$\quad\quad$ *P1* $h'\ trs'\ t2r) \Longrightarrow$
$\quad$ *Ref.get h tr = ITerm(s, Some isr, IVarD)* $\Longrightarrow$
$\quad$ *P1 h trs tr*
and *args-case*: $\bigwedge h\ trs\ tr\ tsp\ s\ f.$
$\quad$ ($\bigwedge h'\ trs'.$
$\quad\quad$ $trs \subseteq trs' \Longrightarrow$
$\quad\quad$ *heap-only-stamp-changed* $trs'\ h\ h' \Longrightarrow$
$\quad\quad$ *P2* $h'\ trs'\ tsp) \Longrightarrow$
$\quad$ ($\bigwedge h'\ trs'\ t2r0\ t2r.$
$\quad\quad$ $trs \subseteq trs' \Longrightarrow$
$\quad\quad$ *heap-only-stamp-changed* $trs'\ h\ h' \Longrightarrow$
$\quad\quad$ $t2r \in$ *i-term-closure h* (*Some t2r0*) $\Longrightarrow$
$\quad\quad$ $t2r0 \in$ *i-terms-set h tsp* $\Longrightarrow$
$\quad\quad$ *P1* $h'\ trs'\ t2r) \Longrightarrow$
$\quad$ *Ref.get h tr = ITerm(s, None, ITermD(f, tsp))* $\Longrightarrow$
$\quad$ *P1 h trs tr*
and *terms-nil-case*: $\bigwedge h\ trs.$ *P2 h trs None*
and *terms-case*: $\bigwedge h\ trs\ tr\ tsr\ tsnextp.$
$\quad$ ($\bigwedge h'\ trs'.$
$\quad\quad$ $trs \subseteq trs' \Longrightarrow$
$\quad\quad$ *heap-only-stamp-changed* $trs'\ h\ h' \Longrightarrow$
$\quad\quad$ *P2* $h'\ trs'\ tsnextp) \Longrightarrow$
$\quad$ ($\bigwedge h'\ trs'\ t2r.$
$\quad\quad$ $trs \subseteq trs' \Longrightarrow$
$\quad\quad$ *heap-only-stamp-changed* $trs'\ h\ h' \Longrightarrow$
$\quad\quad$ $t2r \in$ *i-term-closure h* (*Some tr*) $\Longrightarrow$
$\quad\quad$ *P1* $h'\ trs'\ t2r) \Longrightarrow$
$\quad$ *Ref.get h tsr = ITerms (tr, tsnextp)* $\Longrightarrow$
$\quad$ *P2 h trs (Some tsr)*
shows *P1 h trs tr*
proof –
{
fix *tp*
have *i-term-acyclic h tp* $\Longrightarrow$
$\quad$ ($\bigwedge\ tr\ h'\ trs'.$
$\quad\quad$ $tr \in$ *i-term-closure h tp* $\Longrightarrow$
$\quad\quad$ *heap-only-stamp-changed* $trs'\ h\ h' \Longrightarrow$
$\quad\quad$ *P1* $h'\ trs'\ tr \wedge$
$\quad\quad$ ($\forall\ s\ f\ tsp0\ tsp.$
$\quad\quad\quad$ *Ref.get h tr = ITerm(s, None, ITermD(f, tsp0))* $\longrightarrow$
$\quad\quad\quad$ $tsp \in$ *i-terms-sublists h tsp0* $\longrightarrow$

```
        P2 h' trs' tsp))
  proof (induction
      taking: λh tsp. (∀ trs' h'.
        heap-only-stamp-changed trs' h h' ⟶ (
          (∀ tsp'.
            tsp' ∈ i-terms-closure-sublists 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 hosc-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 hosc-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** *Ref.get* h' *tr* = *ITerm* (s', *Some isr*, *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' *trs tr* **using** *link-case* **by** *meson*
then show *?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' *trs*)
then have *get-tref*: *Ref.get h tref* = *ITerm* (*s*, *None*, *ITermD* (*f*, *tsref*))
and *tr-clos-tref*: *tr* ∈ *i-term-closure h* (*Some tref*)
and *hosc-h-h'*: *heap-only-stamp-changed trs h* h'
and *IH1*: ⋀*trs'* h' *tsp'*.
heap-only-stamp-changed trs' *h* h' ⟹
tsp' ∈ *i-terms-closure-sublists h tsref* ⟹
P2 h' *trs'* *tsp'*
and *IH2*: ⋀*trs'* h' *tr*.
heap-only-stamp-changed trs' *h* h' ⟹
tr ∈ *i-terms-closure h tsref* ⟹
P1 h' *trs'* *tr* **by** *blast+*
have *tr-clos'-tref*: *tr* ∈ *i-term-closure* h' (*Some tref*)
using *hosc-h-h'* *heap-only-stamp-ch-closure tr-clos-tref* **by** *auto*
have ∗: ⋀h'' *trs'*. *trs* ⊆ *trs'* ⟹ *heap-only-stamp-changed trs'* h' h'' ⟹ *P2*
h'' *trs'* *tsref*
proof −
fix h'' *trs'*
assume *trs* ⊆ *trs'*
and *heap-only-stamp-changed trs'* h' h''
then have *heap-only-stamp-changed trs'* *h* h''
using *heap-only-stamp-ch-antimono heap-only-stamp-ch-trans*
t-acyclic-step-ITerm.prems(*2*) **by** *blast*
then show *P2* h'' *trs'* *tsref*
using *IH1 i-terms-sublists.self* **by** *fast*
qed
have ∗∗: ⋀h'' *trs'* *t2r0 t2r*.
trs ⊆ *trs'* ⟹
heap-only-stamp-changed trs' h' h'' ⟹
t2r ∈ *i-term-closure* h' (*Some t2r0*) ⟹
t2r0 ∈ *i-terms-set* h' *tsref* ⟹ *P1* h'' *trs'* *t2r*
proof −
fix h'' *trs'* *t2r0 t2r*
assume *trs* ⊆ *trs'*
and *hosch-h'-h''*: *heap-only-stamp-changed trs'* h' h''
and *t2r-clos'-t2r0*: *t2r* ∈ *i-term-closure* h' (*Some t2r0*)
and *t2r0-terms'*: *t2r0* ∈ *i-terms-set* h' *tsref*
then have *hosc-h-h''*: *heap-only-stamp-changed trs'* *h* h''
using *heap-only-stamp-ch-antimono heap-only-stamp-ch-trans*

t-acyclic-step-ITerm.prems(2) **by** *blast*

have *t2r0-terms-set-tsref*: *t2r0* ∈ *i-terms-set h tsref*
using *t2r0-terms′ hosc-h-h′[symmetric] heap-only-stamp-ch-terms-set* **by** *blast*
have *t2r-clos-tsref*: *t2r* ∈ *i-terms-closure h tsref*
using *UN-I t2r-clos′-t2r0 t2r0-terms-set-tsref*
heap-only-stamp-ch-closure hosc-h-h′ **by** *fast*
then show *P1 h′′ trs′ t2r* **using** *IH2[OF hosc-h-h′′ t2r-clos-tsref]* **by** *blast*
qed
consider (*a*) *tr* ∈ *i-terms-closure h tsref* | (*b*) *tr* = *tref*
using *get-tref 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 hosc-h-h′* **by** *blast*
show *?thesis*
proof (*intro conjI*, *simp add*: *t1*, *intro allI impI*)
fix *s f tsp tsp0*
assume *get-tr*: *Ref.get h tr* = *ITerm* (*s*, *None*, *ITermD* (*f*, *tsp0*))
and *tsp-sublist-tsp0*: *tsp* ∈ *i-terms-sublists h tsp0*
have *tsp* ∈ (⋃ *tr*∈*i-terms-closure h tsref*. *i-term-sublists h tr*)
by (*metis* (*no-types*) *UN-iff a get-ITerm-args-iff-ex get-tr tsp-sublist-tsp0*)
then have *tsp* ∈ *i-terms-closure-sublists h tsref*
by *blast*
then show *P2 h′ trs tsp* **using** *IH1 hosc-h-h′* **by** *presburger*
qed
next
case *b*
then obtain *s′* **where** *Ref.get h′ tr* = *ITerm* (*s′*, *None*, *ITermD* (*f*, *tsref*))
using *get-tref heap-only-stamp-ch-get-term hosc-h-h′* **by** *blast*
from ∗ ∗∗ *args-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*: *Ref.get h tr* = *ITerm* (*s*, *None*, *ITermD* (*f*, *tsp0*))
and *tsp-sublist-tsp0*: *tsp* ∈ *i-terms-sublists h tsp0*
then have *tsp* ∈ *i-terms-closure-sublists h tsref*
using *get-tref b* **by** *fastforce*
then show *P2 h′ trs tsp*
using *IH1 hosc-h-h′* **by** *blast*
qed
qed
next
case (*ts-acyclic-nil uy*)
then show *?case*
using *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*: $\bigwedge$*tr trs′ h′*.
tr ∈ *i-terms-closure h ts2ref* ⟹
heap-only-stamp-changed trs′ h h′ ⟹
P1 h′ trs′ tr
and *IH1b*: $\bigwedge$*tr trs′ tsp′ h′*.
tsp′ ∈ *i-terms-closure-sublists h ts2ref* ⟹
heap-only-stamp-changed trs′ h h′ ⟹
P2 h′ trs′ tsp′
and *get-tsref*: *Ref.get h tsref* = *ITerms* (*tref*, *ts2ref*)
and *tref-acyclic*: *i-term-acyclic h* (*Some tref*)
by *blast+*
have *IH2a*: $\bigwedge$*tr trs′ tr h′*.
tr ∈ *i-term-closure h* (*Some tref*) ⟹
heap-only-stamp-changed trs′ h h′ ⟹
P1 h′ trs′ tr
and *IH2b*: $\bigwedge$*tr trs′ tr h′ s f tsp0 tsp*. *tr* ∈ *i-term-closure h* (*Some tref*) ⟹
heap-only-stamp-changed trs′ h h′ ⟹
Ref.get h tr = *ITerm* (*s*, *None*, *ITermD* (*f*, *tsp0*)) ⟹
tsp ∈ *i-terms-sublists h tsp0* ⟹
P2 h′ trs′ tsp
by (*simp add*: *ts-acyclic-step-ITerms.IH*)+

show *?case*
proof (*intro allI impI conjI*, *goal-cases terms term*)
case (*term trs′ h′ tr*)
then have *hosc-h-h′*: *heap-only-stamp-changed trs′ h h′*
and *tr-clos-tsref*: *tr* ∈ *i-terms-closure h* (*Some tsref*)
by *blast+*
consider (*a*) *tr* ∈ *i-terms-closure h ts2ref* |
(*b*) *tr* ∈ *i-term-closure h* (*Some tref*)
using *get-tsref*
by (*metis tr-clos-tsref UnE i-terms-closure-terms*)
then show *?case*
proof (*cases*)
case *a*
then show *?thesis* **using** *IH1a hosc-h-h′* **by** *presburger*
next
case *b*
then show *?thesis* **using** *IH2a hosc-h-h′* **by** *presburger*
qed
next
case (*terms trs′ h′ tsp′*)
then have *tsp′-clsl-tsref*: *tsp′* ∈ *i-terms-closure-sublists h* (*Some tsref*)
and *hosc-h-h′*: *heap-only-stamp-changed trs′ h h′*
by *blast+*
have *get′-tsref*: *Ref.get h′ tsref* = *ITerms* (*tref*, *ts2ref*)

using *get-tsref hosc-h-h′ heap-only-stamp-ch-get-terms* **by** *simp*
consider (a) $tsp' = None$ |
(b) tsr'
where $tsp' = Some\ tsr'$
and $tsp' \in$ *i-term-closure-sublists h* $(Some\ tref)$ |
(c) tsr'
where $tsp' = Some\ tsr'$
and $tsp' \in$ *i-terms-closure-sublists h ts2ref* |
(d) $tsp' = Some\ tsref$
using *i-term-closure-sublists-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*: $tr \in$ *i-term-closure h* $(Some\ tref)$
and *tsr′-sublist-tr*: $Some\ tsr' \in$ *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*: $Some\ tsr' \in$ *i-terms-sublists h tsp0*
using *i-terms-sublists-someE acyclic-args-nil-is* **by** *auto*
show *?thesis* **using** *IH2b*[*OF tr-clos-tref hosc-h-h′ get-tr tsr′-sublist-tsp0*]

b

by *fast*
next
case *c*
then show *?thesis* **using** *IH1b hosc-h-h′* **by** *presburger*
next
case *d*
have $*$: $\bigwedge h''\ trs''$.
$trs' \subseteq trs'' \Longrightarrow$
heap-only-stamp-changed $trs''\ h'\ h'' \Longrightarrow$
P2 $h''\ trs''$ *ts2ref*
by (*meson IH1b UnCI heap-only-stamp-ch-antimono heap-only-stamp-ch-trans*
hosc-h-h′ i-terms-sublists.self)
have $**$: $\bigwedge h''\ trs''\ t2r$.
$trs' \subseteq trs'' \Longrightarrow$
heap-only-stamp-changed $trs''\ h'\ h'' \Longrightarrow$
$t2r \in$ *i-term-closure* h' $(Some\ tref) \Longrightarrow$
P1 $h''\ trs''\ t2r$
proof −
fix $h''\ trs''\ t2r$
assume *trs′-subset-trs″*: $trs' \subseteq trs''$

```
          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-trans
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
 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.Ref
   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 time:: nat ref
    and v:: i-termP
begin


partial-function (heap) i-occ-p:: i-termP + i-termsP ⇒ bool Heap where [code]:
  i-occ-p XX = (
    case XX of
      (Inl (Some t)) ⇒ do {
        tv ← !t;
        case tv of
          ITerm (-, -, IVarD) ⇒ return (v = Some t)
        | ITerm (stamp, None, ITermD(f, args)) ⇒ do {
            timev ← !time;
            if (stamp = timev) then return False
            else do {
              t := ITerm(timev, None, ITermD(f, args));
              i-occ-p (Inr args)
            }
          }
      }
    | (Inr None) ⇒ return False
    | (Inr (Some ts)) ⇒ do {
      tsv ← !ts;
      case tsv of
        ITerms (t, next) ⇒ do {
          find-res ← i-find (Some t);
          occ-res ← i-occ-p (Inl find-res);
          if occ-res then return True
            else i-occ-p (Inr next) }
      }
  )

definition i-occurs:: i-termP ⇒ bool Heap where
  i-occurs t = do {
    timev ← !time;
    time := timev + 1;
    i-occ-p (Inl t)
  }
end

end
```

A.5 Equivalence of imperative and functional formulation

```
theory ImpEqFunc
  imports Main
    Unification-Functional
    Unification-Imperative
```

```
  HOL−Imperative-HOL.Ref
  HOL−Imperative-HOL.Heap-Monad
begin
```

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

```
partial-function (heap)
 i-term-to-term-p:: i-term ref + i-termsP ⇒ (term + term list) Heap
 where [code]:
 i-term-to-term-p XX = (case XX of
  (Inl tr) ⇒ do {
   t ← !tr;
   case t of
    ITerm (-, None, IVarD) ⇒
     return (Inl(V(''x'', int (addr-of-ref tr))))
   | ITerm (-, (Some t2p), -) ⇒ i-term-to-term-p (Inl t2p)
   | ITerm (-, None, ITermD(f, termsp)) ⇒ do {
     v ← i-term-to-term-p (Inr termsp);
     case v of
      Inr(terms) ⇒ return (Inl(T(f, terms))) }
  }
 | (Inr None) ⇒
   return (Inr([]))
 | (Inr (Some termsr)) ⇒ do {
   termsv ← !termsr;
   case termsv of
    (ITerms(tthis, tnext)) ⇒ do {
     vtthis ← i-term-to-term-p (Inl tthis);
     vtnext ← i-term-to-term-p (Inr tnext);
     case (vtthis, vtnext) of
      (Inl(term), Inr(terms)) ⇒ return (Inr(term#terms)) }
  }
 )

lemma i-term-to-term-p-mr:
 fixes h :: heap
  and XX :: i-term ref + i-termsP
 assumes term-acyclic: ⋀tr. XX = Inl tr ⟹ i-term-acyclic h (Some tr)
  and terms-acyclic: ⋀tp. XX = Inr tp ⟹ i-terms-acyclic h tp
 shows ∃ r. (Some(r, h) = execute (i-term-to-term-p XX) h ∧ isl r = isl XX)
proof −
 {
  fix tp trp
  let ?cond XX0 h0 = ∃ r. (Some(r, h) = execute (i-term-to-term-p XX0) h ∧
isl r = isl XX0)
  have (i-term-acyclic h trp ⟶
     trp ≠ None ⟶ ?cond (Inl (case trp of Some tr ⇒ tr)) h) ∧
   (i-terms-acyclic h tp ⟶ ?cond (Inr tp) h)
  proof (induction rule: i-term-acyclic-i-terms-acyclic.induct)
  case (t-acyclic-nil h)
```

then show *?case* **by** *simp*
next
case (*t-acyclic-step-link h t tref stamp*)
then consider (*a*) *Ref.get h tref* = *ITerm*(*stamp*, *None*, *IVarD*) |
(*b*) *tn iv* **where** *Ref.get h tref* = *ITerm* (*stamp*, (*Some tn*), *iv*)
by *auto*
then show *?case* **using** *t-acyclic-step-link.IH*
proof (*cases*)
case *a*
then show *?thesis*
by (*subst i-term-to-term-p.simps*,
simp add: *lookup-def tap-def bind-def return-def*
execute-heap isl-def)
next
case *b*
then show *?thesis* **using** *t-acyclic-step-link.IH*
by (*subst i-term-to-term-p.simps*,
simp add: *lookup-def tap-def bind-def return-def*
execute-heap t-acyclic-step-link.hyps(*2*))
qed
next
case (*t-acyclic-step-ITerm h tsref tref stamp f*)
then obtain *r0* **where** *r0-def*: *Some* (*r0*, *h*) = *execute* (*i-term-to-term-p* (*Inr tsref*)) *h* ∧ ¬*isl r0*
by *auto*
then have ∗: *Some* (*r0*, *h*) = *execute* (*i-term-to-term-p* (*Inr tsref*)) *h* **by** *simp*
obtain *r0v* **where** ∗∗: *Inr r0v* = *r0*
using *r0-def sum.collapse*(*2*) **by** *blast*

show *?case* **using** *t-acyclic-step-ITerm*
apply (*subst i-term-to-term-p.simps*,
simp add: *lookup-def tap-def bind-def return-def*
execute-heap)
apply (*fold* ∗ ∗∗)
by (*simp add*: *execute-heap*)
next
case (*ts-acyclic-nil h*)
then show *?case* **by** (*subst i-term-to-term-p.simps*, *simp add*: *return-def execute-heap*)
next
case (*ts-acyclic-step-ITerms h ts2ref tref tsref*)
then obtain *r0* **where** *r0-def*: *Some* (*r0*, *h*) = *execute* (*i-term-to-term-p* (*Inl tref*)) *h* ∧ *isl r0*
by *auto*
then have *a1*: *Some* (*r0*, *h*) = *execute* (*i-term-to-term-p* (*Inl tref*)) *h* **by** *simp*
obtain *r0v* **where** *a2*: *Inl r0v* = *r0* **using** *r0-def*[*unfolded isl-def*]
by *auto*

obtain *r1* **where** *r1-def*: *Some* (*r1*, *h*) = *execute* (*i-term-to-term-p* (*Inr ts2ref*)) *h* ∧ ¬*isl r1*
    using *ts-acyclic-step-ITerms* **by** *auto*
  then have *b1*: *Some* (*r1*, *h*) = *execute* (*i-term-to-term-p* (*Inr ts2ref*)) *h* **by** *simp*
  obtain *r1v* **where** *b2*: *Inr r1v* = *r1*
    using *r1-def sum.collapse*(*2*) **by** *blast*

  from *ts-acyclic-step-ITerms* **show** *?case*
    apply (*subst i-term-to-term-p.simps*,
      simp add: *lookup-def tap-def bind-def return-def execute-heap*)
    by (*fold a1 a2*, *simp*, *fold b1 b2*, *simp add*: *return-def execute-heap*)
  qed
}
note *proof0* = *this*
show *?thesis*
proof (*cases XX*)
  case (*Inl a*)
  then show *?thesis*
    using *proof0 term-acyclic* **by** *fastforce*
next
  case (*Inr b*)
  then show *?thesis*
    using *proof0 terms-acyclic* **by** *simp*
qed
qed

definition *i-term-to-term*:: *i-term ref* ⇒ *term Heap* **where**
i-term-to-term tr = *do* { *r* ← *i-term-to-term-p* (*Inl tr*); *case r of* (*Inl v*) ⇒ *return v* }

abbreviation *i-term-to-term-e*:: *heap* ⇒ *i-term ref* ⇒ *term* **where**
i-term-to-term-e h tr ≡ (*case* (*execute* (*i-term-to-term tr*) *h*) *of Some*(*r*, -) ⇒ *r*)

lemma *i-term-to-term-value-iff*:
fixes *tr*:: *i-term ref*
  and *r*:: *term*
  and *h*:: *heap*
assumes *i-term-acyclic h* (*Some tr*)
shows (*r* = *i-term-to-term-e h tr*) = (*Some*(*r*, *h*) = *execute* (*i-term-to-term tr*) *h*)
proof −
{
  obtain *XX* **where** ∗: *Some* (*XX*, *h*) = *execute* (*i-term-to-term-p* (*Inl tr*)) *h* **and** *isl XX*
    using *i-term-to-term-p-mr assms isl-def* **by** *fast*
  then obtain *r′* **where** ∗∗: *Inl r′* = *XX*
    using *isl-def* **by** *metis*

assume r = *i-term-to-term-e h tr*
then have *Some*(r, h) = *execute* (*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*:: *i-term ref*
and *h*:: *heap*
assumes *i-term-acyclic h* (*Some tr*)
shows *execute* (*i-term-to-term tr*) *h* = *Some*(*i-term-to-term-e h tr*, *h*)
using *assms i-term-to-term-value-iff* **by** *metis*

definition *i-terms-to-terms*:: *i-termsP* ⇒ *term list Heap* **where**
i-terms-to-terms tp = *do* { *r* ← *i-term-to-term-p* (*Inr tp*); *case r of* (*Inr v*) ⇒ *return v* }

abbreviation *i-terms-to-terms-e*:: *heap* ⇒ *i-termsP* ⇒ *term list* **where**
i-terms-to-terms-e h tr ≡ (*case* (*execute* (*i-terms-to-terms tr*) *h*) *of Some*(*r*, -) ⇒ *r*)

lemma *i-terms-to-terms-value-iff*:
fixes *tsp*:: *i-termsP*
and *r*:: *term list*
and *h*:: *heap*
assumes *i-terms-acyclic h tsp*
shows (*r* = *i-terms-to-terms-e h tsp*) = (*Some*(*r*, *h*) = *execute* (*i-terms-to-terms tsp*) *h*)
proof −
{
obtain *XX* **where** $*$: *Some* (*XX*, *h*) = *execute* (*i-term-to-term-p* (*Inr tsp*)) *h* **and** ¬*isl XX*
using *i-term-to-term-p-mr assms isl-def* **by** *fast*
then obtain r' **where** $**$: *Inr* r' = *XX*
using *sum.collapse*(*2*) **by** *blast*
assume *r* = *i-terms-to-terms-e h tsp*
then have *Some*(*r*, *h*) = *execute* (*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*:: *i-termsP*

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 i-term-to-term-p-mr tthis-acyclic
    by (metis Inr-not-Inl sum.disc(1) sum.sel(1))
  then have a1: Some(r0, h) = execute (i-term-to-term-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-res[unfolded i-term-to-term-def]
    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 (i-term-to-term-p (Inr tnext))
h ∧ ¬isl r1
    using i-term-to-term-p-mr[where XX=Inr tnext] tnext-acyclic
    by auto
  then have b1: Some(r1, h) = execute (i-term-to-term-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(Inr terms, h)
    using tnext-res[unfolded i-terms-to-terms-def]
    by (simp add: bind-def return-def execute-heap)
       (fold b1 b2, simp add: execute-heap return-def)

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

lemma i-terms-to-terms-e-step:
  fixes termsr:: i-terms ref
    and tthis:: i-term ref
    and tsnext:: i-termsP
    and h:: heap
  assumes acyclic: i-terms-acyclic h (Some termsr)
    and get-termsr: Ref.get h termsr = ITerms (tthis, tsnext)
  shows i-terms-to-terms-e h (Some termsr) =
          (i-term-to-term-e h tthis)#(i-terms-to-terms-e h tsnext)
proof −
  have i-term-acyclic h (Some tthis)
    by (meson acyclic get-termsr i-terms-set-acyclic i-terms-setp.intros
        i-terms-setp-i-terms-set-eq i-terms-sublistsp.self)
  moreover have i-terms-acyclic h tsnext
    using acyclic acyclic-terms-terms-simp get-termsr by blast
  ultimately have execute (i-terms-to-terms (Some termsr)) h =
    Some((i-term-to-term-e h tthis)#(i-terms-to-terms-e h tsnext), h)
   using acyclic get-termsr i-term-to-term-value i-terms-to-terms-step i-terms-to-terms-value
```

```
    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))) ∧
    (∀ (tsr :: 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: λh 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 *ttt-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-ITerm.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' *tsref* = *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-acyclic 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*

qed

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*:: *'a*::*heap ref*
and *v*:: *'a*::*heap*
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-uc*:
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 *Ref.get h r* = *ITerm*(*s*, *is*, *d*)
and *v* = *ITerm*(*s'*, *is*, *d'*)
shows
(*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* **proof**
(*induction rule*: *i-term-acyclic-induct'*)
case (*var h tr s*)
then show *?case*
by (*subst* (*1 2*) *i-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-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-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 ∃ *tdestr s d*.
execute (*i-find* (*Some tr*)) *h* = *Some*(*Some tdestr*, *h*) ∧
tdestr ∈ *i-term-chain h tr* ∧
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-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 ∗: *execute* (*i-find* (*Some tr*)) *h* = *execute* (*i-find* (*Some isr*)) *h*
by (*subst i-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.IH 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′*

and *acyclic*: *i-term-acyclic h* (*Some tr0*)
and *get-tr0*: *Ref.get h tr0* = *ITerm*(*s*, *None*, *ITermD*(*f*, *tsp*))
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 *not-occurs*: ¬ *occurs* (″*x*″, *int* (*addr-of-ref vr*)) (*i-term-to-term-e h′ tr0*)
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*
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 *tr0-acyclic′*: *i-term-acyclic h′* (*Some tr0*)
using *acyclic heap-only-stamp-ch-term-acyclic terms-hosc* **by** *blast*
then have *tsp-acyclic′*: *i-terms-acyclic h′ tsp*
using *acyclic-terms-term-simp get′-tr0* **by** *blast*
have *ttt*: *i-term-to-term-e h′ tr0* = *T*(*f*, *i-terms-to-terms-e h′ tsp*)
by (*simp add*: *tr0-acyclic′ get′-tr0 i-term-to-term-e-terms*)
{
fix *tr*
assume *tr-tsp-set*: *tr* ∈ *i-terms-set h tsp*
assume *occ-tr*: *occurs* (″*x*″, *int* (*addr-of-ref vr*)) (*i-term-to-term-e h′ tr*)
have *tr-tsp-set′*: *tr* ∈ *i-terms-set h′ tsp*
using *tr-tsp-set get-tr0 heap-only-stamp-ch-terms-set terms-hosc* **by** *blast*
have (*list-ex* (*occurs* (″*x*″, *int* (*addr-of-ref vr*))) (*i-terms-to-terms-e h′ tsp*)) =
(∃ *t* ∈ *i-term-to-term-e h′* ‘ *i-terms-set h′ tsp*. *occurs* (″*x*″, *int* (*addr-of-ref vr*)) *t*)
using *i-terms-to-terms-list-set*[*OF tsp-acyclic′*] *list-ex-iff* **by** *auto*
then have *list-ex* (*occurs* (″*x*″, *int* (*addr-of-ref vr*))) (*i-terms-to-terms-e h′ tsp*)
using *occ-tr tr-tsp-set′* **by** *blast*
then have *occurs* (″*x*″, *int* (*addr-of-ref vr*)) (*i-term-to-term-e h′ tr0*)
by (*simp add*: *ttt*)
then have *False*
using *not-occurs* **by** *simp*
}
then have *terms-scno′*: ⋀ *tr*. *tr* ∈ *i-terms-set h tsp* ⟹ *stamp-current-not-occurs time vr tr h′*
using *terms-scno* **by** *auto*

consider (*a*) *tr′* = *tr0* |
(*b*) *tr′0* **where**
tr′0 ∈ *i-terms-set h′ tsp* **and**
tr′ ∈ *i-term-closure h′* (*Some tr′0*)
using *tr′-clos i-term-closure-args*[*OF get′-tr0*] **by** *blast*
then show ¬ *occurs* (″*x*″, *int* (*addr-of-ref vr*)) (*i-term-to-term-e h′ tr′*)
proof (*cases*)

case *a*
then show *?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 $*$: *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 *?thesis* **using** *ttt* $*$[*unfolded stamp-current-not-occurs-def*]
using *get′-tr′ s′-time′ i-term-closure.intros*(*1*) **by** *blast*
qed
qed

lemma *stamp-current-not-occurs-terms-set*:
assumes *terms-scno*: $\bigwedge$ *tr*. *tr* $\in$ *i-terms-set h′ tsp* $\Longrightarrow$ *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*: ¬*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 ¬ *occurs* (*′′x′′*, *int* (*addr-of-ref vr*)) (*i-term-to-term-e h′ tr′*)
proof (*cases*)
case *a*
then show *?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 $*$: *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 ?thesis using ttt *[unfolded stamp-current-not-occurs-def]
      using get'-tr' s'-time' i-term-closure.intros(1) by blast
  qed
qed

lemma stamp-current-not-occurs-terms-set-None:
  assumes hosc: heap-only-stamp-changed-tr tr h h'
  and get-tr: 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' ∈ 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: 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 t-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 ¬ occurs (''x'', int (addr-of-ref vr)) (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 ∃ h'. execute (i-occ-p time (Some vr) (Inl(Some tr))) h = Some(r, h') ∧
        heap-only-stamp-changed-tr tr h h' ∧
        stamp-current-not-occurs' time vr tr h'
proof −
  let ?occ-vr = occurs (''x'', int (addr-of-ref vr))
```

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

let *?cond tr* = $\exists h'$. *execute* (*i-occ-p time* (*Some vr*) (*Inl*(*Some tr*))) *h* =
Some(*?occ-vr fun-term*, *h'*) $\wedge$
heap-only-stamp-changed-tr tr h h' $\wedge$
stamp-current-not-occurs' time vr tr h'
{
fix *trs*:: *i-term ref set*
have *trs* = *UNIV* $\Longrightarrow$ *?cond tr*
using *acyclic assms*(*2*) *assms*(*3*) *assms*(*4*) *assms*(*5*) *acyclic*
proof (*induction h trs tr*
arbitrary: *fun-term s1 s2 td*
taking:
λ*h trs tsp*.
$\forall$ *s2*.
Ref.get h vr = *ITerm*(*s2*, *None*, *IVarD*) $\longrightarrow$
trs = *UNIV* $\longrightarrow$
($\forall$ *tr* $\in$ *i-terms-set h tsp*. *stamp-current-not-occurs time vr tr h*) $\longrightarrow$
i-terms-acyclic h tsp $\longrightarrow$
($\exists h'$. *execute* (*i-occ-p time* (*Some vr*) (*Inr tsp*)) *h* =
Some (*?occ-ts h tsp*, *h'*) $\wedge$
heap-only-stamp-changed-ts tsp h h' $\wedge$
stamp-current-not-occurs'-ts time vr tsp h')
rule: *acyclic-closure-ch-stamp-inductc'*)
case (*var h trs tr s*)
then have *get-tr*: *Ref.get h tr* = *ITerm* (*s*, *None*, *IVarD*)
and *get-tr'*: *Ref.get h tr* = *ITerm* (*s1*, *None*, *td*)
and *scno*: *stamp-current-not-occurs time vr tr h*
and *acyclic*: *i-term-acyclic h* (*Some tr*)
and *fun-term*: *Some* (*fun-term*, *h*) = *execute* (*i-term-to-term tr*) *h* **by** *simp-all*

from *fun-term acyclic* **have** *fun-term* = *i-term-to-term-e h tr*
using *i-term-to-term-value-iff*
by *simp*
then have $**$: (*vr* = *tr*) = *?occ-vr fun-term*
using *var i-term-to-term-var-none* **by** *force*
show *?case* **using** *var*
apply (*subst i-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 (*link h tr isr s*)
then show *?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*
get-tr i-term-closure.intros(*1*)
by *fast*
show *?case* **using** *s-eq-time*
apply (*subst i-occ-p.simps*,
simp add: *lookup-def update-def tap-def bind-def return-def execute-heap*
args s-eq-time ∗)
by (*unfold heap-only-stamp-changed-def*, *simp*)
next
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-changed-tr tr h ?h'*
using *heap-only-stamp-ch-term*[*OF get-tr*] *i-term-closure.intros*(*1*) **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*
?h'
unfolding *stamp-current-not-occurs-def*
proof (*intro ballI allI impI*)
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.intros*(*1*) *i-term-closure.intros*(*3*)
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 *?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 ?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 ¬ *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 ¬ *occurs* ($''x''$, *int* (*addr-of-ref vr*)) (*i-term-to-term-e ?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 ?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-conv*)
have *tsp-acyclic'*: *i-terms-acyclic ?h' tsp*
using *heap-only-stamp-ch-terms-acyclic hosc-h-h' tsp-acyclic* **by** *blast*

have *hosc-h-h'-trs*: *heap-only-stamp-changed trs h ?h'*
using *hosc-h-h' trs-val*
get-tr heap-only-stamp-ch-term **by** *auto*
have *i-terms-closure ?h' tsp* = *i-terms-closure h tsp*
using *heap-only-stamp-ch-terms-closure hosc-h-h'* **by** *presburger*

obtain h'' **where**
IH-exec: *execute* (*i-occ-p time* (*Some vr*) (*Inr tsp*)) *?h'* = *Some*(*?occ-ts ?h' tsp*, h'') **and**
IH-hosc: *heap-only-stamp-changed-ts tsp ?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 *?case*
proof (*rule case-split*)
assume *i-terms-set ?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-terms-nil*)

then have $*$: $\neg$ *?occ-vr fun-term*
by *simp*
have $**$: *heap-only-stamp-changed-tr tr h*
(*Ref.set tr* (*ITerm* (*Ref.get h time*, *None*, *ITermD* (*f*, *None*))) *h*)
using *hosc-h-h′ tsp-none* **by** *auto*
have $***$: *stamp-current-not-occurs′ time vr tr*
(*Ref.set tr* (*ITerm* (*Ref.get h time*, *None*, *ITermD* (*f*, *None*))) *h*)
using *stamp-current-not-occurs-terms-set-None*
get-tr hosc-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 $*$ $**$ $***$)
next
assume *tsp-set-not-empty*: *i-terms-set ?h′ tsp* $\neq$ {}
have *i-terms-closure ?h′ tsp* $\subseteq$ *i-term-closure ?h′* (*Some tr*)
using *get′-tr i-term-closure-args* **by** *blast*
then have *hosc*: *heap-only-stamp-changed-tr tr ?h′ h″* **using** *IH-hosc*
get′-tr heap-only-stamp-ch-antimono **by** *meson*
have *fun-term′*: *i-term-to-term-e ?h′ tr* = *fun-term*
using *acyclic fun-term-e hosc-h-h′ i-term-to-term-only-stamp-changed* **by**
auto
have *occ-tr-eq-occ-tsp*: *?occ-vr fun-term* = *?occ-ts h tsp*
by (*simp add*: *acyclic fun-term-e get-tr i-term-to-term-e-terms*)
also have *occ-tr-eq-occ′-tsp*: ... = *?occ-ts ?h′ tsp*
using *i-terms-to-terms-only-stamp-changed′*[*OF tsp-acyclic get-tr hosc-h-h′*]
by *presburger*
have *occ′-tr-eq-occ′-tsp*: *?occ ?h′ tr* = *?occ-ts ?h′ tsp*
by (*simp add*: *fun-term′ occ-tr-eq-occ′-tsp occ-tr-eq-occ-tsp*)
have *hosc-h-h″*: *heap-only-stamp-changed-tr tr h h″*
using *heap-only-stamp-ch-trans hosc hosc-h-h′ heap-only-stamp-ch-closure*
by (*metis* (*no-types*, *lifting*))

have *i-terms-closure h″ tsp* $\subseteq$ *i-term-closure h″* (*Some tr*)
using *i-term-closure-args IH-hosc get′-tr heap-only-stamp-ch-get-term* **by**
blast
have *fun-term″*: *i-term-to-term-e h″ tr* = *fun-term*
using *acyclic fun-term-e hosc-h-h″ i-term-to-term-only-stamp-changed*
by *auto*
have *ttt-tsp″*: *i-terms-to-terms-e h tsp* = *i-terms-to-terms-e h″ tsp*
using *get-tr hosc-h-h″ i-terms-to-terms-only-stamp-changed′ tsp-acyclic*
by *blast*
have *tr-acyclic′*: *i-term-acyclic ?h′* (*Some tr*)
using *get′-tr t-acyclic-step-ITerm tsp-acyclic′* **by** *blast*
have *terms-set′-tsp-to″*: *i-terms-set ?h′ tsp* = *i-terms-set h″ tsp*
using *IH-hosc get′-tr heap-only-stamp-ch-terms-set* **by** *blast*
then have *scno-h″*: *stamp-current-not-occurs′ time vr tr h″*
using *stamp-current-not-occurs′-terms-set IH-concl IH-hosc fun-term″*
get′-tr

occ-tr-eq-occ-tsp terms-set′-tsp-to″ tr-acyclic′ ttt-tsp″
by (*metis* (*no-types*))
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
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* = *ITerms* (*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*:
$\bigwedge$*tr*. *tr* ∈ *i-terms-set h* (*Some tsr*) ⟹
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* ∈ *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 *thisr-acyclic*: *i-term-acyclic h* (*Some tthisr*)
using *terms-acyclic get-tsr i-terms-set.intros i-terms-set-acyclic*

i-terms-sublists.self get-tsr **by** *blast*
have *exec-ifind'*: *execute* (*i-find* (*Some tthisr*)) *h* = *Some* (*Some tdestr*, *h*)
using *exec-ifind thisr-acyclic*
i-find-heap-change-is-uc **by** *blast*

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

have *tthisr-terms-set-tsp*: *tthisr* ∈ *i-terms-set h* (*Some tsr*)
using *get-tsr i-terms-set.intros 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-thisr-closure thisr-acyclic*
by *presburger*

have *stamp-current-not-occurs time vr tthisr h*
using *terms-scno*
by (*simp add*: *tthisr-terms-set-tsp*)
moreover have *tdestr-clos-subset-tthisr-clos*:
i-term-closure h (*Some tdestr*) ⊆ *i-term-closure h* (*Some tthisr*)
using *i-term-closure-trans tdestr-thisr-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-thisr-closure thisr-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-hosc: *heap-only-stamp-changed-tr tdestr h* h' **and**
IH-scno: *stamp-current-not-occurs' time vr tdestr* h'
using *terms.IH*[*OF - heap-only-stamp-ch-refl*
tdestr-thisr-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-tthisr-clos tthisr-terms-set-tsp* **by** *auto*

have *hosc-tsr*: *heap-only-stamp-changed-ts* (*Some tsr*) *h* h'
using *heap-only-stamp-ch-antimono IH-hosc 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 tsr-acyclic': i-terms-acyclic h' (Some tsr)
    by (meson heap-only-stamp-ch-terms-acyclic hosc-tsr terms-acyclic)
have get'-tsr: Ref.get h' tsr = ITerms (tthisr, tsnextp)
  using get-tsr heap-only-stamp-ch-get-terms hosc-tsr by auto

have ttt-tdestr: i-term-to-term-e h tthisr = i-term-to-term-e h tdestr
  using i-term-to-term-chain tdestr-mem thisr-acyclic by presburger
then have ttt-tsr: i-terms-to-terms-e h (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-tsr': 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-tsr: stamp-current-not-occurs'-ts time vr (Some tsr) h'
  unfolding stamp-current-not-occurs-def
proof (intro impI allI ballI)
  fix tr tr' s' is 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 h' tdestr
    using ttt-tsr' not-occ-tsr by auto
  show ¬ ?occ h' tr'
  proof (rule case-split)
    assume tr' ∈ 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' ∉ i-term-closure h' (Some tdestr)
    then have get-tr': 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 by force
    have tr-tsr-term-set: tr ∈ i-terms-set h (Some tsr)
      using heap-only-stamp-ch-terms-set IH-hosc tr-tsr-term-set' by auto
    have tr-clos-tr: tr' ∈ i-term-closure h (Some tr)
      using IH-hosc heap-only-stamp-ch-closure tr-clos'-tr by auto
    have s'-time': s' = Ref.get h time
      using IH-hosc heap-only-stamp-ch-get-nat s'-time' by auto
    have ¬?occ h tr'
      using terms-scno[unfolded stamp-current-not-occurs-def]
        tr-tsr-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
```

```
      i-terms-set-acyclic terms-acyclic tr-clos-tr tr-tsr-term-set by blast
    ultimately show ?thesis by fastforce
  qed
qed

show ?case
proof (rule case-split)
  assume occ-tdestr: ?occ h tdestr
  then have *: ?occ-ts h (Some tsr) using ttt-tsr by simp
  show ?thesis
  apply (subst i-occ-p.simps,
        simp add: lookup-def tap-def bind-def return-def execute-heap
      get-tsr exec-ifind IH-exec occ-tdestr * terms-scno)
    using scno-tsr hosc-tsr by auto
next
  assume not-occ-tdestr: ¬?occ h tdestr
  obtain s2' where get'-vr: Ref.get h' vr = ITerm(s2', None, IVarD)
    using get-vr IH-hosc
      heap-only-stamp-ch-get-term by blast
  have hosc-h-h'-trs: heap-only-stamp-changed trs h h'
    using heap-only-stamp-ch-antimono hosc-tsr trs-val by blast
  have tsnextp-acyclic': i-terms-acyclic h' tsnextp
   using IH-hosc heap-only-stamp-ch-terms-acyclic tsnextp-acyclic by blast

  have scno-tsnextp': ⋀tr. tr ∈ i-terms-set h' tsnextp ⟹
    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 ∈ i-terms-set h' tsnextp
      and tr-clos'-tr: tr' ∈ i-term-closure h' (Some tr)
      and get'-tr': Ref.get h' tr' = ITerm (s', is, d)
      and s-eq'-time: s' = Ref.get h' time
    have not-occurs'-tdestr: ¬ ?occ h' tdestr
      using hosc-h-h'-trs i-term-to-term-only-stamp-changed not-occ-tdestr
        tdestr-acyclic by auto
    show ¬ occurs (''x'', int (addr-of-ref vr)) (i-term-to-term-e h' tr')
    proof (rule case-split)
      assume tr' ∈ i-term-closure h' (Some tdestr)
   then show ?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' ∉ i-term-closure h' (Some tdestr)
      then have get-tr': Ref.get h tr' = ITerm (s', is, 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 ∈ i-terms-set h (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′-clos-tr*: $tr' \in$ *i-term-closure h* (*Some tr*)
using *IH-hosc*
heap-only-stamp-ch-closure tr-clos′-tr **by** *blast*
moreover have $s' =$ *Ref.get h time*
using *heap-only-stamp-ch-get-nat hosc-h-h′-trs s-eq′-time* **by** *presburger*
ultimately have $\neg$ *?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
qed
obtain h'' **where**
IHn-exec:
execute (*i-occ-p time* (*Some vr*) (*Inr tsnextp*)) h' = *Some*(*?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′-vr trs-val scno-tsnextp′ tsnextp-acyclic′] **by** *fast*

have *i-terms-closure h tsnextp* $\subseteq$ *i-terms-closure h* (*Some tsr*)
by (*simp add*: *get-tsr 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 (*simp add*: *heap-only-stamp-ch-antimono*)
then have $*$: *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* = *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 *thisr-acyclic′*: *i-term-acyclic h′* (*Some tthisr*)
using *IH-hosc heap-only-stamp-ch-term-acyclic thisr-acyclic* **by** *blast*
have *ttt-tdestr″*: *i-term-to-term-e h″ tthisr* = *i-term-to-term-e h″ tdestr*
using $*$ *IH-hosc i-term-to-term-only-stamp-changed tdestr-acyclic tdestr-acyclic′*
thisr-acyclic thisr-acyclic′ ttt-tdestr **by** *auto*
have *ttt-tsr″*: *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 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' is 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 h'' 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'-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-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-scno*[*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′ ∈ i-term-closure h′ (Some tdestr)
              using IHn-hosc b(2) heap-only-stamp-ch-closure by auto
            moreover have s′ = Ref.get h′ time
              using IHn-hosc heap-only-stamp-ch-get-nat s′-time′′ by auto
           ultimately have ¬ occurs (′′x′′, int (addr-of-ref vr)) (i-term-to-term-e
h′ tr′)
              using IH-scno[unfolded stamp-current-not-occurs-def]
                   not-occ′-tdestr by blast
            then show ?thesis using ttt-h′-h′′-tr′ by simp
          next
            case (c)
            then have Ref.get h′ tr′ = ITerm (s′, is, d)
              using IHn-hosc get′′-tr′ heap-only-stamp-ch-get-term-nclos
                heap-only-stamp-ch-terms-closure by fastforce
            then have Ref.get h tr′ = ITerm (s′, is, d)
              using c(2)
           ∗ IH-hosc heap-only-stamp-ch-closure heap-only-stamp-ch-get-term-nclos
              by force
            moreover have tr ∈ i-terms-set h (Some tsr)
              using ∗ heap-only-stamp-ch-terms-set tr-tsr-term-set′′ by blast
            moreover have tr′ ∈ i-term-closure h (Some tr)
              using ∗ heap-only-stamp-ch-closure tr′-clos′′-tr by blast
            moreover have s′ = Ref.get h time
              using ∗ heap-only-stamp-ch-get-nat s′-time′′ by presburger
            ultimately have ¬ ?occ h tr′
              using terms-scno[unfolded stamp-current-not-occurs-def]
              by blast
            then show ?thesis
              using ttt-h-h′′-tr′ by argo
          qed
        qed
        have ?occ-ts h′ tsnextp = ?occ-ts h′ (Some tsr)
            using hosc-h-h′-trs i-term-to-term-only-stamp-changed not-occ-tdestr
tdestr-acyclic ttt-tsr′ by auto
        then have ∗∗: ?occ-ts h′ tsnextp = ?occ-ts h (Some tsr)
          using i-terms-to-terms-only-stamp-changed
            hosc-h-h′-trs i-terms-sublists.self terms-acyclic by presburger
        show ?thesis
        apply (subst i-occ-p.simps,
              simp add: lookup-def tap-def bind-def return-def execute-heap
              get-tsr exec-ifind IH-exec IHn-exec not-occ-tdestr ∗∗ scno′′-tsr)
          using ∗ by simp
      qed
    qed
  qed
 }
 then show ?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* = *ITerm* (*s1*, *None*, *td*)
and *get-vr*: *Ref.get h vr* = *ITerm* (*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* (′′*x*′′, *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* (′′*x*′′, *int* (*addr-of-ref vr*)) *fun-term′*
have *scno*: *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′* = *ITerm* (*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 *maxstamp-tr′*: *i-maxstamp h* (*Some tr′*) ≤ *i-maxstamp 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 $s' \leq$ *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 $s' \leq Ref.get\ h\ time$ **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