Imperative Self-Adjusting Computation

Umut A. Acar  Amal Ahmed  Matthias Blume

Toyota Technological Institute at Chicago
{umut,amal,blume}@tti-c.org

Abstract

Self-adjusting computation enables writing programs that can automatically and efficiently respond to changes to their data (e.g., inputs). The idea behind the approach is to store all data that can change over time in modifiable references and to let computations construct traces that can drive change propagation. After changes have occurred, change propagation updates the result of the computation by re-evaluating only those expressions that depend on the changed data. Previous approaches to self-adjusting computation require that modifiable references be written at most once during execution—this makes the model applicable only in a purely functional setting.

In this paper, we present techniques for imperative self-adjusting computation where modifiable references can be written multiple times. We define a language SAIL (Self-Adjusting Imperative Language) and prove consistency, i.e., that change propagation and from-scratch execution are observationally equivalent. Since SAIL programs are imperative, they can create cyclic data structures. To prove equivalence in the presence of cycles in the store, we formulate and use an untyped, step-indexed logical relation, where step indices are used to ensure well-foundedness. We show that SAIL accepts an asymptotically efficient implementation by presenting algorithms and data structures for its implementation. When the number of operations (reads and writes) per modifiable is bounded by a constant, we show that change propagation becomes as efficient as in the non-imperative case. The general case incurs a slowdown that is logarithmic in the maximum number of such operations. We describe a prototype implementation of SAIL as a Standard ML library.

1 Introduction

Self-adjusting computation concerns the problem of updating the output of a computation while its input undergoes small changes over time. Recent work showed that a combination of dynamic dependence graphs Acar et al. [2006c] and a particular form of memoization Acar et al. [2003] can be combined to update computation orders of magnitudes faster than re-computing from scratch Acar et al. [2006b]. The approach has been applied to a number of problems including invariant checking Shankar and Bodik [2007], computational geometry and motion simulation Acar et al. [2006d], and statistical inference on graphical models Acar et al. [2007c].

In self-adjusting computation, the programmer stores all data that can change over time in modifiable references or modifiables for short. Modifiables are write-once references: programs can read a modifiable as many times as desired but must write it exactly once. After a self-adjusting program is executed, the programmer can change the contents of modifiables and update the computation by performing change propagation. For efficient change propagation, as a program executes, an underlying system records the operations on modifiables in a trace and memoizes the function calls. Change propagation uses the trace, which is represented by a dynamic dependence graph, to identify the reads of changed modifiables and re-evaluates them. When re-evaluating a read, change-propagation re-uses previous computations via memoization.

By requiring that modifiables be written exactly once at the time of their creation, the approach ensures that self-adjusting programs are purely functional; this enables 1) inspecting the values of modifiables at any time in the past, 2) re-using computations via memoization. Since change propagation critically relies
on these two properties, it was not known if self-adjusting computation could be made to work in an imperative setting. Although purely functional programming is fully general (i.e., Turing complete), it can be asymptotically slower than imperative models of computation Pippenger [1997]. Also, for some applications (e.g., graphs), imperative programming can be more natural.

In this paper, we generalize self-adjusting computation to support imperative programming by allowing modifiables to be written multiple times. We describe an untyped language, called SAIL (Self-Adjusting Imperative Language), that is similar to a higher-order language with mutable references. As in a conventional imperative language, a write operation simply updates the specified modifiable. A read operation takes the modifiable being read and the expression (the body of the scoped read) that uses the contents of the modifiable. To guarantee that all dependencies are tracked, SAIL ensures that values that depend on the contents of modifiables are themselves communicated via modifiables by requiring that read operations return a fixed (unit) value. Compared to a purely functional language for self-adjusting computation, SAIL is somewhat simpler because it does not have to enforce the write-once requirement on modifiables. In particular, purely functional languages for self-adjusting computation make a modal distinction between stable and changeable computations, e.g., Acar et al. [2006c], which is not necessary in SAIL.

We describe a Standard ML library for SAIL that allows the programmer to transform ordinary imperative programs into self-adjusting programs (Section 2). The transformation requires replacing the references in the input with modifiables and annotating the code with SAIL primitives. As an example, we consider depth-first-search and topological sorting on graphs. The resulting self-adjusting programs are algorithmically identical to the standard approach to DFS and topological sort, but via change propagation they can respond to changes faster than a from-scratch execution when the input is changed.

We formalize the operational semantics of SAIL and its change propagation semantics (Section 3). In the operational semantics, evaluating an expression returns a value and a trace that records the operations on modifiables. The semantics models memoization by using non-determinism: a memoized expression can either be evaluated to a value (a memo miss) or its trace and result can be re-used by applying change propagation to the trace of a previous evaluation (a memo hit).

We prove that the semantics of SAIL is consistent, i.e., that any two evaluations of the same expressions are observationally (or contextually) equivalent (Section 4). Since SAIL programs are imperative, it is possible to construct cycles in the store. This makes reasoning about equivalence challenging. In fact, in our prior work Acar et al. [2007b], we proved consistency for purely functional self-adjusting computation by taking critical advantage of the absence of cycles. More specifically, we defined the notion of equivalence by using lifting operations that eliminated the store by substituting the contents of locations directly into expressions; lifting cannot even be defined in the presence of cycles.

Reasoning about equivalence of programs in the presence of mutable state has long been recognized as a difficult problem. The problem has been studied extensively starting with ALGOL-like languages which have local updatable variables but no first-class references O’Hearn and Tennent [1995], Sieber [1993], Pitts [1996]. The problem gets significantly harder in the presence of first class references and dynamic allocation Pitts and Stark [1993], Stark [1994], Benton and Leperchey [2005] and harder still in the presence of cyclic stores. We know of only two recent results for proving equivalence of programs with first-class mutable references and cyclic stores, a proof method based on bisimulation Koutavas and Wand [2006] and another on (denotational) logical relations Bohr and Birkedal [2006] (neither of which is immediately applicable to proving the consistency of SAIL).

We prove consistency for imperative self-adjusting programs using syntactic logical relations, that is, logical relations based on the operational semantics of the language (not on denotational models). We use logical relations that are indexed not by types (since SAIL is untyped), but by a natural number that, intuitively, records the number of steps available for future evaluation Appel and McAllester [2001], Ahmed [2006]. The stratification provided by the step indices is essential for modeling the recursive functions (available via encoding fix) and cyclic stores present in the language.

We show that SAIL accepts an asymptotically efficient implementation by describing data structures for supporting its primitives and by giving a change propagation algorithm (Section 5). For each modifiable we keep all of its different contents, or versions, over time. The write operations create the versions. For
example, writing the values values 0 and then 5 into the same modifiable \( m \) creates two versions of \( m \). This versioning technique, which is inspired by previous work on persistent data structures Driscoll et al. [1989], enables keeping track of the relationship between read operations and the values that they depend on. We keep the versions of a modifiable in a \textit{version-set} and its readers in a \textit{reader-set}. We represent these sets as searchable time-ordered sets that support various operations such as \textit{find}, \textit{insert}, and \textit{delete}, all in time logarithmic in the size of the set. Using these data structures, we describe a change-propagation algorithm that implements the trace-based change propagation of the SAIL semantics efficiently. In particular, for computations that write to each modifiable a constant number of times and read each location a constant number of times, we show that change propagation is asymptotically as efficient as in the non-imperative case. When the number of writes is not bounded by a constant, then change propagation incurs a logarithmic overhead in the maximum number of writes and reads to any modifiable.

2 Programming with Mutable Modifiables

We give an overview of our framework based on our ML library and a self-adjusting version of depth-first search on graphs.

2.1 The ML library

Figure 1 shows the signature of our library. The library defines the equality type for modifiables \( \text{'}a\text{ mod} \) and provides functions to create (\text{mod}), read (\text{read}), write (\text{write}), and hash (\text{hashMod}) modifiables. For memoization, the library provides the \text{memo} function. We define a \text{self-adjusting program} as a Standard ML program that uses these functions. In addition, the library provides \textit{meta functions} for initializing the library (\text{init}), inspecting and changing modifiable references (\text{deref, change}) and propagating changes (\text{propagate}). Meta functions cannot be used by a self-adjusting program.

A modifiable reference is created by the \text{mod} function that takes a conservative equality test on the contents of the modifiable and returns an uninitialized modifiable. A conservative equality function returns \text{false} when the values are different but may return \text{true} or \text{false} when the values are the same. This equality function is used to stop unnecessary change propagation by detecting that a \text{write} or \text{change} operation does not change the contents of a modifiable. For each modifiable allocated, the \text{mod} function generates a unique integer tag, which is returned by the \text{hashMod} function. The hashes are used when memoizing function calls. A \text{read} takes the modifiable to be read and a \text{reader} function, applies the contents of the modifiable to the reader, and returns unit. By making \text{read} operations return unit, the library ensures that no variables other than those bound by the readers can depend on the contents of modifiables. This forces readers to communicate by writing to modifiable references and makes it possible to track all dependencies by recording the operations on modifiables.
datatype node = empty | node of int * bool ref * node ref * node ref

fun depthFirstSearch f root = let
    fun dfs rn =
      case !rn of
        empty => f (NONE, NONE)
      | node (id, rv, rn1, rn2) =>
        if !rv then
          let
            val () = rf := true
            val res1 = dfs rn1
            val res2 = dfs rn2
          in
            if !rv then
              let
                val () = write rf true
                val rres1 = dfs rn1
                val rres2 = dfs rn2
              in
                read rres1 (fn res1 =>
                  read rres2 (fn res2 =>
                    write rres (f (SOME id, SOME(res1, res2))))
                )
              end
            else
              write rres NONE
          end
        else
          NONE
        end in
    dfs root
  end

Figure 2: The code for ordinary (left) and self-adjusting (right) depth-first search programs.

The memo function creates a memo table and returns a memoized function associated with that memo table. A memoized function takes a list of arguments and the function body, looks up the memo table based on the arguments, and computes and stores the result in the memo table if the result is not already found in it. To facilitate efficient memo lookups, memoized functions must hash their arguments to integers uniquely. This can be achieved by using standard boxing or tagging techniques.

Our library does not provide mechanisms for statically or dynamically ensuring the correctness of self-adjusting programs, i.e., for ensuring that change propagation updates computations correctly. We instead expect the programmer to adhere to certain correct-usage requirements. In particular, correct-usage requires that all the free variables of a memoized function be listed as an argument to the memoized function and be hashed, and that a memoized function be used to memoize only one function. It may be possible to enforce these requirements but this may require a significant burden on the programmer Acar et al. [2006a].

2.2 Writing Self-Adjusting Programs

As an example of how modifiables can be used, Figure 2 shows the complete code for a depth-first-search (DFS) function on a graph with ordinary references (left) and with modifiable references (right). A graph is defined to be either empty or a node consisting of an integer identifier, a boolean visited flag, and two pointers to its neighbors. In the non-self-adjusting code, the neighbor pointers and the visited flag are placed in ordinary references; in the self-adjusting code, these are placed in modifiable references.

The static depthFirstSearch function takes a visitor function f and the root of a graph as its arguments and performs a depth-first-search (dfs) starting at the root by visiting each node that has not been previously visited. The dfs function visits a node by allocating a modifiable that will hold the result and reading the node pointed to by its argument. If the node is empty, then the visitor is applied with arguments that indicate that the node is empty. If the node is not empty, then the visited flag is read. If the flag is set to true, then the node was visited before and the function writes NONE to its result. If the flag is false (the node was not visited before), then the flag is first set to true, the neighboring nodes are visited recursively, the results of the neighbors along with the identity of the visited node are passed to the visitor (f) to compute the result, and the result is written. Since during a DFS, the visited flag starts with value false and is later set to true, the DFS requires updateable modifiables.
We transform the static code into self-adjusting code by first replacing all the references in the input (the graph) with modifiables. We then replace dereference operations with read operations. Since the contents of modifiables are accessible only locally within the body of the read, when inserting a read operation, we identify the part of the code that becomes the body of the read. In our example, we insert a read for accessing the visitor flag (line 12). Since read operations can only return unit, we need to allocate a modifiable for the result to be written (line 7). To allocate the result modifiable, we use an equality function on the result type provided as an argument to the depthFirstSearch function. We finish the transformation by memoizing the dfs function. This requires creating a memo function (line 11) and applying it to the recursive branch of the case statement: since the base case performs constant work, it does not benefit from memoization. For brevity, in the code, we use # for hashMod.

One application of DFS is topological sort, which requires finding an ordering of nodes where every edge goes from a smaller to a larger node. As an example, consider the graphs in Figure 3, where each node is tagged with the first and the last time they were visited by the DFS algorithm. For node A these are 1 and 16, respectively. The topological sort of a graph can be determined by sorting the nodes according to their last-visit time, e.g., Cormen et al. [1990]. In Figure 3, the left graph is sorted as A,B,C,D,E,F,G,H and the right graph is sorted as A,B,F,G,H,C,D,E. We can compute the topological sort of a graph by using the depthFirstSearch function (Figure 2). To do this, we first define the result type and its equality function, in this case a list consisting of the identifiers of the nodes in topologically sorted order. Since modifiable references accept equality, we can use ML’s “equals” operator for comparing modifiables as follows.

datatype 'a list = nil | cons 'a * ('a list) mod
fun eqList (a,b) =
    case (a,b) of
        (nil,nil) => true
        (ha::ta,hb::tb) => ha=hb andalso ta=tb
    | _ => false
We then write a visitor function that concatenates its argument lists (if any), and then inserts the node being visited at the head of the resulting list. This ordering corresponds to the topological sort ordering because a node is added to the beginning of the ordering after all of its out-edges are traversed. We can sort a graph with depthFirstSearch by passing the eqList function on lists and the visitor function.

2.3 Propagation

In self-adjusting computation, after the programmer executes a program, she can change the input to the program and update the output by performing change propagation. The programmer can repeat this change-and-propagate process with different input changes as many times as desired. Figure 4 shows an example
structure SAIL: SELF_ADJUSTING = ...
structure Graph =
  struct
    fun fromFile s = ...
    fun node i = ...
    fun newEdgeFrom (i) = ...
    fun DFSVisitSort = ...
  end
fun test (s,i,j) =
  let
    val _ = SAIL.init ()
    val (root,graph,n) = Graph.fromFile s
    val r = depthFirstSearch Graph.DFSVisitSort (root)
    val nr = Graph.newEdgeFrom (i)
    val () = SAIL.change nr (Graph.node j)
    val () = SAIL.propagate ();
  in
    r
  end

Figure 4: Example of changing input and change propagation.

that uses depthFirstSearch to perform a topological sort. The example assumes an implementation of a library, SAIL, that supplies primitives for self-adjusting computation and a Graph library that supplies functions for constructing graphs from a file, finding a node, making an edge starting at a particular node, etc.

The test function first constructs a graph from a file and then computes its topological sort using depthFirstSearch. The DFSVisitSort function from the Graph library, whose code we omit, is a typical visitor that can be used with depthFirstSearch as described above. After the initial run is complete, the test function inserts a new edge from node i to node j as specified by its arguments. To insert the new edge, the function first gets a modifiable for inserting the edge at i and then changes the modifiable to point to node j. The function then performs change-propagation to update the result. For example, after sorting the graph in Figure 3, we can insert the edge from H to C and update the topological-sort by performing change propagation.

2.4 Performance

We show that the self-adjusting version of the standard DFS algorithm responds to changes efficiently. For the proof, we introduce some terminology. Let \( G \) be an ordered graph, i.e., a graph where the out-edges are totally ordered. Consider performing a DFS on \( G \) such that the out-edges of each node are visited in the order specified by their total order. Let \( T \) be the DFS-tree of the traversal, i.e., the tree that consists of the edges \((u,v)\) whose destinations \( v \) are not visited during the time that the edge is traversed. Consider now a graph \( G' \) that is obtained from \( G \) by inserting/deleting an edge. Consider performing DFS on \( G' \) and let \( T' \) be its DFS-tree. We define the affected nodes as the nodes of \( T \) (or \( G \)) whose paths to the root are different in \( T \) and \( T' \). Figure 3 shows two example graphs \( G \) and \( G' \), where \( G' \) is obtained from \( G \) by inserting the edge \((H,C)\). The DFS-trees of these graphs consist of the thick edges. The affected nodes are C, D, and E, because these are the only nodes that are accessible through the newly inserted edge \((H,C)\) from the root A.

Based on these definitions, we prove that DFS takes time proportional to the number of affected nodes. Since the total time will depend on the visitor \( f \) that determines the result of the DFS, we first show a bound disregarding visitor computations. We then consider a particular instantiation of the visitor for performing topological sort and show that the same bound holds for this application as well. For the proofs, which will be given in Section 5.5 after the change-propagation algorithm has been described, we assume that each node has constant out-degree.
Theorem 2.1 (DFS Response Time). Disregarding the operations performed by the visitor, the \texttt{depthFirstSearch} program responds to changes in time $O(m)$, where $m$ is the number of affected nodes after an insertion/deletion.

Our bound for topological sort is the same as that for DFS, i.e., we only pay for those nodes that are affected.

Theorem 2.2 (Topological Sort). Change propagation updates the topological sort of a graph in $O(m)$ time where $m$ is the number of affected nodes.

2.5 Implementation

We present techniques for implementing the library efficiently in Section 5. A prototype implementation of the library is available on the web page of the first author.

3 The Language

In this section, we present our Self-Adjusting Imperative Language SAIL. Since our consistency proof does not depend on type safety, we leave our language untyped. For simplicity, we assume all expressions to be in A-normal form Felleisen and Hieb [1992]. Unlike in our previous work where it was necessary to enforce a write-once policy for modifiable references, we do not distinguish between stable and changeable computations. This simplifies the syntax of SAIL considerably. Modifiable references now behave much like ordinary ML-style references: they are initialized at creation time and can be updated arbitrarily often by the program.

3.1 Values and expressions

The syntax of the language is shown in Figure 5. Value forms $v$ include unit $()$, variables $x$, integers $n$, locations $l$, $\lambda$-abstractions $\lambda x.e$, pairs of values $(v_1, v_2)$, and injections $\text{inl} v$ and $\text{inr} v$ into a sum.

Expressions $e$ that are not themselves values $v$ can be applications of primitive operations $o (v_1, \ldots, v_n)$ (where $o$ is something like $+$, $-$, or $<$), function applications $v_1 v_2$, allocation and initialization of modifiable references ($\text{mod} v$), scoped read-operations $\text{let} x = v_1 \in e$ that bind the value stored at the location given by $v$ to variable $x$ and execute $e$ in the scope of $x$, write-operations $\text{write} v_1 \leftarrow v_2$ that store $v_2$ into the location given by $v_1$, let-bindings $\text{let} x = v_1 \in e_2$, projections from pairs ($\text{fst} v$ and $\text{snd} v$), and case analysis on sums ($\text{case} v$ of $\text{inl} x_1 \Rightarrow e_1 | \text{inr} x_2 \Rightarrow e_2$). The form $\text{memo} e$ marks an expression that is subject to memoization: evaluation of $e$ may take advantage of an earlier evaluation of the same expression, possibly using change propagation to account for changes to the store.

3.2 Traces

Change propagation requires access to the “history” of an evaluation. A history is represented by a trace, and every evaluation judgment specifies an output trace. The syntax of traces is as follows:

$$T ::= \varepsilon \mid \text{let} T_1 T_2 \mid \text{mod} l \leftarrow v \mid \text{read} l \leftarrow x = v, e \mid \text{write} l \leftarrow v$$
Traces can be empty ($\varepsilon$), combine two sub-traces obtained by evaluating the two sub-terms of a let-form ($\text{let } T_1 T_2$), or record the allocation of a new modifiable reference $l$ that was initialized to $v$ ($\text{mod } l \leftarrow v$). A trace of the form $\text{read } l \leftarrow v.e$ indicates that reading $l$ produced a value $v$ that was bound to $x$ for the evaluation of $e$ which produced the sub-trace $T$. Finally, the trace $\text{write } l \leftarrow v$ records that an existing location $l$’s contents have been updated to contain the new value $v$. The only difference between the traces $\text{write } l \leftarrow v$ and $\text{mod } l \leftarrow v$ is that the former is not counted as an allocation: $\text{alloc}(\text{write } l \leftarrow v) = \emptyset$ while $\text{alloc}(\text{mod } l \leftarrow v) = \{l\}$. In general, $\text{alloc}(T)$ denotes the set of locations that appear in $\text{mod } l \leftarrow v$ within the trace $T$ (formal definition elided).

### 3.3 Stores

As mentioned in Section 1, the actual implementation of the store maintains multiple time-stamped versions of the contents of each cell. In our formal semantics, the version-tracking store is present implicitly in the trace: to look up the current version of the contents of each location, one only needs the most recent write operation on $l$.

Formalizing this idea, while possible, would require the semantics to pass around a representation of a global trace representing execution from the very beginning up to the current program point. Moreover, such a trace would need more internal structure to be able to support change propagation.

The alternative formalization that we use here takes advantage of the following observation: at any given point in time, we only need the current version of the global version-tracking store. We refer to this view as “the store” because it simply maps locations to values. Thus, instead of representing the version store explicitly, our semantics keeps track of the changes to the current view of the version store by manipulating ordinary stores.

### 3.4 Operational Semantics

The operational semantics consists of rules for deriving evaluation judgments of the form $\sigma, e \downarrow^k v, \sigma', T$, which should be read as: “In store $\sigma$, expression $e$ evaluates in $k$ steps to value $v$, resulting in store $\sigma'$.

The computation is described by trace $T$.” Step counts are irrelevant to the evaluation itself, but we will use them later in the logical relation we formulate for reasoning about consistency. The rules for deriving evaluation judgments are shown in Figure 6.

The rule for memo has a premise of the form $\sigma, T \hookrightarrow^k \sigma', T'$. This is a change propagation judgment and should be read as: “The computation described by $T$ is adjusted in $k$ steps to a new computation described by $T'$ and a corresponding new store $\sigma'$.” The rules for deriving change propagation judgments are shown in Figure 7. Memoization is modeled (Figure 6) by starting at some “previous” evaluation of $e$ (in some other store $\sigma_0$) that is now adjusted to the current store $\sigma$.

We use the following abbreviations:

$$
\begin{align*}
\sigma, e \downarrow v, \sigma', T & \overset{\text{def}}{=} \exists k. \sigma, e \downarrow^k v, \sigma', T \\
\sigma, T \hookrightarrow \sigma', T' & \overset{\text{def}}{=} \exists k. \sigma, T \hookrightarrow^k \sigma', T'
\end{align*}
$$

**Evaluation rules.** Values and primitive operations, which are considered pure, add nothing to the trace (rules value, primop). Application evaluates the body of the function after substituting the argument for the formal parameter. The resulting trace is the one produced while evaluating the body (rule apply). A let-expression is evaluated by running the two sub-terms in sequence, substituting the result of the first for the bound variable in the second. The trace is the concatenation (using the let-construct for traces) of the two sub-traces (rule let). Evaluating mod $v$ picks a location $l$, stores $v$ at $l$, and returns $l$. This action (including $l$ and $v$) is recorded in the trace (rule mod). A read-expression substitutes the value stored at location to be read for the bound variable in the body. Evaluating the resulting body must return the unit value ($\bot$). The read-operation—including location, bound variable, value read, and body—is recorded in the trace (rule read). A write-operation modifies the store by associating the location to be written with the new value. The result of a write is unit. Both value and location are recorded in the trace (rule write).
Evaluation of a memo-expression is non-deterministic. When evaluating an expression \( \text{memo } e \) in a store \( \sigma \), we can either reuse an evaluation of \( e \) in some arbitrary ("previous") store \( \sigma_0 \)—not necessarily the same as the current store \( \sigma \)—provided that the evaluation can be adjusted to the current store via change propagation, or we can evaluate \( e \) from scratch in the current store \( \sigma \). The corresponding evaluation rules, \text{memo/hit} and \text{memo/miss} respectively, may be written as follows:

\[
\frac{\sigma_0, e \downarrow v, \sigma', T}{\sigma, \text{memo } e \downarrow v, \sigma', T} \quad (\text{memo/hit})
\]

\[
\frac{\sigma, e \downarrow v, \sigma', T}{\sigma, \text{memo } e \downarrow v, \sigma', T} \quad (\text{memo/miss})
\]

Our evaluation rule for \text{memo} (Figure 6) does not distinguish between memo hits and memo misses. The high degree of freedom in the choice of \( \sigma_0 \) makes a memo miss a special case of a memo hit: in the \text{memo-rule}, to simulate a memo miss we pick \( \sigma_0 = \sigma \). If evaluation of \( e \) in \( \sigma \) produces a trace \( T \), then change propagation of trace \( T \) in store \( \sigma \) does nothing (i.e., yields the same \( \sigma \) and \( T \)). Hence, picking \( \sigma_0 = \sigma \) captures the essence of the memo miss—evaluation proceeds directly in the current store \( \sigma \), not some other "previous" store \( \sigma_0 \).

The rules \text{fst} and \text{snd} are the standard projection rules. Projections are pure and, therefore, add nothing to the trace. Similarly, rules \text{case/inl} and \text{case/inr} are the standard elimination rules for sums. In each case, the trace records whatever happened in the branch that was taken. The case analysis itself is pure and does not need to be recorded.
of the evaluation rule for model as important, but disallowing the possibility (e.g., by adding a side-condition of previous allocations. The ability to allocate an existing garbage location during ordinary evaluation is not during change propagation is crucial, since otherwise change propagation would not be able to retain any locations must not be reachable from the initial expression (see Section 3.5).

Discussion. Our rules are given in a non-deterministic, declarative style. For example, the memo-rule does not place any special requirements on the location being allocated, i.e., the location could already exist in the store. For correctness, however, we insist that all locations allocated during the course of the entire program run be pairwise distinct. (This is enforced by side conditions on our let-rules.) Furthermore, allocated locations must not be reachable from the initial expression (see Section 3.5).

As in our previous work Acar et al. [2007b], the ability for mod to allocate an existing (garbage-) location during change propagation is crucial, since otherwise change propagation would not be able to retain any previous allocations. The ability to allocate an existing garbage location during ordinary evaluation is not as important, but disallowing the possibility (e.g., by adding a side-condition of \( l \not\in \text{dom}(\sigma) \) to the premise of the evaluation rule for mod) would have two undesirable effects: it would weaken our result by reducing...
\[
\sigma : \eta \rightsquigarrow \mathcal{L} \quad \text{def} \quad \mathcal{L} = \eta \cup \bigcup_{l \in \mathcal{L}} FL(l) \quad \land \quad \text{dom}(\sigma) \supseteq \mathcal{L} \quad \land \quad \forall L^1 \subseteq \mathcal{L}. \quad \eta \subseteq L^1 \quad \land \quad (\forall l \in L^1. \ FL(l) \subseteq L^1) \quad \implies \quad \mathcal{L} = L^1
\]

Figure 8: Store Reachability Relation

the number of possible evaluations, and it would make our formal framework for reasoning about program equivalence more complicated.

Evaluating a \texttt{read}-form returns the unit value. Therefore, the only way for the body of the \texttt{read}-form to communicate to the rest of the program is by writing into other modifiable references, or even possibly the same reference that it read. This convention guarantees stability of values and justifies the rule for \texttt{memo} where we return the value computed during an arbitrary “earlier” evaluation in some other store \(\sigma_0\). The value so computed cannot actually depend on the contents of \(\sigma_0\). It can, of course, be a location pointing to values that do depend on \(\sigma_0\), but those will be adjusted during change propagation.

3.5 Reachability and Valid Evaluations

Consistency holds only for so-called \textit{valid evaluations}. Informally, an evaluation is valid if it does not allocate locations reachable from the initial expression \(e\). Our technique for identifying the locations reachable from an expression is based on the technique used by Ahmed et al. [2005] in their work on substructural state. Let \(FL(e)\) be the \textit{free locations} of \(e\), i.e., those locations that are subexpressions of \(e\). The locations \(FL(e)\) are said to be \textit{directly accessible} from \(e\). The store reachability relation \(\sigma : \eta \rightsquigarrow \mathcal{L}\) (Figure 8) allows us to identify the set of locations \(\mathcal{L}\) reachable in a store \(\sigma\) from a set of “root” locations \(\eta\). The relation \(\sigma : \eta \rightsquigarrow \mathcal{L}\) requires that the reachable set \(\mathcal{L}\) include the root locations \(\eta\) as well as all locations directly accessible from each \(l \in \mathcal{L}\). It also ensures that all reachable locations are in \(\sigma\). Furthermore, it requires that \(\mathcal{L}\) be \textit{minimal}—that is, it ensures that the set \(\mathcal{L}\) does not contain any locations not reachable from the roots.

Thus, \(\mathcal{L}\) is the set of locations reachable from an expression \(e\) in a store \(\sigma\) iff \(\sigma : FL(e) \rightsquigarrow \mathcal{L}\). We define valid evaluations \(\sigma, e \Downarrow^k v, \sigma', T\) as follows.

\textbf{Definition 3.1 (Valid Evaluation).}

\[
\sigma, e \Downarrow^k v, \sigma', T \quad \text{def} \quad \sigma, e \Downarrow^k v, \sigma', T \quad \land \quad \exists \mathcal{L}. \quad \sigma : FL(e) \rightsquigarrow \mathcal{L} \quad \land \quad \mathcal{L} \cap \text{alloc}(T) = \emptyset
\]

4 Consistency via Logical Relations

In this section, we prove that the semantics of SAIL is \textit{consistent}—i.e., that the non-determinism in the operational semantics is harmless—by showing that any two \textit{valid} evaluations of the same program in the same store yield observationally (contextually) equivalent results.

4.1 Contextual Equivalence

A context \(C\) is an expression with a hole in it. We write \(C : (\Gamma)\) to denote that \(C\) is a closed context (i.e. \(FV(C) = \emptyset\)) that provides bindings for variables in the set \(\Gamma\). Thus, if \(FV(e) \subseteq \Gamma\), then \(C[e]\) is a closed term. We write \(\sigma : \eta\) as shorthand for: \(\exists \mathcal{L}. \quad \sigma : \eta \rightsquigarrow \mathcal{L}\). We say \(e_1\) contextually approximates \(e_2\) if, given an arbitrary \(C\) that provides bindings for the free variables of both terms, running \(C[e_1]\) in a store \(\sigma\) (that contains all the appropriate roots) returns \(n\), then (1) there exists an evaluation for \(C[e_2]\) in \(\sigma\), and (2) all such evaluations also return \(n\).
Definition 4.1 (Contextual Equivalence).

Let $\Gamma = FV(e_1) \cup FV(e_2)$.

$$
\Gamma \vdash e_1 \precctx e_2 \overset{\text{def}}{=} \forall C : (\Gamma). \forall \sigma, \eta, n.
\eta = FL(C) \cup FL(e_1) \cup FL(e_2) \land \sigma : \eta \land
\sigma, C[e_1] vok n, -,- \implies
(\exists v. \sigma, C[e_2] vok v, -,-) \land
(\forall v. \sigma, C[e_2] vok v, -,- \implies n = v)
$$

$$
\Gamma \vdash e_1 \approxctx e_2 \overset{\text{def}}{=} \Gamma \vdash e_1 \precctx e_2 \land \Gamma \vdash e_2 \precctx e_1
$$

4.2 Proving Consistency

Having defined contextual equivalence, we can be more precise about what we mean by consistency: if $e$ is a closed program, we wish to show that $\emptyset \vdash e \approxctx e$, which means that if we run $C[e]$ (where $C$ is an arbitrary context) twice in the same store $\sigma$, then we get the same result value $n$.

It is difficult to prove $\emptyset \vdash e \approxctx e$ directly due to the quantification over all contexts in the definition of $\approxctx$. Instead we use the standard approach of using a logical relation in order to prove contextual equivalence—that is, we will show that any term $e$ is logically related to itself (Theorem 4.5), and that the latter implies that $e$ is contextually equivalent to itself (Theorem 4.6).

Logical relations specify relations on terms, typically via structural induction on the syntax of types. (Since $\text{SAIL}$ is untyped, we will define a logical relation via induction on (available) steps as discussed below.) Thus, for instance, logically related functions take logically related arguments to related results, while logically related pairs consist of components that are related pairwise. Two expressions are logically related if either they both diverge, or they both terminate and yield related values. For any logical relation, one must first prove the so-called Fundamental Property of the logical relation (also called the Basic Lemma) which says that any (well-typed) term is related to itself. If the logical relation is intended to be used for contextual equivalence, the next step is to show that if two terms are logically related, then they are contextually equivalent, which typically follows from the Fundamental Property. Since our logical relation is intended to be used to prove consistency, we will show that any term $e$ that is logically related to itself—by the Fundamental Property of our logical relation, this is true of every $e$—is contextually equivalent to itself (i.e., $e \approxctx e$).

The two sources of non-determinism in $\text{SAIL}$ are allocation and memoization. Since they differ in nature, we deal with them using different techniques. The non-determinism caused by allocation only concerns the identity of locations. We handle this by maintaining a bijection between the locations allocated in different runs of the same program. We use the meta-variable $S$ to denote sets of location pairs. We define the following abbreviations:

$$
S^1 \equiv \{ l_1 \mid (l_1, l_2) \in S \} \quad S^2 \equiv \{ l_2 \mid (l_1, l_2) \in S \}
$$

We define the set of location bijections as follows:

$$
bij(S) \overset{\text{def}}{=} \forall l \in S^1. \exists l_2 \in S^2, (l_1, l_2) \in S \land
\forall l \in S^2. \exists l_1 \in S^1, (l_1, l_2) \in S
$$

$$
\text{LocBij} = \{ S \in 2^{\text{Locs} \times \text{Locs}} \mid bij(S) \}
$$

When both runs execute a $\text{mod} v$, we extend the bijection with the pair of locations $(l_1, l_2)$ returned by $\text{mod} v$. Notice that it will always be possible to prove that the result is a bijection because valid evaluations cannot reuse reachable locations.

If we start with identical programs (modulo the location bijection), then they will execute in lock-step until they encounter a $\text{memo}$, at which point the derivation trees for the evaluation judgments can differ dramatically. There is no way of relating the two executions directly. Fortunately, this is not necessary, since they need to be related only after change propagation has brought them back into sync. A key insight is that at such sync points it is always possible to establish a bijection between those locations that are
reachability from each of the two running programs. To show that change propagation does, in fact, bring the two executions into sync, we prove that each memo hit can be replaced by a regular evaluation (Section 4.7).

Our logical relation for consistency of SAIL is based on the step-indexed logical relations for purely functional languages by Appel and McAllester [2001] and Ahmed [2006]. In those models, the relational interpretation $V[\tau]$ of a (closed) type $\tau$ is a set of triples of the form $(k, v_1, v_2)$ where $k$ is a natural number (called the approximation index or step index) and $v_1$ and $v_2$ are closed values. Intuitively, $(k, v_1, v_2) \in V[\tau]$ says that in any computation running for no more than $k$ steps, $v_1$ approximates (or “looks like”) $v_2$. Informally, we say that $v_1$ and $v_2$ are related for $k$ steps.

A novel aspect of the logical relation that we present below is that it is untyped—that is, it is indexed only by step counts, unlike logical relations in the literature which are always indexed by types (or in the case of prior step-indexed logical relations—e.g., Appel and McAllester [2001], Ahmed et al. [2005], Ahmed [2006]—by both types and step counts).

Another novelty is the way in which our model tracks reachability of the stores of the two computations. The intuition is to start at those variables of each program that point into the respective stores (i.e., the roots of a tracing garbage collector), and construct graphs of the reachable memory cells by following pointers. Then the two program stores are related for $k$ steps if (1) these graphs are isomorphic, and (2) the contents of related locations (i.e., bijectively related vertices of the graphs) are related for $k - 1$ steps. (Since reading a location consumes a step, $k - 1$ suffices here.)

4.3 Related Values

The value relation $V$ specifies when two values are related. $V$ is a set of tuples of the form $(k, \psi, v_1, v_2)$, where $k$ is the step index, $v_1$ and $v_2$ are closed values, and $\psi \in \text{LocBij}$ is a local store description. A set of “beliefs” $\psi$ is a bijection on the locations directly accessible from $v_1$ and $v_2$ (i.e., $\text{FL}(v_1), \text{FL}(v_2)$). We refer to the locations in $\psi^1$ and $\psi^2$ as the roots of $v_1$ and $v_2$, respectively.

The definition of the value relation $V$ is given in Figure 10. The value $(\cdot)$ is related to itself for any number of steps. Clearly, no locations appear as subexpressions of $(\cdot)$; hence, the definition demands an empty local store description if they are equal.

Two locations $l_1$ and $l_2$ are related if the local store description says that they are related. Furthermore, from the values $l_1$ and $l_2$, the only locations that are directly accessible are, respectively, the locations $l_1$ and $l_2$ themselves. Hence, the local store description must be $\{(l_1, l_2)\}$.

The pairs $(v_1, v'_1)$ and $(v_2, v'_2)$ are related for $k$ steps if there exist local store descriptions $\psi$ and $\psi'$ such that the components of the pairs are related (i.e., $(k, \psi, v_1, v_2) \in V$ and $(k, \psi', v'_1, v'_2) \in V$) and if $\psi$ and $\psi'$ can be combined into a single set of beliefs (written $\psi \uplus \psi'$, see Figure 9). Informally, two local store descriptions $\psi$ and $\psi'$ can be combined only if they are compatible; that is, if the beliefs in $\psi$ do not contradict the beliefs in $\psi'$, or more precisely, if the union of the two bijections is also a bijection.

The left (right) injections into a sum $\text{inl} v_1$ and $\text{inr} v_2$ with local store description $\psi$ are related for $k$ steps if $v_1$ and $v_2$ are related for $k$ steps with the same local store description (i.e., $(k, \psi, v_1, v_2) \in V$).

Since functions are suspended computations, their relatedness is defined in terms of the relatedness of computations (Section 4.5). Two functions $\lambda x. e_1$ and $\lambda \bar{x}. e_2$ with local store description $\psi_c$—where $\psi_c$ describes at least the sets of locations directly accessible from the closures of the respective functions—are related for $k$ steps if, at some point in the future, when there are $j < k$ steps left to execute, and there are related arguments $v_1$ and $v_2$ such that $(j, \psi_c, v_1, v_2) \in V$, and the beliefs $\psi_c$ and $\psi_a$ are compatible, then $e_1[v_1/x]$ and $e_2[v_2/x]$ are related as computations for $j$ steps. Note that $j$ must be strictly smaller than $k$. 

\[
\psi_1 \uplus \psi_2 \overset{\text{def}}{=} \begin{cases} \psi_1 \cup \psi_2 & \text{if } (\psi_1 \cup \psi_2) \in \text{LocBij} \\ \text{undefined} & \text{otherwise} \end{cases}
\]

Figure 9: Join Local Store Descriptions
The latter requirement is essential for ensuring that the logical relation is well-founded (despite the fact that it is not indexed by types). Intuitively, \( j < k \) suffices because beta-reduction consumes a step.

Notice that the step-indexed technique of defining a logical relation yields not only a specification of the relation, but also guarantees the existence of the relation by making its well-foundedness explicit.

A crucial property of the relation \( \mathcal{V} \) is that it is closed under decreasing step index—intuitively, if \( v_1 \) "looks like" \( v_2 \) for up to \( k \) steps, then they should look alike for fewer steps.

**Lemma 4.2 (Downward Closed).**

If \( (k, \psi, v_1, v_2) \in \mathcal{V} \) and \( j \leq k \), then \( (j, \psi, v_1, v_2) \in \mathcal{V} \).

### 4.4 Related Stores

The store satisfaction relation \( \sigma_1, \sigma_2 : k \psi \rightsquigarrow S \) (see Figure 11) says that the stores \( \sigma_1 \) and \( \sigma_2 \) are related (to approximation \( k \)) at the local store description \( \psi \) and the "global" store description \( S \) (where \( S \in \text{LocBij} \)).
There must exist a set of beliefs $\sigma_1, \sigma_2 : k \psi \rightsquigarrow S$ defined as $S \in \text{LocBij} \land \exists F_\psi : S \rightarrow \text{LocBij},$

$$S = \psi \odot \bigcup_{(l_1, l_2) \in S} F_\psi(l_1, l_2) \land \text{dom}(\sigma_1) \supseteq S^1 \land \text{dom}(\sigma_2) \supseteq S^2 \land \forall (l_1, l_2) \in S. \forall j < k. \forall v \in V (j, F_\psi(l_1, l_2), \sigma_1(l_1), \sigma_2(l_2)) \in V$$

Figure 11: Related Stores

We motivate the definition of $\sigma_1, \sigma_2 : k \psi \rightsquigarrow S$ by analogy with a tracing garbage collector. Here $\psi$ corresponds to (beliefs about) the portions of the stores directly accessible from a pair of values (or multiple pairs of values, when $\psi$ corresponds to $\odot$-ed store descriptions). Hence, informally $\psi$ corresponds to the (two sets of) root locations. Meanwhile, $S$ corresponds to the set of reachable (root and non-root) locations in the two stores that would be discovered by the garbage collector.\(^1\) In the definition of $\sigma_1, \sigma_2 : k \psi \rightsquigarrow S$, the function $F_\psi$ maps each location pair $(l_1, l_2) \in S$ to a local store description. It is our intention that, for each pair of locations $(l_1, l_2)$, $F_\psi(l_1, l_2)$ is an appropriate local store description for the values $\sigma_1(l_1)$ and $\sigma_2(l_2)$. Hence, we can consider $(F_\psi(l_1, l_2))^j$ as the set of child locations traced from the contents of $l_1$ in store $\sigma_1$ (and similarly for $(F_\psi(l_1, l_2))^k$ and the contents of $l_2$ in $\sigma_2$).

Having chosen $F_\psi$, we must ensure that the choice is consistent with $S$, which should in turn be consistent with the stores $\sigma_1$ and $\sigma_2$. The “global” store description $S$ combines the local store descriptions of the roots with the local store descriptions of the contents of every pair of related reachable locations; the implicit requirement that $S$ is defined ensures that the local beliefs of the roots and all the (pairs of) store contents are all compatible. The clauses $\text{dom}(\sigma_1) \supseteq S^1$ and $\text{dom}(\sigma_2) \supseteq S^2$ require that all of the reachable locations are actually in the two stores. Finally, $(j, F_\psi(l_1, l_2), \sigma_1(l_1), \sigma_2(l_2)) \in V$ ensures that the contents of locations $l_1$ and $l_2$ (in stores $\sigma_1$ and $\sigma_2$, respectively) with the local store description assigned by $F_\psi$ are related (for $j < k$ steps).

Note that we do not require that $S$ be the minimal set of locations reachable from the roots $\psi$. Such a requirement can be added but, as we will explain, is not necessary.

### 4.5 Related Computations

The computation relation $C$ (see Figure 10) specifies when two closed terms $e_1$ and $e_2$ (with beliefs $\psi$, again corresponding to at least the locations appearing as subexpressions of $e_1$ and $e_2$) are related for $k$ steps. Informally, $C$ says that if $e_1$ evaluates to a value $v_1$ in less than $k$ steps and the evaluation is valid, then given any valid evaluation of $e_2$ to some value $v_2$, it must be that $v_1$ and $v_2$ are related (with beliefs $\psi_f$). More precisely, we pick two starting stores $\sigma_1$ and $\sigma_2$ and a global store description $S$ such that $\sigma_1, \sigma_2 : k (\psi_s \odot \psi_r) \rightsquigarrow S$, where $\psi_s$ is the set of beliefs about the two stores held by the rest of the computation, i.e., the respective continuations. If a valid evaluation of $(\sigma_1, e_1)$ (where locations allocated during evaluation are disjoint from those initially reachable in $S^1$) results in $(v_1, \sigma'_1, T_1)$ in $j < k$ steps, then given any valid evaluation $\sigma_2, e_2 \downarrow v_2, \sigma'_2, T_2$ (which may take any number of steps), the following conditions should hold:

1. There must exist a set of beliefs $\psi_f$ such that the values $v_1$ and $v_2$ are related for the remaining $(k - j)$ number of steps.
2. The following two sets of beliefs must be compatible: $\psi_f$ (what $v_1$ and $v_2$ believe) and $\psi_r$ (what the continuations believe—note that these beliefs remain unchanged).
3. There must exist a set of beliefs $S_f$ about locations reachable from the new roots $(\psi_s \odot \psi_r)$ such that the final stores $\sigma'_1$ and $\sigma'_2$ satisfy the combined set of local beliefs $(\psi_f \odot \psi_r)$ and the global beliefs $S_f$ for the remaining $k - j$ steps.

\(^1\)To be precise, our definition requires only that $S$ include the set of reachable locations.
The set of reachable locations in $\psi$ from the roots ($\mathcal{FV}$)

$$\mathcal{V}^M = \{(k, ( )) \} \cup \{(k, v)\} \cup \{(k, l)\} \cup \{(k, \lambda x.e) \mid \forall j < k. \forall v. (j, v) \in \mathcal{V}^M \implies \langle j, e[v/x]\rangle \in \mathcal{C}^M\} \cup \{(k, (v, v')) \mid (k, v) \in \mathcal{V}^M \land (k, v') \in \mathcal{V}^M\} \cup \{(k, \text{inl} v) \mid (k, v) \in \mathcal{V}^M\} \cup \{(k, \text{inr} v) \mid (k, v) \in \mathcal{V}^M\}$$

$$\mathcal{C}^M = \{(k, e) \mid \forall j < k. \forall \sigma_0, \sigma_0', \sigma, \sigma', v, T, T', j_1, j_2.
\sigma_0, e \Downarrow j_1 v, \sigma_0', T \land \sigma, T' \land j_2 \sigma', T' \land j = j_1 + j_2 \implies \sigma, e \Downarrow j v, \sigma', T' \land (k - j, v) \in \mathcal{V}^M\}$$

$$\mathcal{G}^M[\emptyset] = \{(k, \emptyset)\}$$

$$\mathcal{G}^M[\Gamma, x] = \{(k, \gamma[x \mapsto v]) \mid (k, \gamma) \in \mathcal{G}^M[\Gamma] \land (k, v) \in \mathcal{V}^M\}$$

$$\Gamma \vdash e \overset{\text{def}}{=} \forall k \geq 0. \forall \gamma. (k, \gamma) \in \mathcal{G}^M[\Gamma] \implies (k, \gamma(e)) \in \mathcal{C}^M$$

Figure 12: Logical Predicate for Memo Elimination

4. The set of reachable locations in $\sigma^1_1$ (and $\sigma^1_2$), given by $\mathcal{S}^1_1$ (and $\mathcal{S}^1_2$), must be a subset of the locations reachable before evaluating $e_1$ (respectively $e_2$)—given by $\mathcal{S}^1$ (respectively $\mathcal{S}^2$)—and the locations allocated during this evaluation.

As noted earlier, the global store description $\mathcal{S}$ is not required to be the minimal set of locations reachable from the roots ($\psi_s \odot \psi_r$), it only needs to include that set. This suffices because $\mathcal{S}^1_1$ and $\mathcal{S}^2_2$ only need to be subsets of $\mathcal{S}^1$ and $\mathcal{S}^2$ and the locations allocated during evaluation ($\text{alloc}(T_1)$ and $\text{alloc}(T_2)$). Thus, even though we may pick larger-than-necessary sets at the beginning of the evaluation, we can add to them in a minimal way as the two evaluations progress.

### 4.6 Related Substitutions and Open Terms

Let $\Gamma = \mathcal{FV}(e_1) \cup \mathcal{FV}(e_2)$. We write $\Gamma \vdash e_1 \ll e_2$ (pronounced “$e_1$ approximates $e_2$”) to mean that for all $k \geq 0$, if $\gamma_1$ and $\gamma_2$ (mapping variables in $\Gamma$ to closed values) are related substitutions with beliefs $\psi_T$ (which is the combined local store description for the values in the range of $\gamma_1$ and $\gamma_2$), then $\gamma_1(e_1)$ and $\gamma_2(e_2)$, with root beliefs $\psi_T$, are related as computations for $k$ steps. We write $\Gamma \vdash e_1 \approx e_2$ when $e_1$ approximates $e_2$ and vice versa, meaning that $e_1$ and $e_2$ are observationally equivalent.

### 4.7 Memo Elimination

We wish to prove $\Gamma \vdash e \approx e$ from which consistency—the property that any two valid evaluations of a closed term $e$ in the same store yield observationally equivalent results—follows as a corollary. The proof of $\Gamma \vdash e \approx e$ proceeds by induction on the structure of $e$ (see Theorem 4.5). Unfortunately, in the case of memo $e$, we cannot directly appeal to the induction hypothesis. To see why, consider the special case of the closed term memo $e$. We must show $(k, \{\}, \text{memo } e, \text{memo } e) \in \mathcal{C}$. Suppose (1) $\sigma_1, \sigma_2 \triangleq \{\} \odot \psi_T \rightsquigarrow S$, (2) $\sigma_1, \text{memo } e \Downarrow j v_1, \sigma_1', T_1$, and (3) $\mathcal{S}^1 \cap \text{alloc}(T_1) = \emptyset$, where $j < k$. By the induction hypothesis we have $\emptyset \vdash e \approx e$ and hence $(k, \{\}, e, e) \in \mathcal{C}$. In order to proceed, we must instantiate the latter with two related stores ($\sigma_1$ and $\sigma_2$ are the only two stores we know of that are related) and provide a valid evaluation of $e$ in the first store (i.e., we need $\sigma, e \Downarrow \ll k \rightarrow, \rightarrow, T$ where $T$ is such that $\mathcal{S}^1 \cap \text{alloc}(T) = \emptyset$). From (2), by the operational semantics, we have $\sigma_0, e \Downarrow j_1 v_1, \sigma_0', T_{01}$ and $\sigma_1, T_{01} \cap j_2 \sigma_1', T_1$, where $j = j_1 + j_2$. But we know nothing about the store $\sigma_{01}$ in which $e$ was evaluated. What we need is a derivation for $\sigma_1, e \Downarrow \ll j_1 v_1, \sigma_1', T_1$. 16
That is, we must show that evaluation in some store \( \sigma_{01} \) followed by change propagation yields the same results as a from-scratch run in the store \( \sigma_1 \).

To prove that each memo hit can be replaced by a regular evaluation (Lemma 4.4), we define a logical predicate (i.e., a unary logical relation) for memo elimination. Figure 12 defines \( V^M \) and \( C^M \) as sets of pairs \((k,v)\) and \((k,e)\) respectively, where \( k \) is the step index, \( v \) is a closed value, and \( e \) is a closed term. Essentially, \((k,e) \in C^M\) means that \( e \) has the memo-elimination property (i.e., if \( \sigma_0, e \Downarrow^{j_1} v, \sigma'_0, T \) and \( \sigma, T \curvearrowright^{j_2} \sigma', T' \), then \( \sigma, e \Downarrow^{\leq j_1 + j_2} v, \sigma', T' \)), and if the combined evaluation plus change propagation consumed \( j = j_1 + j_2 \) steps, then \( v \) has the memo-elimination property for the remaining \( k - j \) steps.

Clearly, all values \( v \) have the memo-elimination property: since \( v \) is already a value, it evaluates to itself in zero steps, producing the empty trace, which means that change propagation takes zero steps and leaves both store and trace unchanged. Since a function is a suspended computation, we must require that its body also have the memo-elimination property for one fewer step (see Figure 12).

As for any proof based on logical predicates, the first lemma that must be proved is what is known as the Fundamental Property of the logical predicate, which says that any term \( e \) satisfies the predicate. The proof of memo elimination for SAIL follows from the Fundamental Property of the logical predicate defined in Figure 12.

**Lemma 4.3 (Fundamental Property of Logical Predicate for Memo Elim).**

If \( \Gamma = FV(e) \), then \( \Gamma \vdash e \).

**Proof.** See Appendix A for detailed proof.

**Proof sketch:** By induction on the step index \( k \) and nested induction on the structure of \( e \). All cases are straightforward. The only interesting case is that of read/no ch. where we read a value \( v \) out of the store and then have to use the outer induction hypothesis to show that \( v \) has the memo-elim property for a strictly fewer number of steps before we can plug \( v \) into the body of the read, appealing to the inner induction hypothesis to complete the proof. \( \square \)

**Corollary 4.4 (Memo Elimination).** Let \( e \) be a closed term, possibly with free locations. If \( \sigma_0, e \Downarrow^{j_1} v, \sigma'_0, T \) and \( \sigma, T \curvearrowright^{j_2} \sigma', T' \), then \( \sigma, e \Downarrow^{\leq j_1 + j_2} v, \sigma', T' \).

### 4.8 Consistency

As mentioned earlier, in any proof based on logical relations, the first lemma that must be proved is what is known as the Fundamental Property of the logical relation (also known as the Basic Lemma) which says that any term \( e \) is related to itself. The proof of our main theorem, the consistency of SAIL, follows from the Fundamental Property of the logical relation for consistency that we defined in Figure 10.

**Theorem 4.5 (Fundamental Property of Logical Relation for Consistency).**

If \( \Gamma = FV(e) \), then \( \Gamma \vdash e \approx e \).

**Proof.** See Appendix B for detailed proof.

**Proof sketch:** By induction on the structure of \( e \). As explained above (Section 4.7), in the memo case we use Lemma 4.4 before we can appeal to the induction hypothesis. Other interesting cases include mod, where the valid evaluation requirement (that the evaluation not allocate locations reachable from the initial expression) is critical in order to extend the bijection on locations; write, where the fact that the locations \( l_1 \) and \( l_2 \) being written to are reachable from the initial expression guarantees that \( (l_1, l_2) \) is already in the bijection; and read, where we need to know that the values being read are related, which we can conclude from the fact that the locations being read are reachable and related, together with the fact that related locations have related contents which follows from store relatedness. \( \square \)

**Theorem 4.6 (Consistency).** If \( \Gamma = FV(e) \), then \( \Gamma \vdash e \approx_{ctx} e \).

**Proof.** See Appendix C for detailed proof.
Let us write $\downarrow^k_\emptyset$ instead of $\downarrow^k$ for evaluation judgments that have at least one derivation where every use of the memo rule picks $\sigma_0 = \sigma$. Such a derivation describes an evaluation without memo hits, i.e., that of an ordinary imperative program. Since memo elimination (Lemma 4.4) can be applied repeatedly until no more memo-hits remain, we obtain the following result, which can be seen as a statement of correctness since it relates the self-adjusting semantics to an ordinary non-adjusting semantics:

**Lemma 4.7 (Complete Memo Elimination).** Let $e$ be a closed term, possibly with free locations. If $\sigma, e \downarrow^k v, \sigma', T$, then $\sigma, e \downarrow^\leq k v, \sigma', T'$.

5 Implementation

We describe data structures and algorithms for implementing SAIL.

5.1 Data Structures

We use order-maintenance, searchable ordered-sets, and standard priority-queue data structures.

**Order Maintenance (Time Stamps).** An order-maintenance data structure maintains a set of time-stamps while supporting all of the following operations in constant time: insert a newly created time-stamp after another, delete a time stamp, and compare two time stamps Dietz and Sleator [1987].

**Searchable Time-Ordered Sets.** A time-ordered set data structure that supports the following operations.

- **new:** return an empty set.
- **build $S$:** allocate and return a data structure containing all the elements in the set $S$.
- **insert ($x, t$):** insert the element $x$ into the set at time $t$.
- **delete ($x, t$):** delete the element $x$ with time $t$ from the set.
- **find ($t$):** return the earliest element (if any) in the set at time $t$ or later.
- **prev ($t$):** return the element (if any) in the set preceding $t$.

If a data structure contains no more than one element with a given time-stamp, then we can support all operations except for build in logarithmic time (in the size of the set) by using a balanced binary search tree keyed by the time-stamps. If the size of the set is bounded by a constant, then we can support all operations in constant time by using a simple representation that keeps all elements in a list.

5.2 The Primitives

To support the self-adjusting computation primitives we maintain a global time line and a global priority queue. The time line is an instance of an order-maintenance data structure with current-time pointing to the “current time” of the computation. During the initial run, the current-time is always the last time stamp, but during change propagation it can be any time in the past. During evaluation and change propagation, we advance the time by inserting a new time stamp $t$ immediately after current-time and setting the current time to $t$. In addition to the current-time, we also maintain a time stamp called end-of-memo for memoization purposes. For change propagation, we maintain a global priority queue that contains the affected readers prioritized by their start time (we define readers more precisely below).
Figure 13: A modifiable, its writes and reads before (top) and after performing a write (bottom).

Modifiable References. We represent a modifiable reference as a triple consisting of a version-set, a reader-set, and an equality function. The version-set and the reader-set are both instances of searchable, time-ordered sets. The version set contains all the different contents of the modifiable over time—that is, it contains pairs \((v, t)\) consisting of a value \(v\) and a time stamp \(t\). The reader set of a modifiable \(l\) contains all the read operations whose source is \(l\). More precisely, the reader set contains readers, each of which is a triple \((t_s, t_e, f)\) consisting of a start time \(t_s\), end time \(t_e\), and a function \(f\) corresponding to the body of the read operation.

Based on this representation, the operations on modifiable references can be performed as follows.

\textbf{mod eq}: Create an empty version-set and an empty reader-set. Return a pointer to the triple consisting of the equality function \(eq\), the version-set, and the reader-set. Since pointers in ML are equality types, so are modifiables—they can be compared by using ML’s “equal” operator.

\textbf{read} \(l\) \(f\): Identify the version \((v, t_v)\) of the modifiable being read \(l\) that comes immediately before \texttt{current-time} by performing a combination of \texttt{find} and \texttt{prev} operations on the version set. Advance time to a new time stamp \(t_s\). Apply the body of the read \(f\) to \(v\). When \(f\) returns, advance time again to a new time-stamp \(t_e\). Insert the reader \(r\) consisting of the body and the time interval \((t_s, t_e)\) into the reader set of the modifiable being read.

\textbf{write} \(l\) \(v\): Advance time to a new \(t_w\). Create a new version with the value \(v\) being written at time \(t_w\). Insert it into the modifiable \(l\) being written. Since creating a new version for \(l\) can change the value that further reads of \(l\) may access, it can affect the readers whose start time comes after \(t_w\) but before the next version. To identify the affected readers, we check first that the value \(v\) being written is different
than that of the previous version by using the equality test of $l$; if not, then no readers are affected. Otherwise, we find the readers that come at or after $t_w$ by repeatedly performing find operations on the reader set of $l$; we stop when we find a reader that comes after the next version. We then delete these readers from the reader set and insert them into the priority queue of affected readers. Note that during the initial run, all writes take place at the most recent time. Thus, there are no affected readers.

deref $l$: Identify the version $(v,t)$ of the dereferenced modifiable $l$ at the current-time by using a find operation and return $v$.

change $l$ $v$: Identify the earliest version $(v',t)$ of the changed modifiable $l$ (at the beginning of time) by using a find operation, change the value of this version to $v$. If $v$ is equal to $v'$, then the change does not affect the readers of $l$. Otherwise, inserts all the readers of the initial version into the priority queue. The readers can be found by finding the next version (if any) and inserting all the readers between the two versions.

Figure 13 illustrates a particular representation of modifiables assuming that time stamps are real numbers and time-ordered sets are represented as sorted lists. The modifiable $x$ points to a version list (versions are drawn as squares) consisting of versions at the specified write; the versions are sorted with respect to their times. Each version points to a reader list (readers are drawn as diamonds) whose start and end times are specified. The readers stored in the reader list of a version are the ones that read that version; they are sorted with respect to their start times. Thus, all the readers take place between the time of the version and the time of the next version. For example, in the top figure, all readers of version A take place between times 0.0 and 8.0; the readers of version C take place after 8.0. The bottom figure illustrates how the reads may be arranged if we create a new version B at time 4.0. When this happens the reader that starts at time 4.0 will become affected and will be inserted into the priority queue.

Memoization and Change Propagation. Figure 14 shows the pseudo code for memoization and change propagation. These operations are based on an undo function for rolling back the effects of a computation between two time stamps.

The undo function takes a start and an end time-stamp, $t_s$ and $t_e$ respectively, and undoes the computation between $t_s$ and $t_e$ by deleting all the versions, readers, memo entries, and time stamps between $t_s$ and $t_e$. For each time stamp $t$ between $t_s$ and $t_e$, it checks if there is a version, reader, or memo entry at $t$. To delete a reader starting at $t$, we drop it from both its reader set and the queue (if it was inserted into the priority queue). To delete a memo entry that starts at $t$, we remove it from the memo table. Deleting a version is more complicated because it can affect the reads that come after it by changing the value that they read. To delete a version $(v,t)$ of a modifiable $l$ at time $t$, we first identify the time $t'$ of the earliest version of $l$ that comes after it. (If none exists, then $t'$ will be $t_\infty$.) We then find all readers between $t$ and $t'$ and insert them into the priority queue; Since they may now read a different value than they did before, these reads are affected by the deletion of the version.

To create memoized functions, the library provides a memo primitive. A memoized function has access to a memo table for storing and re-using results. Each call takes the list of the arguments of the client function (key) and the client function itself. Before executing the client function, a memo lookup is performed. If no result is found, then a start time stamp $t_s$ is created, the client is run, an end time stamp $t_e$ is created, and the result along with interval $(t_s,t_e)$ is stored in the memo table. If a result is found, then computations between the current time and the start of the memoized computation are undone, a change-propagation is performed on the computation being re-used, and the result is returned. A memo lookup succeeds if and only if there is a result in the memo table whose key is the same key as that of the current call and whose time interval is nested within the current time interval defined by the current-time and end-of-memo. This lookup rule is critical to correctness. Informally, it ensures that side-effects are incorporated into the current computation accurately. (In the formal semantics, it corresponds to the integration of the trace of the re-used computation into the current trace.)
undo \((t_s, t_e)\) =

for each \(t\). \(t_s < t < t_e\) do
if there is a version \(v = (v, t)\) then
    \(t' \leftarrow \) time of successor\((v)\)
    delete version \(v\) from its version-set
    \(R \leftarrow \{r \mid (t_1,t_2,f)\) is a reader \(\land t < t_1 < t'\}\)
    for each \(r \in R\) do
        delete \(r\) from its reader set
    if there is a reader \(r = (t, _, _)\) then
        delete \(r\) from its readers-set and from \(Q\)
    if there is a memo entry \(m = (t, _, _)\) then
        delete \(m\) from its table
    delete \(t\) from time-stamps

For each \(t\), \(t_s < t < t_e\) do
    memo () =
    let table \(\leftarrow\) new memo table
    fun mfun key f =
        case (find (table, key, now)) of
            NONE \(\rightarrow\)
                \(t_1 \leftarrow\) advance-time ()
                \(v \leftarrow\) f ()
                \(t_2 \leftarrow\) advance-time ()
                insert \((v, t_1, t_2)\) into table
                return \(v\)
            SOME \((v, t_1, t_2) \rightarrow\)
                undo \((current-time, t_1)\)
                propagate \((t_2)\)
                return \(v\)
        in mfun
    end
    propagate \((t)\) =
    while \(Q \neq \emptyset\) do
        \((t_s, t_e, f) \leftarrow\) checkMin \((Q)\)
        if \(t_s < t\) then
            deleteMin \((Q)\)
            current-time \(\leftarrow\) \(t_s\)
            tmp \(\leftarrow\) end-of-memo
            end-of-memo \(\leftarrow\) \(t_e\)
            f ()
            undo \((current-time, t_e)\)
            end-of-memo \(\leftarrow\) tmp
        else
            return
    return

Figure 14: Pseudo code for undo, memo, and propagate.

Undoing the computation between current-time and the start of the memoized computation serves some critical purposes: (1) it ensures that all versions read by the memoized computation are updated, and (2), it ensures that all computations that contain this computation are deleted and, thus, cannot be re-used.

The change propagation algorithm takes a queue of affected readers (set up by change operations) and processes them until the queue becomes empty. The queue is prioritized with respect to start time so that readers are processed in correct chronological order. To process a reader, we set current-time to the start time of the reader \(t_s\), remember the end-of-memo in a temporary variable, and run the body of the reader. After the body returns, we undo the computation between the current time and \(t_e\) and restore end-of-memo.

5.3 Relationship to the Semantics

A direct implementation of the semantics of SAIL (Section 3) is not efficient because change propagation relies on a complete traversal of the trace 1) to find the affected readers, and 2) to find the version of a modifiable at a given time during the computation and update all versions correctly. To find the versions and the affected readers quickly, the implementation maintains the version-set and the readers-set of each modifiable in a searchable time-ordered set data structure. By using these data structures and the undo
function, the implementation avoids a complete traversal of the trace during change propagation.

The semantics of SAIL does not specify how to find memoized computations for re-use. In our implementation, we remember the results and the time frames of memoized computations in a memo table and re-use them when possible. For a memoized computation to be re-usable, we require its time-frame to fall within the interval defined by \texttt{current-time} and \texttt{end-of-memo}. This ensures that when a memoized computation is re-used, the write operations performed by the computation are available in the current store. When we re-use a memoized computation, we delete the computations between the \texttt{current-time} and the beginning of the memoized computation. This guarantees that any computation is re-used at most once (by deleting all other computations that may contain it) and updates the versions of modifiables.

The semantics of SAIL uses term equality to determine whether a reader is affected or not. Since in ML we do not have access to such equality checks, we rely on user-provided equality tests. Since modifiables are equality types, the user can use ML’s “equals” operator for comparing them.

5.4 Asymptotic Complexity

We analyze the asymptotic complexity of self-adjusting computation primitives. For the analysis, we distinguish between an \textit{initial-run}, i.e., a from-scratch run of a self-adjusting program, and change propagation. Due to space constraints, we omit the proofs of these theorems and make them available separately Acar et al. [2007a].

\textbf{Theorem 5.1 (Overhead).} All self-adjusting computation primitives can be supported in expected constant time during the initial run, assuming that all memo functions have unique sets of keys. The expectation is taken over internal randomization used for representing memo tables.

\textit{Proof.} By using standard order maintenance data structures, all operations on time stamps can be performed in constant time. During the initial run, there are no priority queue operations. Memo operations can be performed in constant time by using standard hash tables. We now show that the operations on modifiables can be performed by increasing the overall running time by a constant factor by considering two representations for time-ordered sets.

- We use time-ordered lists for maintaining the version sets and reader sets. Since during initial run, we only access the last version and insert only at the end of the version set and reader set, we can perform all the operations in worst-case constant time.

- We use balanced binary search trees such as AVL trees for representing version sets and reader sets. We cannot afford to create these structures during execution, however, because we will need to pay logarithmic factor overhead. Instead, for each modifiable reference, we maintain all the \texttt{readers} and \texttt{versions} in a time-ordered list and delay the the construction of the version sets and reader sets. We separately maintain a list of all modifiable created. After the execution completes, we traverse the list of modifiable references and construct their version sets and reader set by performing a build step as follows. The construction for version sets and reader sets are symmetric. Consider version sets. We first traverse the set and copy each element into an array. We then build a balanced tree by placing the middle element of the array to the root and constructing the left and the right subtrees from the two halves of the array to the left and to the right of the middle respectively. This requires linear time in the size of the set. Since the total number of readers and version cannot be more than the running time. The process does not change the asymptotic bounds.

The operations on modifiable reference require more time during change propagation than during initial run. This is because modifiables can be read at arbitrary times (not just at the end). We show that, nevertheless all of these operations can be supported in logarithmic time in the number of readers and versions.
Consider an operation on a modifiable and let \( m \) be the maximum of the number of readers and versions (writes) that it has. During change propagation, \( \text{newMod} \) operation takes \( O(1) \) time, a \( \text{read} \) operation takes \( O(\log m) \) time, and \( \text{write} \) operation takes \( O(\log m + a \log m) \) time where \( a \) is the number of affected readers inserted into the queue.

**Proof.** With balanced-binary-tree representations of version lists and reader lists, note all operations on these data structure take \( O(\log n) \) time in their size. The \( \text{newMod} \) operations take constant time trivially. The read operations perform a \( \text{find} \) operation to locate the version to be read, create time-stamps, and perform an \( \text{insert} \) operation to insert the read into the reader list. All of these operations require \( O(\log m) \) time. A \( \text{write} \) operation, creates one time stamp, performs an \( \text{insert} \) operation to insert a new version into the version set, and performs one \( \text{find} \) operation for each affected reader inserted into the priority queue. These operations take \( O(\log m) \) time plus \( O(\log m) \) time per affected reader.

The operations on modifiable references require more time during change propagation than during initial run. This is because modifiables can be read at arbitrary times (not just at the end).

**Lemma 5.3 (Modifiable operations & change propagation).** During change propagation, \( \text{newMod}, \text{read}, \text{write} \) operations take \( O(1) \) time and \( \text{write} \) operation takes \( O(1 + a \log q) \) time where \( a \) is the number of affected readers inserted into the queue.

**Proof.** With sorted-list representations of version lists and reader lists, all operations take \( O(1) \). The \( \text{newMod} \) operations take constant time trivially. The read operations perform a \( \text{find} \) operation to locate the version to be read, create time-stamps, and perform an \( \text{insert} \) operation to insert the read into the reader list. All of these operations require \( O(1) \) time. A \( \text{write} \) operation, creates one time stamp, performs an \( \text{insert} \) operation to insert a new version into the version set, and performs one \( \text{find} \) operation for each affected reader inserted into the priority queue. These operations take \( O(1) \) time plus \( O(\log q) \) time per affected reader.

For the analysis of change propagation, we define several performance measures. Consider running the change-propagation algorithm, and let \( A \) denote the set of all affected readers, i.e., the readers that are inserted into the priority queue. Some of the affected readers are re-evaluated and the others are deleted; we refer to the set of re-evaluated readers as \( A_e \) and the set of deleted readers as \( A_d \). For a re-evaluated reader \( r \in A_e \), let \( |r| \) be its re-evaluation time complexity assuming that all self-adjusting primitives take constant time. Note that a re-evaluated \( r \) may re-use part of a previous computation via memoization and, therefore, take less time than a from-scratch re-execution. Let \( n_t \) denote the number of time stamps deleted during change propagation. Let \( n_q \) be the maximum size of the priority queue at any time during the algorithm. Let \( n_{rw} \) denote the maximum number of readers and versions (writes) that each modifiable may have.

**Theorem 5.4 (Change Propagation).** Change propagation takes

\[
O \left( |A| \log n_q + |A| \log n_{rw} + n_t \log n_{rw} + \sum_{r \in A_e} |r| \log n_{rw} \right)
\]

time.

**Proof.** The time for change propagation can be partitioned into four items: (1) priority queue operations, (2) re-evaluation of readers, (3) undo operations.

Since each priority queue operations takes \( O(\log n_q) \) and since each affected read is inserted into and deleted from the priority queue once, the time for priority queue operations is \( O(|A| \log n_q) \).

Since each modifiable operation requires constant time plus time to insert the affected readers into the priority queue (Lemma 5.2) and since each memo operation takes amortized constant time, re-evaluation of the readers require constant time plus time to insert the affected readers into the priority queue, which is \( O(|A| \log q) \). The cost of re-evaluation of the readers is thus \( O(|A| \log q + \sum_{r \in A_e} |r|) \).

23
Consider the undo operations. Each operation deletes a number of time stamps and for each time stamps performs a delete operation on a time-ordered set in $O(1)$ time and perhaps from the priority queue if a reader is being deleted, a memo table in $O(1)$ expected time. Since the deletions for the readers is being accounted separately, the total number of time for the undo operations is $O(n_t)$. Since each time stamp is created and deleted exactly once, the time for deletions can be amortized over a sequence of change propagations at constant time cost per time stamp.

Summing all of these terms gives the bound in the theorem.

For a special class of computations, where there is a constant bound on the number of times each modifiable is read and written, i.e., $n_{rw} = O(1)$, we have the following corollary.

**Corollary 5.5 (Change Propagation with Constant Reads & Writes).** In the presence of a constant bound on the number of reads and writes per modifiable, change propagation takes

\[ O\left(|A| \log n_q + \sum_{r \in A_c} |r|\right). \]

amortized time where the amortization is over a sequence of change propagations.

**Proof.** By plugging in $n_{rw} = O(1)$ into Theorem 5.4, we get $O\left(|A| \log n_q + n_t + \sum_{r \in A_c} |r|\right)$. Since each time-stamp is deleted at most once, and since the work performed during the deletion is constant when $n_{rw} = O(1)$, we can amortize the time for deletion to the creation of the time-stamp. Thus, change-propagation takes $O\left(|A| \log n_q + \sum_{r \in A_c} |r|\right)$ amortized time.

### 5.5 Complexity of Depth First Search

We prove the theorems from Section 2 for DFS and topological-sorting. Both theorems use the fact that the DFS algorithm shown in Figure 2 reads from and writes to each modifiable at most once, if the visitor function does the same. Since initializing a graph requires writing to each modifiable at most once, an application that constructs a graph and then performs a DFS with a single-read and single-write visitor reads from each modifiable once and writes to each modifiable at most twice.

**Theorem 5.6 (DFS).** Disregarding read operations performed by the visitor function and the reads of the values returned by the visitor function, the `depthFirstSearch` program responds to changes in time $O(m)$, where $m$ is the number of affected nodes after an insertion/deletion.

**Proof.** Let $G$ be a graph and $T$ be its DFS-tree. Let $G'$ be a graph obtained from $G$ by inserting an edge $(u,v)$ into $G$. The first read affected by this change will be the read of the edge $(u,v)$ performed when visiting $u$. There are a few cases to consider. If $v$ has been visited, then $v$ will not be visited again and change propagation will complete. If $v$ has not been visited, then it will be visited now and the algorithm will start exploring out from $v$ by traversing its out-edges. Since all of these out-edge traversals will be writing their results into newly allocated destinations, none of these visits will cause a memo match. Since each visited node now has a different path to the root of the DFS tree that passes through the new edge $(u,v)$, each node visited during this exploration process is affected. Since each visit takes constant time, this will require a total of $O(m)$ time. After the algorithm completes the exploration of the affected nodes, it will return to $v$ and then to $u$. From this point on, there will be no other executed reads and change propagation will complete. Since the only read that is ever inserted into the queue is the one that corresponds to the edge $(u,v)$, the queue size will not exceed one. By Theorem 5.4, the total time for change propagation is $O(m)$. The case for deletions is symmetric.

We show that the same bound holds for topological sort, which is an application of DFS. For computing the topological sort of a graph with DFS, we use a visitor function that takes as arguments the topological sorts of the subgraph originating at each neighbor of a node $u$, concatenates them and adds $u$ to the head of the resulting list and returns that list. These operations can be performed in constant time by writing to
the tails of the lists involved. Since a modifiable ceases to be at a tail position after a concatenation with a non-empty list, each modifiable in the output list is written at most once by the visitor function. Including the initialization, the total number of writes to each modifiable is bounded by two.

**Theorem 5.7 (Topological Sort).** Change propagation updates the topological sort of a graph in $O(m)$ time where $m$ is the number of affected nodes.

**Proof.** Consider change propagation after inserting an edge $(u,v)$. Since the visitor function takes constant time, the traversal of the affected nodes takes $O(m)$ time. After the traversal of the affected nodes completes, `depthFirstSearch` will return a result list that starts with the node $u$. Since this list is equal to the list that is returned in the previous execution based on the equality tests on modifiable lists (Section 2), it will cause no more reads to be re-executed, and change propagation completes.

6 Related Work

The problem of enabling computations to respond to changes automatically has been studied extensively. Most of the early work took place under the title of *incremental computation*. Here we review the previously proposed techniques that are based on dependence graphs and memoization and refer the reader to the bibliography of Ramalingam and Reps [1993] for other approaches such as those based on partial evaluation, e.g., Field and Teitelbaum [1990], Sundaresh and Hudak [1991].

Dependence-graph techniques record the dependences between data in a computation, so that a change-propagation algorithm can update the computation when the input is changed. Demers, Reps, and Teitelbaum (1981) and Reps [1982] introduced the idea of *static dependence graphs* and presented a change-propagation algorithm for them. The main limitation of static dependence graphs is that they do not permit the change-propagation algorithm to update the dependence structure. This significantly restricts the types of computations to which static-dependence graphs can be applied. For example, the INC language Yellin and Strom [1991], which uses static dependence graphs for incremental updates, does not permit recursion. To address this limitation, Acar, Blelloch, and Harper (2006c) proposed *dynamic dependence graphs* (or DDGs), presented language-techniques for constructing DDGs as programs execute, and showed that change-propagation can update the dependence structure as well as the output of the computation efficiently. The approach makes it possible to transform purely functional programs into self-adjusting programs that can respond to changes to its data automatically. Carlsson [2002] gave an implementation of the approach in the Haskell language. Further research on DDGs showed that, in some cases, they can support incremental updates as efficiently as special-purpose algorithms Acar et al. [2006c, 2004].

Another approach to incremental computation is based on memoization, where we remember function calls and reuse them when possible Bellman [1957], McCarthy [1963], Michie [1968]. Pugh [1988] and Pugh and Teitelbaum [1989] were the first to apply memoization (also called function caching) to incremental computation. One motivation behind their work was the lack of a general-purpose technique for incremental computation—static-dependence-graph techniques that existed then applied only to certain computations Pugh [1988]. Since Pugh and Teitelbaum’s work, other researchers investigated applications of various kinds of memoization to incremental computation Abadi et al. [1996], Liu et al. [1998], Heydon et al. [2000], Acar et al. [2003].

Until recently dependence-graph based techniques and memoization were treated as two independent approaches to incremental computation. Recent work Acar et al. [2006b] showed that there is, in fact, an interesting duality between DDGs and memoization in the way that they provide for result re-use and presented techniques for combining them. Other work Acar et al. [2007b] presented a semantics for the combination and proved that change propagation is consistent with respect to a standard purely functional semantics. The work on this paper builds on these findings. Initial experimental results based on the combination of DDGs and memoization show the combination to be effective in practice for a reasonably broad range of applications Acar et al. [2006b].

Self-adjusting computation based on DDGs and memoization has recently been applied to other problems. Shankar and Bodik [2007] gave an implementation of the approach in the Java language that targets...
invariant checking. They show that the approach is effective in speeding up run-time invariant checks significantly compared to non-incremental approaches. Other applications of self-adjusting computation include motion simulation Acar et al. [2006d], hardware-software codesign Santambrogio et al. [2007], and machine learning Acar et al. [2007c].

7 Conclusions

Self-adjusting computation has been shown to be effective for a reasonably broad range of applications where computation data changes slowly over time. Previously proposed techniques for self-adjusting computation, however, were applicable only in a purely functional setting. In this paper, we introduce an imperative programming model for self-adjusting computation by allowing modifiable references to be written multiple times.

We develop a set of primitives for imperative self-adjusting computation and provide implementation techniques for supporting these primitives. The key idea is to maintain different versions that modifiables take over time and keep track of dependences between versions and their readers. We prove that the approach can be implemented efficiently (essentially with the same efficiency as in the purely functional case) when the number of reads and writes of the modifiables is constant. In the general case, the implementation incurs a logarithmic-time overhead in the number of reads and writes per modifiable. As an example, we consider the depth-first search (DFS) problem on graphs and show that it can be expressed naturally. We show that change propagation requires time proportional to the number of nodes whose paths to the root of the DFS tree changes after insertion/deletion of an edge.

Since imperative self-adjusting programs can write to memory without any restrictions, they can create cyclic data structures making it difficult to prove consistency, i.e., that the proposed techniques respond to changes correctly. To prove consistency, we formulate a syntactic logical relation and show that any two evaluations of an expression e.g., a from-scratch evaluation or change propagation, are contextually equivalent. An interesting property of the logical relation is that it is untyped and is indexed only by the number of steps available for future evaluation. To handle the unobservable effects of non-deterministic memory allocation, our logical relations carry location bijections that pair corresponding locations in the two evaluations.

Remaining challenges include giving an improved implementation and a practical evaluation of the proposed approach, reducing the annotation requirements by simplifying the primitives or developing an automatic transformation from static/ordinary into self-adjusting programs that can track dependences selectively.

References


A Proof : Memo Elimination

This appendix provides details of the proof of memo elimination. For the logical predicate defined in Figure 12, we first prove the Fundamental Property (Lemma 4.3, proof given below), from which Memo Elimination (Lemma 4.4) follows as an immediate corollary.

Lemma A.1 (Value Predicate Downward Closed).
If $(k,v) \in \mathcal{V}^M$ and $j \leq k$, then $(j,v) \in \mathcal{V}^M$.

Lemma A.2 (Substitution Predicate Downward Closed).
If $(k,\gamma) \in \mathcal{G}^M[\Gamma]$ and $j \leq k$, then $(j,\gamma) \in \mathcal{G}^M[\Gamma]$.

Proof
By induction on $\Gamma$. Follows from Lemma A.1.

Lemma A.3 (Lemma 4.3 : Memo Elimination Fundamental Property).
If $\Gamma = FV(e)$, then $\Gamma \vdash e$.

Proof
By induction on the step index $k$ and nested induction on the structure of $e$.
All value cases except $\lambda x. e$ are immediate; we give details for Loc, Var, and Lam below.
We also provide proof details for Memo and Read; the latter is the only case where we need to make use of the outer induction hypothesis. All remaining cases are handled simply by applying the (inner) induction hypothesis.

Case (Loc) $l$:
Note that $\Gamma = FV(l) = \emptyset$.
Consider arbitrary $k$ and $\gamma$ such that
- $k \geq 0$ and
- $(k,\gamma) \in \mathcal{G}^M[\Gamma]$.

Note that $\text{dom}(\gamma) = \Gamma = \emptyset$.
We are required to show that $(k,\gamma(l)) \in \mathcal{C}^M \equiv (k,l) \in \mathcal{C}^M$.
Consider arbitrary $j$, $\sigma_0$, $\sigma'_0$, $\sigma$, $\sigma'$, $v$, $T$, $T'$, $j_1$ and $j_2$ such that
- $j < k$,
- $\sigma_0, l \Downarrow_{j_1} v, \sigma'_0, T$,
- $\sigma, T \leftarrow_{j_2} \sigma', T'$, and
- $j = j_1 + j_2$.
Hence, by inspection of the evaluation rules, it follows that
- $v = l$,
- $\sigma'_0 = \sigma_0$,
- $T = \varepsilon$, and
• \( j_1 = 0 \).

Hence, we have \( \sigma, \varepsilon \mathcal{C}^{j_2} \sigma', T' \).

Hence, by inspection of the change propagation rules, it follows that

• \( j_2 = 0 \),
• \( \sigma' = \sigma \), and
• \( T' = \varepsilon \).

Take \( j' = 0 \).

We are required to show that

• \( \sigma, l \mathcal{U}^{j'} \nu, \sigma', T' \),
  \( \equiv \sigma, l \mathcal{U}^{0} \nu, \sigma, \varepsilon \),
  which follows by inspection of the evaluation rules,
• \( j' \leq j \),
  \( \equiv 0 \leq j_1 + j_2 \),
  which follows from \( j_1 = j_2 = 0 \) (from above), and
• \( (k - j, v) \in \mathcal{V}^M \),
  \( \equiv (k - 0, l) \in \mathcal{V}^M \),
  \( \equiv (k, l) \in \mathcal{V}^M \),
  which follows from the definition of \( \mathcal{V}^M \).
Case (VAR) $x$:

Note that $\Gamma = \text{FV}(x) = \{x\}$.

Consider arbitrary $k$ and $\gamma$ such that
- $k \geq 0$ and
- $(k, \gamma) \in G^M[\Gamma]$.

Note that $\text{dom}(\gamma) = \Gamma = \{x\}$.

Note that there exists $v_x$ such that $\gamma(x) = v_x$ and $(k, v_x) \in \mathcal{V}$.

We are required to show that $(k, \gamma(x)) \in C^M \equiv (k, v_x) \in C^M$.

Consider arbitrary $j$, $\sigma_0$, $\sigma'$, $\sigma$, $v$, $T$, $T'$, $j_1$ and $j_2$ such that
- $j < k$,
- $\sigma_0, v_x \downarrow^{j_1} v, \sigma_0', T$,
- $\sigma, T \curvearrowright^{j_2} \sigma', T'$, and
- $j = j_1 + j_2$.

Hence, by inspection of the evaluation rules, since $v_x$ is a value, it follows that
- $v = v_x$,
- $\sigma_0' = \sigma_0$,
- $T = \varepsilon$, and
- $j_1 = 0$.

Hence, we have $\sigma, \varepsilon \curvearrowright^{j_2} \sigma', T'$.

Hence, by inspection of the change propagation rules, it follows that
- $j_2 = 0$,
- $\sigma' = \sigma$, and
- $T' = \varepsilon$.

Take $j' = 0$.

We are required to show that
- $\sigma, v_x \downarrow^{j'} v, \sigma', T'$,
  $\equiv \sigma, v_x \downarrow^{j} v_x, \sigma, \varepsilon$,
  which follows by inspection of the evaluation rules,
- $j' \leq j$,
  $\equiv 0 \leq j_1 + j_2$,
  which follows from $j_1 = j_2 = 0$ (from above), and
- $(k - j, v) \in \mathcal{V}^M$,
  $\equiv (k - 0, v_x) \in \mathcal{V}^M$,
  $\equiv (k, v_x) \in \mathcal{V}^M$,
  which follows from above.
Case (LAM) $\lambda x. e$:

Note that $\Gamma = FV(\lambda x. e)$.

Consider arbitrary $k$ and $\gamma$ such that

- $k \geq 0$
- $(k, \gamma) \in G^M[\Gamma]$.

We are required to show that $(k, \gamma(\lambda x. e)) \in C^M \equiv (k, \lambda x. \gamma(e)) \in C^M$.

Consider arbitrary $j$, $\sigma_0$, $\sigma'_0$, $\sigma$, $\sigma'$, $v$, $T$, $T'$, $j_1$ and $j_2$ such that

- $j < k$,
- $\sigma_0, \lambda x. \gamma(e) \triangledown v, \sigma'_0, T$,
- $\sigma, T \hookrightarrow \sigma', T'$, and
- $j = j_1 + j_2$.

Hence, by inspection of the evaluation rules, since $\lambda x. \gamma(e)$ is a value, it follows that

- $v = \lambda x.\gamma(e)$,
- $\sigma'_0 = \sigma_0$,
- $T = \varepsilon$, and
- $j_1 = 0$.

Hence, we have $\sigma, \varepsilon \hookrightarrow \sigma', T'$.

Hence, by inspection of the change propagation rules, it follows that

- $j_2 = 0$,
- $\sigma' = \sigma$, and
- $T' = \varepsilon$.

Take $j' = 0$.

We are required to show that

- $\sigma, \lambda x. \gamma(e) \triangledown v, \sigma', T'$,
- $\equiv \sigma, \lambda x. \gamma(e) \triangledown \varepsilon \lambda x. \gamma(e), \sigma, \varepsilon$,
- which follows by inspection of the evaluation rules,
- $j' \leq j$,
- $\equiv 0 \leq j_1 + j_2$,
- which follows from $j_1 = j_2 = 0$ (from above), and
- $(k - j, v) \in V^M$,
- $\equiv (k - 0, \lambda x. \gamma(e)) \in V^M$,
- $\equiv (k, \lambda x. \gamma(e)) \in V^M$,
- which we conclude as follows:

Consider arbitrary $j$ and $v_a$ such that

- $j < k$ and
- $(j, v_a) \in V^M$.

We are required to show that $(j, \gamma(e)[v_a/x]) \in C^M$.

Consider arbitrary $i$, $\sigma_{01}$, $\sigma'_{01}$, $\sigma_1$, $\sigma'_1$, $v_1$, $T_1$, $T'_1$, $i_1$ and $i_2$ such that

- $i < j$,
- $\sigma_{01}, \gamma(e)[v_a/x] \triangledown v_1, \sigma'_{01}, T_1$,
• \( \sigma_1, T_1 \cap^{i_2} \sigma_1', T_1' \), and 
• \( i = i_1 + i_2 \).

Applying the induction hypothesis to \( e \) (noting \( FV(e) = \Gamma \cup \{x\} \)), we conclude that \( \Gamma, x \vdash e \).

Instantiate this with \( j \) and \( \gamma[x \mapsto v_a] \). Note that

• \( j \geq 0 \) and 
• \( (j, \gamma[x \mapsto v_a]) \in \mathcal{G}^{\mathcal{M}[\Gamma, x]} \), 
which follows from
  • \( (j, \gamma) \in \mathcal{G}^{\mathcal{M}[\Gamma]} \), 
    which follows from Lemma A.2 applied to \( (k, \gamma) \in \mathcal{G}^{\mathcal{M}[\Gamma]} \) and \( j < k \), and 
  • \( (j, v_a) \in \mathcal{V}^{\mathcal{M}} \), 
    which follows from above.

Hence, \( (j, \gamma[x \mapsto v_a](e)) \in \mathcal{C}^{\mathcal{M}} \).

Thus \( (j, \gamma(e)[v_a/x]) \in \mathcal{C}^{\mathcal{M}} \) as we needed to show.
Case (Memo) memo e:

Note that $\Gamma = FV(\text{memo } e)$.

Consider arbitrary $k$ and $\gamma$ such that

- $k \geq 0$
- $(k, \gamma) \in \mathcal{G}_{\text{M}}[\Gamma]$

We are required to show that $(k, \gamma(\text{memo } e)) \in \mathcal{C}_{\text{M}} \equiv (k, \text{memo } \gamma(e)) \in \mathcal{C}_{\text{M}}$.

Consider arbitrary $j, \sigma_0, \sigma'_0, \sigma, \sigma', v, T, T', j_1$ and $j_2$ such that

- $j < k$,
- $\sigma_0, \text{memo } \gamma(e) \Downarrow^{ji} v, \sigma'_0, T$,
- $\sigma, T \searrow^{j_2} \sigma', T'$, and
- $j = j_1 + j_2$.

Hence, by inspection of the evaluation rules (memo rule), it follows that

- $\sigma_0, \gamma(e) \Downarrow^{j_1} v, \sigma'_0, T_0$,
- $\sigma_0, T_0 \searrow^{j_2} \sigma'_0, T$, and
- $j_1 = j_{11} + j_{12}$.

Applying the induction hypothesis to e, (noting $FV(e) = \Gamma$), we conclude that $\Gamma \vdash e$.

Instantiate this with $k$ and $\gamma$. Note that

- $k > 0$
- $(k, \gamma) \in \mathcal{G}_{\text{M}}[\Gamma]$

Hence, $(k, \gamma(e)) \in \mathcal{C}_{\text{M}}$.

Instantiate the latter with $j_1, \sigma_{00}, \sigma'_0, \sigma_0, \sigma'_0, v, T_0, T, j_{11}$, and $j_{12}$. Note that

- $j_1 < k$,
  which follows from $j_1 \leq j < k$ (from above),
- $\sigma_{00}, \gamma(e) \Downarrow^{j_{11}} v, \sigma'_{00}, T_0$,
  which follows from above,
- $\sigma_0, T_0 \searrow^{j_{12}} \sigma'_0, T$,
  which follows from above, and
- $j_1 = j_{11} + j_{12}$,
  which follows from above.

Hence, there exists $j'_1$ such that

- $\sigma_0, \gamma(e) \Downarrow^{j'_1} v, \sigma'_0, T$,
- $j'_1 \leq j_1$, and
- $(k - j_1, v) \in \mathcal{V}_{\text{M}}$.

Take $j' = j'_1 + j_2$.

We are required to show that

- $\sigma, \text{memo } \gamma(e) \Downarrow^{j'} v, \sigma', T'$,
  which follows by inspection of the evaluation rules (by memo rule) since
  - $\sigma_0, \gamma(e) \Downarrow^{j'_1} v, \sigma'_0, T$ (from above),
  - $\sigma, T \searrow^{j_2} \sigma', T'$ (from above), and
• $j' = j'_1 + j_2$ (from above).

• $j' \leq j$,
  $\equiv j'_1 + j_2 \leq j_1 + j_2$,
  which follows from $j'_1 \leq j_1$ (from above), and

• $(k - j, v) \in \mathcal{V}^M$,
  $\equiv (k - j_1 - j_2, v) \in \mathcal{V}^M$, (since $j = j_1 + j_2$)
  which follows from Lemma A.1 applied to $(k - j_1, v) \in \mathcal{V}^M$ (from above) and
  $k - j \leq k - j_1$ (which follows from $k - j = k - j_1 - j_2 \leq k - j_1$).
Case (READ) read $v$ as $x$ in $e$:

Note that $\Gamma = FV(\text{read } v \text{ as } x \text{ in } e) \equiv FV(v) (FV(e) \backslash \{x\})$.

Let $\Gamma_1 = FV(v)$ and $\Gamma_2 = (FV(e) \backslash \{x\})$. Hence, $\Gamma = \Gamma_1 \cup \Gamma_2$.

Consider arbitrary $k$ and $\gamma$ such that

- $k \geq 0$ and
- $(k, \gamma) \in G^M[\Gamma]$.

Note that we have

- $\gamma = \gamma_1 \cup \gamma_2$,
- $(k, \gamma_1) \in G^M[\Gamma_1]$, and
- $(k, \gamma_2) \in G^M[\Gamma_2]$.

We are required to show that $(k, \gamma(\text{read } v \text{ as } x \text{ in } e)) \in C^M$

\[ \equiv (k, \text{read } \gamma(v) \text{ as } x \text{ in } \gamma(e)) \in C^M \]

\[ \equiv (k, \text{read } \gamma_1(v) \text{ as } x \text{ in } \gamma_2(e)) \in C^M \].

Consider arbitrary $j$, $\sigma_0, \sigma_0', \sigma, \sigma', v, T, T', j_1$ and $j_2$ such that

- $j < k$,
- $\sigma_0, \text{read } \gamma_1(v) \text{ as } x \text{ in } \gamma_2(e) \not\vdash v, \sigma_0', T$,
- $\sigma, T \not\vdash \text{read } \gamma_2(v) \not\vdash \text{read } \gamma_2(v) = T', \text{ and}$
- $j = j_1 + j_2$.

Hence, by inspection of the evaluation rules (read rule), it follows that

- $\gamma_1(v) = l$,
- $v = ()$,
- $T = \text{read}_{l \rightarrow x = \sigma_0(l), \gamma_2(e)} T_1$, and
- $\sigma_0, \gamma_2(e) | \sigma_0(l) / x, \gamma_j = \gamma_1(v), \sigma_0', T_1$.

Note that we have $\sigma, T \not\vdash \text{read } \gamma_2(v) \not\vdash \text{read } \gamma_2(v) = T'$.

Thus, there are two change propagation cases to consider: read/no-change and read/change.

Case (read/no ch.) :

From $\sigma, \text{read}_{l \rightarrow x = \sigma_0(l), \gamma_2(e)} T_1 \vdash \text{read } \gamma_2(v) \not\vdash \sigma', T'$, by inspection of the change propagation rules (read/no ch. rule), it follows that

- $\sigma(l) = \sigma_0(l)$,
- $\sigma, T_1 \vdash \text{read } \gamma_2(v) \not\vdash \sigma', T_1'$, and
- $T' = \text{read}_{l \rightarrow x = \sigma_0(l), \gamma_2(e)} T_1'$.

Applying the induction hypothesis to $e$, (noting $FV(e) = \Gamma_2 \cup \{x\}$), we conclude that $\Gamma_2, x \vdash e$.

Instantiate this with $k - 1$ and $\gamma_2[x \mapsto \sigma(l)]$. Note that

- $k - 1 \geq 0$,
  which follows from $j_1 + j_2 = j < k$ and $j_1 - 1 \geq 0$, and
- $(k - 1, \gamma_2[x \mapsto \sigma(l)]) \in G^M[\Gamma_2, x]$, which follows from
  - $(k - 1, \gamma_2) \in G^M[\Gamma_2]$, which follows from Lemma A.2 applied to $(k, \gamma_2) \in G^M[\Gamma_2]$ (from above) and $k - 1 < k$, and
  - $\sigma(l) \in \mathcal{V}^M$, which follows from the outer induction hypothesis applied to $k - 1$, noting $k - 1 < k$. 36
Hence, we have $(k - 1, \gamma_2[ x \mapsto \sigma(l)](c)) \in \mathcal{C}^M \equiv (k - 1, \gamma_2(e)[\sigma(l)/x]) \in \mathcal{C}^M$.

Instantiate the latter with $j_1 + j_2 - 1 < k$, $\sigma_0, \sigma, \sigma', (\cdot), T_1, T_1'$, $j_1, j_1 - 1$, and $j_2$. Note that

- $j_1 + j_2 - 1 < k$,
  which follows from $j_1 + j_2 = j < k$ (from above),
- $\sigma_0, \gamma_2(e)[\sigma(l)/x] \upharpoonright^{j_1-1} (\cdot), \sigma_0', T_1$,
  which follows from $\sigma_0, \gamma_2(e)[\sigma_0(l)/x] \upharpoonright^{j_1-1} (\cdot), \sigma_0', T_1$ (from above) since $\sigma(l) = \sigma_0(l)$,
- $\sigma, T_1 \bowtie^{j_1} \sigma', T_1'$,
  which follows from above, and
- $j_1 + j_2 - 1 = (j_1 - 1) + j_2$,
  which is immediate.

Hence, there exists $j_1'$ such that

- $\sigma, \gamma_2(e)[\sigma(l)/x] \upharpoonright^{j_1'} (\cdot), \sigma', T_1'$,
- $j_1' \leq j_1 + j_2 - 1$, and
- $(k - (j_1 + j_2 - 1), (\cdot)) \in \mathcal{V}^M$.

Take $j' = j_1' + 1$.

We are required to show that

- $\sigma, \text{read } \gamma_1(v) \text{ as } x \text{ in } \gamma_2(e) \upharpoonright^{j'} v, \sigma', T'$,
  $\equiv \sigma, \text{read } l \text{ as } x \text{ in } \gamma_2(e) \upharpoonright^{j_1+1} (\cdot), \sigma', \text{read } l \rightarrow x = \sigma_0(l), \gamma_2(e) \text{ } T_1'$,
  $\equiv \sigma, \text{read } l \text{ as } x \text{ in } \gamma_2(e) \upharpoonright^{j_1+1} (\cdot), \sigma', \text{read } l \rightarrow x = \sigma(l), \gamma_2(e) \text{ } T_1'$ (since $\sigma(l) = \sigma_0(l)$),
  which follows by inspection of the evaluation rules (read rule) since
  - $\sigma, \gamma_2(e)[\sigma(l)/x] \upharpoonright^{j_1'} (\cdot), \sigma', T_1'$,
    which follows from above.

- $j' \leq j$,
  $\equiv j_1' + 1 \leq j_1 + j_2$,
  which follows from $j_1' \leq j_1 + j_2 - 1$ (from above), and
- $(k - j, v) \in \mathcal{V}^M$,
  $\equiv (k - j, (\cdot)) \in \mathcal{V}^M$,
  which follows from the definition of $\mathcal{V}^M$.

**Case (read/ch.)**:

From $\sigma, \text{read } l \rightarrow x = \sigma_0(l), \gamma_2(e) \text{ } T_1 \bowtie^{j_1} \sigma', T'$, by inspection of the change propagation rules (read/ch. rule), it follows that

- $\sigma(l) \neq \sigma_0(l)$,
- $\sigma, \gamma_2(e)[\sigma(l)/x] \upharpoonright^{j_2-1} (\cdot), \sigma', T_1'$, and
- $T' = \text{read } l \rightarrow x = \sigma(l), \gamma_2(e) \text{ } T_1'$.

Take $j' = j_2$.

We are required to show that

- $\sigma, \text{read } \gamma_1(v) \text{ as } x \text{ in } \gamma_2(e) \upharpoonright^{j'} v, \sigma', T'$,
  $\equiv \sigma, \text{read } l \text{ as } x \text{ in } \gamma_2(e) \upharpoonright^{j_2} (\cdot), \sigma', \text{read } l \rightarrow x = \sigma(l), \gamma_2(e) \text{ } T_1'$,
  which follows by inspection of the evaluation rules (read rule) since
  - $\sigma, \gamma_2(e)[\sigma(l)/x] \upharpoonright^{j_2-1} (\cdot), \sigma', T_1'$.

- $j' \leq j$,
  $\equiv j_2 \leq j_1 + j_2$,
  which is immediate, and
- $(k - j, v) \in \mathcal{V}^M$,
  $\equiv (k - j, (\cdot)) \in \mathcal{V}^M$,
  which follows from the definition of $\mathcal{V}^M$.

\[ \square \]
B  Proof : Fundamental Property of Logical Relation

This appendix gives details of the proof of the Fundamental Property (Theorem 4.5) for the logical relation defined in Figure 10. In the next section (Appendix C), we show how consistency (Lemma 4.6) follows from the Fundamental Property.

- Section B.1 presents a series of lemmas required for the proof of the Fundamental Property.
- Section B.2 contains the proof of the Fundamental Property (Theorem 4.5).

B.1  Supporting Lemmas for Fundamental Property

Lemma B.1 (Value Relation Downward Closed).

If \((k, \psi, v_1, v_2) \in V\) and \(j \leq k\),
then \((j, \psi, v_1, v_2) \in V\).

Proof

By induction on \(\Gamma\).

Case \(\Gamma = \emptyset\): Trivial.

Case \(\Gamma = \Gamma_1 \uplus \{x\}\): Follows by applying the induction hypothesis to \(\Gamma_1\) and by Lemma B.1.

Lemma B.2 (Substitution Relation Downward Closed).

If \((k, \psi, \gamma_1, \gamma_2) \in G[\Gamma]\) and \(j \leq k\),
then \((j, \psi, \gamma_1, \gamma_2) \in G[\Gamma]\).

Proof

Follows by Lemma B.1.

Lemma B.3 (Store Relation Downward Closed).

If \(\sigma_1, \sigma_2 : k \psi \rightarrow S\) and \(j \leq k\),
then \(\sigma_1, \sigma_2 : j \psi \rightarrow S\).

Proof

Follows by Lemma B.1.
Lemma B.4 (Stores Related at Root Subset).

If $\sigma_1, \sigma_2 : k (\psi_1 \odot \psi_2) \rightsquigarrow S$, then there exists $S_1$ such that $\sigma_1, \sigma_2 : k \psi_1 \rightsquigarrow S_1$ and $S_1 \subseteq S$.

Proof

We have

$\sigma_1, \sigma_2 : k (\psi_1 \odot \psi_2) \rightsquigarrow S$

$\equiv S \in \text{LocBij} \land$

$\exists F : S \rightarrow \text{LocBij}$.

$S = \psi_1 \odot \psi_2 \odot \bigoplus_{(l_1, l_2) \in S} F(l_1, l_2) \land$

$\text{dom}(\sigma_1) \supseteq S^1 \land \text{dom}(\sigma_2) \supseteq S^2 \land$

$\forall (l_1, l_2) \in S. \forall j < k.$

$(j, F(l_1, l_2), \sigma_1(l_1), \sigma_2(l_2)) \in V$.

Pick $S_1$ such that $S_1 \subseteq S$ and such that $\psi_1 \subseteq S_1$ and $(\forall (l_1, l_2) \in S_1. F(l_1, l_2) \subseteq S_1)$.

Note that $S_1 \in \text{LocBij}$.

Take $F_1 = F|_{S_1}$.

We are required to show that

- $F_1 : S_1 \rightarrow \text{LocBij}$, which follows from $F_1 = F|_{S_1}$ and $F : S \rightarrow \text{LocBij}$,

- $S_1 = \psi_1 \odot \bigoplus_{(l_1, l_2) \in S_1} F_1(l_1, l_2)$

$\equiv S_1 = \psi_1 \odot \bigoplus_{(l_1, l_2) \in S_1} F_1(l_1, l_2)$ (since $F_1 = F|_{S_1}$) which follows by the definition of $S_1$.

- $\text{dom}(\sigma_1) \supseteq S_1^1$, which follows from $\text{dom}(\sigma_1) \supseteq S^1$ and $S^1 \subseteq S_1^1$,

- $\text{dom}(\sigma_2) \supseteq S_1^2$, which follows from $\text{dom}(\sigma_2) \supseteq S^2$ and $S^2 \subseteq S_1^2$, and

- $\forall (l_1, l_2) \in S_1. \forall j < k.$ $(j, F_1(l_1, l_2), \sigma_1(l_1), \sigma_2(l_2)) \in V$.

which we conclude as follows:

Consider arbitrary $l_1, l_2$, and $j$ such that $(l_1, l_2) \in S_1$ and $j < k$.

Hence, $(l_1, l_2) \in S$, since $S_1 \subseteq S$.

Hence, from the definition of $\sigma_1, \sigma_2 : k (\psi_1 \odot \psi_2) \rightsquigarrow S$, we have

$(j, F_1(l_1, l_2), \sigma_1(l_1), \sigma_2(l_2)) \in V$

$\equiv (j, F_1(l_1, l_2), \sigma_1(l_1), \sigma_2(l_2)) \in V$.

which follows from $(l_1, l_2) \in S_1$ and $F_1 = F|_{S_1}$.
B.2 Fundamental Property

Theorem B.5 (Thm 4.5 : Fundamental Property).
If \( \Gamma = \text{FV}(e) \) and \( \text{FL}(e) = \emptyset \), then \( \Gamma \vdash e \approx e \).

Proof

By induction on the structure of \( e \).

Note that it suffices to prove \( \Gamma \vdash e \triangleleft e \).

Case (VAR) \( x \): 
Note that \( \Gamma = \text{FV}(x) = \{x\} \).
Consider arbitrary \( k, \psi_\Gamma, \gamma_1, \) and \( \gamma_2 \) such that

- \( k \geq 0 \), and
- \( (k, \psi_\Gamma, \gamma_1, \gamma_2) \in \mathcal{G}[\Gamma] \).

Note that \( \text{dom}(\gamma_1) = \text{dom}(\gamma_2) = \Gamma = \{x\} \).
Note that there exist \( v_1, v_2 \) such that \( \gamma_1(x) = v_1 \) and \( \gamma_2(x) = v_2 \) and \( (k, \psi_\Gamma, v_1, v_2) \in \mathcal{V} \).
We are required to show that \( (k, \psi_\Gamma, \gamma_1(x), \gamma_2(x)) \in \mathcal{C} \equiv (k, \psi_\Gamma, v_1, v_2) \in \mathcal{C} \).
Consider arbitrary \( j, \sigma_1, \sigma_2, \psi_r, S, v_{f1}, \sigma_{f1} \), and \( T_{f1} \) such that

- \( j < k \),
- \( \sigma_1, \sigma_2 \vdash_k (\psi_\Gamma \oplus \psi_r) \leadsto S \),
- \( \sigma_1, v_1 \Downarrow^j v_{f1}, \sigma_{f1}, T_{f1} \), and
- \( S^1 \cap \text{alloc}(T_{f1}) = \emptyset \).

Hence, by inspection of the evaluation rules, it follows that

- \( v_{f1} = v_1 \),
- \( j = 0 \),
- \( \sigma_{f1} = \sigma_1 \), and
- \( T_{f1} = \varepsilon \).

Consider arbitrary \( v_{f2}, \sigma_{f2} \), and \( T_{f2} \) such that

- \( \sigma_2, v_2 \Downarrow v_{f2}, \sigma_{f2}, T_{f2} \), and
- \( S^2 \cap \text{alloc}(T_{f2}) = \emptyset \).

Hence, by inspection of the evaluation rules, it follows that

- \( v_{f2} = v_2 \),
- \( \sigma_{f2} = \sigma_2 \), and
- \( T_{f2} = \varepsilon \).
Take $\psi_f = \psi_T$. Take $S_f = S$.

We are required to show:

- $(k - j, \psi_f, v_{f1}, v_{f2}) \in V$
  \[ \equiv (k - 0, \psi_T, v_1, v_2) \in V, \]
  which follows from above,

- $\sigma_{11}, \sigma_{22} : k - j (\psi_f \otimes \psi_r) \rightsquigarrow S_f$
  \[ \equiv \sigma_1, \sigma_2 : k_0 (\psi_T \otimes \psi_r) \rightsquigarrow S, \]
  which follows from above,

- $S_1^f \subseteq S_1 \cup \text{alloc}(T_{f1})$
  \[ \equiv S_1 \subseteq S_1 \cup \text{alloc}(\epsilon), \]
  which is immediate, and

- $S_2^f \subseteq S_2 \cup \text{alloc}(T_{f2})$
  \[ \equiv S_2 \subseteq S_2 \cup \text{alloc}(\epsilon), \]
  which is immediate.
Case (Int) \( n \):

Note that \( \Gamma = FV(n) = \emptyset \).

Consider arbitrary \( k, \psi_T, \gamma_1 \), and \( \gamma_2 \) such that

- \( k \geq 0 \), and
- \((k, \psi_T, \gamma_1, \gamma_2) \in G[\Gamma]\).

Note that \( \text{dom}(\gamma_1) = \text{dom}(\gamma_2) = \Gamma = \emptyset \) and \( \psi_T = \{\} \)

We are required to show that \((k, \psi_T, \gamma_1(n), \gamma_2(n)) \in C \equiv (k, \{\}, n, n) \in C\).

Consider arbitrary \( j, \sigma_1, \sigma_2, \psi_r, S, v_{f_1}, \sigma_{f_1}, \) and \( T_{f_1} \) such that

- \( j < k \),
- \( \sigma_1, \sigma_2 : k (\{\} \odot \psi_r) \rightsquigarrow S \),
- \( \sigma_1, n \downarrow^j v_{f_1}, \sigma_{f_1}, T_{f_1} \), and
- \( S_1 \cap \text{alloc}(T_{f_1}) = \emptyset \).

Hence, by inspection of the evaluation rules, it follows that

- \( v_{f_1} = n \),
- \( j = 0 \),
- \( \sigma_{f_1} = \sigma_1 \), and
- \( T_{f_1} = \varepsilon \).

Consider arbitrary \( v_{f_2}, \sigma_{f_2}, \) and \( T_{f_2} \) such that

- \( \sigma_2, n \downarrow v_{f_2}, \sigma_{f_2}, T_{f_2} \), and
- \( S_2 \cap \text{alloc}(T_{f_2}) = \emptyset \).

Hence, by inspection of the evaluation rules, it follows that \( v_{f_2} = n \) and \( \sigma_{f_2} = \sigma_2 \) and \( T_{f_2} = \varepsilon \).

Take \( \psi_f = \{\} \). Take \( S_f = S \).

We are required to show:

- \((k - j, \psi_f, v_{f_1}, v_{f_2}) \in V \equiv (k - 0, \{\}, n, n) \in V \),
  which follows from the definition of \( V \),
- \( \sigma_{f_1}, \sigma_{f_2} : k - j (\{\} \odot \psi_r) \rightsquigarrow S_f \)
  \( \equiv \sigma_1, \sigma_2 : k - 0 (\{\} \odot \psi_r) \rightsquigarrow S \),
  which follows from above,
- \( S_f \subseteq S^1 \cup \text{alloc}(T_{f_1}) \)
  \( \equiv S^1 \subseteq S^1 \cup \text{alloc}(\varepsilon) \),
  which is immediate, and
- \( S_f \subseteq S^2 \cup \text{alloc}(T_{f_2}) \)
  \( \equiv S^2 \subseteq S^2 \cup \text{alloc}(\varepsilon) \),
  which is immediate.
Case (UNIT) () :

Note that $\Gamma = FV((() = \emptyset$.
Consider arbitrary $k, \psi_T, \gamma_1$, and $\gamma_2$ such that

• $k \geq 0$, and
• $(k, \psi_T, \gamma_1, \gamma_2) \in G[\Gamma]$.

Note that $\text{dom}(\gamma_1) = \text{dom}(\gamma_2) = \Gamma = \emptyset$ and $\psi_T = \{\}$
We are required to show that $(k, \psi_T, \gamma_1((), \gamma_2((()))) \in C \equiv (k, \{\}, (),()) \in C$.
Consider arbitrary $j, \sigma_1, \sigma_2, \psi_r, S, v_{f_1}, \sigma_{f_1}$, and $T_{f_1}$ such that

• $j < k$,
• $\sigma_1, \sigma_2 : k (\{\} \odot \psi_r) \rightsquigarrow S$,
• $\sigma_1, () \downarrow^j v_{f_1}, \sigma_{f_1}, T_{f_1}$, and
• $S^1 \cap \text{alloc}(T_{f_1}) = \emptyset$.

Hence, by inspection of the evaluation rules, it follows that

• $v_{f_1} = ()$,
• $j = 0$,
• $\sigma_{f_1} = \sigma_1$, and
• $T_{f_1} = \varepsilon$.

Consider arbitrary $v_{f_2}, \sigma_{f_2}$, and $T_{f_2}$ such that

• $\sigma_2, () \downarrow v_{f_2}, \sigma_{f_2}, T_{f_2}$, and
• $S^2 \cap \text{alloc}(T_{f_2}) = \emptyset$.

Hence, by inspection of the evaluation rules, it follows that $v_{f_2} = ()$ and $\sigma_{f_2} = \sigma_2$ and $T_{f_2} = \varepsilon$.

Take $\psi_f = \{\}$. Take $S_f = S$.
We are required to show:

• $(k - j, \psi_f, v_{f_1}, v_{f_2}) \in V$
  $\equiv (k - 0, \{\}, (),()) \in V$,
  which follows from the definition of $V$,
• $\sigma_{f_1}, \sigma_{f_2} : (\{\} \odot \psi_r) \rightsquigarrow S_f$
  $\equiv \sigma_1, \sigma_2 : (\{\} \odot \psi_r) \rightsquigarrow S$,
  which follows from above,
• $S_{f_1} \subseteq S^1 \cup \text{alloc}(T_{f_1})$
  $\equiv S^1 \subseteq S^1 \cup \text{alloc}(\varepsilon)$,
  which is immediate, and
• $S_{f_2} \subseteq S^2 \cup \text{alloc}(T_{f_2})$
  $\equiv S^2 \subseteq S^2 \cup \text{alloc}(\varepsilon)$,
  which is immediate.
Case (LAM) $\lambda x. e$

Note that $\Gamma = FV(\lambda x. e)$.

Consider arbitrary $k, \psi_\Gamma, \gamma_1,$ and $\gamma_2$ such that

- $k \geq 0$,
- $(k, \psi_\Gamma, \gamma_1, \gamma_2) \in \mathcal{G}[\Gamma]$. 

We are required to show that $(k, \psi_\Gamma, \gamma_1(\lambda x. e), \gamma_2(\lambda x. e)) \in C \equiv (k, \psi_\Gamma, \lambda x. \gamma_1(e), \lambda x. \gamma_2(e)) \in C$.

Consider arbitrary $j, \sigma_1, \sigma_2, \psi_r, S, v_{f1}, \sigma_{f1}$, and $T_{f1}$ such that

- $j < k$,
- $\sigma_1, \sigma_2 : k (\psi_\Gamma \odot \psi_r) \leadsto S$,
- $\sigma_1, \lambda x. \gamma_1(e) \nvdash^j v_{f1}, \sigma_{f1}, T_{f1}$, and
- $S^1 \cap \text{alloc}(T_{f1}) = \emptyset$.

Hence, by inspection of the evaluation rules, since $\lambda x. \gamma_1(e)$ is a value, it follows that

- $v_{f1} = \lambda x. \gamma_1(e)$,
- $j = 0$,
- $\sigma_{f1} = \sigma_1$, and
- $T_{f1} = \varepsilon$.

Consider arbitrary $v_{f2}, \sigma_{f2}$, and $T_{f2}$ such that

- $\sigma_2, \lambda x. \gamma_2(e) \nvdash v_{f2}, \sigma_{f2}, T_{f2}$, and
- $S^2 \cap \text{alloc}(T_{f2}) = \emptyset$.

Hence, by inspection of the evaluation rules, since $\lambda x. \gamma_2(e)$ is a value, it follows that

- $v_{f2} = \lambda x. \gamma_2(e)$,
- $\sigma_{f2} = \sigma_2$, and
- $T_{f2} = \varepsilon$.

Take $\psi_f = \psi_\Gamma$. Take $S_f = S$.

We are required to show:

- $(k - j, \psi_f, v_{f1}, v_{f2}) \in V \equiv (k - 0, \psi_\Gamma, \lambda x. \gamma_2(e), \lambda x. \gamma_2(e)) \in V \equiv (k, \psi_\Gamma, \lambda x. \gamma_2(e), \lambda x. \gamma_2(e)) \in V$, which we conclude as follows:

Consider arbitrary $i, \psi_a, v_1, v_2$ such that

- $i < k$,
- $(i, \psi_a, v_1, v_2) \in V$, and
- $(\psi_\Gamma \odot \psi_a)$ defined.

We are required to show that $(i, \psi_\Gamma \odot \psi_a, \gamma_1(e_1)[v_1/x], \gamma_2(e_2)[v_2/x]) \in C$.

Applying the induction hypothesis to $e$ (noting $FV(e) = \Gamma \cup \{x\}$), we conclude that $\Gamma, x \vdash e \iff e$.

Instantiate this with $i, (\psi_\Gamma \odot \psi_a), \gamma_1[x \mapsto v_1], \text{ and } \gamma_2[x \mapsto v_2]$. Note that

- $i \geq 0$, 

44
\( \bullet (i, \psi_T \odot \psi_a, \gamma_1[x \mapsto v_1], \gamma_2[x \mapsto v_2]) \in G[\Gamma, x], \)
which follows from

\( \bullet (i, \psi_T, \gamma_1, \gamma_2) \in G[\Gamma], \)
which follows from Lemma B.2 applied to \((k, \psi_T, \gamma_1, \gamma_2) \in G[\Gamma]\) and \(i < k\), and

\( \bullet (i, \psi_a, v_1, v_2) \in V, \)
which follows from above.

Hence, \((i, \psi_T \odot \psi_a, \gamma_1[x \mapsto v_1](e_1), \gamma_2[x \mapsto v_2](e_2)) \in C.\)
Thus, \((i, \psi_T \odot \psi_a, \gamma_1(e_1)[x/v_1], \gamma_2(e_2)[v_2/x]) \in C\) as we needed to show.

\( \bullet \sigma_{f_1}, \sigma_{f_2} : k \rightarrow j (\psi_f \odot \psi_r) \rightsquigarrow S_f \)
\( \equiv \sigma_1, \sigma_2 : k \rightarrow 0 (\psi_T \odot \psi_r) \rightsquigarrow S, \)
which follows from above,

\( \bullet S_f^1 \subseteq S^1 \cup \text{alloc}(T_{f_1}) \)
\( \equiv S^1 \subseteq S^1 \cup \text{alloc}(T_{f_1}), \)
which is immediate, and

\( \bullet S_f^2 \subseteq S^2 \cup \text{alloc}(T_{f_2}) \)
\( \equiv S^2 \subseteq S^2 \cup \text{alloc}(T_{f_2}), \)
which is immediate.
Case (Apply) \( v_1 \rightarrow v_2 \):

Note that \( \Gamma = FV(v_1) \cup FV(v_2) \).
Let \( \Gamma_{01} = FV(v_1) \) and \( \Gamma_{02} = FV(v_2) \). Hence, \( \Gamma = \Gamma_{01} \cup \Gamma_{02} \).
Consider arbitrary \( k, \psi, \gamma_1, \) and \( \gamma_2 \) such that

- \( k \geq 0 \), and
- \( (k, \psi, \gamma_1, \gamma_2) \in \mathcal{G}[\Gamma] \).

Note that we have \( \gamma_1, \gamma_2, \gamma_{21}, \gamma_{22}, \) and \( \psi_{T01}, \psi_{T02} \) such that

- \( \gamma_1 = \gamma_{11} \cup \gamma_{12} \),
- \( \gamma_2 = \gamma_{21} \cup \gamma_{22} \),
- \( \psi = \psi_{T01} \cup \psi_{T02} \),
- \( (k, \psi_{T01}, \gamma_1, \gamma_2, \gamma_{21}, \gamma_{22}) \in \mathcal{G}[\Gamma_{01}], \) and
- \( (k, \psi_{T02}, \gamma_2, \gamma_{22}) \in \mathcal{G}[\Gamma_{02}] \).

We are required to show that \( (k, \psi, \gamma_1(v_1v_2), \gamma_2(v_1v_2)) \in \mathcal{C} \equiv (k, \psi, (\gamma_1(v_1) \gamma_1(v_2)), (\gamma_2(v_1) \gamma_2(v_2))) \in \mathcal{C} \equiv (k, \psi, (\gamma_{11}(v_1) \gamma_{12}(v_2)), (\gamma_{21}(v_1) \gamma_{22}(v_2))) \in \mathcal{C} \).

Consider arbitrary \( j, \sigma_1, \sigma_2, \psi, \sigma, \psi, v_1, \sigma f_1, \) and \( T f_1 \) such that

- \( j < k \),
- \( \sigma_1, \sigma_2 : (\psi_0 \cup \psi) \rightarrow \mathcal{S} \),
- \( \sigma_1, \gamma_{11}(v_1) \gamma_{12}(v_2) \upharpoonright v_1, \sigma f_1, \) and \( \mathcal{S} \cap \text{alloc}(T f_1) = \emptyset \).

Hence, by inspection of the evaluation rules, it follows that

- \( \gamma_{11}(v_1) = \lambda x. e_1 \),
- \( \sigma_1, e_1 \gamma_{12}(v_2)/x \downarrow v_1, \sigma f_1, \) and
- \( j > 0 \).

Consider arbitrary \( v_2, \sigma f_2, \) and \( T f_2 \) such that

- \( \sigma_2, \gamma_{21}(v_1) \gamma_{22}(v_2) \downarrow v_2, \sigma f_2, \) and
- \( \mathcal{S} \cap \text{alloc}(T f_2) = \emptyset \).

Hence, by inspection of the evaluation rules, it follows that

- \( \gamma_{21}(v_1) = \lambda x. e_2 \), and
- \( \sigma_2, e_2 \gamma_{22}(v_2)/x \downarrow v_2, \sigma f_2, \).

Applying the induction hypothesis to \( v_1 \) (recall \( \Gamma_{01} = FV(v_1) \)), we conclude that \( \Gamma_{01} \vdash v_1 \Rightarrow v_1 \).

Instantiate this with \( k, \psi_{T01}, \gamma_{11}, \) and \( \gamma_{21} \). Note that

- \( k \geq 0 \) and
- \( (k, \psi_{T01}, \gamma_{11}, \gamma_{21}) \in \mathcal{G}[\Gamma_{01}], \)
  which follows from above.

Hence, \( (k, \psi_{T01}, \gamma_{11}(v_1), \gamma_{21}(v_1)) \in \mathcal{C} \).
Instantiate the latter with \( 0, \sigma_1, \sigma_2, (\psi_{T02} \cup \psi), \mathcal{S}, \lambda x. e_1, \sigma_1, \) and \( \varepsilon \). Note that

- \( 0 < k, \)
  which follows from \( 0 < j \) and \( j < k \),
- \( \sigma_1, \sigma_2 : k_1 (\psi_{\Gamma_01} \odot (\psi_{\Gamma_02} \odot \psi_r)) \Rightarrow \mathcal{S} \),
  which follows from \( \sigma_1, \sigma_2 : k_1 (\psi_r \odot \psi_r) \Rightarrow \mathcal{S} \) (from above) and \( \psi_T = \psi_{\Gamma_01} \odot \psi_{\Gamma_02} \) (from above),

- \( \sigma_1, \gamma_{11} (v_1) \downarrow^0 \lambda_x. e_1, \sigma_1, \varepsilon \),
  which follows by inspection of the evaluation rules since \( \gamma_{11} (v_1) = \lambda_x. e_1 \), and

- \( \mathcal{S}^1 \cap \text{alloc}(\varepsilon) = \emptyset \),
  which follows from alloc(\varepsilon) = \emptyset.

Hence, we have

\[
\forall v_{21}, \sigma_{21}, T_{21}.
  \begin{align*}
    \sigma_{21}, \gamma_{21} (v_1) \downarrow v_{21}, \sigma_{21}, T_{21} \land \\
    \mathcal{S}^2 \cap \text{alloc}(T_{21}) = \emptyset \implies \\
    \exists \psi_{f_{01}}, \mathcal{S}_{f_{01}}. (k - 0, \psi_{f_{01}}, \lambda_x. e_1, v_{21}) \in \mathcal{V} \land \\
    \sigma_{1}, \sigma_{2} : k - 0 (\psi_{f_{01}} \odot (\psi_{\Gamma_02} \odot \psi_r)) \Rightarrow \mathcal{S}_{f_{01}} \land \\
    \mathcal{S}_{f_{01}} \subseteq \mathcal{S}^1 \cup \text{alloc}(\varepsilon) \land \\
    \mathcal{S}_{f_{01}} \subseteq \mathcal{S}^2 \cup \text{alloc}(T_{21})
  \end{align*}
\]

Instantiate this with \( \lambda_x. e_2, \sigma_2, \varepsilon \). Note that

- \( \sigma_2, \gamma_{21} (v_1) \downarrow \lambda_x. e_2, \sigma_2, \varepsilon \),
  which follows by inspection of the evaluation rules since \( \gamma_{21} (v_1) = \lambda_x. e_2 \), and

- \( \mathcal{S}^2 \cap \text{alloc}(\varepsilon) = \emptyset \),
  which follows from alloc(\varepsilon) = \emptyset.

Hence, there exist \( \psi_{f_{01}} \) and \( \mathcal{S}_{f_{01}} \) such that

- \( (k - 0, \psi_{f_{01}}, \lambda_x. e_1, \lambda_x. e_2) \in \mathcal{V} \),
- \( \sigma_{1}, \sigma_{2} : k - 0 (\psi_{f_{01}} \odot (\psi_{\Gamma_02} \odot \psi_r)) \Rightarrow \mathcal{S}_{f_{01}} \),
- \( \mathcal{S}_{f_{01}} \subseteq \mathcal{S}^1 \cup \text{alloc}(\varepsilon) \), and
- \( \mathcal{S}_{f_{01}} \subseteq \mathcal{S}^2 \cup \text{alloc}(\varepsilon) \).

Applying the induction hypothesis to \( v_2 \) (recall \( \Gamma_{02} = FV (v_2) \)), we conclude that \( \Gamma_{02} \vdash v_2 \preceq v_2 \).

Instantiate this with \( k, \psi_{\Gamma_02}, \gamma_{12}, \gamma_{22} \). Note that

- \( k \geq 0 \) and
- \( (k, \psi_{\Gamma_02}, \gamma_{12}, \gamma_{22}) \in \mathcal{G} \| \Gamma_{02} \| \),
  which follows from above.

Hence, \( (k, \psi_{\Gamma_02}, \gamma_{12}(v_2), \gamma_{22}(v_2)) \in \mathcal{C} \).

Instantiate the latter with \( 0, \sigma_1, \sigma_2, (\psi_{f_{01}} \odot \psi_r), \mathcal{S}_{f_{01}}, \gamma_{12}(v_2), \sigma_1, \varepsilon \). Note that

- \( 0 < k \),
  which follows from \( 0 < j \) and \( j < k \),

- \( \sigma_1, \sigma_2 : k (\psi_{\Gamma_02} \odot (\psi_{f_{01}} \odot \psi_r)) \Rightarrow \mathcal{S}_{f_{01}} \),
  which follows from \( \sigma_1, \sigma_2 : k (\psi_{f_{01}} \odot (\psi_{\Gamma_02} \odot \psi_r)) \Rightarrow \mathcal{S}_{f_{01}} \) (from above),

- \( \sigma_1, \gamma_{12}(v_2) \downarrow^0 \gamma_{12}(v_2), \sigma_1, \varepsilon \),
  which follows by inspection of the evaluation rules since \( \gamma_{12}(v_2) \) is a value, and

- \( \mathcal{S}_{f_{01}} \cap \text{alloc}(\varepsilon) = \emptyset \),
  which follows from alloc(\varepsilon) = \emptyset.
Hence, we have
\[ \forall v_{22}, \sigma_{22}, T_{22}. \]
\[ \sigma_{22}, \gamma_{22}(v_{22}) \downarrow v_{22}, \sigma_{22}, T_{22} \land \]
\[ S^{f_{01}} \cap \text{alloc}(T_{22}) = \emptyset \implies \]
\[ \exists \psi_{f_{02}}, S_{f_{02}}. (k - 0, \psi_{f_{02}}, \gamma_{12}(v_{22}), v_{22}) \in V \land \]
\[ \sigma_{1}, \sigma_{22} : k \rightarrow (\psi_{f_{02}} \circ (\psi_{f_{01}} \circ \psi_{r})) \leadsto S_{f_{02}} \land \]
\[ S^{f_{02}} \subseteq S^{f_{01}} \cup \text{alloc}(\epsilon) \land \]
\[ S^{f_{02}} \subseteq S^{f_{01}} \cup \text{alloc}(T_{22}) \]

Instantiate the latter with \( \gamma_{22}(v_{22}), \sigma_{22}, \) and \( \epsilon \). Note that

- \( \sigma_{2}, \gamma_{22}(v_{22}) \downarrow \gamma_{22}(v_{22}), \sigma_{2}, \epsilon \),
  which follows by inspection of the evaluation rules since \( \gamma_{22}(v_{22}) \) is a value, and
- \( S^{f_{01}} \cap \text{alloc}(\epsilon) = \emptyset \),
  which follows from \( \text{alloc}(\epsilon) = \emptyset \).

Hence, there exist \( \psi_{f_{02}} \) and \( S_{f_{02}} \) such that

- \( (k - 0, \psi_{f_{02}}, \gamma_{12}(v_{22}), \gamma_{22}(v_{22})) \in V \),
- \( \sigma_{1}, \sigma_{2} : k \rightarrow (\psi_{f_{02}} \circ (\psi_{f_{01}} \circ \psi_{r})) \leadsto S_{f_{02}} \),
- \( S^{f_{02}} \subseteq S^{f_{01}} \cup \text{alloc}(\epsilon) \), and
- \( S^{f_{02}} \subseteq S^{f_{01}} \cup \text{alloc}(\epsilon) \).

Instantiate \( (k, \psi_{f_{01}}, \lambda x. e_{1}, \lambda x. e_{2}) \in V \) with \( k - 1, \psi_{f_{02}}, \gamma_{12}(v_{22}), \) and \( \gamma_{22}(v_{22}) \). Note that

- \( k - 1 < k \),
- \( (k - 1, \psi_{f_{02}}, \gamma_{12}(v_{22}), \gamma_{22}(v_{22})) \in V \),
  which follows from Lemma B.1 applied to \( (k, \psi_{f_{02}}, \gamma_{12}(v_{22}), \gamma_{22}(v_{22})) \in V \) (from above) and \( k - 1 < k \), and
- \( (\psi_{f_{01}} \circ \psi_{f_{02}}) \) defined,
  which follows from \( \sigma_{1}, \sigma_{2} : k \rightarrow (\psi_{f_{02}} \circ (\psi_{f_{01}} \circ \psi_{r})) \leadsto S_{f_{02}} \) (from above).

Hence, \( (k - 1, \psi_{f_{01}} \circ \psi_{f_{02}}, e_{1}[\gamma_{12}(v_{22})/x], e_{2}[\gamma_{22}(v_{22})/x]) \in C \).

Instantiate the latter with \( j - 1, \sigma_{1}, \sigma_{2}, \psi_{r}, S_{f_{02}}, v_{f_{1}}, \sigma_{f_{1}}, T_{f_{1}} \). Note that

- \( j - 1 < k - 1 \),
  which follows from \( j < k \),
- \( \sigma_{1}, \sigma_{2} : k \rightarrow ((\psi_{f_{01}} \circ \psi_{f_{02}}) \circ \psi_{r}) \leadsto S_{f_{02}} \),
  which follows from Lemma B.3 applied to
  - \( \sigma_{1}, \sigma_{2} : k \rightarrow ((\psi_{f_{01}} \circ \psi_{f_{02}}) \circ \psi_{r}) \leadsto S_{f_{02}} \),
    which follows from \( \sigma_{1}, \sigma_{2} : k \rightarrow ((\psi_{f_{01}} \circ \psi_{f_{02}}) \circ \psi_{r}) \leadsto S_{f_{02}} \) (from above),
    and
  - \( k - 1 < k \)
- \( \sigma_{1}, e_{1}[\gamma_{12}(v_{22})/x] \downarrow^{j - 1} v_{f_{1}}, \sigma_{f_{1}}, T_{f_{1}} \),
  which follows from above, and
- \( S^{f_{02}} \cap \text{alloc}(T_{f_{1}}) = \emptyset \),
  which follows from above, and
- \( S^{f_{02}} \subseteq S^{f_{01}} \subseteq S^{f_{1}} \) (from above) and \( S^{f_{1}} \cap \text{alloc}(T_{f_{1}}) = \emptyset \) (from above).
Hence, we have
\[ \forall v_{f2}, \sigma_{f2}, T_{f2}, \]
\[ \sigma_2, e_2[\gamma_{22}(v_2)/x] \downarrow v_{f2}, \sigma_{f2}, T_{f2} \land \]
\[ S_{f02}^2 \cap \text{alloc}(T_{f2}) = \emptyset \implies \]
\[ \exists \psi_f, S_f. \ ((k - 1) - (j - 1), \psi_f, v_{f1}, v_{f2}) \in \mathcal{V} \land \]
\[ \sigma_{f1}, \sigma_{f2} : (k-1)-(j-1) (\psi_f \circ \psi_r) \Rightarrow S_f \land \]
\[ S_f^1 \subseteq S_{f02}^1 \cup \text{alloc}(T_{f1}) \land \]
\[ S_f^2 \subseteq S_{f02}^2 \cup \text{alloc}(T_{f2}) \]

Instantiate the latter with \( v_{f2}, \sigma_{f2}, \) and \( T_{f2} \). Note that
1. \( \sigma_2, e_2[\gamma_{22}(v_2)/x] \downarrow v_{f2}, \sigma_{f2}, T_{f2} \)
   which follows from above, and
2. \( S_{f02}^2 \cap \text{alloc}(T_{f2}) = \emptyset \)
   which follows from \( S^2 \cap \text{alloc}(T_{f2}) = \emptyset \) (from above) and \( S_{f02}^2 \subseteq S_{f01}^2 \subseteq S^2 \) (from above).

Hence, there exist \( \psi_f \) and \( S_f \) such that
1. \( (k - j, \psi_f, v_{f1}, v_{f2}) \in \mathcal{V} \),
2. \( \sigma_{f1}, \sigma_{f2} : (k-1)-(j-1) (\psi_f \circ \psi_r) \Rightarrow S_f \),
3. \( S_f^1 \subseteq S_{f02}^1 \cup \text{alloc}(T_{f1}) \), and
4. \( S_f^2 \subseteq S_{f02}^2 \cup \text{alloc}(T_{f2}) \).

Take \( \psi_f = \psi_f \). Take \( S_f = S_f \).

We are required to show:
1. \( (k - j, \psi_f, v_{f1}, v_{f2}) \in \mathcal{V} \), which follows from above,
2. \( \sigma_{f1}, \sigma_{f2} : (k-1)-(j-1) (\psi_f \circ \psi_r) \Rightarrow S_f \)
which follows from above,
3. \( S_f^1 \subseteq S^1 \cup \text{alloc}(T_{f1}) \)
which follows from
   \[ S_f^1 \subseteq S_{f02}^1 \cup \text{alloc}(T_{f1}) \]
   which follows from above
   \[ \subseteq S_{f01}^1 \cup \text{alloc}(T_{f1}) \]
   which follows from \( S_{f02}^1 \subseteq S_{f01}^1 \) (from above)
   \[ \subseteq S^1 \cup \text{alloc}(T_{f1}) \]
   which follows from \( S_{f01}^1 \subseteq S^1 \) (from above)
4. \( S_f^2 \subseteq S^2 \cup \text{alloc}(T_{f2}) \)
which follows from
   \[ S_f^2 \subseteq S_{f02}^2 \cup \text{alloc}(T_{f2}) \]
   which follows from above
   \[ \subseteq S_{f01}^2 \cup \text{alloc}(T_{f2}) \]
   which follows from \( S_{f02}^2 \subseteq S_{f01}^2 \) (from above)
   \[ \subseteq S^2 \cup \text{alloc}(T_{f2}) \]
   which follows from \( S_{f01}^2 \subseteq S^2 \) (from above).
Case (LET) let \( x = e_1 \) in \( e_2 \):

Note that \( \Gamma = FV(\text{let } x = e_1 \text{ in } e_2) \equiv FV(e_1) \cup (FV(e_2) \setminus \{x\}) \).

Let \( \Gamma_01 = FV(e_1) \) and \( \Gamma_02 = (FV(e_2) \setminus \{x\}) \). Hence, \( \Gamma = \Gamma_01 \cup \Gamma_02 \).

Consider arbitrary \( k, \psi_\Gamma, \gamma_1, \) and \( \gamma_2 \) such that

- \( k \geq 0 \), and
- \( (k, \psi_\Gamma, \gamma_1, \gamma_2) \in \mathcal{G}[\Gamma] \).

Note that we have \( \gamma_{11}, \gamma_{12}, \gamma_{21}, \gamma_{22}, \) and \( \psi_{01}, \psi_{02} \) such that

- \( \gamma_1 = \gamma_{11} \cup \gamma_{12} \),
- \( \gamma_2 = \gamma_{21} \cup \gamma_{22} \),
- \( \psi_\Gamma = \psi_{01} \circ \psi_{02} \),
- \( (k, \psi_{01}, \gamma_{11}, \gamma_{21}) \in \mathcal{G}[\Gamma_{01}] \), and
- \( (k, \psi_{02}, \gamma_{12}, \gamma_{22}) \in \mathcal{G}[\Gamma_{02}] \).

We are required to show that
\[
(k, \psi_{01}, \gamma_{11}(\text{let } x = e_1 \text{ in } e_2), \gamma_{21}(\text{let } x = e_1 \text{ in } e_2)) \in C
\]

\[
\equiv (k, \psi_{01}, \text{let } x = \gamma_{11}(e_1) \text{ in } \gamma_{21}(e_2), \text{let } x = \gamma_{12}(e_1) \text{ in } \gamma_{22}(e_2)) \in C
\]

Consider arbitrary \( j, \sigma_1, \sigma_2, \psi_r, S, v_{f_1}, \sigma_{f_1}, \) and \( T_{f_1} \) such that

- \( j < k \),
- \( \sigma_1, \sigma_2 : k (\psi_{01} \circ \psi_r) \hookrightarrow \mathcal{S} \),
- \( \sigma_1, \text{let } x = \gamma_{11}(e_1) \text{ in } \gamma_{12}(e_2) \psi_r v_{f_1}, \sigma_{f_1}, T_{f_1} \),
- \( \mathcal{S}^1 \cap \text{alloc}(T_{f_1}) = \emptyset \).

Hence, by inspection of the evaluation rules, it follows that there exist \( j_1, j_2, v_{11}, \sigma_{11}, T_{11}, \) and \( T_{12} \) such that

- \( \sigma_1, \gamma_{11}(e_1) \psi_r v_{11}, \sigma_{11}, T_{11} \),
- \( \sigma_{11}, \gamma_{12}(e_2)[v_{11}/x] \psi_r v_{f_1}, \sigma_{f_1}, T_{12} \),
- \( \text{alloc}(T_{11}) \cap \text{alloc}(T_{12}) = \emptyset \),
- \( T_{f_1} = \text{let } T_{11} T_{12} \), and
- \( j = j_1 + j_2 + 1 \).

Consider arbitrary \( v_{f_2}, \sigma_{f_2}, \) and \( T_{f_2} \) such that

- \( \sigma_2, \text{let } x = \gamma_{21}(e_1) \text{ in } \gamma_{22}(e_2) \psi_r v_{f_2}, \sigma_{f_2}, T_{f_2} \), and
- \( \mathcal{S}^2 \cap \text{alloc}(T_{f_2}) = \emptyset \).

Hence, by inspection of the evaluation rules, it follows that there exist \( v_{21}, \sigma_{21}, T_{21}, \) and \( T_{22} \) such that

- \( \sigma_2, \gamma_{21}(e_1) \psi_r v_{21}, \sigma_{21}, T_{21} \),
- \( \sigma_{21}, \gamma_{22}(e_2)[v_{21}/x] \psi_r v_{f_2}, \sigma_{f_2}, T_{22} \),
- \( \text{alloc}(T_{21}) \cap \text{alloc}(T_{22}) = \emptyset \), and
- \( T_{f_2} = \text{let } T_{21} T_{22} \).

Applying the induction hypothesis to \( e_1 \) (recall \( \Gamma_{01} = FV(e_1) \)), we conclude that \( \Gamma_{01} \vdash e_1 \ll e_1 \).

Instantiate this with \( k, \psi_{01}, \gamma_{11}, \) and \( \gamma_{21} \). Note that

- \( k \geq 0 \) and
• \((k, \psi_{\Gamma_01}, \gamma_{11}, \gamma_{21}) \in \mathcal{G}[\Gamma_{01}]\),
  which follows from above.

Hence, \((k, \psi_{\Gamma_01}, \gamma_{11}(e_1), \gamma_{21}(e_1)) \in C\).

Instantiate the latter with \(j_1, \sigma_1, \sigma_2, (\psi_{\Gamma_02} \circ \psi_r), S, v_{11}, \sigma_{11}, \) and \(T_{11}\). Note that

• \(j_1 < k\),
  which follows from \(j_1 < j \) and \(j < k\),
• \(\sigma_1, \sigma_2 \vdash (\psi_{\Gamma_01} \circ (\psi_{\Gamma_02} \circ \psi_r)) \leadsto S\),
  which follows from \(\sigma_1, \sigma_2 \vdash (\psi_T \circ \psi_r) \leadsto S\) (from above) and \(\psi_T = \psi_{\Gamma_01} \circ \psi_{\Gamma_02}\) (from above),
• \(\sigma_1, \gamma_{11}(e_1) \downarrow j_1, v_{11}, \sigma_{11}, T_{11}\),
  which follows from above, and
• \(S^1 \cap \text{alloc}(T_{11}) = \emptyset\),
  which follows from \(S^1 \cap \text{alloc}(\text{let} T_{11} T_{12}) = \emptyset\),
  which in turn follows from \(S^1 \cap \text{alloc}(T_{j_1}) = \emptyset\) (since \(T_{j_1} = \text{let} T_{11} T_{12}\)),
  which follows from above.

Hence, we have

\[
\forall v_{21}, \sigma_{21}, T_{21},
  \sigma_2, \gamma_{21}(e_1) \downarrow v_{21}, \sigma_{21}, T_{21} \land
  S^2 \cap \text{alloc}(T_{21}) = \emptyset \implies
  \exists \psi_{f_{01}}, S_{f_{01}}, (k - j_1, \psi_{f_{01}}, v_{11}, v_{21}) \in \mathcal{V} \land
  \sigma_{11}, \sigma_{21} \vdash (\psi_{f_{01}} \circ (\psi_{\Gamma_02} \circ \psi_r)) \leadsto S_{f_{01}} \land
  S_{f_{01}}^1 \subseteq S^1 \cup \text{alloc}(T_{11}) \land
  S_{f_{01}}^2 \subseteq S^2 \cup \text{alloc}(T_{21})
\]

Instantiate the latter with \(v_{21}, \sigma_{21}, \) and \(T_{21}\). Note that

• \(\sigma_2, \gamma_{21}(e_1) \downarrow v_{21}, \sigma_{21}, T_{21}\), and
• \(S^2 \cap \text{alloc}(T_{21}) = \emptyset\),
  which follows from \(S^2 \cap \text{alloc}(\text{let} T_{21} T_{22}) = \emptyset\),
  which in turn follows from \(S^2 \cap \text{alloc}(T_{j_2}) = \emptyset\) (since \(T_{j_2} = \text{let} T_{21} T_{22}\)),
  which follows from above.

Hence, there exist \(\psi_{f_{01}}\) and \(S_{f_{01}}\) such that

• \((k - j_1, \psi_{f_{01}}, v_{11}, v_{21}) \in \mathcal{V}\),
• \(\sigma_{11}, \sigma_{21} \vdash (\psi_{f_{01}} \circ (\psi_{\Gamma_02} \circ \psi_r)) \leadsto S_{f_{01}}\),
• \(S_{f_{01}}^1 \subseteq S^1 \cup \text{alloc}(T_{11})\), and
• \(S_{f_{01}}^2 \subseteq S^2 \cup \text{alloc}(T_{21})\).

Applying the induction hypothesis to \(e_2\) (noting that \(FV(e_2) = \Gamma_{02} \cup \{x\}\)), we conclude that \(\Gamma_{02}, x \vdash e_2 \equiv e_2\).

Instantiate this with \((k - j_1, (\psi_{\Gamma_02} \circ \psi_{f_{01}}), \gamma_{12}[x \mapsto v_{11}], \gamma_{22}[x \mapsto v_{21}]\). Note that

• \(k - j_1 \geq 0\),
  which follows from \(j_1 < j < k\) (from above), and
• \((k - j_1, \psi_{\Gamma_02} \circ \psi_{f_{01}}, \gamma_{12}[x \mapsto v_{11}], \gamma_{22}[x \mapsto v_{21}]) \in \mathcal{G}[\Gamma_{02}, x]\),
  which follows from
  • \((k - j_1, \psi_{\Gamma_02}, \gamma_{12}, \gamma_{22}) \in \mathcal{G}[\Gamma_{02}]\),
    which follows from Lemma B.2 applied to \((k, \psi_{\Gamma_02}, \gamma_{12}, \gamma_{22}) \in \mathcal{G}[\Gamma_{02}]\) (from above) and \(k - j_1 \leq k\), and
  • \(k - j_1 \leq k\), and
• \((k - j_1, \psi_{j1}, v_{i11}, v_{i21}) \in \mathbb{V}\),
which follows from above.

Hence, \((k - j_1, \psi_{j2} \circ \psi_{j1}, \gamma_{i2}[x \mapsto v_{i11}](e_2), \gamma_{i22}[x \mapsto v_{i21}](e_2)) \in \mathbb{C}\)
\(\equiv (k - j_1, \psi_{j2} \circ \psi_{j1}, \gamma_{i2}(e_2)[v_{i11}/x], \gamma_{i22}(e_2)[v_{i21}/x]) \in \mathbb{C}\).

Instantiate the latter with \(j_2, \sigma_{i1}, \sigma_{i2}, \psi_r, S_{f01}, v_{f1}, \sigma_{f1}, T_{i12}\). Note that

- \(j_2 < k - j_1\),
which follows from \(j_2 + j_1 + 1 = j\) (from above) and \(j < k\) (from above),
- \(\sigma_{i1}, \sigma_{i2} \vdash_{k-j_1} ((\psi_{j2} \circ \psi_{j1}) \circ \psi_r) \rightharpoonup S_{f01}\),
which follows from \(\sigma_1, \sigma_2 \vdash_{k} (\psi_{j1} \circ (\psi_{j2} \circ \psi_r)) \rightharpoonup S_{f01}\) (from above),
- \(\sigma_{i1}, \gamma_{i2}(e_2)[v_{i11}/x] \downarrow v_{f1}, \sigma_{f1}, T_{i12}\),
which follows from above, and
- \(S_{j01} \cap \text{alloc}(T_{i12}) = \emptyset\),
which follows from \(S_{j01} \subseteq S^1 \cup \text{alloc}(T_{i11})\) (from above), together with:

- \(S^1 \cap \text{alloc}(T_{i12}) = \emptyset\),
which follows from \(S^1 \cap \text{alloc(let } T_{i11} T_{i12}) = \emptyset\),
which in turn follows from \(S^1 \cap \text{alloc}(T_{i1}) = \emptyset\) (since \(T_{i1} = \text{let } T_{i1} T_{i12}\)),
which follows from above, and
- \(\text{alloc}(T_{i11}) \cap \text{alloc}(T_{i12}) = \emptyset\),
which follows from above.

Hence, we have
\[\forall v_{f2}, \sigma_{f2}, T_{i22}.
\sigma_{i21}, \gamma_{i2}(e_2)[v_{i21}/x] \downarrow v_{f2}, \sigma_{f2}, T_{i22} \land
S_{j01} \cap \text{alloc}(T_{i22}) = \emptyset \Rightarrow
\exists \psi_{j02}, S_{j02}.\ (k - j_1 - j_2, \psi_{j02}, v_{f1}, v_{f2}) \in \mathbb{V} \land
\sigma_{j1}, \sigma_{j2} \vdash_{k-j_1-j_2} \psi_{j02} \circ \psi_r \rightharpoonup S_{j02} \land
S_{j02} \subseteq S_{j01} \cup \text{alloc}(T_{i12}) \land
S_{j02} \subseteq S^2 \cup \text{alloc}(T_{i22})\]

Instantiate the latter with \(v_{f2}, \sigma_{f2}, T_{i22}\). Note that

- \(\sigma_{i21}, \gamma_{i2}(e_2)[v_{i21}/x] \downarrow v_{f2}, \sigma_{f2}, T_{i22}\), and
- \(S_{j01} \cap \text{alloc}(T_{i22}) = \emptyset\),
which follows from \(S_{j01} \subseteq S^2 \cup \text{alloc}(T_{i21})\) (from above), together with:

- \(S^2 \cap \text{alloc}(T_{i22}) = \emptyset\),
which follows from \(S^2 \cap \text{alloc(let } T_{i21} T_{i22}) = \emptyset\) (since \(T_{i2} = \text{let } T_{i21} T_{i22}\)),
which follows from above, and
- \(\text{alloc}(T_{i21}) \cap \text{alloc}(T_{i22}) = \emptyset\),
which follows from above.

Hence, there exist \(\psi_{j02}\) and \(S_{j02}\) such that

- \((k - j_1 - j_2, \psi_{j02}, v_{f1}, v_{f2}) \in \mathbb{V}\),
- \(\sigma_{j1}, \sigma_{j2} \vdash_{k-j_1-j_2} \psi_{j02} \circ \psi_r \rightharpoonup S_{j02}\),
- \(S_{j02} \subseteq S_{j01} \cup \text{alloc}(T_{i12})\), and
- \(S_{j02} \subseteq S^2 \cup \text{alloc}(T_{i22})\).
Take $\psi_f = \psi_{f02}$. Take $S_f = S_{f02}$.

We are required to show:

- $(k - j, \psi_f, v_{f1}, v_{f2}) \in \mathcal{V}$,
  $\equiv (k - j, \psi_{f02}, v_{f1}, v_{f2}) \in \mathcal{V}$,
  which follows from Lemma B.1 applied to $(k - j_1 - j_2, \psi_{f02}, v_{f1}, v_{f2}) \in \mathcal{V}$ (from above) and $k - j < k - j_1 - j_2$ (from $j = j_1 + j_2 + 1$),

- $\sigma_{f1}, \sigma_{f2} : k - j \ (\psi_f \odot \psi_r) \leadsto S_f$
  $\equiv \sigma_{f1}, \sigma_{f2} : k - j \ (\psi_{f02} \odot \psi_r) \leadsto S_{f02}$,
  which follows from Lemma B.3 applied to $\sigma_{f1}, \sigma_{f2} : k - j_1 - j_2 \ (\psi_{f02} \odot \psi_r) \leadsto S_{f02}$ (from above) and $k - j < k - j_1 - j_2$ (from $j = j_1 + j_2 + 1$),

- $S_{f1} \subseteq S^1 \cup \text{alloc}(T_{f1})$
  $\equiv S_{f02}^{1} \subseteq S^1 \cup \text{alloc}(T_{f1})$
  which follows from

\[
S_{f02}^{1} \subseteq S_{f01}^1 \cup \text{alloc}(T_{f1})
\]
  which follows from above
\[
\subseteq S^1 \cup \text{alloc}(T_{f1}) \cup \text{alloc}(T_{f1})
\]
  which follows from $S_{f01}^1 \subseteq S^1 \cup \text{alloc}(T_{f1})$ (from above)
\[
\equiv S^1 \cup \text{alloc(let T_{11} T_{12})}
\]
  which follows from the definition of alloc
\[
\equiv S^1 \cup \text{alloc}(T_{f1})
\]
  which follows from $T_{f1} = \text{let T_{11} T_{12}}$ (from above).

- $S_{f2} \subseteq S^2 \cup \text{alloc}(T_{f2})$
  $\equiv S_{f02}^2 \subseteq S^2 \cup \text{alloc}(T_{f2})$
  which follows from

\[
S_{f02}^2 \subseteq S_{f01}^2 \cup \text{alloc}(T_{f2})
\]
  which follows from above
\[
\subseteq S^2 \cup \text{alloc}(T_{f2}) \cup \text{alloc}(T_{f2})
\]
  which follows from $S_{f01}^2 \subseteq S^2 \cup \text{alloc}(T_{f2})$ (from above)
\[
\equiv S^2 \cup \text{alloc(let T_{21} T_{22})}
\]
  which follows from the definition of alloc
\[
\equiv S^2 \cup \text{alloc}(T_{f2})
\]
  which follows from $T_{f2} = \text{let T_{21} T_{22}}$ (from above).
Case \((\text{MOD}) \mod v\):

Note that \(\Gamma = \text{FV}(v)\).

Consider arbitrary \(k, \psi_T, \gamma_1, \) and \(\gamma_2\) such that

- \(k \geq 0,\)
- \((k, \psi_T, \gamma_1, \gamma_2) \in G[\Gamma]\).

We are required to show that \((k, \psi_T, \gamma_1(v) \mod v, \gamma_2(v)) \in C\)

\[\equiv (k, \psi_T, \mod \gamma_1(v), \mod \gamma_2(v)) \in C.\]

Consider arbitrary \(j, \sigma_1, \sigma_2, \psi_r, S, v_{f1}, \sigma_{f1},\) and \(T_{f1}\) such that

- \(j < k,\)
- \(\sigma_1, \sigma_2 : k (\psi_T \odot \psi_r) \twoheadrightarrow S,\)
- \(\sigma_1, \mod \gamma_1(v) \Downarrow v_{f1}, \sigma_{f1}, T_{f1},\) and
- \(S^1 \cap \text{alloc}(T_{f1}) = \emptyset.\)

Hence, by inspection of the evaluation rules, it follows that

- \(v_{f1} = l_1,\)
- \(\sigma_{f1} = \sigma_1[l_1 \leftarrow \gamma_1(v)],\)
- \(T_{f1} = \mod l_1 \leftarrow \gamma_1(v),\) and
- \(j = 1.\)

Note that \(S^1 \cap \{l_1\} = \emptyset,\) which follows from

\[S^1 \cap \text{alloc}(T_{f1}) = \emptyset\]

which follows from above

\[\equiv S^1 \cap \text{alloc}(\mod l_1 \leftarrow \gamma_1(v)) = \emptyset\]

which follows from \(T_{f1} = \mod l_1 \leftarrow \gamma_1(v)\) (from above)

\[\equiv S^1 \cap \{l_1\} = \emptyset\]

which follows from the definition of \(\text{alloc}.\)

Consider arbitrary \(v_{f2}, \sigma_{f2},\) and \(T_{f2}\) such that

- \(\sigma_2, \mod \gamma_2(v) \Downarrow v_{f2}, \sigma_{f2}, T_{f2},\) and
- \(S^2 \cap \text{alloc}(T_{f2}) = \emptyset.\)

Hence, by inspection of the evaluation rules, it follows that

- \(v_{f2} = l_2,\)
- \(\sigma_{f2} = \sigma_2[l_2 \leftarrow \gamma_2(v)],\) and
- \(T_{f2} = \mod l_2 \leftarrow \gamma_2(v).\)

Note that \(S^2 \cap \{l_2\} = \emptyset,\) which follows from

\[S^2 \cap \text{alloc}(T_{f2}) = \emptyset\]

which follows from above

\[\equiv S^2 \cap \text{alloc}(\mod l_2 \leftarrow \gamma_2(v)) = \emptyset\]

which follows from \(T_{f2} = \mod l_2 \leftarrow \gamma_2(v)\) (from above)

\[\equiv S^2 \cap \{l_2\} = \emptyset\]

which follows from the definition of \(\text{alloc}.\)

Applying the induction hypothesis to \(v\) (noting that \(\Gamma = \text{FV}(v)\)), we conclude that \(\Gamma \vdash v \ll v.\)

Instantiate this with \(k, \psi_T, \gamma_1, \) and \(\gamma_2.\) Note that

- \(k \geq 0\) and
• \((k, \psi_T, \gamma_1, \gamma_2) \in G[\Gamma]\), which follows from above.

Hence, \((k, \psi_T, \gamma_1(v), \gamma_2(v)) \in C\).

Instantiate the latter with 0, \(\sigma_1, \sigma_2, \psi_r, \gamma_1(v), \sigma_1, \) and \(\epsilon\). Note that

• \(0 < k\), which follows from \(j = 1\) and \(j < k\),

• \(\sigma_1, \sigma_2 \vdash_k (\psi_T \circ \psi_r) \leadsto S\), which follows from above,

• \(\sigma_1, \gamma_1(v) \downarrow^0 \gamma_1(v), \sigma_1, \epsilon\), which follows by inspection of the evaluation rules since \(\gamma_1(v)\) is a value, and

• \(S^1 \cap alloc(\epsilon) = \emptyset\), which follows from \(alloc(\epsilon) = \emptyset\).

Hence, we have

\[
\forall v_{20}, \sigma_{20}, T_{20}.
\]

\[
\sigma_2, \gamma_2(v) \downarrow v_{20}, \sigma_{20}, T_{20} \land S^2 \cap alloc(T_{20}) = \emptyset \implies \exists \psi_{f0}, S_{f0}. (k - 0, \psi_{f0}, \gamma_1(v), v_{20}) \in V \land \sigma_1, \sigma_{20} \vdash_{k-0} (\psi_{f0} \circ \psi_r) \leadsto S_{f0} \land S^1_{f0} \subseteq S^1 \cup alloc(\epsilon) \land S^2_{f0} \subseteq S^2 \cup alloc(T_{20})
\]

Instantiate this with \(\gamma_2(v), \sigma_2, \) and \(\epsilon\). Note that

• \(\sigma_2, \gamma_2(v) \downarrow \gamma_2(v), \sigma_2, \epsilon\), which follows by inspection of the evaluation rules since \(\gamma_2(v)\) is a value, and

• \(S^2 \cap alloc(\epsilon) = \emptyset\), which follows from \(alloc(\epsilon) = \emptyset\).

Hence, there exist \(\psi_{f0}\) and \(S_{f0}\) such that

• \((k - 0, \psi_{f0}, \gamma_1(v), \gamma_2(v)) \in V\),

• \(\sigma_1, \sigma_{20} \vdash_{k-0} (\psi_{f0} \circ \psi_r) \leadsto S_{f0}\),

• \(S^1_{f0} \subseteq S^1 \cup alloc(\epsilon)\), and

• \(S^2_{f0} \subseteq S^2 \cup alloc(\epsilon)\).

Take \(\psi_f = \{(l_1, l_2)\}\). Take \(S_f = (S_{f0} \circ \{(l_1, l_2)\})\).

Note that \((S_{f0} \circ \{(l_1, l_2)\})\) defined (i.e., \((S_{f0} \circ \{(l_1, l_2)\}) \in LocB_{ij}\), which follows from

• \(S^1_{f0} \cap \{l_1\} = \emptyset\), which follows from

• \(S^1_{f0} \subseteq S^1\) (from above), and

• \(S^1 \cap \{l_1\} = \emptyset\) (from above)

and

• \(S^2_{f0} \cap \{l_2\} = \emptyset\), which follows from

• \(S^2_{f0} \subseteq S^2\) (from above), and

• \(S^2 \cap \{l_2\} = \emptyset\) (from above)
We are required to show:

- \((k - j, \psi_f, v_{f1}, v_{f2}) \in \mathcal{V}\)
  \(\equiv (k - 1, \{(l_1, l_2)\}, l_1, l_2) \in \mathcal{V}\),
  which follows from the definition of \(\mathcal{V}\),

- \(\sigma_f, \sigma_f : k \to (\psi_f \circ \psi_r) \mapsto \mathcal{S}_f\)
  \(\equiv \sigma_1[l_1 \leftarrow \gamma_1(v)], \sigma_2[l_2 \leftarrow \gamma_2(v)] : k \to \{(l_1, l_2)\} \circ \psi_r \mapsto (\mathcal{S}_{f0} \circ \{(l_1, l_2)\})\),
  which we conclude as follows:
  From \(\sigma_1, \sigma_2 : k \to (\psi_f \circ \psi_r) \mapsto \mathcal{S}_{f0}\), it follows that there exists \(\mathcal{F}_0\) such that

  - \(\mathcal{S}_{f0} \in \text{LocBij}\),
  - \(\mathcal{F}_0 : \mathcal{S}_{f0} \to \text{LocBij}\),
  - \(\mathcal{S}_{f0} = \psi_{f0} \circ \psi_r \bigcirc_{(l_{11}, l_{22}) \in \mathcal{S}_{f0}} \mathcal{F}_0(l_{11}, l_{22})\),
  - \(\text{dom}(\sigma_1) \supseteq \mathcal{S}_{f0}^{l_1}\),
  - \(\text{dom}(\sigma_2) \supseteq \mathcal{S}_{f0}^{l_2}\), and
  - \(\forall (l_{11}, l_{22}) \in \mathcal{S}_{f0}, \forall i < k\).

Take

\[ \mathcal{F}(l_{11}, l_{22}) = \begin{cases} \mathcal{F}_0(l_{11}, l_{22}) & \text{if } (l_{11}, l_{22}) \in \mathcal{S}_{f0} \\ \psi_{f0} & \text{if } (l_{11}, l_{22}) \in \{(l_1, l_2)\} \end{cases} \]

We are required to show:

- \(\mathcal{S}_f \in \text{LocBij}\)
  \(\equiv (\mathcal{S}_{f0} \circ \{(l_1, l_2)\}) \in \text{LocBij}\),
  which follows from above.

- \(\mathcal{F} : \mathcal{S}_f \to \text{LocBij}\)
  \(\equiv \mathcal{F} : (\mathcal{S}_{f0} \circ \{(l_1, l_2)\}) \to \text{LocBij}\),
  which follows from

  - \(\text{dom}(\mathcal{F}) = \mathcal{S}_{f0} \circ \{(l_1, l_2)\}\),
    which follows from the definition of \(\mathcal{F}\),

  - \(\text{rng} = (\psi_{f0} \circ \bigcirc_{(l_{11}, l_{22}) \in \mathcal{S}_{f0}} \mathcal{F}_0(l_{11}, l_{22})) \in \text{LocBij}\),
    which follows from \(\text{rng} \subseteq \mathcal{S}_{f0}\) (from above) and \(\mathcal{S}_{f0} \in \text{LocBij}\).
\[ S_f = (\{(l_1, l_2)\} \odot \psi_r \circ \bigcup_{(l_{11}, l_{22}) \in S_f} F(l_{11}, l_{22}), \]
which follows from
\[
S_f \equiv (S_{f_0} \odot \{(l_1, l_2)\})
\]
which follows by definition of \( S_f \) above
\[
\equiv S_{f_0} \cup \{(l_1, l_2)\}
\]
which follows from \( l_1 \notin S_{f_0} \) and \( l_2 \notin S_{f_0} \) (from above)
\[
\equiv (\psi_{f_0} \circ \psi_r \circ \bigcup_{(l_{11}, l_{22}) \in S_{f_0}} F_0(l_{11}, l_{22})) \cup \{(l_1, l_2)\}
\]
which follows from \( S_{f_0} = \ldots \) above
\[
\equiv (\psi_{f_0} \circ \psi_r \circ \bigcup_{(l_{11}, l_{22}) \in S_{f_0}} F(l_{11}, l_{22})) \cup \{(l_1, l_2)\}
\]
which follows from the definition of \( F \) above
\[
\equiv (\psi_r \circ \bigcup_{(l_{11}, l_{22}) \in S_f} F(l_{11}, l_{22})) \cup \{(l_1, l_2)\}
\]
which follows from definition of \( S_f \) above
\[
\equiv (\psi_r \circ \bigcup_{(l_{11}, l_{22}) \in S_f} F(l_{11}, l_{22})) \cup \{(l_1, l_2)\}
\]
which follows from the definition of \( \circ \)
\[
\equiv \{(l_1, l_2)\} \odot \psi_r \circ \bigcup_{(l_{11}, l_{22}) \in S_f} F(l_{11}, l_{22})
\]
which follows by the commutativity of \( \circ \).

\[ \text{dom}(\sigma_1[l_1 \leftarrow \gamma_1(v)]) \supseteq S^1_f, \]
which follows from
\[
\text{dom}(\sigma_1[l_1 \leftarrow \gamma_1(v)]) \equiv \text{dom}(\sigma_1) \cup \{l_1\}
\]
\[
\supseteq \bigcup \{(l_1, l_2)\}
\]
which follows from \( \text{dom}(\sigma_1) \supseteq S^1_{f_0} \) (from above)
\[
\equiv (S_{f_0} \odot \{(l_1, l_2)\})
\]
\[ \equiv S^1_f\]
which follows from the definition of \( S_f \) above.

\[ \text{dom}(\sigma_2[l_2 \leftarrow \gamma_2(v)]) \supseteq S^2_f, \]
which follows from
\[
\text{dom}(\sigma_2[l_2 \leftarrow \gamma_2(v)]) \equiv \text{dom}(\sigma_2) \cup \{l_2\}
\]
\[
\supseteq \bigcup \{(l_1, l_2)\}
\]
which follows from \( \text{dom}(\sigma_2) \supseteq S^2_{f_0} \) (from above)
\[
\equiv (S_{f_0} \odot \{(l_1, l_2)\})
\]
\[ \equiv S^2_f\]
which follows from the definition of \( S_f \) above.

\[ \forall (l_{11}, l_{22}) \in S_f, \forall i < k - 1. \ (i, F(l_{11}, l_{22}), \sigma_1[l_1 \leftarrow \gamma_1(v)](l_{11}), \sigma_2[l_2 \leftarrow \gamma_2(v)](l_{22})) \in \mathcal{V}, \]
which we conclude as follows:

Consider arbitrary \( l_{11}, l_{22} \) and \( i \) such that
- \( (l_{11}, l_{22}) \in S_f \) and
- \( i < k - 1. \)

Note that \( S_f = S_{f_0} \cup \{(l_1, l_2)\} \).

Hence, either \( (l_{11}, l_{22}) \in S_{f_0} \) or \( (l_{11}, l_{22}) \in \{(l_1, l_2)\} \).

**Case** \( (l_{11}, l_{22}) \in S_{f_0} \):

Note that \( l_{11} \neq l_1 \) and \( l_{22} \neq l_2 \).

Instantiate \( \forall (l_{11}, l_{22}) \in S_{f_0}, \forall i < k. \ (i, F_0(l_{11}, l_{22}), \sigma_1(l_{11}), \sigma_2(l_{22})) \in \mathcal{V} \) (from above) with \( (l_{11}, l_{22}) \) and \( i \). Note that
• \((l_{11}, l_{22}) \in S_{f0}\) and
• \(i < k\),
  which follows from \(i < k - 1\) (from above).
Hence, \((i, F_0(l_{11}, l_{22}), \sigma_1(l_{11}), \sigma_2(l_{22})) \in V\)
  \[\equiv (i, F_0(l_{11}, l_{22}), \sigma_1[l_1 \leftarrow \gamma_1(v)](l_{11}), \sigma_2[l_2 \leftarrow \gamma_2(v)](l_{22})) \in V,\]
  since \(l_{11} \neq l_1\) and \(l_{22} \neq l_2\)
  \[\equiv (i, F(l_{11}, l_{22}), \sigma_1[l_1 \leftarrow \gamma_1(v)](l_{11}), \sigma_2[l_2 \leftarrow \gamma_2(v)](l_{22})) \in V,\]
  by the definition of \(F\) since \((l_{11}, l_{22}) \in S_{f0}\).
Thus, \((i, F(l_{11}, l_{22}), \sigma_1[l_1 \leftarrow \gamma_1(v)](l_{11}), \sigma_2[l_2 \leftarrow \gamma_2(v)](l_{22})) \in V\) as required.

Case \((l_{11}, l_{22}) \in \{(l_1, l_2)\}:\)
Hence, \(l_{11} = l_1\) and \(l_{22} = l_2\).
We are required to show that
  \( (i, F(l_{11}, l_{22}), \sigma_1[l_1 \leftarrow \gamma_1(v)](l_{11}), \sigma_2[l_2 \leftarrow \gamma_2(v)](l_{22})) \in V \)
  \[\equiv (i, F(l_1, l_2), \sigma_1[l_1 \leftarrow \gamma_1(v)](l_1), \sigma_2[l_2 \leftarrow \gamma_2(v)](l_2)) \in V \]
  \[\equiv (i, F(l_1, l_2, \gamma_1(v), \gamma_2(v)) \in V,\]
  which follows from Lemma B.1 applied to \((k, \psi_{f0}, \gamma_1(v), \gamma_2(v)) \in V\) (from above)
  and \(i < k\) (which follows from \(i < k - 1\)).
Thus, \((i, F(l_{11}, l_{22}), \sigma_1[l_1 \leftarrow \gamma_1(v)](l_{11}), \sigma_2[l_2 \leftarrow \gamma_2(v)](l_{22})) \in V\) as required.
Case (WRITE) write $v_1 \leftarrow v_2$:

Note that $\Gamma = FV(v_1) \cup FV(v_2)$.

Let $\Gamma_{01} = FV(v_1)$ and $\Gamma_{02} = FV(v_2)$. Hence, $\Gamma = \Gamma_{01} \cup \Gamma_{02}$.

Consider arbitrary $k, \psi_T, \gamma_1,$ and $\gamma_2$ such that

- $k \geq 0$, and
- $(k, \psi_T, \gamma_1, \gamma_2) \in G[\Gamma]$. 

Note that we have $\Gamma_{01}, \gamma_{11}, \gamma_{12}, \gamma_{21}, \gamma_{22},$ and $\psi_{T01}, \psi_{T02}$ such that

- $\gamma_1 = \gamma_{11} \cup \gamma_{12}$,
- $\gamma_2 = \gamma_{21} \cup \gamma_{22}$,
- $\psi_T = \psi_{T01} \circ \psi_{T02}$,
- $(k, \psi_{T01}, \gamma_{11}, \gamma_{21}) \in G[\Gamma_{01}]$, and
- $(k, \psi_{T02}, \gamma_{12}, \gamma_{22}) \in G[\Gamma_{02}]$. 

We are required to show that $(k, \psi_T, \gamma_{1}(\text{write } v_1 \leftarrow v_2), \gamma_2(\text{write } v_1 \leftarrow v_2)) \in C$

$\equiv (k, \psi_T, (\text{write } \gamma_{11}(v_1) \leftarrow \gamma_{12}(v_2)), (\text{write } \gamma_{21}(v_1) \leftarrow \gamma_{22}(v_2))) \in C$

$\equiv (k, \psi_T, (\text{write } \gamma_{11}(v_1) \leftarrow \gamma_{12}(v_2)), (\text{write } \gamma_{21}(v_1) \leftarrow \gamma_{22}(v_2))) \in C$. 

Consider arbitrary $j, \sigma_1, \sigma_2, \psi_r, S, v_{f1}, \sigma_{f1}$, and $T_{f1}$ such that

- $j < k$,
- $\sigma_1, \sigma_2 : k (\psi_T \circ \psi_r) \hookrightarrow S$,
- $\sigma_1, \text{write } \gamma_{11}(v_1) \leftarrow \gamma_{12}(v_2) \downarrow^j v_{f1}, \sigma_{f1}, T_{f1}$, and
- $S^1 \cap \text{alloc}(T_{f1}) = \emptyset$. 

Hence, by inspection of the evaluation rules, it follows that

- $\gamma_{11}(v_1) = l_1$,
- $v_{f1} = ()$,
- $\sigma_{f1} = \sigma_1[l_1 \leftarrow \gamma_{12}(v_2)]$,
- $T_{f1} = \text{write } l_1 \leftarrow \gamma_{12}(v_2)$, and
- $j = 1$. 

Consider arbitrary $v_{f2}, \sigma_{f2}$, and $T_{f2}$ such that

- $\sigma_2, \text{write } \gamma_{21}(v_1) \leftarrow \gamma_{22}(v_2) \downarrow v_{f2}, \sigma_{f2}, T_{f2}$, and
- $S^2 \cap \text{alloc}(T_{f2}) = \emptyset$. 

Hence, by inspection of the evaluation rules, it follows that

- $\gamma_{21}(v_1) = l_2$,
- $v_{f2} = ()$,
- $\sigma_{f2} = \sigma_2[l_2 \leftarrow \gamma_{22}(v_2)]$, and
- $T_{f2} = \text{write } l_2 \leftarrow \gamma_{22}(v_2)$. 

Applying the induction hypothesis to $v_1$ (recall $\Gamma_{01} = FV(v_1)$), we conclude that $\Gamma_{01} \vdash v_1 \leq v_1$.

Instantiate this with $k, \psi_{T01}, \gamma_{11},$ and $\gamma_{21}$. Note that

- $k \geq 0$ and
- $(k, \psi_{T01}, \gamma_{11}, \gamma_{21}) \in G[\Gamma_{01}]$, which follows from above.
Hence, \((k, \psi_{T01}, \gamma_{11}(v_1), \gamma_{21}(v_1)) \in C\).

Instantiate the latter with 0, \(\sigma_1, \sigma_2, (\psi_{T02} \circ \psi_r), S, l_1, \sigma_1, \text{ and } \varepsilon\). Note that

- \(0 < k\),
  which follows from \(0 < j\) and \(j < k\),
- \(\sigma_1, \sigma_2 : k \ (\psi_{T01} \circ (\psi_{T02} \circ \psi_r)) \rightarrow S\),
  which follows from \(\sigma_1, \sigma_2 : k \ (\psi_T \circ \psi_r) \rightarrow S\) (from above) and \(\psi_T = \psi_{T01} \circ \psi_{T02}\) (from above),
- \(\sigma_1, \gamma_{11}(v_1) \nvdash 0 l_1, \sigma_1, \varepsilon\),
  which follows by inspection of the evaluation rules since \(\gamma_{11}(v_1) = l_1\), and
- \(S^1 \cap \text{alloc}(\varepsilon) = \emptyset\),
  which follows from \(\text{alloc}(\varepsilon) = \emptyset\).

Hence, we have

\[
\forall v_{21}, \sigma_{21}, T_{21}.
\]
\[
\sigma_2, \gamma_{21}(v_1) \vdash v_{21}, \sigma_{21}, T_{21} \land
S^2 \cap \text{alloc}(T_{21}) = \emptyset \implies
\exists \psi_{f01}, S_{f01}. (k - 0, \psi_{f01}, l_1, v_{21}) \in \mathcal{V} \land
\sigma_1, \sigma_{21} : k - 0 \ (\psi_{f01} \circ (\psi_{T02} \circ \psi_r)) \rightarrow S_{f01} \land
S^2_{f01} \subseteq S^1 \cup \text{alloc}(\varepsilon) \land
S_{f01}^1 \subseteq S^2 \cup \text{alloc}(T_{21})
\]

Instantiate this with \(l_2, \sigma_2, \text{ and } \varepsilon\). Note that

- \(\sigma_2, \gamma_{21}(v_1) \vdash l_2, \sigma_2, \varepsilon\),
  which follows by inspection of the evaluation rules since \(\gamma_{21}(v_1) = l_2\), and
- \(S^2 \cap \text{alloc}(\varepsilon) = \emptyset\),
  which follows from \(\text{alloc}(\varepsilon) = \emptyset\).

Hence, there exist \(\psi_{f01}\) and \(S_{f01}\) such that

- \((k - 0, \psi_{f01}, l_1, l_2) \in \mathcal{V}\),
- \(\sigma_1, \sigma_{21} : k - 0 \ (\psi_{f01} \circ (\psi_{T02} \circ \psi_r)) \rightarrow S_{f01}\),
- \(S^1_{f01} \subseteq S^1 \cup \text{alloc}(\varepsilon)\), and
- \(S^2_{f01} \subseteq S^2 \cup \text{alloc}(\varepsilon)\).

Note that from \((k, \psi_{f01}, l_1, l_2) \in \mathcal{V}\), it follows that

- \(\psi_{f01} = \{(l_1, l_2)\}\).

Applying the induction hypothesis to \(v_2\) (recall \(\Gamma_{02} = FV(v_2)\)), we conclude that \(\Gamma_{02} \vdash v_2 \nleq v_2\).

Instantiate this with \(k, \psi_{T02}, \gamma_{12}, \text{ and } \gamma_{22}\). Note that

- \(k \geq 0\) and
- \((k, \psi_{T02}, \gamma_{12}, \gamma_{22}) \in \Theta[\Gamma_{02}]\),
  which follows from above.

Hence, \((k, \psi_{T02}, \gamma_{12}(v_2), \gamma_{22}(v_2)) \in C\).

Instantiate the latter with 0, \(\sigma_1, \sigma_2, (\psi_{f01} \circ \psi_r), S_{f01}, \gamma_{12}(v_2), \sigma_1, \text{ and } \varepsilon\). Note that

- \(0 < k\),
  which follows from \(0 < j\) and \(j < k\),
- \(\sigma_1, \sigma_2 : k \ (\psi_{f01} \circ (\psi_{T02} \circ \psi_r)) \rightarrow S_{f01}\),
  which follows from \(\sigma_1, \sigma_2 : k \ (\psi_{f01} \circ (\psi_{T02} \circ \psi_r)) \rightarrow S_{f01}\) (from above),

60
• $\sigma_1, \gamma_{12}(v_2) \downarrow^0 \gamma_{12}(v_2), \sigma_1, \varepsilon$,
  which follows by inspection of the evaluation rules since $\gamma_{12}(v_2)$ is a value, and

• $S_{f01}^1 \cap \text{alloc}(\varepsilon) = \emptyset$,
  which follows from $\text{alloc}(\varepsilon) = \emptyset$.

Hence, we have

\[ \forall \nu_{v_2}, \sigma_{v_2}, T_{v_2}, \]
\[ \sigma_2, \gamma_{22}(v_2) \downarrow \nu_{v_2}, \sigma_{v_2}, T_{v_2} \land \]
\[ S_{f01}^1 \cap \text{alloc}(T_{v_2}) = \emptyset \implies \]
\[ \exists \psi_{f02}, S_{f02}, \exists (k - 0, \nu_{f02}, \gamma_{12}(v_2), \nu_{v_2}) \in \mathcal{V} \land \]
\[ \sigma_1, \sigma_2 : k \rightarrow 0 (\psi_{f02} \circ (\psi_{f01} \circ \psi_{r})) \rightsquigarrow S_{f02} \land \]
\[ S_{f02}^1 \subseteq S_{f01}^1 \cup \text{alloc}(\varepsilon) \land \]
\[ S_{f02}^2 \subseteq S_{f01}^2 \cup \text{alloc}(T_{v_2}) \]

Instantiate the latter with $\gamma_{22}(v_2), \sigma_2$, and $\varepsilon$. Note that

• $\sigma_2, \gamma_{22}(v_2) \downarrow \gamma_{22}(v_2), \sigma_2, \varepsilon$,
  which follows by inspection of the evaluation rules since $\gamma_{22}(v_2)$ is a value, and

• $S_{f01}^2 \cap \text{alloc}(\varepsilon) = \emptyset$,
  which follows from $\text{alloc}(\varepsilon) = \emptyset$.

Hence, there exist $\psi_{f02}$ and $S_{f02}$ such that

• $(k - 0, \nu_{f02}, \gamma_{12}(v_2), \gamma_{22}(v_2)) \in \mathcal{V}$,

• $\sigma_1, \sigma_2 : k \rightarrow 0 (\psi_{f02} \circ (\psi_{f01} \circ \psi_{r})) \rightsquigarrow S_{f02}$,

• $S_{f02}^1 \subseteq S_{f01}^1 \cup \text{alloc}(\varepsilon)$, and

• $S_{f02}^2 \subseteq S_{f01}^2 \cup \text{alloc}(\varepsilon)$.

From $\sigma_1, \sigma_2 : k (\psi_{f01} \circ \psi_{f02} \circ \psi_{r}) \rightsquigarrow S_{f02}$ (from above), it follows that there exists $\mathcal{F}_0$ such that

• $S_{f02} \in \text{LocBij}$,

• $\mathcal{F}_0 : S_{f02} \rightarrow \text{LocBij}$,

• $S_{f02} = \psi_{f01} \circ \psi_{f02} \circ \psi_{r} \circ \bigcup (l_{11}, l_{22}) \in S_{f02} \mathcal{F}_0(l_{11}, l_{22})$,

• $\text{dom}(\sigma_1) \supseteq S_{f02}^1$,

• $\text{dom}(\sigma_2) \supseteq S_{f02}^2$, and

• $\forall (l_{11}, l_{22}) \in S_{f02}, \forall i < k$.
  \[ (i, \mathcal{F}_0(l_{11}, l_{22}), \sigma_1(l_{11}), \sigma_2(l_{22})) \in \mathcal{V} \]

Let $\psi_v = \mathcal{F}_0(l_1, l_2)$.

Note that $\sigma_1[l_1 \leftarrow \gamma_{12}(v_2)], \sigma_2[l_2 \leftarrow \gamma_{22}(v_2)] : k \leftarrow j (\psi_{f01} \circ \psi_v \circ \psi_{r}) \rightsquigarrow S_{f02}$,
which we conclude as follows:

Take $\mathcal{F}$ such that $\text{dom}(\mathcal{F}) = S_{f02}$ and

\[ \mathcal{F}(l_{11}, l_{22}) = \left\{ \begin{array}{ll} \psi_{f02} & \text{if } (l_{11}, l_{22}) \in \{(l_1, l_2)\} \\ \mathcal{F}_0(l_{11}, l_{22}) & \text{otherwise} \end{array} \right. \]

We are required to show:

• $S_{f02} \in \text{LocBij}$,
  which follows from above.
• \( F : S_{f02} \to \text{LocBij} \),
  which follows from

  • \( \text{dom}(F) = S_{f02} \),
  which follows from the definition of \( F \),

  • \( \text{rng} = (\psi_{f02} \circ (l_{11}, l_{22}) \in (S_{f02} \setminus \{l_{11}, l_{22}\})) \cdot F_0(l_{11}, l_{22}) \in \text{LocBij} \),
  which follows from \( \text{rng} \subseteq S_{f02} \) (from above) and \( S_{f02} \in \text{LocBij} \).

• \( S_{f02} = (\psi_{f01} \circ \psi_v \circ \psi_r \circ (l_{11}, l_{22}) \in S_{f02} \cdot F(l_{11}, l_{22}) \),
  which follows from

\[
S_{f02} \equiv \psi_{f01} \circ \psi_v \circ \psi_r \circ (l_{11}, l_{22}) \in S_{f02} \cdot F_0(l_{11}, l_{22})
\]
which follows from \( S_{f02} = \ldots \) above
\[
\equiv \{(l_{11}, l_{22}) \}\circ \psi_{f02} \circ \psi_v \circ \psi_r \circ (l_{11}, l_{22}) \in S_{f02} \cdot F_0(l_{11}, l_{22})
\]
which follows from \( \psi_{f01} = \{(l_{11}, l_{22})\} \) (from above)
\[
\equiv \{(l_{11}, l_{22}) \}\circ \psi_{f02} \circ \psi_v \circ \psi_r \circ (l_{11}, l_{22}) \in S_{f02} \cdot F_0(l_{11}, l_{22}) \circ F_0(l_{11}, l_{22})
\]
which follows since \( (l_{11}, l_{22}) \in S_{f02} \)
\[
\equiv \{(l_{11}, l_{22}) \}\circ \psi_{f02} \circ \psi_v \circ \psi_r \circ (l_{11}, l_{22}) \in S_{f02} \cdot F_0(l_{11}, l_{22}) \circ \psi_v
\]
which follows from \( \psi_v = F_0(l_{11}, l_{22}) \) (from above)
\[
\equiv \{(l_{11}, l_{22}) \}\circ \psi_{f02} \circ \psi_v \circ \psi_r \circ (l_{11}, l_{22}) \in S_{f02} \cdot F(l_{11}, l_{22}) \circ \psi_v
\]
which follows from definition of \( F \) above
\[
\equiv \{(l_{11}, l_{22}) \}\circ \psi_v \circ \psi_r \circ (l_{11}, l_{22}) \in S_{f02} \cdot F(l_{11}, l_{22})
\]
which follows from the definition of \( F \), since \( F(l_{11}, l_{22}) = \psi_{f02} \)
\[
\equiv \psi_{f01} \circ \psi_v \circ \psi_r \circ (l_{11}, l_{22}) \in S_{f02} \cdot F(l_{11}, l_{22})
\]
which follows from \( \psi_{f01} = \{(l_{11}, l_{22})\} \) (from above).

• \( \text{dom}(\sigma_1[l_1 \leftarrow \gamma_{12}(v_2)] \supseteq S_{f02}^1 \),
  which follows from

\[
\text{dom}(\sigma_1[l_1 \leftarrow \gamma_{12}(v_2)]) \equiv \text{dom}(\sigma_1) \cup \{l_1\}
\]
\[
\supseteq S_{f02}^1 \cup \{l_1\},
\]
  which follows from \( \text{dom}(\sigma_1) \supseteq S_{f02}^1 \) (from above)
\[
\equiv (S_{f02} \circ \{(l_1, l_2)\})^1
\]
\[
\equiv S_{f02}^1
\]
  which follows from \( (l_1, l_2) \in S_{f02} \) (from above).

• \( \text{dom}(\sigma_2[l_2 \leftarrow \gamma_{22}(v_2)] \supseteq S_{f02}^2 \),
  which follows from

\[
\text{dom}(\sigma_2[l_2 \leftarrow \gamma_{22}(v_2)]) \equiv \text{dom}(\sigma_2) \cup \{l_2\}
\]
\[
\supseteq S_{f02}^2 \cup \{l_2\},
\]
  which follows from \( \text{dom}(\sigma_2) \supseteq S_{f02}^2 \) (from above)
\[
\equiv (S_{f02} \circ \{(l_1, l_2)\})^2
\]
\[
\equiv S_{f02}^2
\]
  which follows from \( (l_1, l_2) \in S_{f02} \) (from above).

\[
\forall (l_{11}, l_{22}) \in S_{f02}, \forall i < j \cdot (i, F(l_{11}, l_{22}), \sigma_1[l_1 \leftarrow \gamma_{12}(v_2)](l_{11}), \sigma_2[l_2 \leftarrow \gamma_{22}(v_2)](l_{22})) \in \mathcal{V},
\]
which we conclude as follows:

Consider arbitrary \( l_{11}, l_{22} \) and \( i \) such that

• \( (l_{11}, l_{22}) \in S_{f02} \) and
• \( i < k – j \).
Hence, either \((l_{11}, l_{22}) \in \{(l_1, l_2)\}\) or \((l_{11}, l_{22}) \in (S_{f02} \setminus \{(l_1, l_2)\})\).

**Case** \((l_{11}, l_{22}) \in \{(l_1, l_2)\}):

Hence, \(l_{11} = l_1\) and \(l_{22} = l_2\).

We are required to show that
\[
(i, F(l_{11}, l_{22}), \sigma_1[l_1 \leftarrow \gamma_{12}(v_2)](l_{11}), \sigma_2[l_2 \leftarrow \gamma_{22}(v_2)](l_{22})) \in V
\]
\[
\equiv (i, F(l_1, l_2), \sigma_1[l_1 \leftarrow \gamma_{12}(v_2)](l_1), \sigma_2[l_2 \leftarrow \gamma_{22}(v_2)](l_2)) \in V
\]
\[
\equiv (i, F(l_1, l_2), \gamma_{12}(v_2), \gamma_{22}(v_2)) \in V
\]
\[
\equiv (i, \psi_{f02}, \gamma_{12}(v_2), \gamma_{22}(v_2)) \in V,
\]
which follows from Lemma B.1 applied to \((k, \psi_{f0}, \gamma_{12}(v_2), \gamma_{22}(v_2)) \in V\) (from above) and \(i < k\) (which follows from \(i < k - j = k - 1\)).

Thus, \((i, F(l_{11}, l_{22}), \sigma_1[l_1 \leftarrow \gamma_{12}(v_2)](l_{11}), \sigma_2[l_2 \leftarrow \gamma_{22}(v_2)](l_{22})) \in V\) as required.

**Case** \((l_{11}, l_{22}) \in (S_{f0} \setminus \{(l_1, l_2)\})\):

Hence, \(l_{11} \neq l_1\) and \(l_{22} \neq l_2\).

Instantiate \(\forall (l_{11}, l_{22}) \in S_{f02}, \forall i < k. (i, F_0(l_{11}, l_{22}), \sigma_1(l_{11}), \sigma_2(l_{22})) \in V\) (from above) with \((l_{11}, l_{22})\) and \(i\). Note that
1. \((l_{11}, l_{22}) \in S_{f02}\) and
2. \(i < k\),
which follows from \(i < k - j = k - 1\) (from above).

Hence, \((i, F_0(l_{11}, l_{22}), \sigma_1(l_{11}), \sigma_2(l_{22})) \in V\)
\[
\equiv (i, F_0(l_1, l_2), \sigma_1[l_1 \leftarrow \gamma_{12}(v_2)](l_1), \sigma_2[l_2 \leftarrow \gamma_{22}(v_2)](l_2)) \in V,
\]
since \(l_{11} \neq l_1\) and \(l_{22} \neq l_2\)
\[
\equiv (i, F(l_1, l_2), \sigma_1[l_1 \leftarrow \gamma_{12}(v_2)](l_1), \sigma_2[l_2 \leftarrow \gamma_{22}(v_2)](l_2)) \in V,
\]
by the definition of \(F\) since \((l_{11}, l_{22}) \in (S_{f0} \setminus \{(l_1, l_2)\})\).

Thus, \((i, F(l_{11}, l_{22}), \sigma_1[l_1 \leftarrow \gamma_{12}(v_2)](l_{11}), \sigma_2[l_2 \leftarrow \gamma_{22}(v_2)](l_{22})) \in V\) as required.

From Lemma B.4 applied to \(\sigma[l_1 \leftarrow \gamma_{12}(v_2)], \sigma_2[l_2 \leftarrow \gamma_{22}(v_2)];_{k-j} (\psi_r \circ (\psi_{f1} \circ \psi_{v})) \rightsquigarrow S_{f02}\) (from above), it follows that there exists \(S_f\) such that

1. \(\sigma_1[l_1 \leftarrow \gamma_{12}(v_2)], \sigma_2[l_2 \leftarrow \gamma_{22}(v_2)];_{k-j} \psi_r \rightsquigarrow S_f\) and
2. \(S_f \subseteq S_{f02}\).

Take \(\psi_f = \{\}\). Take \(S_f = S_f\).

We are required to show:

1. \((k - j, \psi_f, v_{f1}, v_{f2}) \in V\)
\[
\equiv (k - 1, \{\}, \{\}, \{\}) \in V,
\]
which follows from the definition of \(V\),
2. \(\sigma_{f1}, \sigma_{f2} ;_{k-j} (\psi_f \circ \psi_r) \rightsquigarrow S_f\)
\[
\equiv \sigma[l_1 \leftarrow \gamma_{12}(v_2)], \sigma_2[l_2 \leftarrow \gamma_{22}(v_2)] ;_{k-j} (\{\} \circ \psi_r) \rightsquigarrow S_{f02},
\]
\[
\equiv \sigma[l_1 \leftarrow \gamma_{12}(v_2)], \sigma_2[l_2 \leftarrow \gamma_{22}(v_2)] ;_{k-j} \psi_r \rightsquigarrow S_{f02},
\]
which follows from above.
3. \(S^1 \subseteq S^1 \cup \text{alloc}(T_{f1})\)
\[
\equiv S^1 \subseteq S^1 \cup \text{alloc}(\text{write} l_1 \leftarrow \gamma_{12}(v_2)),
\]
\[
\equiv S^1 \subseteq S^1 \cup \{\}, \text{since \text{alloc}(\text{write} l_1 \leftarrow \gamma_{12}(v_2)) = \{\}}
\]
\[
\equiv S^1 \subseteq S^1,
\]
which follows from \(S_f \subseteq S_{f02}\) (from above) and \(S^1_{f02} \subseteq S^1_{f01} \subseteq S^1\) (from above), and
\[ S_2^f \subseteq S^2 \cup \text{alloc}(T_{f2}) \]
\[ \equiv S_2^f \subseteq S^2 \cup \text{alloc}(\text{write } l_2 \leftarrow \gamma_{22}(v_2)) \]
\[ \equiv S_2^f \subseteq S^2 \cup \{\} \text{, (since alloc(\text{write } l_2 \leftarrow \gamma_{22}(v_2)) = \{\})} \]
\[ \equiv S_2^f \subseteq S^2 \]
which follows from \( S_f \subseteq S_{f02} \) (from above) and \( S_{f02}^2 \subseteq S_{f01}^2 \subseteq S^2 \) (from above).
Case (READ) \textit{read} \(v\) \textit{as} \(x\) \textit{in} \(e\):

Note that \(\Gamma = FV(\text{read} v \text{ as } x \text{ in } e) = FV(v) \cup (FV(e) \setminus \{x\})\).

Let \(\Gamma_{01} = FV(v)\) and \(\Gamma_{02} = (FV(e) \setminus \{x\})\). Hence, \(\Gamma = \Gamma_{01} \cup \Gamma_{02}\).

Consider arbitrary \(k, \psi, \gamma_1,\) and \(\gamma_2\) such that

- \(k \geq 0,\) and
- \((k, \psi, \gamma_1, \gamma_2) \in \mathcal{G}[\Gamma]\).

Note that we have \(\gamma_1, \gamma_2, \gamma_{21}, \gamma_{22},\) and \(\psi_{01},\) \(\psi_{02}\) such that

- \(\gamma_1 = \gamma_{11} \cup \gamma_{12},\)
- \(\gamma_2 = \gamma_{21} \cup \gamma_{22},\)
- \(\psi = \psi_{01} \circ \psi_{02},\)
- \((k, \psi_{01}, \gamma_{11}, \gamma_{21}) \in \mathcal{G}[\Gamma_{01}],\) and
- \((k, \psi_{02}, \gamma_{12}, \gamma_{22}) \in \mathcal{G}[\Gamma_{02}].\)

We are required to show that \((k, \psi, \gamma_1(\text{read} v \text{ as } x \text{ in } e), \gamma_2(\text{read} v \text{ as } x \text{ in } e)) \in C\)
\[
\equiv (k, \psi, \text{read} \gamma_1(v) \text{ as } x \text{ in } \gamma_1(e), \text{read} \gamma_2(v) \text{ as } x \text{ in } \gamma_2(e)) \in C
\]
\[
\equiv (k, \psi, \text{read} \gamma_{11}(v) \text{ as } x \text{ in } \gamma_{12}(e), \text{read} \gamma_{21}(v) \text{ as } x \text{ in } \gamma_{22}(e)) \in C.
\]

Consider arbitrary \(j, \sigma_1, \sigma_2, \psi_r, \mathcal{S}, v_{f_1}, \sigma_{f_1},\) and \(T_{f_1}\) such that

- \(j < k,\)
- \(\sigma_1, \sigma_2 \vdash_k (\psi \circ \psi_r) \rightarrow \mathcal{S},\)
- \(\sigma_1, \text{read} \gamma_{11}(v) \text{ as } x \text{ in } \gamma_{12}(e) \downarrow^j v_{f_1}, \sigma_{f_1}, T_{f_1},\) and
- \(S^1 \cap \text{alloc}(T_{f_1}) = \emptyset.\)

Hence, by inspection of the evaluation rules, it follows that there exist \(j_1, l_1,\) and \(T_1\) such that

- \(\gamma_{11}(v) = l_1,\)
- \(v_{f_1} = (),\)
- \(T_{f_1} = \text{read}_{l_1 \rightarrow x = \sigma_1(l_1), \gamma_{12}(e)} T_1,\)
- \(\sigma_1, \gamma_{12}(e)[\sigma_1(l_1)/x] \downarrow v_{f_1}, \sigma_{f_1}, T_1,\) and
- \(j = j_1 + 1.\)

Note that \(S^1 \cap \text{alloc}(T_1) = \emptyset,\) which follows from

\[
S^1 \cap \text{alloc}(T_{f_1}) = \emptyset \quad \text{which follows from above}
\]
\[
\equiv S^1 \cap \text{alloc(\text{read}_{l_1 \rightarrow x = \sigma_1(l_1), \gamma_{12}(e)} T_1)} = \emptyset 
\]
\[
\quad \text{which follows from } T_{f_1} = \text{read}_{l_1 \rightarrow x = \sigma_1(l_1), \gamma_{12}(e)} T_1 \text{ (from above)}
\]
\[
\equiv S^1 \cap \text{alloc(T}_1) = \emptyset \quad \text{which follows from the definition of alloc.}
\]

Consider arbitrary \(v_{f_2}, \sigma_{f_2},\) and \(T_{f_2}\) such that

- \(\sigma_2, \text{read} \gamma_{21}(v) \text{ as } x \text{ in } \gamma_{22}(e) \downarrow v_{f_2}, \sigma_{f_2}, T_{f_2},\)
- \(S^2 \cap \text{alloc}(T_{f_2}) = \emptyset.\)

Hence, by inspection of the evaluation rules, it follows that there exist \(l_2\) and \(T_2\) such that

- \(\gamma_{21}(v) = l_2,\)
- \(v_{f_2} = (),\)
- \(T_{f_2} = \text{read}_{l_2 \rightarrow x = \sigma_2(l_2), \gamma_{22}(e)} T_2,\) and
Applying the induction hypothesis to $v$
Hence, there exist $\psi_l$
Hence, we have
Instantiate the latter with 0,
Hence, $\kappa, \psi$
Note that
 Instantiate this with $k$, $\psi_{T_01}$, $\gamma_{11}$, and $\gamma_{21}$. Note that

- $k \geq 0$ and
- $(k,\psi_{T_01},\gamma_{11},\gamma_{21}) \in \mathcal{G}[[\Gamma_01]]$, which follows from above.

Hence, $(k,\psi_{T_01},\gamma_{11}(v),\gamma_{21}(v)) \in \mathcal{C}$.

Instantiate the latter with 0, $\sigma_1$, $\sigma_2$, $(\psi_{T_02} \odot \psi_r)$, $S$, $l_1$, $\sigma_1$, and $\varepsilon$.

Note that
- $0 < k$, which follows from $j = j_1 + 1$ and $j < k$,
- $\sigma_1,\sigma_2, k : (\psi_{T_01} \odot (\psi_{T_02} \odot \psi_r)) \rightsquigarrow S$, which follows from $\sigma_1,\sigma_2, k : (\psi_T \odot \psi_r) \rightsquigarrow S$ (from above) and $\psi_T = \psi_{T_01} \odot \psi_{T_02}$ (from above),
- $\sigma_1,\gamma_{11}(v) \not\in l_1,\sigma_1,\varepsilon$, which follows by inspection of the evaluation rules since $\gamma_{11}(v) = l_1$, and
- $S^1 \cap \text{alloc}(\varepsilon) = \emptyset$, which follows from $\text{alloc}(\varepsilon) = \emptyset$.

Hence, we have

\[
\forall v_{21}, \sigma_{21}, T_{21}. \\
\sigma_2, \gamma_{21}(V) \Downarrow v_{21}, \sigma_{21}, T_{21} \land \\
S^2 \cap \text{alloc}(T_{21}) = \emptyset \implies \\
\exists \psi_{f_01}, S_{f_01}, (k - 0, \psi_{f_01}, l_1, v_{21}) \in \mathcal{V} \land \\
\sigma_1, \sigma_{21} : k - 0 (\psi_{f_01} \odot (\psi_{T_02} \odot \psi_r)) \rightsquigarrow S_{f_01} \land \\
S^1_{f_01} \subseteq S^1 \cup \text{alloc}(\varepsilon) \land \\
S^2_{f_01} \subseteq S^2 \cup \text{alloc}(T_{21})
\]

Instantiate the latter with $l_2, \sigma_2, \varepsilon$. Note that
- $\sigma_2, \gamma_{21}(v) \Downarrow l_2, \sigma_2, \varepsilon$, which follows by inspection of the evaluation rules since $\gamma_{21}(v) = l_2$, and
- $S^2 \cap \text{alloc}(\varepsilon) = \emptyset$, which follows from $\text{alloc}(\varepsilon) = \emptyset$.

Hence, there exist $\psi_{f_01}$ and $S_{f_01}$ such that

- $(k - 0, \psi_{f_01}, l_1, l_2) \in \mathcal{V}$,
- $\sigma_1, \sigma_2 : k - 0 (\psi_{f_01} \odot (\psi_{T_02} \odot \psi_r)) \rightsquigarrow S_{f_01}$,
- $S^1_{f_01} \subseteq S^1 \cup \text{alloc}(\varepsilon)$, and
- $S^2_{f_01} \subseteq S^2 \cup \text{alloc}(\varepsilon)$.
Note that
\[ \psi_{f01} = \{(l_1, l_2)\}, \]
which follows from \((k, \psi_{f01}, l_1, l_2) \in \mathcal{V}\) (from above) and the definition of \(\mathcal{V}\).

From \(\sigma_1, \sigma_2 : (\psi_{f01} \circ \psi_{r02} \circ \psi_r) \rightsquigarrow \mathcal{S}_{f01}\) (from above), it follows that there exists \(\mathcal{F}_0\) such that

- \(\mathcal{S}_{f01} \in \text{LocBij}\),
- \(\mathcal{F}_0 : \mathcal{S}_{f01} \rightarrow \text{LocBij}\),
- \(\mathcal{S}_{f01} = \psi_{f01} \circ \psi_{r02} \circ \psi_r \circ \bigcirc_{(l_1, l_2) \in \mathcal{S}_{f01}} \mathcal{F}_0(l_1, l_2)\),
- \(\text{dom}(\sigma_1) \supseteq \mathcal{S}_{f01}^1\),
- \(\text{dom}(\sigma_2) \supseteq \mathcal{S}_{f01}^2\), and
- \(\forall (l_1, l_2) \in \mathcal{S}_{f01}, \forall i < k, (i, \mathcal{F}_0(l_1, l_2), \sigma_1(l_1), \sigma_2(l_2)) \in \mathcal{V}\).

Let \(\psi_v = \mathcal{F}_0(l_1, l_2)\).

Instantiate \(\forall (l_1, l_2) \in \mathcal{S}_{f01}, \forall i < k, (i, \mathcal{F}_0(l_1, l_2), \sigma_1(l_1), \sigma_2(l_2)) \in \mathcal{V}\) (from above) with \((l_1, l_2)\) and \(k - 1\). Note that

- \((l_1, l_2) \in \mathcal{S}_{f01}\),
  which follows from
  - \(\{(l_1, l_2)\} \equiv \psi_{f01}\),
    which follows from above, and
  - \(\psi_{f01} \subseteq \mathcal{S}_{f01}\),
    which follows from \(\mathcal{S}_{f01} = \psi_{f01} \circ \ldots\) (from above).

  and

- \(k - 1 < k\).

Hence, \((k - 1, \mathcal{F}_0(l_1, l_2), \sigma_1(l_1), \sigma_2(l_2)) \in \mathcal{V}\)

\[ \equiv (k - 1, \psi_v, \sigma_1(l_1), \sigma_2(l_2)) \in \mathcal{V}.\]

Note that \(\sigma_1, \sigma_2 : (\psi_{f01} \circ \psi_{r02} \circ \psi_v \circ \psi_r) \rightsquigarrow \mathcal{S}_{f01}\),
which we conclude as follows:

Take \(\mathcal{F} = \mathcal{F}_0\).

We are required to show:

- \(\mathcal{S}_{f01} \in \text{LocBij}\),
  which follows from above.
- \(\mathcal{F} : \mathcal{S}_{f01} \rightarrow \text{LocBij}\),
  which follows from \(\mathcal{F} = \mathcal{F}_0 : \mathcal{S}_{f01} \rightarrow \text{LocBij}\).
\[ S_{f01} = (\psi_{f01} \circ \psi_{T02} \circ \psi_v \circ \psi_r \circ \bigotimes_{(l1, l2) \in S_{f01}} F(l1, l2), \]
which follows from
\[ S_{f01} = \psi_{f01} \circ \psi_{T02} \circ \psi_v \circ \psi_r \circ \bigotimes_{(l1, l2) \in S_{f01}} F_0(l1, l2) \]
which follows since \((l1, l2) \in S_{f01}\)
\[ S_{f01} = \psi_{f01} \circ \psi_{T02} \circ \psi_v \circ \psi_r \circ \bigotimes_{(l1, l2) \in S_{f01}} F_0(l1, l2) \]
which follows from \(\psi_v = F_0(l1, l2)\) (from above)
\[ S_{f01} = \psi_{f01} \circ \psi_{T02} \circ \psi_v \circ \psi_r \circ \bigotimes_{(l1, l2) \in S_{f01}} F(l1, l2) \]
which follows from definition of \(F\) above
\[ S_{f01} = \psi_{f01} \circ \psi_{T02} \circ \psi_v \circ \psi_r \circ \bigotimes_{(l1, l2) \in S_{f01}} F(l1, l2). \]

- \(\text{dom}(\sigma_1) \supseteq S_{f01}^1\),
which follows from above, and
- \(\text{dom}(\sigma_2) \supseteq S_{f01}^2\),
which follows from above.
- \(\forall (l1, l2) \in S_{f01}, \forall i < k. (i, F(l1, l2), \sigma_1(l1), \sigma_2(l2)) \in \mathcal{V}\),
which follows from
- \(\forall (l1, l2) \in S_{f01}, \forall i < k. (i, F_0(l1, l2), \sigma_1(l1), \sigma_2(l2)) \in \mathcal{V}\) (since \(F = F_0\)),
which follows from above.

Note that from Lemma B.4 applied to \(\sigma_1, \sigma_2 \upharpoonright_k ((\psi_{T02} \circ \psi_v \circ \psi_r) \circ \psi_{f01}) \rightarrow S_{f01}\) (from above), it follows that there exists \(S_{f02}\) such that

- \(\sigma_1, \sigma_2 \upharpoonright_k (\psi_{T02} \circ \psi_v \circ \psi_r) \rightarrow S_{f02}\), and
- \(S_{f02} \subseteq S_{f01}\).

Applying the induction hypothesis to \(e\) (noting that \(FV(e) = \Gamma_{02} \cup \{x\}\)), we conclude that \(\Gamma_{02}, x \vdash e \rightarrow e\).

Instantiate this with \(k - 1, (\psi_{T02} \circ \psi_v), \gamma_{12}[x \mapsto \sigma_1(l1)], \) and \(\gamma_{22}[x \mapsto \sigma_2(l2)].\) Note that

- \(k - 1 \geq 0,\)
which follows from \(j_1 + 1 = j < k\) (from above), and
- \((k - 1, \psi_{T02} \circ \psi_v, \gamma_{12}[x \mapsto \sigma_1(l2)], \gamma_{22}[x \mapsto \sigma_2(l2)]) \in \mathcal{G}[[\Gamma_{02}, x]],\)
which follows from
- \((k - 1, \psi_{T02}, \gamma_{12}, \gamma_{22}) \in \mathcal{G}[[\Gamma_{02}]],\)
which follows from Lemma B.2 applied to \((k, \psi_{T02}, \gamma_{12}, \gamma_{22}) \in \mathcal{G}[[\Gamma_{02}]]\) (from above) and \(k - 1 < k\), and
- \((k - 1, \psi_v, \sigma_1(l1), \sigma_2(l2)) \in \mathcal{V},\)
which follows from above.

Hence, \((k - 1, \psi_{T02} \circ \psi_v, \gamma_{12}[x \mapsto \sigma_1(l1)](e), \gamma_{22}[x \mapsto \sigma_2(l2)](e)) \in \mathcal{C}\)
\(\equiv (k - 1, \psi_{T02} \circ \psi_v, \gamma_{12}(e)[\sigma_1(l1)/x], \gamma_{22}(e)[\sigma_2(l2)/x]) \in \mathcal{C}.\)

Instantiate the latter with \(j_1, \sigma_1, \sigma_2, \psi_r, S_{f02}, (\), \sigma_f1, and \(T_1\). Note that

- \(j_1 < k - 1,\)
which follows from \(j_1 + 1 = j\) (from above) and \(j < k\) (from above),

68
Hence, we have

\[ \forall v, \sigma_{f1}, T_{22}, \sigma_{f2} : \gamma_{22}(e)[\sigma_{f2}(l_2)/x] \vdash v_{f2}, \sigma_{f2}; T_{22} \land \\
S_{f02} \cap \text{alloc}(T_{22}) = \emptyset \implies \exists \psi_f, S_f. (k - 1 - j_1, \psi_f, (), v_{f2}) \in \mathcal{V} \land \\
\sigma_{f1}, \sigma_{f2} : k - 1 - j_1 (\psi_f \land \psi_r) \rightsquigarrow S_f \land \\
S_f^1 \subseteq S_{f02} \cup \text{alloc}(T_1) \land \\
S_f^2 \subseteq S_{f02} \cup \text{alloc}(T_{22}) \]

Instantiate the latter with ( ), \( \sigma_{f2} \), and \( T_2 \). Note that

- \( \sigma_{f2}, \gamma_{22}(e)[\sigma_{f2}(l_2)/x] \vdash ( ), \sigma_{f2}, T_2 \),

which follows from above, and

- \( S_{f02} \cap \text{alloc}(T_2) = \emptyset \),

which follows from

- \( S_{f02} \subseteq S_{f01} \subseteq S^1 \)

which follows from \( S_{f02} \subseteq S_{f01} \) (from above) and \( S_{f01}^1 \subseteq S^1 \) (from above), and

- \( S^1 \cap \text{alloc}(T_2) = \emptyset \) (from above).

Hence, there exist \( \psi_f \) and \( S_f \) such that

- \( (k - 1 - j_1, \psi_f, (), ()) \in \mathcal{V} \),

- \( \sigma_{f1}, \sigma_{f2} : k - 1 - j_1 (\psi_f \land \psi_r) \rightsquigarrow S_f \),

- \( S_f^1 \subseteq S_{f02} \cup \text{alloc}(T_1) \), and

- \( S_f^2 \subseteq S_{f02} \cup \text{alloc}(T_2) \).

Note that \( \psi_f = \{ \} \), which follows from \( (k - 1 - j_1, \psi_f, (), ()) \) \in \( \mathcal{V} \) above.

Hence, we have \( \sigma_{f1}, \sigma_{f2} : k - j_1 - 1 (\{ \} \land \psi_r) \rightsquigarrow S_f \) from above.

Take \( \psi_f = \{ \} \). Take \( S_f = S_f \).

We are required to show:

- \( (k - j, \psi_f, v_{f1}, v_{f2}) \in \mathcal{V} \),

  \[ \equiv (k - j, \{ \}, (), ()) \in \mathcal{V}, \]

  which follows from the definition of \( \mathcal{V} \),

- \( \sigma_{f1}, \sigma_{f2} : k - j (\psi_f \land \psi_r) \rightsquigarrow S_f \)

  \[ \equiv \sigma_{f1}, \sigma_{f2} : k - j (\{ \} \land \psi_r) \rightsquigarrow S_f, \]

  \[ \equiv \sigma_{f1}, \sigma_{f2} : k - j_1 - 1 (\{ \} \land \psi_r) \rightsquigarrow S_f, \] (since \( j = j_1 + 1 \))

  which follows from above,
\begin{itemize}
  \item $S_f^1 \subseteq S^1 \cup \text{alloc}(T_f^1)$
    which follows from
    
    $S_f^1 \subseteq S_f^{02} \cup \text{alloc}(T_1)$
    which follows from above
    
    $\subseteq S_f^{01} \cup \text{alloc}(T_1)$
    which follows from $S_f^{02} \subseteq S_f^{01}$ (from above)
    
    $\subseteq S^1 \cup \text{alloc}(T_1)$
    which follows from $S_f^{01} \subseteq S^1$ (from above)
    
    $\equiv S^1 \cup \text{alloc} \left( \text{read}_{l_1 \rightarrow x = \sigma_{1(l_1), \gamma_{12}(e)}} T_1 \right)$
    which follows from the definition of alloc
    
    $\equiv S^1 \cup \text{alloc}(T_f^1)$
    which follows from $T_f^1 = \text{read}_{l_1 \rightarrow x = \sigma_{1(l_1), \gamma_{12}(e)}} T_1$ (from above).

  \item $S_f^2 \subseteq S^2 \cup \text{alloc}(T_f^2)$
    which follows from
    
    $S_f^2 \subseteq S_f^{02} \cup \text{alloc}(T_2)$
    which follows from above
    
    $\subseteq S_f^{01} \cup \text{alloc}(T_2)$
    which follows from $S_f^{02} \subseteq S_f^{01}$ (from above)
    
    $\subseteq S^2 \cup \text{alloc}(T_2)$
    which follows from $S_f^{01} \subseteq S^2$ (from above)
    
    $\equiv S^2 \cup \text{alloc} \left( \text{read}_{l_2 \rightarrow x = \sigma_{2(l_2), \gamma_{22}(e)}} T_2 \right)$
    which follows from the definition of alloc
    
    $\equiv S^2 \cup \text{alloc}(T_f^2)$
    which follows from $T_f^2 = \text{read}_{l_2 \rightarrow x = \sigma_{2(l_2), \gamma_{22}(e)}} T_2$ (from above).
\end{itemize}
Case (MEMO) memo $e$:

Note that $\Gamma = FV(memo\ e)$.

Consider arbitrary $k, \psi_T, \gamma_1$, and $\gamma_2$ such that

1. $k \geq 0$, and
2. $(k, \psi_T, \gamma_1, \gamma_2) \in G[\Gamma]$.

We are required to show that $(k, \psi_T, \gamma_1(memo\ e), \gamma_2(memo\ e)) \in C$

\[ \equiv (k, \psi_T, memo\ \gamma_1(e), memo\ \gamma_2(e)) \in C. \]

Consider arbitrary $j, \sigma_1, \sigma_2, \psi_r, S, v_{f_1}, \sigma_{f_1}$, and $T_{f_1}$ such that

1. $j < k$,
2. $\sigma_1, \sigma_2, \psi : k (\psi_T \odot \psi_r) \rightsquigarrow S$,
3. $\sigma_1, memo\ \gamma_1(e) \downarrow^j v_{f_1}, \sigma_{f_1}, T_{f_1}$, and
4. $S^1 \cap alloc(T_{f_1}) = \emptyset$.

Hence, by inspection of the evaluation rules, it follows that there exist $j_1, j_2, \sigma_{01}, \sigma_{f01}$, and $T_{01}$, such that

1. $\sigma_{01}, \gamma_1(e) \downarrow^{j_1} v_{f_1}, \sigma_{f01}, T_{01}$,
2. $\sigma_1, T_{01} \rightsquigarrow^{j_2} \sigma_{f1}, T_{f1}$, and
3. $j = j_1 + j_2$.

From Lemma 4.4 (Memo Elimination) applied to $\gamma_1(e)$ (a closed term, possibly with free locations), noting $\sigma_{01}, \gamma_1(e) \downarrow^{j_2} v_{f_1}, \sigma_{f01}, T_{01}$ and $\sigma_1, T_{01} \rightsquigarrow^{j_2} \sigma_{f1}, T_{f1}$, it follows that there exists $j'$ such that

1. $\sigma_1, \gamma_1(e) \downarrow^{j'} v_{f1}, \sigma_{f1}, T_{f1}$ and
2. $j' \leq j_1 + j_2$.

Consider arbitrary $v_{f2}, \sigma_{f2}$, and $T_{f2}$ such that

1. $\sigma_2, memo\ \gamma_2(e) \downarrow v_{f2}, \sigma_{f2}, T_{f2}$, and
2. $S^2 \cap alloc(T_{f2}) = \emptyset$.

Hence, by inspection of the evaluation rules, it follows that there exist $\sigma_{02}, \sigma_{f02}, T_{02}$, such that

1. $\sigma_{02}, \gamma_2(e) \downarrow v_{f2}, \sigma_{f02}, T_{02}$, and
2. $\sigma_2, T_{02} \rightsquigarrow \sigma_{f2}, T_{f2}$.

From Lemma 4.4 (Memo Elimination) applied to $\gamma_2(e)$ (a closed term, possibly with free locations), noting $\sigma_{02}, \gamma_2(e) \downarrow v_{f2}, \sigma_{f02}, T_{02}$ and $\sigma_2, T_{02} \rightsquigarrow \sigma_{f2}, T_{f2}$, it follows that

1. $\sigma_2, \gamma_2(e) \downarrow v_{f2}, \sigma_{f2}, T_{f2}$.

Applying the induction hypothesis to $e$ (noting that $\Gamma = FV(e)$), we conclude that $\Gamma \vdash e \not\approx e$.

Instantiate this with $k, \psi_T, \gamma_1$, and $\gamma_2$. Note that

1. $k \geq 0$ and
2. $(k, \psi_T, \gamma_1, \gamma_2) \in G[\Gamma]$,

which follows from above.

Hence, $(k, \psi_T, \gamma_1(e), \gamma_2(e)) \in C$.

Instantiate the latter with $j', \sigma_1, \sigma_2, \psi_r, S, v_{f1}, \sigma_{f1}$, and $T_{f1}$. Note that
• $j' < k$,
  which follows from $j' \leq j_1 + j_2 = j < k$ (from above),
• $\sigma_1, \sigma_2 : k (\psi_T \otimes \psi_r) \leadsto S$,
  which follows from above,
• $\sigma_1, \gamma_1 (e) : j' v_{f1}, \sigma_{f1}, T_{f1}$,
  which follows from above, and
• $S^1 \cap \text{alloc}(T_{f1}) = \emptyset$,
  which follows from above.

Hence, we have

$$\forall v_{f2}, \sigma_{f2}, T_{f2}.
\sigma_2, \gamma_2 (e) \downarrow v_{f2}, \sigma_{f2}, T_{f2} \wedge
S^2 \cap \text{alloc}(T_{f2}) = \emptyset \implies
\exists \psi_f, S_f, (k - j', \psi_f, v_{f1}, v_{f2}) \in \mathcal{V} \wedge
\sigma_{f1}, \sigma_{f2} : k - j' (\psi_f \otimes \psi_r) \leadsto S_f \wedge
S_f^1 \subseteq S^1 \cup \text{alloc}(T_{f1}) \wedge
S_f^2 \subseteq S^2 \cup \text{alloc}(T_{f2})$$

Instantiate the latter with $v_{f2}, \sigma_{f2},$ and $T_{f2}$. Note that

• $\sigma_2, \gamma_2 (e) \downarrow v_{f2}, \sigma_{f2}, T_{f2}$ (from above), and
• $S^2 \cap \text{alloc}(T_{f2}) = \emptyset$ (from above).

Hence, there exist $\psi_f$ and $S_f$ such that

• $(k - j', \psi_f, v_{f1}, v_{f2}) \in \mathcal{V},$
• $\sigma_{f1}, \sigma_{f2} : k - j' (\psi_f \otimes \psi_r) \leadsto S_f,$
• $S_f^1 \subseteq S^1 \cup \text{alloc}(T_{f1}),$ and
• $S_f^2 \subseteq S^2 \cup \text{alloc}(T_{f2}).$

Take $\psi_f = \psi_f$. Take $S_f = S_f$.

We are required to show:

• $(k - j, \psi_f, v_{f1}, v_{f2}) \in \mathcal{V},$
  which follows from Lemma B.1 applied to $(k - j', \psi_f, v_{f1}, v_{f2}) \in \mathcal{V}$ (from above) and $k - j \leq k - j'$ (which follows from $j' \leq j_1 + j_2 = j$, noting $j < k$),
• $\sigma_{f1}, \sigma_{f2} : k - j (\psi_f \otimes \psi_r) \leadsto S_f$
  which follows from Lemma B.3 applied to $\sigma_{f1}, \sigma_{f2} : k - j' (\psi_f \otimes \psi_r) \leadsto S_f$ (from above) and $k - j \leq k - j'$ (which follows from $j' \leq j_1 + j_2 = j$, noting $j < k$),
• $S_f^1 \subseteq S^1 \cup \text{alloc}(T_{f1})$
  which follows from above, and
• $S_f^2 \subseteq S^2 \cup \text{alloc}(T_{f2})$
  which follows from above.
C  Proof : Consistency

This appendix presents the detailed proof of consistency (Theorem 4.6).

Theorem C.1 (Consistency). If $\Gamma = FV(e)$, then $\Gamma \vdash e \approx_{ctx} e$.

Proof

Consider arbitrary $C, \sigma, \eta, n, \sigma_1$, and $T_1$ such that

- $C : (\Gamma)$,
- $\eta = FL(C) \cup FL(e) \cup FL(e)$,
- $\sigma : \eta$, and
- $\sigma, C[e] \downarrow_{ok} n, \sigma_1, T_1$.

We are required to show:

- $\exists v. \sigma, C[e] \downarrow_{ok} v, -, -$,
  which we conclude as follows:
  Note that $\exists v. \sigma, C[e] \downarrow_{ok} v, -, -$
  $\equiv \exists v, \sigma_2, T_2. \sigma, C[e] \downarrow_{ok} v, \sigma_2, T_2$.

  Take $v = n$. Take $\sigma_2 = \sigma_1$. Take $T_2 = T_1$.
  We are required to show: $\sigma, C[e] \downarrow_{ok} n, \sigma_1, T_1$,
  which follows from above.

- $\forall v. \sigma, C[e] \downarrow_{ok} v, -, - \implies n = v$,
  which we conclude as follows:
  Consider arbitrary $v, \sigma_2$, and $T_2$ such that

  - $\sigma, C[e] \downarrow_{ok} v, \sigma_2, T_2$.

  We are required to show that $n = v$.

From $\sigma, C[e] \downarrow_{ok} n, \sigma_1, T_1$, it follows that there exists $j_1$ such that $\sigma, C[e] \downarrow_{ok}^{j_1} n, \sigma_1, T_1$.

Hence, there exists $L_1$ such that

- $\sigma, C[e] \downarrow^{j_1} n, \sigma_1, T_1$,
- $\sigma : FL(C[e]) \hookrightarrow L_1$, and
- $L_1 \cap alloc(T_1) = \emptyset$.

From $\sigma, C[e] \downarrow_{ok} v, \sigma_2, T_2$, it follows that there exists $j_2$ such that $\sigma, C[e] \downarrow_{ok}^{j_2} v, \sigma_2, T_2$.

Hence, there exists $L_2$ such that

- $\sigma, C[e] \downarrow^{j_2} v, \sigma_2, T_2$,
- $\sigma : FL(C[e]) \hookrightarrow L_2$, and
- $L_2 \cap alloc(T_2) = \emptyset$. 

73
From $\sigma : FL(C[e]) \rightarrow \mathcal{L}_1$, it follows that

- $\mathcal{L}_1 = FL(C[e]) \cup \bigcup_{l \in \mathcal{L}_1} FL(\sigma(l)),$
- $\text{dom}(\sigma) \supseteq \mathcal{L}_1,$ and
- $\forall \mathcal{L}^1 \subseteq \mathcal{L}_1. \; FL(C[e]) \subseteq \mathcal{L}^1 \land (\forall l \in \mathcal{L}^1. \; FL(\sigma(l)) \subseteq \mathcal{L}^1) \implies \mathcal{L}_1 = \mathcal{L}^1.$

From $\sigma : FL(C[e]) \rightarrow \mathcal{L}_2$, it follows that

- $\mathcal{L}_2 = FL(C[e]) \cup \bigcup_{l \in \mathcal{L}_2} FL(\sigma(l)),$
- $\text{dom}(\sigma) \supseteq \mathcal{L}_2,$ and
- $\forall \mathcal{L}^1 \subseteq \mathcal{L}_2. \; FL(C[e]) \subseteq \mathcal{L}^1 \land (\forall l \in \mathcal{L}^1. \; FL(\sigma(l)) \subseteq \mathcal{L}^1) \implies \mathcal{L}_2 = \mathcal{L}^1.$

Note that $\mathcal{L}_1 = \mathcal{L}_2$, which follows from the minimality requirements (above) on $\mathcal{L}_1$ and $\mathcal{L}_2$.

Take $\psi = \{(l, l) \mid l \in FL(C[e])\}$.
Take $S = \{(l, l) \mid l \in \mathcal{L}_1\}$.
Note that $\sigma, \sigma \cdot j_{1+1} \psi \sim S$.

which follows we conclude as follows:

Take $F_\psi$ such that $\text{dom}(F_\psi) = S$ and

$$F_\psi(l, l) = \{(l_v, l_v) \mid l_v \in FL(\sigma(l))\}$$

We are required to show:

- $S \in \text{LocBij}$,
  which follows from the definition of $S$ as an identity relation,
- $F_\psi : S \rightarrow \text{LocBij}$,
  which follows from the definition of $F_\psi$ since $\text{rng}$ consists of identity relations,
- $S = \psi \odot \bigcup_{(l_1, l_2) \in S} F_\psi(l_1, l_2)$,
  which follows from the fact that $S, \psi,$ and each $\psi'$ in the range of $F_\psi$ are all identity relations, together with the following (from above):
  - $S^1 = \mathcal{L}_1 = \mathcal{L}_2 = S^2,$
  - $\mathcal{L}_1 = FL(C[e]) \cup \bigcup_{l \in \mathcal{L}_1} FL(\sigma(l)),$
  - $\mathcal{L}_2 = FL(C[e]) \cup \bigcup_{l \in \mathcal{L}_2} FL(\sigma(l)),$ and
  - $\psi^1 = \psi^2 = FL(C[e]).$
- $\text{dom}(\sigma) \supseteq S^1,$
  which follows from $\text{dom}(\sigma) \supseteq \mathcal{L}_1 = S^1$ (from above),
- $\text{dom}(\sigma) \supseteq S^2,$
  which follows from $\text{dom}(\sigma) \supseteq \mathcal{L}_2 = S^2$ (from above),
- $\forall (l_1, l_2) \in S. \forall i < j_{1+1}. \; (i, F_\psi(l_1, l_2), \sigma(l_1), \sigma(l_2)) \in \mathcal{V},$
  which we conclude as follows:
Consider arbitrary $l_1, l_2,$ and $i$ such that
- $(l_1, l_2) \in S$ and
- $i < j_{1+1}.$
Hence, note that \( l_1 = l_2 \), which follows from \((l_1, l_2) \in S\) since \( S \) is the identity relation.
Thus, we are required to show \((i, F_\psi(l_1, l_1), \sigma(l_1), \sigma(l_1)) \in V\).

Note that \( \sigma(l_1) \) is a closed value with possibly free locations.
Let \( v_1 \) be the result of substituting a fresh variable \( x_i \) for every free location \( l \) in \( \sigma(l_1) \).
Let \( \Gamma_1 = FV(v_1) \). Note that \( FL(S) = \emptyset \).
Applying Lemma 4.5 to \( v_1 \), we conclude that \( \Gamma_1 \vdash v_1 \approx v_1 \).
Hence, we have \( \Gamma_1 \vdash v_1 \approx v_1 \).

Let \( \gamma_1 = \{ x_i \mapsto l \mid l \in FL(\sigma(l_1)) \} \), where each \( x_i \) is the fresh variable that was substituted for the free location \( l \) in \( \sigma(l_1) \) above.
Let \( \psi_1 = \{ (l, l) \mid l \in FL(\sigma(l_1)) \} \).
Hence, note that \( \psi_1 = F_\psi(l_1, l_1) \).

Instantiate \( \Gamma_1 \vdash v_1 \approx v_1 \) (from above) with \( j_1 + 1, \psi_1, \gamma_1 \), and \( \gamma_1 \). Note that

- \( j_1 + 1 \geq 0 \) and
- \( (j_1 + 1, \psi_1, \gamma_1, \gamma_1) \in G[\Gamma_1] \),
which follows from the fact that

- for each \( x_i \in dom(\gamma_1) \),
  \( (j_1 + 1, \{ (l, l) \}, \gamma_1(x_i), \gamma_1(x_i)) \in V \)
  \( \equiv (j_1 + 1, \{ (l, l) \}, l, l) \in V \)
  which follows from the definition of \( V \), and

- \( \psi_1 \equiv \{ x_i \in FL(\sigma(l_1)) \} \{ (l, l) \} \)
  \( \equiv \{ x_i \in G_1 \} \{ (\gamma_1(x_i), \gamma_1(x_i)) \} \).

Hence, we have \( (j_1 + 1, \psi_1, \gamma_1(v_1), \gamma_1(v_1)) \in C \)
\( \equiv (j_1 + 1, \psi_1, \sigma(l_1), \sigma(l_1)) \in C \)
\( \equiv (j_1 + 1, F_\psi(l_1, l_1), \sigma(l_1), \sigma(l_1)) \in C \).

From the latter, since \( \sigma(l_1) \) is a value, it follows that \( (j_1 + 1, F_\psi(l_1, l_1), \sigma(l_1), \sigma(l_1)) \in V \).
Applying Lemma 4.5 to \( (j_1 + 1, F_\psi(l_1, l_1), \sigma(l_1), \sigma(l_1)) \in V \) and \( i < j_1 + 1 \), it follows that \( (i, F_\psi(l_1, l_1), \sigma(l_1), \sigma(l_1)) \in V \) as required.

Note that \( C[e] \) is a closed term with possibly free locations.
Let \( e_1 \) be the result of substituting a fresh variable \( x_i \) for every free location in \( C[e] \).
Let \( \Gamma_{10} = FV(e_1) \). Note that \( FL(e_1) = \emptyset \).
Applying Lemma 4.5 to \( e_1 \), we conclude that \( \Gamma_{10} \vdash e_1 \approx e_1 \).
Hence, we have \( \Gamma_{10} \vdash e_1 \approx e_1 \).

Let \( \gamma_{10} = \{ x_i \mapsto l \mid l \in FL(C[e]) \} \), where each \( x_i \) is the fresh variable that was substituted for the free location \( l \) in \( C[e] \) above.
Recall that \( \psi = \{ (l, l) \mid l \in FL(C[e]) \} \).

Instantiate \( \Gamma_{10} \vdash e_1 \approx e_1 \) with \( j_1 + 1, \psi, \gamma_{10} \), and \( \gamma_{10} \). Note that

- \( j_1 + 1 \geq 0 \) and
- \( (j_1 + 1, \psi, \gamma_{10}, \gamma_{10}) \in G[\Gamma_{10}] \),
which follows from
• for each $x_l \in \text{dom}(\gamma_{10})$,
  
  $$\{(j_1 + 1, \{(l, l), \gamma_{10}(x_l), \gamma_{10}(x_l)\}) \in \mathcal{V} \equiv (j_1 + 1, \{(l, l), l, l\}) \in \mathcal{V}$$

  which follows from the definition of $\mathcal{V}$, and

• $\psi \equiv \bigcirc_{l \in FL(C[e])} \{(l, l)\}$
  
  $$\equiv \bigcirc_{x_l \in \Gamma_{10}} \{(\gamma_{10}(x_l), \gamma_{10}(x_l))\}.$$ 

  Hence, we have $(j_1 + 1, \psi, \gamma_{10}(e_1), \gamma_{10}(e_1)) \in \mathcal{C}$

  $$\equiv (j_1 + 1, \psi, C[e], C[e]) \in \mathcal{C}$$

  Instantiate the latter with $j_1$, $\sigma$, $\sigma$, $\{\}$, $S$, $n$, $\sigma_1$, and $T_1$. Note that

• $j_1 < j_1 + 1$,

• $\sigma, \sigma : j_1 + 1 \ (\psi \odot \{\}) \leadsto S$,
  
  $$\equiv \sigma, \sigma : j_1 + 1 \psi \leadsto S,$$

  which follows from above,

• $\sigma, C[e] \downarrow j_1 n, \sigma_1, T_1$,

  which follows from above, and

• $S^1 \cap \text{alloc}(T_1) = \emptyset$,

  which follows from $S^1 = \mathcal{L}_1$ (from above) and $\mathcal{L}_1 \cap \text{alloc}(T_1) = \emptyset$ (from above).

  Hence, we have the following:

  $$\forall v_2, \sigma_2, T_2,$$

  $\sigma, C[e] \downarrow v_2, \sigma_2, T_2 \land$

  $$S^2 \cap \text{alloc}(T_2) = \emptyset \implies$$

  $$\exists \psi, S_f, (j_1 + 1 - j_1, \psi, n, v_2) \in \mathcal{V} \land$$

  $$\sigma_1, \sigma_2 : j_1 + 1 - j_1 \ (\psi \odot \{\}) \leadsto S_f \land$$

  $$S^1_f \subseteq S^1 \cup \text{alloc}(T_1) \land$$

  $$S^2_f \subseteq S^2 \cup \text{alloc}(T_2).$$

  Instantiate this with $v$, $\sigma_2$, and $T_2$. Note that

• $\sigma, C[e] \downarrow v, \sigma_2, T_2$,

  which follows from above, and

• $S^2 \cap \text{alloc}(T_2) = \emptyset$,

  which follows from $S^2 = \mathcal{L}_2$ (from above) and $\mathcal{L}_2 \cap \text{alloc}(T_2) = \emptyset$ (from above).

  Hence, there exist $\psi_f$ and $S_f$ such that

• $(1, \psi_f, n, v) \in \mathcal{V}$,

• $\sigma_1, \sigma_2 : 1 \psi \leadsto S_f$,

• $S^1_f \subseteq S^1 \cup \text{alloc}(T_1)$, and

• $S^2_f \subseteq S^2 \cup \text{alloc}(T_2)$.

  Note that from $(1, \psi_f, n, v) \in \mathcal{V}$, by the definition of $\mathcal{V}$, it follows that $n = v$ as we needed to show.

\[\square\]