Region Analysis for Imperative Languages

Radu Rugina and Sigmund Cherem
Computer Science Department
Cornell University
Ithaca, NY 14853
{rugina,siggi}@cs.cornell.edu

Abstract

This paper presents a region inference framework designed specifically for imperative programs with dynamic allocation and destructive updates. Given an input program, the algorithm automatically translates it into an output program with region annotations on procedures and allocation commands, and with explicit region creation and removal commands.

Our framework formulates the analysis problem as a three-step algorithm. In the first phase, it infers region annotations for record declarations in the input language. Second, it performs a unification-based flow analysis of the program, inferring region types at each point in the program. In particular, it determines region types for allocation commands and procedure calls. In the third phase, it uses a single-pass algorithm to inspect each point in the program and insert region creation and removal commands in the control flow of the output program. This transformation ensures that regions are live whenever they are being used, while minimizing region lifetimes. The algorithm is simple, efficient, and provably correct.

Furthermore, we show that the framework can be extended with more aggressive analyses (at the expense of making it less modular or more complex), such as interprocedural region liveness or shape analysis, to further improve the accuracy and performance of memory management. More generally, our framework allows existing analysis technology for imperative languages, such as points-to or shape analysis, to be easily integrated and applied to the region inference problem.

1 Introduction

Memory management is a fundamental problem in languages, compilers, and virtually in any programming system. The difficulty of manually managing the data in the program has led to a large body of research to improve the data management process using either run-time systems, language support, or static analysis; all of these techniques have the common goal of making memory management simpler to use, safer, more efficient, and more precise.

Region allocation is an approach that has been proposed in the past decade for reaching these goals. In a region-based system, the compiler groups objects together in regions, and then deallocates all regions at once. Region-based systems have a number of appealing properties which makes them a good match for addressing the memory management problem: they efficiently deallocate larger groups of data; they are easier to use because they allow users to reason about collections of objects rather than individual objects; and they can be incorporated in the language and statically checked by the compiler to ensure safety. Two main research directions have emerged in this area: one concerns adding language support for regions, as in the case of the Cyclone [10], or RC [8] systems; the other direction is to develop static analyses for automatically inferring regions. The work presented in this paper falls in the latter category. So far, the proposed analysis techniques for region inference include the seminal work of Tofte and Talpin [15, 3], and

other similar, but more flexible systems. However, these approaches are all targeted to functional languages and their commonly used features, such as polymorphism and higher-order functions; they are not directly applicable to imperative languages such as Java or C.

This paper presents a region inference algorithm specifically formulated in the context of imperative languages. Although we use the same general ideas as the Tofte and Talpin approach, our algorithm focuses on control flow constructs, program state, dynamic allocation, and destructive updates. Given an input program, our algorithm automatically translates it into an output program with region-based data management. The output program has region-annotated procedures and allocation commands, and contains region creation and removal commands [1] at appropriate points in the control flow. The translation guarantees that the regions are live whenever they are being used; at the same time, the translation minimizes the lifetimes of each region in the program.

The algorithm consists of three steps. In the first phase, it computes region annotations for declarations of data structures in the program. Second, it infers region types for each variable, at each program point. In particular, it infers region types for each allocation command and procedure call. This step consists of two subphases: an intra-procedural flow analysis of each procedure; and an inter-procedural propagation of unifications. The insight behind having a flow analysis is that we want to allow variables to have different region types at different program points. For instance, a variable may point into different regions at different program points. Finally, in the third phase the algorithm uses a single-pass algorithm to infer the appropriate points in the program for inserting region creation and removal commands. Unlike many existing region-based systems [15, 10], our framework doesn't use lexically scoped region lifetimes at the intra-procedural level; a region can even be created or removed in our framework at multiple points, along different paths. However, regions are created and removed in the same procedure.

The paper also presents extensions to the basic algorithm which allow more aggressive optimizations and further improve the generated code. *Interprocedural creation and removal* performs an interprocedural region liveness analysis to determine when regions can be allocated late, in the invoked procedures, or removed early, before returning to the procedures that create them. The downside is that this adds more inter-procedural analysis overhead and makes the analysis less modular. The second extension, *object migration*, allows the compiler to extract objects from existing regions and move them to different regions, whenever the analysis can precisely determine that each extracted object is no longer connected to the old region. This transformation requires precise shape analysis information; it also requires a special command for the migration of objects across regions.

Our region inference framework has a number of appealing properties:

- *Simplicity:* The framework consists of two simple and lightweight algorithms.
- *Correctness:* The algorithm is provably correct.
- *Flexibility:* The basic algorithm can be optionally extended with more aggressive analyses.
- *Applicability:* The algorithm provides a core region analysis for imperative languages. With few changes, the algorithm can be extended and applied to widely used imperative languages such as Java, C, or C++.

The contributions of this paper are twofold. The first contribution is the formulation of the region analysis problem in the context of imperative languages. We give a clean, formal presentation of the proposed region analysis algorithms. The second contribution of the paper is

[1] We use the terms *region creation* and *removal*, instead of *allocation* and *deallocation*, in order to avoid ambiguities between object allocation and region allocation.

that it provides a general region inference framework, where more aggressive analyses can build upon our basic algorithm and be applied to the region inference problem. In particular, it enables the direct application of existing analyses, such as points-to or shape analyses, to the memory management problem.

The paper is structured as follows. Sections 2 and 3 present the translation input and languages. Next, Section 4 shows an example of how our translation works. Section 5 presents the basic region inference algorithm, and Section 6 shows several extensions to this algorithm. Finally, we present related work in Section 7 and conclude in Section 8.

2 Input Language

Figure 1a) presents the abstract syntax for our input language. This is a simple imperative language which manipulates integers and dynamically allocated records, which we refer to as objects. A program consists of a sequence of type declarations for records, followed by a sequence of procedure declarations. For simplicity, we assume that procedures take exactly one argument and always return a value. The language allows directly and mutually recursive procedures. There is one special procedure main in the program, whose signature is $\mathsf{main} : \mathsf{int} \to \mathsf{int}$. The execution of the program consists of running the body of this function with an argument value of zero.

The commands in the language include declarations of local variables, control flow commands (sequencing, conditionals, and loops), assignments, and return commands. Assignments, in turn, include updates to variables and to fields of heap objects; the value being assigned can be either the object returned by new; the result of a procedure call; or the result of an expression. Finally, expressions include variable values, field values, the null reference, integers constants, and arithmetic operations denoted by $\oplus$.

Appendix A.1 gives a precise specification using large-step operational semantics. The rules for evaluating expressions use judgments of the form $\langle S, H, \mathsf{expr}\rangle \to v$, with the meaning that in the stack environment S and heap environment H, expression expr evaluates to value v. The rules for commands use judgments of the form $\langle S, H, \mathsf{com}\rangle \to \langle S', H', v_{\mathsf{ret}}\rangle$ meaning that in the stack environment S and heap environment H, the evaluation of command com yields a new stack S' and heap H', and a return value v_{ret}. The latter is being used to ensure that the body of each procedure returns a value. We use a special value $v_{\mathsf{ret}} = \mathsf{nr}$ to indicate when a command doesn't return.

We informally describe the semantics of the input language as follows. An if statement executes its false branch if the condition expression evaluates to 0 (or null), and executes its true branch otherwise; similarly, the body of a while loop is executed as long as the condition expression is non zero (or not null). At declarations, variables are initialized with their default values (0 for integers, and null for references); the fields of dynamically allocated records are initialized with their default values; and variables must be declared before being used. Finally, the command representing the body of a procedure must always return a value.

The program types τ include integers, function types, references types to records, and a bottom type to model null references:

$$\tau \quad ::= \quad \mathsf{int} \mid \tau_1 \to \tau_2 \mid \bot \mid \mathsf{ref}\ (\overline{\tau}\ \overline{\mathsf{f}})$$

The null reference can be assigned to a reference variable of field of any reference type.

$$
\begin{array}{rcl}
\mathsf{Program} & ::= & \mathsf{typeDecl} * \ \mathsf{procDecl} * \\
\mathsf{typeDecl} & ::= & \mathsf{record\ s\ =\ (\bar{t}\ \bar{f})} \\
\mathsf{procDecl} & ::= & \mathsf{t\ p\ (t\ z)\ =\ com} \\
\mathsf{t} & ::= & \mathsf{int} \mid \mathsf{s} \\
\\
\mathsf{com} & ::= & \mathsf{int\ x\,;\ com} \mid \mathsf{s\ x\,;\ com} \mid \\
 & & \mathsf{com_1\,;\ com_2} \mid \mathsf{while\ (expr)\ com} \mid \\
 & & \mathsf{if\ (expr)\ then\ com_1\ else\ com_2} \mid \\
 & & \mathsf{x = expr} \mid \mathsf{x.f = expr} \mid \\
 & & \mathsf{x = new\ s} \mid \mathsf{x = p\ (y)} \mid \mathsf{return\ x} \\
\\
\mathsf{expr} & ::= & \mathsf{x} \mid \mathsf{x.f} \mid \mathsf{null} \mid \mathsf{n} \mid \mathsf{expr_1} \oplus \mathsf{expr_2}
\end{array}
$$

$\mathsf{x, y, z} \in$ Variables $\quad \mathsf{f} \in$ Fields $\quad \mathsf{s} \in$ Records
$\mathsf{p} \in$ Procedures $\quad \mathsf{n} \in$ Integer constants

$$
\begin{array}{rcl}
\mathsf{Program} & ::= & \mathsf{typeDecl} * \ \mathsf{procDecl} * \\
\mathsf{typeDecl} & ::= & \mathsf{record\ s}\,[\mathsf{r_0}, \bar{\mathsf{r}}]\ =\ (\bar{\mathsf{t}}\ \bar{\mathsf{f}}) \\
\mathsf{procDecl} & ::= & \mathsf{t\ p}\ [\bar{\mathsf{r}}]\ (\mathsf{t\ z})\ =\ \mathsf{com} \\
\mathsf{t} & ::= & \mathsf{int} \mid \mathsf{s}\ [\mathsf{r_0}, \bar{\mathsf{r}}] \\
\\
\mathsf{com} & ::= & \ldots \mid \mathsf{create\ r} \mid \mathsf{remove\ r} \\
 & & \mid \mathsf{x = new\ s\ in\ r} \\
 & & \mid \mathsf{x = p}\ [\bar{\mathsf{r}}]\ (\mathsf{y})
\end{array}
$$

where: $\mathsf{r} \in$ Region names

Figure 1: Syntax of the input and output languages

3 Output Language

Figure 1b) presents the abstract syntax for the output language. This language extends the input language in two ways. First, it adds regions annotations for declarations of record types and procedures. Second, it adds commands for manipulating regions: region creation and removal commands, dynamic allocation of objects into regions, and region based procedure calls.

For types, the construct $[\mathsf{r_0}, \bar{\mathsf{r}}]$ denotes the type of a reference which points into region $\mathsf{r_0}$, and is such that all of the regions reachable from this reference, except $\mathsf{r_0}$, are in the sequence $\bar{\mathsf{r}}$. We call $\mathsf{r_0}$ the *base region* of the record, and $\bar{\mathsf{r}}$ the *region parameters* of the record. This construct allows the compiler to grant access to the structure pointed to by such a reference only if it can guarantee that $\mathsf{r_0}$ and all of the regions in $\bar{\mathsf{r}}$ have not been removed.

For procedure declarations, the construct $[\bar{\mathsf{r}}]$ denotes the *formal region parameters* of the procedure. The sequence $\bar{\mathsf{r}}$ must include all of the regions in the argument and return type. For procedure calls, the construct $\bar{\mathsf{r}}$ denotes the *actual region parameters* at the call site. We require that all of the actual regions are live at the call site.

The commands for manipulating regions have the standard meanings: create r dynamically creates a new region, remove r dynamically removes a region, and new s in r dynamically allocates an object of record type s in region r. We require that region r is not created when remove is invoked, and that r is not removed when it is used in any other command or annotation.

Appendix A.2 provides a specification for the commands in the output language. The semantics is defined using judgments of the form $\langle S, H, R, \mathsf{com}\rangle \rightarrow \langle S', H', R', v_{\mathsf{ret}}\rangle$, where S, H, S', H' and v_{ret} are as before, and R and R' are the sets of live regions before and after the evaluation of command com. The majority of rules are identical to the corresponding rules from the input language, and are therefore not shown; we present only the rules which involve regions.

For types, we add region variables ρ and type schemes σ which include universal quantification over regions:

$$
\begin{array}{rcl}
\rho & \in & \text{Region variables} \\
\sigma & ::= & \tau \mid \forall \overline{\rho}\ .\ \tau \\
\tau & ::= & \mathsf{int} \mid \tau_1 \rightarrow \tau_2 \mid \bot \mid \mathsf{ref}\ [\rho_0, \overline{\rho}]\ (\overline{\tau}\ \bar{\mathsf{f}})
\end{array}
$$

The recursive syntax for types corresponds to recursive types. For simplicity, we omit the

recursive type operator (μ) from the syntax, and express recursive types by their recursive type equations. For instance, the recursive type definition:

$$\tau = \mathsf{ref}\ [\rho]\ (\mathsf{int}\ \mathsf{data},\ \tau\ \mathsf{next})$$

stands for the recursive type:

$$\tau_{\mathsf{r}} = \mu\tau\ .\ \mathsf{ref}\ [\rho]\ (\mathsf{int}\ \mathsf{data},\ \tau\ \mathsf{next})$$

Note that the universal quantification always encloses recursive types. In other words, we require that the recursive occurrences of types have identical region parameters. For instance, all the elements of a list or a tree structure have to be allocated in the same region. Consider the following type scheme for a list whose elements have a data field and a next link:

$$\sigma_{\mathsf{List}} = \forall\rho\ .\ (\mu\tau\ .\ \mathsf{ref}\ [\rho]\ (\mathsf{int}\ \mathsf{data},\ \tau\ \mathsf{next}))$$

This shows that all of the elements of the list must be allocated in the same region ρ.

4 Example

Figure 2 presents an example output program program generated by our translation, that illustrates several features of our region inference algorithm. The region annotations are shown in bold. The input program is the same, but without the region annotations.

The program uses two functions, main and copy, to perform list manipulations. In this program, list elements and data are being allocated as different heap objects, as shown by the record declarations for List and Data; we refer to the list elements alone as the spine of the list. The copy procedure is a recursive procedure which takes a List as argument and produces a list with a different spine, but with the same elements as the original list. The main procedure works as follows. First, it creates a list using the first loop. At each iteration it allocates a new element in the spine, and a new data element. Next, the procedure performs a "re-spining" operation using the assignment x = copy(x): it replaces its old spine with a new one; after the assignment, the old spine becomes garbage. Finally, the program runs another loop. At each iteration in this second loop, it creates a list y with a new spine, processes it, then discards the spine by nullifying y. The procedure process is omitted from the code; it takes a list as argument, uses it for internal computation, and returns an integer value.

We highlight several details of this transformation. First, the analysis identifies that procedure copy returns a list with data in the same region as the parameter list, as shown by the two occurrences of r_5 in the argument and the return value for copy. The transformation also annotates the allocation command in this function with region r_7, which is the region for the spine of the returned list. The analysis infers that r_7 is a parameter region and does not generate a creation command for it in the body of the procedure; it is the responsibility of the caller to create this region.

Second, the analysis accurately determines the region creation points for the objects allocated in the two loops of procedure main. For the first loop, which iteratively creates a list, the analysis places the region creation commands for regions r_1 and r_2 outside the loop. For the second loop, the analysis determines that the lifetime of the region returned by copy doesn't span outside the loop body; it therefore places the region creation and removal commands for this region inside the loop.

Third, for the "re-spining" assignment x = copy(x), the transformation places a region creation command for region r_3, the region for the new spine, before invoking copy. It also generates a

```
record Data [r_d] = (int i)
record List [r_l,r_d] = (Data[r_d] d, List [r_l,r_d] n)

int main(int i) {
  List x;

  create r_1;
  create r_2;
  while (i - 10) {
    Data d = new Data in r_1;
    d.i = i;
    t = new List in r_2;
    t.d = d;
    t.n = x; x = t;
    i = i + 1
  }

  create r_3;
  x = copy[r_1,r_2,r_3](x);
  remove r_2;

  while (i) {
    create r_4;
    List y = copy[r_1,r_3,r_4](x);
    i = process[r_1,r_4](y);
    remove r_4
  }
  remove r_1;
  remove r_3;
  return i
}

List[r_5,r_7] copy[r_5,r_6,r_7]
      (List[r_5,r_6] x) {
  List y,z,t;

  if (x) {
    y = new List in r_7;
    y.d = x.d;
    z = x.n;
    t = copy[r_5,r_6,r_7](z);
    y.n = t;
    return y
  } else {
    t = null;
    return t
  }
}

int process[r_8,r_9]
      (List[r_8,r_9] x) {
  ...
}
```

Figure 2: Example translated program. The region annotations are shown in bold. The input program is the same, but without the region annotations.

region removal command for region r_2, the region of the old spine, detecting that this region becomes garbage as a result of the assignment.

Fourth, the analysis automatically determines the appropriate actual region parameters at call sites. For the first call to copy it computes the region parameters $[\mathsf{r}_1, \mathsf{r}_2, \mathsf{r}_3]$. For the second call, it computes parameters $[\mathsf{r}_1, \mathsf{r}_3, \mathsf{r}_4]$.

Finally, this example shows that region lifetimes are not necessarily nested into each other. For instance, the placement of create r_3 before remove r_2 makes the lifetimes of these regions to overlap. Our the region inference algorithm doesn't require a particular discipline of region lifetimes, such as the stack discipline when regions are lexically scoped.

5 Region Inference Algorithm

This section presents the region analysis algorithm and the translation process. The algorithm consists of three phases. First, a *class declarations phase* infers region parameterized types for records. Second, a *record declaration phase* computes region types for variables at each point in the program. It consists of an intra-procedural flow analysis of each procedure; and an inter-procedural analysis which propagates type unifications between procedures. Third, a *translation phase* produces the output program. It generates region annotations for records, procedures, and allocations using the result from the first two phases. It also places region creation and removal statements at appropriate points in the control flow of the output program. We describe each of these steps in the following sections.

5.1 Record Declarations

The inference algorithm for record declarations is fairly straightforward. We require that the region parameters in each field of a record be a subset of the region parameters of the enclosing region; if the field is an instance of the recursive type, then these region parameters must be identical. For each record s, the analysis generates a fresh region name r_s to represent the base region of the record.

We define the translation using a function $\mathcal{T}[\![\cdot]\!]$ which maps a piece of syntax from the input language into a piece of syntax in the output language. The translation of record declarations is:

$$\begin{array}{rcll} \mathcal{T}[\![\mathsf{record\ s\ (\bar{t}\ \bar{f})}]\!] & = & \mathsf{record}\ \mathcal{T}[\![\mathsf{s}]\!]\ (\overline{\mathcal{T}[\![\mathsf{t}]\!]}\ \bar{\mathsf{f}}) & \\ \mathcal{T}[\![\mathsf{int}]\!] & = & \mathsf{int} & \\ \mathcal{T}[\![\mathsf{s}]\!] & = & \mathsf{s}\ [\mathsf{r}_\mathsf{s}, \bar{\mathsf{r}}_\mathsf{ps}] & (\mathsf{r}_\mathsf{s}\ \text{fresh}) \end{array}$$

where each sequence $\bar{\mathsf{r}}_\mathsf{ps}$ of region parameters is an arbitrary permutation of the set $\mathsf{RV(s)}$ of region variables, computed as the least fixed point of the following constraints:

$$\frac{\mathsf{record\ s\ (\bar{t}\ \bar{f})} \quad \mathsf{s'} \in \bar{\mathsf{t}}}{\mathsf{RV(s')} \subseteq \mathsf{RV(s)}} \qquad \frac{}{\mathsf{r_s} \subseteq \mathsf{RV(s)}}$$

We compute the least fixed point using a standard iterative computation which initializes each $\mathsf{RV(s)}$ to the empty set, and then applies the rules until no changes occur.

For example, the translation of the following declaration:

```
record Data = (int x, int y)
record List = (Data data, List next))
record Iterator = (List crt)
```

is:

$$\begin{array}{l}\mathsf{record\ Data\ [rd]\ =\ (int\ x,\ int\ y)}\\ \mathsf{record\ List\ [rl, rd]\ =\ (Data\ [rd]\ data,\ List\ [rl, rd]\ next))}\\ \mathsf{record\ Iterator\ [ri, rl, rd]\ =\ (List\ [rl, rd]\ crt)}\end{array}$$

At this point, the algorithm computes types for records, using the result of the above translation:

$$\frac{\mathsf{record\ s\ [r_0, \bar{r}]\ (\bar{t}\ \bar{f})} \quad \rho_0 = \mathrm{fresh}(\mathsf{r_0}) \quad \overline{\rho} = \mathrm{fresh}(\bar{\mathsf{r}})}{\mathsf{type(s)\ =\ ref\ [\rho_0, \overline{\rho}]\ (\bar{t}[\rho_0/r_0, \overline{\rho}/\bar{r}]\ \bar{f})}}$$

The function fresh return a fresh region variable for each region name passed as argument.

5.2 Region Inference

The algorithm next infers region-polymorphic types for methods, and region types for variables at each program point. First, the algorithm uses a unification-based intra-procedural flow analysis which processes each method in the program once. Then, it performs an inter-procedural analysis which propagates type unifications beyond procedure boundaries. We describe each of these steps in turn.

The intra-procedural analysis of each procedure first builds a type environment Δ which assigns the most general types to procedures, call sites, and allocation commands, using fresh region variables. For procedures and call sites, the analysis assigns fresh region variables for each occurrence of a reference in the parameter and in the result value of the procedure being declared or invoked. For allocation sites, is assigns fresh region variables to the allocated record. To formalize the construction of this environment, we introduce labels for each allocation expression $\mathsf{a : new\ s}$ and each call expression $\mathsf{c : p(y)}$. The rules for computing Δ are as follows:

$$\frac{\begin{array}{cc}\mathsf{t_2\ p\ (t_1\ z)} & \tau_1 = \mathrm{fresh}(\mathsf{type(t_1)})\\ \overline{\rho} = \mathsf{FV}(\tau_1 \to \tau_2) & \tau_2 = \mathrm{fresh}(\mathsf{type(t_2)})\end{array}}{\Delta \vdash \mathsf{p}\ :\ \forall\overline{\rho}\ .\ \tau_1 \to \tau_2} \qquad \frac{\begin{array}{cc}\Delta \vdash \mathsf{p}\ :\ \forall\overline{\rho}\ .\ \tau_1 \to \tau_2\\ \mathsf{c : p(y)} & \tau_1' \to \tau_2' = \mathrm{fresh}(\tau_1 \to \tau_2)\end{array}}{\Delta \vdash \mathsf{c}\ :\ \tau_1' \to \tau_2'}$$

$$\frac{\begin{array}{c}\mathsf{a : new\ s} \quad \mathsf{type(s)} = \tau = \mathsf{ref}\ [\rho_0, \overline{\rho}]\ (\overline{\tau}\ \bar{\mathsf{f}})\\ \rho_0' = \mathrm{fresh}(\rho_0) \quad \overline{\rho'} = \mathrm{fresh}(\overline{\rho})\end{array}}{\Delta \vdash \mathsf{a}\ :\ \tau[\rho_0'/\rho_0, \overline{\rho'}/\overline{\rho}]}$$

where $\mathsf{FV}(\tau)$ is the set of free region variables in τ. In the first two rules, the function fresh takes a type and replaces each of its region variables with fresh variables; it then returns the new type. In the last rule, fresh yields a fresh region variable for each of its argument regions. Note that we initialize all fields of a record with fresh region variables, although they are being initialized to null in the operational semantics. In other words we fix the regions for the values that will be later be stored in these fields. This is important for ensuring that the translation is sound.

A type environment Γ maps variables to their types. We define standard typing judgments for expressions $\Gamma \vdash e : \tau$, with the meaning that, in the environment Γ, expression e is well-typed and has type τ:

$$\frac{}{\Gamma \vdash \mathsf{n : int}} \qquad \frac{}{\Gamma \vdash \mathsf{null} : \bot} \qquad \frac{\mathsf{x} : \tau \in \Gamma}{\Gamma \vdash \mathsf{x} : \tau}$$

$$\frac{\Gamma \vdash \mathsf{expr_1 : int} \quad \Gamma \vdash \mathsf{expr_2 : int}}{\Gamma \vdash \mathsf{expr_1} \oplus \mathsf{expr_2 : int}}$$

$$\frac{\Gamma \vdash \mathsf{x : ref}\ [\rho_0, \overline{\rho}]\ (\overline{\tau}\bar{\mathsf{f}}) \quad \tau\mathsf{f} \in \overline{\tau}\bar{\mathsf{f}}}{\Gamma \vdash \mathsf{x.f} : \tau}$$

$$\frac{\Delta, \Gamma \cup \{\mathsf{x} \mapsto \mathsf{int}\} \vdash \mathsf{com}, \Gamma' \cup \{\mathsf{x} \mapsto \tau\}, E}{\Delta, \Gamma \vdash \mathsf{int\ x}\,;\ \mathsf{com}, \Gamma', E}$$

$$\frac{\Delta, \Gamma \cup \{\mathsf{x} \mapsto \bot\} \vdash \mathsf{com}, \Gamma' \cup \{\mathsf{x} \mapsto \tau\}, E}{\Delta, \Gamma \vdash \mathsf{s\ x}\,;\ \mathsf{com}, \Gamma', E}$$

$$\frac{\Delta, \Gamma \vdash \mathsf{com}_1, \Gamma_1, E_1 \quad \Delta, \Gamma_1 \vdash \mathsf{com}_2, \Gamma_2, E_2}{\Delta, \Gamma \vdash \mathsf{com}_1; \mathsf{com}_2, \Gamma_2, E_1 \cup E_2}$$

$$\frac{\begin{array}{c}\Delta, \Gamma \vdash \mathsf{com}_1, \Gamma_1, E_1 \quad \Delta, \Gamma \vdash \mathsf{com}_2, \Gamma_2, E_2 \\ (E, \Gamma) = \mathsf{unify}(\Gamma_1, \Gamma_2)\end{array}}{\Delta, \Gamma \vdash \mathsf{if\ (expr)\ then\ com_1\ else\ com_2}, \Gamma, E_1 \cup E_2 \cup E}$$

$$\frac{\Delta, \Gamma_1 \vdash \mathsf{com}, \Gamma_2, E_1 \quad (E, \Gamma_1) = \mathsf{unify}(\Gamma, \Gamma_2)}{\Delta, \Gamma \vdash \mathsf{while\ (expr)\ com}, \Gamma_1, E_1 \cup E}$$

Figure 3: Analysis rules for variable declarations and control flow constructs

Next, the algorithm performs the actual type inference via unification. It uses a flow analysis which generates unification constraints as a result of analyzing commands. Each unification constraint in E is a pair of unified region variables. The reason for using a flow analysis instead of a flow-insensitive approach is the presence of destructive updates in our input and output languages. Destructive updates to variables in the program yields new types for the updated variables, with new region instantiations. The algorithm must compute these types in a flow sensitive manner. The result of the algorithm is a type environment Γ at each program point.

We formulate our flow analysis for region inference using judgments of the form $\Delta, \Gamma \vdash \mathsf{com}, \Gamma', E$, which express the fact that in the typing environment Δ for procedures and allocations, and given the region type environment Γ for variables, the command com yields a new type environment Γ' and a set of unification constraints E. Figure 3 shows the analysis rules for variable declarations and control flow constructs. For variable declarations, the initialize types of references to $\bot$ and the types of integer variables to int.

The rules for control flow merge type information at control flow points using the unification function unify. This function takes two type environments, unifies the types of the corresponding variables, and yields the generated unification constraints and a resulting type environment. The reason for returning a type environment is that the analysis must handle subtyping relations for bottom types $\bot$) Unifying the bottom type $\bot$ with another type τ, yields no unification constraints, and returns τ in the environment. If none of the arguments to unify are $\bot$, the unification function recurses on the structures of the types.

Note that the flow analysis domain is the cartesian product of lattices of height 1, corresponding to the two possible types for references, $\bot$ and ref, ordered by subtyping. Hence, the analysis domain is a lattice with height equal to the number of reference variables. As a result, the number of iterations in the analysis of each loop is bounded by the number of reference variables updated in the loop body.

Figure 4 presents the analysis rules for assignments and return commands. In these rules, ret represents a dummy variable introduced in the type environment to keep track of the return type. The unification function works as before, but on pairs of types rather than type environments.

These represent the core rules of our analysis. They show how the analysis handles destructive

$$\frac{\Gamma \vdash \mathsf{expr} : \tau}{\Delta, \Gamma \vdash \mathsf{x} = \mathsf{expr}, \Gamma[\mathsf{x} \mapsto \tau], \emptyset}$$

$$\frac{\Gamma \vdash \mathsf{x.f} : \tau_1 \quad \Gamma \vdash \mathsf{expr} : \tau_2 \quad (E, \_) = \mathsf{unify}(\tau_1, \tau_2)}{\Delta, \Gamma \vdash \mathsf{x.f} = \mathsf{expr}, \Gamma, E}$$

$$\frac{\Delta \vdash \mathsf{a} : \tau}{\Delta, \Gamma \vdash \mathsf{x} = \mathsf{a} : \mathsf{new\ s}, \Gamma[\mathsf{x} \mapsto \tau], \emptyset}$$

$$\frac{\Delta \vdash \mathsf{c} : \tau_1 \to \tau_2 \quad \mathsf{y} : \tau_3 \in \Gamma \quad (E, \_) = \mathsf{unify}(\tau_1, \tau_3)}{\Delta, \Gamma \vdash \mathsf{x} = \mathsf{c} : \mathsf{p(y)}, \Gamma[\mathsf{x} \mapsto \tau_2], E}$$

$$\frac{\mathsf{x} : \tau_1 \in \Gamma \quad \mathsf{ret} : \tau_2 \in \Gamma \quad (E, \_) = \mathsf{unify}(\tau_1, \tau_2)}{\Delta, \Gamma \vdash \mathsf{return\ x}, \emptyset, E}$$

Figure 4: Analysis rules for assignments and return.

updates of variables and heap objects. The main observation is that updates to stack locations also update their types, while updates to heap locations require type unification. We refer to the former as *strong type updates*, and to the latter as *weak type updates*. The reason for having two kinds of updates lays in the reachability properties of the updated locations. Remember that the region types of variables model all of the regions reachable from that variable; the type update of a variable must maintain this invariant. But updates in the program can change the reachability information in two ways: 1) they may change the regions reachable from the updated variable; and 2) they may change the reachability information for other locations that reach the updated location. Therefore, the analysis must correctly model these two effects. Our analysis performs strong type updates for variable assignments (rules for $\mathsf{x} = ...$), and performs type unification, i.e. weak type updates, for modifications of heap locations (rule for $\mathsf{x.f} = \dots$); these rules correctly model the changes in reachability, according to the language semantics. Strong type updates are possible because our language forbids pointers to variables. Our rules are applicable to languages with similar semantics, like Java.

For pointer-based languages like C, the compiler can apply strong type updates only when it is able to determine that the assigned variable is not pointed to by other locations. Existing pointer analyses can give such guarantees and extend the above analysis rules in those cases. Similarly, shape analyses can provide guarantees that destructive heap updates only modify individual, disconnected heap locations; this would enable the application of strong type updates on heap locations as well.

The overall analysis of the current procedure p instantiates its argument and return type in the environment, then analyzes its body:

$$\frac{\begin{array}{c}\mathsf{t_2\ p\ (t_1\ z)} = \mathsf{com} \quad \Delta \vdash \mathsf{p} : \forall \overline{\rho}\ .\ \tau_1 \to \tau_2 \\ \Delta, \{\mathsf{z} \mapsto \tau_1, \mathsf{ret} \mapsto \tau_2\} \vdash \mathsf{com}, \Gamma, E_\mathsf{p}\end{array}}{\vdash E_\mathsf{p}}$$

The analysis result is the set E_p of unification constraints for procedure p. Its transitive closure yields the sets of unified region variables. Each region variable is then replaced with the representative of its equivalence class.

In the second part of the region inference phase, the algorithm performs inter-procedural region unifications, to ensure that the region type of each procedure at each call site is a valid

$$\frac{(\rho = \rho') \in E_{intra}}{(\rho = \rho') \in E_{inter}} \qquad \frac{\begin{array}{c}\Delta_P \vdash \mathsf{p} : \forall\overline{\rho}\ .\ \tau_1 \to \tau_2 \quad \mathsf{x} = \mathsf{c} : \mathsf{p}(\mathsf{y}) \\ \rho_i, \rho_j \in \overline{\rho} = \{\rho_k\}_{\overline{1,n}} \quad (\rho_i = \rho_j) \in E_{inter} \\ \Delta_P \vdash \mathsf{s} : \tau_1' \to \tau_2' \quad \overline{\rho'} = FV(\tau_1' \to \tau_2')\end{array}}{(\rho_i' = \rho_j') \in E_{inter}}$$

Figure 5: Inter-procedural analysis rules

instantiation of its declared region-polymorphic type. More precisely, if two region parameters of a procedure get unified, then the corresponding regions at each call site must also be unified. Such unifications can recursively propagate beyond multiple procedures. Figure 5 formalizes these unification rules. The environment Δ_P is the union of initialization environments of all procedures, and E_{intra} is the set of unifications produced by the intra-procedural analysis of all procedures. The final set of unifications E_{inter} is the least fixed point solution of these constraints.

5.2.1 Example

Figure 6 shows how the type inference algorithm works for the copy procedure from the example program in Section 4. The first column shows the code after the parameterized region types for Data and List have been inferred. For simplicity, we omit the region parameters in the recursive occurrence of List in its declaration. The second column shows the initializations for the type environment Δ. The algorithm assigns types to procedure copy (universally quantified over regions), to the allocation site new s, and to the recursive call copy(z). For simplicity, we denote list types as $\mathsf{List}[\rho, \rho']$ to mean the recursive type $\tau = \mathsf{ref}[\rho, \rho']\ (\mathsf{ref}[\rho'](\mathsf{int\ i})\ \mathsf{d},\ \tau\ \mathsf{n})$.

Program Code	Initialization Types	Computed Types	Unifications (intra)	Unifications (inter)
record Data[rd] = (int i) record List[rl, rd] = (Data[rd] d, List[rl, rd] n)				
List copy(List x) {	copy : $\forall\rho_1, \rho_2, \rho_3, \rho_4$. $\mathsf{List}[\rho_1, \rho_2] \to \mathsf{List}[\rho_3, \rho_4]$	x : $\mathsf{List}[\rho_1, \rho_2]$ ret : $\mathsf{List}[\rho_3, \rho_4]$		
List y, z, t;		y, z, t : $\bot$		
if (x) {				
y = new s;	$\mathsf{List}[\rho_5, \rho_6]$	y : $\mathsf{List}[\rho_5, \rho_6]$		
y.d = x.d;			$\rho_2 = \rho_6$	
z = x.n;		z : $\mathsf{List}[\rho_1, \rho_2]$		
t = copy(z);	$\mathsf{List}[\rho_7, \rho_8] \to \mathsf{List}[\rho_9, \rho_{10}]$	t : $\mathsf{List}[\rho_9, \rho_{10}]$	$\rho_7 = \rho_1, \rho_8 = \rho_2$ $\rho_5 = \rho_9, \rho_6 = \rho_{10}$	
y.n = t;				
return y			$\rho_3 = \rho_5, \rho_4 = \rho_6$	$\rho_8 = \rho_{10}$
} else {				
t = null;				
return t				
}				
}				

Figure 6: Region type inference example. Inferred type of copy is $\forall\rho_1, \rho_2, \rho_3$. $\mathsf{List}[\rho_1, \rho_2] \to \mathsf{List}[\rho_3, \rho_2]$, the type of the recursive call is $\mathsf{List}[\rho_1, \rho_2] \to \mathsf{List}[\rho_3, \rho_2]$, and the type of the allocation is $\mathsf{List}[\rho_3, \rho_2]$

The third column shows the types that the analysis computes at each program point, and the last two columns shows the generated intra-procedural and inter-procedural unification constraints. In this example, the destructive update $\mathsf{y.n} = \mathsf{t}$ requires the unification of the types of the objects referenced by y and t, as a result of adding an edge between them. The next command ($\mathsf{return\ y}$) then unifies the return type of the procedure with the type of t at this point. As a result of these two unification, the analysis infers that $\rho_2 = \rho_4$, i.e., that the returned list has the data in the same region as the argument list. Because these are region parameters in the signature of the function, the inter-procedural part of the analysis must propagate this constraint to the recursive call site. The generated inter-procedural constraint is $\rho_8 = \rho_{10}$. However, this relation was already implied by the intra-procedural constraints, so it doesn't change the existing region equivalence classes.

5.3 Translation

Once unification has been performed and types for procedures and allocation sites have been computed, it is straightforward to translate procedures and commands in the output language. The algorithm first maps each region variable to a region name: $\mathcal{T}[\![\rho]\!] = \mathsf{r}$ (where unified variables map to the same region name). It then uses this map to translate occurrences of record names in the program. If a record s is such that $\mathsf{type(s)} = \mathsf{ref}\ [\rho_0, \overline{\rho}]\ (\overline{\tau}\overline{\mathsf{f}})$ then the algorithm translates each occurrence of s in the argument or result type of a procedure as: $\mathcal{T}[\![\mathsf{s}]\!] = \mathsf{s}\ [\mathcal{T}[\![\rho_0]\!], \overline{\mathcal{T}[\![\rho]\!]}]$, and each occurrence of s in an allocation command as: $\mathcal{T}[\![\mathsf{new\ s}]\!] = \mathsf{new\ s\ in}\ \mathcal{T}[\![\rho_0]\!]$. For each procedure whose type in the environment Δ is $\forall\overline{\rho}\ .\tau_1 \rightarrow \tau_2$, we translate the name in its declaration as: $\mathcal{T}[\![\mathsf{p}]\!] = \mathsf{p}\ [\overline{\mathcal{T}[\![\rho]\!]}]$. Finally, we use a similar translation for procedure names at each call sites, but using the actual region parameters $\overline{\rho}$ at these points, as given by region types of these calls in the environment Δ. All of the other constructs remain unchanged in the generated code.

At this point, the analysis has inferred region types for procedure definitions, allocation commands, and procedure calls. To complete the algorithm, the compiler needs to insert region creation and removal statements at appropriate points in the program. This is the main job of the translation phase.

Let us denote by $\mathsf{R_p}$ the region parameters of the currently analyzed procedure, by $\mathsf{R_a}$ the regions at allocation sites, and by $\mathsf{R_c}$ the regions at call sites. Of all these regions, only the ones in $\mathsf{R_p}$ are already created at the entry point of the procedure. For each region in $(\mathsf{R_a} \cup \mathsf{R_c}) - \mathsf{R_p}$, the compiler must insert appropriate region creation commands to ensure that the required regions exist when the command is executed. We say that the regions in $(\mathsf{R_a} \cup \mathsf{R_c}) - \mathsf{R_p}$ are *local* to the current procedure. We give a precise definition of the placement problem for region creation and removal commands, and then present our solution.

5.3.1 Problem definition

We introduce the notions of a definition, use, and liveness of a region, as follows. The *definition* of a region is the procedure entry point, for parameter regions, or the creation point, for local regions. The *uses* of a region are the program points where its corresponding region variable occurs in the type environment at that point. We say that a local region is *live* at a point if it is used at that point; parameter regions are live throughout the body of the procedure, regardless of whether they are being used or not. The reason is that the compiler doesn't know whether the caller holds live data in those regions, and it must conservatively assume they do.

With these definitions, the solution to the region placement problem must satisfy the following constraints:

- **Safety:** For each procedure and each path in the flow graph of that procedure, the following conditions must hold:

  - Each use of a region must be preceded by a definition of the region, and there should be no removal of the region since the last creation of the region.
  - For each region there cannot be consecutive occurrences of creation or removal commands.
  - Each removal of a region must be preceded by a definition of that region.

- **Precision:** The placement of region creation and removal commands should minimize the lifetimes of each region.

We next argue that, given the above conditions and assuming no interprocedural analysis, the algorithm should not insert creation and removal commands for region parameters. First, the region parameters may contain live data in the caller, which means that, in the absence of interprocedural information, the compiler cannot remove them in the callee. As a result, it cannot insert creation sites either, because they are already defined at procedure entry and additional creation sites would violate the second safety condition above.

Hence, the problem of placement of region commands applies only to the local regions in $(\mathsf{R_a} \cup \mathsf{R_c}) - \mathsf{R_p}$. For each of them, the algorithm must insert creation and removal commands that satisfy the safety and optimality conditions listed above.

5.3.2 Placement of Region Commands

Despite its apparent complexity, the command placement problem has a short solution: the compiler places a `create r` command at each point in the program where `r` becomes live, and inserts a `remove r` command at each point where it ceases to be live. There are two situations when a region becomes live:

1. When it is not live before a join point in the control flow, but is is live after the join. The join points in the control flow occur at the end of conditional commands and at the entry points of while loops.

2. When the region is not live before an allocation site or a call site which uses that region.

There are three situations when a region ceases to be live:

1. When the region is live before an assignment, but not after, because the assignment updates the last variable whose type contains the region.

2. When a variable goes out of scope, and that variable is the last variable whose type contains the region. Variables go out of scope either at the end of structured scope declarations (`int x; c` or `s x; c`), or before return commands (which represent unstructured forms of scope termination points).

Intuitively, our approach for region command placement models a finite state machine which transitions between two states: one where the region is live, and another where it isn't live. At the beginning of the procedure, the state machine starts in the latter state. Then, during the execution of a program, it transitions between the two states whenever it reaches a point where the region becomes live or ceases to be live. Region creation and removal commands occur only on the transitions between these states.

With this automaton model, it is straightforward to check that the resulting program satisfies all three safety conditions. It is also easy to verify that the algorithm optimally places creation and removal commands, minimizing the region lifetimes under the given safety constraints. Moving a

$$\frac{\mathsf{com} \equiv \mathsf{if}\ (\mathsf{e})\ \mathsf{then}\ \mathsf{com}_1\ \mathsf{else}\ \mathsf{com}_2 \quad \mathsf{r} \notin \mathsf{Live}(\mathsf{com}_1\bullet) \quad \mathsf{r} \in \mathsf{Live}(\mathsf{com}\bullet)}{\mathsf{com}_1\bullet \implies \mathsf{com}_1}$$

$$\frac{\mathsf{com} \equiv \mathsf{if}\ (\mathsf{e})\ \mathsf{then}\ \mathsf{com}_1\ \mathsf{else}\ \mathsf{com}_2 \quad \mathsf{r} \notin \mathsf{Live}(\mathsf{com}_2\bullet) \quad \mathsf{r} \in \mathsf{Live}(\mathsf{com}\bullet)}{\mathsf{com}_2\bullet \implies \mathsf{com}_2}$$

$$\frac{\mathsf{com} \equiv \mathsf{while}\ (\mathsf{e})\ \mathsf{com}_1 \quad \mathsf{r} \notin \mathsf{Live}(\bullet\mathsf{com}) \quad \mathsf{r} \in \mathsf{Live}(\bullet\mathsf{com}_1)}{\bullet\mathsf{com} \implies \mathsf{create}\ \mathsf{r}}$$

$$\frac{\mathsf{com} \equiv \mathsf{while}\ (\mathsf{e})\ \mathsf{com}_1 \quad \mathsf{r} \notin \mathsf{Live}(\mathsf{com}_1\bullet) \quad \mathsf{r} \in \mathsf{Live}(\bullet\mathsf{com}_1)}{\mathsf{com}_1\bullet \implies \mathsf{create}\ \mathsf{r}}$$

$$\frac{\mathsf{com} \equiv \mathsf{x} = \mathsf{new}\ \mathsf{s}\ \mathsf{in}\ \mathsf{r} \quad \mathsf{r} \notin \mathsf{Live}(\bullet\mathsf{com})}{\bullet\mathsf{com} \implies \mathsf{create}\ \mathsf{r}} \qquad \frac{\mathsf{com} \equiv \mathsf{x} = \mathsf{p}\ [\bar{\mathsf{r}}]\ (\mathsf{y}) \quad \mathsf{r_i} \in \bar{\mathsf{r}} \quad \mathsf{r_i} \notin \mathsf{Live}(\bullet\mathsf{com})}{\bullet\mathsf{com} \implies \mathsf{create}\ \mathsf{r_i}}$$

Figure 7: Regions for Region Creation

$$\frac{\mathsf{com} \equiv \mathsf{t}\ \mathsf{x};\ \mathsf{com}' \quad \mathsf{r} \in \mathsf{Live}(\mathsf{com}'\bullet) \quad \mathsf{r} \notin \mathsf{Live}(\mathsf{com}\bullet)}{\mathsf{com}\bullet \implies \mathsf{remove}\ \mathsf{r}}$$

$$\frac{\mathsf{r} \in \mathsf{Live}(\bullet\mathsf{com}) \quad \mathsf{r} \notin \mathsf{Live}(\mathsf{com}\bullet)}{\mathsf{com}\bullet \implies \mathsf{remove}\ \mathsf{r}} \qquad \frac{\mathsf{com} \equiv \mathsf{return}\ \mathsf{x}; \quad \mathsf{r} \in \mathsf{Live}(\bullet\mathsf{com})}{\bullet\mathsf{com} \implies \mathsf{remove}\ \mathsf{r}}$$

Figure 8: Rules for Region Removal

region creation later or a region removal earlier immediately causes the program to use a region at a point where it hasn't been created yet, or has already been removed.

Figure 8 formalizes the region placement algorithm using relations of the form $\mathsf{pp} \implies \mathsf{create\ r}$ and $\mathsf{pp} \implies \mathsf{remove\ r}$, indicating that the algorithm inserts a region creation or removal command at program point pp. Given a command com, we denote by $\bullet\mathsf{com}$ the program point before the command, and by $\mathsf{com}\bullet$ the program point after the command. With the exception of the rule for local declarations (first removal rule), all of the commands com represent assignments. All the rules refer only to local regions $\mathsf{r} \notin \mathsf{R_p}$, since these are the only ones for which we insert such commands. For a program point pp, we denote by $\mathsf{Live(pp)}$ the set of live regions at this point, which includes all the regions that occur in the computed type environment at this program point.

5.4 Soundness

We briefly summarize the formal soundness properties of our algorithm. The safety condition is based on consistency relations between stacks, heaps, regions, values, and type information. Informally, a stack and heap in the output program is consistent with a stack and heap in the input program, written $(S_o, H_o) \sim (S_i, H_i)$, if there is a homomorphism between them. A value in the output program is consistent with a value in the input program, written $v_o \sim v_i$, if they are identical, except for the region information. A consistency relation $(S_o, H_o, R) \sim \Gamma(\mathsf{pp})$ holds if the stack and heap structure (S_o, H_o) agrees with the type information $\Gamma(\mathsf{pp})$ and if $R = \mathsf{Live}(\Gamma(\mathsf{pp}))$. Finally, a consistency relation $(v_o, H_o) \sim \Gamma(\mathsf{pp})$ holds if the structure of the portion of heap H_o reachable from v_o is consistent with the type information. Given the type information Γ computed by our algorithm, the following two properties hold:

Progress. Assuming consistent starting states, if the input command doesn't get stuck, then neither does the transformed command.

$$\begin{array}{ll} & \mathsf{c_o} = \mathcal{T}[\![\mathsf{c_i}]\!] \wedge \\ & \langle S_i, H_i, \mathsf{c_i}\rangle \rightarrow \langle S_i', H_i', v_i\rangle \wedge \\ & (S_o, H_o) \sim (S_i, H_i) \ \wedge \ (S_o, H_o) \sim \Gamma(\bullet\mathsf{c_i}) \\ \Rightarrow & \langle S_o, H_o, R, \mathsf{c_o}\rangle \rightarrow \langle S_o', H_o', R', v_o\rangle \end{array}$$

Preservation. If the execution of the input and output commands starts with consistent states which agree with the type information, then states after the execution of the commands are also consistent, and agree with the type information :

$$\begin{array}{ll} & \mathsf{c_o} = \mathcal{T}[\![\mathsf{c_i}]\!] \wedge \\ & \langle S_i, H_i, \mathsf{c_i}\rangle \rightarrow \langle S_i', H_i', v_i\rangle \wedge \\ & \langle S_o, H_o, R, \mathsf{c_o}\rangle \rightarrow \langle S_o', H_o', R', v_o\rangle \wedge \\ & (S_o, H_o) \sim (S_i, H_i) \ \wedge \ (S_o, H_o) \sim \Gamma(\bullet\mathsf{c_i}) \\ \\ \Rightarrow & (S_o', H_o') \sim (S_i', H_i') \ \wedge \ v_o \sim v_i \\ & (S_o', H_o', R') \sim \Gamma(\mathsf{c_i}\bullet) \ \text{ if } v_i = \mathsf{nr} \\ & (v_o, H_o') \sim \Gamma(\bullet\mathsf{c_i}) \qquad \text{if } v_i \neq \mathsf{nr} \end{array}$$

Appendix B provides a detailed definition of the consistency relations and gives full formal proofs for these results.

6 Extensions

This section presents extensions that improve the accuracy of the basic region inference algorithm.

6.1 Interprocedural Creation and Removal

Even though region lifetimes in our algorithm are not lexically scoped and don't follow a stack discipline, the region creation and removal commands are always placed in the same procedure. The reason for this is to keep the basic algorithm modular.

A simple extension is to relax this condition and perform interprocedural placement of region creation and removal commands. The downside is the loss of modularity in the algorithm. We can extend the algorithm with interprocedural computation of region liveness analysis as follows. A backwards dataflow analysis can determine when region parameters are always dead at the end of the procedure. It can then use this information to perform region removal in the invoked procedure. Another analysis can determine when region parameters are never used before the function is invoked. In that case, they can be treated as not defined at procedure entry.

The overall algorithm executes as follows. It first performs the flow analysis the same way as before. It then applies the interprocedural region liveness algorithms. Once region liveness is available at procedure entry and exit points, it incorporates it in the Live sets of our analyses and performs the placement of region commands according to the rules from Section 5.3.2.

6.2 Object migration

Another extension to the region inference algorithm is to support migration of objects between regions. We want to allow individual objects to move from a region $\mathsf{r_1}$ to a region $\mathsf{r_2}$, if we are guaranteed that the object was completely disconnected from the region $\mathsf{r_1}$ before moving it to the region $\mathsf{r_2}$. To support this feature we extend the output language with an additional command:

$$\mathsf{com} ::= \ldots \mid \mathsf{move\ x\ r_1\ r_2}$$

This extension requires a precise analysis information about when individual object become completely disconnected from the rest of the heap. This kind of information can be provided, for instance, by existing pointer and shape analysis algorithms. We just assume an interface of such an analysis, which provides a function $\mathsf{D} : \mathsf{PP} \to \mathcal{P}(\mathsf{V})$, that, given program point $\mathsf{pp} \in \mathsf{PP}$, returns the set of variables that point to individual objects, unaliased and disconnected in the heap. Here V is the set of program variables.

The object migration analysis requires only two simple extensions to the basic algorithm. The first change is that the analysis inserts the move instructions in the program before running the region inference algorithm. It determines the program points where to insert these instructions using the available shape or points-to information in D. Object migration situations can only arise for statements of the form x.f = expr, which may disconnect the object from one region and then connect it to another. When expr is null, the assignment only disconnects the object. Since the analysis must identify the point when objects become disconnected, we assume that the program is in a form such that each assignment x.f = expr is preceded by a nullification assignment x.f = null. The analysis uses the rewriting rule below to place migration commands when objects become disconnected:

$$\frac{\mathsf{com} \equiv \mathsf{x.f} = \mathsf{null} \quad \mathsf{x} \notin \mathsf{D}(\bullet\mathsf{com}) \quad \mathsf{x} \in \mathsf{D}(\mathsf{com}\bullet) \quad \rho_0, \rho_1 \text{ fresh}}{\mathsf{com} \implies \mathsf{com}\,;\, \mathsf{move}\ \mathsf{x}\ \rho_0\ \rho_1}$$

The analysis then runs the basic region inference algorithm on the resulting resulting program, using the analysis rules from Section 5.2. For analyzing move commands, it uses the following rule:

$$\frac{\Gamma \vdash \mathsf{x} : \mathsf{ref}\ [\rho', \overline{\rho}]\ (\overline{\tau}\overline{\mathsf{f}}) \quad \tau' = \mathsf{ref}\ [\rho_1, \overline{\rho}]\ (\overline{\tau}\overline{\mathsf{f}})}{\Delta, \Gamma \vdash \mathsf{move}\ \mathsf{x}\ \rho_0\ \rho_1, \Gamma[\mathsf{x} \mapsto \tau'], \{\rho' = \rho_0\}}$$

The object migration extension requires no other modification of the algorithm.

Figure 9 shows an example of object migration. The program creates two lists a and b, then moves an element from list a to list b. One can think of this example as modeling a process scheduler that manages processes using lists; this operation can be thought as moving a process from the running queue to the blocked queue of the scheduler.

The first column shows the original code before performing any transformation, the second column shows the result using the original region inference algorithm, and the third shows the resulting code after the migration extension. This example shows that the original algorithm cannot support object migration, and requires the two lists to be allocated in the same region. This is a result of not being able to perform strong type updates during the flow analysis of field assignments. The object migration extension essentially uses precise shape analysis in the premises of the analysis rules to enable strong type updates even for heap locations, whenever can determine that it is safe to do so.

7 Related Work

The algorithm presented in this paper is closely related to the region inference algorithm of Tofte and Talpin [15, 3]. They present a region inference algorithm for a simply typed lambda calculus, which translates expressions in the input program into region-annotated terms in the output program. The regions in their output language are lexically scoped, which imposes a stack discipline on the lifetimes of regions. In an extended version of their algorithm, they propose additional constructs such as region resetting , which alleviate the shortcomings due to the stack discipline constraints on region lifetimes. Further extensions of their framework for ML references include analysis rules similar to our typing rules for destructive updates in our algorithm.

Original Code	Region inference	Migration extension

record List = (int i, List n)
 	
int main(int x) {
  List a,b,t;
 	
  a = makeList(x);
 	
  b = makeList(x);
  t = a;
  a = t.n;
  t.n = null;
 	
  t.n = b;
  b = t;
 	
  return x
}
	record List[rl] = (int i, List[rl] n)
	
	int main(int x) {
	  List a,b,t;
	
	  create r1;
	  a = makeList [r1] (x);
	
	  b = makeList [r1] (x);
	  t = a;
	  a = t.n;
	  t.n = null;
	
	  // at this point the algorithm
	  // unifies the regions of a and b
	
	  t.n = b;
	  b = t;
	
	  remove r1;
	  return x
	}
		record List[rl] = (int i, List[rl] n)
		
		int main(int x) {
		  List a,b,t;
		
		  create r1;
		  a = makeList [r1] (x);
		  create r2;
		  b = makeList [r2] (x);
		  t = a;
		  a = t.n;
		  t.n = null;
		
		  move t r1 r2;
		
		  t.n = b;
		  b = t;
		  remove r1;
		  remove r2;
		  return x
		}

Figure 9: Example of extending the region inference algorithm with object migration.

Aiken, Fahndrich, and Levien [1] build on the Tofte/Talpin algorithm and extend it with separate region allocation and deallocation constructs. Object lifetimes are unrestricted and need not follow a stack discipline. The placement of region allocation and deallocation constructs is determined by building and solving a system of constraints which impose that regions are allocated when they are used. The separation of allocation and deallocation points yields significant improvements in performance compared to the Tofte/Talpin algorithm. Henglein, Makholm, and Niss [11] present a region inference system whose output language extends the core lambda calculus with imperative commands for region manipulation. They also decouple region allocation and deallocation. Moreover, they propose commands for region aliases and region assignments.

Motivated by the success of the region-inference work, researchers have recently investigated the alternative of providing language and compiler support for regions. In contrast to the research on region inference, this work has focused more on imperative languages. The RC system [7, 8] provides run-time support for C programs, using library functions for allocation and deallocation of regions, and allocation of objects in specific regions. The system uses reference counts to identify unsafe region deallocations. The Cyclone language [10], a memory-safe version of the C language, uses a more sophisticated region system, providing a full featured type system and run-time support for regions. The system statically checks region annotations and statically ensures safe region deallocation. We believe that our region inference algorithm is a good match for automatically providing region annotations in such systems.

Another related area of research is that of statically inferring object lifetimes. Early work in this direction [13] proposed intraprocedural and interprocedural dataflow analyses. More recent work is based on escape analysis analysis to determine object lifetimes and enable stack allocation for objects whenever objects do not escape from the current procedure [4, 6, 16, 5, 9]. Although they have the similar goals ours, these algorithms are significantly more complex than the unification-based allocation algorithm in this paper. Moreover, the region polymorphic types

of procedures allow our algorithm to characterize object lifetimes even when they escape from the procedure that created them, which the escape-based algorithms don't. Lattner and Adve [12] propose an algorithm for pool allocation of objects. Their approach is to inline procedures to duplicate allocation sites into their callers. Duplicated allocation sites will then allocate objects in different in regions. This approach, however, doesn't work well in the presence of recursion or for programs with deep call graphs, because of the exponential blow-up in code size (or analysis abstractions). Our region approach is cleaner and more general.

Finally, precise shape analysis algorithms such as [14] can complement our algorithm to improve the analysis results. These algorithms can provide accurate information about when individual objects are being removed from their regions. The results of a combined region inference and shape analysis can then be used in conjunction with run-time systems such as the recently proposed *reaps* [2], which combine region allocation with individual deallocation of objects within each region.

8 Conclusions

In this paper we have presented a region inference framework specifically designed for imperative programs with control flow constructs, dynamic allocation, and destructive updates of variables and heap locations. The basic algorithm in this framework first infers region types for variables, procedures, and dynamic allocation commands using a unification based flow analysis. Then, the analysis uses a one-pass algorithm to place region creation and removal commands in the program. We show that the basic framework can be augmented with more aggressive analyses, such as inter-procedural region liveness analysis, live variable analysis, and shape analysis, to achieve better memory management. Because of its simplicity and flexibility, we believe that this framework is a first step towards the adoption of region inference into existing popular imperative programming languages such as Java or C.

References

[1] A. Aiken, M. Fahndrich, and R. Levien. Better static memory management: Improving region-based analysis of higher-order languages. In *Proceedings of the SIGPLAN '95 Conference on Program Language Design and Implementation*, La Jolla, CA, June 1995.

[2] E. Berger, B. Zorn, and K. McKinley. Reconsidering custom memory allocation. In *Proceedings of the 17th Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, Seattle, WA, November 2002.

[3] L. Birkedal, M. Tofte, and M. Vejlstrup. From region inference to von neumann machines via region representation inference. In *Proceedings of the 23rd Annual ACM Symposium on the Principles of Programming Languages*, St. Petersburg Beach, FL, January 1996.

[4] B. Blanchet. Escape analysis for object oriented languages. Application to Java. In *Proceedings of the 14th Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, Denver, CO, November 1999.

[5] J. Bogda and U. Hoelzle. Removing unnecessary synchronization in Java. In *Proceedings of the 14th Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, Denver, CO, November 1999.

[6] J. Choi, M. Gupta, M. Serrano, V. Sreedhar, and S. Midkiff. Escape analysis for Java. In *Proceedings of the 14th Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, Denver, CO, November 1999.

[7] D. Gay and A. Aiken. Memory management with explicit regions. In *Proceedings of the SIGPLAN '98 Conference on Program Language Design and Implementation*, Montreal, Canada, June 1998.

[8] D. Gay and A. Aiken. Language support for regions. In *Proceedings of the SIGPLAN '01 Conference on Program Language Design and Implementation*, Snowbird, UT, June 2001.

[9] O. Gheorghioiu, A. Salcianu, and M. Rinard. Lifetime analysis of dynamically allocated objects. In *Proceedings of the 30th Annual ACM Symposium on the Principles of Programming Languages*, New Orleans, LA, January 2003.

[10] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney. Region-based memory management in Cyclone. In *Proceedings of the SIGPLAN '02 Conference on Program Language Design and Implementation*, Berlin, Germany, June 2002.

[11] F. Henglein, H. Makholm, and H. Niss. A direct approach to control-flow sensitive region-based memory management. In *Proceedings of the 3rd international ACM SIGPLAN conference on Principles and Practice of Declarative Programming*, Florence, Italy, September 2001.

[12] C. Lattner and V. Adve. Automatic pool allocation for disjoint data structures. In *Proceedings of The Workshop on Memory Systems Performance*, Berlin, Germany, June 2002.

[13] C. Ruggieri and T. Murtagh. Lifetime analysis of dynamically allocated objects. In *Proceedings of the 15th Annual ACM Symposium on the Principles of Programming Languages*, San Diego, CA, January 1988.

[14] M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. In *Proceedings of the 23rd Annual ACM Symposium on the Principles of Programming Languages*, St. Petersburg Beach, FL, January 1996.

[15] M. Tofte and J.-P. Talpin. Implementation of the typed call-by-value lambda-calculus using a stack of regions. In *Proceedings of the 21st Annual ACM Symposium on the Principles of Programming Languages*, Portland, OR, January 1994.

[16] J. Whaley and M. Rinard. Compositional pointer and escape analysis for Java programs. In *Proceedings of the 14th Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, Denver, CO, November 1999.

A Semantics

The following sections present the large-step operational semantics for the input and output languages.

We use the following notation for maps. If M is a map, then $M \cup \{x \mapsto v\}$ represents a map M' which *extends* M with a value for x: $x \notin \mathsf{dom}(M)$, $M'(x) = v$ and $M'(y) = M(x)$, for $y \neq x$. Hence, $\mathsf{dom}(M') = \mathsf{dom}(M) \cup \{x\}$. Furthermore, $M \cup \{x \mapsto v\}$ represents a map M'' which *updates* M with a new value for x: $x \in \mathsf{dom}(M)$, $M''(x) = v$ and $M''(y) = M(x)$, for $y \neq x$. Hence, $\mathsf{dom}(M'') = \mathsf{dom}(M)$.

A.1 Semantics of Input Language

The domains and domain equations for the operational semantics of the input language are:

$$\begin{array}{rcl} \mathit{Value} & = & \mathit{Integer} + \{\mathsf{null}\} + \mathit{Location} \\ \mathit{Retval} & = & \mathit{Value} + \{\mathsf{nr}\} \\ \mathit{Stack} & = & \mathit{Variable} \to \mathit{Value} \\ \mathit{Heap} & = & \mathit{Location} \to (\mathit{Field} \to \mathit{Value}) \end{array}$$

The value nr indicates that a command doesn't return.

A.1.1 Expressions

The semantics of expressions in the input language is defined using a relation $\langle S, H, \mathsf{expr}\rangle \to v$ in the domain:

$$(\mathit{Stack} \times \mathit{Heap} \times \mathit{Expr}) \to \mathit{Value}$$

The meaning of $\langle S, H, \mathsf{expr}\rangle \to v$ is that, given a stack S and a heap H, expression expr evaluates to value v. We define this evaluation relation using the following rules:

$$\frac{S(\mathsf{x}) = v}{\langle S, H, \mathsf{x}\rangle \to v} \quad \frac{}{\langle S, H, n\rangle \to n} \quad \frac{}{\langle S, H, \mathsf{null}\rangle \to \mathsf{null}}$$

$$\frac{\begin{array}{c}\langle S, H, e_1\rangle \to n_1 \\ \langle S, H, e_2\rangle \to n_2 \\ n = n_1 \oplus n_2\end{array}}{\langle S, H, e_1 \oplus e_2\rangle \to n} \quad \frac{\begin{array}{c}S(\mathsf{x}) = l_1 \\ H(l_1)(\mathsf{f}) = v\end{array}}{\langle S, H, \mathsf{x.f}\rangle \to v}$$

A.1.2 Commands

We express the semantics of commands using a relation $\langle S, H, \mathsf{com}\rangle \to \langle S', H', v_{\mathsf{ret}}\rangle$ in the domain:

$$(\mathit{Stack} \times \mathit{Heap} \times \mathit{Com}) \to (\mathit{Stack} \times \mathit{Heap} \times \mathit{Retval})$$

The meaning of the relation $\langle S, H, \mathsf{com}\rangle \to \langle S', H', v\rangle$ is that, given a stack S and a heap H, the evaluation of command com yields a new stack S', a new heap H' and a result value v. The evaluation rules are:

$$\frac{\langle S, H, \mathsf{e}\rangle \to v}{\langle S, H, \mathsf{x} = \mathsf{e}\rangle \to \langle S[\mathsf{x} \mapsto v], H, \mathsf{nr}\rangle}$$

$$\frac{S(\mathsf{x}) = l \quad \langle S, H, \mathsf{e}\rangle \to v}{\langle S, H, \mathsf{x.f} = \mathsf{e}\rangle \to \langle S, H[l \mapsto H(l)[\mathsf{f} \mapsto v]], \mathsf{nr}\rangle}$$

$$\frac{\mathsf{s} : \mathsf{ref}(\bar{\mathsf{t}}\ \bar{\mathsf{f}}) \quad m = \mathsf{init}(\bar{\mathsf{t}}\ \bar{\mathsf{f}})}{\langle S, H, \mathsf{x} = \mathsf{new\ s}\rangle \to \langle S[\mathsf{x} \mapsto l], H \cup \{l \mapsto m\}, \mathsf{nr}\rangle}$$

$$\frac{S(\mathsf{y}) = v_1 \quad \mathsf{t\ p\ (t\ z)} = \mathsf{c} \quad \langle \{\mathsf{z} \mapsto v_1\}, H, \mathsf{c}\rangle \to \langle S_1, H_1, v\rangle \quad v \neq \mathsf{nr}}{\langle S, H, \mathsf{x} = \mathsf{p(y)}\rangle \to \langle S[x \to v], H_1, \mathsf{nr}\rangle}$$

$$\frac{S(\mathsf{x}) = v}{\langle S, H, \mathsf{return\ x}\rangle \to \langle \emptyset, H, v\rangle}$$

$$\frac{\langle S \cup \{\mathsf{x} \mapsto 0\}, H, \mathsf{c}\rangle \to \langle S' \cup \{\mathsf{x} \mapsto v\}, H', v_\mathsf{r}\rangle}{\langle S, H, \mathsf{int\ x;\ c}\rangle \to \langle S', H', v_\mathsf{r}\rangle}$$

$$\frac{\langle S \cup \{\mathsf{x} \mapsto \mathsf{null}\}, H, \mathsf{c}\rangle \to \langle S' \cup \{x \mapsto v\}, H', v_\mathsf{r}\rangle}{\langle S, H, \mathsf{s\ x;\ c}\rangle \to \langle S', H', v_\mathsf{r}\rangle}$$

$$\frac{\langle S, H, \mathsf{c}_1\rangle \to \langle S_1, H_1, v\rangle \quad v \neq \mathsf{nr}}{\langle S, H, \mathsf{c}_1\ ;\ \mathsf{c}_2\rangle \to \langle S_1, H_1, v\rangle}$$

$$\frac{\langle S, H, \mathsf{c}_1\rangle \to \langle S_1, H_1, \mathsf{nr}\rangle \quad \langle S_1, H_1, \mathsf{c}_2\rangle \to \langle S_2, H_2, v_\mathsf{r}\rangle}{\langle S, H, \mathsf{c}_1\ ;\ \mathsf{c}_2\rangle \to \langle S_2, H_2, v_\mathsf{r}\rangle}$$

$$\frac{\langle S, H, \mathsf{e}\rangle \to v \quad v \notin \{0, \mathsf{null}\} \quad \langle S, H, \mathsf{c}_1\rangle \to \langle S_1, H_1, v_\mathsf{r}\rangle}{\langle S, H, \mathsf{if\ (e)\ then\ c}_1\ \mathsf{else\ c}_2\rangle \to \langle S_1, H_1, v_\mathsf{r}\rangle}$$

$$\frac{\langle S, H, \mathsf{e}\rangle \to v \quad v \in \{0, \mathsf{null}\} \quad \langle S, H, \mathsf{c}_2\rangle \to \langle S_2, H_2, v_\mathsf{r}\rangle}{\langle S, H, \mathsf{if\ (e)\ then\ c}_1\ \mathsf{else\ c}_2\rangle \to \langle S_2, H_2, v_\mathsf{r}\rangle}$$

$$\frac{\langle S, H, \mathsf{e}\rangle \to v \quad v \in \{0, \mathsf{null}\}}{\langle S, H, \mathsf{while\ (e)\ c}\rangle \to \langle S, H, \mathsf{nr}\rangle}$$

$$\frac{\langle S, H, \mathsf{e}\rangle \to v \quad v \notin \{0, \mathsf{null}\} \quad \langle S, H, \mathsf{c}\rangle \to \langle S_1, H_1, v'\rangle \quad v' \neq \mathsf{nr}}{\langle S, H, \mathsf{while\ (e)\ c}\rangle \to \langle S_1, H_1, v'\rangle}$$

$$\frac{\langle S, H, \mathsf{e}\rangle \to v \quad v \notin \{0, \mathsf{null}\} \quad \langle S, H, \mathsf{c}\rangle \to \langle S_1, H_1, \mathsf{nr}\rangle \quad \langle S_1, H_1, \mathsf{while\ (e)\ c}\rangle \to \langle S_2, H_2, v_\mathsf{r}\rangle}{\langle S, H, \mathsf{while\ (e)\ c}\rangle \to \langle S_2, H_2, v_\mathsf{r}\rangle}$$

A.2 Semantics of Output Language

We present only the evaluation rules which involve regions. The others are similar the corresponding rules in the input language. The domains and domain equations for the operational semantics of the output language are:

$$\begin{array}{rcl} \mathit{Value} & = & \mathit{Integer} + \{\mathsf{null}\} + \mathit{Location} \times \mathit{Region} \\ \mathit{Retval} & = & \mathit{Value} + \{\mathsf{nr}\} \\ \mathit{Stack} & = & \mathit{Variable} \to \mathit{Value} \\ \mathit{Heap} & = & \mathit{Location} \to (\mathit{Field} \to \mathit{Value}) \\ \mathit{RegEnv} & = & \mathit{RegVar} \to \mathit{Region} \end{array}$$

Again, the value nr indicates that a command doesn't return.

A.2.1 Expressions

The semantics of expressions in the input language is defined using a relation $\langle S, H, R, \mathsf{expr}\rangle \to v$ in the domain:

$$(\mathit{Stack} \times \mathit{Heap} \times \mathit{RegEnv} \times \mathit{Expr}) \to \mathit{Value}$$

The meaning of $\langle S, H, R, \mathsf{expr}\rangle \to v$ is that, given a stack S, a heap H, and a region environment R, expression expr evaluates to value v. The rule for field access expressions is:

$$\frac{S(\mathsf{x}) = (l, \rho) \quad \rho \in \mathsf{range}(R) \quad H(l)(\mathsf{f}) = v}{\langle S, H, R, \mathsf{x.f}\rangle \to v}$$

A.2.2 Commands

We express the semantics of commands using a relation $\langle S, H, R, \mathsf{com}\rangle \to \langle S', H', R', v\rangle$ in the domain:

$$\begin{array}{l} (\mathit{Stack} \times \mathit{Heap} \times \mathit{RegEnv} \times \mathit{Com}) \to \\ \qquad \to (\mathit{Stack} \times \mathit{Heap} \times \mathit{RegEnv} \times \mathit{Retval}) \end{array}$$

The meaning of the relation $\langle S, H, R, \mathsf{com}\rangle \to \langle S', H', R', v\rangle$ is that, given a stack S, a heap H, and a region environment R, the evaluation of command com yields a new stack S', a new heap H', a new region environment R', and a result value v. Let $\mathsf{range}(R)$ be the range of map R: $\mathsf{range}(R) = \{y \mid \exists x \,.\, R(x) = y\}$. The evaluation rules are:

$$\frac{\rho \notin \mathsf{range}(R)}{\langle S, H, R, \mathsf{create\ r}\rangle \to \langle S, H, R \cup \{\mathsf{r} \mapsto \rho\}, \mathsf{nr}\rangle}$$

$$\frac{S'(x) = \begin{cases} \mathsf{null} & \text{if } \mathsf{reach}(\mathsf{x}, S, H, \rho) \\ S(x) & \text{otherwise} \end{cases}}{\langle S, H, R \cup \{\mathsf{r} \mapsto \rho\}, \mathsf{remove\ r}\rangle \to \langle S', H, R, \mathsf{nr}\rangle}$$

$$\frac{S(\mathsf{x}) = (l, \rho) \quad \rho \in \mathsf{range}(R) \quad \langle S, H, R, \mathsf{e}\rangle \to v}{\langle S, H, R, \mathsf{x.f} = \mathsf{e}\rangle \to \langle S, H[l \mapsto H(l)[\mathsf{f} \mapsto v]], R, \mathsf{nr}\rangle}$$

$$\frac{R(\mathsf{r}) = \rho \quad \mathsf{s} : \mathsf{ref}(\bar{\mathsf{t}}\ \bar{\mathsf{f}}) \quad m = \mathsf{init}(\bar{\mathsf{t}}\ \bar{\mathsf{f}})}{\langle S, H, R, \mathsf{x} = \mathsf{new\ s\ in\ r}\rangle \to \langle S[\mathsf{x} \mapsto (l, \rho)], H \cup \{l \mapsto m\}, R, \mathsf{nr}\rangle}$$

$$\frac{S(\mathsf{y}) = v_1 \quad \mathsf{t}\ \mathsf{p}\,[\overline{\mathsf{r}_\mathsf{p}}]\ (\mathsf{t}\ \mathsf{z}) = \mathsf{c} \quad R' = R\,|_{\bar{\mathsf{r}}} \quad \langle\{\mathsf{z} \mapsto v_1\}, H, R', \mathsf{c}\ [\bar{\mathsf{r}}/\overline{\mathsf{r}_\mathsf{p}}]\rangle \to \langle S_1, H_1, R', v\rangle \quad v \neq \mathsf{nr}}{\langle S, H, R, \mathsf{x} = \mathsf{p}[\bar{\mathsf{r}}]\ (\mathsf{y})\rangle \to \langle S[x \to v], H_1, R, \mathsf{nr}\rangle}$$

The initialization function init yields a map m such that:

$$m(\mathsf{f}) = \begin{cases} 0 & \text{if } \mathsf{int}\ \mathsf{f} \in \bar{\mathsf{t}}\ \bar{\mathsf{f}} \\ \mathsf{null} & \text{if } \mathsf{s}\ \mathsf{f} \in \bar{\mathsf{t}}\ \bar{\mathsf{f}} \end{cases}$$

The $\mathsf{reach}(\mathsf{x}, S, H, \rho)$ predicate indicates if a region ρ is reachable from a variable x:

$$\frac{S(\mathsf{x}) = (l, \rho)}{(l, \rho) \in \mathsf{reachLoc}(S, H)} \qquad \frac{(l, \rho) \in \mathsf{reachLoc}(S, H) \quad H(l)(\mathsf{f}) = (l', \rho')}{(l', \rho') \in \mathsf{reachLoc}(S, H)}$$

$$\mathsf{reachReg}(S, H, \rho) = \exists l\ .\ (l, \rho) \in \mathsf{reachLoc}(S, H)$$

$$\mathsf{reach}(\mathsf{x}, S, H, \rho) = \mathsf{reachReg}(\{\mathsf{x} \mapsto S(\mathsf{x})\}, H, \rho)$$

B Soundness

B.1 Partial Ordering

The partial ordering for types is the subtyping relation:

$$\frac{}{\bot <: \mathsf{ref}\ [\overline{\rho}, \rho_0]\ (\overline{\tau}\ \bar{\mathsf{f}})} \qquad \frac{}{\tau <: \tau}$$

Type environments are ordered pointwise and empty environments are less than any other environments:

$$\frac{\mathsf{dom}(\Gamma) = \mathsf{dom}(\Gamma') \quad \forall \mathsf{x}\ .\ \Gamma(\mathsf{x}) <: \Gamma'(\mathsf{x})}{\Gamma \leq \Gamma'} \qquad \frac{}{\emptyset \leq \Gamma}$$

The new environment computed by the unification function is the meet operator with respect to this ordering. That is, if $(\Gamma, E) = \mathsf{unify}(\Gamma_1, \Gamma_2)$, then $\Gamma = \Gamma_1 \sqcap \Gamma_2$.

B.2 Live Regions

The live regions at a program point represent all of the regions that occur in the type environment at that point. We formally define live regions as follows.

Definition 1 *If τ is a type and $\mathcal{T}$ is a translation of region variables to region names, then the regions of τ are:*

$$\mathsf{Regions}(\mathsf{int}) = \mathsf{Regions}(\mathsf{null}) = \emptyset$$
$$\mathsf{Regions}(\mathsf{ref}\ [\rho_0, \overline{\rho}]\ (\overline{\tau}\ \bar{\mathsf{f}}), \mathcal{T}) = \{\mathcal{T}[\![\rho]\!] \mid \rho \in (\overline{\rho} \cup \{\rho_0\})\}$$

The set of live regions in Γ with respect to $\mathcal{T}$ is:

$$\mathsf{Live}(\Gamma, \mathcal{T}) = \bigcup_{x \in \mathsf{dom}(\Gamma)} \mathsf{Regions}(\Gamma(x), \mathcal{T})$$

With respect to the live sets, the meet operation translates into set union. That is, when $\Gamma = \Gamma_1 \sqcap \Gamma_2$, we have $\mathsf{Live}(\Gamma) = \mathsf{Live}(\Gamma_1) \cup \mathsf{Live}(\Gamma_2)$.

B.3 Unification

Intuitively, the purpose of the unification function unify in the algorithm is to make its arguments comparable in the lattice. We formally define this function, as follows. We first define the function unifyReg which unifies regions and region sets:

$$\frac{\rho_1 \neq \rho_2}{\mathsf{unifyReg}(\rho_1, \rho_2) = \{\rho_1 = \rho_2\}} \qquad \frac{}{\mathsf{unifyReg}(\rho, \rho) = \emptyset}$$

$$\frac{\forall i \;.\; E_i = \mathsf{unifyReg}(\rho_i, \rho_i')}{\mathsf{unifyReg}(\rho_1 \ldots \rho_n \;,\; \rho_1' \ldots \rho_n') = E_1 \cup \ldots \cup E_n}$$

Using this function, type unification is defined as follows:

$$\frac{}{\mathsf{unify}(\mathsf{int}, \mathsf{int}) = (\mathsf{int}, \emptyset)}$$

$$\frac{\tau = \mathsf{ref}\ [\rho_0, \overline{\rho}]\ (\overline{\tau}\ \overline{\mathsf{f}})}{\mathsf{unify}(\tau, \bot) = (\tau, \emptyset)} \qquad \frac{\tau = \mathsf{ref}\ [\rho_0, \overline{\rho}]\ (\overline{\tau}\ \overline{\mathsf{f}})}{\mathsf{unify}(\bot, \tau) = (\tau, \emptyset)}$$

$$\frac{\begin{array}{cc} \tau[\rho_0'/\rho_0, \overline{\rho'}/\overline{\rho}] = \tau' & \\ \tau = \mathsf{ref}\ [\rho_0, \overline{\rho}]\ (\overline{\tau}\ \overline{\mathsf{f}}) & \mathsf{unifyReg}(\overline{\rho}, \overline{\rho'}) = E \\ \tau' = \mathsf{ref}\ [\rho_0', \overline{\rho'}]\ (\overline{\tau'}\ \overline{\mathsf{f}}) & \mathsf{unifyReg}(\rho_0, \rho_0') = E_0 \end{array}}{\mathsf{unify}(\tau, \tau') = (\tau, E_0 \cup E)}$$

$$\frac{\mathsf{unify}(\tau_1, \tau_1') = E_1 \quad \mathsf{unify}(\tau_2, \tau_2') = E_2}{\mathsf{unify}(\tau_1 \to \tau_2, \tau_1' \to \tau_2') = (\tau_1 \to \tau_2, E_1 \cup E_2)}$$

Note that the analysis rules and unification function are such that bottom types ($\bot$) are never being used as arguments to function type constructors or reference type constructors.

B.4 Analysis Results

The region inference algorithm computes type information at each program point. Our type inference rules ensure that the following relations hold between the type environments at different points in the program.

$$\Gamma(\bullet(\mathsf{s}\ \mathsf{x}\,;\,\mathsf{c}_1)) \cup \{\mathsf{x} \mapsto \bot\} = \Gamma(\bullet\mathsf{c}_1) \tag{1}$$
$$\Gamma((\mathsf{s}\ \mathsf{x}\,;\,\mathsf{c}_1)\bullet) \cup \{\mathsf{x} \mapsto \tau\} = \Gamma(\mathsf{c}_1\bullet) \tag{2}$$
$$\Gamma(\bullet(\mathsf{int}\ \mathsf{x}\,;\,\mathsf{c}_1)) \cup \{\mathsf{x} \mapsto \mathsf{int}\} = \Gamma(\bullet\mathsf{c}_1) \tag{3}$$
$$\Gamma((\mathsf{int}\ \mathsf{x}\,;\,\mathsf{c}_1)\bullet) \cup \{\mathsf{x} \mapsto \tau\} = \Gamma(\mathsf{c}_1\bullet) \tag{4}$$

$$\Gamma(\bullet(\mathsf{c}_1\,;\,\mathsf{c}_2)) = \Gamma(\bullet\mathsf{c}_1) \tag{5}$$
$$\Gamma((\mathsf{c}_1\,;\,\mathsf{c}_2)\bullet) = \Gamma(\mathsf{c}_2\bullet) \tag{6}$$
$$\mathsf{c}_1\,;\,\mathsf{c}_2 \;\Rightarrow\; \Gamma(\mathsf{c}_1\bullet) = \Gamma(\bullet\mathsf{c}_2) \tag{7}$$

$$\Gamma((\mathsf{if}\ (\mathsf{expr})\ \mathsf{then}\ \mathsf{c}_1\ \mathsf{else}\ \mathsf{c}_2)\bullet) = \Gamma(\mathsf{c}_1\bullet) \sqcap \Gamma(\mathsf{c}_2\bullet) \tag{8}$$
$$\Gamma(\bullet(\mathsf{if}\ (\mathsf{expr})\ \mathsf{then}\ \mathsf{c}_1\ \mathsf{else}\ \mathsf{c}_2)) = \Gamma(\bullet\mathsf{c}_1) = \Gamma(\bullet\mathsf{c}_2) \tag{9}$$

$$\Gamma(\bullet(\mathsf{while\ (expr)\ c_1})) \sqcap \Gamma(\mathsf{c_1}\bullet) = \Gamma(\bullet\mathsf{c_1}) \quad (10)$$

$$\Gamma((\mathsf{while\ (expr)\ c_1})\bullet) = \Gamma(\bullet\mathsf{c_1}) \quad (11)$$

For assignments and return commands, the analysis result is such that the following relations hold:

$$\mathsf{c} \equiv (\mathsf{x} = \mathsf{e}) \ : \ \Gamma(\bullet\mathsf{c}) \vdash \mathsf{e} : \tau \quad (12)$$

$$\Gamma(\mathsf{c}\bullet) = \Gamma(\bullet\mathsf{c})[x \mapsto \tau] \quad (13)$$

$$\mathsf{c} \equiv (\mathsf{x.f} = \mathsf{e}) \ : \ \Gamma(\bullet\mathsf{c}) \vdash \mathsf{x.f} : \tau \quad (14)$$

$$\Gamma(\bullet\mathsf{c}) \vdash \mathsf{e} : \tau',\ \tau' <: \tau \quad (15)$$

$$\Gamma(\mathsf{c}\bullet) = \Gamma(\bullet\mathsf{c}) \quad (16)$$

$$\mathsf{c} \equiv (\mathsf{x} = \mathsf{a} : \mathsf{new\ s}) \ : \ \Delta \vdash \mathsf{a} : \tau \quad (17)$$

$$\Gamma(\mathsf{c}\bullet) = \Gamma(\bullet\mathsf{c})[x \mapsto \tau] \quad (18)$$

$$\mathsf{c} \equiv (\mathsf{x} = \mathsf{cs} : \mathsf{p(y)}) \ : \ \Delta \vdash \mathsf{cs} : \tau_1 \to \tau_2 \quad (19)$$

$$\Gamma \vdash \mathsf{y} : \tau,\ \tau <: \tau_1 \quad (20)$$

$$\Gamma(\mathsf{c}\bullet) = \Gamma(\bullet\mathsf{c})[\mathsf{x} \mapsto \tau_2] \quad (21)$$

$$\mathsf{c} \equiv (\mathsf{return\ x}) \ : \ \Gamma \vdash \mathsf{ret} : \tau \quad (22)$$

$$\Gamma \vdash \mathsf{x} : \tau',\ \tau' <: \tau \quad (23)$$

$$\Gamma(\mathsf{c}\bullet) = \emptyset \quad (24)$$

B.5 Translation

We give a full definition for the translation that occurs in the two phases of the algorithm. The translation $\mathcal{T}$ uses the initialization environment Δ and the type environment Γ which contains type information at each program point. The translation for each procedure is:

$$\frac{\Delta \vdash \mathsf{p} : \tau_1 \to \tau_2 \quad \mathsf{FV}(\tau_1 \to \tau_2) = \overline{\rho}}{\begin{array}{c}\mathcal{T}[\![\mathsf{t_2\ p(t_1\ z) = com}]\!]\Delta\Gamma = \\ \mathcal{T}[\![\tau_2]\!]\Delta\Gamma\ \mathsf{p}\ [\overline{\rho}]\ (\mathcal{T}[\![\tau_1]\!]\Delta\Gamma\ z) = \mathcal{T}[\![\mathsf{com}]\!]\Delta\Gamma\end{array}} \quad (25)$$

At several points in our translation, the algorithm inserts sequences of create or remove commands. Given a set S of regions, we denote by create S the sequence of region creation commands for each region in S. Similarly, we denote by remove S the sequence of region removal commands for all regions in S. If $\mathsf{S} = \emptyset$, these are empty sequences.

B.5.1 Control Flow Constructs

The translation for declarations and control flow constructs recurses on the structure of commands. At join points in the control flow it may create new regions; when variables go out of scope, the translation may remove existing regions. We define the translation $\mathsf{c_o} = \mathcal{T}[\![\mathsf{c_i}]\!]$of an input command $\mathsf{c_i}$ into an output command $\mathsf{c_o}$ as follows:

$$\frac{\begin{array}{c}\mathcal{T}[\![\mathsf{c_{i1}}]\!]\Delta\Gamma = \mathsf{c_{o1}} \\ \mathsf{R_d} = \mathsf{Live}(\mathsf{c_{i1}}\bullet) - \mathsf{Live}(\mathsf{c_i}\bullet)\end{array}}{\mathcal{T}[\![\mathsf{s\ x\,;\ c_{i1}}]\!]\Delta\Gamma = (\mathsf{s\ x\,;\ c_{o1}})\,;\ \mathsf{remove\ R_d}} \tag{26}$$

$$\frac{\begin{array}{c}\mathcal{T}[\![\mathsf{c_{i1}}]\!]\Delta\Gamma = \mathsf{c_{o1}} \\ \mathsf{R_d} = \mathsf{Live}(\mathsf{c_{i1}}\bullet) - \mathsf{Live}(\mathsf{c_i}\bullet)\end{array}}{\mathcal{T}[\![\mathsf{int\ x\,;\ c_{i1}}]\!]\Delta\Gamma = (\mathsf{int\ x\,;\ c_{o1}})\,;\ \mathsf{remove\ R_d}} \tag{27}$$

$$\frac{\mathcal{T}[\![\mathsf{c_{i1}}]\!]\Delta\Gamma = \mathsf{c_{o1}} \quad \mathcal{T}[\![\mathsf{c_{i2}}]\!]\Delta\Gamma = \mathsf{c_{o2}}}{\mathcal{T}[\![\mathsf{c_{i1}\ ;\ c_{i2}}]\!]\Delta\Gamma = \mathsf{c_{o1}\ ;\ c_{o2}}} \tag{28}$$

$$\frac{\begin{array}{c}\mathcal{T}[\![\mathsf{c_{i1}}]\!]\Delta\Gamma = \mathsf{c_{o1}} \quad \mathcal{T}[\![\mathsf{c_{i2}}]\!]\Delta\Gamma = \mathsf{c_{o2}} \\ \mathsf{C_{c1}} = \mathsf{Live}(\mathsf{c_i}\bullet) - \mathsf{Live}(\mathsf{c_{i1}}\bullet) \\ \mathsf{C_{c2}} = \mathsf{Live}(\mathsf{c_i}\bullet) - \mathsf{Live}(\mathsf{c_{i2}}\bullet)\end{array}}{\begin{array}{l}\mathcal{T}[\![\mathsf{if\ (e)\ then\ c_{i1}\ else\ c_{i2}}]\!]\Delta\Gamma = \\ \quad \mathsf{if\ (e)\ then\ (c_{o1}\,;\ create\ C_{c1})} \\ \quad\quad \mathsf{else\ (c_{o2}\,;\ create\ C_{c2})}\end{array}} \tag{29}$$

$$\frac{\begin{array}{c}\mathcal{T}[\![\mathsf{c_{i1}}]\!]\Delta\Gamma = \mathsf{c_{o1}} \\ \mathsf{C_{w1}} = \mathsf{Live}(\bullet\mathsf{c_{i1}}) - \mathsf{Live}(\bullet\mathsf{c_i}) \\ \mathsf{C_{w2}} = \mathsf{Live}(\bullet\mathsf{c_{i1}}) - \mathsf{Live}(\mathsf{c_{i1}}\bullet)\end{array}}{\begin{array}{l}\mathcal{T}[\![\mathsf{while\ (e)\ c_{i1}}]\!]\Delta\Gamma = \\ \quad \mathsf{create\ C_{w1}\ ;\ while\ (e)\ (c_{o1}\ ;\ create\ C_{w2})}\end{array}} \tag{30}$$

B.5.2 Basic Commands

The translation for basic commands (assignments and return commands) consists of two translations $\mathcal{T}_\mathsf{a}$ and $\mathcal{T}_\mathsf{i}$. The former corresponds to the first phase of the algorithm, which adds region type annotations; the latter corresponds to the second phase, which inserts region creation and removal commands. The translation for basic commands is then the composition of the two:

$$\frac{\mathcal{T}_\mathsf{a}[\![\mathsf{c}]\!]\Delta\Gamma = \mathsf{c}' \quad \mathcal{T}_\mathsf{i}[\![\mathsf{c}']\!]\Delta\Gamma = \mathsf{c}''}{\mathcal{T}[\![\mathsf{c}]\!]\Delta\Gamma = \mathsf{c}''} \tag{31}$$

B.5.3 Region Type Annotations

We first define the translation for insertion of region annotations. Except for allocation commands and procedure calls, this is the identity function.

$$\frac{\mathsf{c} \not\equiv (\mathsf{x} = \mathsf{new\ c}) \quad \mathsf{c} \not\equiv (\mathsf{x} = \mathsf{p(y)})}{\mathcal{T}_\mathsf{a}[\![\mathsf{c}]\!]\Delta\Gamma = \mathsf{c}} \tag{32}$$

$$\frac{\mathsf{c} \equiv (\mathsf{x} = \mathsf{a} : \mathsf{new\ c}) \quad \Delta \vdash \mathsf{a} : \mathsf{ref}[\rho_0, \overline{\rho}]\ (\overline{\tau}\ \overline{\mathsf{f}})}{\mathcal{T}_\mathsf{a}[\![\mathsf{c}]\!]\Delta\Gamma = \ \mathsf{x} = \mathsf{new\ s\ in}\ \mathcal{T}[\![\rho_0]\!]} \tag{33}$$

$$\frac{\begin{array}{c}\mathsf{c} \equiv (\mathsf{x} = \mathsf{c} : \mathsf{p(y)}) \quad \Delta \vdash \mathsf{c} : \tau_1 \to \tau_2 \\ \overline{\rho} = \mathsf{FV}(\tau_1) \cup \mathsf{FV}(\tau_2)\end{array}}{\mathcal{T}_\mathsf{a}[\![\mathsf{x} = \mathsf{p(y)}]\!]\Delta\Gamma = \ \mathsf{x} = \mathsf{p}\ [\overline{\mathcal{T}[\![\rho]\!]}]\ (\mathsf{y})} \tag{34}$$

B.5.4 Region Creation and Removal

The translation $\mathcal{T}_\mathsf{a}$ for inserting region creation and removal commands for basic statements is:

$$\frac{\begin{array}{c}\mathsf{c} \not\equiv (\mathsf{return\ x}) \quad \mathsf{c} \not\equiv (\mathsf{x} = \mathsf{p}[\overline{\mathsf{r}}](\mathsf{y})) \\ \mathsf{C_b} = \mathsf{Live}(\mathsf{c}\bullet) - \mathsf{Live}(\bullet\mathsf{c}) \\ \mathsf{R_b} = \mathsf{Live}(\bullet\mathsf{c}) - \mathsf{Live}(\mathsf{c}\bullet)\end{array}}{\mathcal{T}_\mathsf{i}[\![\mathsf{c}]\!]\Delta\Gamma = \mathsf{create}\ \mathsf{C_b}\ ;\ \mathsf{c}\ ;\ \mathsf{remove}\ \mathsf{R_b}} \tag{35}$$

$$\frac{\begin{array}{c}\mathsf{c} \equiv (\mathsf{x} = \mathsf{p}[\overline{\mathsf{r}}](\mathsf{y})) \\ \mathsf{C_b} = (\mathsf{Live}(\mathsf{c}\bullet) \cup \overline{\mathsf{r}}) - \mathsf{Live}(\bullet\mathsf{c}) \\ \mathsf{R_b} = \mathsf{Live}(\bullet\mathsf{c}) - \mathsf{Live}(\mathsf{c}\bullet)\end{array}}{\mathcal{T}_\mathsf{i}[\![\mathsf{c}]\!]\Delta\Gamma = \mathsf{create}\ \mathsf{C_b}\ ;\ \mathsf{c}\ ;\ \mathsf{remove}\ \mathsf{R_b}} \tag{36}$$

$$\frac{\mathsf{c} \equiv (\mathsf{return\ x}) \quad \mathsf{R_r} = \mathsf{Live}(\bullet\mathsf{c})}{\mathcal{T}_\mathsf{i}[\![\mathsf{c}]\!]\Delta\Gamma = \mathsf{remove}\ \mathsf{R_r}\ ;\ \mathsf{c}} \tag{37}$$

B.6 Consistency

We define the projection of entities from the output domains by removing all of their region information.

Definition 2 *If v is a variable in the output domain then the projection $v\downarrow$ is defined inductively as:*

$$v\downarrow \quad = \quad \begin{cases} v & \text{if } v = n \text{ or } v = \mathsf{null} \text{ or } v = \mathsf{nr} \\ l & \text{if } v = (l, r) \end{cases}$$

If S is a stack and H is a heap in the input domain, their projections $S\downarrow$ and $S\downarrow$ are defined by:

$$\begin{array}{rcl}(S\downarrow)(v) & = & (S(v))\downarrow \\ (H\downarrow)(v)(f) & = & (H(v)(f))\downarrow\end{array}$$

We say that an output state (S_o, H_o) is homomorphic *to an input state (S_i, H_i), and write $(S_o, H_o) \sim (S_i, H_i)$, if : $S_o\downarrow = S_i$ and $H_o\downarrow = H_i$. Similarly, $(S_o, H_o, v_o) \sim (S_i, H_i, v_i)$ if $S_o\downarrow = S_i$, $H_o\downarrow = H_i$, and $v_o\downarrow = v_i$.*

Definition 3 *A configuration (S, H) is consistent with the type environment Γ, relative to a translation $\mathcal{T}$ of region variables to region names, written $(S, H) \sim (\Gamma, \mathcal{T})$, if there exists a mapping μ of locations to reference types and regions $\mu(l) = (\tau, \mathsf{r})$, such that:*

$$\forall l\ .\ \mu(l) = (\mathsf{ref}[\rho_0, \overline{\rho}]\ (\overline{\tau}\ \overline{\mathsf{f}}), \mathsf{r})\ \wedge \mathsf{r} = \mathcal{T}[\![\rho_0]\!] \tag{38}$$

$$S(x) = (l, \mathsf{r}) \quad \Rightarrow \quad x : \tau \in \Gamma \ \wedge \ \mu(l) = (\tau, \mathsf{r}) \tag{39}$$

$$\begin{aligned} H(l')(f) = (l, \mathsf{r}) \quad \Rightarrow \quad & \mu(l') = (\mathsf{ref}[\rho_), \overline{\rho}] \ (\overline{\tau} \ \overline{\mathsf{f}}), \mathsf{r}') \ \wedge \\ & \mu(l) = (\tau, \mathsf{r}) \ \wedge \ \tau \ \mathsf{f} \in \overline{\tau} \ \overline{\mathsf{f}} \end{aligned} \tag{40}$$

We use the shorthand notation $(v_o, H_o) \sim (\Gamma, \mathcal{T})$ for:

$$(\{\mathsf{ret} \mapsto v_o\}, H_o) \sim (\Gamma, \mathcal{T})$$

Definition 4 *We say that a state (S, H, R) in the output domain is consistent with a type environment Γ, relative to $\mathcal{T}$, and write $(S, H, R) \sim (\Gamma, \mathcal{T})$, if:*

$$(S, H) \sim (\Gamma, \mathcal{T}) \tag{41}$$

$$\mathsf{Live}(\Gamma, \mathcal{T}) = R \tag{42}$$

The following properties about the consistency between states and type information easily follow from the above definitions:

Property 1 *If $(S, H) \sim (\Gamma, \mathcal{T})$ then:*

$$\mathsf{reachReg}(S, H, \mathsf{r}) \quad \Rightarrow \quad \mathsf{r} \in \mathsf{Live}(\Gamma, \mathcal{T})$$

Property 2 *If $(S, H, R) \sim (\Gamma, \mathcal{T})$ then:*

$$\mathsf{reachReg}(S, H, \mathsf{r}) \quad \Rightarrow \quad \mathsf{r} \in R$$

Property 3 *If $(S, H, R) \sim (\Gamma, \mathcal{T})$ and $\Gamma \sqsubseteq \Gamma'$, then:*

$$(S, H, R) \sim (\Gamma', \mathcal{T})$$

Property 4 *If $(S, H, R) \sim (\Gamma, \mathcal{T})$, $\langle S, H, R, \mathsf{e}\rangle \to (l, \mathsf{r})$, and $\Gamma \vdash \mathsf{e} : \tau$, then:*

$$\mu(l) = (\tau, \mathsf{r})$$

Property 5 *If $(S, H, R) \sim (\Gamma, \mathcal{T})$ and $\Gamma \vdash \mathsf{e} : \bot$, then:*

$$\langle S, H, R, \mathsf{e}\rangle \to \mathsf{null}$$

B.7 Lemmas for Expressions

Lemma 1 (Expression Preservation) *Given an expression e, we have that:*

$$\langle S_i, H_i, \mathsf{e}\rangle \to v_i \tag{43}$$

$$\langle S_o, H_o, R, \mathsf{e}\rangle \to v_o \tag{44}$$

$$(S_o, H_o) \sim (S_i, H_i) \tag{45}$$

$$\Rightarrow \quad v_o \downarrow = v_i \tag{46}$$

Proof. We prove this lemma by induction on the derivations for $\langle S_i, H_i, \mathsf{e}\rangle \to v_i$. We must examine each rule in the operational semantics for expressions.

Rule for $\mathsf{e} = \mathsf{x}$. Using relations (43) and (44) and the operational semantics of the input and output languages, we have: $S_i(\mathsf{x}) = v_i$ and $S_o(\mathsf{x}) = v_o$. Then relation (45) implies $v_o \downarrow = v_i$.

Rule for $\mathsf{e} = \mathsf{n}$. By the operational semantics of the input and output languages, we have: $v_i = n$ and $v_o = n$. By the definition of projection, $v_o \downarrow = v_i$.

Rule for $\mathsf{e} = \mathsf{null}$. By the operational semantics of the input and output languages, we have: $v_i = \mathsf{null}$ and $v_o = \mathsf{null}$. By the definition of projection, $v_o \downarrow = v_i$.

Rule for $\mathsf{e} = \mathsf{e_1} \oplus \mathsf{e_2}$. Because of relations (43) and (44) and the operational semantics of the input and output languages, we have:

$$\begin{array}{ll} \langle S_i, H_i, e_1 \rangle \to n_1 & \langle S_o, H_o, e_1 \rangle \to n_1' \\ \langle S_i, H_i, e_2 \rangle \to n_2 & \langle S_o, H_o, e_2 \rangle \to n_2' \end{array}$$

where $n_1 \oplus n_2 = n = v_i$ and $n_1' \oplus n_2' = n' = v_o$. Applying the induction hypothesis for e_1 and e_2, we get: $n_1' = n_1$ and $n_2' = n_2$. Hence $n = n'$. Therefore, $v_o \downarrow = v_i$.

Rule for $\mathsf{e} = \mathsf{x.f}$. Using relations (43) and (44) and the operational semantics of the input and output languages, we have:

$$\begin{array}{lll} S_i(\mathsf{x}) = l & H_i(l)(\mathsf{f}) = v_i & \\ S_o(\mathsf{x}) = (l', \mathsf{r}) & H_o(l')(\mathsf{f}) = v_o & \mathsf{r} \in R \end{array}$$

From relation (45) we have $S_o \downarrow = S_i$ and $H_o \downarrow = H_i$. Relation $S_o \downarrow = S_i$ implies $l = l'$. Relation $H_o \downarrow = H_i$ then implies $v_o \downarrow = v_i$. □

Lemma 2 (Expression Progress) *Given an expression* e*, we have that:*

$$\begin{array}{rll} & \langle S_i, H_i, \mathsf{e} \rangle \to v_i & (47) \\ & (S_o, H_o) \sim (S_i, H_i) & (48) \\ & (S_o, H_o, R) \sim (\Gamma, \mathcal{T}) & (49) \\ \Rightarrow & & \\ & \langle S_o, H_o, R, \mathsf{e} \rangle \to v_o & (50) \end{array}$$

Proof. We prove this lemma by induction on the derivations for $\langle S_i, H_i, \mathsf{e} \rangle \to v_i$. We must examine each rule in the operational semantics for expressions.

Rule for $\mathsf{e} = \mathsf{x}$. Because of relation (47), it means that $\mathsf{x} \in \mathsf{dom}(S_i)$. But $(S_o, H_o) \sim (S_i, H_i)$, so $\mathsf{x} \in \mathsf{dom}(S_o)$. Therefore, we can apply the rule for evaluation of variables in the output semantics, and get: $\langle S_o, H_o, R, \mathsf{x} \rangle \to v_o$.

Rules for $\mathsf{e} = \mathsf{n}$ *or* $\mathsf{e} = \mathsf{null}$. The output operational semantics for integers and null are axioms, hence they apply directly: $\langle S_o, H_o, R, n \rangle \to n$ or $\langle S_o, H_o, R, \mathsf{null} \rangle \to \mathsf{null}$.

Rule for $\mathsf{e} = \mathsf{e_1} \oplus \mathsf{e_2}$. Using relation (47) and the operational semantics of the input language, we have:

$$\begin{array}{l} \langle S_i, H_i, e_1 \rangle \to n_1 \\ \langle S_i, H_i, e_2 \rangle \to n_2 \end{array}$$

By induction hypothesis for e_1 and e_2:

$$\begin{array}{l} \langle S_o, H_o, R, e_1 \rangle \to v_1 \\ \langle S_o, H_o, R, e_2 \rangle \to v_2 \end{array}$$

By Expression Preservation Lemma, $v_1 = n_1$ and $v_2 = n_2$. Hence, we can apply the rule for evaluation of arithmetic expressions in the output language and get:

$$\langle S_o, H_o, R, e\rangle \rightarrow n$$

where $n = n_1 \oplus n_2$.

Rule for $\mathsf{e} = \mathsf{x.f}$. Using relation (47) and the operational semantics of the input language, we have:

$$S_i(\mathsf{x}) = l \qquad H_i(l)(\mathsf{f}) = v_i$$

From relation (48) we have $S_o \downarrow = S_i$ and $H_o \downarrow = H_i$. Relation $S_o \downarrow = S_i$ implies $S_o(\mathsf{x}) = (l, \mathsf{r})$. Relation $H_o \downarrow = H_i$ then implies $H_o(l)(\mathsf{f}) = v_o$, with $v_o \downarrow = v_i$. Finally, relation (49) and Property 2 show that $\mathsf{r} \in R$. We can apply the operational semantics for field accesses and evaluate the expression $\mathsf{x.f}$: $\langle S_o, H_o, R, \mathsf{x.f}\rangle \rightarrow v_o$. □

B.8 Preservation Theorem

Theorem 1 (Preservation) *Given a command* $\mathsf{c_i}$ *in the input language, type environments* $\Gamma(\bullet\mathsf{c_i})$ *and* $\Gamma(\mathsf{c_i}\bullet)$ *at program points before and after the command, and a translation* $\mathcal{T}$*, we have that:*

$$\mathsf{c_o} = \mathcal{T}[\![\mathsf{c_i}]\!] \tag{51}$$
$$\langle S_i, H_i, \mathsf{c_i}\rangle \rightarrow \langle S_i', H_i', v_i\rangle \tag{52}$$
$$\langle S_o, H_o, R, \mathsf{c_o}\rangle \rightarrow \langle S_o', H_o', R', v_o\rangle \tag{53}$$
$$(S_o, H_o) \sim (S_i, H_i) \tag{54}$$
$$(S_o, H_o, R) \sim (\Gamma(\bullet\mathsf{c_i}), \mathcal{T}) \tag{55}$$
$$\Rightarrow$$
$$(S_o', H_o', v_o) \sim (S_i', H_i', v_i) \tag{56}$$
$$(S_o', H_o', R') \sim (\Gamma(\mathsf{c_i}\bullet), \mathcal{T}) \quad \textit{if } v_i = \mathsf{nr} \tag{57}$$
$$(v_o, H_o') \sim (\Gamma(\bullet\mathsf{c_i}), \mathcal{T}) \quad \textit{if } v_i \neq \mathsf{nr} \tag{58}$$

Proof. The proof is by induction on the derivations of the evaluation $\langle S_i, H_i, \mathsf{c_i}\rangle \rightarrow \langle S_i', H_i', v_i\rangle$. We examine each of the rules in the operational semantics of the input language. We first prove the cases for declarations and control flow commands, and then prove the cases for basic commands.

Rule for $\mathsf{c_i} = \mathsf{s\ x}$. The translation of $\mathsf{c_i}$ is:

$$\begin{array}{rcl} \mathsf{c_o} & = & (\mathsf{s\ x\,;\ c_{i1}})\,;\ \mathsf{remove\ R_d} \\ \mathsf{R_d} & = & \mathsf{Live(c_{i1}\bullet) - Live(c_i\bullet)} \end{array}$$

Consider input and output states satisfying conditions (52)-(55). From the semantic rule considered in this case, we get:

$$\langle S_i \cup \{\mathsf{x} \mapsto \mathsf{null}\}, H_i, \mathsf{c_{i1}}\rangle \rightarrow \langle S_i' \cup \{\mathsf{x} \mapsto v_{xi}\}, H_i', v_i\rangle$$

From condition (53) and the operational semantics of declarations, sequencing, and region removal in the output language, we have:

$$\langle S_o \cup \{\mathsf{x} \mapsto \mathsf{null}\}, H_o, R, \mathsf{c_{o1}}\rangle \rightarrow \langle S_o'' \cup \{\mathsf{x} \mapsto v_{xo}\}, H_o', R'', v_o\rangle$$
$$R'' - \mathsf{R_d} = R'$$

because the sequence of region removal commands $\mathsf{remove}\ \mathsf{R_d}$ only removes regions from R''. Then, using Property 1 and the fact that $\mathsf{R_d}$ and $\mathsf{Live}(\bullet\mathsf{c_i})$ are disjoint by definition of $\mathsf{R_d}$, we conclude that $\neg\mathsf{reach}(S'_o, H'_o, \mathsf{r})$ for each $\mathsf{r} \in \mathsf{R_d}$. Thus, the removal commands leaves the stack and heap unchanged, $S'_o = S''_o$.

Condition (54) implies:

$$(S_o \cup \{\mathsf{x} \mapsto \mathsf{null}\}, H_o) \sim (S_i \cup \{\mathsf{x} \mapsto \mathsf{null}\}, H_i)$$

Condition (55) implies:

$$(S_o \cup \{\mathsf{x} \mapsto \mathsf{null}\}, H_o, R) \sim (\Gamma(\bullet\mathsf{c_i}) \cup \{\mathsf{x} \mapsto \bot\}, \mathcal{T})$$

Analysis result relation (1) shows that: $\Gamma(\bullet\mathsf{c_i}) \cup \{\mathsf{x} \mapsto \bot\} = \Gamma(\bullet\mathsf{c_{i1}})$. Therefore:

$$(S_o \cup \{\mathsf{x} \mapsto \mathsf{null}\}, H_o, R) \sim (\Gamma(\bullet\mathsf{c_{i1}}), \mathcal{T})$$

All these equations show that conditions (52)-(55) are satisfied for the translation of the subcommand $\mathsf{c_{o1}} = \mathcal{T}[\![\mathsf{c_{i1}}]\!]$, and the derivation $\langle S_i \cup \{\mathsf{x} \mapsto \mathsf{null}\}, H_i, c_{i1}\rangle \to \langle S'_i \cup \{\mathsf{x} \mapsto v_{xi}\}, H'_i, v_i\rangle$ is a sub-derivation of $\langle S_i, H_i, \mathsf{c_i}\rangle \to \langle S'_i, H'_i, v_i\rangle$. By induction hypothesis, we have:

$$\begin{array}{ll}
(S'_o \cup \{\mathsf{x} \mapsto v_{xo}\}, H'_o, v_o) \sim (S'_i \cup \{\mathsf{x} \mapsto v_{xi}\}, H'_i, v_i) & \\
(S'_o \cup \{\mathsf{x} \mapsto v_{xo}\}, H'_o, R'') \sim (\Gamma(\mathsf{c_{i1}}\bullet), \mathcal{T}) & \text{if } v_i = \mathsf{nr} \\
(v_o, H'_o) \sim (\Gamma(\bullet\mathsf{c_{i1}})(\mathsf{ret}), \mathcal{T}) & \text{if } v_i \neq \mathsf{nr}
\end{array}$$

Relations (1) and (2) show that the analysis result is such that: $\Gamma(\bullet\mathsf{c_{i1}})(\mathsf{ret}) = \Gamma(\bullet\mathsf{c_i})(\mathsf{ret})$ and $\Gamma(\mathsf{c_{i1}}\bullet) = \Gamma(\mathsf{c_i}\bullet) \cup \{\mathsf{x} \mapsto \tau\}$. Hence:

$$\begin{array}{ll}
(S'_o \cup \{\mathsf{x} \mapsto v_{xo}\}, H'_o, v_o) \sim (S'_i \cup \{\mathsf{x} \mapsto v_{xi}\}, H'_i, v_i) & \\
(S'_o \cup \{\mathsf{x} \mapsto v_{xo}\}, H'_o, R'') \sim & \\
\qquad \sim (\Gamma(\mathsf{c_i}\bullet) \cup \{\mathsf{x} \mapsto \tau\}, \mathcal{T}) & \text{if } v_i = \mathsf{nr} \\
(v_o, H'_o) \sim (\Gamma(\bullet\mathsf{c_i})(\mathsf{ret}), \mathcal{T}) & \text{if } v_i \neq \mathsf{nr}
\end{array}$$

The last relation above is precisely condition (58). The first relation above implies condition (56):

$$(S'_o, H'_o, v_o) \sim (S'_i, H'_i, v_i)$$

Finally, to show condition (57), we prove that:

$$(S'_o \cup \{\mathsf{x} \mapsto v_{xo}\}, H'_o, R'') \sim (\Gamma(\mathsf{c_i}\bullet) \cup \{\mathsf{x} \mapsto \tau\}, \mathcal{T})$$

implies:

$$(S'_o, H'_o, R') \sim (\Gamma(\mathsf{c_i}\bullet), \mathcal{T})$$

For this, we inspect the conditions from the definition of consistency. Consistency condition (41) immediately follow from the former consistency relation because the stacks and the type environments in the two relations are identical except for x. For condition (42), the former consistency relation shows that: $R'' = \mathsf{Live}(\mathsf{c_{i1}}\bullet)$, since $\Gamma(\mathsf{c_{i1}}\bullet) = \Gamma(\mathsf{c_i}\bullet) \cup \{\mathsf{x} \mapsto \tau\}$. Hence:

$$\begin{array}{rcl}
R' & = & R'' - \mathsf{R_d} \\
& = & \mathsf{Live}(\mathsf{c_{i1}}\bullet) - (\mathsf{Live}(\mathsf{c_{i1}}\bullet) - \mathsf{Live}(\mathsf{c_i}\bullet)) \\
& = & \mathsf{Live}(\mathsf{c_i}\bullet)
\end{array}$$

The above relation holds because $\mathsf{Live}(\mathsf{c_i}\bullet) \subseteq \mathsf{Live}(\mathsf{c_{i1}}\bullet)$. We have therefore proved all the conditions for the consistency relation $(S'_o, H'_o, R') \sim (\Gamma(\mathsf{c_i}\bullet), \mathcal{T})$, which completes the proof in this case.

Rule for $\mathsf{c_i} = \mathsf{int\ x}$. The proof is similar to the case above.

Rule 1 for $\mathsf{c_i} = \mathsf{c_{i1}} ; \mathsf{c_{i2}}$. Let $\mathsf{c_{o1}} = \mathcal{T}[\![\mathsf{c_{i1}}]\!]$ and $\mathsf{c_{o2}} = \mathcal{T}[\![\mathsf{c_{i2}}]\!]$. The translation of $\mathsf{c_i}$ is $\mathsf{c_o} = \mathsf{c_{o1}} ; \mathsf{c_{o2}}$. Consider input and output states satisfying conditions (52)-(55). From the semantic rule considered in this case, we have:

$$\langle S_i, H_i, \mathsf{c_{i1}}\rangle \to \langle S_i', H_i', v_i\rangle \quad v_i \neq \mathsf{nr}$$

From condition (52), one of the two rules for sequencing in the output language applies. Both of these evaluate the first command in the sequence:

$$\langle S_o, H_o, R, \mathsf{c_{o1}}\rangle \to \langle S_o'', H_o'', R'', v_o''\rangle$$

From condition (54), $(S_o, H_o) \sim (S_i, H_i)$. We also have $\Gamma(\bullet\mathsf{c_i}) = \Gamma(\bullet\mathsf{c_{i1}})$, from (5), so $(S_o, H_o, R) \sim (\Gamma(\bullet\mathsf{c_{i1}}), \mathcal{T})$. All conditions (52)-(55) are satisfied for the translation of $\mathsf{c_{i1}}$, with the evaluation $\langle S_i, H_i, \mathsf{c_{i1}}\rangle \to \langle S_i'', H_i'', \mathsf{nr}\rangle$. Thus, we can apply the induction hypothesis for the translation of $\mathsf{c_{i1}}$. By induction hypothesis, we get $v_o'' \downarrow = \mathsf{nr}$, so $v_o'' = \mathsf{nr}$. Therefore, we must also use the first sequencing rule in the output semantics:

$$\langle S_o, H_o, R, \mathsf{c_{o1}}\rangle \to \langle S_o', H_o', R', v_o\rangle$$

Because the evaluation of $\mathsf{c_{i1}}$ returns, we directly use the conclusion of relation (58). By induction hypothesis for the translation of $\mathsf{c_{i1}}$, we have:

$$(S_o', H_o', v_o) \sim (S_i', H_i', v_i)$$
$$(v_o, H_o') \sim (\Gamma(\bullet\mathsf{c_{i1}}), \mathcal{T})$$

This completes the proof in this case, since $\Gamma(\bullet\mathsf{c_i}) = \Gamma(\bullet\mathsf{c_{i1}})$, by relation (5), and the type of ret is the same at all program points: $\Gamma(\mathsf{c_i}\bullet) = \Gamma(\mathsf{c_{i2}}\bullet)$.

Rule 2 for $\mathsf{c_i} = \mathsf{c_{i1}} ; \mathsf{c_{i2}}$. Let $\mathsf{c_{o1}} = \mathcal{T}[\![\mathsf{c_{i1}}]\!]$ and $\mathsf{c_{o2}} = \mathcal{T}[\![\mathsf{c_{i2}}]\!]$. The translation of $\mathsf{c_i}$ is $\mathsf{c_o} = \mathsf{c_{o1}} ; \mathsf{c_{o2}}$. Consider input and output states satisfying conditions (52)-(55). From the semantic rule considered in this case, we have:

$$\langle S_i, H_i, \mathsf{c_{i1}}\rangle \to \langle S_i'', H_i'', \mathsf{nr}\rangle$$
$$\langle S_i'', H_i'', \mathsf{c_{i2}}\rangle \to \langle S_i', H_i', v_i\rangle$$

From condition (52), one of the two rules for sequencing in the output language applies. Both of these evaluate the first command in the sequence:

$$\langle S_o, H_o, R, \mathsf{c_{o1}}\rangle \to \langle S_o'', H_o'', R'', v_o''\rangle$$

From condition (54), $(S_o, H_o) \sim (S_i, H_i)$. We also have $\Gamma(\bullet\mathsf{c_i}) = \Gamma(\bullet\mathsf{c_{i1}})$, from (5), so $(S_o, H_o, R) \sim (\Gamma(\bullet\mathsf{c_{i1}}), \mathcal{T})$.

All conditions (52)-(55) are satisfied for the translation of $\mathsf{c_{i1}}$, with the evaluation $\langle S_i, H_i, \mathsf{c_{i1}}\rangle \to \langle S_i'', H_i'', \mathsf{nr}\rangle$. Thus, we can apply the induction hypothesis for the translation of $\mathsf{c_{i1}}$. By induction hypothesis, we get $v_o'' \downarrow = \mathsf{nr}$, so $v_o'' = \mathsf{nr}$. Therefore, we must also use the second sequencing rule in the output semantics:

$$\langle S_o'', H_o'', R'', \mathsf{c_{i2}}\rangle \to \langle S_o', H_o', R', v_o\rangle$$

Because the evaluation of $\mathsf{c_{i1}}$ doesn't return, we directly use the conclusion of relation (57). By induction hypothesis for the translation of $\mathsf{c_{i1}}$, we have:

$$(S_o'', H_o'', \mathsf{nr}) \sim (S_i'', H_i'', \mathsf{nr})$$
$$(S_o'', H_o'', R'') \sim (\Gamma(\mathsf{c_{i1}}\bullet), \mathcal{T})$$

The analysis result is such that $\Gamma(\mathsf{c_{i1}}\bullet) = \Gamma(\bullet\mathsf{c_{i2}})$, by relation (7). Thus, all conditions (52)-(55) are satisfied for the translation of $\mathsf{c_{i2}}$, with the evaluation $\langle S_i'', H_i'', \mathsf{c_{i2}}\rangle \to \langle S_i', H_i', v_i\rangle$. By induction hypothesis we get:

$$\begin{array}{ll}(S_o', H_o', v_o) \sim (S_i', H_i', v_i) & \\ (S_o', H_o', R') \sim (\Gamma(\mathsf{c_{i2}}\bullet), \mathcal{T}) & \text{if } v_i = \mathsf{nr} \\ (v_o, H_o') \sim (\Gamma(\bullet\mathsf{c_{i2}})(\mathsf{ret}), \mathcal{T}) & \text{if } v_i \neq \mathsf{nr}\end{array}$$

This completes the proof in this case, since $\Gamma(\mathsf{c_i}\bullet) = \Gamma(\mathsf{c_{i2}}\bullet)$, by relation (6), and the type of ret is the same at all program points: $\Gamma(\mathsf{c_i}\bullet) = \Gamma(\mathsf{c_{i2}}\bullet)$.

Rule 1 for $\mathsf{c_i}$ = if (e) then $\mathsf{c_{i1}}$ else $\mathsf{c_{i2}}$. This rule evaluates the true branch $\mathsf{c_{i1}}$ of the command. Let $\mathsf{c_{o1}} = \mathcal{T}[\![\mathsf{c_{i1}}]\!]$ and $\mathsf{c_{o2}} = \mathcal{T}[\![\mathsf{c_{i2}}]\!]$. The translation of $\mathsf{c_i}$ is:

$$\mathsf{c_o} \;=\; \mathsf{if\ (e)\ then\ (c_{o1}\,;\ create\ C_{c1})\ else\ (c_{o2}\,;\ create\ C_{c2})}$$

$$\begin{array}{rrcl}\text{where:} & \mathsf{C_{c1}} & = & \mathsf{Live(c_i\bullet) - Live(c_{i1}\bullet)} \\ & \mathsf{C_{c2}} & = & \mathsf{Live(c_i\bullet) - Live(c_{i2}\bullet)}\end{array}$$

Consider input and output states satisfying conditions (52)-(55). From the semantic rule considered in this case, we have:

$$\begin{array}{l}\langle S_i, H_i, \mathsf{e}\rangle \to v \notin \{0, \mathsf{null}\} \\ \langle S_i, H_i, \mathsf{c_{i1}}\rangle \to \langle S_i', H_i', v_i\rangle\end{array}$$

From condition (53) and the operational semantics of the output language, e must evaluate to a value (since both rules have the evaluation of e in the premise):

$$\langle S_o, H_o, R, \mathsf{e}\rangle \to v'$$

Because consistency relation (54), we can apply the Expression Preservation Lemma, and get that $v' \downarrow = v$. But $v \notin \{0, \mathsf{null}\}$. Hence, $v' \notin \{0, \mathsf{null}\}$. Hence, the first rule for conditionals in the output semantics must apply:

$$\langle S_o, H_o, R, \mathsf{c_{o1}\,;\ create\ C_{c1}}\rangle \to \langle S_o', H_o', R', v_o\rangle$$

Therefore:

$$\begin{array}{l}\langle S_o, H_o, R, \mathsf{c_{o1}}\rangle \to \langle S_o', H_o', R'', v_o\rangle \\ R'' \cup \mathsf{C_{c1}} = R'\end{array}$$

By condition (54): $(S_o, H_o) \sim (S_i, H_i)$. By condition (55) and analysis result relation (9): $(S_o, H_o, R) \sim (\Gamma(\bullet\mathsf{c_{i1}}), \mathcal{T})$.

All conditions (52)-(55) are satisfied for the translation of $\mathsf{c_{i1}}$, with the evaluation $\langle S_i, H_i, \mathsf{c_{i1}}\rangle \to \langle S_i', H_i', v_i\rangle$. By induction hypothesis we get:

$$\begin{array}{ll}(S_o', H_o', v_o) \sim (S_i', H_i', v_i) & \\ (S_o', H_o', R'') \sim (\Gamma(\mathsf{c_{i1}}\bullet), \mathcal{T}) & \text{if } v_i = \mathsf{nr} \\ (v_o, H_o') \sim (\Gamma(\bullet\mathsf{c_{i1}})(\mathsf{ret}), \mathcal{T}) & \text{if } v_i \neq \mathsf{nr}\end{array}$$

The first relation above is precisely relation (56). The last relation above implies relation (58), since the return type is the same at all program points. Finally, to prove relation (57), we show that:

$$(S_o', H_o', R'') \sim (\Gamma(\mathsf{c_{i1}}\bullet), \mathcal{T})$$

implies:

$$(S_o', H_o', R') \sim (\Gamma(\mathsf{c_i}\bullet), \mathcal{T})$$

For this, we inspect the conditions from the definition of consistency. By the analysis result relation (8) we have that $\Gamma(\mathsf{c_{i1}}\bullet) \sqsubseteq \Gamma(\mathsf{c_i}\bullet)$. By Property 3, consistency condition (41) holds at program point $\mathsf{c_i}\bullet$. For condition (42), the consistency relation $(S'_o, H'_o, R'') \sim (\Gamma(\mathsf{c_{i1}}\bullet), \mathcal{T})$ shows that: $R'' = \mathsf{Live}(\mathsf{c_{i1}}\bullet)$. Hence:

$$\begin{array}{rcl} R' & = & R'' \cup \mathsf{C_{c1}} \\ & = & \mathsf{Live}(\mathsf{c_{i1}}\bullet) \cup (\mathsf{Live}(\mathsf{c_i}\bullet) - \mathsf{Live}(\mathsf{c_{i1}}\bullet)) \\ & = & \mathsf{Live}(\mathsf{c_i}\bullet) \end{array}$$

The above relation holds because $\mathsf{Live}(\mathsf{c_{i1}}\bullet) \subseteq \mathsf{Live}(\mathsf{c_i}\bullet)$. We have therefore proved all the conditions for the consistency relation $(S'_o, H'_o, R') \sim (\Gamma(\mathsf{c_i}\bullet), \mathcal{T})$, which completes the proof in this case.

Rule 2 for $\mathsf{c_i}$ = if (e) then $\mathsf{c_{i1}}$ else $\mathsf{c_{i2}}$. This rule evaluates the false branch $\mathsf{c_{i2}}$ of the command and the proof is similar to the previous case.

Rules for $\mathsf{c_i}$ = while (e) $\mathsf{c_{i1}}$. We consider all of the three rules for the evaluation of while loops. The conclusion of all of these rules is:

$$(S_i, H_i, \mathsf{while\ (e)\ c_{i1}}) \rightarrow (S'_i, H'_i, v_i)$$

Let $\mathsf{c_{o1}} = \mathcal{T}[\![\mathsf{c_{i1}}]\!]$. The translation of $\mathsf{c_i}$ is:

$$\mathsf{c_o} \quad = \quad \mathsf{create\ C_{w1}\,;\ while\ (e)\ (c_{o1}\,;\ create\ C_{w2})}$$

$$\begin{array}{rrcl} \text{where:} & \mathsf{C_{w1}} & = & \mathsf{Live}(\bullet\mathsf{c_{i1}}) - \mathsf{Live}(\bullet\mathsf{c_i}) \\ & \mathsf{C_{w2}} & = & \mathsf{Live}(\bullet\mathsf{c_{i1}}) - \mathsf{Live}(\mathsf{c_{i1}}\bullet) \end{array}$$

Consider input and output states satisfying conditions (52)-(55). From condition (53), the output command first executes the region creation commands $\mathsf{create\ C_{w1}}$, and then evaluates the loop. If we denote by $\mathsf{c'_{o1}} = (\mathsf{c_{o1}}\,;\ \mathsf{create\ C_{w2}})$, we have:

$$\begin{array}{l} \langle S_o, H_o, R, \mathsf{create\ C_{w1}} \rangle \rightarrow \langle S_o, H_o, R_{\mathsf{w1}}, \mathsf{nr} \rangle \\ \langle S_o, H_o, R_{\mathsf{w1}}, \mathsf{while\ (e)\ c'_{o1}} \rangle \rightarrow \langle S'_o, H'_o, R', v_o \rangle \\ \text{where: } R_{\mathsf{w1}} = R \cup \mathsf{C_{w1}} \end{array}$$

By induction hypothesis, each sub-derivation of the evaluation $\langle S_i, H_i, \mathsf{c_i} \rangle \rightarrow \langle S'_i, H'_i, v_i \rangle$ satisfies the implication in the theorem. These sub-derivations include the evaluation of $\mathsf{c_{i1}}$ in the second and the third rule for loops. Given this fact, the following property holds:

Property 6 *Consider the commands $\mathsf{c_{i1}}$ and $\mathsf{c'_{o1}}$ in the current case. Let (S_{pi}, H_{pi}), (S'_{pi}, H'_{pi}, v_i) (S_{po}, H_{po}, R_p), $(S'_{po}, H'_{po}, R'_p, v_o)$ be states and values such that:*

$$\langle S_{pi}, H_{pi}, \mathit{while}\ (\mathsf{e})\ \mathsf{c_{i1}} \rangle \rightarrow \langle S'_{pi}, H'_{pi}, v_{pi} \rangle \tag{59}$$
$$\langle S_{po}, H_{po}, R_p, \mathit{while}\ (\mathsf{e})\ \mathsf{c'_{o1}} \rangle \rightarrow \langle S'_{po}, H'_{po}, R'_p, v_{po} \rangle \tag{60}$$
$$(S_{po}, H_{po}) \sim (S_i, H_i) \tag{61}$$
$$(S_{po}, H_{po}, R_p) \sim (\Gamma(\bullet\mathsf{c_{i1}}), \mathcal{T}) \tag{62}$$
$$\tag{63}$$

and such that the derivation for:

$$(S_{pi}, H_{pi}, \mathit{while}\ (\mathsf{e})\ \mathsf{c_{i1}}) \rightarrow (S'_{pi}, H'_{pi}, v_{pi})$$

is a sub-derivation of, or is the same as, the derivation for:

$$(S_i, H_i, \mathit{while}\ (\mathsf{e})\ \mathsf{c_{i1}}) \rightarrow (S'_i, H'_i, v_i)$$

Then, the following relations hold:

$$(S'_{po}, H'_{po}, v_{po}) \sim (S'_i, H'_i, v_i) \tag{64}$$
$$(S'_{po}, H'_{po}, R'_p) \sim (\Gamma(\bullet\mathsf{c_{i1}}), \mathcal{T}) \quad \mathit{if} \quad v_{pi} = \mathsf{nr} \tag{65}$$
$$(v_{po}, H'_{po}) \sim (\Gamma(\bullet\mathsf{c_{i1}})(\mathsf{ret}), \mathcal{T}) \quad \mathit{if} \quad v_{pi} \neq \mathsf{nr} \tag{66}$$

In other words, if we start with a output state consistent with the type information at the top of the loop body (i.e., program point $\bullet\mathsf{c_{i1}}$), then the execution of the loop preserves this invariants.

Using this result, we can easily prove the relations required for this case, using the following instantiation:

$$\begin{array}{ccccc} (S_{pi}, H_{pi}) = (S_i, H_i) & & (S'_{pi}, H'_{pi}) = (S'_i, H'_i) & & v_{pi} = v_i \\ (S_{po}, H_{po}) = (S_o, H_o) & & (S'_{po}, H'_{po}) = (S'_o, H'_o) & & v_{po} = v_o \\ R_p = R_{\mathsf{w1}} & & R'_p = R' & & \end{array}$$

With these substitutions, relations (59)-(61) immediately follow from (52)-(54). We can also prove relation (62) using condition (55), as follows. We inspect the conditions from the definition of consistency. By the analysis result relation (10) we have that $\Gamma(\bullet\mathsf{c_i}) \sqsubseteq \Gamma(\bullet\mathsf{c_{i1}})$. By Property 3 and relation (55), consistency condition (41) holds at program point $\bullet\mathsf{c_{i1}}$. Finally, consistency condition (42) holds because:

$$\begin{array}{rcll} R_{\mathsf{w1}} & = & R \cup \mathsf{C_{w1}} & \\ & = & \mathsf{Live}(\bullet\mathsf{c_i}) \cup \mathsf{C_{w1}} & \text{by condition (55)} \\ & = & \mathsf{Live}(\bullet\mathsf{c_i}) \cup & \\ & & \quad \cup\ (\mathsf{Live}(\bullet\mathsf{c_{i1}}) - \mathsf{Live}(\bullet\mathsf{c_i})) & \text{by definition of } \mathsf{C_{w1}} \\ & = & \mathsf{Live}(\bullet\mathsf{c_{i1}}) & \mathsf{Live}(\bullet\mathsf{c_i}) \sqsubseteq \mathsf{Live}(\bullet\mathsf{c_{i1}}) \\ & & & \text{by relation (10)} \end{array}$$

We have therefore proved all the conditions for the consistency relation $(S_o, H_o, R_{\mathsf{w1}}) \sim (\Gamma(\bullet\mathsf{c_{i1}}), \mathcal{T})$. Hence, relation (62) also holds. We now apply the above property and get:

$$\begin{array}{ll} (S'_o, H'_o, v_o) \sim (S'_i, H'_i, v_i) & \\ (S'_o, H'_o, R') \sim (\Gamma(\bullet\mathsf{c_{i1}}), \mathcal{T}) & \text{if} \quad v_i = \mathsf{nr} \\ (v_o, H'_o) \sim (\Gamma(\bullet\mathsf{c_{i1}})(\mathsf{ret}), \mathcal{T}) & \text{if} \quad v_i \neq \mathsf{nr} \end{array}$$

Since the analysis result is such that $\Gamma(\bullet\mathsf{c_{i1}}) = \Gamma(\mathsf{c_i}\bullet)$, by relation (11), and $\Gamma(\bullet\mathsf{c_{i1}})(\mathsf{ret}) = \Gamma(\bullet\mathsf{c_i})(\mathsf{ret})$, this proves the relations for this case.

We now prove the property by induction on the derivations of the evaluation $\langle S_{pi}, H_{pi}, \mathsf{while}\ (\mathsf{e})\ \mathsf{c_{i1}}\rangle \rightarrow \langle S'_{pi}, H'_{pi}, v_{pi}\rangle$. Consider input and output states and values satisfying relations (59) - (62). From condition (59), one of the three rules for while loops in the input operational semantics must apply. We consider each of these rules in turn.

Subcase: While Rule 1. Assume the first rule for while loops applies:

$$\begin{array}{l} \langle S_{pi}, H_{pi}, \mathsf{e}\rangle \rightarrow v \quad v \in \{0, \mathsf{null}\} \\ S'_{pi} = S_{pi} \quad H'_{pi} = H_{pi} \quad v_{pi} = \mathsf{nr} \end{array}$$

Next consider the evaluation $\langle S_{po}, H_{po}, R_p, \mathsf{while}\ (\mathsf{e})\ \mathsf{c}'_{\mathsf{o1}}\rangle \rightarrow \langle S'_{po}, H'_{po}, R', v_{po}\rangle$. Because of the evaluation of $\mathsf{while}\ (\mathsf{e})\ \mathsf{c}'_{\mathsf{o1}}$, one of the three rules for while loops in the input operational semantics must apply. Each of these rules evaluates the expression e:

$$\langle S_{po}, H_{po}, R_p, \mathsf{e}\rangle \rightarrow v'$$

By the Expression Preservation Lemma, $v' \downarrow = v$. But $v \in \{0, \mathsf{null}\}$, therefore $v' \in \{0, \mathsf{null}\}$. Hence, the first rule for while loops in the output operational semantics also applies. Therefore:

$$S'_{po} = S_{po} \quad H'_{po} = H_{po} \quad R'_p = R_p \quad v_{po} = \mathsf{nr}$$

From condition (61), we get: $(S'_{po}, H'_{po}, v_{po}) \sim (S'_{pi}, H'_{pi}, v_{pi})$. Finally, because $v_{pi} = \mathsf{nr}$, we need only to check relation (65). This relation immediately follows from (62), because $S'_{po} = S_{po}$, $H'_{po} = H_{po}$, and $R'_p = R_p$. This completes the proof for this subcase.

Subcase: While Rule 2. Assume the second rule for while loops applies:

$$\begin{array}{l}\langle S_{pi}, H_{pi}, \mathsf{e}\rangle \rightarrow v \quad v \notin \{0, \mathsf{null}\}\\ \langle S_{pi}, H_{pi}, \mathsf{c_{i1}}\rangle \rightarrow \langle S_{pi}, H_{pi}, v_{pi}\rangle\\ S'_{pi} = S_{pi} \quad H'_{pi} = H_{pi} \quad v_{pi} \neq \mathsf{nr}\end{array}$$

Next consider the evaluation $\langle S_{po}, H_{po}, R_p, \mathsf{while}\ (\mathsf{e})\ \mathsf{c}'_{\mathsf{o1}}\rangle \rightarrow \langle S'_{po}, H'_{po}, R', v_{po}\rangle$. Because of the evaluation of $\mathsf{while}\ (\mathsf{e})\ \mathsf{c}'_{\mathsf{o1}}$, one of the three rules for while loops in the input operational semantics must apply. Each of these rules evaluates the expression e:

$$\langle S_{po}, H_{po}, R_p, \mathsf{e}\rangle \rightarrow v'$$

By the Expression Preservation Lemma, $v' \downarrow = v$. But $v \notin \{0, \mathsf{null}\}$, therefore $v' \notin \{0, \mathsf{null}\}$. Hence, one of the last two rules for while loops in the output operational semantics applies. Each of these rules evaluates the loop body $\mathsf{c}'_{\mathsf{o1}}$. This first evaluates $\mathsf{c_{o1}}$:

$$\langle S_{po}, H_{po}, R_p, \mathsf{c_{o1}}\rangle \rightarrow \langle S''_{po}, H''_{po}, R''_p, v'_{po}\rangle$$

From condition (61), $(S_{po}, H_{po}) \sim (S_{pi}, H_{pi})$. From condition (62), $(S_{po}, H_{po}, R_p) \sim (\Gamma(\bullet\mathsf{c_{i1}}), \mathcal{T})$. We can therefore apply the *outer induction hypothesis*, for the translation of $\mathsf{c_{i1}}$ with the evaluation $\langle S_{pi}, H_{pi}, \mathsf{c_{i1}}\rangle \rightarrow \langle S_{pi}, H_{pi}, v_{pi}\rangle$, because we know that its derivation is a sub-derivation of $(S_i, H_i, \mathsf{while}\ (\mathsf{e})\ \mathsf{c_{i1}}) \rightarrow (S'_i, H'_i, v_i)$. Since $v_{pi} \neq \mathsf{nr}$, we get:

$$\begin{array}{l}(S''_{po}, H''_{po}, v'_{po}) \sim (S'_{pi}, H'_{pi}, v_{pi})\\ (v'_{po}, H''_{po}) \sim (\Gamma(\bullet\mathsf{c_{i1}})(\mathsf{ret}), \mathcal{T})\end{array}$$

Hence, $v'_{po} \downarrow = v_{pi}$. But $v_{pi} \neq \mathsf{nr}$. So $v'_{po} \neq \mathsf{nr}$. Therefore, the evaluation of the transformed program also uses the second rule for while loops (but from the output operational semantics):

$$S''_{po} = S'_{po} \quad H''_{po} = H'_{po} \quad R''_p = R'_p \quad v'_{po} = v_{po}$$

We then substitute these values in the result of the outer induction hypothesis:

$$\begin{array}{l}(S'_{po}, H'_{po}, v_{po}) \sim (S'_{pi}, H'_{pi}, v_{pi})\\ (v_{po}, H'_{po}) \sim (\Gamma(\bullet\mathsf{c_{i1}})(\mathsf{ret}), \mathcal{T})\end{array}$$

Along with the fact that $v_{po} \neq \mathsf{nr}$, this completes the proof for this subcase.

Subcase: While Rule 3. Assume the third rule for while loops applies:

$$\begin{array}{l}\langle S_{pi}, H_{pi}, \mathsf{e}\rangle \to v \quad v \notin \{0, \mathsf{null}\}\\ \langle S_{pi}, H_{pi}, \mathsf{c_{i1}}\rangle \to \langle S''_{pi}, H''_{pi}, \mathsf{nr}\rangle\\ \langle S''_{pi}, H''_{pi}, \mathsf{c_i}\rangle \to \langle S'_{pi}, H'_{pi}, v_{pi}\rangle\end{array}$$

Next consider the evaluation $\langle S_{po}, H_{po}, R_p, \mathsf{while\ (e)\ c'_{o1}}\rangle \to \langle S'_{po}, H'_{po}, R', v_{po}\rangle$. Because of this evaluation, one of the three rules for while loops in the input operational semantics must apply. Each of these rules evaluates the expression e:

$$\langle S_{po}, H_{po}, R_p, \mathsf{e}\rangle \to v'$$

By the Expression Preservation Lemma, $v' \downarrow = v$. But $v \notin \{0, \mathsf{null}\}$, therefore $v' \notin \{0, \mathsf{null}\}$. Hence, one of the last two rules for while loops in the output operational semantics applies. Each of these rules evaluates the loop body $\mathsf{c'_{o1}}$. This first evaluates $\mathsf{c_{o1}}$:

$$\langle S_{po}, H_{po}, R_p, \mathsf{c_{o1}}\rangle \to \langle S''_{po}, H''_{po}, R''_p, v'_{po}\rangle$$

From condition (61), $(S_o, H_o) \sim (S_i, H_i)$. From condition (62), $(S_{po}, H_{po}, R_p) \sim (\Gamma(\bullet\mathsf{c_{i1}}), \mathcal{T})$. We can apply the *outer induction hypothesis*, for the translation of $\mathsf{c_{i1}}$ with the evaluation $\langle S_{pi}, H_{pi}, \mathsf{c_{i1}}\rangle \to \langle S''_{pi}, H''_{pi}, \mathsf{nr}\rangle$. Using the fact that $\mathsf{c_{i1}}$ yields a nr value, we get:

$$\begin{array}{r}(S''_{po}, H''_{po}, v'_{po}) \sim (S'_{pi}, H'_{pi}, v_{pi})\\ (S''_{po}, H''_{po}, R''_p) \sim (\Gamma(\mathsf{c_{i1}}\bullet), \mathcal{T})\end{array} \tag{67}$$

Hence, $v'_{po} \downarrow = \mathsf{nr}$. So $v'_{po} = \mathsf{nr}$. Therefore, the evaluation of the transformed program uses the third rule for while loops in the output operational semantics:

$$\begin{array}{l}\langle S_{po}, H_{po}, R_p, \mathsf{c'_{o1}}\rangle \to \langle S''_{po}, H''_{po}, R''_p \cup C_{w2}, \mathsf{nr}\rangle\\ \langle S''_{po}, H''_{po}, R''_p \cup C_{w2}, \mathsf{while\ (e)\ c'_{o1}}\rangle \to \langle S'_{po}, H'_{po}, R'_p, v_{po}\rangle\end{array}$$

because the evaluation of $\mathsf{c'_{o1}}$ consists of the evaluation of $\mathsf{c_{o1}}$ followed by the sequence of region commands $\mathsf{create\ C_{w2}}$. We know that $(S''_{po}, H''_{po}) \sim (S'_{pi}, H'_{pi})$. We can also prove that $(S''_{po}, H''_{po}, R''_p \cup C_{w2}) \sim (\Gamma(\bullet\mathsf{c_{i1}}), \mathcal{T})$, as follows. We inspect the conditions from the definition of consistency. By the analysis result relation (10) we have that $\Gamma(\mathsf{c_{i1}}\bullet) \sqsubseteq \Gamma(\bullet\mathsf{c_{i1}})$. By Property 3 and above relation (67), consistency condition (41) holds at program point $\bullet\mathsf{c_{i1}}$. And condition (42) holds because:

$$\begin{array}{lll}R''_p \cup \mathsf{C_{w2}} = & & \\ \quad = \mathsf{Live(c_{i1}}\bullet) \cup \mathsf{C_{w2}} & & \text{by relation (67)}\\ \quad = \mathsf{Live(c_{i1}}\bullet) \cup & & \\ \qquad \cup (\mathsf{Live}(\bullet\mathsf{c_{i1}}) - \mathsf{Live(c_{i1}}\bullet)) & & \text{by definition of } \mathsf{C_{w2}}\\ \quad = \mathsf{Live}(\bullet\mathsf{c_{i1}}) & & \mathsf{Live(c_{i1}}\bullet) \sqsubseteq \mathsf{Live}(\bullet\mathsf{c_{i1}})\\ & & \text{by relation (10)}\end{array}$$

We have therefore proved all the conditions for the consistency relation $(S''_{po}, H''_{po}, R''_p \cup R_{w2}) \sim (\Gamma(\bullet\mathsf{c_{i1}}), \mathcal{T})$. We can now apply the *inner induction hypothesis* for the evaluations:

$$\begin{array}{l}\langle S''_{pi}, H''_{pi}, \mathsf{c_i}\rangle \to \langle S'_{pi}, H'_{pi}, v_{pi}\rangle\\ \langle S''_{po}, H''_{po}, R''_p \cup C_{w2}, \mathsf{while\ (e)\ c'_{o1}}\rangle \to \langle S'_{po}, H'_{po}, R'_p, v_{po}\rangle\end{array}$$

By the inner induction hypothesis we get:

$$\begin{array}{lll}(S'_{po}, H'_{po}, v_{po}) \sim (S'_{pi}, H'_{pi}, v_{pi}) & & \\ (S'_{po}, H'_{po}, R'_p) \sim (\Gamma(\bullet\mathsf{c_{i1}}), \mathcal{T}) & \text{if} & v_{pi} = \mathsf{nr}\\ (v_{po}, H'_{po}) \sim (\Gamma(\bullet\mathsf{c_{i1}})(\mathsf{ret}), \mathcal{T}) & \text{if} & v_{pi} \neq \mathsf{nr}\end{array}$$

Finally, because $\Gamma(\bullet \mathsf{c_{i1}}) = \Gamma(\mathsf{c_i}\bullet)$ by relation (11), and because $\Gamma(\bullet \mathsf{c_{i1}})(\mathsf{ret}) = \Gamma(\bullet \mathsf{c_i})(\mathsf{ret})$, this completes the proof for this subcase and the proof of the property.

Rule for $\mathbf{c_i} = (\mathbf{x = e})$. The translation of $\mathsf{c_i}$ is:

$$\begin{array}{rcl} \mathsf{c_o} & = & \mathsf{create}\ \mathsf{C_b}\ ;\ \mathsf{c_i}\ ;\ \mathsf{remove}\ \mathsf{R_b} \\ \\ \text{where}\ \ \mathsf{R_b} & = & \mathsf{Live}(\bullet \mathsf{c_i}) - \mathsf{Live}(\mathsf{c_i}\bullet) \\ \mathsf{C_b} & = & \mathsf{Live}(\mathsf{c_i}\bullet) - \mathsf{Live}(\bullet \mathsf{c_i}) \end{array}$$

Consider input and output states satisfying conditions (52)-(55). From the semantic rules of assignments in the input and output languages, we have:

$$\begin{array}{lll} \mathsf{x} \in \mathsf{dom}(S_i) & \langle S_i, H_i, \mathsf{e}\rangle \to v & \\ S_i' = S_i[\mathsf{x} \to v] & H_i' = H_i & v_i = \mathsf{nr} \\ \\ \mathsf{x} \in \mathsf{dom}(S_o) & \langle S_o, H_o, R, \mathsf{e}\rangle \to v' & \\ S_o'' = S_o[\mathsf{x} \to v'] & H_o' = H_o & v_o = \mathsf{nr} \\ R' = (R \cup \mathsf{C_b}) - \mathsf{R_b} & & \end{array}$$

First, we prove that $(S_o'', H_o', v_o) \sim (S_i', H_i', v_i)$ using the hypothesis $(S_o, H_o) \sim (S_i, H_i)$. The equivalence is trivial for return values, because $v_i = \mathsf{nr} = v_i$, and for heaps, because $H_o' = H_o$, $H_i' = H_i$, and $H_o \downarrow = H_i$. For stacks we have:

$$\begin{array}{l} (S_o''(\mathsf{y})) \downarrow = (S_o(\mathsf{y})) \downarrow = S_i(\mathsf{y}) = S_i'(\mathsf{y}) \text{ for } \mathsf{y} \neq \mathsf{x} \\ (S_o''(\mathsf{x})) \downarrow = v' \downarrow = v = S_i'(\mathsf{x}) \end{array}$$

because $v' \downarrow = v$ from the Expression Preservation Lemma. Hence $S_o'' \downarrow = S_i'$. Therefore $(S_o'', H_o', v_o) \sim (S_i', H_i', v_i)$.

Because assignments never return values, we only need to prove the conclusion of implication (57): $(S_o'', H_o', R') \sim (\Gamma(\mathsf{c_i}\bullet), \mathcal{T})$. We prove this relation using hypothesis relation (55): $(S_o, H_o, R) \sim (\Gamma(\bullet \mathsf{c_i}), \mathcal{T})$.

We first prove that consistency relation (41) holds. By hypothesis, $(S_o, H_o) \sim (\Gamma(\bullet \mathsf{c_i}), \mathcal{T})$. That is, there is a mapping μ which witnesses this consistency and satisfies relations (38)-(40). We show that the same mapping witnesses the equivalence $(S_o'', H_o) \sim (\Gamma(\mathsf{c_i}\bullet), \mathcal{T})$. Since the heap and the mapping μ are unchanged, relations (38) and (40) trivially hold. Then, since the stack S_o'' and the type environment $\Gamma(\mathsf{c_i}\bullet)$ are the same as S_o and $\Gamma(\bullet \mathsf{c_i})$, except for x, it means that relation (39) holds for all $\mathsf{y} \neq \mathsf{x}$. Finally, consider (l, r) such that $S_o''(\mathsf{x}) = (l, \mathsf{r})$. The analysis result is such that $\mathsf{x} : \tau \in \Gamma(\mathsf{c_i}\bullet)$, where $\Gamma(\bullet \mathsf{c_i}) \vdash \mathsf{e} : \tau$. By the operational semantics of assignments, the execution of $\mathsf{c_o}$ must evaluate the assigned expression: $\langle S_o, H_o, R, \mathsf{e}\rangle \to (l, \mathsf{r})$. By hypothesis, $(S_o, H_o, R) \sim (\Gamma(\bullet \mathsf{c_i}), \mathcal{T})$. We can apply Property 4 and get: $\mu(l) = (\tau, \mathsf{r})$. Hence, we proved that:

$$S_o''(\mathsf{x}) = (l, \mathsf{r}) \ \Rightarrow\ \mathsf{x} : \tau \in \Gamma(\mathsf{c_i}\bullet) \ \wedge\ \mu(l) = (\tau, \mathsf{r})$$

Thus, relation (39) is satisfied for all x. This proves that $(S_o'', H_o) \sim (\Gamma(\mathsf{c_i}\bullet), \mathcal{T})$.

Using Property 1 and the fact that $\mathsf{R_b}$ and $\mathsf{Live}(\mathsf{c_i}\bullet)$ are disjoint by definition of $\mathsf{R_b}$, we conclude that $\neg\mathsf{reach}(S_o'', H_o', \mathsf{r})$ for each $\mathsf{r} \in \mathsf{R_b}$. Therefore, by the operational semantics of the region removal command we have that $S_o' = S_o''$.

Finally, we prove condition (42), that R' is consistent with the set of live regions after the statement. We have:

$$\begin{array}{rcl} \mathsf{C_b} & = & \mathsf{Live}(\mathsf{c_i}\bullet) - \mathsf{Live}(\bullet \mathsf{c_i}) \\ \mathsf{R_b} & = & \mathsf{Live}(\bullet \mathsf{c_i}) - \mathsf{Live}(\mathsf{c_i}\bullet) \\ R & = & \mathsf{Live}(\bullet \mathsf{c_i}) \end{array}$$

Hence:

$$R' = (R \cup \mathsf{C_b}) - \mathsf{R_b} = \mathsf{Live}(\mathsf{c_i}\bullet)$$

This relation is due to the set equality:

$$(A \ \cup \ (B - A)) \ - \ (A - B) \ = \ B$$

with $A = \mathsf{Live}(\bullet\mathsf{c_i})$ and $B = \mathsf{Live}(\mathsf{c_i}\bullet)$.

This proves that $(S'_o, H'_o, R') \sim (\Gamma(\mathsf{c_i}\bullet), \mathcal{T})$ and completes the proof for this case.

Rule for $\mathsf{c_i} = (\mathsf{x.f = e})$. The translation of $\mathsf{c_i}$ is:

$$\mathsf{c_o} \ \equiv \ (\mathsf{create\ C_b\,;\ c_i\,;\ remove\ R_b})$$

Consider input and output states satisfying conditions (52)-(55). From the operational semantics of the input and output languages, we have:

$$\begin{array}{c} S_i(\mathsf{x}) = l_0 \quad \langle S_i, H_i, \mathsf{e}\rangle \to v \\ S'_i = S_i \quad H'_i = H_i[l_0 \mapsto H_i(l_0)[\mathsf{f} \mapsto v]] \quad v_i = \mathsf{nr} \end{array}$$

$$\begin{array}{c} S_o(\mathsf{x}) = (l'_0, \mathsf{r}_0) \quad \langle S_o, H_o, R, \mathsf{e}\rangle \to v' \\ S''_o = S_o \quad H'_o = H_o[l'_0 \mapsto H_i(l'_0)[\mathsf{f} \mapsto v']] \quad v_o = \mathsf{nr} \\ \mathsf{r}_0 \in R \cup \mathsf{C_b} \quad R' = (R \cup \mathsf{C_b}) - \mathsf{R_b} \end{array}$$

First, we prove that $(S''_o, H'_o, v_o) \sim (S'_i, H'_i, v_i)$ using the hypothesis $(S_o, H_o) \sim (S_i, H_i)$. The equivalence is trivial for return values, because $v_i = \mathsf{nr} = v_i$, and for stacks, because $S''_o = S_o$, $S'_i = S_i$, and $S_o \downarrow = S_i$. From the stack consistency $S_o \downarrow = S_i$ we have: $(l'_0, \mathsf{r}) \downarrow = l_0$, so $l'_0 = l_0$. For heaps we have:

$$\begin{array}{l} (H'_o(l'')) \downarrow = (H_o(l'')) \downarrow = H_i(l'') = H'_i(l'') \text{ for } l'' \neq l_0 \\ (H'_o(l_0)) \downarrow = (H_o(l_0)[\mathsf{f} \mapsto v']) \downarrow = H_i(l_0)[\mathsf{f} \mapsto v] = H'_i(l_0) \end{array}$$

because $v' \downarrow = v$ from the Expression Preservation Lemma. Hence $H'_o \downarrow = H'_i$. Therefore $(S''_o, H'_o, v_o) \sim (S'_i, H'_i, v_i)$.

Because assignments never return values, we only need to prove the conclusion of implication (57): $(S''_o, H'_o, R') \sim (\Gamma(\mathsf{c_i}\bullet), \mathcal{T})$. We prove this relation using hypothesis relation (55): $(S_o, H_o, R) \sim (\Gamma(\bullet\mathsf{c_i}), \mathcal{T})$.

We first prove that consistency relation (41) holds. By hypothesis, $(S_o, H_o) \sim (\Gamma(\bullet\mathsf{c_i}), \mathcal{T})$. That is, there is a mapping μ which witnesses this consistency and satisfies relations (38)-(40). We show that the same mapping witnesses the equivalence $(S''_o, H_o) \sim (\Gamma(\mathsf{c_i}\bullet), \mathcal{T})$. Since the stack and the type environment are unchanged after the execution of $\mathsf{c_i}$, and we use the same mapping μ, relations (38) and (39) trivially hold. Then, since the stack H'_o is the same as H_o, except for l_0 and field f, it means that relation (40) holds for all $l' \neq l_0$.

Finally, consider $l' = l_0$ and (l_1, r_1) such that $H'_o(l_0)(\mathsf{f}) = (l_1, \mathsf{r}_1)$. Since $S_o(\mathsf{x}) = (l_0, \mathsf{r}_0)$, by hypothesis $((S_o, H_o) \sim (\Gamma(\bullet\mathsf{c_i}), \mathcal{T}))$ and relation (39), we have $\mu(l_0) = (\tau_0, \mathsf{r}_0)$. The analysis result is such that $\Gamma(\bullet\mathsf{c_i}) \vdash \mathsf{x.f} : \tau$. By the typing rules, we get $\mathsf{x} : \mathsf{ref}[\rho_0, \overline{\rho}](\overline{\tau}\ \overline{\mathsf{f}}) \in \Gamma(\bullet\mathsf{c_i})$ and $\tau\ \mathsf{f} \in \overline{\tau}\ \overline{\mathsf{f}}$. By relation (39), we have $\mathsf{r}_0 = \mathsf{ref}[\rho_), \overline{\rho}](\overline{\tau}\ \overline{\mathsf{f}})$, so:

$$\mu(l_0) = (\mathsf{ref}[\rho_0\overline{\rho}](\overline{\tau}\ \overline{\mathsf{f}}), \mathsf{r}_0) \quad \tau\ \mathsf{f} \in \overline{\tau}\ \overline{\mathsf{f}}$$

By the operational semantics of assignments, the execution of $\mathsf{c_o}$ must evaluate the expression. Because $H'_o(l_0)(\mathsf{f}) = (l_1, \mathsf{r}_1)$, the evaluation must yield a location: $\langle S_o, H_o, R, \mathsf{e}\rangle \to (l_1, \mathsf{r}_1)$. We

also know that the analysis result is such that $\Gamma(\bullet \mathsf{c_i}) \vdash \mathsf{e} : \tau'$, with $\tau' <: \tau$. Suppose $\tau' \neq \tau$. Then, by the definition of subtyping, $\tau' = \bot$. By property 5, $\langle S_o, H_o, R, \mathsf{e}\rangle \rightarrow \mathsf{null}$, contradiction. Hence $\tau' = \tau$ and $\Gamma(\bullet \mathsf{c_i}) \vdash \mathsf{e} : \tau'$. Also, by hypothesis, $(S_o, H_o, R) \sim (\Gamma(\bullet \mathsf{c_i}), \mathcal{T})$. We can apply Property 4 and get: $\mu(l_1) = (\tau, \mathsf{r}_1)$. Therefore, we proved that:

$$\begin{array}{rcl} H(l_0)(f) = (l_1, \mathsf{r}_1) & \Rightarrow & \mu(l_0) = (\mathsf{ref}[\rho_0, \overline{\rho}]\ (\overline{\tau}\ \overline{\mathsf{f}}), \mathsf{r}') \wedge \\ & & \mu(l_1) = (\tau, \mathsf{r}_1)\ \wedge\ \tau\ \mathsf{f} \in \overline{\tau}\ \overline{\mathsf{f}} \end{array}$$

Thus, relation (40) is satisfied for all x, so we have that $(S''_o, H_o) \sim (\Gamma(\mathsf{c_i}\bullet), \mathcal{T})$.

Using Property 1 and the fact that $\mathsf{R_b}$ and $\mathsf{Live}(\mathsf{c_i}\bullet)$ are disjoint by definition of $\mathsf{R_b}$, we conclude that $\neg\mathsf{reach}(S''_o, H'_o, \mathsf{r})$ for each $\mathsf{r} \in \mathsf{R_b}$. Therefore, by the operational semantics of the region removal command we have that $S'_o = S''_o$.

Finally, we can show the consistency relation (42), $R' = (R \cup \mathsf{C_b}) - \mathsf{R_b} = \mathsf{Live}(\mathsf{c_i}\bullet)$, using the same proof as in the case for $\mathsf{x} = \mathsf{e}$. Hence, $(S'_o, H'_o, R') \sim (\Gamma(\mathsf{c_i}\bullet), \mathcal{T})$. This completes the proof for this case.

Rule for $\mathsf{c_i} = (\mathsf{x} = \mathsf{new}\ \mathsf{s})$. The translation of $\mathsf{c_i}$ is :

$$\mathsf{c_o} \equiv (\mathsf{create}\ \mathsf{C_b}\,;\, \mathsf{x} = \mathsf{new}\ \mathsf{s}\ \mathsf{in}\ \mathsf{r}\,;\, \mathsf{remove}\ \mathsf{R_b})$$

where:

$$\Delta \vdash \mathsf{a} : \mathsf{ref}\ [\rho_0, \overline{\rho}]\ (\overline{\tau}\ \overline{\mathsf{f}}) \quad \mathsf{r} = \mathcal{T}[\![\rho_0]\!]$$

Consider input and output states satisfying conditions (52)-(55). Consider that l is the newly created location. We assume that the same location l is created during the evaluation of the input and output languages, whenever the evaluation starts in consistent input and output states. Therefore $l \notin \mathsf{dom}(H_i)$ and $l \notin \mathsf{dom}(H_o)$. From the operational semantics of the input and output languages, we have:

$$\begin{array}{lll} S'_i = S_i[\mathsf{x} \mapsto l] & H'_i = H_i \cup \{l \mapsto m\} & v_i = \mathsf{nr} \\ S''_o = S_o[\mathsf{x} \mapsto (l, \mathsf{r})] & H'_o = H_o \cup \{l \mapsto m\} & v_o = \mathsf{nr} \\ R' = (R \cup \mathsf{C_b}) - \mathsf{R_b} & m = \mathsf{init}(\overline{\mathsf{t}}) & \mathsf{s}\ :\ \mathsf{ref}(\overline{\mathsf{t}}\ \overline{\mathsf{f}}) \end{array}$$

From the relations, we immediately have $(S''_o, H'_o, v_o) \sim (S'_i, H'_i, v_i)$ because $v_o = v_i = \mathsf{nr}$, $(S_o, H_o) \sim (S_i, H_i)$ and:

$$\begin{array}{l} (S''_o(\mathsf{x})) \downarrow = (l, \mathsf{r}) \downarrow = l = S'_i(\mathsf{x}) \\ (H'_o(l)) \downarrow = m \downarrow = m = H'_i(l) \end{array}$$

Because assignments don't return values, we only need to prove the conclusion of implication (57): $(S'_o, H'_o, R') \sim (\Gamma(\mathsf{c_i}\bullet), \mathcal{T})$.

We first prove that consistency relation (41) holds. By hypothesis, $(S_o, H_o) \sim (\Gamma(\bullet \mathsf{c_i}), \mathcal{T})$. That is, there is a mapping μ which witnesses this consistency and satisfies relations (38)-(40). We show that extended mapping $\mu' = \mu \cup \{l \mapsto (\tau, \mathsf{r})\}$ is a witness for the consistency $(S''_o, H'_o) \sim (\Gamma(\mathsf{c_i}\bullet), \mathcal{T})$, where type τ is the type of the allocation command: $\tau = \mathsf{ref}\ [\rho_0, \overline{\rho}]\ (\overline{\tau}\ \overline{\mathsf{f}})$ and $\mathsf{r} = \mathcal{T}[\![\rho_0]\!]$.

From hypothesis, $(S_o, H_o) \sim (\Gamma(\bullet \mathsf{c_i}), \mathcal{T})$. Since S''_o updates S_o with a new value for x, $\Gamma(\mathsf{c_i}\bullet)$ updates $\Gamma(\bullet \mathsf{c_i})$ with a new type for x, and H'_o extends H_o with a value for l, it means that: relation (38) holds for all locations except l, relation (39) holds for all variables except x, and relation (40) holds for all locations except l. We next prove that these relations also hold for l and x.

Relation (38) immediately holds for the new location l, by construction of the extended mapping μ': $\mu'(l) = (\tau, \mathsf{r})$, where $\tau = \mathsf{ref}\ [\rho_0, \overline{\rho}]\ (\overline{\tau}\ \overline{\mathsf{f}})$ and $\mathsf{r} = \mathcal{T}[\![\rho_0]\!]$.

Relation (39) holds for variable x because all the relations in this equation hold: by the operational semantics we have $S''_o(x) = (l, \mathsf{r})$, by the analysis result relation (18) we get $\Gamma(\mathsf{c_i}\bullet) \vdash \mathsf{x} : \tau$, and by construction of μ' we have $\mu'(l) = (\tau, \mathsf{r})$.

Relation (40) trivially holds for the new location l, because the initialization function init sets all of the fields to 0 and null values. Hence $H'_o(l)(\mathsf{f}) \in \{0, \mathsf{null}\}$, so $H'_o(l)(\mathsf{f}) \neq (l', \mathsf{r})$ for all fields f. Since the premise is false, the implication is trivially true.

Thus, μ' satisfies all of the relations (38) - (40) for all variables and all locations. This proves that $(S''_o, H'_o) \sim (\Gamma(\mathsf{c_i}\bullet), \mathcal{T})$.

Using Property 1 and the fact that $\mathsf{R_b}$ and $\mathsf{Live}(\mathsf{c_i}\bullet)$ are disjoint by definition of $\mathsf{R_b}$, we conclude that $\neg\mathsf{reach}(S''_o, H'_o, \mathsf{r})$ for each $\mathsf{r} \in \mathsf{R_b}$. Therefore, by the operational semantics of the region removal command we have that $S'_o = S''_o$.

Finally, we can show the consistency relation (42), $R' = (R \cup \mathsf{C_b}) - \mathsf{R_b} = \mathsf{Live}(\mathsf{c_i}\bullet)$, using the same proof as in the case for $\mathsf{x} = \mathsf{e}$. Hence, $(S'_o, H'_o, R') \sim (\Gamma(\mathsf{c_i}\bullet), \mathcal{T})$. This completes the proof for this case.

Rule for $\mathsf{c_i} = (\mathsf{x} = \mathsf{p(y)})$. The translation of $\mathsf{c_i}$ is :

$$\mathsf{c_o} \equiv (\mathsf{create}\ \mathsf{C_p}\,;\, \mathsf{x} = \mathsf{p}\ [\bar{\mathsf{r}}]\ (\mathsf{y})\,;\, \mathsf{remove}\ \mathsf{R_b})$$

Consider input and output states satisfying conditions (52)-(55). Let $\mathsf{c_{ip}}$ be the command representing the body of the invoked procedure in the input program, and $\mathsf{c_{op}}$ the body of that procedure in the transformed program: $\mathsf{c_{op}} = \mathcal{T}[\![\mathsf{c_{ip}}]\!]$. From the operational semantics of the input and output languages, we have:

$$\begin{array}{l} S_i(\mathsf{y}) = v_1 \quad \langle\{\mathsf{z} \mapsto v_1\}, H_i, \mathsf{c_{ip}}\rangle \to \langle S''_i, H''_i, v''_i\rangle \\ v''_i \neq \mathsf{nr} \quad S'_i = S_i[\mathsf{x} \mapsto v''_i] \quad H'_i = H''_i \end{array}$$

$$\begin{array}{l} S_o(\mathsf{y}) = v'_1 \quad \langle\{\mathsf{z} \mapsto v'_1\}, H_o, \bar{\mathsf{r}}, \mathsf{c_{op}}[\bar{\mathsf{r}}/\bar{\mathsf{r}}_p]\rangle \to \langle S''_o, H''_o, \bar{\mathsf{r}}, v''_o\rangle \\ v''_o \neq \mathsf{nr} \quad S'''_o = S_o[\mathsf{x} \mapsto v''_o] \quad H'_o = H''_o \\ \bar{\mathsf{r}} \subseteq R'' \quad R'' = R \cup \mathsf{C_b} \quad R' = (R \cup \mathsf{C_b}) - \mathsf{R_b} \end{array}$$

By hypothesis, $(S_o, H_o) \sim (S_i, H_i)$. Hence, $v'_1 \downarrow = v_1$. Thus, $(\{\mathsf{z} \mapsto v'_1\}, H_o) \sim (\{\mathsf{z} \mapsto v_1\}, H_i)$.

We can also show that $(\{\mathsf{z} \mapsto v'_1\}, H_o, \bar{\mathsf{r}}) \sim (\Gamma(\bullet\mathsf{c_{ip}}), \mathcal{T})$, as follows. First, we obtain relation $(\{\mathsf{z} \mapsto v'_1\}, H_o) \sim (\Gamma(\bullet\mathsf{c_{ip}}), \mathcal{T})$ from the hypothesis $(S_o, H_o) \sim (\Gamma(\bullet\mathsf{c_i}), \mathcal{T})$, since it can be witnessed using the same mapping μ. We only need to ensure condition (39) for variable z. This fact comes from the subtype relation $\Gamma(\bullet\mathsf{c_i})(\mathsf{y}) <: \Gamma(\bullet\mathsf{c_{ip}})(\mathsf{z})$. Second, we have that $\mathsf{Live}(\bullet\mathsf{c_{ip}}) = \bar{\mathsf{r}}$, because the analysis starts the evaluation of the procedure body using an environment which has the argument type and the return type of the procedure.

We can apply the induction hypothesis for the translation of the procedure body and get:

$$\begin{array}{l} (S''_o, H''_o, v''_o) \sim (S''_i, H''_i, v''_i) \\ (\{\mathsf{ret} \mapsto v''_o\}, H''_o) \sim (\Gamma(\bullet\mathsf{c_{ip}}), \mathcal{T}) \end{array}$$

From the first relation, it follows that $(S_o[\mathsf{x} \mapsto v''_o], H''_o) \sim (S_i[\mathsf{x} \mapsto v''_i], H''_i)$. That is, $(S'''_o, H'_o) \sim (S'_i, H'_i)$.

From the second relation, we get $(S_o[\mathsf{x} \mapsto v''_o], H''_o) \sim (\Gamma(\mathsf{c_i}\bullet), \mathcal{T})$, using the same witness mapping μ. Therefore, $(S'''_o, H'_o) \sim (\Gamma(\mathsf{c_i}\bullet), \mathcal{T})$. Using Property 1 and the fact that $\mathsf{R_b}$ and $\mathsf{Live}(\mathsf{c_i}\bullet)$ are disjoint by definition of $\mathsf{R_b}$, we conclude that $\neg\mathsf{reach}(S'''_o, H'_o, \mathsf{r})$ for each $\mathsf{r} \in \mathsf{R_b}$. Therefore, by the operational semantics of the region removal command we have that $S'_o = S'''_o$. Finally, we show the consistency relation (42), $R' = (R \cup \mathsf{C_b}) - \mathsf{R_b} = \mathsf{Live}(\mathsf{c_i}\bullet)$, using the same proof as in the case for $\mathsf{x} = \mathsf{e}$. Hence, $(S'_o, H'_o, R') \sim (\Gamma(\mathsf{c_i}\bullet), \mathcal{T})$. This completes the proof for this case.

Rule for $\mathsf{c_i} = \textbf{return x}$. The translation of $\mathsf{c_i}$ is:

$$\mathsf{c_o} = \mathsf{remove\ R_r\ ;\ c_i}$$

Consider input and output states satisfying conditions (52)-(54). From the operational semantics of the languages:

$$\begin{array}{cccc} S_i(\mathsf{x}) = v & S'_i = \emptyset & H'_i = H_i & v_i = v \\ S_o(\mathsf{x}) = v'' & S'_o = \emptyset & H'_o = H_o & S''_o(\mathsf{x}) = v_o = v' \\ R' = R & & & \end{array}$$

Where S''_o is the stack after $\mathsf{remove\ R_r}$ but before $\mathsf{c_i}$. Since x is returned, by (**??**) we have that x doesn't reach any region in $\mathsf{R_r}$. Thus, $S_o(\mathsf{x}) = S''_o(\mathsf{x})$. Because $S_o \downarrow = S_i$, it means that $v' \downarrow = v$. Hence, we get $(S'_o, H'_o, v_o) \sim (S'_i, H'_i, v_i)$. Finally, we can show that $(\{\mathsf{ret} \mapsto v_o\}, H'_o) \sim (\Gamma(\bullet\mathsf{c_i}), \mathcal{T})$ using the same mapping μ as the one that witnesses the consistency relation $(S_o, H_o) \sim (\Gamma(\bullet\mathsf{c_i}), \mathcal{T})$. As the heap is unchanged ($H'_o = H_o$), relations (38) and (40) trivially hold. For the stack, relation (39), we only need to prove that:

$$v_o = (l, \mathsf{r}) \Rightarrow \mathsf{ret} : \tau \in \Gamma(\bullet\mathsf{c_i}) \wedge \mu(l) = (\tau, \mathsf{r})$$

Assume that $v_o = (l, \mathsf{r})$. The analysis result is such that $\mathsf{x} : \tau' \in \Gamma(\bullet\mathsf{c_i})$ and $\tau' <: \tau'$. Assume that $\tau' \neq \tau$. Then $\tau' = \bot$. By hypothesis, $(S_o, H_o, R) \sim (\Gamma(\bullet\mathsf{c_i}), \mathcal{T})$. By language semantics, $\langle S_o, H_o, R, \mathsf{x}\rangle \rightarrow v_o$. By Property 5, $v_o = \mathsf{null}$, contradiction. Therefore, $\tau' = \tau$ and $\mathsf{x} : \tau \in \Gamma(\bullet\mathsf{c_i})$ and $\mathsf{ret} : \tau \in \Gamma(\bullet\mathsf{c_i})$.

By hypothesis, $(S_o, H_o, R) \sim (\Gamma(\bullet\mathsf{c_i}), \mathcal{T})$. Since $S_o(\mathsf{x}) = v_o = (l, \mathsf{r})$ and $\mathsf{x} : \tau \in \Gamma(\bullet\mathsf{c_i})$, it means that $\mu(l) = (\tau, \mathsf{r})$. Hence, we proved that $\mathsf{ret} : \tau \in \Gamma(\bullet\mathsf{c_i})$ and $\mu(l) = (\tau, \mathsf{r})$. This completes the proof for this case and the proof of the theorem. □

B.9 Progress Theorem

Theorem 2 (Progress) *Given a command $\mathsf{c_i}$ in the input language, a type environment $\Gamma(\bullet\mathsf{c_i})$ at the program point before the command, and a translation $\mathcal{T}$, we have that:*

$$\begin{array}{rll} & \mathsf{c_o} = \mathcal{T}[\![\mathsf{c_i}]\!] & (68) \\ & \langle S_i, H_i, \mathsf{c_i}\rangle \rightarrow \langle S'_i, H'_i, v_i\rangle & (69) \\ & (S_o, H_o) \sim (S_i, H_i) & (70) \\ & (S_o, H_o, R) \sim (\Gamma(\bullet\mathsf{c_i}), \mathcal{T}) & (71) \\ \Rightarrow & & \\ & \langle S_o, H_o, R, \mathsf{c_o}\rangle \rightarrow \langle S'_o, H'_o, R', v_o\rangle & (72) \end{array}$$

Proof. The proof is by induction on the derivations of the evaluation $\langle S_i, H_i, \mathsf{c_i}\rangle \rightarrow \langle S'_i, H'_i, v_i\rangle$.

Rule for $\mathsf{c_i} = \textbf{s x}$. The translation of $\mathsf{c_i}$ is

$$\begin{array}{rcl} \mathsf{c_o} & = & (\mathsf{s\ x\ ;\ c_{i1}})\ ;\ \mathsf{remove\ R_d} \\ \mathsf{R_d} & = & \mathsf{Live(c_{i1}\bullet)} - \mathsf{Live(c_i\bullet)} \end{array}$$

Consider input and output states satisfying conditions (69)-(71). From semantic rule for this case:

$$\langle S_i \cup \{\mathsf{x} \mapsto \mathsf{null}\}, H_i, c_{i1}\rangle \rightarrow \langle S'_i \cup \{\mathsf{x} \mapsto v_{xi}\}, H'_i, v_i\rangle$$

By induction hypothesis for the translation $c_{o1} = \mathcal{T}[\![c_{i1}]\!]$, command $\mathsf{c_{o1}}$ evaluates to a value:

$$\langle S_o \cup \{\mathsf{x} \mapsto \mathsf{null}\}, H_o, R, \mathsf{c_{o1}}\rangle \rightarrow \langle S'', H'_o, R'', v_o\rangle$$

From the Preservation Theorem, the resulting output state is consistent with $\langle S'_i \cup \{\mathsf{x} \mapsto v_{xi}\}, H'_i, v_i\rangle$. Therefore, S'' is of the form $S'' = S_o \cup \{\mathsf{x} \rightarrow v_{xo}\}$. Hence, we can apply the evaluation rule for declarations:

$$\langle S_o, H_o, R, \mathsf{c_o}\rangle \rightarrow \langle S'_o, H'_o, R'', v_o\rangle$$

If $v_o \neq \mathsf{nr}$, this completes the evaluation of $\mathsf{c_o}$. Otherwise, $R'' = \mathsf{Live}(\mathsf{c_{i1}}\bullet)$, so $\mathsf{R_d} \subseteq R''$. Thus, the evaluation can proceed and execute the trailing region removal commands.

Rule for $\mathsf{c_i} = \mathbf{int}\ \mathsf{x}$. The proof is similar to the case above.

Rule 1 for $\mathsf{c_i} = \mathsf{c_{i1}} ; \mathsf{c_{i2}}$. Let $\mathsf{c_{o1}} = \mathcal{T}[\![\mathsf{c_{i1}}]\!]$ and $\mathsf{c_{o2}} = \mathcal{T}[\![\mathsf{c_{i2}}]\!]$. The translation of $\mathsf{c_i}$ is $\mathsf{c_o} = \mathsf{c_{o1}} ; \mathsf{c_{o2}}$. Consider input and output states satisfying conditions (69)-(71). From the semantic rule considered in this case, we have:

$$\langle S_i, H_i, \mathsf{c_{i1}}\rangle \rightarrow \langle S'_i, H'_i, v_i\rangle, \quad v_i \neq \mathsf{nr}$$

By induction hypothesis for the translation $\mathsf{c_{i1}}$ with the above evaluation, we have:

$$\langle S_o, H_o, R, \mathsf{c_{o1}}\rangle \rightarrow \langle S'_o, H'_o, R', v_o\rangle$$

By Preservation, $v_o \neq \mathsf{nr}$, so:

$$\langle S_o, H_o, R, \mathsf{c_o}\rangle \rightarrow \langle S'_o, H'_o, R', v_o\rangle$$

which proves that the evaluation of $\mathsf{c_o}$ doesn't get stuck.

Rule 2 for $\mathsf{c_i} = \mathsf{c_{i1}} ; \mathsf{c_{i2}}$. Let $\mathsf{c_{o1}} = \mathcal{T}[\![\mathsf{c_{i1}}]\!]$ and $\mathsf{c_{o2}} = \mathcal{T}[\![\mathsf{c_{i2}}]\!]$. The translation of $\mathsf{c_i}$ is $\mathsf{c_o} = \mathsf{c_{o1}} ; \mathsf{c_{o2}}$. Consider input and output states satisfying conditions (69)-(71). From the semantic rule considered in this case, we have:

$$\langle S_i, H_i, \mathsf{c_{i1}}\rangle \rightarrow \langle S''_i, H''_i, \mathsf{nr}\rangle$$
$$\langle S''_i, H''_i, \mathsf{c_{i2}}\rangle \rightarrow \langle S'_i, H'_i, v_i\rangle$$

By induction hypothesis for the translation of $\mathsf{c_{i1}}$ with the evaluation $\langle S_i, H_i, \mathsf{c_{i1}}\rangle \rightarrow \langle S''_i, H''_i, \mathsf{nr}\rangle$, command $\mathsf{c_{o1}}$ evaluates to a value, and by the Preservation Theorem, that value is nr:

$$\langle S_o, H_o, R, \mathsf{c_{o1}}\rangle \rightarrow \langle S''_o, H''_o, R'', \mathsf{nr}\rangle$$

Also, by the Preservation Theorem, $(S''_o, H''_o) \sim (S'_i, H'_i)$ and $(S''_o, H''_o, R'') \sim (\Gamma(\mathsf{c_{i1}}\bullet), \mathcal{T})$. Because $\Gamma(\mathsf{c_{i1}}\bullet) = \Gamma(\bullet\mathsf{c_{i2}})$, we can apply the induction hypothesis for the translation of $\mathsf{c_{i2}}$ with the evaluation $\langle S''_i, H''_i, \mathsf{c_{i2}}\rangle \rightarrow \langle S'_i, H'_i, v_i\rangle$, and get:

$$\langle S''_o, H''_o, R'', \mathsf{c_{o2}}\rangle \rightarrow \langle S'_o, H'_o, R', v_o\rangle$$

From the evaluations of $\mathsf{c_{o1}}$ and $\mathsf{c_{o2}}$, we conclude that the evaluation of $\mathsf{c_o}$ doesn't get stuck.

Rule 1 for $\mathsf{c_i} = \mathbf{if}\ (\mathsf{e})\ \mathbf{then}\ \mathsf{c_{i1}}\ \mathbf{else}\ \mathsf{c_{i2}}$. This rule evaluates the true branch $\mathsf{c_{i1}}$ of the command. Let $\mathsf{c_{o1}} = \mathcal{T}[\![\mathsf{c_{i1}}]\!]$ and $\mathsf{c_{o2}} = \mathcal{T}[\![\mathsf{c_{i2}}]\!]$. The translation of $\mathsf{c_i}$ is:

$$\mathsf{c_o} \quad = \quad \mathsf{if\ (e)\ then\ (c_{o1}\ ;\ create\ C_{c1})\ else\ (c_{o2}\ ;\ create\ C_{c2})}$$

Consider input and output states satisfying conditions (69)-(71). From the semantic rule considered in this case, we have:

$$\langle S_i, H_i, \mathsf{e}\rangle \to v \notin \{0, \mathsf{null}\}$$
$$\langle S_i, H_i, \mathsf{c_{i1}}\rangle \to \langle S_i', H_i', v_i\rangle$$

By Progress and Preservation of Expressions, e evaluates to a value $v' \notin \{0, \mathsf{null}\}$:

$$\langle S_o, H_o, R, \mathsf{e}\rangle \to v' \notin \{0, \mathsf{null}\}$$

By induction hypothesis for the translation of $\mathsf{c_{i1}}$ with the evaluation $\langle S_i, H_i, \mathsf{c_{i1}}\rangle \to \langle S_i', H_i', v_i\rangle$, command $\mathsf{c_{o1}}$ evaluates to a value v_o:

$$\langle S_o, H_o, R, \mathsf{c_{o1}}\rangle \to \langle S_o', H_o', R'', v_o\rangle$$

If $v_o \neq \mathsf{nr}$, this completes the evaluation of $\mathsf{c_o}$. Otherwise, we use the Preservation Theorem, to get $(S_o', H_o', R'') \sim (\Gamma(\mathsf{c_{i1}}\bullet), \mathcal{T})$. Hence, $R'' = \mathsf{Live}(\mathsf{c_{i1}}\bullet)$, so $R'' \cap \mathsf{C_{c1}} = \emptyset$. Therefore, the evaluation is not stuck and can execute the trailing region creation commands.

Rule 2 for $\mathsf{c_i}$ = if (e) then $\mathsf{c_{i1}}$ else $\mathsf{c_{i2}}$. This rule evaluates the false branch $\mathsf{c_{i2}}$ of the command and the proof is similar to the previous case.

Rules for $\mathsf{c_i}$ = while (e) $\mathsf{c_{i1}}$. We consider all of the three rules for the evaluation of while loops. The conclusion of all of these rules is:

$$(S_i, H_i, \mathsf{while\ (e)\ c_{i1}}) \to (S_i', H_i', v_i)$$

Let $\mathsf{c_{o1}} = \mathcal{T}[\![\mathsf{c_{i1}}]\!]$. The translation of $\mathsf{c_i}$ is:

$$\mathsf{c_o} \quad = \quad \mathsf{create\ C_{w1}\,;\ while\ (e)\ (c_{o1}\,;\ create\ C_{w2})}$$

Consider input and output states satisfying conditions (69)-(71). The output command first executes the region creation commands $\mathsf{create\ C_{w1}}$, and then evaluates the loop. Since $(S_o, H_o, R) \sim (\Gamma(\bullet\mathsf{c_i}), \mathcal{T})$, $R = \mathsf{Live}(\bullet\mathsf{c_i})$, so $R \cap \mathsf{C_{w1}} = \emptyset$. Therefore, we can evaluate the sequence of region commands $\mathsf{create\ C_{w1}}$:

$$\langle S_o, H_o, R, \mathsf{create\ C_{w1}}\rangle \to \langle S_o, H_o, \mathsf{R_{w1}}, \mathsf{nr}\rangle$$
where: $R_{\mathsf{w1}} = R \cup \mathsf{C_{w1}}$

By induction hypothesis, each sub-derivation of the evaluation $\langle S_i, H_i, \mathsf{c_i}\rangle \to \langle S_i', H_i', v_i\rangle$ satisfies the implication in the theorem. These sub-derivations include the evaluation of $\mathsf{c_{i1}}$ in the second and the third rule for loops. Given this fact, the following property holds:

Property 7 *Consider the commands $\mathsf{c_{i1}}$ and $\mathsf{c_{o1}'}$ in the current case. Let (S_{pi}, H_{pi}), (S_{pi}', H_{pi}', v_i) (S_{po}, H_{po}, R_p) be states and values such that:*

$$\langle S_{pi}, H_{pi}, \mathit{while}\ (\mathsf{e})\ \mathsf{c_{i1}}\rangle \to \langle S_{pi}', H_{pi}', v_{pi}\rangle \tag{73}$$
$$(S_{po}, H_{po}) \sim (S_{pi}, H_{pi}) \tag{74}$$
$$(S_{po}, H_{po}, R_p) \sim (\Gamma(\bullet\mathsf{c_{i1}}), \mathcal{T}) \tag{75}$$

and such that the derivation for:

$$(S_{pi}, H_{pi}, \mathit{while}\ (\mathsf{e})\ \mathsf{c_{i1}}) \to (S_{pi}', H_{pi}', v_{pi})$$

is a sub-derivation of, or is the same as, the derivation for:

$$(S_i, H_i, \mathit{while}\ (\mathsf{e})\ \mathsf{c_{i1}}) \to (S_i', H_i', v_i)$$

Then, there exists $(S_{po}', H_{po}', R_p', v_o)$ such that:

$$\langle S_{po}, H_{po}, R_p, \mathit{while}\ (\mathsf{e})\ \mathsf{c_{o1}'}\rangle \to \langle S_{po}', H_{po}', R_p', v_{po}\rangle \tag{76}$$

Using this result, we directly get the required relations for this case, using the following instantiation:

$$\begin{array}{ccc}(S_{pi}, H_{pi}) = (S_i, H_i) & (S'_{pi}, H'_{pi}) = (S'_i, H'_i) & v_{pi} = v_i \\ (S_{po}, H_{po}) = (S_o, H_o) & (S'_{po}, H'_{po}) = (S'_o, H'_o) & v_{po} = v_o \\ R_p = R_{\mathsf{w1}} & R'_p = R' & \end{array}$$

We can apply the property because one can show that: $(S_o, H_o, R_{\mathsf{w1}}) \sim (\Gamma(\bullet \mathsf{c_{i1}}), \mathcal{T})$, using the same proof as the one from the corresponding case in the Preservation Theorem.

We next prove the property by induction on the derivations of $\langle S_{pi}, H_{pi}, \mathsf{while\ (e)\ c_{i1}} \rangle \rightarrow \langle S'_{pi}, H'_{pi}, v_{pi} \rangle$. Consider input and output states and values satisfying relations (73) - (75). From condition (73), one of the three rules for while loops in the input operational semantics must apply. We consider each of these rules in turn.

Subcase: While Rule 1. Assume the first rule for while loops applies:

$$\begin{array}{l}\langle S_{pi}, H_{pi}, \mathsf{e} \rangle \rightarrow v \quad v \in \{0, \mathsf{null}\} \\ S'_{pi} = S_{pi} \quad H'_{pi} = H_{pi} \quad v_{pi} = \mathsf{nr}\end{array}$$

By Progress and Preservation of Expressions, e evaluates to a value $v' \in \{0, \mathsf{null}\}$ in the transformed program: $\langle S_o, H_o, \mathsf{e} \rangle \rightarrow v' \in \{0, \mathsf{null}\}$. Therefore, we can evaluate $\mathsf{while\ (e)\ c'_{o1}}$ using the first rule for while loops in the output operational semantics.

Subcase: While Rule 2. Assume the second rule for while loops applies:

$$\begin{array}{l}\langle S_{pi}, H_{pi}, \mathsf{e} \rangle \rightarrow v \quad v \notin \{0, \mathsf{null}\} \\ \langle S_{pi}, H_{pi}, \mathsf{c_{i1}} \rangle \rightarrow \langle S_{pi}, H_{pi}, v_{pi} \rangle \\ S'_{pi} = S_{pi} \quad H'_{pi} = H_{pi} \quad v_{pi} \neq \mathsf{nr}\end{array}$$

By Progress and Preservation of Expressions, e evaluates to a value $v' \notin \{0, \mathsf{null}\}$ in the transformed program: $\langle S_o, H_o, \mathsf{e} \rangle \rightarrow v' \notin \{0, \mathsf{null}\}$. We can apply the *outer induction hypothesis* to the translation of $\mathsf{c_{i1}}$ with the evaluation $\langle S_{pi}, H_{pi}, \mathsf{c_{i1}} \rangle \rightarrow \langle S_{pi}, H_{pi}, v_{pi} \rangle$, because we know that its derivation is a sub-derivation of $(S_i, H_i, \mathsf{while\ (e)\ c_{i1}}) \rightarrow (S'_i, H'_i, v_i)$. We get:

$$\langle S_{po}, H_{po}, R_p, \mathsf{c_{o1}} \rangle \rightarrow \langle S''_{po}, H''_{po}, R''_p, v'_{po} \rangle$$

By the Preservation Theorem, $v'_{po} \neq \mathsf{nr}$. Thus, we can evaluate $\mathsf{while\ (e)\ c'_{o1}}$ using the second rule for while loops in the output operational semantics.

Subcase: While Rule 3. Assume the third rule for while loops applies:

$$\begin{array}{l}\langle S_{pi}, H_{pi}, \mathsf{e} \rangle \rightarrow v \quad v \notin \{0, \mathsf{null}\} \\ \langle S_{pi}, H_{pi}, \mathsf{c_{i1}} \rangle \rightarrow \langle S''_{pi}, H''_{pi}, \mathsf{nr} \rangle \\ \langle S''_{pi}, H''_{pi}, \mathsf{c_i} \rangle \rightarrow \langle S'_{pi}, H'_{pi}, v_{pi} \rangle\end{array}$$

By Progress and Preservation of Expressions, e evaluates to a value $v' \notin \{0, \mathsf{null}\}$ in the transformed program: $\langle S_o, H_o, \mathsf{e} \rangle \rightarrow v' \notin \{0, \mathsf{null}\}$. We can apply the *outer induction hypothesis*, for the translation of $\mathsf{c_{i1}}$ with the evaluation $\langle S_{pi}, H_{pi}, \mathsf{c_{i1}} \rangle \rightarrow \langle S''_{pi}, H''_{pi}, \mathsf{nr} \rangle$, and get:

$$\langle S_{po}, H_{po}, R_p, \mathsf{c_{o1}} \rangle \rightarrow \langle S''_{po}, H''_{po}, R''_p, v'_{po} \rangle$$

By Preservation, $v'_{po} = \mathsf{nr}$, $(S''_{po}, H''_{po}) \sim (S''_{pi}, H''_{pi})$, and $(S''_{po}, H''_{po}, R''_p) \sim (\Gamma(\mathsf{c_{i1}}\bullet), \mathcal{T})$. Hence, $R''_p = \mathsf{Live(c_{i1}}\bullet)$, so $R''_p \cap \mathsf{C_{w2}} = \emptyset$. Therefore, we can evaluate the sequence of region commands $\mathsf{create\ C_{w2}}$:

$$\langle S''_{po}, H''_{po}, R''_p, \mathsf{create\ C_{w2}} \rangle \rightarrow \langle S''_{po}, H''_{po}, R''_p \cup \mathsf{C_{w2}}, \mathsf{nr} \rangle$$

One can show that: $(S''_{po}, H''_{po}, R''_p \cup \mathsf{C_{w2}}) \sim (\Gamma(\mathsf{c_{i1}}\bullet), \mathcal{T})$, using the same proof as the one from the corresponding case in the Preservation Theorem. We can then apply the *inner induction hypothesis* for $\langle S''_{pi}, H''_{pi}, \mathsf{c_i}\rangle \rightarrow \langle S'_{pi}, H'_{pi}, v_{pi}\rangle$ and get:

$$\langle S''_{po}, H''_{po}, R''_p \cup \mathsf{C_{w2}}, \mathsf{while\ (e)\ c'_{o1}}\rangle \rightarrow \langle S'_{po}, H'_{po}, R'_p, v_{po}\rangle$$

Thus, we can evaluate $\mathsf{while\ (e)\ c'_{o1}}$ in state (S_{po}, H_{po}, R_p) using the third rule for while loops in the output operational semantics. This concludes the proof of the property.

Rule for $\mathbf{c_i} = (\mathbf{x = e})$. The translation of $\mathsf{c_i}$ is:

$$\mathsf{c_o} \quad = \quad \mathsf{create\ C_b\,;\, c_i\,;\, remove\ R_b}$$

Consider input and output states satisfying conditions (69)-(71). By condition (71), $(S_o, H_o, R) \sim (\Gamma(\bullet\mathsf{c_i}), \mathcal{T})$, so $R = \mathsf{Live}(\bullet\mathsf{c_i})$. Thus, $R \cap \mathsf{C_b} = \emptyset$. Therefore, the execution can proceed by evaluating the region creation commands:

$$\langle S_o, H_o, R, \mathsf{create\ C_b}\rangle \rightarrow \langle S_o, H_o, R \cup \mathsf{C_b}, \mathsf{nr}\rangle$$

From the semantic rule considered in this case:

$$\begin{array}{l}\langle S_i, H_i, \mathsf{e}\rangle \rightarrow v \\ S'_i = S_i[\mathsf{x} \rightarrow v] \quad H'_i = H_i \quad v_i = \mathsf{nr}\end{array}$$

By condition (70), and Progress and Preservation for Expressions:

$$\langle S_o, H_o, \mathsf{e}\rangle \rightarrow v'$$

Hence, we can further evaluate $\mathsf{c_i}$ in the transformed program:

$$\begin{array}{l}\langle S_o, H_o, R \cup \mathsf{C_b}, \mathsf{c_i}\rangle \rightarrow \langle S'_o, H_o, R \cup \mathsf{C_b}, \mathsf{nr}\rangle \\ S'_o = S_o[\mathsf{x} \rightarrow v']\end{array}$$

We know that $\mathsf{R_b} \subseteq \mathsf{Live}(\bullet\mathsf{c_i})$ and $R = \mathsf{Live}(\bullet\mathsf{c_i})$. Thus, $\mathsf{R_b} \subseteq R \cup \mathsf{C_b}$, so the evaluation can proceed and execute the trailing region removal commands.

Rule for $\mathbf{c_i} = (\mathbf{x.f = e})$. The translation of $\mathsf{c_i}$ is:

$$\mathsf{c_o} \quad = \quad \mathsf{create\ C_b\,;\, c_i\,;\, remove\ R_b}$$

Consider input and output states satisfying conditions (69)-(71). By condition (71), $(S_o, H_o, R) \sim (\Gamma(\bullet\mathsf{c_i}), \mathcal{T})$, so $R = \mathsf{Live}(\bullet\mathsf{c_i})$. Thus, $R \cap \mathsf{C_b} = \emptyset$. Therefore, the execution can proceed by evaluating the region creation commands:

$$\langle S_o, H_o, R, \mathsf{create\ C_b}\rangle \rightarrow \langle S_o, H_o, R \cup \mathsf{C_b}, \mathsf{nr}\rangle$$

From the semantic rule considered in this case:

$$\begin{array}{l}S_i(\mathsf{x}) = l_0 \quad \langle S_i, H_i, \mathsf{e}\rangle \rightarrow v \\ S'_i = S_i \quad H'_i = H_i[l_0 \mapsto H_i(l_0)[\mathsf{f} \mapsto v]] \quad v_i = \mathsf{nr}\end{array}$$

By condition (70), and Progress and Preservation for Expressions:

$$\langle S_o, H_o, \mathsf{e}\rangle \rightarrow v'$$

Because $(S_o, H_o) \sim (S_i, H_i)$, $S_o(\mathsf{x}) = (l_0, \mathsf{r}_0)$. Thus, we have $\mathsf{reach}(S_o, H_o, \mathsf{r}_0)$. Also, by condition (71), $(S_o, H_o, R) \sim (\Gamma(\bullet\mathsf{c_i}), \mathcal{T})$. By Property 2, $\mathsf{r}_0 \in R$. So $\mathsf{r}_0 \in R \cup \mathsf{C_b}$.

Hence, we can further evaluate $\mathsf{c_i}$ in the transformed program:

$$\langle S_o, H_o, R \cup \mathsf{C_b}, \mathsf{c_i} \rangle \rightarrow \langle S_o, H_o', R \cup \mathsf{C_b}, \mathsf{nr} \rangle$$

We know that $\mathsf{R_b} \subseteq \mathsf{Live}(\bullet\mathsf{c_i})$ and $R = \mathsf{Live}(\bullet\mathsf{c_i})$. Thus, $\mathsf{R_b} \subseteq R \cup \mathsf{C_b}$, so the evaluation can proceed and execute the trailing region removal commands.

Rule for $\mathsf{c_i} = (\mathsf{x} = \mathsf{new\ s})$. The translation of $\mathsf{c_i}$ is :

$$\mathsf{c_o} \quad = \quad \mathsf{create\ C_b\,;\, x = new\ s\ in\ r\,;\, remove\ R_b}$$

Consider input and output states satisfying conditions (69)-(71). By condition (71), $(S_o, H_o, R) \sim (\Gamma(\bullet\mathsf{c_i}), \mathcal{T})$, so $R = \mathsf{Live}(\bullet\mathsf{c_i})$. Thus, $R \cap \mathsf{C_b} = \emptyset$. Therefore, the execution can proceed by evaluating the region creation commands:

$$\langle S_o, H_o, R, \mathsf{create\ C_b} \rangle \rightarrow \langle S_o, H_o, R \cup \mathsf{C_b}, \mathsf{nr} \rangle$$

From the semantic rule considered in this case:

$$\begin{array}{ll} \mathsf{s} : \mathsf{ref}(\bar{\mathsf{t}}\ \bar{\mathsf{f}}) & m = \mathsf{init}(\bar{\mathsf{t}}\ \bar{\mathsf{f}}) \\ S_i' = S_i[\mathsf{x} \mapsto l] & H_i' = H_i \cup \{l \mapsto m\} \quad v_i = \mathsf{nr} \end{array}$$

After the allocation statement, the analysis assigns to x the type of the allocation: $\Gamma(\mathsf{c_i}\bullet)(\mathsf{x}) = \mathsf{ref}\ [\rho_0, \overline{\rho}]\ (\overline{\tau}\ \bar{\mathsf{f}})$, with $\mathcal{T}[\![\rho_0]\!] = \mathsf{r}$. Therefore, $\mathsf{r} \in \mathsf{Live}(\mathsf{c_i}\bullet)$. Hence, either $\mathsf{r} \in \mathsf{Live}(\bullet\mathsf{c_i}) = R$, or $\mathsf{r} \in \mathsf{Live}(\mathsf{c_i}\bullet) - \mathsf{Live}(\bullet\mathsf{c_i}) = \mathsf{C_b}$. In either case, $\mathsf{r} \in R \cup \mathsf{C_b}$. Thus, the output program can further execute the allocation command:

$$\begin{array}{l} \langle S_o, H_o, R \cup \mathsf{C_b}, \mathsf{c_i} \rangle \rightarrow \langle S_o', H_o', R \cup \mathsf{C_b}, \mathsf{nr} \rangle \\ S_o' = S_o[\mathsf{x} \mapsto l] \\ H_o' = H_o \cup \{l \mapsto m\} \end{array}$$

We know that $\mathsf{R_b} \subseteq \mathsf{Live}(\bullet\mathsf{c_i})$ and $R = \mathsf{Live}(\bullet\mathsf{c_i})$. Thus, $\mathsf{R_b} \subseteq R \cup \mathsf{C_b}$, so the evaluation can proceed and execute the trailing region removal commands.

Rule for $\mathsf{c_i} = (\mathsf{x} = \mathsf{p(y)})$. The translation of $\mathsf{c_i}$ is :

$$\mathsf{c_o} \equiv (\mathsf{create\ C_p\,;\, x = p\ [\bar{r}]\ (y)\,;\, remove\ R_b})$$

Consider input and output states satisfying conditions (69)-(71). By condition (71), $(S_o, H_o, R) \sim (\Gamma(\bullet\mathsf{c_i}), \mathcal{T})$, so $R = \mathsf{Live}(\bullet\mathsf{c_i})$. Thus, $R \cap \mathsf{C_b} = \emptyset$. Therefore, the execution can proceed by evaluating the region creation commands:

$$\langle S_o, H_o, R, \mathsf{create\ C_b} \rangle \rightarrow \langle S_o, H_o, R \cup \mathsf{C_b}, \mathsf{nr} \rangle$$

Let $\mathsf{c_{ip}}$ be the command representing the body of the invoked procedure in the input program, and $\mathsf{c_{op}}$ the body of that procedure in the transformed program: $\mathsf{c_{op}} = \mathcal{T}[\![\mathsf{c_{ip}}]\!]$. From the semantic rule considered in this case:

$$\begin{array}{l} S_i(\mathsf{y}) = v_1 \quad \langle \{\mathsf{z} \mapsto v_1\}, H_i, \mathsf{c_{ip}} \rangle \rightarrow \langle S_i'', H_i'', v_i'' \rangle \\ v_i'' \neq \mathsf{nr} \quad S_i' = S_i[\mathsf{x} \mapsto v_i''] \quad H_i' = H_i'' \end{array}$$

By hypothesis, $(S_o, H_o) \sim (S_i, H_i)$. Hence, $S_o(\mathsf{y}) = v_1'$, with $v_1' \downarrow = v_1$. Thus, $(\{\mathsf{z} \mapsto v_1'\}, H_o) \sim (\{\mathsf{z} \mapsto v_1\}, H_i)$. Also, by relation (**??**), $\bar{\mathsf{r}} \subseteq R \cup \mathsf{C_b}$.

We can also show that $(\{\mathsf{z} \mapsto v_1'\}, H_o, \bar{\mathsf{r}}) \sim (\Gamma(\bullet \mathsf{c_{ip}}), \mathcal{T})$, in the same way we proved this result for the corresponding case in the Preservation Theorem. By the induction hypothesis for the translation of the procedure body $\mathsf{c_{ip}}$ with the evaluation $\langle \{\mathsf{z} \mapsto v_1\}, H_i, \mathsf{c_{ip}} \rangle \rightarrow \langle S_i'', H_i'', v_i'' \rangle$, we get:

$$\langle \{\mathsf{z} \mapsto v_1'\}, H_o, \bar{\mathsf{r}}, \mathsf{c_o p}[\bar{\mathsf{r}}/\bar{\mathsf{r}}_\mathsf{p}] \rangle \rightarrow \langle S_o'', H_o'', \bar{\mathsf{r}}, v_o'' \rangle$$

By the Preservation Theorem, $v_o'' \downarrow = v_i''$. Hence, $v_o'' \neq \mathsf{nr}$.

All of the premises in the semantic rule for procedure calls in the output language are satisfied, so the evaluation can further execute the procedure call $\mathsf{x} = \mathsf{p}\ [\bar{\mathsf{r}}]\ (\mathsf{y})$:

$$\begin{array}{l} \langle S_o, H_o, R \cup \mathsf{C_b}, \mathsf{x} = \mathsf{p}\ [\bar{\mathsf{r}}]\ (\mathsf{y}) \rangle \rightarrow \langle S_o', H_o', R \cup \mathsf{C_b}, \mathsf{nr} \rangle \\ S_o' = S_o[\mathsf{x} \rightarrow v_o''] \\ H_o' = H_o'' \end{array}$$

We know that $\mathsf{R_b} \subseteq \mathsf{Live}(\bullet \mathsf{c_i})$ and $R = \mathsf{Live}(\bullet \mathsf{c_i})$. Thus, $\mathsf{R_b} \subseteq R \cup \mathsf{C_b}$, so the evaluation can proceed and execute the trailing region removal commands.

Rule for $\mathsf{c_i} = \mathbf{return}\ \mathsf{x}$. The translation of $\mathsf{c_i}$ is:

$$\mathsf{c_o} = \mathsf{remove}\ \mathsf{R_r}\ ;\ \mathsf{c_i}$$

Consider input and output states satisfying conditions (69)-(71). By condition (71), $(S_o, H_o, R) \sim (\Gamma(\bullet \mathsf{c_i}), \mathcal{T})$, so $R = \mathsf{Live}(\bullet \mathsf{c_i})$. Thus, $\mathsf{R_r} \subseteq R$. Therefore, the execution can proceed by evaluating the region removal commands:

$$\langle S_o, H_o, R, \mathsf{remove}\ \mathsf{R_r} \rangle \rightarrow \langle S_o'', H_o, R - \mathsf{R_r}, \mathsf{nr} \rangle$$

Finally, from (69) and the operational semantics of the input language $S_i(\mathsf{x}) = v$. But $(S_o, H_o) \sim (S_i, H_i)$, so $S_o(\mathsf{x}) = v'$, with $v' \downarrow = v$. Since $\mathsf{dom}(S_o) = \mathsf{dom}(S_o'')$ we know that $S_o''(\mathsf{x}) = v''$, for some v''. We can therefore apply the semantic rule for return and execute the command. This completes the proof for this case and the proof of the theorem.

□