Predictable Memory Usage for Object-Oriented Programs

Wei-Ngan Chin[1,2], Huu Hai Nguyen[1], Shengchao Qin[1,2] *, and Martin Rinard[3]

[1] Computer Science Programme, Singapore-MIT Alliance
[2] Department of Computer Science, National University of Singapore
[3] Laboratory for Computer Science, Massachusetts Institute of Technology
{chinwn,nguyenh2,qinsc}@comp.nus.edu.sg, rinard@lcs.mit.edu

Abstract

We present a new (size-)polymorphic type system (for an object-oriented language) that characterizes the sizes of data structures and the amount of heap and stack memory required to successfully execute methods that operate on these data structures. Key components of this type system include type assertions that use symbolic Presburger arithmetic expressions to capture data structure sizes, the effect of methods on the sizes of the data structures that they manipulate, and the amount of memory that methods allocate and deallocate. For each method, it can provide expressions that (conservatively) capture the amount of memory required to execute the method as a function of the sizes of the method's inputs. The safety guarantee is that the method will never attempt to use more memory than its type expressions specify.

We have implemented a type checker to verify memory usages of object-oriented programs, and also an inference system to predict on memory usages. Our experience is that the type system can effectively capture memory bounds of object-oriented programs.

1. Introduction

Memory management is a key concern for many applications. Over the years researchers have developed a range of memory management approaches; examples include explicit allocation and deallocation, copying garbage collection, and region-based memory allocation. A key safety concern in every approach is the possibility of dangling references that allow the program to unsafely access memory that has been deallocated. Some approaches (garbage collection) rule out this possiblity altogether; for others researchers have developed type systems that ensure that well-typed programs either have no dangling references or never use dangling references to unsafely access deallocated memory. Examples include linear type systems [17, 19, 16] and type systems that eliminate the possibility of dangling references in programs that use region-based memory allocation [33, 20, 5, 12].

In this paper we address a complementary aspect of memory safety : the possibility that the program may attempt to allocate more memory than the execution platform can give it. We present a (size-)polymorphic type system that characterizes the amount of memory required to execute each program component. The key components of this type system include:

- **Data Structure Sizes and Size Constraints:** The type of each data structure includes parameters that characterize its size properties, which are expressed in terms of the sizes of data structures that it contains. In many cases the sizes of these data structures are correlated; our approach uses size constraints expressed using symbolic Presburger arithmetic expressions to precisely capture these correlations.
- **Memory Recovery:** Our type system captures the distinction between shared and unaliased objects and supports the safe explicit deallocation of unaliased objects.
- **Preconditions and Postconditions:** Each method comes with a precondition that captures both the expected sizes of the data structures on which it operates and any correlations between these sizes. The method's postcondition expresses the new size and correlations of these data structures after the method executes as a function of the original sizes when the method was invoked.
- **Memory Usage Effects:** Each method comes with a memory effect. This effect uses symbolic values to specify both the maximum amount of memory that the method may *consume* and the minimum amount of memory that it will *recover*. Memory effects are expressed at the granularity of classes and can capture not only the net change in the number of instances of each class but also the stack frames required to successfully execute the method. Note that our type system correctly takes tail call optimizations into account when computing the total size of stack frames required to execute the program.

Our type system therefore captures both the amount of memory required to successfully execute each method and the net effect of that execution on the amount of memory available to execute the rest of the program. To determine the amount of heap and stack memory required for the entire program, one merely examines the effect of the `main` method. By combining the various components of this effect, one can obtain symbolic expressions (where the free variables denote sizes of the inputs) that capture the amount of heap and stack memory required to execute the program. Our type checking algorithm guarantees that well-typed programs will execute safely given this amount of memory.

We believe that our type system can be of use whenever the memory consumption of the program is of interest. It should be especially useful for embedded and safety-critical software because 1) such software often operates on platforms with limited amounts

* now at Durham University, UK. Email: shengchao.qin@durham.ac.uk

Technical Report, SoC, NUS, Nov., 2004

of memory, and 2) failing because of insufficient memory can have severe real-world consequences. Our type system enables the developers of these systems to determine a safe upper bound on the amount of memory that the program may consume and to provision their systems accordingly. This paper makes the following contributions:

- **Type System:** We propose an advanced type system for object-oriented (OO) paradigm that is able to specify memory usage in a precise manner. To the best of our knowledge, this is probably the first *memory usage type system* for OO paradigm.
- **Memory Specification:** Our proposal includes a specification mechanism for memory usage. A *bag abstraction* notation is used to capture symbolic counts of memory consumed and recovered by class types.
- **Heap Recovery:** We advocate for *explicit heap recovery* to provide more timely reclamation of dead objects in support of tighter bounds on memory usage. We show that this recovery mechanism may be systematically and safely inserted.
- **Stack Recovery:** Explicit recovery mechanism is also extended to the runtime stack. However, the operations for stack recovery shall be inserted automatically. With these recovery operations, we show how *tail-call optimization* can be fully accounted.
- **Soundness:** Our set of type checking rules have been proven sound. Each well-typed program is guaranteed to meet its memory usage specification, and will *never fail due to insufficient memory* whenever its memory precondition is met.
- **Implementation:** We have built a type checker that is both precise and practical. This prototype is used to confirm the viability of our approach. We have successfully verified the memory needs for a suite of benchmark programs with the help of this checker.

The primary goal of our work is a framework for static verification of memory usage for object-oriented programs. Sec 2 illustrates the basic idea using a stack example. Sec 3 presents a core object-oriented language, called MEMJ, with size, alias and memory annotations. Sec 4 describes our mechanism for memory usage specification which requires explicit recovery of both heap and stack spaces. Sec 5 presents a set of syntax-directed type rules for verifying the memory needs of user programs. Sec 6 presents the dynamic semantics for our MEMJ language. Sec 7 outlines a set of safety theorems which confirm that well-typed programs never fail due to insufficient memory. Sec 8 describes how to perform memory usage inference. Sec 9 describes our implementation and highlights a suite of programs whose memory requirements have been successfully verified by our type checker. Related works are described in Sec 10, followed by a short conclusion.

2. Overview

Memory usage occurs primarily in the heap and the runtime stack. The heap is used to hold dynamically created objects; the stack holds parameters of method calls and variables of local blocks. In our model, heap space is consumed via the `new` operation for newly created objects, while unused objects may be recovered via an explicit deallocation primitive, called `dispose`. Correspondingly, stack space is consumed upon entry to either a method call or a local block, and is recovered at the end of their respective scopes.

Memory usage (based on consumption and recovery) must be calculated over the entire computation of each program. This calculation is done in a safe manner to help identify the high-water mark on memory space needed. We achieve this through the use of a conservative upper bound on memory consumed, and a conservative lower bound on memory recovered for each expression (and function). Moreover, it is possible to characterize the memory consumption and recovery as a symbolic expression which depends on the (sizes of) program's inputs.

To help predict the memory usage of each program, we propose a *sized type system* for object-oriented programs with support for interprocedural size analysis. In this type system, size properties of both user-defined types and primitive types are captured. In the case of primitive integer type $\texttt{int}\langle v\rangle$, the size variable v captures its integer value, while for boolean type $\texttt{bool}\langle b\rangle$, the size variable b is either 0 or 1 denoting `false` or `true`, respectively. For user-defined class types, we use $c\langle n_1,\ldots,n_p\rangle$ `where` ϕ ; ϕ_I with size variables $n_1,\ldots,n_p$ to denote size properties that are defined in size relation ϕ, and invariant constraint ϕ_I. As an example, consider a stack class (which is implemented with a linked list) and a binary tree class as shown below.

```
class List⟨n⟩ where n=m+1 ; n≥0 {
  Object⟨⟩@S val;
  List⟨m⟩@U next;
    :
class Stack⟨n⟩ where n=m ; n≥0 {
  List⟨m⟩@U head;
    :
class BTree⟨s, d⟩ where s=1+s1+s2∧d=1+max(d1, d2) ; s≥0∧d≥0 {
  Object⟨⟩@S val;
  BTree⟨s1, d1⟩@U left;
  BTree⟨s2, d2⟩@U right;
    :
```

$\texttt{List}\langle n\rangle$ denotes a linked-list data structure of size n, and similarly for $\texttt{Stack}\langle n\rangle$. The size relations $n{=}m{+}1$ and $n{=}m$ define some size properties of the objects in terms of the sizes of their components, while the constraint $n{\geq}0$ signifies an invariant associated with the class type. Class $\texttt{BTree}\langle s, d\rangle$ represents a binary tree with size variables *s* and *d* denoting the total number of nodes and the depth of the tree, respectively. Due to the need to track the states of mutable objects, our type system requires the support of alias controls of the form $A{=}\texttt{U} \mid \texttt{S} \mid \texttt{R} \mid \texttt{L}$. We use `U` and `S` to mark each reference that is (definitely) *unaliased* and (possibly) *shared*, respectively. We use `R` to mark read-only fields which must never be updated after object initialization. We use `L` to mark unique references that are temporarily borrowed by a parameter for the duration of its method's execution.

To specify memory usage precisely, we decorate each method with the following declaration:

$$t\ mn\ (t_1 v_1,\ldots,t_n v_n)\ \texttt{where}\ \ \phi_{pr};\phi_{po};\epsilon_c;\epsilon_r\ \{e\}$$

Note that ϕ_{pr} and ϕ_{po} denote the precondition and postcondition of the method, expressed in terms of constraints/formulae on the size variables of the method's parameters and result. Precondition ϕ_{pr} denotes an applicability condition of the method in terms of the sizes of its parameters. Postcondition ϕ_{po} can provide a precise size relation for the parameters and result of the declared method. The memory effect is captured by ϵ_c and ϵ_r. ϵ_c denotes *memory requirement*, i.e., the maximum memory space that *may be consumed*, while ϵ_r denotes *net release*, i.e., the minimum memory space that *will be recovered* at the end of method invocation. Memory effects (consumption and recovery) are expressed using a bag notation of the form $\{(c_i,\alpha_i)\}_{i=1}^{m}$, where c_i is either a class type or $\mathcal{S}$ (the runtime stack), while α_i denotes its symbolic count.

Examples of method declarations for the `Stack` class are given in Fig 1. The notation $(A \mid)$ prior to each method captures the alias annotation of the current `this` parameter. Note our use of the primed notation, advocated in [22, 27], to capture imperative changes on size properties. For the `push` method, $n'{=}n{+}1$ captures the fact that the size of the stack object has increased by 1; similarly, the postcondition for the `pop` method, $n'{=}n{-}1$, denotes that the

```
class Stack⟨n⟩ where n=m ; n≥0 {
      List⟨m⟩@U head;

L | void⟨⟩@S push(Object⟨⟩@S o)
    where true; n′=n+1; {(List,1),(S,5)}; {}
  { List⟨⟩@U tmp=this.head;
    this.head=new List(o,tmp)}

L | void⟨⟩@S pop()  where n>0; n′=n−1;
                          {(S,5)}; {(List,1)}
  { List⟨⟩@U t1 = this.head; List⟨⟩@U t2 = t1.next;
    t1.dispose(); this.head = t2}

L | bool⟨b⟩@S isEmpty()  where n≥0; n′=n ∧
    (n=0∧b=1 ∨ n>0∧b=0); {(S,5)}; {}
  { List⟨⟩@U t = this.head; bool⟨⟩@S v = isNull(t);
    this.head = t; v}

L | void⟨⟩@S push3pop2(Object⟨⟩@S o)
    where true; n′=n+1; {(List,2),(S,9)}; {(List,1)}
  { this.push(o); this.push(o); this.pop(o);
    this.push(o); this.pop(o)}}
```

Figure 1. Methods for the `Stack` Class

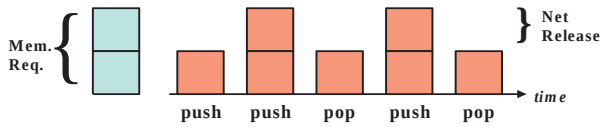

Figure 2. `push3pop2`: Heap Consumption and Recovery

size of the stack is decreased by 1 after the operation. The memory requirement for the `push` method, $\{(\texttt{List},1),(\mathcal{S},5)\}$, captures the fact that one `List` node will be consumed and the runtime stack $\mathcal{S}$ must have at least five words - two words for the parameters and one word for the local block, one for return address and one for previous frame pointer. (To provide a specification that is independent of the compiler used, we could use a symbolic constant, say $F=2$, to denote the extra words needed to support each call frame. This symbolic constant may be changed for different compilers. For simplicity, we ignore this feature here.)

For the `pop` method, $\epsilon_r = \{(\texttt{List},1)\}$ indicates that one `List` object will be recovered. For the `isEmpty` method, $n'=n$ captures the fact the size of the receiver object (`this`) is not changed by the method. Furthermore, its output of type $\texttt{bool}\langle b\rangle\texttt{@S}$ is related to the object's size through a disjunctive constraint $n=0\wedge b=1 \vee n>0\wedge b=0$. Note that primitive types are annotated with alias `S` because their values are immutable and can therefore be freely shared and yet remains trackable.

For the `push3pop2` method, the memory consumed (or required) from the heap is $\{(\texttt{List},2)\}$, while the net release is $\{(\texttt{List},1)\}$. This is illustrated by Fig. 2.

The recovery of stack space is *perfect* at method boundary and need not be explicitly specified. That is, if $\{(\mathcal{S},k)\}$ space from the runtime stack is consumed, then $\{(\mathcal{S},k)\}$ memory space will also be recovered for the stack. Also, size variables and their constraints are only specified at method boundary, and need not be specified for the local variables. Hence, our use of $\texttt{bool}\langle\rangle\texttt{@S}$ instead of $\texttt{bool}\langle v\rangle\texttt{@S}$ for a boolean-type local variable.

3. Language and Annotations

The language we focus on is a non-trivial core for an object-oriented language with size, alias, and memory annotations. We call this language MEMJ. The syntax is given in Fig 3. A suffix notation y^* denotes a list of zero or more distinct syntactic terms that are suitably separated. For example, $(t\ v)^*$ denotes $(t_1\ v_1,\ldots,t_n\ v_n)$ where $n\geq 0$. Local variable declarations are supported by block structure of the form: $(t\ v = e_1; e_2)$ with e_2 denoting the result.

$$
\begin{array}{rl}
P & ::= de^*\ m^* \\
de & ::= \texttt{class}\ c_1\langle n_{1..p}\rangle\ [\,\texttt{extends}\ c_2\langle n_{1..q}\rangle]\ \texttt{where}\ \phi;\phi_I\{fd^*\ (A\,|\,m)^*\} \\
fd & ::= t\ f \\
m & ::= t\ mn\ ((t\ v)^*)\ \texttt{where}\ \ \phi_{pr};\phi_{po};\epsilon_c;\epsilon_r\ \{e\} \\
t & ::= \tau\ \langle n^*\rangle\texttt{@}A \\
A & ::= \texttt{U}\ |\ \texttt{L}\ |\ \texttt{S}\ |\ \texttt{R} \\
\tau & ::= c\ |\ pr \\
pr & ::= \texttt{int}\ |\ \texttt{bool}\ |\ \texttt{void}\ |\ \texttt{float} \\
w & ::= v\ |\ v.f \\
e & ::= (c)\,\texttt{null}\ |\ k\ |\ w\ |\ w=e\ |\ t\ v=e_1;e_2 \\
 & \quad |\ \texttt{new}\ c(v^*)\ |\ v.mn\ (v^*)\ |\ mn\ (v^*) \\
 & \quad |\ \texttt{if}\ v\ \texttt{then}\ e_1\ \texttt{else}\ e_2\ |\ v.\texttt{dispose()} \\
\epsilon & = \{(c,\alpha)^*\} \qquad \textit{(Memory Space Abstraction)} \\
\phi & \in \mathbf{F} \qquad \textit{(Presburger Size Constraint)} \\
 & ::= b\ |\ \phi_1\wedge\phi_2\ |\ \phi_1\vee\phi_2\ |\ \neg\phi\ |\ \exists n\cdot\phi\ |\ \forall n\cdot\phi \\
b & \in \mathbf{BExp} \qquad \textit{(Boolean Expression)} \\
 & ::= \texttt{true}\ |\ \texttt{false}\ |\ \alpha_1=\alpha_2\ |\ \alpha_1<\alpha_2\ |\ \alpha_1\leq\alpha_2 \\
\alpha & \in \mathbf{AExp} \qquad \textit{(Arithmetic Expression)} \\
 & ::= k^{\texttt{int}}\ |\ n\ |\ k^{\texttt{int}}*\alpha\ |\ \alpha_1+\alpha_2\ |\ -\alpha \\
 & \quad |\ max(\alpha_1,\alpha_2)\ |\ min(\alpha_1,\alpha_2)
\end{array}
$$

where $k^{\texttt{int}} \in Z$ *is an integer constant*

$n \in SV$ *is a size variable*

$f \in Fd$ *is a field name*

$v \in Var$ *is an object variable*

Figure 3. Syntax for the MEMJ Language

We assume a call-by-value semantics for MEMJ, where values (primitives or references) are passed as arguments to parameters of methods. For simplicity, we do not allow the parameters to be updated (or re-assigned) with different values. There is no loss of generality, as we can always copy such parameters to local variables for updating, without altering the external behaviour of method calls.

The MEMJ language is deliberately kept simple to facilitate the formulation of static and dynamic semantics. This core language can be extended with syntactic abbreviations to make programming more convenient. Some syntactic conveniences include:

- Multi-declarations block is an abbreviation for nested expression blocks.
$$(d_1\ ;\cdots;d_n\ ;e) \equiv (d_1\ ;(\cdots(d_n\ ;e)\cdots))$$
- Sequence is a special case of local block, where e_1 is of `void` type; as shown below.
$$(e_1\ ;e_2) \equiv (\texttt{void}\langle\rangle\texttt{@S}\ v=e_1\ ;e_2)$$
For convenience, we introduce an alternative sequence which returns the first sub-expression as its result.
$$(e_1<;e_2) \equiv (t\ v=e_1\ ;e_2;v)$$
- Expressions may be used where variables are expected, aided by the following equivalences.
$$\begin{array}{l} m(e_1,\ldots,e_n) \equiv \\ \quad (t_1\ v_1=e_1\ ;\cdots;t_n\ v_n=e_n\ ;m(v_1,\ldots,v_n)) \\ \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \equiv \\ \quad (\texttt{bool}\langle\rangle\texttt{@S}\ v=e_1\ ;\ \texttt{if}\ v\ \texttt{then}\ e_2\ \texttt{else}\ e_3) \end{array}$$

- Loops can either be supported directly or be viewed as syntactic abbreviations for tail-recursive functions. With interconvertibility, we use the former for execution and the latter (which uses slightly more stack space) for type checking.

Several other language features, including downcast operation, `while` loop (to eliminate stack frame) and a field-binding construct (similar to pattern-matching), are also supported in our implementation. For simplicity, we put them in the appendix, as they play supporting roles and are not core to the main ideas proposed here.

To support sized typing, our programs are augmented with size variables and constraints. For size constraints, we presently restrict to Presburger form, as decidable (and practical) constraint solvers exist, e.g. [30]. For simplicity, we are only interested in tracking size properties of objects. We therefore restrict the relation ϕ in each class declaration of $c_1\langle n_{1..p}\rangle$ which extends $c_2\langle n_{1..q}\rangle$ to the form $\bigwedge_{i=q+1}^{p} n_i{=}\alpha_i$ whereby $V(\alpha_i)\cap\{n_1,..,n_p\}=\emptyset$. Note that $V(\alpha_i)$ returns the set of size variables that appeared in α_i. This restricts size properties to depend solely on the components of their objects. (Size constraints between components, such as those found for balanced heights of AVL trees are disallowed here, but may be placed in ϕ_I instead.)

Note that each class declaration has a set of instance methods whose main purpose is to manipulate objects of the declared class. For convenience, we also provide a set of static methods with the same syntax as instance methods, except for access to the `this` object.

One important characteristic of MEMJ is that memory recovery is done explicitly but safely (without creating dangling references). In particular, dead objects may be reclaimed via a v.`dispose`() primitive. While heap space recovery is the responsibility of the programmer, the stack recovery commands shall be automatically inserted (see Sec 4.3 for a translation scheme). To achieve automatic stack recovery, we add two primitives into an extended intermediate language, as follows.

$$e ::= \ldots \mid \texttt{rel}_A(k_1^{\texttt{int}}, k_2^{\texttt{int}}, e) \mid \texttt{rel}_B(k_1^{\texttt{int}}, k_2^{\texttt{int}}, e)$$

Note that $\texttt{rel}_B$ and $\texttt{rel}_A$ are for the explicit recovery of $k_1{+}k_2$ space from the stack *before* and *after* (respectively) the evaluation of its expression e. The k_1 value can only be either 0 or 2 depending on whether it is a local or method block. k_2 denotes the number of non-void parameters/variables in the frame. The $\texttt{rel}_B$ command is used to support tail-call optimization, whereby the stack recovery is made *before* the last call is evaluated.

3.1 Alias Checking

We introduce four alias control mechanisms `U` | `S` | `R` | `L` adopted from [6, 9, 1]. These alias mechanisms shall be used to support precise size tracking in the presence of mutable objects, and also for the explicit recovery of memory space when unique objects become dead. For size-tracking, we introduce `R`-mode fields to allow size-immutable properties to be accurately tracked for all objects. For example, an alternative class declaration for the list data type is given below, where its `next` field is marked as read-only (or immutable). Note that the `val` field remains mutable.

```
class RList⟨n⟩ where n=m+1 ; n≥0 {
  Object⟨⟩@S val;
  RList⟨m⟩@R next;
    :
```

The size property of such an `RList` type can be analysed at compile-time, while allowing its objects to be freely shared. However, this comes at the cost of losing both mutability and uniqueness.

We make use of `L`-mode parameters, with the *limited unique* (or *lent-once*) property [9], to capture unique references that are temporarily lent out to method calls. They allow the preservation of uniqueness together with precise size-tracking across methods. Consider the following method with two `List` parameters.

```
void⟨⟩@S join(List⟨m⟩@L x, List⟨n⟩@U y)
    where n > 0; m'=n+m; ···
  { if isNull(x.next) then x.next = y
    else join(x.next, y) }
```

The first parameter is annotated as *lent-once* and can thus be tracked for size properties without loss of uniqueness. However, the second parameter is marked *unique* as its reference escape the method body (into the tail of the `List` from the first parameter). In other words, the parameter `y` can have its uniqueness consumed but not `x`, as reflected in the body of the above method declaration. Given two unique lists, `a` and `b`, the call `join(a, b)` would consume the uniqueness of `b` but not that of `a`. Our lent-once policy is more restrictive than the policy of normal lending [1] as we require each lent-once parameter to be unaliased within the scope of its method. For example, `join(a, a)` is allowed by the type rules of [1], but disallowed by our lent-once's policy.

In our alias type system, uniqueness may be transferred (by either assignment or parameter-passing) from one location (variable, field or parameter) to another location. Consider a type environment {`x`::`Object`⟨⟩`@U`,`y`::`Object`⟨⟩`@U`, `z`::`Object`⟨⟩`@S`} where variables `x` and `y` are unique, while `z` is shared. In the code {`x = y; z = x`}, the uniqueness of `y` is first transferred to location `x`, followed by the consumption of uniqueness of `x` that is lost to the shared variable `z`. Alias subtyping rules (shown below) allow unique references to be passed to shared and lent-once locations (in addition to other unique locations), but not vice-versa.

$$A \leq_a A \quad \texttt{U} \leq_a \texttt{L} \quad \texttt{U} \leq_a \texttt{S}$$

A key difference of our alias checking rules, when compared to [1], is that we do not require an external "last-use" analysis for variables. Neither do we need to change the underlying semantics of programs to nullify each field location whose uniqueness is lost. We achieve this with a special set of references whose uniqueness have been consumed, called *dead-set* of the form $\{w^*\}$ where $w = v \mid v.f$. This dead-set is tracked flow-sensitively in our system using type judgement of the form:

$$\Gamma;\Theta \vdash e :: t, \Theta_1$$

Here, each dead-set $\Theta(\Theta_1)$ captures the set of references with consumed uniqueness before(after) the evaluation of expression e. Γ is a type enviroment which maps variables to their annotated types. Other type judgements for methods, classes and programs have the following forms.

$$\Gamma \vdash_{meth} meth \qquad \vdash_{def} def \qquad \vdash_P def_{i:1..p}\ meth_{i:1..q}$$

More details of our proposed alias annotation mechanism are described in another report [13]. In the Appendix, we give details of a separate set of alias checking type rules in Fig 10.

4. Memory Usage Specification

To allow memory usage to be precisely specified, we propose a bag abstraction of the form $\{(c_i,\alpha_i)\}_{i=1}^{n}$ where c_i denotes its classification, while α_i is its cardinality. In this paper, we shall use $c_i \in CN \cup \{\mathcal{S}\}$ where *CN* denotes all class types. For instance, $\Upsilon_1 = \{(c_1,2),(c_2,4),(\mathcal{S},3)\}$ denotes a bag with c_1 appearing twice, c_2 appearing four times and $\mathcal{S}$ appearing thrice. We provide the fol-

lowing two basic operations for bag abstraction to capture both the domain and the count of its element, as follows:

$$\begin{array}{lll} \mathit{dom}(\Upsilon) & =_{df} & \{c \mid (c,n) \in \Upsilon\} \\ \Upsilon(c) & =_{df} & n, \ \mathit{if}\ (c,n) \in \Upsilon \\ & & 0, \ \mathit{otherwise} \end{array}$$

We define several operations (union, difference, exclusion) over bags:

$$\begin{array}{lll} \Upsilon_1 \uplus \Upsilon_2 & =_{df} & \{(c, \Upsilon_1(c)+\Upsilon_2(c)) \mid \forall c \in \mathit{dom}(\Upsilon_1) \cup \mathit{dom}(\Upsilon_2)\} \\ \Upsilon_1 - \Upsilon_2 & =_{df} & \{(c, \Upsilon_1(c)-\Upsilon_2(c)) \mid \forall c \in \mathit{dom}(\Upsilon_1) \cup \mathit{dom}(\Upsilon_2)\} \\ \Upsilon \setminus X & =_{df} & \{(c, \Upsilon(c)) \mid \forall c \in \mathit{dom}(\Upsilon) - X\} \end{array}$$

To check for adequacy of memory, we provide a bag comparator operation under a size constraint Δ, as follows:

$$\Delta \vdash \Upsilon_1 \sqsupseteq \Upsilon_2 =_{df} (\Delta \Rightarrow (\forall c \in Z \cdot \Upsilon_1(c) \geq \Upsilon_2(c)))$$
$$\texttt{where}\ Z = \mathit{dom}(\Upsilon_1) \cup \mathit{dom}(\Upsilon_2)$$

The bag abstraction notation for memory is quite general and can be used in different ways. For example, if a small memory footprint is needed, we could bundle our system with a memory compacter and then change the memory abstraction to a coarser one with only two classifications, namely heap (denoted by $\mathcal{H}$) and runtime stack (denoted by denoted by $\mathcal{S}$). For simplicity, we shall only use the memory abstraction that is grouped by class types (and stack $\mathcal{S}$).

4.1 Memory Consumption

Heap space is consumed when objects are created by the `new` primitive, and also by method calls, except that the latter is aggregated to include recovery prior to consumption. Our aggregation (of recovery prior to consumption) is designed to identify a high-water mark of maximum memory needed for safe program execution. For each expression, we predict a conservative upper bound on the memory that the expression *may* consume, and also a conservative lower bound on the memory that the expression *will* release. If the expression releases some memory before consumption, we will use the released memory to obtain a lower memory requirement. Such aggregated calculations on both consumption and recovery can help capture both a net change in the level of memory, as well as the high-water mark of memory needed for safe execution.

For example, consider a recursive function which does p pops from one stack object, followed by the same number of pushes on another stack. (For simplicity, we omit the usage specification of runtime stack.)

```
void⟨⟩@S moverec(Stack⟨n⟩@L s, Stack⟨m⟩@L t, int⟨p⟩@S i)
    where n≥p≥0; n′=n−p∧m′=m+p; {} ; {}
  { if i<1 then ()
    else {Object⟨⟩@S o = s.top(); s.pop();
          moverec(s, t, i−1); t.push(o)} }
```

Due to aggregation (involving recovery before consumption), the heap space that may be consumed is zero. For each recursive call, the space for a `List` node is released by `s.pop()` before it is reused by `t.push(o)`. Aggregated over the recursive calls, we will have p number of `List` nodes that have been released before the same number of nodes are consumed. Hence, no new heap space is needed. Such aggregation is obviously sensitive to the order of the operations.

Consider now a different function which performs p pushes on `t`, followed by the same number of pops from `s`.

```
void⟨⟩@S moverec2(Stack⟨n⟩@L s, Stack⟨m⟩@L t, int⟨p⟩@S i)
    where n≥p≥0; n′=n−p∧m′=m+p; {(List, p)}; {(List, p)}
  { if i<1 then ()
    else {Object⟨⟩@S o = s.top(); t.push(o);
          moverec2(s, t, i−1); s.pop()} }
```

Though the net change in memory usage is also zero, the memory effect for this function is different as we require p number of `List` nodes to be consumed on entry, *before* the same number of `List` nodes be recovered. This new memory effect has the potential to push up the high-water mark of memory used by p `List` nodes.

4.2 Heap Recovery

In MEMJ, heap space recovery is achieved explicitly (and safely) through the `dispose` primitive. Explicit recovery of heap space has several advantages. It facilitates the timely recovery of dead objects, which allows memory usage to be predicted more accurately (with tighter bounds). It also permits the use of more efficient custom allocators[4], where desired.

Moreover, we shall provide an automatic technique to insert `dispose` primitives with the help of alias annotation. With such guidelines, the programmers' main role is to ensure that objects that are being disposed are non-null. This non-nullness information can be captured by a non-nullness analyser, such as [18], for a complete solution to explicit heap recovery. Let us see where heap recovery should be made.

Memory recovery via `dispose` should occur when unique references that are still alive (not in dead-set) are being discarded. This could occur at four places[1] : (i) end of local block, (ii) end of method block, (iii) prior to assignment operation, and (iv) at conditional expression. We would like to recover the memory space for each non-null reference that is about to become dead. For example, consider the `pop` method's definition:

```
L | void⟨⟩@S pop() where ···
    { List⟨⟩@U tmp = head.next; head = tmp}
```

The object pointed to by `head` is about to become dead prior to the operation, `head = tmp`. To recover this dead object, we may insert a `dispose` command to obtain `head = (tmp <; head.dispose())`. As another example, consider a definition of the `destroy` method which calls `emptyStack` with an L-mode parameter.

```
void⟨⟩@S destroy(Stack⟨n⟩@U s) where ···
  {emptyStack(s)}
void⟨⟩@S emptyStack(Stack⟨n⟩@L s) where ···
   { bool⟨⟩@S v = s.isEmpty();
     if v then () else {s.pop(); emptyStack(s)}}
```

A unique `s` object is about to become dead at the end of the `destroy` method. To recover this space, we shall insert `s.dispose()`, prior to the method's exit.

We present an automatic technique for the explicit recovery of dead objects that are known at compile-time. Given an expression e, we utilize the alias annotation to obtain a new expression e_1 where suitable explicit heap `dispose` operations have been safely inserted, as follows, where $\Theta(\Theta_1)$ denotes the set of dead references before (after) the evaluation of expression e.

$$\Gamma; \Theta \vdash e \hookrightarrow_H e_1 :: t, \Theta_1$$

Most rules are structure-preserving (c.f. identity) rewritings, except for four rules, where a sequence of disposal can be effected through *dispose*(D), with D containing a set of variable/field references that are to be disposed at the end of expression e.

$$\frac{\begin{array}{c}[\textsc{h:assign}] \\ \neg\, \mathit{isParam}(w) \quad \Gamma(w) = t \quad D = \{w \mid \mathit{ann}(t) = \texttt{U}\} - \Theta_1 \\ \Gamma; \Theta \vdash e \hookrightarrow_H e_1 :: t_1, \Theta_1 \quad \vdash t_1 <: t \\ e_2 = (e_1 \lhd D{=}\emptyset \rhd e_1 <; \mathit{dispose}(D)) \end{array}}{\Gamma; \Theta \vdash w = e \hookrightarrow_H w = e_2 :: \texttt{void@S}, \Theta_1 \setminus w}$$

[1] Note that unique reference cannot escape through e_1 in $e_1; e_2$ as we require e_1 to be of the `void` type.

$$\frac{\begin{array}{c}\Gamma_1 = \Gamma + \{v_1 :: t_1, .., v_p :: t_p\} \\ \Gamma_1; \emptyset \vdash e \hookrightarrow_H e_1 :: t, \Theta \quad \vdash t <: t_0 \quad \mathit{ann}(t_0) \neq \mathtt{L} \\ \forall i \in 1..p \cdot (\mathit{ann}(t_i) = \mathtt{L}) \Rightarrow (\forall f \cdot v_i.f \notin \Theta) \\ D = \{w \mid (w :: t) \in \Gamma_1, \mathit{ann}(t) = \mathtt{U}\} - \Theta \\ e_2 = (e_1 \lhd D{=}\emptyset \rhd e_1 <; \mathit{dispose}(D))\end{array}}{\Gamma \vdash_{\mathit{meth}} t_0\ mn((t_i\ v_i)_{i:1..p})\{e\} \hookrightarrow_H t_0\ mn((t_i\ v_i)_{i:1..p})\ \{e_2\}}\ [\textsc{H:METH}]$$

$$\frac{\begin{array}{c}\Gamma; \Theta \vdash e_1 \hookrightarrow_H e_3 :: t_1, \Theta_1 \quad \vdash t_1 <: t \\ \mathit{ann}(t) \notin \{\mathtt{L}, \mathtt{R}\} \quad \Gamma + \{v :: t\}; \Theta_1 \vdash e_2 \hookrightarrow_H e_4 :: t_2, \Theta_2 \\ D = \{v \mid \mathit{ann}(t) = \mathtt{U}\} - \Theta_2 \\ e_5 = (e_4 \lhd D{=}\emptyset \rhd e_4 <; \mathit{dispose}(D))\end{array}}{\Gamma; \Theta \vdash (t\ v = e_1\,;e_2) \hookrightarrow_H (t\ v = e_3\,;e_5) :: t_2, \Theta_2 \backslash v}\ [\textsc{H:LOCAL}]$$

$$\frac{\begin{array}{c}\Gamma(v) = \mathtt{bool}\langle b\rangle\mathtt{@S} \quad \Gamma; \Theta \vdash e_i \hookrightarrow_H \hat{e}_i :: t_i, \Theta_i \quad i = 1,2 \\ t = \mathit{msst}(t_1, t_2) \quad \Theta_3 = \Theta_1 \cup \Theta_2 \quad D_i = \Theta_3 - \Theta_i \quad i = 1,2 \\ E_i = (\hat{e}_i \lhd D_i{=}\emptyset \rhd \hat{e}_i <; \mathit{dispose}(D)) \quad i = 1,2\end{array}}{\Gamma; \Theta \vdash \mathtt{if}\ v\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 \hookrightarrow_H \mathtt{if}\ v\ \mathtt{then}\ E_1\ \mathtt{else}\ E_2 :: t, \Theta_3}\ [\textsc{H:IF}]$$

For the assignment rule [H:ASSIGN], we add w to the disposal set if it is unique and is not yet in dead-set using $D = \{w \mid \mathit{ann}(t){=}\mathtt{U}\} - \Theta_1$. For the method declaration rule [H:METH], we add to the disposal set those parameters which are unique but not yet dead using $\{w \mid (w :: t) \in \Gamma_1, \mathit{ann}(t) = \mathtt{U}\} - \Theta$. For the local declaration rule [H:LOCAL], we add v to the disposal set if it is unique but not yet dead using $\{v \mid \mathit{ann}(t) = \mathtt{U}\} - \Theta_2$. For the [H:IF] rule, the uniqueness that are consumed in only one branch may have their heap spaces recovered in the other branch. This is captured by $D_i = \Theta_3 - \Theta_i\ , i = 1, 2$.

Some other notations used are described below. The function $\mathit{isParam}(w)$ returns `true` if w is a parameter variable, otherwise it returns `false` (for fields and local variables). The function *ann* extracts the alias of an annotated type, $\mathit{ann}(\tau\langle v^*\rangle@A) = A$. The conditional is expressed as $\xi_1 \lhd b \rhd \xi_2 =_{df} \begin{cases} \xi_1, & \mathit{if}\ b; \\ \xi_2, & \mathit{otherwise}. \end{cases}$ Furthermore, we have:

$$\Theta\backslash v =_{df} \Theta - \{v, v.f^*\} \qquad \Theta\backslash v.f =_{df} \Theta - \{v.f\}$$

while $\mathit{msst}(t_1, t_2)$ returns the minimal supertype of both t_1 and t_2, as follows:

$$\frac{\begin{array}{c}\tau_1 <: \tau \quad \tau_2 <: \tau \quad \forall \tau_3 \cdot (\tau_1, \tau_2 <: \tau_3 \Rightarrow \tau <: \tau_3) \\ A_1 \leq_a A \quad A_2 \leq_a A \quad \forall A_3 \cdot (A_1, A_2 \leq_a A_3 \Rightarrow A \leq_a A_3)\end{array}}{\mathit{msst}(\tau_1@A_1, \tau_2@A_2) =_{df} \tau@A}$$

Note that $\tau_1 <: \tau_2$ denotes the subtype relation for underlying types (without annotations).

4.3 Automatic Stack Recovery

A crucial difference of the stack, when compared to the heap, is that the former is lexically scoped and has perfect recovery of its space. Its memory usage could thus be tracked without leakage. For stack space, the main rationale for explicit recovery is to facilitate tail-call optimization to minimise on stack usage. We propose to achieve this through a transformation scheme. Specifically, we introduce the following set of translations (for expressions, methods and class declarations) to automatically insert stack recovery operations.

$$e \hookrightarrow_S e_1 \qquad \Gamma \vdash \mathit{meth} \hookrightarrow_S \mathit{meth}_1 \qquad \mathit{def} \hookrightarrow_S \mathit{def}_1$$

Most of the rules are structure-preserving (c.f. identity) rewritings, except for the following two cases where a $\mathtt{rel}_A$ command is being inserted.

$$\frac{k = (0 \lhd \mathit{type}(t) = \mathtt{void} \rhd 1) \quad e_1 \hookrightarrow_S e_3 \qquad e_2 \hookrightarrow_S e_4}{(t\ v = e_1; e_2) \hookrightarrow_S \mathtt{rel}_A(0, k, (t\ v = e_3; e_4))}$$

$$\frac{e \hookrightarrow_S e_1 \qquad k = |\mathit{dom}(\Gamma)| + |v^*|}{\begin{array}{l}\Gamma \vdash t\ mn\ ((t\ v)^*)\ \mathtt{where}\ \phi_{pr}; \phi_{po}; \epsilon_c; \epsilon_r\{e\} \hookrightarrow_S \\ t\ mn\ ((t\ v)^*)\ \mathtt{where}\ \phi_{pr}; \phi_{po}; \epsilon_c; \epsilon_r\{\mathtt{rel}_A(2, k, e_1)\}\end{array}}$$

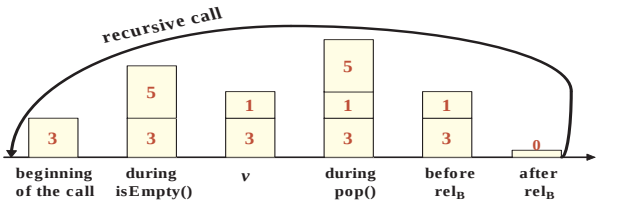

Figure 4. Stack Configurations for Execution of `emptystack`

Recall that the first argument of the $\mathtt{rel}_A$ command is to distinguish between local and method block. We assign a value of 2 for the latter, so as to provide recovery for two extra words present in each method's stack frame. For the second rule, we set k to the number of parameters using $|\mathit{dom}(\Gamma)| + |v^*|$, where Γ may contain a `this` parameter. We also use a set of rewritings (denoted by $\hookrightarrow_O$) to aggregate $\mathtt{rel}_A$ commands of nested blocks together, and to push each $\mathtt{rel}_A$ command towards the last subexpression that may be evaluated. The first rule is in a special form as only the outermost $\mathtt{rel}_A$ command may originate from the method block.

$$\mathtt{rel}_A(b, k_1, \mathtt{rel}_A(0, k_2, e)) \hookrightarrow_O \mathtt{rel}_A(b, k_1 + k_2, e))$$

$$\mathtt{rel}_A(b, k, (t\ v = e_1; e_2)) \hookrightarrow_O (t\ v = e_1; \mathtt{rel}_A(b, k, e_2))$$

$$\begin{array}{c}\mathtt{rel}_A(b, k, \mathtt{if}\ v\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2) \hookrightarrow_O \\ \mathtt{if}\ v\ \mathtt{then}\ \mathtt{rel}_A(b, k, e_1)\ \mathtt{else}\ \mathtt{rel}_A(b, k, e_2)\end{array}$$

For $\mathtt{rel}_A$ command that originates from method block, and if the last sub-expression is a call, we optimize its stack usage as a tail-call. This is done by replacing with a $\mathtt{rel}_B$ command, as shown below.

$$\frac{e = \mathit{mn}(v^*) \mid v.\mathit{mn}(v^*) \qquad b = 2}{\mathtt{rel}_A(b, k, e) \hookrightarrow_O \mathtt{rel}_B(b, k, e)}$$

Consider the following `emptyStack` method.

```
void⟨⟩@S emptyStack(Stack⟨n⟩@L s) where ···
  { bool⟨⟩@S v = s.isEmpty();
    if v then () else {s.pop(); emptyStack(s)}}
```

After the insertion of $\mathtt{rel}_A$ primitives by $\hookrightarrow_S$, we initially obtain the following method body where the stack requirement is linear to its input size.

```
void⟨⟩@S emptyStack(Stack⟨n⟩@L s) where n≥0∧d=n;
    n′=0; {(S, 4×d+8)}; {(List, d)}
{rel_A(2, 1,rel_A(0, 1, ( Bool⟨⟩@S v = s.isEmpty();
  if v then () else (s.pop(); emptyStack(s)))))}
```

Further rewriting by $\hookrightarrow_O$ would result in the following code with the effect of tail-call optimization.

```
void⟨⟩@S emptyStack(Stack⟨n⟩@L s) where n≥0∧d=n;
    n′=0; {(S, 9)}; {(List, d)}
  {( Bool⟨⟩@S v = s.isEmpty();
    if v then rel_A(2, 2, ())
    else (s.pop(); rel_B(2, 2, emptyStack(s))))}
```

One benefit of this new code is that its stack space requirement is now $\{(\mathcal{S}, 9)\}$ instead of $\{(\mathcal{S}, 4{\times}d{+}8)\}$. Fig. 4 illustrates the effect of tail-call optimization on the runtime stack.

5. Type Rules for Memory Checking

We present type judgements for *expressions*, *method declarations*, *class declarations* and *programs* to check for adequacy of memory, using relations of the form:

$$\Gamma; \Delta; \Upsilon \vdash e :: t, \Delta_1, \Upsilon_1 \qquad \Gamma \vdash_{\mathit{meth}} \mathit{meth} \qquad \vdash_{\mathit{class}} \mathit{def} \qquad \vdash P$$

Note that Γ is a type environment mapping program variables to their annotated types; $\Delta(\Delta_1)$ denotes the size constraint, which

holds for the size variables associated with Γ (Γ and t) for expression e before (after) its evaluation; t is an annotated type. Also, $\Upsilon(\Upsilon_1)$ is used to denote the available memory space in terms of bag abstraction before (after) the evaluation.

We present a few key syntax-directed type rules in Fig 5, with the rest of the rules in the Appendix (Fig 11). Before that, let us describe some notations used by the type rules.

5.1 Notations

We use function *V* to return size variables in a size formula, e.g., $V(x'{=}z{+}1{\wedge}y{=}2) = \{x', y, z\}$. We extend it to annotated type, type environment, and memory specification, e.g., $V(\tau\langle n^*\rangle@A){=}\{n^*\}$, $V(\{(\mathcal{S}, 4{\times}d{+}8)\}){=}\{d\}$. While the function V_u is used to return size variables in unprimed form, e.g. $V_u(x'{=}z{+}1{\wedge}y{=}2) = \{x, y, z\}$.

The function *prime* takes a set of size variables and returns their primed version, e.g. $\mathit{prime}(\{s_1, \ldots, s_n\}){=}\{s'_1, \ldots, s'_n\}$. Note that prime operation is idempotent, namely $v''{=}v'$. We extend this to (annotated) type, type environment, and even substitution. For example, $\mathit{prime}(\tau\langle n_1, \ldots, n_k\rangle) = \tau\langle n'_1, \ldots, n'_k\rangle$, and $\mathit{prime}([x{\mapsto}a, y{\mapsto}b]) = [x'{\mapsto}a', y'{\mapsto}b']$. Often, we need to express a no-change condition on a set of size variables. We define a $no\mathcal{X}$ operation as follows which returns a formula for which the original and primed variables are made equal.

$$no\mathcal{X}(\{\}) =_{df} \texttt{true} \qquad no\mathcal{X}(\{x\}{\cup}X) =_{df} (x'{=}x){\wedge}no\mathcal{X}(X)$$

We extend this function to annotated types (and type environments), as follows: $no\mathcal{X}(t) =_{df} no\mathcal{X}(V(t))$. Also, we use $n^* = \mathit{fresh}()$ to generate new size variables n^*. We extend it to annotated type, so that $\hat{t} = \mathit{fresh}(t)$ will return a new type $\hat{t}$ with the same underlying type as t but with fresh size variables instead. The function $\mathit{equate}(t_1, t_2)$ generates equality constraints for the corresponding size variables of its two arguments, usually when both arguments share the same underlying type. For example, we have $\mathit{equate}(\texttt{Int}\langle r\rangle, \texttt{Int}\langle s'\rangle) = (r{=}s')$. The function $\mathit{rename}(t_1, t_2)$ returns an equality substitution, e.g. $\mathit{rename}(\texttt{Int}\langle r\rangle, \texttt{Int}\langle s'\rangle){=}[r{\mapsto}s']$. The operator $\cup$ combines two domain-disjoint substitutions into one.

The function *fdList* is used to retrieve a full list of fields for a given class, together with its size relation. The function *inv* is used to retrieve the size invariant that is associated with each type. This function shall also be extended to type environment and list of types. The function $\mathbf{V}_{\mathit{field}}$ classifies size variables from each field into three groups : (i) immutable, (ii) mutable but unique, (iii) otherwise (non-trackable).

5.2 Assignment

The [ASSIGN] rule captures imperative updates (to object fields and variables) by modifying the current size constraint to a new updated state with changes to the imperative size variables from the LHS. From the rule, note that $\Gamma \vdash w :: t, \phi, Y$ is to identify Y as a set of imperative size variables and also to gather an invariant ϕ for this set. The subtype relation $\vdash t_1 <: t, \rho$ will return a substitution that maps the size variables of supertype to that of the subtype. This mapping ignores all non-trackable size variables that may be globally aliased, but immutable and unique mutable size variables are captured by it.

Given $\Gamma = \{\texttt{n} :: \texttt{int}\langle n\rangle@\texttt{S}, \texttt{b} :: \texttt{int}\langle b\rangle@\texttt{S}\}$, consider:

$$\frac{\begin{array}{c}\Gamma; \Delta; \Upsilon \vdash \texttt{n} + \texttt{b} :: \texttt{int}\langle s\rangle@\texttt{S}, \Delta_1, \Upsilon \\ \Gamma \vdash \texttt{n} :: \texttt{int}\langle r\rangle, r{=}n', \{n\} \quad \vdash \texttt{int}\langle s\rangle{<:}\texttt{int}\langle r\rangle, [r{\mapsto}s] \\ \Delta_1{=}\Delta{\wedge}s{=}n'{+}b' \quad \Delta_2{=}\exists s, r \cdot (\Delta_1 \circ_{\{n\}} [r{\mapsto}s] r{=}n')\end{array}}{\Gamma; \Delta; \Upsilon \vdash \texttt{n} = \texttt{n} + \texttt{b} :: \texttt{void}\langle\rangle@\texttt{S}, \Delta_2, \Upsilon}$$

This example illustrates how *primed* notation is used to represent the latest values of size variables at each post-state. It also shows how updates are effected by a sequential composition operator, $\circ_Y$, with $Y = \{n\}$ to denote the set of size variables that is being updated (*cf.* [22]). For example, if $\Delta{\equiv}n'{=}n{+}b{\wedge}b'{=}3$, we expect the program state after the assignment $\texttt{n} = \texttt{n} + \texttt{b}$ to be $\Delta_2{\equiv}n'{=}(n{+}b){+}b'{\wedge}b'{=}3$. This can be obtained by:

$$\begin{array}{rcl}\Delta_2 & \equiv & \exists s, r \cdot \Delta{\wedge}s{=}n'{+}b' \circ_{\{n\}} s{=}n' \\ & \equiv & \exists s, r \cdot \exists n_0 \cdot n_0{=}n{+}b{\wedge}b'{=}3{\wedge}s{=}n_0{+}b'{\wedge}s{=}n' \\ & \equiv & \exists s, r \cdot b'{=}3{\wedge}s{=}(n{+}b){+}b'{\wedge}s{=}n' \\ & \equiv & b'{=}3{\wedge}(n{+}b){+}b'{=}n'\end{array}$$

More formally, sequential composition is defined as:

$$\begin{array}{l}\Delta \circ_Y \phi =_{df} \exists\, r_1 \cdots r_n \cdot \rho_2(\Delta) \wedge \rho_1(\phi) \\ \text{where} \quad Y = \{s_1, \ldots, s_n\}\ ;\ \{r_1, \ldots, r_n\} = \mathit{fresh}() \\ \qquad \rho_1 = [s_i \mapsto r_i]_{i=1}^n\ ;\ \rho_2 = [s'_i \mapsto r_i]_{i=1}^n\end{array}$$

5.3 Memory Operations

The heap space is directly changed by the `new` and `dispose` primitives. Their corresponding type rules, [NEW] and [DISPOSE], would ensure that sufficient memory is available for consumption by `new` and will credit back space relinquished by `dispose`. The memory effect is accumulated according to the flow of computation. Consider:

$$\frac{\begin{array}{c}\Delta{\vdash}\Upsilon{\sqsupseteq}\{(\texttt{List}, 1)\} \quad \Delta_1{=}\Delta{\circ}_{\{x\}}x'{=}x{+}1 \\ \Gamma; \Delta; \Upsilon \vdash \texttt{x} = \texttt{new List(o, x)} :: \texttt{void}\langle\rangle@\texttt{S},\ \Delta_1, \Upsilon{-}\{(\texttt{List}, 1)\} \\ \Upsilon_1{=}(\Upsilon{-}\{(\texttt{List}, 1)\}){\uplus}\{(\texttt{List}, 1)\} \\ \Gamma; \Delta_1; \Upsilon{-}\{(\texttt{List}, 1)\} \vdash \texttt{y.dispose()} :: \texttt{void}\langle\rangle@\texttt{S},\ \Delta_1, \Upsilon_1\end{array}}{\Gamma; \Delta; \Upsilon \vdash \texttt{x} = \texttt{new List(o, x); y.dispose()} :: \texttt{void}\langle\rangle@\texttt{S},\ \Delta_1, \Upsilon}$$

The `new` operation consumes a `List` node, while the `dispose` operation releases back a `List` node. The net effect is that available memory Υ is unchanged. However, due to the order of the two operations, we require $\Delta{\vdash}\Upsilon{\sqsupseteq}\{(\texttt{List}, 1)\}$ which affects the maximum memory required.

Several other rules also have a direct effect on memory. The rules [REL–B] and [REL–A] are used to recover stack space before and after the evaluation of their expressions, respectively. For the method invocation rule [IMI], sufficient memory must be available for consumption prior to the call (as specified by $\Delta_1 \vdash \Upsilon{\sqsupseteq}\epsilon_c$), with the net memory release added back in the end (as specified by $\Upsilon_1 = \Upsilon{-}(\epsilon_c{\setminus}\{\mathcal{S}\}){\uplus}\epsilon_r$). Note that stack space is fully recovered (as $\mathcal{S}$ is excluded from consumption by $\epsilon_c{\setminus}\{\mathcal{S}\}$). Each method precondition must be met by the pre-state of its caller. This is checked by $\Delta{\approx\!\!>}_{V(\Gamma)}\exists V(\epsilon_c){\cup}V(\epsilon_r){\cdot}\rho\ \phi_{pr}$ which uses the logical relation ${\approx\!\!>}_X$, defined as:

$$\begin{array}{l}\Delta \approx\!\!>_X \phi =_{df} (\Delta \Rightarrow \rho\phi), \text{ where} \\ \quad \rho = [s_1 \mapsto s'_1, .., s_n \mapsto s'_n] \text{ and } V_u(\phi) \cap X = \{s_1, .., s_n\}.\end{array}$$

5.4 Conditional

Our type rule for conditional [IF] is able to track both the size-constraints and memory usages in a path-sensitive manner. Path-sensitivity is encoded by adding $b'{=}1$ and $b'{=}0$ to the pre-states of the two branches, respectively. We achieve path-sensitivity for memory usage specification by integrating it with relational size constraints derived.

Given $\Gamma = \texttt{b}{::}\texttt{Bool}\langle b\rangle@\texttt{S}, \texttt{s}{::}\texttt{Stack}\langle s\rangle@\texttt{U}$, we can derive:

$$\frac{\begin{array}{c}\Delta{\wedge}b'{=}1{\vdash}\Upsilon{\sqsupseteq}\{(\texttt{List}, 1), (\mathcal{S}, 5)\} \quad \Delta_1{=}\Delta{\wedge}b'{=}1 \circ_{\{s\}} s'{=}s{+}1 \\ \Gamma; \Delta{\wedge}b'{=}1; \Upsilon \vdash \texttt{s.push()} :: \texttt{void}\langle\rangle@\texttt{S},\ \Delta_1, \Upsilon{-}\{(\texttt{List}, 1)\} \\ \Delta{\wedge}b'{=}0{\vdash}\Upsilon{\sqsupseteq}\{(\mathcal{S}, 5)\} \quad \Delta_2{=}\Delta{\wedge}b'{=}0{\circ}_{\{s\}}s'{=}s{-}1 \\ \Gamma; \Delta{\wedge}b'{=}0; \Upsilon \vdash \texttt{s.pop()} :: \texttt{void}\langle\rangle@\texttt{S},\ \Delta_2, \Upsilon{\uplus}\{(\texttt{List}, 1)\} \\ (\_, \Upsilon_3, \Delta_3){=}\mathit{unify}(\_, \_, \Upsilon{-}\{(\texttt{List}, 1)\}, \Upsilon{\uplus}\{(\texttt{List}, 1)\}, \Delta_1, \Delta_2)\end{array}}{\Gamma; \Delta; \Upsilon \vdash \texttt{if b then s.push() else s.pop()} :: \texttt{void}\langle\rangle@\texttt{S},\ \Delta_3, \Upsilon_3}$$

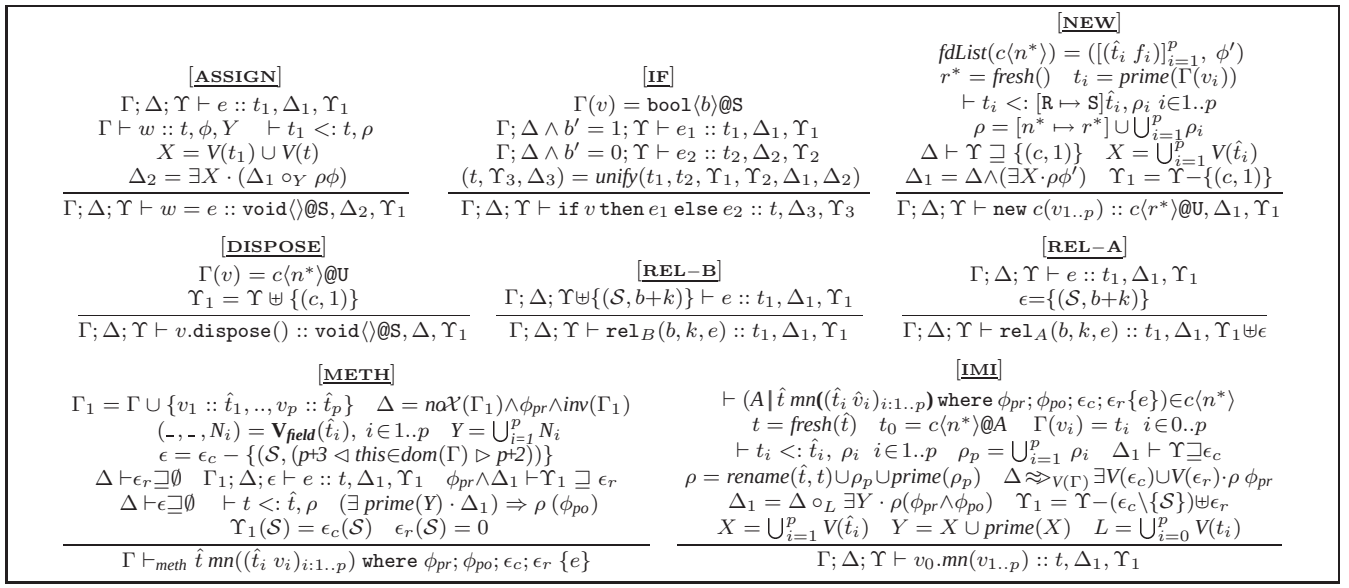

Figure 5. Some Type Rules for Memory Checking

Take note that the *unify* operation merges the post-state constraints and memory usages of the two branches together via a disjunction, as follows:

$$\frac{\begin{array}{c} t = \mathit{unify}(t_1, t_2) \quad \rho_i = \mathit{rename}(t_i, t),\ i{=}1,2 \\ S = \mathit{dom}(\Upsilon_1) \cup \mathit{dom}(\Upsilon_2) \quad \Upsilon = \{(c, n_c) \mid c \in S, n_c = \mathit{fresh}()\} \\ \phi = \rho_1(\phi_1 \wedge \bigwedge_{c \in S}\ n_c{=}\Upsilon_1(c)) \vee \rho_2(\phi_2 \wedge \bigwedge_{c \in S}\ n_c{=}\Upsilon_2(c)) \end{array}}{\mathit{unify}(t_1, t_2, \Upsilon_1, \Upsilon_2, \phi_1, \phi_2) =_{df}\ (t, \Upsilon, \phi)}$$

$$\frac{\tau@A{=}\mathit{msst}(\tau_1@A_1, \tau_2@A_2) \qquad r^* = \mathit{fresh}()}{\mathit{unify}(\tau_1\langle n^*\rangle@A_1, \tau_2\langle m^*\rangle A_2) =_{df}\ \tau\langle r^*\rangle@A}$$

For example, if $\Delta = (s'{=}s)$ and $\Upsilon = \{(\texttt{List}, 2), (\mathcal{S}, 5)\}$, we can obtain $\Delta_3 = (b'{=}1 \wedge s'{=}s{+}1 \wedge d{=}1 \vee b'{=}0 \wedge s'{=}s{-}1 \wedge d{=}3)$ and $\Upsilon_3 = \{(\texttt{List}, d), (\mathcal{S}, 5)\}$. Path-sensitivity adds precision to our analysis and is instrumental in analysing the memory requirement of recursive methods.

5.5 Method Declaration

Each method declaration is checked to see if its definition is consistent with the memory usage specification given in its declaration header by the [METH] rule. The initial memory is built from $\epsilon{=}\epsilon_c{-}\{(\mathcal{S}, \cdots)\}$ after subtracting the method's stack frame. The final available memory of the method body e is Υ_1 which must not be less that the declared net memory release (as specified by $\phi_{pr} \wedge \Delta_1 \vdash \Upsilon_1 \sqsupseteq \epsilon_r$). Perfect recovery of stack space is mandated by $\Upsilon_1(\mathcal{S}) = \epsilon_c(\mathcal{S})$.

For example, consider the following definition of `push` after explicit stack recovery operations have been inserted.

```
L | void⟨⟩@S push(Object⟨⟩@S o)
      where true; n'=n+1; {(List, 1), (S, 5)}; {}
    { List⟨⟩@U tmp = this.head;
      rel_A(2, 3, this.head = new List(o, tmp))}
```

Under the above declaration, we have $\epsilon_c = \{(\texttt{List}, 1), (\mathcal{S}, 5)\}$ and $\epsilon_r = \{\}$. Letting e_b to denote the body, we could type check the method, as shown below.

$$\frac{\begin{array}{c} \Delta_1 \vdash \epsilon_1 \sqsupseteq \{(\texttt{List}, 1)\} \quad \epsilon_2 = \epsilon_1 - \{(\texttt{List}, 1)\} \\ \Delta_2 = \Delta_1 \circ_{\{n\}} n' = m' + 1 \\ \Gamma_1; \Delta_1; \epsilon_1 \vdash \texttt{this.head} = \texttt{new List(o, tmp)} \\ :: \texttt{void}\langle\rangle@\texttt{S}, \Delta_2, \epsilon_2 \end{array} \quad \begin{array}{c} \epsilon_3 = \epsilon_2 \uplus \\ \{(\mathcal{S}, 2{+}3)\} \end{array}}{\begin{array}{c} \Gamma_1; \Delta_1; \epsilon_1 \vdash \mathtt{rel}_A(2, 3, \texttt{this.head} = \texttt{new List(o, tmp)}) :: \\ \texttt{void}\langle\rangle@\texttt{S}, \Delta_2, \epsilon_3 \end{array}}$$

$$\frac{\begin{array}{c} \Gamma; \Delta; \epsilon \vdash \texttt{this.head} :: \texttt{List}\langle s\rangle@\texttt{U},\ \Delta \wedge s{=}n', \epsilon \\ \Gamma_1 = \Gamma + \{\texttt{tmp} :: \texttt{List}\langle m\rangle@\texttt{U}\} \quad \Delta_1 = (\Delta \wedge m'{=}n') \\ \epsilon_1 = \epsilon - \{(\mathcal{S}, 1)\} \quad \Delta \vdash \epsilon \sqsupseteq \{(\mathcal{S}, 1)\} \\ \Gamma; \Delta; \epsilon \vdash e_b :: \texttt{void}\langle\rangle@\texttt{S},\ \Delta_2, \epsilon_3 \\ \Gamma = \{\texttt{this} :: \texttt{Stack}\langle n\rangle@\texttt{L}, o :: \texttt{Object}\langle\rangle@\texttt{S}\} \\ \Delta = (n'{=}n) \quad \epsilon = \{(\texttt{List}, 1), (\mathcal{S}, 5)\} - \{(\mathcal{S}, 4)\} \\ \Delta_2 \Rightarrow (n'{=}n{+}1) \quad \Delta_2 \vdash \epsilon_3 \sqsupseteq \{\} \quad \texttt{true} \vdash \epsilon \sqsupseteq \emptyset \\ \texttt{true} \vdash \{\} \sqsupseteq \emptyset \quad \epsilon_3(\mathcal{S}) = 5 \quad \{\}(\mathcal{S}) = 0 \end{array}}{\begin{array}{c} \{\texttt{this} :: \texttt{Stack}\langle n\rangle@\texttt{L}\} \vdash_{\mathit{meth}} \texttt{void}\langle\rangle@\texttt{S push(Object}\langle\rangle@\texttt{S o)} \\ \texttt{where true}; n'{=}n{+}1; \{(\texttt{List}, 1), (\mathcal{S}, 5)\}; \{\}\ \{e_b\} \end{array}}$$

Note that $\Delta = (n'{=}n)$ captures the initial size constraint on method entry, while ϵ captures the available memory after subtracting the method's stack frame. Also, $\epsilon_3{=}\{(\mathcal{S}, 5)\}$ is the memory space that remains after the evaluation of the method body. This must be no less than the (declared) memory to recover, ϵ_r. Furthermore, stack recovery is expected to be perfect, with $\epsilon_3(\mathcal{S}) = \epsilon_c(\mathcal{S})$. We also check that postcondition is correct with $\Delta_2 \Rightarrow (n'{=}n{+}1)$.

Behaviour subtyping is supported for method overriding. This is captured by the [OVERRIDE] rule in Fig 11. Each method which overrides another is expected to be *contravariant* on its precondition (and memory consumption) and *covariant* on its postcondition (and memory releases), in accordance with the subtyping property of functions.

6. Dynamic Semantics

The dynamic operational semantics is described in small steps. Notations used are defined as follows.

Locations :	ι	$\in$	*Location*
Primitives :	k	$\in$	$prim = \texttt{int} \uplus \texttt{bool} \uplus \texttt{float} \uplus \texttt{null} \uplus \texttt{void}$
Values :	δ	$\in$	$Value = prim \uplus Location$
AnnVal :	ν	$\in$	$AVal = (A^+ \times Value)$
Aliases :	A	$\in$	$A^+ = A \uplus \texttt{U}_D$
Store :	ϖ	$\in$	$Store = Location \rightarrow_{fin} ObjVal$
Variable Env. :	Π	$\in$	$VEnv = Var \rightarrow_{sfin} AVal$
Avail. Mem. :	σ	$=$	$\{ (c, k^{int})^* \mid c \in CN \cup \{\mathcal{S}\}, k^{int} \in Z \}$
Object values :	η	$\in$	$ObjVal = Type \times (Fd \rightarrow_{fin} AVal)$
Type :	$\tau\langle n^* \rangle$	$\in$	$Type = CN \times SV^*$

Note that $f : A \rightarrow_{sfin} B$ denotes a *stackable* mapping from A to B. Such a stackable mapping is defined as:

$$A \rightarrow_{sfin} B =_{df} (A^{\texttt{int}} \rightarrow_{fin} B,\ \texttt{int})$$

where `int` denotes the current frame number of the mapping, while $A^{\texttt{int}}$ denotes the domain that has been marked with frame numbers. Each time a new frame is created, the current frame number is increased by 1. Frame numbers are useful for formulating the lent-once (or limited unique) property for stack frames. Main operators are:

Start a new frame :
$newframe(f, n) = (f, n{+}1)$
Push variables into current frame :
$(f, n) + \{(v \mapsto \nu)^*\} = (f + \{(v^n \mapsto \nu)^*\},\ n)$
Pop variables from the topmost frame :
$(f, n) - \{v^*\} =$
$\texttt{let}\ \hat{f} = f \backslash \{(v^n)^*\}\ \texttt{in}\ (\hat{f},\ max\ \{m \mid \exists u \cdot u^m \in dom(\hat{f})\})$

The variable environment Π is such a stackable mapping. We write $\Pi[\nu/v]$ to denote an update of the value of the latest variable v in Π to ν. We write $\Pi + \{v \mapsto \nu\}$ to denote an extension of Π to include a binding of ν to v, while $\Pi - \{v^*\}$ removes a subset of the mappings. Similar notations are used for the update and enhancement of object values and stores. In the case of store, we use $\varpi - \iota$ to denote the store obtained from ϖ by removing ι from $dom(\varpi)$. We also provide an abbreviated notation $\varpi[\nu/\iota.f] =_{df} \texttt{let}\ (t, \rho) = \varpi(\iota)\ \texttt{in}\ \varpi[(t, \rho[\nu/f])/\iota]$. Given an object value, $\eta = (t, \rho)$, we have $Flds(\eta) =_{df} \rho$ and $type(\eta) =_{df} t$.

We maintain alias annotations for variables in the stack and for fields in the store at run-time. We enhance the current set of alias annotations with a new value: $\texttt{U}_D$. A variable/field can be assigned an alias $\texttt{U}_D$ if its original alias annotation was `U`, and it has since relinquished this uniqueness ownership (of its reference). By maintaining this annotation at run-time (corresponding to dead set for static semantics), our system (both dynamic and static) is able to identify all unique references which have been consumed via read operations. The size and alias instruments are used only for proving the soundness of the type system. They can be erased without affecting the underlying semantics.

We require some intermediate expressions for the dynamic semantics to follow through. Our syntax is thus extended from the original expression syntax as follows:

$$e ::= \cdots \mid \iota \mid \nu \mid \texttt{ret}(v^*, \rho, e)$$

The expression $\texttt{ret}(v^*, \rho, e)$ is used to capture the result of evaluating a local block, or the result of a method invocation. The list of variables associated with `ret` is the local variables declared and used by the block. This set of variables is popped from the stack at the end of the block's evaluation. We also supply ρ to capture the mapping of formal parameters to actual arguments for method call. This is used by the soundness proof.

$$\frac{v^* \subseteq dom(\Gamma) \quad \Gamma; \Sigma; \Delta; \Upsilon \vdash e :: t, \Delta_1, \Upsilon_1 \quad v^* = \{v_1, \ldots, v_p\} \quad X = V(\Gamma(v^*)) \quad Y = X \cup prime(X) \quad \rho_2 = \bigcup \{\hat{\rho}_i \mid t_i = \Gamma(v_i),\ \hat{t}_i = \Gamma(\rho\, v_i),\ \vdash \hat{t}_i <: t_i, \hat{\rho}_i\}_{i=1}^{p}}{\Gamma; \Sigma; \Delta; \Upsilon \vdash \texttt{ret}(v^*, \rho, e) :: t, \exists Y \cdot (prime(\rho_2)\Delta_1), \Upsilon_1}\ [\textbf{ELF}]$$

$$\frac{}{\Gamma; \Sigma; \Delta; \Upsilon \vdash \iota :: \Sigma(\iota)@\texttt{U}, \Delta, \Upsilon}\ [\textbf{LOC}]$$

$$\frac{\Gamma; \Sigma; \Delta; \Upsilon \vdash \delta :: t, \Delta_1, \Upsilon_1 \quad A_t = ann(t) \quad A_t \leq_a A}{\Gamma; \Sigma; \Delta; \Upsilon \vdash (A, \delta) :: [A_t \mapsto A]t, \Delta_1, \Upsilon_1}\ [\textbf{A-DATA}]$$

Figure 6. Type Rules for Intermediates

The type rules for intermediate expressions are given in Fig 6. The subsumption rules for size and memory are given below.

$$\frac{\Gamma; \Sigma; \Delta; \Upsilon \vdash e :: t, \Delta_1, \Upsilon_1 \quad \Delta_1 \vdash \Upsilon_1 \sqsupseteq \Upsilon_2 \quad \Delta_1 \Rightarrow \Delta_2}{\Gamma; \Sigma; \Delta; \Upsilon \vdash e :: t, \Delta_2, \Upsilon_2}\ [\textbf{SUBS1}]\ :\ \text{(Covariant)}$$

$$\frac{\Gamma; \Sigma; \Delta_1; \Upsilon_1 \vdash e :: t, \Delta, \Upsilon \quad \Delta_2 \vdash \Upsilon_2 \sqsupseteq \Upsilon_1 \quad \Delta_2 \Rightarrow \Delta_1}{\Gamma; \Sigma; \Delta_2; \Upsilon_2 \vdash e :: t, \Delta, \Upsilon}\ [\textbf{SUBS2}]\ :\ \text{(Contravariant)}$$

The dynamic evaluation rules are of the following form.

$$\langle \Pi, \varpi, \sigma \rangle\ [e] \hookrightarrow \langle \Pi_1, \varpi_1, \sigma_1 \rangle\ [e_1]$$

We shall formulate the rules using an exception-style semantics with four possible errors, namely $\mathbf{E} =$ **Err-Alias** | **Err-Mem** | **Err-Null** | **Err-Prim**. Whenever one such error is raised, the evaluation aborts. This error occurrence can be stated using $\langle \Pi, \varpi, \sigma \rangle\ [e] \hookrightarrow \mathbf{E}$. The complete set of evaluation rules is given in Fig. 7.

7. Soundness

We shall formulate and prove several novel safety properties that our type system possess. Our alias checking rules ensures that well-typed programs are subject to several alias properties, including:

- the *uniqueness* property: all unique references are unaliased during the evaluation;
- the *lent-once* property: each unique reference can only be lent once within each stack frame; and
- the *read-only* property: `R`-mode fields never change during evaluation.

We have formally defined and proven all these properties for an object-based imperative language OIMP in an earlier work [13]. The main new properties to prove concern the safety of `dispose` and the soundness of memory specification. Before stating a theorem on the safety of `dispose`, we give several formal definitions.

DEFINITION 1 (Liveness & No-Dangling).

- *A runtime value δ is said to* live *wrt the store ϖ, denoted as $live(\delta, \varpi)$, if δ is a primitive k, or $\delta \in dom(\varpi)$.*
- *A runtime environment (Π, ϖ) is said to be* no-dangling *wrt the dead set Θ, denoted as $\Theta \vdash_{noDang} \langle \Pi, \varpi \rangle$, if the following hold:*
  - $\forall v \in dom(\Pi) - \Theta \cdot live(snd(\Pi(v)), \varpi)$
  - $\forall \iota \in dom(\varpi) \cdot (\_, \rho, \_) = \varpi(\iota) \wedge (\forall f \in dom(\rho) \cdot live(\iota, f, \Pi, \Theta) \Rightarrow live(snd(\rho(f)), \varpi))$

Figure 7. Dynamic Semantics

Note that $live(\iota, f, \Pi, \Theta) =_{df} \neg(\exists v{\in}dom(\Pi){\cdot}\Pi(v){=}(\_,\iota){\wedge}\ \ v.f{\in}\Theta)$, while $snd((A,\delta)) =_{df} \delta$.

DEFINITION 2 (Alias Consistency). *The alias consistency relation between static and dynamic semantics is defined as follows:*

$$\frac{\begin{array}{c} dom(\Pi) = dom(\Gamma) \quad dom(\varpi) = dom(\Sigma) \\ \forall v \in dom(\Gamma) \cdot (v \notin \Theta \Rightarrow ann(\Pi(v)) \leq_a ann(\Gamma(v))) \\ \forall \iota \in dom(\varpi) \cdot \rho = Flds(\varpi(\iota)) \wedge \forall f \in dom(\rho) \cdot \\ (live(\iota, f, \Pi, \Theta) \Rightarrow ann(\rho(f)) \leq_a ann\text{-}fd(\Sigma(\iota), f)) \\ \Theta \vdash_{noDang} \langle \Pi, \varpi \rangle \qquad \forall \iota \in locs(e) \cdot \iota \in dom(\varpi) \end{array}}{\Gamma; \Sigma; \Theta \models_{\mathcal{A}} \langle \Pi, \varpi \rangle}$$

Note that $ann\text{-}fd(c\langle u^* \rangle, f) =_{df} A$, where $(\tau\langle m^* \rangle @A\ f) \in c\langle n^* \rangle$. We use *locs* to collect all intermediate locations appearing in e:

$$\begin{array}{lcl} locs(e) =_{df} \ case\ e\ of & & \\ \quad \iota & \rightarrow & \{\iota\} \\ \quad \mathtt{ret}(v^*, \rho, e) \mid w = e \mid \mathtt{rel}_A(k_1, k_2, e) & & \\ \quad \mid \mathtt{rel}_B(k_1, k_2, e) \mid (t\ v = e; e_2) & \rightarrow & locs(e) \\ \quad \mathtt{if}\ v\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 \mid v.\mathtt{dispose}() & & \\ \quad \mid k \mid w \mid \mathtt{new}\ c(v^*) \mid [v.]mn(v^*) & \rightarrow & \emptyset \end{array}$$

The following theorem states that no-dangling property is preserved (together with other alias consistency properties) during evaluation of well-typed expressions.

THEOREM 1 (No-Dangling). *If*

$$\frac{\begin{array}{c} \Gamma; \Sigma; \Theta \vdash eraseS(e) :: t, \Theta_1 \\ \Gamma; \Sigma; \Theta \models_{\mathcal{A}} \langle \Pi, \varpi \rangle \\ \langle \Pi, \varpi, \sigma \rangle\ [e] \hookrightarrow \langle \Pi_1, \varpi_1, \sigma_1 \rangle\ [e_1] \end{array}}{\begin{array}{l} \text{then there exist } \Gamma_\alpha, \Theta_\alpha, \text{ and } \Sigma_\alpha \supseteq \Sigma, \text{ such that} \\ \Gamma - diff(e, e_1) = \Gamma_\alpha - diff(e_1, e) \\ \Gamma_\alpha; \Sigma_\alpha; \Theta_\alpha \vdash eraseS(e_1) :: t, \Theta_1 \\ \Gamma_\alpha; \Sigma_\alpha; \Theta_\alpha \models_{\mathcal{A}} \langle \Pi_1, \varpi_1 \rangle \end{array}}$$

Note that $eraseS(e)$ denotes the expression obtained by erasing all size annotations from e. Function $diff(e, e_1)$ returns a list of local variables that appears in e but not e_1 :

$$\begin{array}{lllll} diff(e, e_1) =_{df} & let & lst & = & local(e) \\ & & lst1 & = & local(e_1) \\ & & n & = & length(lst) - length(lst1) \\ & in & (take(n, lst) \lhd n \geq 0 \rhd [\,]) & & \end{array}$$
$$take(n, lst) =_{df} ([\,] \lhd n{\leq}0 \rhd [hd(lst)] {+\!\!+} take(n{-}1, tl(lst)))$$

Function $local(e)$ returns a list of sets of local variables. It is defined as follows:

$$\begin{array}{lcl} local(e) =_{df} \ case\ e\ of & & \\ \quad \mathtt{ret}(v^*, \rho, e) & \rightarrow & local(e) {+\!\!+} [\{v^*\}] \\ \quad w = e \mid \mathtt{rel}_A(k_1, k_2, e) \mid \mathtt{rel}_B(k_1, k_2, e) & \rightarrow & local(e) \\ \quad (t\ v = e_1; e_2) & \rightarrow & local(e_1) \\ \quad \mathtt{if}\ v\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 \mid \delta \mid w & & \\ \quad \mid \mathtt{new}\ c(v^*) \mid [v.]mn(v^*) \mid v.\mathtt{dispose}() & \rightarrow & [\,] \end{array}$$

Note that $\Gamma - [\,] =_{df} \Gamma,\ \Gamma - ([s] {+\!\!+} S) =_{df} (\Gamma - s) - S$.

Proof: By induction over the depth of alias-type derivation for expression $eraseS(e)$. Details are given in the Appendix. □

Before presenting our main theorem, we define the overall *consistency relation* between static and dynamic semantics as follows.

DEFINITION 3 (Consistency Relation).

$$\frac{\begin{array}{c} \Gamma; \Sigma; \Theta \models_{\mathcal{A}} \langle \Pi, \varpi \rangle \\ D_\Theta = prime(\bigcup_{w \in \Theta} \{Z \mid \Gamma \vdash w :: t, \phi, Z\}) \\ \forall v \in dom(\Gamma) \cdot \Delta_v = size(\Gamma(v), \Pi(v), \varpi) \\ X = (V(\Delta) - prime(V(\Gamma))) \cup D_\Theta \\ \Delta_\Pi = \wedge_{v \in dom(\Gamma)} \Delta_v \quad \Delta_\Pi \Rightarrow \exists X \cdot \Delta \quad \Delta \vdash \sigma \sqsupseteq \Upsilon \end{array}}{\Gamma; \Sigma; \Delta; \Theta; \Upsilon \models \langle \Pi, \varpi, \sigma \rangle}$$

In the above, *size* computes the size of a run-time value, and presents it in constraint form. It is defined as follows:

$$\begin{array}{l} size(\mathtt{int}\langle r \rangle, (A, k), \varpi) =_{df} (r' = k) \\ size(\mathtt{bool}\langle r \rangle, (A, k), \varpi) =_{df} (r' = (1 \lhd k = \mathtt{true} \rhd 0)) \\ size(cn\langle r^* \rangle, (A, \iota), \varpi) =_{df} \ let \quad (cn\langle n_{1..p} \rangle, \rho_f) = \varpi(\iota) \\ \qquad\qquad\qquad\qquad\qquad\qquad \rho = getSize(\iota, \varpi) \\ \qquad\qquad\qquad\qquad\qquad\quad in\ \ \wedge_{i=1}^{p} \{r'_i = (\rho\ n_i)\} \\ size(t, (A, \_), \varpi) =_{df} True \end{array}$$

$$\begin{array}{l} getSize\ ::\ (Location \times Store) \rightarrow (SV \rightarrow_{fin} \mathtt{int}) \\ getSize(\iota, \varpi) = getSize1(\iota, \varpi, [\iota]) \\ getSize1(\iota, \varpi, activeLst) = \\ \quad fdList(cn\langle n_{1..q} \rangle) = ([\tau_i \langle m_i^* \rangle @\hat{A}_i\ f_i]_{i=1}^{p}, \phi); \\ \quad (cn\langle r_{1..q} \rangle,\ \{f_i \mapsto (A_i, \delta_i)\}_{i=1}^{p}) = \varpi(\iota); \\ \quad \rho = []; \\ \quad \rho_\iota = \{n_i \mapsto r_i\}_{i=1..q}; \\ \quad forall\ i = 1, .., p \quad \{ \\ \quad\quad if\ (\tau_i = \mathtt{int}) \quad \rho = \{m_i \mapsto \delta_i\} \uplus \rho; \\ \quad\quad if\ (\tau_i = \mathtt{bool}) \quad \rho = \{m_i \mapsto 1 \lhd \delta_i = \mathtt{true} \rhd 0\} \uplus \rho; \\ \quad\quad if\ (\tau_i = \mathtt{float} \mid \mathtt{void}) \quad \rho = \{m_i \mapsto 0\} \uplus \rho; \\ \quad\quad if\ (member\ (\delta_i, activeLst)) \quad \rho = \{m_i \mapsto 0\} \uplus \rho \\ \quad\quad else\ \{ \quad \rho_1 = getSize1(\delta_i, \varpi, add(\delta_i, activeLst)); \\ \quad\quad\quad\quad \rho = \rho_1 \uplus \rho\} \\ \quad \} \\ \quad \uplus\{(\rho_\iota n_i) \mapsto (\rho\ \alpha_i) \mid \phi = \wedge_{i=1..q}(n_i = \alpha_i)\} \end{array}$$

The following main theorem states that each well-typed expression preserves its type under reduction with a runtime environment and a store which are consistent with the compile-time counterparts. Furthermore, the final size constraint is consistent with the value obtained on termination.

We use $\Gamma; \Sigma; \Delta; \Theta; \Upsilon \vdash e :: t, \Delta_1, \Theta_1, \Upsilon_1$ to represent the overall type judgement. That is,

$$\frac{\Gamma; \Sigma; \Theta \vdash eraseS(e) :: eraseS(t), \Theta_1 \quad \Gamma; \Sigma; \Delta; \Upsilon \vdash e :: t, \Delta_1, \Upsilon_1}{\Gamma; \Sigma; \Delta; \Theta; \Upsilon \vdash e :: t, \Delta_1, \Theta_1, \Upsilon_1}$$

THEOREM 2 (Preservation).

(a) (Expression) If

$$\frac{\begin{array}{c} \Gamma; \Sigma; \Delta; \Theta; \Upsilon \vdash e :: t, \Delta_1, \Theta_1, \Upsilon_1 \\ \Gamma; \Sigma; \Delta; \Theta; \Upsilon \models \langle \Pi, \varpi, \sigma \rangle \\ \langle \Pi, \varpi, \sigma \rangle\ [e] \hookrightarrow \langle \Pi_1, \varpi_1, \sigma_1 \rangle\ [e_1] \end{array}}{\begin{array}{l} \text{then exist } \Sigma_\alpha \supseteq \Sigma, \Gamma_\alpha, \Delta_\alpha, \Theta_\alpha, \text{ and } \Upsilon_\alpha, \text{ such that} \\ \Gamma - diff(e, e_1) = \Gamma_\alpha - diff(e_1, e) \\ \Gamma_\alpha; \Sigma_\alpha; \Delta_\alpha; \Theta_\alpha; \Upsilon_\alpha \vdash e_1 :: t, \Delta_1, \Theta_1, \Upsilon_1 \\ \Gamma_\alpha; \Sigma_\alpha; \Delta_\alpha; \Theta_\alpha; \Upsilon_\alpha \models \langle \Pi_1, \varpi_1, \sigma_1 \rangle\ . \end{array}}$$

(b) (Value) If

$$\frac{\begin{array}{c} \Gamma; \Sigma; \Delta; \Theta; \Upsilon \vdash (A, \delta) :: t, \Delta_1, \Theta_1; \Upsilon_1 \\ \Gamma; \Sigma; \Delta; \Theta; \Upsilon \models \langle \Pi, \varpi, \sigma \rangle \end{array}}{\begin{array}{l} \text{then the following holds:} \\ \Theta = \Theta_1 \\ \Gamma + \{x :: t\}; \Sigma; \Delta_2; \Theta_1; \Upsilon_2 \models \langle \Pi + \{x \mapsto (A, \delta)\}, \varpi, \sigma_1 \rangle \\ \text{where } x = fresh()\ ,\ \Delta_2 = [v \mapsto v']_{v \in V(t)} \Delta_1, \text{ and} \\ \Upsilon_2 = \Upsilon_1 \uplus \{(\mathcal{S}, 1)\}, \sigma_1 = \sigma \uplus \{(\mathcal{S}, 1)\}. \end{array}}$$

Proof: By induction over the depth of type derivation for expression e. Details are given in the Appendix. □

The second safety theorem on progress captures the fact that well-typed programs cannot go wrong. Specifically, this theorem guarantees that no memory adequacy errors (denoted by **Err-Mem**) are ever encountered for well-typed MEMJ programs, as follows:

THEOREM 3 (Progress). *If*
$\Gamma; \Sigma; \Delta; \Theta; \Upsilon \vdash e :: t, \Delta_1, \Theta_1, \Upsilon_1$ *and* $\Gamma; \Sigma; \Delta; \Theta; \Upsilon \models \langle \Pi, \varpi, \sigma \rangle$,

then either e *is a value, or* $\langle \Pi, \varpi, \sigma \rangle\ [e] \hookrightarrow$ **Err-Null**, *or there exist* $\Pi_1, \varpi_1, \sigma_1, e_1$ *such that* $\langle \Pi, \varpi, \sigma \rangle\ [e] \hookrightarrow \langle \Pi_1, \varpi_1, \sigma_1 \rangle\ [e_1]$.

Proof: By induction over the depth of type derivation for expression e. Details are given in the Appendix. □

8. Memory Inference

The goal of memory inference is to derive heap usage effects for each method. We provide a summary-based approach that considers each set of strongly-connected methods (bottoms-up order) for inference through the following steps:

- Calculate symbolic program state, build constraint abstraction and collect memory adequacy constraints.
- Solve constraint abstraction using fixpoint analysis.
- Derive memory availability.
- Derive memory requirement.

Formally, the type inference rule for expression has the form:

$$\Gamma; \Delta; \Upsilon \vdash e :: t, \Delta_1, \Upsilon_1, \Phi$$

where type environment Γ maps variables to their annotated types, $\Upsilon(\Upsilon_1)$ are memory available before and after evaluation of e, respectively, $\Delta(\Delta_1)$ are the symbolic program states before and after evaluation of expression e, respectively. Φ is a set of (Δ, φ) pairs where φ is the constraint that enforces memory adequacy and Δ is the program state where φ was generated. We carry the program state along to allow memory requirement to be accurately derived and suitably simplified.

The inference rule for methods has the following form:

$$\Gamma \vdash_{meth} meth \hookrightarrow meth_1 \mid \mathcal{Q}$$

where Γ is empty for static methods or Γ contains a single `this` entry for instance methods. The method $meth_1$ is the transformed version of *meth* where memory effects annotations are added. The constraint abstraction $\mathcal{Q}$ captures the relations between the sizes of the method's parameters and its memory effects.

In our approach, we compute memory availability in a forward manner, taking into account conditional paths and the memory consumption/release for each subexpression. Simultaneously, we also gather memory requirement in a backward manner by expressing the safety of each consumption in terms of a memory adequacy constraint on the original parameters. As stated, we capture these two pieces of information as a pair (Δ, φ) for each program point where a memory consumption is required. Full details of memory inference system and its implementation is described in [28]. In this paper, we highlight key features of our memory inference system.

8.1 Deriving Memory Requirements

We can convert each memory adequacy constraint φ into a memory requirement by the following formula: $pre \equiv \Delta \approx\!\!>_{V(\varphi)} \varphi$, where Δ is the symbolic state at the program point where φ is generated. We may then project *pre* as a formula of the method's parameters U, as follows:

$$\forall W \cdot pre \texttt{ where } W{=}V(pre){-}U$$

This projection effectively eliminates all free size variables in a formulae ϕ by means of universal quantification, except for those specified in U. For example, consider:

$$\begin{array}{rl} \texttt{f(n)} = & \texttt{if n} \leq \texttt{0 then } \textit{allocate 3 Objects} \\ & \texttt{else } \textit{allocate 2 Objects}\ ;\ \texttt{f(n} - \texttt{1)} \end{array}$$

Using our inference method, we can derive a memory requirement $\{(\texttt{Object}, d)\}$ for this method, together with the constraint $(n{\leq}0{\wedge}d{=}3){\vee}(n{>}0{\wedge}d{=}2n{+}3)$ that will be added to the method's precondition. Disjunction is used to help capture memory effects accurately, where possible.

8.2 Fixpoint Analysis

For each recursive method, we construct a constraint abstraction that relates the sizes of the input parameters, the amount of memory available at the beginning of the method to the sizes of the parameters and the amount of memory available prior to the recursive call. This one-step relation is subjected to a fixpoint procedure to compute its multi-step relation. Let $I\langle n^*, m^*, \hat{n}^*, \hat{m}^* \rangle$ be the one-step relation. The fixpoint computation is formulated below with n^* and m^* being the sizes of the input parameters and memory available at the beginning, whereas $\hat{n}^*$ and $\hat{m}^*$ are those of the recursive call.

$$I_1\langle n^*, m^*, \hat{n}^*, \hat{m}^* \rangle = I\langle n^*, m^*, \hat{n}^*, \hat{m}^* \rangle$$
$$I_{i+1}\langle n^*, m^*, \hat{n}^*, \hat{m}^* \rangle = I_i\langle n^*, m^*, \hat{n}^*, \hat{m}^* \rangle \vee \exists n_0^*, m_0^* \cdot I_i\langle n_0^*, m_0^*, \hat{n}^*, \hat{m}^* \rangle \wedge I\langle n^*, m^*, n_0^*, m_0^* \rangle$$

For the computation to converge, we may need to apply standard techniques such as hulling and widening [15].

For example, consider the following one-step relation for the *moverec2* method from Sec 4.

$$moverec2\langle a, b, p, m, \hat{a}, \hat{b}, \hat{p}, \hat{m} \rangle = \hat{a}{=}a{\wedge}\hat{b}{=}b{+}1{\wedge}\hat{p}{=}p{-}1{\wedge}\hat{m}{=}m{-}1$$

Following fix-point analysis, we obtain:

$$moverec2\langle a, b, p, m, \hat{a}, \hat{b}, \hat{p}, \hat{m} \rangle = \hat{a}{=}a{\wedge}\hat{p}{-}p{=}b{-}\hat{b}{\wedge}\hat{m}{=}m{+}\hat{p}{-}p{\wedge}b{<}\hat{b}$$

From this, our inference can derive both the expected memory requirement and memory availability.

9. Implementation

We have constructed a type checker for MEMJ, and have also built a preprocessor to allow a more expressive language to be accepted. The entire prototype was built using the Glasgow Haskell compiler[29] where we have added a library (based on [30]) for Presburger arithmetic constraint-solving.

The main objective of our initial experiments is to show that our memory usage specification mechanism is expressive and that such an advanced form of type checking is viable. We converted to MEMJ a set of programs from the Java version of the Olden benchmark suite [8] and another set of smaller programs from the RegJava benchmark[14], before subjecting them to memory adequacy checking.

Figure 8 summarises the statistics obtained for each program that we have verified via our type checker. Column 3 illustrates the size and memory annotation overheads which must be made in the header declarations of each class and method. Columns 4 and 5 highlight the CPU times used (in seconds) for alias and memory checking, respectively. Our experiments were done under Redhat Linux 9.0 platform on Pentium 2.4 GHz with 768MB main memory. Except for the perimeter program (which has more conditionals from using a quadtree data structure), all programs take under 10 seconds to verify, despite them being of moderate sizes. We attribute this to the fact that memory declarations are verified in a modular fashion for each method definition. We achieve this despite our reliance on Presburger arithmetic whose worst case time complexity is exponential in the size of formula being solved. The last column highlights the number of methods that have been successfully verified as using memory spaces that are bounded by symbolic Presburger formulae. Ackermann function could *not* be so bounded, as it requires a stack space that is exponential in the size of its original input. A function in Voronoi also has an allocation inside a loop

Programs	Size (lines)		Checking (in sec.)		Verified
	Source	Ann.	Alias	Memory	Methods
bisort	340	7	0.01	2.56	6/6
em3d	462	19	0.05	1.14	20/20
health	562	22	0.05	6.37	15/15
mst	473	31	0.02	1.26	22/22
power	765	24	0.06	4.28	19/19
treeadd	195	6	0.02	0.32	4/4
tsp	545	10	0.02	3.54	9/9
perimeter	745	12	0.02	21.81	8/8
n-body	1128	31	0.60	1.25	22/22
Voronoi	1000	45	0.03	3.51	39/40
stack	122	12	0.01	0.08	10/10
sieve	88	7	0.01	0.09	6/6
m-sort	183	13	0.01	0.36	12/12
life	164	9	0.02	2.95	7/7
Ackermann	15	1	0.01	0.17	0/1
Mandelbrot	194	11	0.01	1.72	10/10
Reynolds3	98	6	0.01	0.18	4/4

Figure 8. Type Checking Experimental Results

that could *not* be bounded by our system. This is due to a complex termination condition used that could not be captured by Presburger arithmetic. We have also conducted memory inference on our benchmark programs. The current prototype inference system takes between 4 to 10 times longer than the type-checking system.

We have also conducted a set of experimental results to evaluate on the effectiveness of memory inference, in conjunction with our explicit memory recovery scheme. We modified IBM's Jikes RVM[2, 25] to provide support for explicit dispose operation and instrumented its memory system to capture total allocation (c) and actual high watermark (b). We then compare it against the predicted memory requirement (a) from our memory inference. We count the number of objects created and reused. As can be seen in Fig 9, our memory inference is accurate for four out of the five programs from the RegJava benchmark. We are conservative on Reynolds program as one of its memory requirement is $log(n)$ which is approximated to n in the Presburger constraint. Alternatively, if the depth of tree had been used as a parameter, our inference would also be precise on the Reynolds example. Except for *sieve*, most of the programs have high degree of memory reuse which were facilitated by our use of the *dispose* operation for explicit memory recovery.

10. Related Work

Past research on improving memory models for object-oriented paradigm has focused largely on efficiency, minimization and safety. We are unaware of any prior work on analysing heap memory usage by OO programs for the purpose of checking for memory adequacy. The closest related work on memory adequacy are based on first-order functional paradigm, where data structures are mostly immutable and thus easier to handle. We shall review two recent works in this area before discussing other complimentary works.

Hughes and Pareto [24] proposed a type and effect system on space usage estimation for a first-order functional language, extended with region language constructs of Tofte and Talpin's[34]. Their sized-region type system maintains upper bounds on the height of the run-time stack, and on the memory used by regions. The use of region model facilitates recovery of heap space. However, as each region is only deleted when all its objects become dead, more memory than necessary may be used, as reported by [4]. Stack usage has been modelled in Hughes and Pareto's work, but tail-call optimization is not supported.

More recently, Hofmann and Jost [23] proposed a solution to obtain linear bounds on the heap space usage of first-order functional programs. A key feature of their solution is the use of linear typing which allows the space of each last-use data constructor (or record) to be directly recycled by a matching allocation. With this approach, memory recovery can be supported within each function, but not across functions unless the dead data structures are explicitly passed. (For example, they cannot handle the `moverec` and `emptyStack` methods since these methods require space reuse across functions.) Moreover, their model does not track the symbolic sizes of data structures. Nevertheless, one significant advance of their work is an inference mechanism for memory effects through linear programming (LP) technique. This technique is similar to the derivation framework proposed by Rugina and Rinard[32] for inferring symbolic bounds. The main advantage of LP technique is that no fix-point analysis is required. However, it restricts the expected memory effects to a linear form without disjunction.

In support of component-based software, Krone et.al. [26] proposed an approach to modular verification of performance (time and space) constraints using pre and post contracts. However, their framework appears to be at a preliminary stage as neither an implementation nor a specific verification technology has been reported. Recently, there have also been several works on analysing the stack space requirement for interrupt-driven programs. Brylow et.al. [7] proposed a stack size analysis using context-free reachability algorithm based on model checking. Chatterjee et.al. [10] investigated the complexity on the stack boundedness problem and the exact maximum stack size problem. Regehr et.al. [31] enhanced previous work with a more accurate context-sensitive abstract interpretation and also advocated for function inlining to reduce stack depth. These techniques apply to a class of interrupt-driven programs but are neither applicable to recursive programs nor to those whose stacks depend on some loop bounds.

Other related works on improving memory models are reviewed here. Chen et. al. [11] reported the use of heap compression techniques to support memory-constrained Java applications. They proposed a set of memory management strategies and associated garbage collection algorithms to reduce heap footprint of embedded Java applications that execute under severe memory constraints. The key techniques employed are object compression, lazy allocation and object break-down. On a set of embedded Java applications, they reported noticeable reduction of heap space requirement, while performance degradations are fairly small in most cases. Berger, Zorn and Mckinley [4] generalized regions and heaps, by allowing programmers to delete individual objects while keeping the high performance of regions. They showed that this approach can reduce memory consumption significantly. Ananian and Rinard [3] presented a set of techniques for reducing the memory consumption of object-oriented programs, by optimizing on the representation of object fields.

These recent works on optimizing memory systems are complimentary to our current efforts on analysing memory usage. As a matter of fact, we have considered how the present work could be applied to region-based memory management system[12]. As already confirmed by others[21, 4], we expect noticeable performance improvements when regions are size bounded with explicit recovery. In the opposite direction, we expect region-based system to further provide timely recovery for shared objects that have become dead, providing us with tighter memory bounds.

11. Concluding Remarks

We have proposed a memory usage type system for a non-trivial object-oriented core language. We have designed a flexible specification mechanism to allow memory needs of user programs to be declared abstractly, and then verifies if memory adequacy property holds for the given definitions. Our approach requires heap space

Program	Input Size	Prediction (a)	Actual (b)	Allocation (c)	Reuse (b/c)	Accuracy (b/a)
sieve	10000	10000	9999	10000	0.9999	0.9999
m-sort	10000	20000	20000	287232	0.0696	1.0000
life	1000	2	2	1000	0.0020	1.0000
Mandelbrot	100	4	4	83692	0.00005	1.0000
Reynolds	10000	40000	20014	40000	0.5004	0.5004

Figure 9. Experimental Results on Memory Prediction and Recovery

to be explicitly deallocated, which can be handled automatically. We have also built a prototype type checker to confirm the viability and practicality of our approach. We envision our framework to be useful for embedded system, where memory is considered to be a critical resource. We also envision the synergy of predicable memory bounds with region-based memory management systems. In particular, bounded memory regions can result in better performance. Synergistically, region-based system can provide timely recovery for shared objects that are dead, providing us with tighter memory bounds.

Acknowledgement The authors would like to acknowledge the invaluable help of Florin Craciun with the evaluation of a set of the benchmark programs.

References

[1] J. Aldrich, V. Kostadinov, and C. Chambers. Alias Annotation for Program Understanding. In *ACM OOPSLA*, Seattle, Washington, November 2002.

[2] B. Alpern, D. Attanasio, J. J. Barton, A. Cocchi, S. F. Hummel, D. Lieber, M. Mergen, T. Ngo, J. Shepherd, and S. Smith. Implementing Jalapeño in Java. In *ACM OOPSLA*, Denver, Colorado, November 1999.

[3] C.S. Ananian and M. Rinard. Data Size Optimization for Java Programs. In *Proceedings of the ACM Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES '03)*, San Diego, California, June 2003.

[4] E. D. Berger, B. G. Zorn, and K. S. Mckinley. Reconsidering Custom Memory Allocation. In *ACM OOPSLA*, November 2002.

[5] C. Boyapati, A. Salcianu, W. Beebee, and M. Rinard. Ownership Types for Safe Region-Based Memory Management in Real-Time Java. In *ACM PLDI*, San Diego, California, June 2003.

[6] J. Boyland, J. Noble, and W. Retert. Capabilities for Sharing: A Generalization of Uniqueness and Read-Only. In *ECOOP*, Budapest, Hungary, June 2001.

[7] D. Brylow, N. damgaard, and J. Palsberg. Static Checking of Interrupt-Driven Software. In *Proceedings of the International Conference on Software Engineering*, Toronto, Canada, May 2001.

[8] M. C. Carlisle and A. Rogers. Software caching and computation migration in Olden. In *4th Principles and Practice of Parallel Programming*, Santa Barbara, California, May 1993.

[9] E. C. Chan, J. Boyland, and W. L. Scherlis. Promises: Limited Specifications for Analysis and Manipulation. In *Proceedings of the International Conference on Software Engineering*, pages 167–176, Kyoto, Japan, April 1998.

[10] K. Chatterjee, D. Ma, R. Majumdar, T. Zhao, T. A. Henzinger, and J. Palsberg. Stack Size Analysis for Interrupt-Driven Programs. In *Proceedings of the 10th Annual International Static Analysis Symposium (SAS '03), LNCS 2694*, pages 109–126, San Diego, California, June 2003.

[11] G. Chen, M. Kandemir, N. Vijaykrishnan, M. J. Irwin, B. Mathiske, and M. Wolczko. Heap Compression for Memory-Constrained Java Environments. In *ACM OOPSLA*, October 2003.

[12] W.N. Chin, F. Craciun, S.C. Qin, and M. Rinard. Region Inference for an Object-Oriented Language. In *ACM PLDI*, Washington, D.C., June 2004.

[13] W.N. Chin, S.C. Khoo, and S.C. Qin. A Sized Type System for Objects with Alias Controls. Technical report, SoC, Natl Univ. of Singapore, January 2004. avail. at http://www.comp.nus.edu.sg /~qinsc/papers/sizedtype.ps.gz.

[14] M. V. Christiansen and P. Velschow. Region-Based Memory Management in Java. Master's Thesis, Department of Computer Science (DIKU), University of Copenhagen, 1998.

[15] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *ACM POPL*, pages 84–96. ACM Press, 1978.

[16] K. Crary, D. Walker, and G. Morrisett. Typed memory management in a calculus of capabilities. In *ACM POPL*, pages 262–275, San Antonio, January 1999. ACM Press.

[17] Robert DeLine and Manuel Fahndrich. Enforcing high-level protocols in low-level software. In *Proceedings of the ACM Conference on Program Language Design and Implementation*, June 2001.

[18] M. Fahndrich and R. Leino. Declaring and checking non-null types in an object-oriented language. In *ACM OOPSLA*, Anaheim, CA, October 2003.

[19] Manuel Fahndrich and Robert DeLine. Adoption and focus: Practical linear types for imperative programming. In *Proceedings of the ACM Conference on Program Language Design and Implementation*, June 2002.

[20] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney. Region-Based Memory Management in Cyclone. In *ACM PLDI*, June 2002.

[21] N. Hallenberg, M. Elsman, and M. Tofte. Combining Region Inference and Garbage Collection. In *ACM PLDI*, Berlin, Germany, June 2002.

[22] C. A. R. Hoare and J. He. *Unifying Theories of Programming*. Prentice-Hall, 1998.

[23] M. Hofmann and S. Jost. Static prediction of heap space usage for first order functional programs. In *ACM POPL*, New Orleans, Louisiana, January 2003.

[24] J. Hughes and L. Pareto. Recursion and Dynamic Data-Structures in Bounded Space: Towards Embedded ML Programming. In *Proceedings of the International Conference on Functional Programming (ICFP '99)*, September 1999.

[25] IBM. Jikes™ Research Virtual Machine (RVM). http://www-124.ibm.com/developerworks/oss/jikesrvm/.

[26] J. Krone, W. F. Ogden, and M. Sitaraman. Modular verification of performance constraints. Technical Report RSRG-03-04, Department of Computer Science, Clemson University, Clemson, SC 29634-0974, May 2003.

[27] L. Lamport. The temporal logic of actions. *ACM Trans. on Programming Languages and Systems*, 16(3):872–923, May 1994.

[28] Huu Hai Nguyen. Memory Usage Inference for Object-Oriented Programs. Technical report, CS Programme, Singapore-MIT Alliance, July 2004. (Term Paper).

[29] S Peyton-Jones and et al. Glasgow Haskell Compiler. http://www.haskell.org/ghc.

[30] W. Pugh. The Omega Test: A fast practical integer programming algorithm for dependence analysis. *Communications of the ACM*, 8:102–114, 1992.

[31] J. Regehr, A. Reid, and K. Webb. Eliminating Stack Overflow by Abstract Interpretation. In *Proceedings of the 3rd International*

Conference on Embedded Software, Philadelphia, PA, October 2003. Springer-Verlag.

[32] R. Rugina and M. Rinard. Symbolic bounds analysis of pointers, array indices, and accessed memory regions. In *ACM PLDI*, pages 182–195. ACM Press, June 2000.

[33] M. Tofte and J. Talpin. Implementing the Call-By-Value λ-calculus Using a Stack of Regions. In *ACM POPL*, January 1994.

[34] M. Tofte and J. Talpin. Region-based memory management. *Information and Computation*, 132(2), 1997.

APPENDIX

12. Alias Checking Rules

The set of alias checking rules are given in Fig. 10.

The auxiliary relation $\Theta \vdash \mathit{consumeU}(w, A), \Theta_1$ (named [AUX1]) adds w to the *dead set*, if A is unique. This relation ensures that uniqueness is not consumed twice, and that each object and its field are not both dead at the same time. These requirements prevent aliases from occurring for unique objects.

The rules for object creation and method call must ensure that U-mode parameters consume uniqueness, and that L-mode parameters adhere to the lent-once policy. In addition, it must ensure that each unique object meets the field uniqueness invariant requirement before being passed into a method. The relation $\Theta, \Lambda \vdash \mathit{conUL}(v, t_S, t_T), \Theta_1, \Lambda_1$ (named [AUX2]) helps ensure the above. It checks that each variable v is neither lent twice, nor has its uniqueness consumed twice. Note that Λ captures unique variables which are temporarily lent out. In addition, fields of such parameters must not be in the dead-set.

13. Other Language Features

Our implementation also supports several other language features, including downcast, `while`-loop and a field-binding construct. Their alias checking and sized-type rules are shown in Figure 12.

For the downcast operation, we allow both the class type and its alias annotation to be changed. Run-time test can then be used to ascertain the validity of each downcast request.

We also provide direct support for `while`-loop construct. The reason for doing so is that tail-recursion is not able to handle certain loops due to the use of the pass-by-value parameter mechanism. Our solution is to analyse each `while` loop using its corresponding tail-recursive function but with two deviations, namely (i) no stack frames are created (ii) use of pass-by-reference parameter mechanism. We achieve this using a set of three rules, named [WHILE−METH], [WHILE−CALL] and [WHILE]. Take note that each loop of form (`while` e_1 `do` e_2) is equivalent to a `while`-call m**(**$v_{1..p}$**)** with the following definition.

$$\texttt{void}\langle\rangle\texttt{@S}\ m((\hat{t}_i\ v_i)_{i:1..p})\ \texttt{where}\ \phi_{pr}; \phi_{po}; \epsilon_c; \epsilon_r$$
$$\{\texttt{if}\ e_1\ \texttt{then}\ e_2; m(v_{1..p})\ \texttt{else}\ ()\}$$

To support precise size-tracking, we also provide a field-binding construct which binds the fields of an unaliased object to a set of local variables, as follows:

$$\texttt{bind}\ (v_{1..p}) = v\ \texttt{in}\ e$$

Such a construct allows the fields of an object v to be temporarily lent-out to a set of local variables, $\{v_1..v_p\}$, in the scope of the expression e. Any update to the local variables is considered as an update to the corresponding field of the object v itself. The `bind` construct is particularly important for objects whose size properties are built from more than one components, such as the `Tree` class below.

$$\texttt{class Tree}\langle n\rangle\ \texttt{where}\ n{=}1{+}a{+}b\ ;\ n{\geq}0\ \{$$
$$\texttt{int}\langle v\rangle\texttt{@S val};$$
$$\texttt{Tree}\langle a\rangle\texttt{@U left};$$
$$\texttt{Tree}\langle b\rangle\texttt{@U right};$$
$$\}$$

Due to a dependency on multiple components, each separate access to the left or right subtree of each node is unable to preserve any relation between a single subtree and its parent. With the `bind` construct, the uniqueness of object `v` is preserved. Moreover, we are able to preserve the size relation of the node with its two subtrees at both the entry and exit of the expression body. This preservation is required for precise size tracking of each node, and is especially relevant when the subtrees are being updated. As an example, consider the following method for inserting a node into a binary search tree.

$$\texttt{Tree}\langle r\rangle\texttt{@U insert(Tree}\langle n\rangle\texttt{@U t, int}\langle v\rangle\texttt{@S v)}$$
$$\texttt{where}\ d = n; r{=}n{+}1; \{(\texttt{Tree}, 1), (\mathcal{S}, d+5)\}, \{\}$$
$$\{\texttt{if isNull(t) then new Tree(v, null, null)}$$
$$\texttt{else bind (i, l, r) = t in}$$
$$\texttt{if v < i then l = insert(l, v)}$$
$$\texttt{else r = insert(r, v);}$$
$$\texttt{t}$$
$$\}$$

Take note that we are able to predict that the size of the output tree is one more than its input. If we had used the following code, without the `bind` construct, our type system can only verify a less precise specification.

$$\texttt{Tree}\langle r\rangle\texttt{@U insert(Tree}\langle n\rangle\texttt{@U t, int}\langle v\rangle\texttt{@S v)}$$
$$\texttt{where true; true;}\ \{(\texttt{Tree}, 1), (\mathcal{S}, d+5)\}, \{\}$$
$$\{\texttt{if isNull(t) then new Tree(v, null, null)}$$
$$\texttt{else}\ \{\texttt{if v < i then t.left = insert(t.left, v)}$$
$$\texttt{else t.right = insert(t.right, v);}$$
$$\texttt{t}\}$$
$$\}$$

We are currently investigating techniques for the automatic insertion of `bind` construct into MEMJ programs.

14. Proofs

The proofs for Theorem 1 and Theorem 2 require a lemma, called *assumption weakening lemma*, that states that the static judgment remains valid despite a variation of its assumption. This assumes the store type Σ to have unbounded mapping of locations to types. However, the type environment Γ takes the form of stackable mapping between variables and types, and is allowed to grow (by pushing in new mappings) and shrink (by popping out mappings from stack). The lemma states that such change to type environment preserves the type judgment, if the change are properly constrained.

LEMMA 1 (Assumption Weakening). *Given that the following judgments hold:*

$$\Gamma; \Sigma; \Theta \vdash \mathit{eraseS}(e) :: \mathit{eraseS}(t), \Theta_1$$
$$\Gamma; \Sigma; \Delta; \Upsilon \vdash e :: t, \Delta_1, \Upsilon_1$$

Let Γ_α *and* Σ_α *be such that:*

$$V_u(\Delta) \subseteq V(\Gamma_\alpha) \qquad V(t) \cap V(\Gamma_\alpha) = \emptyset$$
$$(vars(e) \cup \{v \mid v \in \Theta \vee \exists f \cdot v.f \in \Theta\}) \subseteq \mathit{dom}(\Gamma_\alpha)$$
$$\exists v^* \cdot (\Gamma - \{v^*\} = \Gamma_\alpha) \vee (\Gamma_\alpha - \{v^*\} = \Gamma)$$
$$\Sigma_\alpha \supseteq \Sigma$$

Then,

$$\Gamma_\alpha; \Sigma_\alpha; \Theta \vdash \mathit{eraseS}(e) :: \mathit{eraseS}(t), \Theta_1$$
$$\Gamma_\alpha; \Sigma_\alpha; \Delta; \Upsilon \vdash e :: t, \Delta_1, \Upsilon_1$$

Note that $vars(e)$ returns all variables occurring in e.

Proof: By structural induction: the first judgement is based on alias checking rules, while the second is based on type rules with size and memory analysis.

The proof of Theorem 2 also relies on the following lemma.

LEMMA 2 (Redundant Memory). *If*

$$\Gamma; \Sigma; \Delta; \Upsilon \vdash e :: t, \Delta_1, \Upsilon_1$$

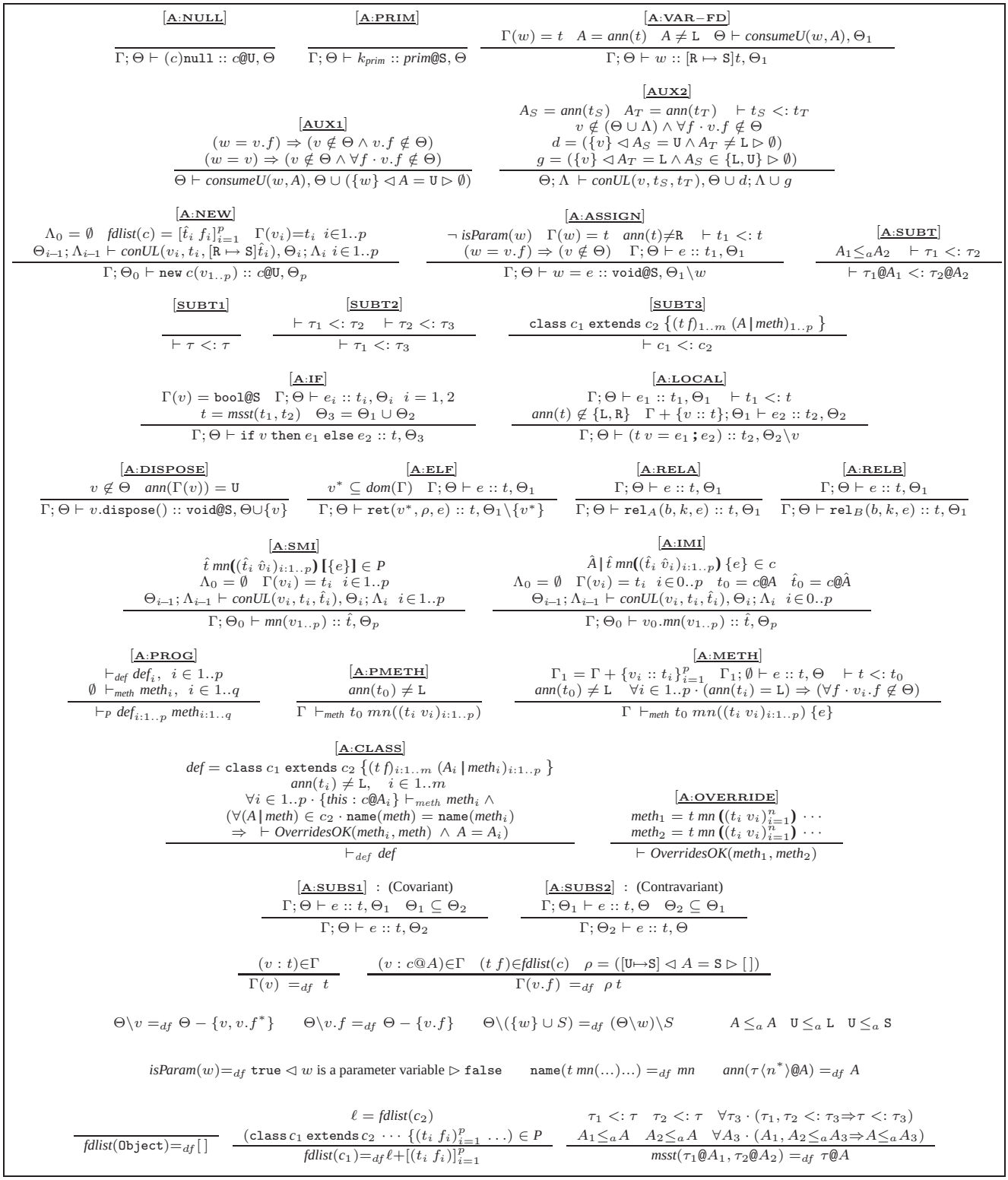

Figure 10. Type Rules for Alias Checking

$$\frac{[\textsc{cons1}]\quad n^* = \mathit{fresh}()}{\Gamma;\Delta;\Upsilon \vdash (c)\texttt{null} :: c\langle n^*\rangle@\texttt{U}, \Delta \wedge (n=0)^*, \Upsilon} \qquad \frac{[\textsc{cons2}]\quad n = \mathit{fresh}()}{\Gamma;\Delta;\Upsilon \vdash k^{\mathit{int}} :: \texttt{int}\langle n\rangle@\texttt{S}, \Delta \wedge (n=k), \Upsilon}$$

$$\frac{[\textsc{cons3}]\quad k{=}\texttt{()} \mid k^{\mathit{float}} \quad \tau{=}(\texttt{void} \lhd k{=}\texttt{()} \rhd \texttt{float})}{\Gamma;\Delta;\Upsilon \vdash k :: \tau\langle\rangle@\texttt{S}, \Delta, \Upsilon} \qquad \frac{[\textsc{cons4}]\quad b = \mathit{fresh}() \quad \phi{=}(b{=}1 \lhd k^{\mathit{bool}}{=}\texttt{true} \rhd b{=}0)}{\Gamma;\Delta;\Upsilon \vdash k^{\mathit{bool}} :: \texttt{bool}\langle b\rangle@\texttt{S}, \Delta \wedge \phi, \Upsilon} \qquad \frac{[\textsc{var-fd}]\quad \Gamma \vdash w :: t, \phi, Y}{\Gamma;\Delta;\Upsilon \vdash w :: [\texttt{R} \mapsto \texttt{S}]t, \Delta \wedge \phi, \Upsilon}$$

$$\frac{[\textsc{aux1b}]\quad \vdash (t\ f) \in c\langle n^*\rangle,\ \varphi \quad m^* = V(t)}{\frac{[\textsc{aux1a}]\quad \Gamma(v) = t \quad t_1 = \mathit{fresh}(t) \quad \phi = \mathit{equate}(t_1, \mathit{prime}(t))}{\Gamma \vdash v :: t_1, \phi, V(t)} \qquad \frac{\Gamma(v) = c\langle n^*\rangle@A \quad \rho_1 = ([\texttt{U} \mapsto \texttt{S}] \lhd A = \texttt{S} \rhd [\,]) \quad a^* = V(\varphi) - (m^* \cup n^*) \quad \rho = [(n \mapsto n')^*] \cup \mathit{rename}(t, t_1) \quad t_1 = \mathit{fresh}(t) \quad Y = \rho(\mathit{depends}(n^*, m^*, \varphi)) \quad \phi = \rho(\exists a^* \cdot \varphi)}{\Gamma \vdash v.f :: (\rho_1\ t_1), \phi \wedge \mathit{inv}(t_1), V_u(Y)} \qquad \frac{[\textsc{aux2a}]\quad (\ell, \phi) = \mathit{fdList}(c\langle u_{1..p}\rangle) \quad (t\ f) \in \ell}{\vdash (t\ f) \in c\langle u_{1..p}\rangle, \phi}}$$

$$\frac{[\textsc{aux2b}]\quad \texttt{class}\, c\langle n_{1..p}\rangle\, \texttt{extends}\, ..\, \{\, ..meth..\} \in P \quad meth_1 = \mathit{fresh}(meth) \quad \rho = [n_i{\mapsto}u_i]_{i=1}^p}{\vdash \rho(meth_1) \in c\langle u_{1..p}\rangle} \qquad \frac{[\textsc{aux2c}]\quad \texttt{class}\, c_1\langle n_{1..p}\rangle\, \texttt{extends}\, c_2\langle n_{1..q}\rangle\ \cdots\ \{(t\,f)^*\ (A_i \mid meth_i)_{i=1}^p\} \in P \quad \vdash meth \in c_2\langle u_{1..q}\rangle \quad \forall i \in 1..p \cdot \texttt{name}(meth) \neq \texttt{name}(meth_i)}{\vdash meth \in c_1\langle u_{1..p}\rangle} \qquad \frac{[\textsc{aux2d}]\quad P = \cdots meth \cdots \quad meth_1 = \mathit{fresh}(meth)}{\vdash meth_1 \in P}$$

$$\frac{[\textsc{smi}]\quad \vdash (\hat{t}\ mn\texttt{(}(\hat{t}_i\ \hat{v}_i)_{i:1..p}\texttt{)}\ \texttt{where}\ \phi_{pr}; \phi_{po}; \epsilon_c; \epsilon_r\ \{e\}\ ) \in P \quad t = \mathit{fresh}(\hat{t}) \quad \Gamma(v_i) = t_i \quad i \in 1..p \quad \vdash t_i <: \hat{t}_i,\ \rho_i \quad i \in 1..p \quad \rho_p = \bigcup_{i=1}^p \rho_i \quad \Delta_1 \vdash \Upsilon \sqsupseteq \epsilon_c \quad \rho = \mathit{rename}(\hat{t}, t) \cup \rho_p \cup \mathit{prime}(\rho_p) \quad \Delta \gg_{V(\Gamma)} \exists V(\epsilon_c) \cup V(\epsilon_r) \cdot \rho\ \phi_{pr} \quad \Delta_1 = \Delta \circ_L \exists Y \cdot \rho(\phi_{pr} \wedge \phi_{po}) \quad \Upsilon_1 = \Upsilon - (\epsilon_c \backslash \{\mathcal{S}\}) \uplus \epsilon_r \quad X = \bigcup_{i=1}^p V(\hat{t}_i) \quad Y = X \cup \mathit{prime}(X) \quad L = \bigcup_{i=1}^p V(t_i)}{\Gamma;\Delta;\Upsilon \vdash mn(v_{1..p}) :: t, \Delta_1, \Upsilon_1}$$

$$\frac{[\textsc{local}]\quad \Gamma;\Delta;\Upsilon \vdash e_1 :: t_1, \Delta_1, \Upsilon_1 \quad t = \mathit{fresh}(\hat{t}) \quad \vdash t_1 <: \mathit{prime}(t), \rho \quad X = V(t_1) \quad Y = V(t) \cup \mathit{prime}(V(t)) \quad \Delta_1 \vdash \Upsilon_1 \sqsupseteq \{(\mathcal{S}, 1)\}}{\Gamma,\ v :: \hat{t}; \exists X \cdot \rho^{-1}\Delta_1; \Upsilon_1 - \{(\mathcal{S}, 1)\} \vdash\ e_2 :: t_2, \Delta_2, \Upsilon_2} \quad \Gamma;\Delta;\Upsilon \vdash (\hat{t}\ v = e_1\ \texttt{;}\ e_2) :: t_2, \exists Y \cdot \Delta_2, \Upsilon_2$$

$$\frac{[\textsc{class}]\quad S_i = V(t_i)\ i \in 1..m \quad \mathit{distinct}\{S_1, \ldots, S_m, \{n_1..n_p\}\} \quad V(\phi_I) \subseteq \{n_i\}_{i=q+1}^p \quad p \geq q \quad \phi = \bigwedge_{i=q+1}^p (n_i{=}\alpha_i) \quad \forall i \in \{q{+}1..p\} \cdot V(\alpha_i) \subseteq \bigcup_{i=1}^m S_i \quad \Gamma_i = \{\mathit{this} :: c_1\langle n_{1..p}\rangle@A_i\} \quad \Gamma_i \vdash_{meth} meth_i \quad i \in 1..r \quad \mathit{inv}([t_i]_{i=1}^m) \wedge \phi \Rightarrow \phi_I}{\vdash_{class}\ \texttt{class}\, c_1\langle n_{1..p}\rangle\, \texttt{extends}\, c_2\langle n_{1..q}\rangle\, \texttt{where}\, \phi\ ;\ \phi_I\{(t_i\ f_i)_{i=1}^m, (A_i \mid meth_i)_{i=1}^r\}} \qquad \frac{[\textsc{prog}]\quad P = def_{i=1}^m\ meth_{i=1}^n \quad \mathit{NoCircClasses}(P) \quad \mathit{FieldsOnce}(P) \quad \mathit{InstanceMethOnce}(P) \quad \vdash \mathit{InheritanceOK}(def_i)\ i \in 1..m \quad \vdash_{class}\ def_i\ i \in 1..m \quad \{\} \vdash_{meth} meth_i\ i \in 1..n}{\vdash P}$$

$$\frac{[\textsc{inhc}]\quad def_1 = \texttt{class}\, c_1\langle n_{1..p}\rangle\, \texttt{extends}\, c_2\langle n_{1..q}\rangle\, \texttt{where} \cdots \{fd^*\ meth_{1..p}\} \quad (\exists meth \cdot \vdash meth \in c_2\langle n_{1..q}\rangle \wedge \texttt{name}(meth) = \texttt{name}(meth_i)) \Rightarrow \vdash \mathit{OverridesOK}(meth_i, meth) \quad i \in 1..p}{\vdash \mathit{InheritanceOK}(def_1)} \qquad \frac{[\textsc{override}]\quad meth_1 = t\ mn((t_i\ v_i)_{i:1..p})\ \texttt{where}\ \phi_{pr_1}; \phi_{po_1}; \epsilon_{1m}; \epsilon_{1n}\ \{\cdots\} \quad meth_2 = t\ mn((t_i\ v_i)_{i:1..p})\ \texttt{where}\ \phi_{pr_2}; \phi_{po_2}; \epsilon_{2m}; \epsilon_{2n}\ \{\cdots\} \quad \phi_{pr_1} \Rightarrow \phi_{pr_2} \quad \phi_{po_2} \Rightarrow \phi_{po_1} \quad \phi_{pr_1} \vdash \epsilon_{1m} \sqsupseteq \epsilon_{2m} \quad \phi_{pr_1} \vdash \epsilon_{2n} \sqsupseteq \epsilon_{1n}}{\vdash \mathit{OverridesOK}(meth_1, meth_2)}$$

$$\frac{[\textsc{subt1}]\quad \mathbf{V}_{\mathit{class}}(\tau\langle s_1, \ldots, s_m\rangle) = (S_I, S_T, \_) \quad \rho_I = [n_i{\mapsto}s_i]_{s_i \in S_I} \quad \rho_T = [n_i{\mapsto}s_i]_{s_i \in S_T} \quad A_1 \leq_a A_2 \quad \rho{=}(\rho_I \lhd A_1{=}\texttt{S} \vee A_2{=}\texttt{S} \rhd \rho_I \uplus \rho_T)}{\vdash\ \tau\langle s_1, .., s_m\rangle@A_1 <: \tau\langle n_1, .., n_m\rangle@A_2, \rho} \qquad \frac{[\textsc{subt2}]\quad \texttt{class}\, c_1\langle n_{1..p}\rangle\, \texttt{extends}\, c_2\langle n_{1..q}\rangle\, \cdots \in P \quad \vdash c_2\langle n_{1..q}\rangle@A_1 <: c_3\langle m_{1..r}\rangle@A_2,\ \rho}{\vdash c_1\langle n_{1..p}\rangle@A_1 <: c_3\langle m_{1..r}\rangle@A_2,\ \rho} \qquad \frac{[\textsc{def-name}]\quad meth = \hat{t}\ mn\texttt{(..)}\ \cdots}{\texttt{name}(meth) =_{df} mn}$$

$$\frac{[\textsc{def-inv1}]}{\mathit{inv}(\texttt{bool}\langle b\rangle) =_{df} 0 \leq b \leq 1} \qquad \frac{[\textsc{def-inv2}]\quad \tau = \texttt{int} \mid \texttt{float} \mid \texttt{void} \mid \texttt{Object}}{\mathit{inv}(\tau\langle m^*\rangle) =_{df} \texttt{true}} \qquad \frac{[\textsc{def-inv3}]\quad \texttt{class}\, c_1\langle n_{1..q}\rangle\, \texttt{extends}\, c_2\langle n_{1..r}\rangle\, \texttt{where}\, \phi_2\ ;\ \phi_I\{(t_i\ f_i)_{i=1}^p\ \cdots\ \} \in P \quad \phi_1 = \mathit{inv}(c_2\langle \hat{n}_{1..r}\rangle) \quad \rho = [n_i{\mapsto}\hat{n}_i]_{i=1}^q}{\mathit{inv}(c_1\langle \hat{n}_{1..q}\rangle) =_{df} \phi_1 \wedge (\rho\ \phi_I))}$$

$$\frac{[\textsc{def-fdList1}]}{\mathit{fdList}(\texttt{Object}\langle\rangle) =_{df} ([\,], \texttt{true})} \qquad \frac{[\textsc{def-fdList2}]\quad (\ell_1, \phi_1) = \mathit{fdList}(c_2\langle \hat{n}_{1..r}\rangle) \quad \texttt{class}\, c_1\langle n_{1..q}\rangle\, \texttt{extends}\, c_2\langle n_{1..r}\rangle\, \texttt{where}\, \phi_2\ ;\ \phi_I\{(t_i\ f_i)_{i=1}^p\ \cdots\ \} \in P \quad \hat{t}_i = \mathit{fresh}(t_i),\ i \in 1..p \quad \rho = [n_i{\mapsto}\hat{n}_i]_{i=1}^q \cup \bigcup_{i=1}^p \mathit{rename}(t_i, \hat{t}_i)}{\mathit{fdList}(c_1\langle \hat{n}_{1..q}\rangle) =_{df} (\ell_1 + [(\hat{t}_i\ f_i)]_{i=1}^p,\ \phi_1 \wedge (\rho\ \phi_2))} \qquad \frac{[\textsc{def-}\mathbf{V}_{\mathit{field}}]\quad \mathbf{V}_{\mathit{class}}(\tau\langle s^*\rangle) = (d_I, d_T, d_N)}{\mathbf{V}_{\mathit{field}}(\tau\langle s^*\rangle@\texttt{R}) =_{df} (d_I, \emptyset, d_T \cup d_N)}$$

$$\frac{[\textsc{def-}\mathbf{V}_{\mathit{field}}]\quad \mathbf{V}_{\mathit{class}}(\tau\langle s^*\rangle) = (d_I, d_T, d_N)}{\mathbf{V}_{\mathit{field}}(\tau\langle s^*\rangle@\texttt{S}) =_{df} (\emptyset, d_I, d_T \cup d_N)} \qquad \frac{[\textsc{def-}\mathbf{V}_{\mathit{field}}]\quad A \in \{\texttt{U}, \texttt{L}\} \quad \mathbf{V}_{\mathit{class}}(\tau\langle s^*\rangle) = (d_I, d_T, d_N)}{\mathbf{V}_{\mathit{field}}(\tau\langle s^*\rangle@A) =_{df} (\emptyset, d_I \cup d_T, d_N)} \qquad \frac{[\textsc{def-}\mathbf{V}_{\mathit{class}}]\quad \tau \in \mathit{prim}}{\mathbf{V}_{\mathit{class}}(\tau\langle n^*\rangle) =_{df} (n^*, \emptyset, \emptyset)}$$

$$\frac{[\textsc{def-}\mathbf{V}_{\mathit{class}}]\quad \mathbf{V}_{\mathit{field}}(t_i) = (d_i^I, d_i^T, d_i^N), \quad i \in \{1..p\} \quad ([t_i\ f_i]_{i=1}^p, \phi) = \mathit{fdList}(c_1\langle n_{1..q}\rangle) \quad n_I = \mathit{depends}(n_{1..q}, \bigcup_{i=1}^p d_i^I, \phi) \quad n_T = \mathit{depends}(n_{1..q}, \bigcup_{i=1}^p d_i^T, \phi) \quad n_N = \mathit{depends}(n_{1..q}, \bigcup_{i=1}^p d_i^N, \phi)}{\mathbf{V}_{\mathit{class}}(c_1\langle n_{1..q}\rangle) =_{df} (n_I - (n_T \cup n_N), n_T - n_N, n_N)} \qquad \frac{[\textsc{def-fresh-meth}]\quad meth = \hat{t}\ mn\texttt{(..)}\ \texttt{where}\ \phi_{pr}; \phi_{po}; \epsilon_c; \epsilon_r\{e\} \quad \{s_1, .., s_p\} = V(\epsilon_c) \cup V(\epsilon_r) \quad u_{1..p} = \mathit{fresh}() \quad \rho = [s_i{\mapsto}u_i]_{i=1}^p \quad \hat{\epsilon}_c = \rho\ \epsilon_c \quad \hat{\epsilon}_r = \rho\ \epsilon_r \quad \hat{\phi}_{pr} = \rho\ \phi_{pr} \quad \hat{\phi}_{po} = \rho\ \phi_{po}}{\mathit{fresh}(meth) =_{df} \hat{t}\ mn\texttt{(..)}\ \texttt{where}\ \hat{\phi}_{pr}; \hat{\phi}_{po}; \hat{\epsilon}_c; \hat{\epsilon}_r\{e\}}$$

$$\mathit{depends}(n^*, s^*, \phi) =_{df} \bigcup \{n \mid (n = \alpha) \in \phi,\ (V(\alpha) \cap s^*) \neq \emptyset\}$$

Figure 11. Other Type Rules for Memory Checking

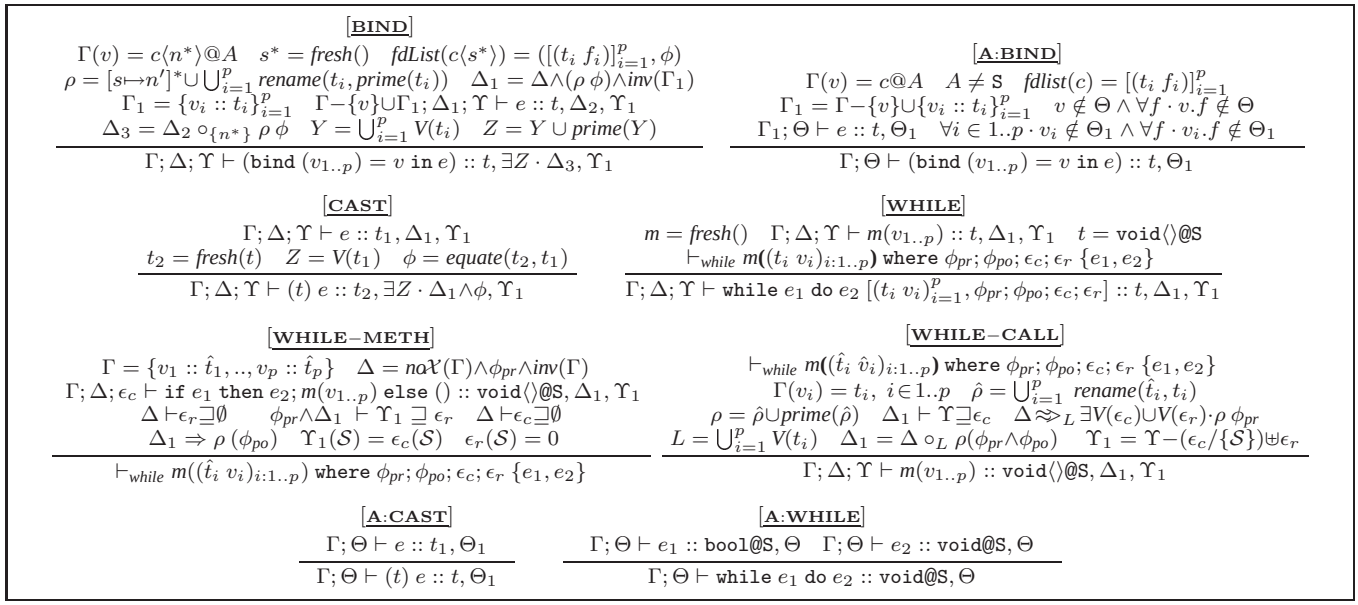

Figure 12. Type Rules for Other Language Features

and $\Delta \vdash \epsilon \sqsupseteq \emptyset$*, then*

$$\Gamma; \Sigma; \Delta; \Upsilon \uplus \epsilon \vdash e :: t, \Delta_1, \Upsilon_1 \uplus \epsilon$$

Proof: By structural induction on e. □

14.1 Proof of Theorem 1

By induction over the depth of the type derivation of alias checking for expression *eraseS*(e).

Case [A:VAR–FD]. Let $\Gamma_\alpha = \Gamma$, $\Sigma_\alpha = \Sigma$, and $\Theta_\alpha = \Theta_1$. Then the type preservation is straightforward. For alias consistency, we specifically need to show the consistency of aliases for those references not in Θ_α. These are consistency because (1) w, if *not* included in Θ_α, does not change its alias annotation during the reduction step (as shown in the function *read* in dynamic semantics); (2) for other variables, their alias annotation remains intact during this reduction; (3) the no-dangling property for Π_1,ϖ_1 is preserved as no live locations are deleted from the store, while the result value (e_1) is live.

Case [A:ASSIGN]. We deal with expression $w = e$. There are two rules by which one step evaluation can be conducted:

Subcase [D-Assign-1] By induction hypothesis, there exist Γ_α,Σ_α, and Θ_α that preserve the type of e_1 and $\Gamma_\alpha; \Sigma_\alpha; \Theta_\alpha \models_{\mathcal{A}} \langle \Pi_1, \varpi_1 \rangle$. By applying the rule [A:ASSIGN], we obtain the type preservation for $w = e_1$.

Subcase [D-Assign-2] Here, e is some value $\nu = (A_s, \delta_s)$. From dynamic semantics, we obtain $\langle \Pi_1, \varpi_1 \rangle = \mathit{upd}(\Pi, \varpi, w, \nu)$. Let $\Gamma_\alpha = \Gamma$, $\Sigma_\alpha = \Sigma$, and $\Theta_\alpha = \Theta/w$. The type preservation is obvious. To show the alias consistency for those references not in Θ/w, we just need to check the alias annotation associated with w, which may have just been removed from Θ. If $w \notin \Theta$, then the conformance is guaranteed from the premise. If $w \in \Theta$, then there are two cases: (1) if (the value associated with) w has an annotation $\mathtt{U}_D$ before reduction, then it has annotation $\mathtt{U}$ after reduction, which is less than any static-annotation of w at compile-time; (2) if w has an annotation other than $\mathtt{U}_D$ before reduction, then its annotation remains the same after reduction. Thus, the alias consistency of w is guaranteed from the premise. The no-dangling property is preserved as no live locations in Π_1 or ϖ_1 are removed from the store. While the target expression is a primitive.

Case [A:NEW]. We deal with expression $\mathtt{new}\ c(v_{1..p})$. We choose $\Gamma_\alpha = \Gamma$, $\Sigma_\alpha = \Sigma \cup \{\iota \mapsto c@\mathtt{U}\}$, and $\Theta_\alpha = \Theta_p$. The proof follows.

Case [A:IMI]. Given that $\Gamma; \Sigma; \Theta_0 \vdash v_0.mn(v_{1..p}) :: \hat{t}, \Theta_p$, and $\langle \Pi, \varpi, \sigma \rangle\ [v_0.mn(v_{1..p})] \hookrightarrow \langle \Pi_1, \varpi, \sigma_1 \rangle\ [\mathtt{ret}(\hat{v}_{0..p}, \rho, e)]$. We need to show there exist $\Gamma_\alpha, \Sigma_\alpha, \Theta_\alpha$ such that $\Gamma_\alpha; \Sigma_\alpha; \Theta_\alpha \models_{\mathcal{A}} \langle \Pi_1, \varpi \rangle$, and $\Gamma_\alpha; \Sigma_\alpha; \Theta_\alpha \vdash \mathtt{ret}(\hat{v}_{0..p}, \rho, e) :: \hat{t}, \Theta_p$. Let Σ_α=Σ, Θ_α=Θ_p, and $\Gamma_\alpha = (\Gamma, \hat{v}_0^q :: \hat{t}_0, \ldots, \hat{v}_p^q :: \hat{t}_p))$, where q is the new stack frame number. The alias consistency is ensured by the definition of *updVE* and the respective premise. Since no variables/fields in Θ_0 will be used by the method body e, checking the method body only involves formal parameters $(v^q)^*$. Thus, we obtain $\Gamma_\alpha; \Sigma_\alpha; \Theta_\alpha \vdash e :: t, \Theta_\alpha \cup \Theta_M$, where Θ_M is derived from the rule [A:METH], and $\Theta_M \subseteq \{v_{0..p}^q\}$. From the rule [A:ELF], we obtain $\Gamma_\alpha; \Sigma_\alpha; \Theta_\alpha \vdash \mathtt{ret}(\hat{v}_{0..p}, \rho, e) :: \hat{t}, \Theta_\alpha$.

Case [A:SMI]. We deal with expression $mn(v_1, .., v_p)$. There are two subcases: [**D-MI**] and [**D-MI-prim**]. The proof is similar to the case [A:IMI].

Case [A:LOCAL]. We deal with expression $t\ v = e_1; e_2$. There are two rules by which one step evaluation can be conducted:

Subcase [D-Blk-1] By induction hypothesis and the assumption weakening lemma.

Subcase [D-Blk-2] By induction hypothesis and the rule [A:ELF].

Case [A:IF]. Both the subcases [**D-If-true**] and [**D-If-false**] can be easily proven by the induction hypothesis and the covariance subsumption rule [A:SUBS1].

Case [A:DISPOSE]. We deal with expression $v.\mathtt{dispose}()$. Let $\Gamma_\alpha = \Gamma$, $\Sigma_\alpha = \Sigma$, and $\Theta_\alpha = \Theta \cup \{v\}$. The type preservation is trivial. After the reduction step ([**D-Dispose**]), $\varpi_1 = \varpi - \delta$, while $\Pi_1 = \Pi[(\mathtt{U}_D, \delta)/v]$. Although δ is deleted from the store, v is moved to Θ_α correspondingly, while other references keep un-

changed. Thus the no-dangling property and other alias consistency properties are obviously preserved.

Case [A:RELA]. The only possible reduction step is [**D-RelA**]. The proof follows by induction hypothesis.

Case [A:ELF].We deal with expression $\texttt{ret}(v^*, \rho, e)$. There are five rules by which one step evaluation can be conducted:

Subcase [D-ret-1] By induction hypothesis.

Subcase [D-ret-2] Let $\Gamma_\alpha = \Gamma$, $\Sigma_\alpha = \Sigma$, and $\Theta_\alpha = \Theta$. The proof follows immediately.

Subcase [D-ret-3] By rule [A:RELA] and [A:ELF].

Subcase [D-TIM] The proof is similar to that for the case [A:IMI].

Subcase [D-TSM] The proof is similar to that for the case [A:SMI].

□

14.2 Proof of Theorem 2

(a) By induction over the depth of the type derivation of size and memory analysis for expression e. In each of the following cases, the Γ_α, Σ_α, and Θ_α are taken the same as the respective cases in the proof of Theorem 1, and the alias consistency is already proved there. We shall focus only on the construction of Δ_α and Υ_α to ensure the size consistency the type preservation.

Case [VAR–FD]. Given that $\Gamma; \Delta; \Upsilon \vdash w :: [\mathtt{R} \mapsto \mathtt{S}]t, \Delta \wedge \phi, \Upsilon$, and $\langle \Pi, \varpi, \sigma\rangle\ [w] \hookrightarrow \langle \Pi_1, \varpi_1, \sigma\rangle\ [\nu]$. Let $\Delta_\alpha = \Delta \wedge \phi$, $\Upsilon_\alpha = \Upsilon$. Then the type judgement $\Gamma_\alpha; \Sigma_\alpha; \Delta_\alpha; \Upsilon_\alpha \vdash \nu :: [\mathtt{R} \mapsto \mathtt{S}]t, \Delta_\alpha, \Upsilon_\alpha$ follows from the covariant subsumption for sizes [SUBS1]. For size and memory consistency, as there is no change in run-time environment during reduction, we have $\Delta_{\Pi_1} = \Delta_\Pi$. Furthermore, $\Delta_{\Pi_1} \Rightarrow \exists X \cdot \Delta$ where $X = (V(\Delta) - \mathit{prime}(V(\Gamma))) \cup D_\Theta$. Let $\hat{X} = (V(\Delta_\alpha) - \mathit{prime}(V(\Gamma))) \cup D_{\Theta_\alpha}$. When w is a variable v, we have $\exists X \cdot \Delta = \exists \hat{X} \cdot \Delta_\alpha$ when $w \notin \Theta_\alpha$, and $\exists X \cdot \Delta \Rightarrow \exists \hat{X} \cdot \Delta_\alpha$ otherwise. So, $\Delta_{\Pi_1} \Rightarrow \exists \hat{X} \cdot \Delta_\alpha$. When w is a field-read $v.f$, we have $\exists X \cdot \Delta = \exists \hat{X} \cdot \Delta_\alpha$, and thus the proof.

Case [ASSIGN]. We deal with expression $w = e$. From the static semantics, we have $\Gamma; \Delta; \Upsilon \vdash e :: t_1, \Delta_1, \Upsilon_1$. There are two rules by which one step evaluation can be conducted:

Subcase [D-Assign-1] By induction hypothesis, there exist Γ_α,Σ_α, Θ_α, Δ_α, and Υ_α, such that $\Gamma_\alpha; \Sigma_\alpha; \Delta_\alpha; \Theta_\alpha; \Upsilon_\alpha \models \langle \Pi_1, \varpi_1, \sigma_1\rangle$. The relationship between Γ_α and Γ for them to be consistent ensures that $\Gamma_\alpha \vdash w :: t, \phi, Y$, as shown in the premise of the rule [ASSIGN]. By applying rule [ASSIGN] on $\Gamma_\alpha, \Sigma_\alpha, \Delta_\alpha$ and Υ_α, we have $\Gamma_\alpha; \Sigma_\alpha; \Delta_\alpha; \Upsilon_\alpha \vdash w = e_1 :: \textbf{void}\langle\rangle@\mathtt{S}, \Delta_2, \Upsilon_1$.

Subcase [D-Assign-2] Here, e is some value $\nu = (A_s, \delta_s)$. From dynamic semantics, we obtain $\langle \Pi_1, \varpi_1\rangle = \mathit{upd}(\Pi, \varpi, w, \nu)$. Let $\Upsilon_\alpha = \Upsilon$ and $\Delta_\alpha = \Delta_2 = \exists Z \cdot \Delta_1 \circ_Y \rho(\phi)$, as given in the static rule, with $Z = V(t_1) \cup V(t)$. Type preservation and consistency relation is obvious.

Case [NEW]. Given $\Gamma; \Delta; \Upsilon \vdash$ **new** $c(v_{1..p}) :: c\langle r^*\rangle@\mathtt{U}, \Delta_1, \Upsilon_1$. We choose $\Delta_\alpha = \Delta_1$, $\Upsilon_\alpha = \Upsilon_1$, and the proof follows.

Case [IMI]. Given that $\Gamma; \Sigma; \Delta; \Upsilon \vdash v_0.\mathit{mn}(v_{1..p}) :: t, \Delta_2, \Upsilon_2$, where $\Delta_2 = \Delta \circ_L \exists Y \cdot \rho(\phi_{pr} \wedge \phi_{po})$. The one-step reduction is $\langle \Pi, \varpi, \sigma\rangle\ [v_0.\mathit{mn}(v_{1..p})] \hookrightarrow \langle \Pi_1, \varpi, \sigma_1\rangle\ [\texttt{ret}(\hat{v}_{0..p}, \rho, e)]$. We need to show size consistency and type preservation. Let $\Delta_\alpha = \Delta \circ_L \Delta_e$ where $\Delta_e = n\alpha\mathcal{X}(\cup_{i=0}^{p}\{V_u(\hat{t}_i)\})$, $\hat{t}_0 = c\langle n^*\rangle@A$, and the method $(A\,|\,\hat{t}\ \mathit{mn}((\hat{t}_i\ \hat{v}_i)_{i:1..p})\ \texttt{where}\ \phi_{pr}; \phi_{po}; \epsilon_c; \epsilon_r\{e\}) \in c\langle n^*\rangle$. Let $\Upsilon_\alpha = \Upsilon - \{(\mathcal{S}, p{+}3)\}$. The size consistency is ensured by the definition of *updVE* and the respective premise. Since no variables/fields in Θ will be used by the method body e, checking the method body only involves size variables of formal parameters $(\hat{v}^q)^*$. Thus, $\Gamma_\alpha; \Sigma_\alpha; \Delta_\alpha; \epsilon \vdash e :: t, \Delta \circ_L \Delta_M, \Upsilon_1$, where Δ_M and Υ_1 are derived from rule [METH]. Note that $\phi_{pr} \wedge \Delta_1 \vdash \Upsilon_1 \sqsupseteq \epsilon_r \uplus \{(\mathcal{S}, \Upsilon_1(\mathcal{S}))\}$ from [METH]. Thus we have

$$\begin{aligned}\phi_{pr} \wedge \Delta_1 \vdash\ & \Upsilon_1 \uplus (\Upsilon_\alpha - \epsilon)\\ =\ & \Upsilon_1 \uplus \Upsilon - \epsilon_c\\ \sqsupseteq\ & \epsilon_r \uplus \{(\mathcal{S}, \Upsilon_1(\mathcal{S}))\} \uplus \Upsilon - \epsilon_c\\ =\ & \epsilon_r \uplus \{(\mathcal{S}, \epsilon_c(\mathcal{S}))\} \uplus \Upsilon - \epsilon_c\\ =\ & \Upsilon - (\epsilon_c / \{\mathcal{S}\}) \uplus \epsilon_r\\ =\ & \Upsilon_2\end{aligned}$$

By Lemma 2 and rule [SUBS1], we have $\Gamma_\alpha; \Sigma_\alpha; \Delta_\alpha; \Upsilon_\alpha \vdash e :: t, \Delta \circ_L \Delta_M, \Upsilon_2$. By the rule [ELF], we have $\Gamma_\alpha; \Sigma_\alpha; \Delta_\alpha; \Upsilon_\alpha \vdash \texttt{ret}(\hat{v}_{1..p}, \rho_m, e) :: t, \Delta_3, \Upsilon_2$, where $\Delta_3 = \exists Y_m \cdot \rho(\Delta \circ_L \Delta_M)$. While $\exists Y_m \cdot \rho(\Delta \circ_L \Delta_M) = \Delta \circ_L \exists Y_m \cdot \rho(\Delta_M) \Rightarrow \Delta \circ_L \exists Y_m \cdot \rho(\phi_{pr} \wedge \phi_{po}) = \Delta_2$.

Case [SMI]. We deal with expression $\mathit{mn}(v_1, .., v_p)$. There are two subcases: [**D-MI**] and [**D-MI-prim**]. The proof is similar to the case [IMI].

Case [LOCAL]. We deal with expression $t\ v = e_1; e_2$. There are two rules by which one step evaluation can be conducted:

Subcase [D-Blk-1] By induction hypothesis and the assumption weakening lemma.

Subcase [D-Blk-2] By induction hypothesis and the rule [ELF].

Case [IF].Both the subcases [**D-If-true**] and [**D-If-false**] can be easily proven by the induction hypothesis and the covariance subsumption rule [SUBS1].

Case [DISPOSE]. We deal with expression $v.\texttt{dispose}()$. Let $\Delta_\alpha = \Delta$, $\Upsilon_\alpha = \Upsilon \uplus \{(c, 1)\}$. The type preservation and the size & memory consistency are straightforward, from the respective premises.

Case [RELA]. The only possible reduction step is [**D-RelA**]. The proof follows by induction hypothesis.

Case [ELF].We deal with expression $\texttt{ret}(v^*, \rho, e)$. There are five rules by which one step evaluation can be conducted:

Subcase [D-ret-1] By induction hypothesis.

Subcase [D-ret-2] Let $\Delta_\alpha = \Delta$, $\Upsilon_\alpha = \Upsilon$. The proof follows immediately.

Subcase [D-ret-3] By rule [RELA] and [ELF].

Subcase [D-TIM] The proof is similar to that for the case [IMI], except that $\Upsilon_\alpha = \Upsilon \uplus \{(\mathcal{S}, b{+}k{-}(p{+}3))\}$.

Subcase [D-TSM] The proof is similar to that for the case [SMI], except that $\Upsilon_\alpha = \Upsilon \uplus \{(\mathcal{S}, b{+}k{-}(p{+}2))\}$.

(b) This can be proved by establishing that each well-typed constant from [CONS1] to [CONS4] does not change Θ. In addition, the resulted post-condition Δ obtained is consistent with the run-time stack and store. □

14.3 Proof of Theorem 3

By induction over the depth of type derivation for expression e.

Case [CONS1–4]. Trivial.

Case [VAR–FD]. We deal with the expression w. As $w = v \mid v.f$ is well-typed, from type rule [VAR–FD], neither v nor w is in $\mathtt{U}_D$ or L mode. Thus from the evaluation rule [**D-Var-FD**] (the function *read*), the evaluation either reports an **Error-Null**, or advances one step yielding a value.

Case [ASSIGN]. We deal with expression $w = e$, where $w = v \mid v.f$. From the type rule, we know $\Gamma; \Delta; \Upsilon \vdash e :: t_1, \Delta_1, \Upsilon_1$. By induction hypothesis, either (i) e is a value ν, or (ii) $\langle \Pi, \varpi, \sigma\rangle\ [e] \hookrightarrow$ **Error-Null**, or (iii) $\langle \Pi, \varpi, \sigma\rangle\ [e] \hookrightarrow \langle \Pi_1, \varpi_1, \sigma_1\rangle\ [e_1]$.
In case (i), from the well-typedness checking, neither w nor v is in L or R mode, and the RHS and LHS subsumes the alias subtyping, from the evaluation rule [**D-Assign-2**] (the function *upd*), either the evaluation reports an **Error-Null**, or the one-step evaluation succeeds, yielding a value $(\mathtt{S}, \textbf{()})$.

In case (ii), the evaluation for the assignment reports an **Error-Null**.

In case (iii), from evaluation rule [**D-Assign-1**], the evaluation advances with one-step.

Case [NEW]. We deal with expression **new** $c(v_{1..p})$. From the type rule [NEW], no arguments v_i are in the consumed set, and the type of each argument and that of its corresponding field conform to the (alias) subtyping relation. From the memory consistency and type rule [NEW], $\sigma \sqsupseteq \Upsilon \sqsupseteq \{(c, 1)\}$, which guarantees the memory adequacy. Thus from the evaluation rule [**D-New**], the evaluation succeeds, yielding a newly created object in the store.

Case [DISPOSE]. We deal with expression $v.\texttt{dispose}()$. Let $\Pi(v) = (A, \delta)$. If $\delta = \mathit{null}$, the evaluation reports an **Error-Null**. Otherwise, as the well-typedness ensures (A=U), the evaluation [**D-Dispose**] succeeds.

Case [REL−A]. We deal with expression $\texttt{rel}_A(b, k, e)$. Note that e cannot be a value, otherwise, the whole expression should be inside a $\texttt{ret}(..)$. If the evaluation of e can make one-step reduction, then so is $\texttt{rel}_A(b, k, e)$ by dynamic rule [**D-RelA**]. If the evaluation of e reports an **Error-Null**, so is $\texttt{rel}_A(b, k, e)$.

Case [IMI]. We deal with expression $v_0.mn(v_{1..p})$. As guaranteed by type rule [IMI], no unique arguments are consumed before the method call, and the type of each argument and that of its corresponding field conform to the (alias) subtyping relation. From memory consistency, type rules [IMI] and [METH], $\sigma \sqsupseteq \Upsilon \sqsupseteq \epsilon_c \sqsupseteq \sigma_0$, the memory is adequate from the evaluation rule [**D-IMI**], thus the evaluation succeeds, yielding the intermediate ret-expression.

Case [SMI]. the proof for two subcases [**D-SMI**] and [**D-SMI-prim**] are similar to the above.

Case [LOCAL]. We deal with expression $t\ v = e_1; e_2$. From the type rule, we have $\Gamma; \Delta; \Upsilon \vdash e_1 :: t_1, \Delta_1, \Upsilon_1$. By induction hypothesis, either (i) e_1 is a value ν, or (ii) $\langle \Pi, \varpi, \sigma \rangle\ [e_1] \hookrightarrow$ **Error-Null**, or (iii) $\langle \Pi, \varpi, \sigma \rangle\ [e_1] \hookrightarrow \langle \Pi_1, \varpi_1, \sigma_1 \rangle\ [\hat{e}_1]$.

In case (i), the type rule [LOCAL] guarantees the alias subtyping relation needed in the evaluation [**D-Blk-2**] (the function *ext*). From the memory consistency and type rule [LOCAL], $\sigma \sqsupseteq \Upsilon \sqsupseteq \{(\mathcal{S}, 1)\}$, which ensures the memory adequacy. Thus the evaluation succeeds. In case (ii), the evaluation reports **Error-Null** immediately. In case (iii), the proof follows from [**D-Blk-1**].

Case [IF]. We deal with expression **if** v **then** e_1 **else** e_2. The type rule ensures v is of type **bool**, thus there always exists an evaluation step to be taken in spite of the value of v, that is, either [**D-If-true**] or [**D-If-false**].

Case [ELF]. We deal with expression $\texttt{ret}(v^*, \rho, e)$. From the type rule, we have $\Gamma; \Sigma; \Delta; \Upsilon \vdash e :: t, \Delta_1, \Upsilon_1$. By induction hypothesis, either

(1) e is $\texttt{rel}_A(b, k, \nu)$, or
(2) e is $\texttt{rel}_B(b, k, v_0.mn(v_{1..p}))$, or
(3) e is $\texttt{rel}_B(b, k, mn(v_{1..p}))$, or
(4) e is $\texttt{ret}(u^*, [\,], \hat{e})$, or
(5) $\langle \Pi, \varpi, \sigma \rangle\ [e] \hookrightarrow$ **Error-Null**, or
(6) $\langle \Pi, \varpi, \sigma \rangle\ [e] \hookrightarrow \langle \Pi_1, \varpi_1, \sigma_1 \rangle\ [e_1]$.

In subcase (1), the proof follows from [**D-Ret-3**]. In subcase (2), the proof follows from [**D-TIM**]. In subcase (3), the proof follows from [**D-TSM**]. Note that for subcase (2) and (3), the memory adequacy is analyzed similar to the case [IMI]. In subcase (4), the proof follows from [**D-Ret-2**]. In subcase (5), the evaluation reports **Error-Null** immediately. In subcase (6), the proof follows from [**D-Ret-1**]. □