# Modular Verification with Shared Abstractions

Uri Juhasz
School of Computer Science
Tel Aviv University
Israel.

July 15, 2008

# Acknowledgements

Mooly Sagiv
Noam Rinetzky
Arnd Poetsch Heffter
Eran Yahav
Greta Yorsh

# CONTENTS

# ABSTRACT

Modular verification of shared data structures is a challenging problem: Side-effects in one module that are observable in another module make it hard to analyze each module separately. We present a novel approach for modular verification of shared data structures. Our main idea is to verify that the inter-module sharing is restricted to a user-provided specification which also enables the analysis to handle side-effects. For our approach, we constructed a novel modular static analysis and implemented a proof-of-concept analyzer. Using the analyzer, we verified some shared data structures which cannot be verified modularly by current tools.

# 1. INTRODUCTION

Abstract data type implementations are naturally implemented with pointers. Pointers permit the sharing of data structure implementations. This representation exposure of internal data structures drastically complicates the task of (modular) reasoning. However, sharing is inherent in many real life situations. For example, sharing is used in certain popular design patterns [6] such as the *model-view-controller pattern* in which a *model* is shared by a *controller*, which modifies the model, and a *view*, which reflects the up-to-date state of the model. Another common design pattern is the *adapter pattern* in which a *wrapper* modifies the behavior of a data structure it wraps (or hides some of its complexities), yet the underlying data structure may still be accessible.
In this thesis, we present a novel *modular approach* for static analysis capable of verifying partial correctness of *shared* data structures. The main results of this thesis can be summarized as follows:

1. We present a novel approach for manual and automatic modular verification of shared composite data structures. The additional proof burden required in our approach is proportional to the expected "degree of sharing".

2. We define a form of specification and enumerate the proof obligations which suffice for the modular verification of a module. The specification allows to control the permitted inter-module sharing patterns.

3. We define $LHM$, a non-standard semantics specifically designed as a foundation for modular analysis. Assertions written in our specification language which hold in $LHM$ also hold in the standard semantics. Therefore analyses which are sound with respect to $LHM$ are also sound with respect to the standard semantics.

4. We demonstrate the usage of our technique and the (reasonable) specification burden by applying it to certain real-life programming paradigms.

5. We implemented a proof-of-concept analyzer and used it to verify certain small but intricate shared implementations that cannot be verified modularly by existing tools.

**Limitations** we assume that:

(a) The programming language is object-based, i.e., with dynamic memory allocation and procedures, but without inheritance or subtyping,

(b) The module dependency relation is acyclic,

(c) Every instance of a data structure (a component) has a single object (the data structure's *header*) which dominates the entire data structure. Meaning that any heap path reaching the objects comprising the data structure passes through the *header*, except for possibly heap paths reaching the headers of sub-components (shared internal data structures) (which must come from a different module)

(d) Every data structure has a bounded number of exposed possibly-shared internal data structures.

(e) We do not handle deallocation or garbage collection

**Overview** We concentrate on presenting an extended informal overview of our approach (Sections Sec. 2 and Sec. 3), and only highlight key formal aspects — which are fully presented in the appendix.

**Outline** The rest of the thesis is organized as follows: Sec. 4.1 defines our program model. Sec. 4 describes the required specification and the proof obligations. Sec. 5 presents our non standard hybrid semantics. Sec. 6 shortly describes the analysis. Sec. 6.4 describes our implementation and a few test cases. Sec. 7 reviews related works. Sec. 9 concludes.

# 2. RUNNING EXAMPLE

Fig 2.1 shows our running example. The code is written in a Java-like language. It is comprised of three data structure instances: a client using a map and a key-set. We refer to a data structure instance as a *component* and to an internal data structure as a *subcomponent*. We refer to data structures sharing an internal data structure as *siblings*. In this example, the key-set and the client are sibling data structures: the map is a *shared subcomponent* of both of them.
In the appendix we give a more detailed and complete version.
The client initialization procedure, `Client`,
shown in Fig 2.1(a), uses a the *map* module to associate keys (integers) to values (floats). The set reflects the domain of the map: In the spirit of the `keySet()` method of a Java's maps, when the set is constructed, it is linked to a map and provides a view of the keys contained in this map. For example, the third invocation of `insert` associating key `9` to value `1.0`, has a *side-effect* on the key-set, it makes `9` a member of the key-set.
The `Keyset` class, shown in Fig 2.1(b), stores in field `map` a reference to the set's underlying map. This field is initialized upon construction and utilized to delegate methods invoked on the set to its map. Note that after the key-set is initialized, the client keeps–and uses–a reference to its underlying map.
Fig 2.1(c) shows the signature of the methods in the `Map` class. (The actual implementation, using a binary search tree, is omitted for brevity, and can be found in App. G.)
**Verification Goal** Our analysis verifies each module separately. In the following, we focus on verifying that all the instances of the `Keyset` class comply with their specification (explained below). Our analysis also modularly extracts information about the dependencies between `Keyset` and `Map` in a way that allows to verify that all the instances of the client satisfy the assertions. The main challenge we face is the propagation of side-effects due to methods invoked on shared subcomponents. For example, verifying the first assert statement stating that `9` is a member of the key-set requires propagating the side-effects of the third invocation of `insert` on the map to the key-set.

```
Module Client
  class Client{
    Map m;
    Keyset s;
   Client(){
    m = new Map();
    s = new Keyset(m);
    m.insert(3,4.0);
    m.insert(2,5.0);
    m.insert(9,1.0);
    assert s.isMember(9);
    s.remove(9);
    assert m.find(9)
        ==NOTFOUND;
   }
  }
```

(a) Client program

```
Module Keyset
  class Keyset {
   Map map;
   Keyset(Map smap){
    map=smap;
   }
   bool isMember(int k){
    return map.find(k)
       != NOTFOUND;
   }
   void remove(int k){
    map.remove(k);
   }
  }
```

(b) `Keyset` class

```
Module Map
  class Map {
   static int NOTFOUND=-1;
   ...
   Map(){...}
   void insert(int k, float val) {
    ...
   }
   float find(int k) {
    ...
   }
   void remove(int k) {
    ...
   }
  }
```

(c) A `Map` class

*Fig. 2.1:* Running Example (code).

```
model $keys \in 2^{\mathbb{N}}$

model pivot Map $amap$;
model function $amap = \mathbf{dom}(amap.map)$

pre $m \neq null$
post $ret\ fresh$
     $ret.amap = m$
     $ret.keys = \mathbf{dom}(m.map)$
Keyset(Map m)

post $ret = k \in keys$
bool isMember(int k)

post $keys' = keys \setminus \{k\}$
     $amap.map' = amap.map|_{\mathbf{dom}(amap.map) \setminus \{k\}}$
void remove(int k)
```

(a) `Keyset` interface specification

```
rep pivot
   $amap = map$

rep function
   $keys = \mathbf{dom}(map.map)$
```

(b) `Keyset` internal specification

```
model $map \in \mathbb{N} \hookrightarrow_{fin} \mathbb{R}$

post $ret.map = \emptyset$
     $ret\ fresh$
Map()

pre $k \geq 0$
post $map' = map[k \mapsto v]$
void insert(int k, float v)

post $ret = k \in dom(map)$ ?
              $map(k)$ : NOTFOUND
float find(int k)


post $map' =$
   $map|_{\mathbf{dom}(map) \setminus \{k\}}$
void remove(int k)
```

(c) `Map` interface specification

*Fig. 2.2:* Running example (specification). Fields that are not modified are not mentioned.

# 3. OUR APPROACH

In this section, we provide an informal overview of our approach. Our approach builds on the seminal work of Hoare [7] which allows for manual modular verification of abstract data structures. Hoare proposes the use of a *representation function* which maps the representation of the data structure's state to an *abstract value*. The specification of the data structure's procedures is given in terms of abstract values. This allows a client of the data structure to be independent of the data structure's actual implementation. Hoare's work does not support pointers and dynamically allocated memory. However, it can be extended naturally to nested (encapsulated) data structures, i.e., where subcomponents can be organized only in the form of a tree.

## *3.1 Controlled exposure of sharing*

In this work, we provide an approach for manual and automatic modular verification of composite data structures in the presence of shared subcomponents. In this setting, any sound analysis needs to consider side-effects due to sharing. *Our main idea is to have a controlled exposure of the sharing of subcomponents instead of forbidding it or ignoring it.*

We support shared subcomponents with the aid of a user specification that: (i) exposes subcomponents that can be referenced by other data structures; (ii) provides the [lack of] effect of every operation invoked on the data structure not only on the abstract value of the data structure, but also *on the abstract value of any subcomponent that it may share with others*. A key feature of this specification is that it exposes the effect of a procedure call on shared *subcomponents*, but it does *not* specify the effect of the procedure on sibling data structures, thus enabling modular reasoning.

## *3.2 The aggregate model function*

The main challenge that we face is the tracking of indirect changes to the abstract values of sibling data structures. Such changes may occur, e.g., when a (shared) subcomponent of two data structures is modified.

We address the above challenge using a user-specified *aggregate model function*. The latter allows us to compute a conservative approximation of the (side-affected) data structure's abstract value after an indirect change. It computes the (updated) data structure's abstract value based on the abstract values of its internal representation (including unshared subcomponents) and the abstract values of its shared subcomponents. When analyzing sibling data structures the analysis tracks which subcomponents are shared and by whom. When the abstract value of a shared subcomponent is modified, the analysis *updates* the abstract values of the affected data structures using their *aggregate model functions*.

Our approach uses two kinds of additional fields to record the abstract value of a data structure and to define the aggregate model function. *Model fields* are used to record the abstract values of data structures. *Model pivot fields* are used to record the inter-component reference structure.

Fig 2.2(c) shows the (user-provided) interface specification of the `Map` class. A map implements a partial function from integers to reals. The abstract value of a map is a function from integers to floats. We use the model field $map$ to record this value. The map is an example of a fully-encapsulated data structure. The `Keyset` data structure, whose interface specification is shown in Fig 2.2(a), is an example of a data structure implementation which allows a controlled exposure of subcomponents. The abstract value of the `Keyset` is a set of integers. The model field $keys$ records this value. In addition, the `Keyset` exposes a *model pivot field*. The model pivot field is a special sort of a model field whose value is a reference to a subcomponent. It is initialized with the map parameter when a key-set is constructed and allows the exposure of the effect of the key-set's method's on this (possibly) shared component. For example, the specification of the `Keyset`'s `remove` method exposes the fact that removing a key from a key-set also removes it from the (abstract value of the) map. The aggregate model function of the `Keyset` class is shown in Fig 2.2(a). It specifies how to compute the value of the model field $keys$ when the abstract value of its underlying map changes. Specifically, it specifies that the value of the key-set's model field $keys$ should be updated to the domain of the abstract value of the map referenced by the key-set's model pivot field $amap$. Recall that the client of the map (and of the key-set) is aware of the connection between the key-set and the map because the key-set exposes its pivot field $amap$.

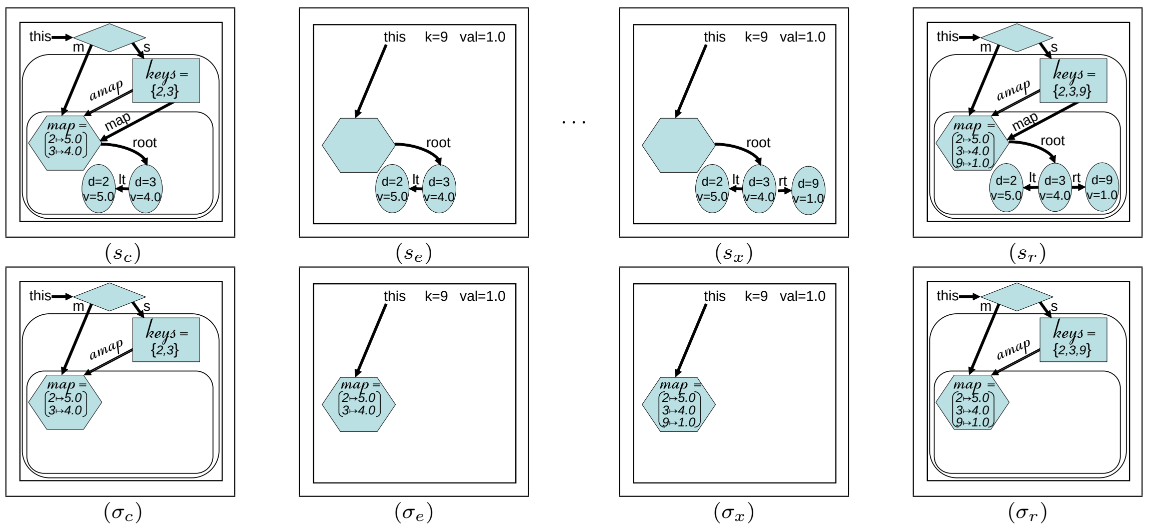

*Fig. 3.1:* Possible memory states occurring in an invocation of `m.insert(9,1.0)` on $\sigma_c$.

## 3.3 Non-standard semantics

We verify that a program is correct according to its specification by abstract interpretation [3] of $LHM$, a non-standard *componentized hybrid semantics*. $LHM$ abstracts the standard semantics (instrumented to also track the values of model fields and model pivot fields) by forgetting the *concrete* (non model) fields of objects that are not the current objects.

Fig 3.1 ($s_c$) depicts a possible memory state that may arise in the execution of the running example before the third invocation of `insert` according to an (instrumented) standard semantics which is instrumented to track also the values of model fields. The state contains a client object (depicted as a diamond). The client object has two fields, depicted as labeled edges: An `s`-field pointing to a key-set, depicted as a rectangle, and an `m`-field pointing to a map, depicted as a hexagon. The key-set's `map`-field also points to the map object. The map contains two associations (shown as ellipses).

The rectangular frames depict the *component-decomposition* of the memory state. We say that two objects *reside within* the same implicit component if they belong to the same module and are connected in the heap via an *undirected heap path* which only goes through objects that belong to the same module. We say that a component is an *unsealed component* if it is the current component (its head is pointed to by this) and a *sealed component* otherwise. A memory state $s_c$ is comprised of three components. The key-set component and the map component are sealed.

The (instrumented) standard semantics records the values of model fields and model pivot fields only for object in sealed components. The value of the model pivot field $amap$ is depicted as a double-line edge emanating from the set object to the map object. The values of the model fields $keys$ and $map$ are depicted inside the key-set and map object, respectively. For example, the (local) model field $map$ of the map is the partial function $[2 \mapsto 5.0, 3 \mapsto 4.0]$.

Fig 3.1 ($s_e$) depicts the part of the memory state which the `insert` procedure can reach after being invoked for the third time. This part of the memory state is transformed into the one shown in Fig 3.1 ($s_x$) by the execution of the `insert` procedure. The standard memory state shown in Fig 3.1 ($s_r$) results when the call to `insert` returns to the `Client`. Note that while `insert` could not directly modify fields in object which it could not reach, it does have a side effect on the abstract value of the key-set's $keys$ model field.

The $LHM$ memory states depicted in Fig 3.1 ($\sigma_c$), Fig 3.1 ($\sigma_e$), Fig 3.1 ($\sigma_x$), and Fig 3.1 ($\sigma_r$), abstract the standard memory states depicted in Fig 3.1 ($s_c$), Fig 3.1 ($s_e$), Fig 3.1 ($s_x$), and Fig 3.1 ($s_r$), respectively. Note that the $LHM$ memory states represents the abstract values of sealed components, but abstract away their representation. Also note that the inter-component reference structure is maintained by the model pivot fields.

The $LHM$ memory state shown in Fig 3.1 ($s_x$), representing the memory state when the `insert` procedure terminates, is computed directly from the memory state shown in Fig 3.1 ($s_e$), representing the memory state when the `insert` is invoked using the specification of `insert`. Furthermore, the model pivot field $amap$ exposing the dependency of the value of the key-set's $keys$ model field on the abstract value of the map allows computing the updated value of $keys$ using its aggregate model function.

# 4. PROGRAM MODEL, SPECIFICATION AND PROOF OBLIGATIONS

In this section, we describe our program model, the required user-provided specification, and the proof obligations. Any approach, be it manual or automatic, which can establish the proof obligations with the given specification verifies soundly that the module respects its specification. In the following sections, we present an automatic approach which achieves this purpose.

## *4.1 Program Model*

We analyze imperative object-based (i.e., without subtyping) programs. A program consists of a collection of procedures and a distinguished `main` procedure. The programmer can also define her own types (à la `Java` classes). We expect to be given a partitioning of the program types and procedures into modules.
**Syntactic domains** We assume the syntactic domains $x \in \mathcal{V}$ of variable identifiers, $f \in \mathcal{F}$ of field identifiers, $T \in \mathcal{T}$ of type identifiers, $p \in \mathcal{PID}$ of procedure identifiers, and $m \in \mathcal{M}$ of module identifiers. We assume that types, procedures, and modules have unique identifiers in every program.
**Modules** We denote the module that a procedure $p$ belongs to by $m(p)$ and the module that a type identifier $T$ belongs to by $m(T)$. A module $m_1$ *depends* on module $m_2$ if $m_1 \neq m_2$ and one of the following holds: (i) a procedure of $m_1$ invokes a procedure of $m_2$; (ii) a procedure of $m_1$ has a local variable whose type belongs to $m_2$; or (iii) a type of $m_1$ has a field whose type belongs to $m_2$.
**Procedures** A procedure $p$ has local variables and formal parameters, which are considered to be local variables. Only local variables are allowed. We assume that the target of a procedure invocation is bound to an implicit `this` parameter. For each field access x.f=e or y = x.f by a procedure $p$ we require that the type $T$ of the object pointed to by x satisfies $m(p) = m(T)$.

### 4.1.1 *Simplifying assumptions*

To simplify the presentation, we make the following assumptions: (a) Every module defines a single data structure which can be used by other modules; (b) Every module has a specially-designated *initialization procedure* which allocates new instances of its data structure and initializes them - a constructor.

## 4.2 *Standard Module Specification*

**Interface specification** We expect to have a user-supplied Hoare-style *"public" specification* of the program's modules:

- Every module may have *model fields*. The model fields represent the abstract value of the data structure implemented by the module. A model field should range over a value domain. For example, the (only) model field of the `Keyset` module is $keys$ and its possible values are sets of integers. The model field of the `Map` module is $map$ and its possible values are maps of integers to floats.
- Every procedure should have a pre-post specification which describes the effect of the procedure on the model fields of its parameters. A pre-post specification is *admissible* if it refers to a relation over expressions over the callee's local model heap — i.e. *access paths* starting at the procedure parameters and following a sequence of model pivot fields or to the values of model fields that can be reached by traversing such access paths. For example, the specification of the map's `insert` procedure indicates the abstract value of a map after the invocation of `insert(k,v)`, indicated by a $map'$, is the same as the map's value before the insertion except that the key $k$ is now associated with the value $v$. As a negative (inadmissible) example — if Map's specification were to mention the keys field of a Keyset object refering to it it would be inadmissible as this Keyset is not reachable from the Map. In the following, we assume that procedure specifications are admissible.

**Internal specification** In addition, we expect an *internal module specification* which provides a *representation function* [7] mapping the data structure's concrete representation to its abstract value. We allow the representation function to be defined only as a function of (i) the values of objects reachable from the component's header via fields defined in the module and (ii) the values of model fields of subcomponents.

## *4.3 Additional Specification Burden*

The additional specification burden required by our approach (compared, e.g., to [7]) is the specification of the *model pivot* fields and the *aggregate model functions*.

The model pivot fields specify the data structure's subcomponents that can be shared with other data structures. Any such subcomponent is given an externally-visible name and any procedure which has a side-effect on the abstract value of such a subcomponent is required to expose it in its specification. For example, the `Keyset` module has a (single) model pivot field $amap$. The set's initialization function, `Keyset` specifies that $amap$ names the map passed as parameter. The set's `remove` function specifies that the removed key is extracted not only from the set's model field, but also from the map which $amap$ refers to. We emphasize that a model field is allowed to have reference (location) value only if it is a model *pivot* field.

We refer to a model field whose value does *not* depend on model fields of externally — visible subcomponents as a *local model field.* An aggregate model function is *admissible* if it is defined as a function of the values of the component's local model fields and the model fields of its subcomponents. For example, the aggregate model function associated with the model field $keys$ of the `Keyset` module specifies that the value of this field is the domain of the abstract value of the key-set's map. If we change the key-set to be a *filtered* key-set which contains only keys whose values are bigger than a given threshold, then the threshold value could be exposed by a local model field and the aggregate model function of the filtered key-set would specify that the key-set contains all elements in the domain of its map that are smaller than the (exposed) threshold.

**Internal specification** The internal specification is used only during the verification of the module itself. It links the model pivot fields and model fields with the actual implementation of the module's data structure.

Every local model field is associated with a *representation function* that specifies its value. The representation function can depend only on properties of objects that are reachable from the header of the data structure through fields defined in the module.

The representation function for every model pivot field is further limited to only access fields inside the current component (so it depends only on its component) and must evaluate to an outgoing pointer.

For example, the model pivot field $amap$ is associated with the Keyset's `map`-field which points to a `Map`.

Our analysis finds *module invariants* (properties which are expected to hold when the data structure is not being manipulated) in an automatic manner. For example, a module invariant of the `Keyset` module which our analysis find is that the field `m` never has a `null` value.

## *4.4 Proof Obligations*

When verifying a module, we need to verify, as usual, that the postcondition of every procedure $p$ and the module invariant are implied from executing $p$'s body in any state which satisfies $p$'s precondition and the module invariant. (When verifying an initialization function only the precondition can be assumed.) In addition, we need to establish the following properties:

1. *model function consistency*, i.e., evaluating the model function of a model field mf over a model heap mh is conservative w.r.t. evaluating the representation function of mf over an equivalent hybrid heap hh (terms formally defined in the appendix).

2. *model pivot consistency*, i.e., in any state which satisfies the module invariant, every reachable subcomponent which is not dominated by the data structure's header (i.e., possibly exposed) is named (pointed to) by a model pivot field

3. *single header*, i.e., every component has a single header.

4. *model function dependency locally acyclic*, i.e., there is no local dependency cycle between model functions of the same component

This admissibility of the aggregate model function in addition to the single header requirement and the acyclicity of the module import relation ensure that these functions are well defined, i.e., there is no cyclic definition of aggregate model functions.

# 5. COMPONENTIZED-HEAP CONCRETE SEMANTICS

In this section we define $\mathcal{CHS}$, a non standard concrete semantics which serves as a foundation for our modular analysis: We establish the proof obligations in $LHM$. Our restrictions on the program model and on the specification ensure that even though the proof obligations are shown to hold in $LHM$, the properties that they assert also hold in the standard semantics.

$\mathcal{CHS}$ is a *store-based* semantics (see, *e.g.*, [21]). A traditional aspect of a store-based semantics is that a memory state represents a heap comprised of all the allocated objects. $\mathcal{CHS}$, on the other hand, is a *local heap* semantics [22]: A memory state which occurs during the execution of a procedure does not represent *objects* which, at the time of the invocation, were not reachable from the actual parameters.

$LHM$ is a hybrid semantics: It only represents the fields of the current component. It represents sealed components using their headers and only records the model and model pivot fields of these headers.

$LHM$ is a "mixed-step" semantics: When executing intraprocedural statements and *intra-module* procedure calls, it acts as small-step operational semantics [20]. However, instead of encoding a stack of activation records inside the memory state, as traditionally done, $LHM$ maintains a *stack of program states* [10]). The use of a stack of program states allows us to represent in every memory state the (values of) local variables and the partial heap of just one (the *current*) procedure. When executing an *inter-module* procedure call, $LHM$ acts as large-step operational semantics [8]: it computes the effect of a procedure invocation in "one step" using the procedure's specification.

$LHM$ checks that the program memory states satisfy certain *admissibility conditions* (listed below). Thus, $LHM$ may abort whereas standard heap semantics would not abort. Such an abort means that a module does not satisfy our restrictions. (Our analysis conservatively detects such aborts.) For brevity, we only informally discuss the relation between $LHM$ and the standard heap semantics and describe key aspects of the operational semantics. In A.4, we formally define $LHM$ and relate it to the standard semantics using Galois connections.

| | | | |
|---|---|---|---|
| $l$ | $\in$ | $\mathit{Loc}$ | Location |
| $var$ | $\in$ | $\mathit{VarId}$ | Variable id |
| $\varepsilon$ | $\in$ | $\mathit{Env} = \mathcal{V} \hookrightarrow \mathit{Val}$ | Local variables |
| $v$ | $\in$ | $\mathit{Val} = \mathit{Loc} \cup \{\mathit{null}\} \cup \mathbb{N} \cup \mathbb{F}$ | Concrete value |
| $h$ | $\in$ | $\mathcal{H} = \mathit{Loc} \hookrightarrow \mathcal{F} \hookrightarrow \mathit{Val}$ | Concrete heap |
| $t$ | $\in$ | $\mathit{Type}$ | Type |
| $t$ | $\in$ | $\mathcal{TM} = \mathit{Loc} \hookrightarrow \mathcal{T}$ | Type heap |
| $a$ | $\in$ | $AVal$ | Model value |
| $ah$ | $\in$ | $\mathcal{AH} = \mathit{Loc} \hookrightarrow \mathcal{F} \hookrightarrow AVal$ | Model heap |
| $ph$ | $\in$ | $\mathcal{PH} = \mathit{Loc} \hookrightarrow \mathcal{F} \hookrightarrow \mathit{Val}$ | Pivot heap |
| $\sigma$ | $\in$ | $\Sigma = \mathit{Env} \times 2^{\mathit{Loc}} \times \mathcal{H} \times \mathcal{TM} \times \mathcal{M} \times \mathcal{AH} \times \mathcal{PH} \times \mathcal{AH}$ | Memory states |

*Fig. 5.1:* Semantic domains.

## 5.1 Memory States

Fig 5.1 defines the concrete semantic domains and the meta-variables ranging over them. We assume $\mathit{Loc}$ to be an unbounded set of locations. A value $v \in \mathit{Val}$ is either a location, the special *null* value, an integer or a float.

A memory state in the $LHM$ semantics is an 8-tuple:
$\sigma = \langle \varepsilon, L, h, t, m, \mathit{alh}, \mathit{ph}, \mathit{ah} \rangle$. The first four components comprise a 2-level store: $\varepsilon \in \mathit{Env}$ is an environment assigning values for the variables of the *current* procedure. $L \subset \mathit{Loc}$ contains the locations of allocated objects. (An object is identified by its location. We interchangeably use the terms object and location.) The *heap* $h \in \mathcal{H}$ assigns values to fields of allocated objects. $t \in \mathcal{TM}$ maps every allocated object to the type-identifier of its (immutable) type. Implicitly, $t$ associates every allocated location to a module: The module that a location $l \in L$ *belongs to in memory state* $\sigma$ is $m(t(l))$. The fifth component, $m \in \mathcal{M}$, is the module of the current procedure. We refer to $m$ as the *current module* of $\sigma$. The sixth component, $\mathit{alh} \in \mathcal{AH}$, is the *local abstract value map*. It records the abstract values of *local model fields*. The seventh component, $\mathit{ph} \in \mathcal{PH}$, is the *model pivot heap*, recording the reference values of pivot model fields. The eighth component, $\mathit{ah} \in \mathcal{AH}$, is the *abstract value map*. It records the abstract values of non-local model fields.

To exclude states that cannot arise in any program, we now define the notion of admissible states. We note that $LHM$ preserves the admissibility of memory states.

A $LHM$ memory state $\sigma = \langle \varepsilon, L, h, t, m, \mathit{alh}, \mathit{ph}, \mathit{ah} \rangle \in \Sigma$ is **admissible** if

$$
\begin{array}{l}
\langle Call_{y=p(x_1,\ldots,x_k)}, \sigma_c\rangle \xrightarrow{CHS} \sigma_r \\
\text{ where:} \\
\quad \sigma_e = \langle \varepsilon_e, L_c, \emptyset, t_c|_{L_{rel}}, m_c, alh_c|_{L_{rel}}, ph_c|_{L_{rel}}, alh_c|_{L_{rel}}\rangle \\
\qquad \varepsilon_e = [z_i \mapsto \varepsilon_c(x_i) \mid 1 \le i \le k] \\
\qquad L_{rel} = R(\{\varepsilon_c(x_i) \in Loc \mid 1 \le i \le k\} \\
\quad \sigma_e \overset{[\![p]\!]}{\leadsto} \sigma_x \\
\quad \sigma_r = \langle \varepsilon_c[y \mapsto \varepsilon_x(ret)], m_c, L_x, h_c, t_c[t_x], alh_r, ph_r, ah_r\rangle \\
\qquad alh_r = alh_c[alh_x] \\
\qquad ph_r = ph_c[ph_x] \\
\qquad ah_r = ah_x \cup \bigcup_{l \in dom(ah_c)\setminus dom(ah_e)} update(l, ah_c, alh_r, ph_r, t_r, ah_x) \\
\\
update(l, ah_c, alh_r, ph_r, t_r, ah_x) = \\
\quad \begin{cases}
ah_x & l \in dom(ah_x) \\
ah_{sub} \cup \{l \mapsto f \mapsto m_f(l, alh_r, ah_{sub}, ph_r) \mid f \in model(t_r(l))\} & \text{otherwise} \\
\quad ah_{sub} = \bigcup_{l' \in \text{subcomponents}} update(l', ah_c|_{L'_{rel}}, alh_r|_{L'_{rel}} ph_r|_{L'_{rel}}, t_r|_{L'_{rel}}, ah_x|_{L'_{rel}}) & \\
\quad\ \text{subcomponents} = \{l' \in \{ph_r(l,p)|p \in pivot(t_r(l))\} & \\
\quad\ L'_{rel} = R(\{l'\}, ph_r) &
\end{cases} \\
R(L, ph) = \{l \in dom(ph) | l \text{ reachable from } l' \in L \text{ in } ph\}
\end{array}
$$

*Fig. 5.2:* The axioms for an arbitrary intermodule procedure call $y = p(x_1, \ldots, x_k)$ assuming $p$'s formal parameters are $z_1, \ldots, z_k$ and $p$ returns its value by assigning it to a specially designated variable $ret$. The call state is $\sigma_c = \langle \varepsilon_c, L_c, h_c, t_c, m_c, alh_c, ph_c, ah_c\rangle$ The exit state is $\sigma_x = \langle \varepsilon_x, L_x, h_x, t_x, m_x, alh_x, ph_x, ah_x\rangle$ is a possible outcome of $p$ when invoked on memoy state $\sigma_c$ according to $p$'s specification, $[\![p]\!]$. For a type $t \in \mathcal{T}$, $model(t)$ denotes the model fields of $t$, $pivot(t)$ denotes the model pivot fields of $t$. The aggregate model function associated with a model field $f$ is denoted by $m_f$.

(i) The domain of the heap and the local abstract value map, the model pivot heap, and the abstract value map are disjoint, i.e., let $A = dom(alh) \cup dom(ph) \cup dom(ah)$, then $dom(h) \cap A = \emptyset$; (ii) Every object in the domain of the heap belongs to the current module, i.e., for every $l \in dom(h)$, $m(t(l)) = m$; (iii) The type of every object in the (local) abstract values map and the model pivot heap does not come from the current module, i.e., for every $l \in A$, $m(t(l)) \neq m$, where $A$ is as defined above; (iv) Fields and model fields can point only to allocated locations, i.e., $\{h(l)f \in Loc, ph(l)f \mid l \in L, f \in \mathcal{F}\} \subseteq L$; and (v) The pivot heap is acyclic.

Figures 3.1($\sigma_c$) and 3.1($\sigma_r$) depict the (admissible) $LHM$ memory states that arise in the execution of the running example before and after the third invocation of `insert`, respectively.

## 5.2 Operational Semantics

We only discuss the key aspects of the operational semantics, formally defined in App. A.4.

### 5.2.1 Intraprocedural Statements

Intraprocedural statements are handled, essentially, as usual in a two-level store semantics for pointer programs (see, *e.g.*, [21]). The main difference from the standard semantics, formalized in App. A.3, is that the semantics checks that the program accesses only fields of objects that belong to the current component.

### 5.2.2 Intra-module interprocedural statements

$LHM$ is a local-heap semantics [22] which maintains a *stack of program states* to handle intra-module procedure calls [23]. The program state of the *current procedure* is stored at the top of the stack, and it is the only one which can be manipulated by intraprocedural statements.

### 5.2.3 Inter-module interprocedural statements

Fig 5.2 defines the axioms for intermodule procedural calls. When an intermodule procedure call is invoked, $LHM$ computes the return state $\sigma_r$ in three steps, as described below.
First, it computes an intermediate memory state, $\sigma_e$, which represents the callee's local heap but with all components remaining sealed. It does this by restricting the heap to the part reachable from the actual parameter. Because the module dependency is acyclic, only headers of sealed components (reachable via the pivot heap) can be passed. It then checks that the resulting memory state satisfies the pre-condition of the invoked procedure, and aborts otherwise.
Then, $LHM$ applies the effect of the invoked procedure in one step: it selects (nondeterministically) any exit state which is a possible outcome of the invoked procedure according to its specification. Because the invocation context is unknown, the specification cannot refer to the current component.
Finally, $LHM$ computes the return state of the procedure by carving out the local heap passed to the callee from the call heap and replacing it with the callee's heap at the exit site. This operation correctly updates the heap, the type map, the local abstract value map, and the model pivot heap. To correctly propagate

the side effect of the invoked procedure on the (aggregate) model fields the semantics updates the value of every model field pertaining to an object which was not passed to the callee using the $update$ procedure. The invocation of $update(l, ah_c, alh_r, ph_r, t_r, ah_x)$ updates the model fields of $l$ by first (recursively) computing $ah_{sub}$, an abstract value map containing the updated values of model fields of every subcomponent of $l$. (Note that to determine the subcomponents, the semantics uses the updated pivot heap.) $LHM$ then computes the value of every model field $f$ of $l$ using $m_f$, the aggregate model function of $f$. Note that this computation is well defined because of the acyclicity of model field dependencies promised by our assumptions.
We note that the above computation is rather inefficient as it recomputes values of fields that cannot be modified. In Sec. A.4, we present an optimized version which updates only fields that *depend on* potentially modified fields. The optimized version behaves better under abstraction.

## 5.3 Observational Soundness

Our goal is to verify (modularly) properties of modules according to the *standard* semantics. The admissibility of $LHM$ memory states, our programming model, and the conditions checked by the operational semantics, ensure that if the $LHM$ semantics never aborts when executing a module, then for any memory state $s$ that can arise according to the *standard* semantics (in a program comprised only of such "well-behaved" modules) there exists a $LHM$ memory state $\sigma$ which arises during the execution of the program in the $LHM$ semantics which abstracts $s$.
An immediate consequence of the above is that any assertion regarding properties of objects reachable from component headers which is shown to hold with respect to $LHM$ semantics also holds with respect to the standard semantics. For a formal definition of the observational soundness theorem, see Sec. B.

# 6. STATIC ANALYSIS

This section presents key aspects of our (conservative) modular static analysis. The analysis is obtained as an abstract interpretation of $LHM$ using a *bounded* conservative abstraction. Our analysis is parametric in the bounded abstraction and can use different (bounded) abstractions when analyzing different modules. In our implementation, we use canonic abstraction [24].
Our static analysis is conducted in an assume-guarantee manner allowing each module to be analyzed separately. The analysis, computes a conservative representation of every possible input state to an intermodule procedure call. This process, in effect, identifies structural *module invariants*. Our analysis verifies conservatively that a module satisfies its specification and respects the restrictions we impose.
The main challenge in our analysis lies in finding all the possible input states to an intermodule procedure calls. We overcome this challenge by utilizing the fact that in $LHM$ whenever a program passes a component of the analyzed module a parameter to an intermodule procedure call, it must be a sealed component which was previously generated by the program. (This is derived from the fact that every component has a single header and that a component can be manipulated only by the module which generated it). In particular, we can *anticipate the possible entry memory states of an intermodule procedure call*: Note that components are sealed only when an intermodule procedure call returns. Furthermore, the only way a sealed component can be mutated is to pass it back as a parameter to a procedure of its own module. Thus, a partial view of the execution trace, which considers only the executions of procedures that belong to the analyzed module, and collects the sealed components generated when an intermodule procedure invocation returns, can anticipate (conservatively) the possible input states for the next intermodule invocations. Specifically, *only* a combination of *already generated sealed components* of the module can be the component parameters in an intermodule procedure invocation.
We conservatively compute the effect of procedure calls on subcomponents using the user-provided specification. The procedure's pre-post specification allows us

to find the effect of the procedure on the components passed to it as parameters (and their subcomponents) and the aggregate model function allows us to propagate side effects to sibling components.

## *6.1 Automatic Verification*

Our analysis verifies each module separately. It computes conservatively all possible input states of intermodule procedure calls by exercising the analyzed module using its most-general-intrusive-client (*MGIC*): a program that simulates all possible procedure invocations on the analyzed component (thus simulating arbitrary usage contexts) and on its exposed subcomponents (thus simulating all possible side-effects).
We obtain an effective analysis by abstracting the hybrid semantics using standard shape abstraction [24] which conservatively represents every (unbounded) hybrid memory state in a bounded way.
**Modularity** Our analysis is modular in two aspects. First, it is modular in the program code: When verifying one module, we only require the code implementing that module and only the specification of the other modules. (Specifically, we determine the effect of an inter-module procedure call on a subcomponent using the procedure's specification.) Second, it is modular in the program state: when reasoning about a module, we

(i) Reason about the *concrete* representation of data structures manipulated by the module,

(ii) Represent the *abstract* values and the topologies (*sharing patterns*) of subcomponents that come from other modules.

(iii) Ignore the data structure context containing the analyzed data structures.

(In comparison, Hoare's approach, by being targeted to verify only fully encapsulated data structures, can avoid reasoning about sharing patterns.) A key reason for the modularity of our analysis in the program state is the heterogenous memory representation of the *hybrid* states. This allows our analysis to require only the specification of dependant modules and not their implementation (as is required, e.g., in [26]).
**Burden on the user** The additional proof burden (beyond the one imposed, e.g., by Hoare's approach) that we place is proportional to (i) the *allowed sharing*, i.e., to the number of subcomponents of the verified data structure which are allowed

to be shared externally and the interconnection between them, and (ii) the *actual sharing*, i.e., the sharing between the data structure subcomponents.

*Recap*

The essence of our work is the propagation of the side-effects on the abstract values of sibling data structures. The aggregate model function provides a conservative approximation of the data structure's abstract value based on the abstract values of its internal, unshared parts and the abstract values of its shared subcomponents.

## 6.2 Bounded abstraction

We provide an effective conservative abstract interpretation [3] algorithm which determines *module invariants* by devising a bounded abstraction of $LHM$ memory states. An abstraction of a $LHM$ memory state, being comprised essentially of an environment of a single procedure and a subheap, is very similar to an abstraction of a standard two-level store. The additional elements that the abstraction tracks are the model pivot fields (which can be abstracted in a similar manner to standard concrete fields) and model fields representing abstract values of data structure. Thus, the abstract domain is expected to be able to represent (conservatively) the abstract values used in the specification.
Abstracting a $LHM$ memory state is simpler than abstracting standard memory states: Instead of abstracting the representation of a data structure (e.g., the representation of a map as a tree) the abstraction needs to record the essential properties of the data structure, e.g., the association between keys and values, the elements in the domain of the map, etc. Furthermore, we believe that a key reason for the success of our analyzer is the fact that while a procedure call might have a complicated effect on the concrete heap, e.g., inserting a node to a tree, its effect on the abstract value of the data structure can be much more limited, e.g., adding an association to a map.

## 6.3 Modular analysis

We conduct modular static analysis by performing an interprocedural analysis of a module together with its MGIC. The module's *MGIC* invokes a sequence of arbitrary sequence procedures of the module using arbitrary input arguments. (In

this respect, the most-general-intrusive-client is similar to the most-general-client of a class [23].) However, it also invokes procedures directly on the exposed subcomponents of the analyzed module. (In this respect, the client is intrusive as it bypasses the component and directly interacts with its subcomponents). This allows to simulate conservatively any arbitrary context in which components of the module can be used.
Our analyzer verifies that every intermodule procedure call made by the MGIC respects the procedure's specification.

## 6.4 Experimental evaluation

We have experimented with expressing several examples in our system. We realized our system by developing a proof-of-concept modular analyzer using canonic abstraction [24] within the TVLA system [15]. Our analyzer was able to verify the `Keyset` module in 10 seconds running on a machine with a 2.66 Ghz Core 2 Duo processor and 2 Gb memory. The verification proved that any Keyset component in any context using any map that conforms to the Map specification would behave as in its specification.
The MGIC for the Keyset of the running example is detailed in Fig 6.1. The analysis uses the MGIC as follows: The Keyset is represented in the concrete space. The Keyset methods are executed using the body. The map is represented in the model space (just m.map - no internal nodes). The map methods are executed using specification only (also calls in Keyset method bodies). The value s.keys is calculated using the representation function in each assertion that uses it. Pre and post conditions are checked before/after each call (in the model space). The last assertion checks that the model function over-approximates the representation function - both calculated from the hybrid view. The question marks represent non-deterministic selections.

```
MGIC(){
   Map m;
   Keyset s;
   //Initialize a general map
   m = new Map();
   while (?){
    switch (?) {
      case 0: m.insert(?,?); break;
      case 1: m.remove(?); break;
    } //switch
   } //while

   //Excercise all use-cases of Keyset
   s = new Keyset(m);
   while (true){
     int key = ?;
     float val = ?;
    switch (?) {
      case 0:
       assume($true$);
       m.insert(key,val);
       assert($m.map' = m.map \ \cup \ (key, val)$);
      case 1:
       assume($true$);
       m.remove(key);
       assert($m.map' = m.map \ \setminus \ \{key\}$);
      case 2:
       assume($true$);
       bool b = s.isMember(key);
       assert($b == key \in \ s.keys$);
       assert($m.map' = m.map$);
       assert($s.keys' = s.keys$);
       break;
      case 3:
       assume($true$);
       s.remove(key);
       assert($m.map' = m.map \ \setminus \ \{key\}$);
       assert($s.keys' = s.keys \ \setminus \ \{key\}$);
       break;
    } //switch
     assert( $s.keys \ \in \ Keyset.keys\_m(s)$);
   } //while
  }
```

*Fig. 6.1:* MGIC for the Keyset component.

# 7. RELATED WORK

In this section, we review related works.
**Modular verification.** In [1], Barnett and Naumann presented verification approach that supports state dependencies across ownership boundaries. A component can grant "friends" the right to depend on their state. Thus, the dependency is publicly visible and treated in a modular way. The downsise of this technique is that the granting component needs to know its clients. To overcome this restriction, Leino and Schulte developed a technique based on history invariants [14] which allow component invariants to depend on other non-encapsulated components, which resembles our method. However, the expressible dependencies are weaker than the ones allowed by our method since they need to be captured by a history constraint.
A modular solution for model fields and non-shared abstractions is developed in [17, 13]. The focus of that work is as well on object invariants, and the problem of when abstract and concrete state have to be consistent is similar. The methods shown in those papers are more suitable for manual verification or theorem provers rather than analysis. Those works inspired some of our definitions for when an abstraction has to hold and the ideas of pivots and dependencies.
In [9], Kassios presents a framework for modular reasoning supporting abstraction, data-hiding, and subtyping. The framework allows asserting properties such as disjointness and inclusion between the domains of different representation functions and to prove modularly *non-modification* of model fields. Classes can have "public" invariants which encode similar information to our "public" model function, however, in a less disciplined way. We give explicit rules for pivots and rely on pivot equality while Kassios uses frames which are sets of pointers and relies on set operations (inclusion, disjointness etc) which we believe are harder to use in practice. Our restrictions give us simpler proof obligations with simpler propagation of side-effects which can be verified automatically.
**Local reasoning** [19] and [2] allow to conduct modularly (manual) local reasoning [21] about abstract data structures and abstract data types with inheritance, respectively. The reasoning requires user-specified resource invariants and loop

invariants. Our analysis automatically infers these invariants based on user provided interface specification and representation functions (and an instance of the bounded shape abstraction). [2], however, allows for more sharing than in our model. Their system can encode "public" invariants of aggregates and model pivots, but does not have an inherent notion of aggregate representation function.

**Modular static analysis** [4] describes the fundamental techniques for modular static program analysis. These techniques allow to compose separate analyses of different program parts. We use their techniques, in particular, we use *user provided specification* to communicate the effect and *side effects* of mutations done by different modules.

[11, 25] also utilize user-specified pre- and post- conditions to achieve modular shape analysis which can handle a bounded number of flat set-like data structures. Our approach, allows for separately-analyzed arbitrarily-nested and possibly shared sets.

[16] presents a modular analysis which infers class invariants. The determined invariants do *not* concern properties of *shared* subobjects.

[26] provides a method for computing the effect of a procedure call which is modular in the program code — but not in the program state: A theorem prover is used to propagate the effect of a procedure call on the abstract field of the caller by inferring it from the call's effect on the concrete fields of the callee. Intuitively, their approach requires maintaining the values of concrete fields for subcomponents.

[23] presents a *modular* shape analysis which identifies structural (shape) invariants for dynamically encapsulated programs: heap-manipulating programs which forbids sharing between components via live (i.e., used before set) references. This paper allows shared components but requires that every shared subcomponent be named by a model pivot field. This may restrict our approach from handling data structures which hold object data which are transferred as parameters. One way we can overcome this restriction is by incorporating in our approach dynamic encapsulation for certain kinds of object parameters.

# 8. DISCUSSION AND FUTURE WORK

Our presentation uses some restrictions on the programming language and the valid programs and requires some additional burden from the programmer, here we discuss the implications of these restrictions and the options for alleviating them.

## *8.1 Discussion*

Our system as is can support some interesting programs like the running example but is lacking in several key aspects. Our main advantage over most existing and proposed systems is that we allow completely modular verification with shared abstractions in a practical manner. The main shortcomings of our approach is that it requires a substantial specification effort from the programmer — the model function (over Hoare's approach), and that several common language features are not supported — subtyping, genericity, casts, multiple headers per component, cyclic module import etc. We believe some additional specification burden is mandatory for the added program complexity of shared abstractions in a modular setting. We discuss the missing features in the next section, and offer ways to include most of them. The additional verification burden (proof obligations) was discussed in Sec. F.

## *8.2 Restrictions*

Most issues of Java-like languages and shared representations are analyzed in detail in [18].

- Module import acyclicity: This is largely a technical requirement to simplify the presentation. The fundamental requirement is model field dependency acyclicity — that can be enforced modularly. A simple and practical way to achieve this is to require that model field dependency is acyclic by type (meaning the dependency relation for model field types is acyclic) however this disallows some existing programming patterns such as alternating lists. A more general way is to require local acyclicity (among model fields of ther same type) and require acyclicity whenever an object is abstracted (e.g. $this$ on return from function) as a proof obligation. This requires more verification work but allows more programs. An issue with cyclic import is that $this$ can be passed (directly or indirectly) as an argument, which means any knowledge of the current component (including local variables pointing into it) is lost upon return unless it is passed as immutable (even if the specification does not specify any modification to its model fields).

- One header per component: This is a serious restriction as it disallows e.g. iterators. We believe it can be removed if all headers of the component (e.g. List and its Iterators) are defined in the same module and verified with "private" or at least "package" knowledge of each other, and each has to have a pivot to each other (which means pivots and representation functions in general are no longer only reference reachable heap as e.g. an Iterator does not have to have a pointer to a List's header — it may only point to a Node). We believe it is practical to allow multiple headers per component with the above mentioned restrictions and some additional modifications to the formulation and leave it as future work. The subject of package access is discussed in detail in [18].

- Local frame for pivots: This is an entirely technical requirement to simplify presentation — as long as dependencies are acyclic pivots can be defined as any other representation function. However we could not find a real use for non-local pivots.

- One to one concrete-model component graph: This enforces a bounded number of pivots and identity of the concrete and abstract component graph. We considered simple extensions to allow our system to support sets, sequences or trees of pivots (e.g. for a List with all members being outgoing pointers rather than the objects themselves we could use $head.next * .element$ as a representation function). We can allow the model component graph to be larger than the concrete one as mentioned earlier for iterators. We can allow the model component graph not to include a pivot for pointers to immutable subcomponents. Another extensions with a large practical value is to enforce the equivalence of concrete and model component graphs on component pack ($this$ on return from function) rather than by construction of the type as done in this thesis — this effectively means adding this equivalence to the class invariant.

- Subtyping: Subtyping is problematic because an upcasts loses specification information. We assume here a standard model of subtyping where the subtype may have additional constructs (concrete functions and specification model fields (the extended state)) and stricter specification for constructs (stricter pre-post for functions). Stricter specification for model fields, which means *less* dependencies and stricter model functions (more deterministic). The problem is that these extra model fields are lost on upcast (e.g. when calling a function that receives a supertype as a parameter) and hence have to be reconstructed on return. We believe that with some restrictions on subtyping and modifications to the call rule, our system can support subtyping, especially: Only stricter (as above) subtypes are allowed (which is a standard requirement), no downcasts are allowed, only new functions are allowed to modify new model fields and the extension of dependencies is limited — the whole topic is discussed in detail in [18] Multiple subtyping, even with methods inherited through more than one path, should not pose additional problems if cross-casts are not allowed, however cross-casts require the handling of downcasts which is discussed later.

- Inheritance: Inheritance in a modular environment requires the descendant class to to have a "protected" specification that is stronger than the "public" specification given to clients, in addition to "protected" functions. We believe the protected interface for a class is a kind of sub-type of the public interface (as most up/down casts are done in the same fashion) and hence the requirements for for subtyping should be sufficient for inheritance under the assumption there are no non-private fields. Multiple inheritance should pose no additional difficulties other than those added by multiple subtyping.

- Genericity: Genericity is in general orthogonal to our work (the running example could be given as a generic Map/Wrapper). The areas where genericity interacts with subtyping (e.g. as in Java 1.5 onwards) needs special care, but we believe that once subtyping is supported genericity is supported as well.

- Reflection: We do not see a way in which reflection can be readily supported as it bypasses several fundamental language mechanism.

- Casts: Casts should be seperated to several kinds, largely following C++:
  - Down/crosscasts: these are problematic because a down cast both requires the caster to reconstruct somehow the extended specification state and allows the downcaster to modify the extended specification (and concrete) state in ways it cannot specify (as the extended specification state is not in the scope of the upcasted type). A worst case handling of this is to "havoc" extended state whenever a downcast is performed (including return from a function with upcasted arguments). A better handling might be to allow the downcaster to include the downcasted type in the specification with a typecase kind specification (e.g. if x is of type T then . . .).
  - Unrestricted casts: Casts in the style of C++'s reinterpret_cast cannot be included in most (if not all) modular verification systems as they can be used to produce pointers to anywhere in memory (other components, unallocated space, OS etc) of any type. A restricted form (e.g. pointer $\rightarrow$ int) may be allowed. Or we can formulate their semantics in such a manner that only certain limited uses of this kind of cast allow us to prove anything about a program.
  - Upcasts, promotions etc: C++'s static_cast is generally safe to use in our system (once subtyping is)
  - const_cast: If we support immutable types (which should be straightforward) the immutable version of a type is in a sense a super-type of the mutable version — hence removing immutability is similar to downcasts.
- Concurrency: We have not done any research in adding concurrency to our system.
- Exceptions: We have not considered exceptions in our system but we believe they can be integrated in a manner similar to [5]

# 9. CONCLUSIONS

We presented a modular analysis for programs with shared heap-allocated data structures. We showed the realizability of our system by developing a proof-of-concept modular analyzer. We believe that the demonstrated ability to automatically analyze programs with inter-component sharing is an important step forward and that the idea of aggregate abstractions and aggregate model functions is of more general applicability (e.g., it can be used in manual verification) as our work is mostly independent of the specification language and of the shape abstraction.
We simplified the verification problem by forbidding callbacks and subtyping. We believe our work can be extended to handle these notions using additional user-provided specification. In particular, it seems that we can allow for a cyclic module dependencies, as long as the model functions and the representation functions are well-defined (i.e., not-cyclicly defined). We regard these extensions as interesting subjects for future work.

# BIBLIOGRAPHY

[1] M. Barnett and D. A. Naumann. Friends need a bit more: Maintaining invariants over shared state. In *LNCS*, number 3125 in LNCS, 2004.

[2] G. Bierman and M. Parkinson. Separation logic and abstractions. In *POPL*, 2005.

[3] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixed points. In *POPL*, 1977.

[4] P. Cousot and R. Cousot. Modular static program analysis, invited paper. In *CC*, 2002.

[5] W. Dietl and P. Müller. Exceptions in ownership type systems.

[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., 1995.

[7] C. A. R. Hoare. Proof of correctness of data representations. *Acta Inf.*, 1:271–281, 1972.

[8] G. Kahn. Natural semantics. In *4th Annual Symposium on Theoretical Aspects of Computer Sciences on STACS 87*, pages 22–39. Springer-Verlag, 1987.

[9] I. T. Kassios. *A Theory of Object Oriented Refinement*. PhD thesis, University of Toronto, 2006.

[10] J. Knoop and B. Steffen. The interprocedural coincidence theorem. In *Int. Conf. on Comp. Construct.*, 1992.

[11] P. Lam, V. Kuncak, and M. Rinard. Hob: A tool for verifying data structure consistency. In *CC (tool demo)*, 2005.

[12] ”Gary T leavens et al”. ”jml: notations and tools supporting detailed design in java”. *”OOPSLA ’00 Companion”*, pages ”105–106”, 2000.

[13] K. R. M. Leino and Peter Müller. A verification methodology for model fields. In *ESOP*, 2006.

[14] K. R. M. Leino and W. Schulte. Using history invariants to verify observers. In *ESOP*, 2007.

[15] T. Lev-Ami and M. Sagiv. Tvla: A framework for kleene based static analysis. In *SAS’00, Static Analysis Symposium*. Springer, 2000.

[16] F. Logozzo. Automatic inference of class invariants. In *VMCAI*, 2004.

[17] P. Müller. *Modular Specification and Verification of Object-Oriented Programs*. PhD thesis, FernUniversität Hagen, 2001.

[18] Peter Muller. Modular specification and verification of object oriented programs. In *LNCS*, number 2262 in LNCS, 2002.

[19] P. O’Hearn, H. Yang, and J. Reynolds. Separation and information hiding. In *POPL*, 2004.

[20] G. D. Plotkin. A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19, University of Aarhus, 1981.

[21] J. Reynolds. Separation logic: a logic for shared mutable data structures. In *LICS*, 2002.

[22] N. Rinetzky, J. Bauer, T. Reps, M. Sagiv, and R. Wilhelm. A semantics for procedure local heaps and its abstractions. In *Symp. on Princ. of Prog. Lang.*, 2005.

[23] N. Rinetzky, A. Poetzsch-Heffter, G. Ramalingam, M. Sagiv, and E. Yahav. Modular shape analysis for dynamically encapsulated programs. In *ESOP*, 2007.

[24] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems*, 2002.

[25] Thomas Wies, Viktor Kuncak, Patrick Lam, Andreas Podelski, and Martin Rinard. Field constraint analysis. In *VMCAI*, 2006.

[26] Greta Yorsh, Alexey Skidanov, Thomas Reps, and Mooly Sagiv. Automatic assume/guarantee reasoning for heap-manupilating programs. In *AIOOL: 1st International Workshop on Abstract Interpretation of Object-Oriented Languages*, 2005.

# APPENDIX

# Overview.

The formal part of our work is presented in the following appendices:
The first appendix presents the language and semantics.
The second appendix states the main theorems.
The third appendix details the requirements on the program and the proof obligations.
The fourth appendix gives the auxiliary definitions.
The fifth appendix includes proofs for all the theorems.
The sixth appendix shows an example specification language which is practical and in which the proof obligations are easy to show.
The seventh appendix gives all the details of the running example.
The top-down order of presentation necessitates some forward references — most refer to the fourth appendix.

# A. LANGUAGE AND SEMANTICS

## *A.1 Language*

### *A.1.1 Syntactic Domains*

The syntactic domains (Fig A.1) are standard.

| | | |
|---|---|---|
| $f$ | $\in \mathit{FId}$ | field id |
| $x, y$ | $\in \mathit{VarId}$ | variable id |
| $T$ | $\in \mathit{TId}$ | type id |
| $M$ | $\in \mathit{MId}$ | module id |
| $mf$ | $\in \mathit{MFId}$ | model field id |

*Fig. A.1:* Syntactic Domains — also used as semantic domains

### *A.1.2 Programming Language*

The language (Fig A.2) is a standard Java-like language with classes, fields, constructors and packages (modules) and without the more advanced features:

- Subtyping
- Inheritance
- Genericity
- Reflection
- Casts
- Concurrency

- Exceptions

Our language is typed in the style of Java without subtyping and inheritance. We do not give details of typing as it is standard - every local variable, function argument and class field has a declared type (reference or primitive) and type identity is checked on assignment. As there is no subtyping this is simple and standard. We use the terms private and public for the main (client visible) and auxiliary (client invisible) types of a module, respectively (as is common in Java).

```
program       =  ⟨module⟩∗
module        =  MId
                 ⟨public − class⟩
                 ⟨class⟩∗
public-class  =  ⟨class⟩
class         =  TId
                 (FId, TId)∗
                 ⟨method⟩∗

method        =  (VarId, TId)∗
                 ⟨statement⟩

statement   =  skip
               y = x
               y = x.f
               y.f = x
               y = new⟨T⟩
               ⟨statement⟩;⟨statement⟩
               let y=⟨exp⟩ in ⟨statement⟩
               if ⟨exp⟩ then ⟨statement⟩ else ⟨statement⟩
               while ⟨exp⟩ do ⟨statement⟩
               call  f( VarId∗)
exp         =  . . .

Where:
  x, y are variable ids
  T is a type id
```

*Fig. A.2:* The programming language.

### *A.1.3 Specification Language*

We do not give a specification language syntax but rather show it in semantic terms. The specification language (Fig A.3) includes the following constructs:

- For each class:
    - A set of typed model (specification) fields with, for each model field:
        - A *representation function* that defines the model field in terms of its containing component and abstract subcomponents. e.g. AVLMap.v: a function that maps the heap of an AVL map to a mathematical partial function
        - A *model function* that defines the model field in terms of its abstract component and abstract subcomponents. e.g. Keyset.keys: a function that takes a model (abstract) heap containing a Keyset wrapper wrapping a map and mapping it to the set of keys of the map
        - A set of *dependencies*: model fields in component and subcomponents upon which this model field's value depends. e.g. dependencies for AVLMap.map is an empty set while dependencies for Keyset.keys is Keyset.map and Keyset.map.v

      We distinguish between two kinds of model fields — normal and pivot:

      *normal* the range of the representation and model functions cannot include locations (but can include null)

      *pivot* the range of the representation and model functions includes only locations and null

- For every function:
    - A functional specification in terms of the model local heaps. Our proof obligations, admissibility rules and call rule ensure that the model local heap is an abstraction of the concrete local heap and that giving the specification for the local heap is sufficient. e.g. the specification for AVLMap.insert would include pairs of model heaps where the first and second heaps in each pair contain only an AVLMap, no Keysets are present but the caller may have any number of Keysets wrapping the AVLMap

| | | |
|---|---|---|
| class-spec | $=$ | model-field$*$ |
| method-spec | $=$ | pre-post-spec |
| model-field | $=$ | $MFId$ |
| | | representation-function |
| | | model-function |
| | | dependencies |

*Fig. A.3:* The specification language.

## A.2 Semantic Domains

### A.2.1 Standard Domains

The primitive semantic domains (Fig A.4) are standard - with domains also for specification constructs. The compound semantic domains used (Fig A.5) are also rather standard, with 3 flavours for concrete, hybrid and model heaps.

| | | |
|---|---|---|
| $l$ | $\in Loc$ | location |
| $null$ | $\in \{null\}$ | null |
| | $\in AtomicVal$ | atomic value |
| | $\in MAtomicVal$ | atomic model value |

*Fig. A.4:* Primitive semantic domains

| | | |
|---|---|---|
| $Env$ | $= VarId \rightharpoonup Val$ | variable environment |
| $Val$ | $= AtomicVal \cup Loc \cup \{null\}$ | value |
| $Heap$ | $= Loc \times FId \rightharpoonup Val$ | heap |
| $State$ | $= Env \times Heap$ | state |
| $MVal$ | $= MAtomicVal \cup Loc \cup \{null\}$ | model value |
| $MHeap$ | $= Loc \times MFId \rightharpoonup MVal$ | model heap |
| $MState$ | $= Env \times MHeap$ | model state |
| $HVal$ | $= Val \cup MVal$ | hybrid value |
| $HFId$ | $= FId \cup MFId$ | hybrid field id |
| $HHeap$ | $= Loc \times HFId \rightharpoonup HVal$ | hybrid heap |
| $HState$ | $= Env \times HHeap$ | hybrid state |

*Fig. A.5:* Compound semantic domains

### A.2.2 Specification Domains

The specification domains are in the style of JML specifications and are shown in (Fig A.6).

Class Specification :

|  |  |  |
|---|---|---|
| $MFId \rightharpoonup (Loc \times HHeap \rightharpoonup MVal)$ | : $\mathrm{T}_{\mathrm{rf}}$ | representation function |
| $MFId \rightharpoonup (Loc \times MHeap \rightharpoonup P(MVal))$ | : $\mathrm{T}_{\mathrm{mf}}$ | public model function |
| $MFId \rightharpoonup P(MFPath)$ | : $\mathrm{T}_{\mathrm{dep}}$ | public dependencies |

Method Specification :

|  |  |  |
|---|---|---|
| $MState \rightharpoonup P(MState)$ | : $\mathrm{f}_{\mathrm{spec}}$ | pre-post specification |

*Fig. A.6:* The specification language.

## A.3 LHS — Local Heap Standard Semantics

In this section we describe the standard, store based,local heap semantics.

### A.3.1 Intra Module Semantics

The intra module semantics (Fig A.7) are the basic semantics for Java statements (and equivalents).
We do not relate to the free list — as its use is standard.

$$
\begin{array}{llcll}
\text{skip} & : \langle \mathbf{skip}, (\varepsilon, \mathrm{h})\rangle & \rightsquigarrow & (\varepsilon, h) & \\
\text{assignment} & : \langle \mathbf{y} = \mathbf{x}, (\varepsilon, \mathrm{h})\rangle & \rightsquigarrow & (\varepsilon[y \mapsto \varepsilon(x)], h) & \mathrm{x}, \mathrm{y} \in \mathrm{dom}(\varepsilon) \\
\text{field read} & : \langle \mathbf{y} = \mathbf{x}.\mathbf{f}, (\varepsilon, \mathrm{h})\rangle & \rightsquigarrow & (\varepsilon[y \mapsto h(\varepsilon(x), f)], h) & \mathrm{x}, \mathrm{y} \in \mathrm{dom}(\varepsilon) \\
 & & & & (\varepsilon(\mathrm{x}), \mathrm{f}) \in \mathrm{dom}(\mathrm{h}) \\
 & & & & \mathrm{f} \in \mathrm{M_{fs}}(\mathrm{current}) \\
 & & & & \mathrm{y} = \mathrm{this} \vee \mathrm{private}(\varepsilon(\mathrm{x}), \mathrm{h}) \\
\text{field write} & : \langle \mathbf{y}.\mathbf{f} = \mathbf{x}, (\varepsilon, \mathrm{h})\rangle & \rightsquigarrow & (\varepsilon, h[(y, f) \mapsto \varepsilon(x)]) & \mathrm{x}, \mathrm{y} \in \mathrm{dom}(\varepsilon) \\
 & & & & (\varepsilon(\mathrm{y}), \mathrm{f}) \in \mathrm{dom}(\mathrm{h}) \\
 & & & & \varepsilon(\mathrm{x}) \in \mathrm{typedom}(\mathrm{h}, \mathrm{y}, \mathrm{f})) \\
 & & & & \mathrm{f} \in \mathrm{M_{fs}}(\mathrm{current}) \\
 & & & & \mathrm{y} = \mathrm{this} \vee \mathrm{private}(\varepsilon(\mathrm{x}), \mathrm{h}) \\
\text{allocation} & : \langle \mathbf{y} = \mathbf{new}\langle \mathbf{T}\rangle, (\varepsilon, \mathrm{h})\rangle & \rightsquigarrow & (\varepsilon[y \mapsto p], h[(\{p\} \times T_{fs}) \mapsto null]) & \mathrm{y} = \mathrm{this} \\
 & & & & \mathrm{p} \in \mathrm{free} \\
 & & & & \mathrm{mdl}(\mathrm{T}) = \mathrm{current}
\end{array}
$$

$$
\begin{array}{lll}
\text{composition} & : \dfrac{\langle \mathbf{S_1}, \sigma\rangle \rightsquigarrow \sigma'' \quad \langle \mathbf{S_2}, \sigma''\rangle \rightsquigarrow \sigma'}{\langle \mathbf{S_1};\mathbf{S_2}, \sigma\rangle \rightsquigarrow \sigma'} & \\
\text{let} & : \dfrac{\langle \mathbf{S_1}, (\varepsilon[\mathrm{y} \mapsto \varepsilon(\mathrm{e})], \mathrm{h})\rangle \rightsquigarrow (\varepsilon'[\mathrm{y} \mapsto \_], \mathrm{h}')}{\langle \mathbf{let\ y{=}e\ in\ S_1}, (\varepsilon, \mathrm{h})\rangle \rightsquigarrow (\varepsilon', \mathrm{h}')} & \mathrm{y} \notin \mathrm{dom}(\varepsilon) \\
\text{if}_{\text{true}} & : \dfrac{\langle \mathbf{S_1}, \sigma\rangle \rightsquigarrow \sigma'}{\langle \mathbf{if\ e\ then\ S_1\ else\ S_2}, \sigma\rangle \rightsquigarrow \sigma'}\, e(\sigma) & \\
\text{if}_{\text{false}} & : \dfrac{\langle \mathbf{S_2}, \sigma\rangle \rightsquigarrow \sigma'}{\langle \mathbf{if\ e\ then\ S_1\ else\ S_2}, \sigma\rangle \rightsquigarrow \sigma'}\, \neg e(\sigma) & \\
\text{while}_{\text{true}} & : \dfrac{\langle \mathrm{S}, \sigma\rangle \rightsquigarrow \sigma'' \quad \langle \mathbf{while\ e\ do\ S}, \sigma''\rangle \rightsquigarrow \sigma'}{\langle \mathbf{while\ e\ do\ S}, \sigma\rangle \rightsquigarrow \sigma'}\, e(\sigma) & \\
\text{while}_{\text{false}} & : \dfrac{}{\langle \mathbf{while\ e\ do\ S}, \sigma\rangle \rightsquigarrow \sigma}\, \neg e(\sigma) &
\end{array}
$$

*Fig. A.7:* Intra-module semantics - current is the current module and typedom is the set of values of the type (not shown here as it is standard)

### A.3.2 Inter Module Semantics

The call rule (Fig A.8) describes the large-step call in 3 steps —

1. Carve out the callee's local heap (using reachability restriction)

2. Execute the function's body

3. Embed the local-heap post state in the caller's pre-state by using an overriding union

Reachability restriction means using the heap reachable from the arguments, Overriding union is standard - both are described in Sec. D.1

$$\mathrm{call_{standard}} \quad : \frac{\langle \mathbf{f_{body}}, \sigma_{\mathrm{e}} \rangle \rightsquigarrow_s \sigma_x}{\langle \mathbf{f}(\overline{\mathbf{v}}), \sigma_{\mathrm{c}} \rangle \rightsquigarrow_s \sigma_r} \overline{\mathrm{v}} \in \varepsilon_{\mathrm{c}}$$

$Where:$

$$\begin{array}{lll}
\sigma_{\mathrm{c}} = \langle \varepsilon_{\mathrm{c}}, \mathrm{h_c} \rangle & & \\
\sigma_{\mathrm{x}} = \langle \varepsilon_{\mathrm{x}}, \mathrm{h_x} \rangle & & \\
\sigma_{\mathrm{e}} = \langle \varepsilon_{\mathrm{e}}, \mathrm{h_e} \rangle : & & \\
 & \varepsilon_{\mathrm{e}} & = \varepsilon_{\mathrm{c}}|_{\overline{\mathrm{v}}}[\overline{\mathrm{a}}|\overline{\mathrm{v}}] \\
 & \mathrm{h_e} & = \mathrm{h_c}||_{\varepsilon_{\mathrm{c}}(\overline{\mathrm{v}})} \\
\sigma_{\mathrm{r}} = \langle \varepsilon_{\mathrm{r}}, \mathrm{h_r} \rangle : & & \\
 & \varepsilon_{\mathrm{r}} & = \varepsilon_{\mathrm{c}}[\varepsilon_{\mathrm{x}}[\overline{\mathrm{v}}|\overline{\mathrm{a}}]] \\
 & \mathrm{h_r} & = \mathrm{h_c}[\mathrm{h_x}]
\end{array}$$

$Where:$

| | |
|---|---|
| $\sigma_{\mathrm{c}}$ | : is the call state |
| $\sigma_{\mathrm{e}}$ | : is the entry state |
| $\sigma_{\mathrm{x}}$ | : is the exit state |
| $\sigma_{\mathrm{r}}$ | : is the return state |
| $\overline{\mathrm{a}}$ | : $\mathrm{f_{args}}$ are the formal arguments for f |
| $\overline{\mathrm{v}}$ | : are the actual arguments for the call |

*Fig. A.8:* LHS call semantics

## *A.4 LHMS — Local Heap Modular Semantics*

### *A.4.1 Intra module semantics*

The intra module semantics are just like in the standard semantics - as we are only allowed to read and write fields of the current component the semantics works for both hybrid and concrete heaps.

### *A.4.2 Inter module semantics*

The modular call rule is described in Fig A.9. The call rule is similar to the standard with several differences resulting from our usage of specification. At the third stage we use the embed function which embeds the callee's post-state local heap into the callers pre-state heap. This is performed by first using an overriding union following the standard semantics and then propagating changes into affected model fields recursively (through dependencies), as shown in Fig D.8 - the function is defined precisely in Sec. D.1. Since our specification is provided in an abstract way using sets of states, the precondition is checked semantically in the side-condition of the rule.

$$\mathrm{call}_{\mathrm{modular}} \quad : \frac{\langle \mathbf{f_{spec}}, \sigma_{\mathrm{se}}\rangle \rightsquigarrow \sigma_{sx}}{\langle \mathbf{f}(\overline{\mathbf{v}}), \sigma_{\mathrm{mc}}\rangle \rightsquigarrow_m \sigma_{mr}} \sigma_{\mathrm{se}} \in \mathrm{f}_{\mathrm{pre}}$$

$$\begin{array}{lll}
\multicolumn{3}{l}{where:} \\
\sigma_{\mathrm{c}} = \langle \varepsilon_{\mathrm{mc}}, \mathrm{h}_{\mathrm{mc}} \rangle & & \\
\sigma_{\mathrm{x}} = \langle \varepsilon_{\mathrm{mx}}, \mathrm{h}_{\mathrm{mx}} \rangle & & \\
\sigma_{\mathrm{se}} = \langle \varepsilon_{\mathrm{se}}, \mathrm{h}_{\mathrm{se}} \rangle : & & \\
& \varepsilon_{\mathrm{se}} & = \varepsilon_{\mathrm{mc}}|_{\overline{\mathrm{v}}}[\overline{\mathrm{a}}|\overline{\mathrm{v}}] \\
& \mathrm{h}_{\mathrm{se}} & = \mathrm{h}_{\mathrm{mc}}||_{\varepsilon_{\mathrm{mc}}(\overline{\mathrm{v}})} \\
\sigma_{\mathrm{mr}} = \langle \varepsilon_{\mathrm{mr}}, \mathrm{h}_{\mathrm{mr}} \rangle : & & \\
& \varepsilon_{\mathrm{mr}} & = \varepsilon_{\mathrm{mc}}[\varepsilon_{\mathrm{sx}}[\overline{\mathrm{v}}|\overline{\mathrm{a}}]] \\
& \mathrm{h}_{\mathrm{mr}} & \in \mathrm{embed}(\mathrm{h}_{\mathrm{mc}}, \mathrm{h}_{\mathrm{sx}}) \\
\hline
\multicolumn{3}{l}{Where:} \\
\sigma_{\mathrm{mc}} & \multicolumn{2}{l}{\text{: is the (hybrid) call state}} \\
\sigma_{\mathrm{mr}} & \multicolumn{2}{l}{\text{: is the (hybrid) return state}} \\
\sigma_{\mathrm{se}} & \multicolumn{2}{l}{\text{: is the (abstract) specification entry state}} \\
\sigma_{\mathrm{sx}} & \multicolumn{2}{l}{\text{: is the (abstract) specification exit state}}
\end{array}$$

*Fig. A.9:* LHMS — Local Heap Modular Semantics — call

# B. THEOREMS

## *B.1 Soundness*

**Theorem B.1.1.** *The soundness theorem:*
*Given*

- *a program* p *for which all proof obligations are satisfied*
- *A function* f *of the main module of* p
- *An admissible state $\sigma$ of* p
- *A state $\sigma$' of* p *s.t.* $\langle \mathbf{f}, \sigma \rangle \rightsquigarrow_s \sigma'$

*We show that* $\langle \mathbf{f}, \mathrm{hyb}(\sigma) \rangle \rightsquigarrow_m \mathrm{hyb}(\sigma')$*. That is, each concrete execution of a valid program on an admissible state is contained in the abstract execution of that program on the abstraction of the state.*
*The* hyb *function maps a concrete heap to the equivalent hybrid heap. It is described in Fig D.7.*

## *B.2 The hyb and embed functions*

The main property of the embed function is that embed is an abstract overapproximation of the standard embedding, that is, using embed on abstract heaps is conservative with respect to using overriding union on the concrete heaps.

**Lemma B.2.1.** *If*

- $\sigma_c$ *is an admissible concrete state*
- $\sigma_x$ *is an admissible concrete state*
- $\forall l \in ldom(h_x) \cap ldom(h_c) : type(l, h_c) = type(l, h_x)$

- $t \in ldom(h_c) \setminus ldom(h_x)$

*then* $hyb(h_c[h_x], t) \in embed(hyb(h_c, t), hyb(h_x, t))$.

**Lemma B.2.2.** *Embed preserves the local heap.*
*If*

- $h_c$ *is an admissible concrete heap*
- $h_x$ *is an admissible concrete heap*
- $\forall l \in ldom(h_x) \cap ldom(h_c) : type(l, h_c) = type(l, h_x)$

*then* $hyb(h_c, t)[abs(h_x)] =_{dom(abs(h_x))} hyb(h_c[h_x], t)$.

**Lemma B.2.3.** *Embed preserves the unmodified lmfs.*
*If*

- $h_c$ *is an admissible concrete heap*
- $h_x$ *is an admissible concrete heap*
- $\forall l \in ldom(h_x) \cap ldom(h_c) : type(l, h_c) = type(l, h_x)$
- $h_r = hyb(h_c[h_x], t)$
- $P = \{(l, mf) \in dom(h_r) : rsframe(mf)(h_r, l) \cap mod(h_c, h_x) = \emptyset\}$

*then* $hyb(h_c, t) =_P hyb(h_c[h_x], t)$

**Lemma B.2.4.** *The hyb function preserves admissibility.*
*If*

- $\sigma$ *is an admissible concrete state*

*then* $hyb(\sigma)$ *is an admissible hybrid state.*

## *B.3 Properties of LHS*

The semantics is relatively standard.
The main properties of the semantics are:

**Lemma B.3.1.** *The semantics preserves admissiblity of heaps:*
*If*

- $\sigma$ *is an admissible concrete state*
- $S$ *is a statement of the semantics*
- $\langle \mathbf{S}, \sigma \rangle \rightsquigarrow_s \sigma'$

*Then* $\sigma'$ *is admissible.*

**Lemma B.3.2.** *The intra-module semantics only modifies the current component:*
*If*

- $\sigma$ *is an admissible concrete state*
- $S$ *is an intra-module statement of the semantics*
- $\langle \mathbf{S}, \sigma \rangle \rightsquigarrow_s \sigma'$

*Then* $\sigma =_{comp(h,\varepsilon(this))^C} \sigma'$.

**Lemma B.3.3.** *The intra-module semantics is unaffected by components other than the current component:*
*If*

- $\sigma_1, \sigma_2$ *are admissible concrete states*
- $S$ *is an intra-module statement of the semantics*
- $\sigma_1 =_{com(h,\varepsilon(this))} \sigma_2$.
- $\langle \mathbf{S}, \sigma_1 \rangle \rightsquigarrow_s \sigma_1'$

*Then exists* $\sigma_2'$ *s.t.* $\langle \mathbf{S}, \sigma_2 \rangle \rightsquigarrow_s \sigma_2'$ *and* $\sigma_1' =_{comp(h,\varepsilon(this))} \sigma_2'$.

## *B.4 Properties of LHMS*

The main properties of the semantics are:

**Lemma B.4.1.** *The semantics preserves admissibility of heaps: If*

- $\sigma$ *is an admissible hybrid state*
- $S$ *is a statement of the semantics*
- $\langle \mathbf{S}, \sigma \rangle \rightsquigarrow_s \sigma'$

*Then* $\sigma'$ *is admissible.*

**Lemma B.4.2.** *The intra-module semantics only modifies the current component: If*

- $\sigma$ *is an admissible hybrid state*
- $S$ *is an intra-module statement of the semantics*
- $\langle \mathbf{S}, \sigma \rangle \rightsquigarrow_s \sigma'$

*Then* $\sigma =_{com(h,\varepsilon(this))^C} \sigma'$*.*

**Lemma B.4.3.** *The intra-module semantics is unaffected by components other than the current component: If*

- $\sigma_1, \sigma_2$ *are admissible hybrid states*
- $S$ *is an intra-module statement of the semantics*
- $\sigma_1 =_{com(h,\varepsilon(this))} \sigma_2$.
- $\langle \mathbf{S}, \sigma_1 \rangle \rightsquigarrow_s \sigma_1'$

*Then exists* $\sigma_2'$ *s.t.* $\langle \mathbf{S}, \sigma_2 \rangle \rightsquigarrow_s \sigma_2'$ *and* $\sigma_1' =_{com(h,\varepsilon(this))} \sigma_2'$*.*

## *B.5 Global invariants*

**Lemma B.5.1.** *Type safety — all transitions preserve type safety*

**Lemma B.5.2.** *Admissibility — all transitions preserve state admissibility*

# C. REQUIREMENTS

## *C.1 Proof Obligations*

The proof obligations are modular restrictions on programs and modular specifications that ensure that the whole program conforms to the specification (as defined later). Most of the proof obligations that we provide, are standard and are given here for completeness. All proof obligation can be verified modularly independent of the implementation of other modules. We present a syntax of a specification language and algorithms for syntactically checking the new obligations.

### *C.1.1 Semantic domains*

- All atomic semantic domains are disjoint in pairs
- All atomic semantic domains are non-empty
- The domain $\mathit{Val}$ include the boolean domain — $\{TRUE, FALSE\}$

### *C.1.2 Specification*

**Proof Obligation C.1.1.** *For each class, the representation and model functions are location independent*

**Proof Obligation C.1.2.** *For each class, the frame of each pivot is local (each location valued model field is a function of just the current component). This is used to ensure pivots cannot be modified indirectly and therefore lmfs that do not depend on a call's local heap at the call site does not depend on it at the return site (for frame rule)*

**Proof Obligation C.1.3.** *For each class, each pivot-representation-function is a projection (no cast to pointer etc). This is used to ensure an abstraction of a downward closed heap is also downward closed.*

**Proof Obligation C.1.4.** *For each class, for every model field, the specification frame is a superset of the frame : (in subcomponents) — meaning:*
*For each hybrid field path* $c.m \in frame(mf)$ *where c is the concrete part (within the component) and m is the model part (in subcomponents), there is a pivot p s.t.* $\forall h, l$ *s.t.* $(l, mf) \in dom(h) : T_{rf}(p)(l, h) = h(l, c)$ *for admissible h, and* $p.m \in T_{dep}(mf)$. *(* $frame(), T_{dep}() and t_{rf}()$ *are defined in Sec. D.1). This is used to make sure a caller, which sees only the abstract heaps of subcomponents, can estimate soundly the frame and hence (non)modification of each model field of each subcomponent.*

**Proof Obligation C.1.5.** *For each class, for every model field mf, for every other model field mf' which is a dependee of mf and is in the same component, mf' has no dependencies — meaning* $\forall mf' \in T_{dep}(mf) : T_{dep}(mf') = \emptyset$ *(a simplification to allow calculating the entire component in one stage - not a real restriction — only acyclicity of dependencies of model fields of same component is a real restriction)*

**Proof Obligation C.1.6.** *For each class, for every model field mf, the dependencies must be a superset(or equal) of the frame of the model function* $T_{mf}$*, meaning:*
$\forall h, h' : h =_{sframe(mf)(h,l)} h' \rightarrow T_{mf}(mf)(h, l) = T_{mf}(mf)(h', l)$.
*(* $sframe()$ *is defined in Sec. D.1).*
*This is used to ensure that the client of a component can determine the preservation of a model field using dependencies.*

**Proof Obligation C.1.7.** *For each class, every outgoing pointer has a corresponding pivot and vice versa. This is used to ensure the concrete and abstract component graphs are identical - therefore the abstract local heap is equivalent to the concrete local heap. Note that one pivot can represent several equal outgoing pointers and vice versa. This limits the number of outgoing pointers (degree of component graph) to be statically bounded - an extension wuth sets or sequencesof outgoing pointers is straightforward.*

### *C.1.3 Implementation*

The 2 most important proof obligations are:

**Proof Obligation C.1.8.** *Each function complies with its specification — given the definitions in Fig C.1: For each* $\sigma_{he}, \sigma_{hx}$ *s.t.* $\sigma_{se} \in pre_f \ \langle \mathbf{f_{body}}, \sigma_{\mathrm{he}} \rangle \rightsquigarrow_m \sigma_{hx}$ *implies* $\langle \mathbf{f_{spec}}, \sigma_{\mathrm{se}} \rangle \rightsquigarrow \sigma_{sx}$ *That is, for every legal hybrid pre-post pair there is a* *corresponding abstract pre-post pair — this is the standard Hoare proof obligation.*

**Proof Obligation C.1.9.** *The model function overapproximates the representation function:*
*For each admissible hybrid state $\sigma = (\varepsilon, h)$ where $l = \varepsilon(this)$,*
*($T_{rf}(mf)(l, h) \in T_{mf}(mf)(l, r_f(l, h))$)*

$$\mathrm{implementation}_{\mathrm{modular}} \quad : \frac{\langle \mathbf{f_{body}}, \sigma_{\mathrm{me}}\rangle \rightsquigarrow_{m} \sigma_{mx}}{\langle \mathbf{f_{spec}}, \sigma_{\mathrm{se}}\rangle \rightsquigarrow \sigma_{sx}} \sigma_{\mathrm{se}} \in \mathrm{f}_{\mathrm{pre}}$$

$$\begin{aligned}
&\mathit{where}: \\
&\sigma_{\mathrm{me}} = \langle \varepsilon_{\mathrm{me}}, \mathrm{h}_{\mathrm{me}} \rangle \\
&\sigma_{\mathrm{mx}} = \langle \varepsilon_{\mathrm{mx}}, \mathrm{h}_{\mathrm{mx}} \rangle \\
&\sigma_{\mathrm{se}} = \langle \varepsilon_{\mathrm{se}}, \mathrm{h}_{\mathrm{se}} \rangle : \\
&\qquad\qquad \varepsilon_{\mathrm{se}} \quad = \varepsilon_{\mathrm{me}} \\
&\qquad\qquad \mathrm{h}_{\mathrm{se}} \quad = \mathrm{r}_{\mathrm{f}}(\mathrm{h}_{\mathrm{me}}, \mathrm{t}) \\
&\sigma_{\mathrm{sx}} = \langle \varepsilon_{\mathrm{sx}}, \mathrm{h}_{\mathrm{sx}} \rangle : \\
&\qquad\qquad \varepsilon_{\mathrm{sx}} \quad = \varepsilon_{\mathrm{mx}} \\
&\qquad\qquad \mathrm{h}_{\mathrm{sx}} \quad = \mathrm{r}_{\mathrm{f}}(\mathrm{h}_{\mathrm{mx}}, \mathrm{t}) \\
&\mathrm{t} = \varepsilon_{\mathrm{se}}(\mathrm{this})
\end{aligned}$$

*Fig. C.1:* Modular implementation proof obligation

## C.2 Program Admissibility

We are only dealing with a subset of the possible programs that we term admissible programs. We define the admissibility constraints for each language construct. Most admissibility constraints are standard and can be checked syntactically or with a simple algorithm over the AST — for others we note how they can be checked.

### C.2.1 Module admissibility

The module declarations of a program are admissible if:

- Module import is acyclic. This is used to ensure dependencies are acyclic (representation functions well-defined)

### C.2.2 Type admissibility

The type declarations of a program are admissible if (all simple type checks):

- The set of fields of each class are mutually disjoint
- The set of classes of each module are mutually disjoint
- Each field is of a private type of the containing module or a public class of an imported module (note that this means that a component cannot point to another component of the same kind publicly - no public recursion only intra-component recursion)

### C.2.3 Method admissibility

The methods of a program are admissible if (all simple type checks):

- The first argument of each method is of the containing class (this)
- All other arguments are of public classes of the containing module or public classes of imported modules
- All local variables are of classes of the containing module or public classes of imported modules

### *C.2.4 Class Specification admissibility*

The class specifications of a program are admissible if:

- Every model-field has exactly one model and one representation function
- The model and representation functions:

  *Location independent* Would give the same result for 2 heaps that are isomorphic up to locations (but may depend on location equality). This can be ensured by contructing representation functions from atomic functions that are location independent

  (as done is Sec. F).

  *Definedness monotonous* (if defined on a heap then defined on every containing heap) This is used to ensure that definedness seen in the local heap is equivalent to that in the caller's heap. This can be ensured by defining them as functions over field paths constructed from monotone functions

  *Monotonous* This is a fundamental requirement to ensure that values calculated by a component (if defined on a heap return the same value for every containing heap) are valid when the component is embeded into a larger heap. This can be ensured by defining them as functions over field paths constructed from monotone functions

  *Depend only on heap reachable from l* (defined for heap h iff defined for the portion of h reachable from l) This is to make sure that the local heap is sufficient for abstracting the current component. This can be ensured by defining them as functions over field paths.

  Are either pivot (range is Location or null) or normal (no locations in range)

- The set of dependencies must be prefix closed. This is a technical requirement to simplify presentation. A simple way to ensure this is to take the prefix closure of the programmer's specification

### *C.2.5 Method Specification admissibility*

A method specification of a method in a program is admissible if:

- It references only model fields reachable from the arguments (local heap). This is necessary for specifications to be definable on the local heap only. This can be ensured easily by allowing the specification to only reference field paths starting at this and arguments.

- The preconditions include the requirement that *this* is not aliased in other arguments. This is necessary to ensure the current component is only modified through *this* - it cannot be referenced indirectly form other arguments because of the acyclicity of module import.

#### *C.2.6 Program Admissibility*

We only consider asmissible programs in this appendix. For our method to be practical, admissibility checking of programs should be simple, practical and modular. All checks are modular by definition, meaning they can be checked with knowledge only of the current module and interfaces of imported modules. We show a simple and practical specification language in Sec. F which gurantees by construction (syntax) all above admissibility constraints.

# D. AUXILIARY DEFINITIONS

## D.1 Auxiliary Definitions

### D.1.1 Declaration mapping

Fig D.1 defines a name and semantic domain for each syntactic constrcut. Fig D.2 defines some functions on these declarations.

The set of field ids of the class T:
$\mathrm{T_{fs}(T)}$: $\mathit{TId} \rightarrow \mathrm{P}(\mathit{HFId})$

The set of model field ids of the class T:
$\mathrm{T_{mfs}(T)}$: $\mathit{TId} \rightarrow \mathrm{P}(\mathit{MFId})$

The class to which the fieldId f belongs:
$\mathrm{type(f)}$: $\mathit{HFId} \rightarrow \mathit{TId}$

The declared class of the fieldId f:
$\mathrm{ftype(f)}$: $\mathit{HFId} \rightarrow \mathit{TId}$

The set of classes of the module m:
$\mathrm{types(M)}$: $\mathit{MId} \rightarrow \mathrm{P}(\mathit{TId})$

The module to which the class T belongs:
$\mathrm{mdl(T)}$: $\mathit{TId} \rightharpoonup \mathit{MId}$

The public class of module M:
$\mathrm{publictype(M)}$: $\mathit{MId} \rightharpoonup \mathit{TId}$

*Fig. D.1:* Auxiliary Definitions — Declarations

### *D.1.2 General definitions*

In order to define a formal semantics of the language we use several auxiliary definitions (Fig D.3):

*Field and Access Paths* We define field and access paths for all flavours of heaps (field paths start from a location (not included), access paths start from a local variable)

*Representation Function* The representation function maps, per model field, a hybrid heap and location to a model value

*Model Function* The model function maps, per model field, a model heap and location to a set of model values

*LF* A (location,fieldId) pair (lf) is the basic heap unit for concrete and hybrid heaps

*LMF* A (location,modelFieldId) pair (lmf) is the basic heap unit for model heaps

### *D.1.3 Auxiliary functions*

The auxiliary functions are shown in (Fig D.4)

- We use overriding unions for functions and sets of functions to embed subheaps into larger heaps.
- The set of locations occupied in the heap h is ldom(h) (dom(h) is a set of location, fieldid pairs)
- Heap reachability restriction is used to get the heap reachable from a set of locations — e.g. to get a function call's local heap
- Heap reachability is the standard reachability,signifying the existence of a field path between 2 locations, we use the same definition for concrete, hybrid and model heaps
- Given 2 heaps h,h', a location l and a set of field paths, we want to determine whether the field paths evaluate to the same values over the 2 heaps. starting at the location. This is used to check that the subheap relevant for a specific model field is isomorphic among 2 heaps

### *D.1.4 Component functions*

The functions describing the component subdivision of a concrete heap and the equivalent hybrid and model heap are given in Fig D.5 :

- A component is all the heap reachable from a location (which must be of a public type) through concrete fields of the same module.

  $M_{fs}(module(l, h))$ are the concrete field Ids of all types in the module of l in the heap h. Our proof obligations ensure that, for each heap that can occur in a program execution, each location belongs to at most one component — no component can have 2 or more in-ports. In the example, a Keyset component at location l consists of l.map. A map component at location l consists of $\{l.head, l.head.(right\|left)*, l.head.(right\|left)*.(k\|v)\}$

- The set of components (identified by location of head) in a heap h is headers(h)

- The set of outgoing pointers (direct subcomponents) of a component l in heap h is outgoing(h,l). This applies to both concrete, hybrid and model heaps

- The component graph of a heap has nodes for components and edges for pointers. Our restrictions ensure it is a DAG and is preserved by hyb and embed

### *D.1.5 Representation and model function definitions*

Auxiliary functions related to the representation and model functions are defined in Fig D.6

- The $r_f$ function flattens one component l in the heap h — erasing the entire concrete component and replacing it with a single model object=component — calculating model values for model fields from heap h which must include the component l as a concrete component and its subcomponents as model components.

- The frame of a model field is the minimal set of field paths which, when evaluate to the same values between 2 heaps, ensure the model field evaluates to the same value in the 2 heaps (starting at the same location). This is defined in the hybrid view of the heap. In the example, in the concrete view of the heap, for a given l pointing to an instance of Keyset, the frame for (l,keys) is $\{l.map[.head.(right\|left) * [.(k\|v)]]\}$ We also have a version for a specific location,heap — given in terms of lmfs.

- The specification frame of a model field is similar to the frame only defined in the model view and derived from specification (dependencies) rather than calculated from the function itself. This is used by clients of a component to determine whether a model field should be recalculated by checking if the recursive specification frame coincides with (potentially) modified lmfs in the local heap

### *D.1.6 The hybrid function*

The mapping between a concrete heap and the equivalent hybrid heap is defined by the $hyb$ function shown in (Fig D.7). The function recursively climbs up the component DAG, at each stage flattening (abstracting) each component at one level of the DAG. The arguments for the recursive function are:

*h* The current partially abstracted heap — initially the concrete heap

*D* The set of components already abstracted in h

*C* The set of locations that still need to be abstracted (in order to be able to do partial hybridization — e.g. exclude the current component )

*L* The current layer of components to be updated

At each stage we calculate the current layer L by taking the components for which all subcomponents have been abstracted, and then we flatten them using the $hyb_l$ function which erases the concrete components and replaces them with their model equivalents.
We also have a version for state that takes $\varepsilon(this)$ as the hybrid current object and versions for getting the model heap and state equivalents — with $null$ as current object.

#### *D.1.6.1 Properties of the hyb function*

**Lemma D.1.1.** *If*

- *$h$ is an admissible concrete heap*
- $l \in ldom(h)$
- $S \subseteq headers(h) \setminus \{l\}$
- $\forall c \in S : outgoing(c, h) \subseteq S$

*then* $hyb(h||_S, l) = hyb(h, l)||_S$

*Proof.* By induction on the recursion of hyb — We term $h_l$ h in iterations of $hyb(h||_S, l)$ and $h_r$ h interations of $hyb(h, l)$. as S is downward closed, in each iteration $h_r||_S =_D h_l$ and hence when $L = \emptyset$ the lemma holds. □

### *D.1.7 The embed function*

The embed function (Fig D.8) is used to calculate the post-state of a function call from the pre-state of the caller's heap and the post-state of the callee's (local) heap. The function propagates modifications to shared model fields recursively. The arguments for $embed_r$ are:

*H* The current set of partially embedded heaps — initially just the callers heap with the callee's heap overriden by the exit heap (shared abstractions not updated)

*D* The set of components already abstracted in h

*C* As in hyb

*M* the set of lmfs that is (potentially) different between the call and return states - initialized to the set modified in the callee's local heap and propagated using the specification frame

*L* The current layer of components to be abstracted

*U* The subset (as lmfs) of L that is actually recalculated — only lmfs for which there is an lmf in the specification frame which has been modified are recalculated (and marked as modified themselves)

The set of fieldIds of all the classes of module M:
$\mathrm{M_{fs}}(\mathrm{M})$: $\mathit{MId} \rightarrow \mathrm{P}(\mathit{HFId})$
$= \cup_{\mathrm{T}\in\mathrm{types}(\mathrm{M})} \mathrm{T_{fs}}(\mathrm{T})$

The set of all public types in the program:
publictypes: $\mathrm{P}(\mathit{TId})$
$= \cup_{\mathrm{M}\in\mathit{MId}} \mathrm{publictype}(\mathrm{M})$

The type of the location l in heap h (when defined):
$\mathrm{type}(\mathrm{l}, \mathrm{h})$: $\mathit{Loc} \times \mathit{HHeap} \rightharpoonup \mathit{TId}$
$$= \begin{cases} \mathrm{T} & : \{\mathrm{T}\} = \{\mathrm{type}(\mathrm{f}) : (\mathrm{l}, \mathrm{f}) \in \mathrm{dom}(\mathrm{h})\} \\ \text{undefined} & : \text{otherwise} \end{cases}$$

Is the object at location l in heap h of a private type:
$\mathrm{private}(\mathrm{l}, \mathrm{h})$: $\mathit{Loc} \times \mathit{HHeap}$
$= \mathrm{type}(\mathrm{l}, \mathrm{h}) \notin \mathrm{publictypes}$

The module of the type of the location l in heap h (when defined):
$\mathrm{mdl}(\mathrm{l}, \mathrm{h})$: $\mathit{Loc} \times \mathit{HHeap} \rightharpoonup \mathit{MId}$
$= \mathrm{mdl}(\mathrm{type}(\mathrm{l}, \mathrm{h}))$

The set of fieldIds of all the classes of module M:
$\mathrm{M_{fs}}(\mathrm{l}, \mathrm{h})$: $\mathit{Loc} \times \mathit{HHeap} \rightharpoonup \mathrm{P}(\mathit{HFId})$
$= \mathrm{M_{fs}}(\mathrm{mdl}(\mathrm{l}, \mathrm{h}))$

Preconditions of a function:
$\mathrm{f_{pre}}$: $\mathrm{P}(\mathit{HState})$
$= \mathrm{dom}(\mathrm{f_{spec}})$

*Fig. D.2:* Auxiliary Definitions — declaration functions

| | | | |
|---|---|---|---|
| $p$ | $\in FPath$ | $= \mathit{FId}*$ | field path |
| $p$ | $\in APath$ | $= \mathit{VarId}, FPath$ | access path |
| $p$ | $\in \mathit{MFPath}$ | $= \mathit{MFId}*$ | model field path |
| $p$ | $\in \mathit{MAPath}$ | $= \mathit{VarId}, \mathit{MFPath}$ | model access path |
| $m$ | $\in \mathit{MFunc}$ | $= \mathit{MFId} \rightharpoonup (\mathit{MHeap} \times \mathit{Loc} \rightarrow P(MVal))$ | model function |
| $p$ | $\in \mathit{HFPath}$ | $= \mathit{FId}*, \mathit{MFId}*$ | hybrid field path |
| $p$ | $\in \mathit{HAPath}$ | $= \mathit{VarId}, \mathit{HFPath}$ | hybrid access path |
| $r$ | $\in \mathit{RFunc}$ | $= \mathit{MFId} \rightharpoonup (\mathit{HHeap} \times \mathit{Loc} \rightharpoonup MVal)$ | rep function |
| $lf$ | $\in \mathit{LF}$ | $= \mathit{Loc} \times \mathit{HFId}$ | Location, fid pair |
| $lmf$ | $\in \mathit{LMF}$ | $= \mathit{Loc} \times \mathit{MFId}$ | Location, mfid pair |

*Fig. D.3:* Auxiliary Definitions

Overriding union for functions:
$\mathrm{f[g]}\colon (\mathrm{D} \rightharpoonup \mathrm{R}) \times (\mathrm{D} \rightharpoonup \mathrm{R}) \to (\mathrm{D} \rightharpoonup \mathrm{R})$
$= \{\mathrm{x} \mapsto \text{if } \mathrm{x} \in \mathrm{dom(g)} \text{ then } \mathrm{g(x)} \text{ else } \mathrm{f(x)}\}$

Extension for sets of functions:
$\mathrm{F[G]}\colon \mathrm{P}((\mathrm{D} \rightharpoonup \mathrm{R})) \times \mathrm{P}((\mathrm{D} \rightharpoonup \mathrm{R})) \to \mathrm{P}((\mathrm{D} \rightharpoonup \mathrm{R}))$
$= \{\mathrm{f[g]} : \mathrm{f} \in \mathrm{F} \wedge \mathrm{g} \in \mathrm{G}\}$

Locations in a heap:
$\mathrm{ldom(h)}\colon \mathit{HHeap} \to \mathrm{P}(\mathit{Loc})$
$= \{\mathrm{l} : \exists \mathrm{f} : (\mathrm{l}, \mathrm{f}) \in \mathrm{dom(h)}\}$

Heap reachability restriction:
$\mathrm{h}||_{\mathrm{S}}\colon \mathit{HHeap} \times \mathrm{P}(\mathit{Loc}) \to \mathit{HHeap}$
$= \mathrm{h}|_{\mathrm{reach(h,S)}}$

Reachable locations:
$\mathrm{reach(h, S)}\colon \mathit{HHeap} \times \mathrm{P}(\mathit{Loc}) \to \mathrm{P}(\mathit{Loc})$
$= \cup_{\mathrm{l} \in \mathrm{S}}\{\mathrm{l}' : \mathrm{reachable1_h} * (\mathrm{l}, \mathrm{l}')\}$

Directly reachable location:
$\mathrm{reachable1_h(l, l')}\colon [\mathit{HHeap}]\mathrm{P}(\mathit{Loc} \times \mathit{Loc})$
$= \exists \mathrm{f} \in \mathit{HFId} : \mathrm{h(l, f)} = \mathrm{l}'$

Heaps that agree on a set of paths from a location:
$\mathrm{h} =_{\mathrm{l,P}} \mathrm{h}'\colon \mathit{HHeap} \times \mathit{HHeap} \times \mathit{Loc} \times \mathrm{P}(\mathit{HFPath})$
$= \forall (\mathrm{p}) \in \mathrm{P} : \mathrm{h(l, p)} = \mathrm{h}'(\mathrm{l}, \mathrm{p})$

Heaps that agree on a set of lmfs:
$\mathrm{h} =_{\mathrm{S}} \mathrm{h}'\colon \mathit{HHeap} \times \mathit{HHeap} \times \mathrm{P}(\mathit{LMF})$
$= \forall ((\mathrm{l}, \mathrm{mf})) \in \mathrm{S} : \mathrm{h(l, mf)} = \mathrm{h}'(\mathrm{l}, \mathrm{mf})$

*Fig. D.4:* Auxiliary Functions

All locations belonging to the component whose header is at l:

$\mathrm{comp}(\mathrm{l}, \mathrm{h})$: $\mathit{Loc} \times \mathit{HHeap} \rightarrow \mathrm{P}(\mathit{Loc})$

$= \{\mathrm{l}' \in \mathrm{ldom}(\mathrm{h}) : \mathrm{mdl}(\mathrm{l}', \mathrm{h}) = \mathrm{mdl}(\mathrm{l}, \mathrm{h}) \wedge \exists \mathrm{p} \in \mathrm{M}_{\mathrm{fs}}(\mathrm{l}, \mathrm{h})* : \mathrm{h}(\mathrm{l}, \mathrm{p}) = \mathrm{l}'\}$

as lmfs:

$\mathrm{comp}_{\mathrm{lmf}}(\mathrm{l}, \mathrm{h})$: $\mathit{Loc} \times \mathit{HHeap} \rightarrow \mathrm{P}(\mathit{LMF})$

$= \{(\mathrm{l}', \mathrm{mf}) \in \mathrm{dom}(\mathrm{h}) : \mathrm{l}' \in \mathrm{comp}(\mathrm{l}, \mathrm{h})\}$

Headers of components in h:

$\mathrm{headers}(\mathrm{h})$: $\mathit{HHeap} \rightarrow \mathrm{P}(\mathit{Loc})$

$= \{\mathrm{l} \in \mathrm{ldom}(\mathrm{h}) : \mathrm{type}(\mathrm{l}) \in \mathrm{publictypes}\}$

A component's (concrete or model) outgoing pointers:

$\mathrm{outgoing}(\mathrm{l}, \mathrm{h})$: $\mathit{Loc} \times \mathit{HHeap} \rightarrow \mathrm{P}(\mathit{Loc})$

$= \{\mathrm{l}' \in \mathit{Loc} : \mathrm{l}' \in \mathrm{h}(\mathrm{com}(\mathrm{h}, \mathrm{l})) \setminus \mathrm{dom}(\mathrm{com}(\mathrm{h}, \mathrm{l}))$

The component graph of h:

$\mathrm{cg}(\mathrm{h})$: $\mathit{HHeap} \rightarrow (\mathrm{P}(\mathit{Loc}) \times \mathrm{P}(\mathit{Loc} \times \mathit{Loc}))$

$= (\mathrm{V}_{\mathrm{cg}(\mathrm{h})}, \mathrm{E}_{\mathrm{cg}(\mathrm{h})})$

$\mathrm{V}_{\mathrm{cg}(\mathrm{h})} = \mathrm{headers}(\mathrm{h})$

$\mathrm{E}_{\mathrm{cg}(\mathrm{h})} = \{(\mathrm{l}, \mathrm{l}') : \mathrm{l}, \mathrm{l}' \in \mathrm{headers}(\mathrm{h}) \wedge \mathrm{l}' \in \mathrm{outgoing}(\mathrm{h}, \mathrm{l})\})$

*Fig. D.5:* Auxiliary functions — components

Representation function of a module m — for the component l:
$\mathrm{r_f}(\mathrm{l}, \mathrm{h})$: $\mathit{Loc} \times \mathit{HHeap} \rightharpoonup \mathit{HHeap}$
$= \mathrm{h}|_{\mathrm{com(l)}^C} \cup \{(\mathrm{l}, \mathrm{mf}) \mapsto \mathrm{T_{rf}}(\mathrm{mf})(\mathrm{l}, \mathrm{h}) : \mathrm{mf} \in \mathrm{T_{mfs}}(\mathrm{type}(\mathrm{l}, \mathrm{h}))\}$

Extension for sets of locations L:
$\mathrm{r_f}(\mathrm{L}, \mathrm{h})$: $\mathrm{P}(\mathit{Loc}) \times \mathit{HHeap} \rightharpoonup \mathit{HHeap}$
$= \mathrm{h}|_{(\cup_{\mathrm{l} \in \mathrm{L}} \mathrm{com(l)})^C} \cup \{(\mathrm{l}, \mathrm{mf}) \mapsto \mathrm{T_{rf}}(\mathrm{mf})(\mathrm{l}, \mathrm{h}) : \mathrm{mf} \in \mathrm{T_{mfs}}(\mathrm{type}(\mathrm{l}, \mathrm{h})), \mathrm{l} \in \mathrm{L}\}$

For a hybrid state $\sigma$:
$\mathrm{r_f}((\varepsilon, \mathrm{h})))$: $\mathit{HState} \rightharpoonup \mathit{MState}$
$= (\varepsilon, \mathrm{r_f}(\varepsilon(\mathrm{this}), \mathrm{h}))$

The frame of a model field mf:
$\mathrm{frame}(\mathrm{mf})$: $\mathit{MFId} \to \mathrm{P}(\mathit{HFPath})$
$= \mathrm{pc}(\cap\{\mathrm{P} \subseteq \mathit{HFPath} : \forall \mathrm{h}, \mathrm{h}', \mathrm{l} : \mathrm{h} =_{\mathrm{l,P}} \mathrm{h}' \to \mathrm{T_{rf}}(\mathrm{mf})(\mathrm{h}, \mathrm{l}) = \mathrm{T_{rf}}(\mathrm{mf})(\mathrm{h}', \mathrm{l})\})$
(pc is prefix closure)
For a specific heap:
$\mathrm{frame}(\mathrm{mf})(\mathrm{l}, \mathrm{h})$: $\mathit{MFId} \times \mathit{Loc} \times \mathit{HHeap} \to \mathrm{P}(\mathit{Loc} \times \mathit{MFId})$
$= \{(\mathrm{h}(\mathrm{l}, \mathrm{p}), \mathrm{f}) : \mathrm{p.f} \in \mathrm{frame}(\mathrm{mf})\}$

The specification frame of a model field mf
$\mathrm{sframe}(\mathrm{mf})$: $\mathit{MFId} \to \mathrm{P}(\mathit{MFPath})$
$= \mathrm{T_{dep}}(\mathrm{mf})$
For a specific heap:
$\mathrm{sframe}(\mathrm{mf})(\mathrm{l}, \mathrm{h})$: $\mathit{MFId} \times \mathit{Loc} \times \mathit{MHeap} \to \mathrm{P}(\mathit{Loc} \times \mathit{MFId})$
$= \{(\mathrm{h}(\mathrm{l}, \mathrm{p}), \mathrm{f}) : \mathrm{p.f} \in \mathrm{sframe}(\mathrm{mf})\}$

The recursive specification frame of a model field mf
For a specific heap:
$\mathrm{rsframe}(\mathrm{mf})(\mathrm{l}, \mathrm{h})$: $\mathit{MFId} \times \mathit{Loc} \times \mathit{MHeap} \to \mathrm{P}(\mathit{Loc} \times \mathit{MFId})$
$= \mathrm{sframe}(\mathrm{mf})(\mathrm{l}, \mathrm{h}) \cup \{\mathrm{rsframe}(\mathrm{mf}')(\mathrm{l}', \mathrm{h}) : (\mathrm{l}', \mathrm{mf}') \in \mathrm{sframe}(\mathrm{mf})(\mathrm{l}, \mathrm{h})\}$

*Fig. D.6:* Auxiliary Functions — representation and model functions

The hybrid heap equivalent of the heap h (t is current)

$\mathrm{hyb}(\mathrm{h}, \mathrm{t})$: $\mathit{Heap} \times \mathit{Loc} \rightharpoonup \mathit{HHeap}$

$= \mathrm{hyb_r}(\mathrm{h}, \emptyset, \mathrm{headers}(\mathrm{h}) \setminus \{\mathrm{t}\})$

The recursive hybrid heap:

$\mathrm{hyb_r}(\mathrm{h}, \mathrm{D}, \mathrm{C})$: $\mathit{HHeap} \times \mathit{MHeap} \times \mathrm{P}(\mathit{Loc}) \times \mathrm{P}(\mathit{Loc}) \rightharpoonup \mathit{HHeap}$

$$= \begin{cases} \mathrm{h} & : \mathrm{L} = \emptyset \\ \mathrm{hyb_r}(\mathrm{hyb_l}(\mathrm{L}, \mathrm{h}), \mathrm{D} \cup \mathrm{L}, \mathrm{C} \setminus \mathrm{L}) & : \mathrm{L} \neq \emptyset \end{cases}$$

$\mathrm{Where:}$

$\mathrm{L} = \{\mathrm{l} \in \mathrm{C} : \mathrm{outgoing}(\mathrm{l}, \mathrm{h}) \subseteq \mathrm{D}\}$

Abstract the components L in heap h:

$\mathrm{hyb_l}(\mathrm{L}, \mathrm{h})$: $\mathrm{P}(\mathit{Loc}) \times \mathit{HHeap} \rightharpoonup \mathit{HHeap}$

$= \mathrm{h}|_{\mathrm{comp}(\mathrm{L},\mathrm{h})^{\mathrm{C}}}[\mathrm{r_f}(\mathrm{L}, \mathrm{h})]$

The hybrid state equivalent of the state $(h, \varepsilon)$

$\mathrm{hyb}((\mathrm{h}, \varepsilon))$: $\mathit{State} \rightharpoonup \mathit{HState}$

$= (\varepsilon, \mathrm{hyb}(\mathrm{h}, \varepsilon(\mathrm{this})))$

The abstract heap equivalent of the heap $h$

$\mathrm{abs}(\mathrm{h}))$: $\mathit{Heap} \rightharpoonup \mathit{MHeap}$

$= \mathrm{hyb}(\mathrm{h}, \mathrm{null})$

The abstract state equivalent of the state $(h, \varepsilon)$

$\mathrm{abs}((\mathrm{h}, \varepsilon))$: $\mathit{State} \rightharpoonup \mathit{MState}$

$= (\varepsilon, \mathrm{hyb}(\mathrm{h}, \mathrm{null}))$

*Fig. D.7:* Auxiliary functions — hybrid

Embedding of a callee's local heap into the caller's heap

$\mathrm{embed}(\mathrm{h_{mc}}, \mathrm{h_{sx}})$: $\mathit{HHeap} \times \mathit{MHeap} \rightarrow \mathrm{P}(\mathit{HHeap})$

$= \mathrm{embed_r}(\{\mathrm{h_{mc}}[\mathrm{h_{sx}}]\}, \emptyset, \mathrm{headers}(\mathrm{h_{mc}}) \setminus \varepsilon_{\mathrm{mc}}(\mathrm{this}), \mathrm{mod}(\mathrm{h_{mc}}, \mathrm{h_{sx}}), \mathrm{dom}(\mathrm{h_{sx}}))$

Recursive embeding

$\mathrm{embed_r}(\mathrm{H}, \mathrm{D}, \mathrm{C}, \mathrm{M}, \mathrm{LH})$

$: \mathrm{P}(\mathit{HHeap}) \times \mathrm{P}(\mathit{Loc}) \times \mathrm{P}(\mathit{Loc}) \times \mathrm{P}(\mathit{LMF}) \times \mathrm{P}(\mathit{LMF}) \rightarrow \mathrm{P}(\mathit{HHeap})$

$$= \begin{cases} \mathrm{H} & : \mathrm{L} = \emptyset \\ \cup\{\mathrm{embed_r}(\mathrm{embed_l}(\mathrm{U(h)}, \mathrm{h}), \mathrm{D} \cup \mathrm{L(h)}, \mathrm{C} \setminus \mathrm{L(h)}, \mathrm{M} \cup \mathrm{U(h)}, \mathrm{LH}) : \mathrm{h} \in \mathrm{H}\} & : \mathrm{L} \neq \emptyset \end{cases}$$

Where :

$\mathrm{L(h)} = \{\mathrm{l} \in \mathrm{C} : \mathrm{outgoing(l, h)} \subseteq \mathrm{D}\}$

$\mathrm{U(h)} = \{(\mathrm{l}, \mathrm{mf}) \in \mathrm{dom(h)} \setminus \mathrm{LH} : \mathrm{l} \in \mathrm{L(h)} \wedge \mathrm{sframe(mf)(l, h)} \cap \mathrm{M} \neq \emptyset\}$

Recalculate the model functions for lmfs U

$\mathrm{embed_l}(\mathrm{U}, \mathrm{h})$: $\mathrm{P}(\mathit{LF}) \times \mathit{HHeap} \rightarrow \mathrm{P}(\mathit{HHeap})$

$= \{\mathrm{h}\}[\cup\{(\mathrm{l}, \mathrm{mf}) \mapsto \mathrm{T_{mf}}(\mathrm{mf})(\mathrm{l}, \mathrm{h}) : (\mathrm{l}, \mathrm{mf}) \in \mathrm{U}\}]$

LFs whose value is different between 2 heaps

$\mathrm{mod}(\mathrm{h_1}, \mathrm{h_2})$: $\mathit{HHeap} \times \mathit{HHeap} \rightarrow \mathrm{P}(\mathit{Loc} \times \mathit{HFId})$

$= \{(\mathrm{l}, \mathrm{f}) \in \mathrm{dom}(\mathrm{h_1}) \cap \mathrm{dom}(\mathrm{h_2}) : \mathrm{h_1}(\mathrm{l}, \mathrm{f}) \neq \mathrm{h_2}(\mathrm{l}, \mathrm{f})\}$

*Fig. D.8:* The embed function

## *D.2 Heap Admissibility*

We define a subset of the possible heaps which are admissible. We later show that only admissible heaps can occur in the execution of an admissible program. An admissible concrete heap satisfies:

*No dangling pointers* $\forall(l, f) \in dom(h) : h(l, f) \in Loc \to h(l, f) \in ldom(h)$

*Type consistency* For each location in the domain, exactly the fields of one type are in the domain. $\forall l \in ldom(h) : type(l, h) \downarrow\}$

*Type safety* for each (location, fieldId) in the domain, its value is of the type declared for the field $\forall(l, f) \in dom(h) : h(l, f) \in Loc \to type(l, h) = ftype(f)$

An admissible hybrid heap:

- No dangling pointers
- Only the current component is concrete — other components are model
- The current component is a root (has no incoming pointers)
- Type consistency
- Type safety

An admissible model heap:

- No dangling pointers
- Type consistency
- Type safety
- Every model field agrees with its model functions: $\forall l, mf : h(l, mf) \in m(mf)(h, l)$

Similarly for states, a state is admissible if

- The heap is admissible
- The environment has pointers only into the current component
- This points to the head of the current component which has no incoming pointers

# E. PROOFS

## *E.1 Soundness*

**Proof for Lemma. B.1.1** — The soundness theorem:
Given

- a program *p* for which all proof obligations are satisfied
- A function *f* of the main module of *p*
- An admissible state $\sigma$ of *p*
- A state $\sigma$' of *p* s.t. $\langle \mathbf{f}, \sigma \rangle \rightsquigarrow_s \sigma'$

We show that $\langle \mathbf{f}, \mathrm{hyb}(\sigma) \rangle \rightsquigarrow_m \mathrm{hyb}(\sigma')$.

*Proof.* The theorem is proven by *induction on the module import relation of the program.* (which is a DAG as module import is acyclic)- as each function of each module can only call functions of imported modules.
We use a slightly stronger induction hypothesis: Each public function of each lower levels in the module hierarchy complies with its specification. Formally: for each public function f' of an imported module:
$\langle \mathbf{f'_{body}}, \sigma_{\mathrm{e}} \rangle \rightsquigarrow_s \sigma_x$ implies $\langle \mathbf{f'_{spec}}, \sigma_{\mathrm{se}} \rangle \rightsquigarrow \sigma_{sx}$
Where: $\sigma_{se} = abs(\sigma_e)$
$\sigma_{sx} = abs(\sigma_x)$
We use this hypothesis to show that ($\forall \sigma_e, \sigma_x$):
$\langle \mathbf{f_{body}}, \sigma_{\mathrm{e}} \rangle \rightsquigarrow_s \sigma_x$ implies $\langle \mathbf{f_{body}}, \sigma_{\mathrm{me}} \rangle \rightsquigarrow_m \sigma_{mx}$
Where:
$\sigma_{me} = hyb(\sigma_e)$
$\sigma_{mx} = hyb(\sigma_x)$
Then we use Proof Ob C.1.8 which states:
$\langle \mathbf{f_{body}}, \sigma_{\mathrm{me}} \rangle \rightsquigarrow_m \sigma_{mx}$ implies $\langle \mathbf{f_{spec}}, \sigma_{\mathrm{se}} \rangle \rightsquigarrow \sigma_{sx}$
Where:

$\sigma_{se} = r_f(\sigma_{me})$
$\sigma_{sx} = r_f(\sigma_{mx})$
To get:
$\langle \mathbf{f_{body}}, \sigma_{\mathrm{e}} \rangle \rightsquigarrow_s \sigma_x$ implies $\langle \mathbf{f_{spec}}, \sigma_{\mathrm{se}} \rangle \rightsquigarrow \sigma_{sx}$
As $r_f(hyb(\sigma)) = abs(\sigma)$
We show this by *structural induction on the body of f.*
**Intra-procedural statements:**
By the definition of hyb, $h =_{com(h,t)} hyb(h,t)$, i.e., the concrete and hybrid heaps are equal on the current component $com(h,t)$.
Lemma. B.3.2 and Lemma. B.4.2 imply that intra-procedural statements only affect the current component. Formally, consider an intra-procedural statement $S$ (for both modular and standard semantics). Assume that $\langle \mathbf{S}, (\varepsilon, \mathrm{h}) \rangle \rightsquigarrow_{s,m} (\varepsilon', \mathrm{h}')$. This implies that $h =_{comp(h,t)^C} h'$, where $t = \varepsilon(this)$.
Lemma. B.3.3 and Lemma. B.4.3 imply that intra-procedural statements are only affected by the current component. Formally, consider an intra-procedural statement $S$ and 2 states (concrete or hybrid) $(\varepsilon, h_1), (\varepsilon, h_2)$ s.t., $h_1 =_{comp(h_1,t)} h_2$ where $t = \varepsilon(this)$, This ensures that $\langle \mathbf{S}, (\varepsilon, \mathrm{h}_1) \rangle \rightsquigarrow_{s,m} (\varepsilon', \mathrm{h}_1')$ if and only if $\langle \mathbf{S}, (\varepsilon, \mathrm{h}_2) \rangle \rightsquigarrow_{s,m} (\varepsilon', \mathrm{h}_2')$ Where $h1' =_{comp(h_2',t)} h_2'$ and $h2 =_{comp(h_2',t)^C} h_2'$
Hence, t if S is intra-procedural then
$\langle \mathbf{S}, \sigma \rangle \rightsquigarrow_s (\sigma')$
iff
$\langle \mathbf{S}, \mathrm{hyb}(\sigma) \rangle \rightsquigarrow_m \mathrm{hyb}(\sigma')$
Hence we get that the theorem holds for intra-procedural statements.
**Inter-procedural calls:**
For a call $f'(\overline{v})$ we have to show that:
$\langle \mathbf{f}'(\overline{\mathbf{v}}), \sigma_{\mathrm{c}} \rangle \rightsquigarrow_s \sigma_r$ implies $\langle \mathbf{f}'(\overline{\mathbf{v}}), \mathrm{hyb}(\sigma_{\mathrm{c}}) \rangle \rightsquigarrow_m hyb(\sigma_r)$.
In the terminology of Sec. A.3 the only derivation of
$\langle \mathbf{f}'(\overline{\mathbf{v}}), \sigma_{\mathrm{c}} \rangle \rightsquigarrow_s \sigma_r$ is from $\langle \mathbf{f}'_{\mathbf{body}}, \sigma_{\mathrm{e}} \rangle \rightsquigarrow_s \sigma_x$
Where

- $\varepsilon_{\mathrm{r}} = \varepsilon_{\mathrm{c}}[\varepsilon_{\mathrm{x}}[\overline{\mathrm{v}}|\overline{\mathrm{a}}]]$
- $\mathrm{h_r} = \mathrm{h_c}[\mathrm{h_x}]$
- $\varepsilon_{\mathrm{e}} = \varepsilon_{\mathrm{c}}|_{\overline{\mathrm{v}}}[\overline{\mathrm{a}}|\overline{\mathrm{v}}]$
- $\mathrm{h_e} = \mathrm{h_c}||_{\varepsilon_{\mathrm{c}}(\overline{\mathrm{v}})}$

We know (from the induction hypothesis) that

$\langle \mathbf{f'_{body}}, \sigma_{\mathrm{e}} \rangle \rightsquigarrow_s \sigma_x$ implies $\langle \mathbf{f'_{spec}}, \mathrm{abs}(\sigma_{\mathrm{e}}) \rangle \rightsquigarrow abs(\sigma_x)$
And from the modular call rule we have, in the terminology of Sec. A.4 :
$\langle \mathbf{f'_{spec}}, \sigma_{\mathrm{se}} \rangle \rightsquigarrow \sigma_{sx}$ implies $\langle \mathbf{f'(\overline{v})}, \sigma_{\mathrm{mc}} \rangle \rightsquigarrow_m \sigma_{mr}$
Where

- $\varepsilon_{\mathrm{mr}} = \varepsilon_{\mathrm{mc}}[\varepsilon_{\mathrm{sx}}[\overline{\mathrm{v}}|\overline{\mathrm{a}}]]$
- $\mathrm{h_{mr}} \in \mathrm{embed}(\mathrm{h_{mc}}, \mathrm{h_{sx}})$
- $\varepsilon_{\mathrm{se}} = \varepsilon_{\mathrm{mc}}|_{\overline{\mathrm{v}}}[\overline{\mathrm{a}}|\overline{\mathrm{v}}]$
- $\mathrm{h_{se}} = \mathrm{h_{mc}}||_{\varepsilon_{\mathrm{mc}}(\overline{\mathrm{v}})}$

The environments use the exact same substitution for the standard and modular semantics, hence we only have to show that
(we use $t = \varepsilon_c(this) = \varepsilon_r(this) = \varepsilon_{mc}(this) = \varepsilon_{mr}(this)$):
$abs(h_e) = h_{se}$
meaning
$\mathrm{abs}(\mathrm{h_c}||_{\varepsilon_{\mathrm{c}}(\overline{\mathrm{v}})}) = \mathrm{hyb}(\mathrm{h_c}, \mathrm{t})||_{\varepsilon_{\mathrm{c}}(\overline{\mathrm{v}})}$
and
$hyb(h_r, t) \in embed(h_{mc}, h_{sx})$
meaning
$\mathrm{hyb}(\mathrm{h_c}[\mathrm{h_x}], \mathrm{t}) \in \mathrm{embed}(\mathrm{hyb}(\mathrm{h_c}, \mathrm{t}), \mathrm{abs}(\mathrm{h_x}))$

- $abs(h_c||_{\varepsilon_c(\overline{v})}) = hyb(h_c, t)||_{\varepsilon_c(\overline{v})}$ - this holds because of Lemma. D.1.1
- $hyb(h_c[h_x], t) \in embed(hyb(h_c, t), abs(h_x))$ this holds by Lemma. B.2.1

So $\langle \mathbf{f'_{body}}, \sigma_{\mathrm{e}} \rangle \rightsquigarrow_s \sigma_x$ implies $\langle \mathbf{f'(\overline{v})}, \mathrm{hyb}(\sigma_{\mathrm{c}}) \rangle \rightsquigarrow_m hyb(\sigma_r)$
and so $\langle \mathbf{f'(\overline{v})}, \sigma_{\mathrm{c}} \rangle \rightsquigarrow_s \sigma_r$ implies $\langle \mathbf{f'(\overline{v})}, \mathrm{hyb}(\sigma_{\mathrm{c}}) \rangle \rightsquigarrow_m hyb(\sigma_r)$.

$\square$

## E.2 The hyb and embed functions

**Proof for Lemma. B.2.1** — Embed is conservatices w.r.t. hyb:
If

- $\sigma_c$ is an admissible concrete state
- $\sigma_x$ is an admissible concrete state

- $\forall l \in \mathit{ldom}(h_x) \cap \mathit{ldom}(h_c) : \mathit{type}(l, h_c) = \mathit{type}(l, h_x)$
- $t \in \mathit{ldom}(h_c) \setminus \mathit{ldom}(h_x)$

then $\mathit{hyb}(h_c[h_x], t) \in \mathit{embed}(\mathit{hyb}(h_c, t), \mathit{hyb}(h_x, t))$.

*Proof.* We prove this by showing an invariant that connects each recursion level for $\mathit{hyb}$ and $\mathit{embed}$, and then showing inductively that this invariant holds until the last iteration which proves the lemma.
We use the following definitions:

$h^i_h, D^i_h, C^i_h, L^i_h$ The repsective arguments of hyb at the ith iteration

$H^i_e, D^i_e, C^i_e, M^i_e, L^i_e, U^i_e,$ The repsective arguments of embed at the ith iteration

$h_r$ $= \mathit{hyb}(h_c[h_x], t)$ — the result of $\mathit{hyb}$

$h^0_r$ $= h_c[h_x]$ — the initial input of $\mathit{hyb}$

$H_{mr}$ $= \mathit{embed}(\mathit{hyb}(h_c, t), \mathit{hyb}(h_x, t))$ — the result of $\mathit{embed}$

$h^0_{mr}$ $= \mathit{hyb}(h_c, t)[\mathit{hyb}(h_x, t)]$ — the inital input of $\mathit{embed}$

$D^i_{lmf}$ $= \{(l, mf) \in \mathit{dom}(h^i_h) : l \in D^i_e\}$ : The lmfs of $D^i_h$ — the layers of iterations before i

$L^i_{lmf}$ $= \{(l, mf) \in \mathit{dom}(h^i_h) : l \in L^i_e\}$ : The lmfs of the objects of $L^i_h$ — the layer handled in iteration i

$LH^i_{lmf}$ $= \{(l, mf) \in \mathit{dom}(h^i_h) : l \in LH \cap D^i_e\}$ : The lmfs of the local heap in the current layer

$P^i$ $= \{D^i_{lmf} \setminus M^i_e \setminus LH^i_{lmf}$ : The preserved (unmodified) lmfs in $D^i_e$

The main invariant is:
$\exists h \in H^i_e : h =_{D^i_e} h^i_h$
We also show and use the following auxiliary properties:

- $\forall i : D^i_e = D^i_h$ this follows from the induction invariant and Proof Ob C.1.2
- $\forall i : \forall h \in H^i_e : L^i_e(h) = L^i_h$

We show induction on the number of iteration of hyb and embed:
**Basis:** At the basis

- $H_e^0 = h_{mr}^0$
- $h_h^0 = h_r^0$
- $D_e^0 = \emptyset$

So the main invariant holds vacuously.
**Step:**
Given:
$\exists h \in H_e^i : h =_{D_e^i} h_h^i$
We have to show that:
$\exists h \in H_e^{i+1} : h =_{D_e^{i+1}} h_h^{i+1}$
We name $h_e^i$ the realization of the existential quantifier in the hypothesis.
So we have to show that:
$\exists h \in embed_l(U_e^i(h_e^i), h_e^i) : h =_{D_e^{i+1}} hyb_l(L_h^i, h_h^i)$
Where $h_e^i =_{D_e^i} h_h^i$
By definition $\forall h \in H_e^{i+1} : dom(h) = dom(h_h^{i+1})$
By definition $D_{lmf}^{i+1} = D_{lmf}^i \cup L_{lmf}^i$ Where $L_{lmf}^i = P^i \cup LH^i \cup M^i$
Using this we partition $D_{lmf}^{i+1}$ to four sets:
$D_{lmf}^{i+1} = D_{lmf}^i \cup P^i \cup LH^i \cup M^i$ Where the unions are all disjoint.
For $S = D_{lmf}^i, LH^i, M^i$ we show $\forall h \in H_e^{i+1} : h =_S h_h^i$
For $P^i$ we show $\exists h \in H_e^{i+1} : h =_{P^i} h_h^i$
We use the following properties of $embed_l$ and $hyb_l$:
$\forall (l, mf) \notin L_{lmf}^i : hyb_l(L_h^i, h_h^i)(l, mf) = h_h^i(l, mf)$
$\forall (l, mf) \notin U_{lmf}^i : \forall h \in embed_l(U_e^i, h_e^i) : h(l, mf) = h_e^i(l, mf)$ $U_{lmf}^i \subseteq L_{lmf}^i$
We also use the property that if t is at the root (no component has outgoing pointers to t) — as is guaranteed by $\sigma_c$ being admissible — then $ldom(hyb(h_c[h_x])) = \cup_i D^i \cup \{t\}$ where the unions are disjoint (by definition)
We now show the proof for each of the four sets:
$\mathbf{D^i_{lmf}}$: All previous layers
As $D_{lmf}^i \cap U_{lmf}^i = \emptyset$ we get for both embed and hyb: $hyb_l(L_h^i, h_h^i)(l, mf) =_{D_{lmf}^i} h_h^i(l, mf)$. $\forall h \in embed_l(U_e^i, h_e^i) : h(l, mf) =_{D_{lmf}^i} h_e^i(l, mf)$ So we can use the induction hypothesis $h_e^i =_{D_e^i} h_h^i$ to get $\forall h \in embed_l(U_e^i, h_e^i) : h =_{D_e^i} hyb_l(L_h^i, h_h^i)$
$\mathbf{LH^i}$: Call local heap in current layer
As $LH^i \cap U_{lmf}^i = \emptyset$ we get: $\forall h \in embed_l(U_e^i, h_e^i) : h =_{LH^i} h_e^i$ As $\forall i \neq j : D^i \cap D^j = \emptyset$ and $LH^i \subseteq D_{lmf}^i$ we get $h_e^i =_{LH^i} h_e^0 = hyb(h_c, t)[hyb(h_x, t)]$ As $LH^i \subseteq LH = dom(hyb(h_x, t))$ we get (by Lemma. B.2.2) $hyb(h_c, t)[hyb(h_x, t)] =_{LH^i} hyb(h_c[h_x], t)$ And as $hyb(h_c[h_x], t) =_{L^i} h_h^{i+1}$ we get $h_e^{i+1} =_{LH^i} h_h^{i+1}$

**$\mathbf{P^i}$**: Unaffected lmfs in current layer (excluding local heap)
As $P^i \cap U^i_{lmf} = \emptyset$ we get $\forall h \in embed_l(U^i_e, h^i_e) : h =_{P^i} h^i_e$. As $\forall i \neq j$ : $D^i \cap D^j = \emptyset$ and $P^i \subseteq D^i_{lmf}$ we get $h^i_e =_{P^i} h^0_e = hyb(h_c, t)[hyb(h_x, t)]$ As $P^i \cap LH = \emptyset$ we get $hyb(h_c, t)[hyb(h_x, t)] =_{P^i} hyb(h_c, t)$ By Lemma. B.2.3 and $P^i \subseteq P$ we get $hyb(h_c, t) =_{P^i} hyb(h_c[h_x], t)$ and as $hyb(h_c[h_x], t) =_{L^i} h^{i+1}_h$ we get $h^{i+1}_e =_{P^i} h^{i+1}_h$ We know $P^i \subseteq P$ because $P^i$ includes only lmfs for which the $sframe$ does not coincide with $LH$ where the lfp calculation for $P$ uses the transitive closure of sframe and its intersection with $LH$.
**$\mathbf{U^i}$**: Modified lmfs in current layer (excluding local heap)
By definition $\forall(l, mf) \in U^i : outgoing(l, h^i_e) \in D^i$ By Proof Ob C.1.5 and and by the admissibility of the model function , we know that $\forall(l, mf) \in U^i$ : $sframe(mf)(l, h^i_e) \subseteq P^i \cup LH^i \cup D^i_{lmf}$ By the previous sections we know that $h^i_e =_{P^i \cup LH^i \cup D^i} h^{i+1}_h$ So by Proof Ob C.1.6 $T_{mf}(mf)(l, h^i_e) = T_{mf}(mf)(l, h^{i+1}_h)$ By Proof Ob C.1.9 $T_{rf}(mf)(l, h^i_h) \in T_{mf}(mf)(l, h^{i+1}_h)$.
Therefore $h^{i+1}_h(l, mf) = T_{rf}(mf)(l, h^i_h) \in T_{mf}(mf)(l, h^{i+1}_h) = T_{mf}(mf)(l, h^i_e)$
And so, as $h^{i+1}_e$ is a cartesian product of all sets of possible values for all lmfs in $U^i$ we get $\exists h \in embed_l(U^i_e, h^i_e) : h =_{U^i} h^{i+1}_h$
And hence combining all results we get: $\exists h \in embed_l(U^i_e, h^i_e) : h =_{D^{i+1}} h^{i+1}_h$
□

**Proof for Lemma. B.2.2** — Embed preserves the local heap:
If

- $h_c$ is an admissible concrete heap
- $h_x$ is an admissible concrete heap
- $\forall l \in ldom(h_x) \cap ldom(h_c) : type(l, h_c) = type(l, h_x)$

then $hyb(h_c, t)[abs(h_x)] =_{dom(abs(h_x))} hyb(h_c[h_x], t)$.

*Proof.* We prove this by induction on the iterations of abs on the left and hyb on the right. The main property which makes this hold is that the frame of each model field in $abs(h_x)$ is inside $abs(h_x)$, and hence adding elements to the heap cannot affect the calculation of the representation function by the admissibility of the representation function (the frame is reachable from this) and by $h_c$ being downward closed.
We use $h^i_a$, $D^i_a$, $L^i_a$ for h,D,L in iterations of $abs(h_x)$ and $h^i_h$, $D^i_h$, $L^i_h$ for iterations of $hyb(h_c[h_x], t)$.

The invariant of the induction at iteration i is : $h_a^i =_{D_a^i \cap ldom(h_x)} h_h^i$ and $D_a^i \cap ldom(h_x) = D_h^i \cap ldom(h_x)$.
Initially this holds trivially as $D_a^i = D_h^i = \emptyset$.
In each step $L_a^i \cap dom(h_x) = L_h^i \cap dom(h_x)$ because for each $(l, mf) \in dom(h_x)$ s.t. $l \in L_h^i$, $frame(mf)(h_a^i, l) \subseteq D_a^i \times \mathit{MFId} \cup comp(l, h_a^i)$ because of the admissibility of the representation function. and similarly $frame(mf)(h_h^i, l) \subseteq D_h^i \times \mathit{MFId} \cup comp(l, h_h^i)$.
As $h_h^i =_{L_a^i \cap dom(h_x)} h_a^i$ and from the induction hypothesis $h_h^i =_{D_a^i \cap dom(h_x)} h_a^i$ we get $h_h^i =_{frame(mf)(h_h^i, l)} h_a^i$ and so, by the definition of frame, $T_{rf}(mf)(h_h^i, l) = T_{rf}(mf)(h_a^i, l)$ and $h_a^{i+1} =_{D_a^{i+1}} h_h^{i+1}$.

□

**Proof for Lemma. B.2.3** — Embed preserves the unmodified lmfs:
If

- $h_c$ is an admissible concrete heap
- $h_x$ is an admissible concrete heap
- $\forall l \in ldom(h_x) \cap ldom(h_c) : type(l, h_c) = type(l, h_x)$
- $h_r = hyb(h_c[h_x], t)$
- $P = \{(l, mf) \in dom(h_r) : rsframe(mf)(h_r, l) \cap mod(h_c, h_x) = \emptyset\}$

then $hyb(h_c, t) =_P hyb(h_c[h_x], t)$

*Proof.* We show this by induction on the dependency DAG of $P$ (which is closed to dependencies by definition).
At the roots are lmfs with no external dependencies for which $T_{rf}$ is calculated on the same concrete component (although possibly in different stages).
As each lmf is calculated at most once in $hyb$, and each lmf is calculated only after its entire $rsframe$ has been calculated, we can use as the induction invariant the rsframe equality of all dependencies of (l,mf) on the partially abstracted heap on both sides. from this follows directly, by Proof Ob C.1.4, that they are calculated on both sides with heaps equal on the frames and therefore the result is identical by the definition of frame.

□

**Proof for Lemma. B.2.4** — The hyb function preserves admissiblitiy of heaps:
If

- $\sigma$ is an admissible concrete state

then $hyb(\sigma)$ is an admissible hybrid state.

*Proof.*

- The no-dangling-pointers property in $hyb(h)$ holds because it holds in $h$ and because of Proof Ob C.1.7 and Proof Ob C.1.3 hence each outgoing pointer in $hyb(h)$ corresponds to an outgoing pointer in $h$ which points to a public class — hence exists in $hyb(h)$
- Type consistency holds because $hyb$ calculates for each location only and all the model fields for the type of that location
- Type safety follows immediately from type safety of the representation function (from its admissibility)
- Only the current component remains concrete because of the stopping condition for $hyb$: at each stage we iterate over all hybrid components but the current component, and as $h$ is admissible every component has all its subcomponents and they form a DAG, hence every component but $h$ is eventually abstracted

□

## E.3 Properties of LHS

**Proof for Lemma. B.3.1** — LHS preserves admissibility:
If

- $\sigma$ is an admissible concrete state
- $S$ is a statement of the semantics
- $\langle \mathrm{S}, \sigma \rangle \rightsquigarrow_s \sigma'$

Then $\sigma'$ is admissible.

*Proof.*

- No dangling pointers: New pointers values are only created on object creation (otherwise just copied from heap/env to heap/env) and object creation returns a pointer to an allocated object — there are no deallocations. No expression can return a dangling pointer because of Proof Ob C.1.3

- Type consistency: The only heap modifying statements are object creation, field write and call:
  - Object creation: for each newly allocated object exactly the fields of one type are created
  - Field write: happens only to existing fields
  - Call: only overriding union of admissible heaps is performed, in which the types of objects agree (because no statement changes the type of an object and the exit heap is reached from the entry heap which is cut from the call heap)
- Type safety: fields get new values only on creation, field write and call:
  - Object creation: All types must support the null value
  - Field write: By the side-condition
  - Call: As above, as types are consistent among the call and exit heaps, overriding union preserves type safety

□

**Proof for Lemma. B.3.2** — The intra-module LHS modifies only the current component:
If

- $\sigma$ is an admissible concrete state
- $S$ is an intra-module statement of the semantics
- $\langle \mathbf{S}, \sigma \rangle \rightsquigarrow_s \sigma'$

Then $\sigma =_{comp(h,\varepsilon(this))^C} \sigma'$.

*Proof.* By the sideconditions for field write and read: Only fields of the current component are modified as local variables can only point to the current component and headers of other components — and cannot modify fields of headers of other components hence can only modify the current component □

**Proof for Lemma. B.3.3** — The intra-module LHS is affected only by the current component:
If

- $\sigma_1, \sigma_2$ are admissible concrete states
- $S$ is an intra-module statement of the semantics
- $\sigma_1 =_{com(h,\varepsilon(this))} \sigma_2$.
- $\langle \mathbf{S}, \sigma_1 \rangle \rightsquigarrow_s \sigma_1'$

Then exists $\sigma_2'$ s.t. $\langle \mathbf{S}, \sigma_2 \rangle \rightsquigarrow_s \sigma_2'$ and $\sigma_1' =_{comp(h,\varepsilon(this))} \sigma_2'$.

*Proof.* By the sidecondition for field read,write □

## *E.4 Properties of LHMS*

**Proof for Lemma. B.4.1** — LHMS presreves admissibility:

- $\sigma$ is an admissible hybrid state
- $S$ is a statement of the semantics
- $\langle \mathbf{S}, \sigma \rangle \rightsquigarrow_s \sigma'$

Then $\sigma'$ is admissible.

*Proof.*
- No dangling pointers: As in Lemma. B.3.1
- Type consistency: As in Lemma. B.3.1 except
  - Call: the embed function preserves type consistency as it only changes the values of fields
- Type safety: As in Lemma. B.3.1 except
  - Call: As above, as types are consistent among the call and exit heaps and the model function is type-consistent

□

**Proof for Lemma. B.4.2** — LHMS intra-module is only modifies the current component:
If

- $\sigma$ is an admissible hybrid state

- $S$ is an intra-module statement of the semantics
- $\langle \mathbf{S}, \sigma \rangle \leadsto_s \sigma'$

Then $\sigma =_{com(h,\varepsilon(this))^C} \sigma'$.
Proof as per Lemma. B.3.2.
**Proof for Lemma. B.4.3** — LHMS intra-module is only affected by the current component:
If

- $\sigma_1, \sigma_2$ are admissible hybrid states
- $S$ is an intra-module statement of the semantics
- $\sigma_1 =_{com(h,\varepsilon(this))} \sigma_2$.
- $\langle \mathbf{S}, \sigma_1 \rangle \leadsto_s \sigma_1'$

Then exists $\sigma_2'$ s.t. $\langle \mathbf{S}, \sigma_2 \rangle \leadsto_s \sigma_2'$ and $\sigma_1' =_{com(h,\varepsilon(this))} \sigma_2'$.
Proof as per Lemma. B.3.3

# F. SPECIFICATION LANGUAGE

## *F.1 The language*

A possible specification language is detailed in Fig F.1. The language is somewhat similar to JML [12] A specification expression (sexpression) is just an expression made up of specification functions and field paths. For representation functions we have expression over hybrid field paths implicitly starting at this — this ensures the frame is reachable. For model functions we use expressions over model field paths. For pre-post specifications we use a set of sets of 2-vocabulary expressions: For each (possibly) modified model-field-path (starting at the arguments) we give its new value as an expression on the pre-state. The atomic specification functions are n-ary functions from $HVal^n \rightharpoonup HVal$ with the following restrictions:

- Location independent — if we swap all instances of one location in the arguments of a function it gives give the same result (swapped if it equaled the swapped location) - only location equality comparison is allowed
- Can only project locations — a function can return a location only if it is one of the arguments — this prevents casting to pointer or pointer arithmetic

It is easy to see these properties hold also in finite compositions of such functions. For model functions, we only use atomic functions from $MVal^n \rightharpoonup MVal$. In order to support recursive data-structures, we allow for representation functions for private classes. For example — the rep-func for list header $List.seq = head$ ? $head.seq$ : $\langle\rangle$ and for $Node.seq = next$ ? $\langle val \rangle.\langle next.seq \rangle$ : $\langle val \rangle$. This can lead to cyclic definitions — we do not offer a way to statically check for them here (except for the obvious conservative type-based way), but several can be thought of — in any case, over a specific state the cyclicity can be determined easily. The modifies clause is omitted — it is implicit in the pre-post specification. The function pre-conditions may not be derivable easily (depending on the atomic specification functions) we omit it here but it can be given explicitly as boolean sexpression over the pre-state. The dependencies of a model field are

given implicitely by the leaves of the expression tree of the model function for that model-field.
The example is given in the said specification language — the specification functions used is dom — domain of a partial function.

## *F.2 Evaluation*

Given an admissible hybrid heap h, evaluating a representation function $T_{rf}$ for the model field mf over h for location l $T_{rf}(mf)(h,l)$ is done by evaluating each field path p in the expression tree's leaves as h(l,p) and then evaluating the functions at the nodes bottom up to the root. For model functions this is done similarly. For pre/post conditions, we have a set of mappings from model field paths to expressions — for each member of the set (which represents one deterministic model post-state) the represented post state is the pre-state (local heap) where the value of each mapped model field path is replaced by evaluating its mapped expression over the pre-state. This may be inconsistent if 2 aliased paths are mapped to expressions that evaluate to different values or if the evaluated post-state is inconsistent with the model function — these can be discarded by the client (or not — does not affect soundness) these may also be chackable by the veifier of the function.

## *F.3 Showing the proof obligations*

Given the definition of the classes in module (as finite partial mappings fieldId to Type) the set of possible field-paths (within the component) is a regular language (a graph with classes as nodes and an edge for each fieldId from its class to its type labled by the fieldID gives the deterministic accepting automaton) We show the proof obligations by: Specification:

- Location independence — automatic by location independence of atomic specification functions.
- Frames of representation and model functions are reachable from component head — by construction.
- Locality of pivot frame — the expressions for pivots include only concrete field paths (syntactic). In the example — the pivot Keyset.map=_map where this._map is a local path.

- Specification frame included in frame (for subcomponents) — the frame for each expression is a finite set of field-paths, for recursive representation functions, the frame of each expression is the union of frames of its leaves, where the frame of each leaf which references a rep-function is the frame of that rep-function. In the example, the frame of Keyset.keys in subcomponents is $\{_map.v\}$ while the specification frame is $\{map.v\}$ — syntactic substitution of pivots gives inclusion.

- Equality of hybrid and model component graph — we restrict the set of *possible* pivots to be finite — as the set of concrete paths is regular, and as we can identify fields with outgoing pointers by their types, we can enumerate the set of possible outgoing field paths of a component as a regular expression (intersection of that component's field paths with all field paths ending with outgoing labels) We can easily see whether this enumeration is finite and covered by the model-fields of the representation function) In the example the only outgoing pointer (for Keyset) is _map — which is covered by the pivot map.

- Dependencies are locally acyclic — we can build a graph with nodes as model fields of current component and dependencies as edges (syntactic appearance of a model-field in the expression for another) — and check its acyclicity. In the example, for Keyset, keys depends on map — acyclic.

Implementation:

- Compliance with specification — We can use any intra-procedural analysis/verification approach that supports both our concrete and abstract domains — The analysis would work on the hybrid heap of each component at a time, showing, for each function, that if the pre-condition is satisfied so is the postcondition (it would have to evaluate the representation function on the post-state and use large step state updates for inter-module function calls)

- Model function approximates the representation function — the decidability of this depends on the atomic specification functions and specification domain, we offer a restrictive yet usable form in which the proof is trivial — the expession for the representation function is defined as the expression for the model function, with each local model-field replaced with its representation expression (where available)) in this form this is true by construction. In the example, this is satisfied as the representation function for keys is a substitution of the representation function for map into the model function for keys. Another way is the MGIC — we can use an analysis/verification tool by building a most general intrusive client for each component type — a loop with non-deterministic choice between the bodies of the components' functions and functions on subcomponents — assuming pre-conditions before the call/body and asserting the model function approximates the representation function after the call (we could also assert each function's post-conditions). We implemented this for our example.

## *F.4 Discussion*

We have presented a simple yet useful restriction on the specification language to allow it to use most analysis techniques with little added verification burden. This method completely ignores invariants — it can be extended easily by adding a decidable language of invariants over e.g. regular expression on field-paths — thus allowing us to prove more programs. For unbounded numbers of pivots (e.g. list with outgoing pointers at each node) we can extend our language with sets or sequences of locations, and correspondingly relax restrictions on pivots. In our language the model-function is single-valued — we can extend it to multi-valued model-functions by allowing multi-valued atomic specification functions — but then, to show correspondence with the representation function syntactically — we would have to have a way to relate multi-valued functions to single valued ones — (e.g. — for not specifying the iteration order of a set, we could have a multi-valued function anyPermutation which receives a set and returns its premutations and then the private represntation function $r_{iterOrd\_i}$ and the public one $r_{iterOrd} = r_{iterOrd\_i}$ would give the iteration order the set value would be $r_{set} = ran(iterOrd)$ and the model function would be $m_{iterOrd} = anyPermutation(s)$)

$$
\begin{array}{lcl}
\mathrm{T}_{\mathrm{rf}} & = & \langle \text{hsexpression} \rangle \\
\mathrm{T}_{\mathrm{mf}} & = & \langle \text{msexpression} \rangle \\
\mathrm{f}_{\mathrm{spec}} & = & P(\langle \text{tvsexpression} \rangle *)
\end{array}
$$

$$
\begin{array}{lcl}
\text{hsexpression} & = & \mathit{HFPath} \\
 & = & \textsf{atomic-sfunction}(\overline{\langle \text{hsexpression} \rangle}) \\
 & & \\
\text{msexpression} & = & \mathit{MAPath} \\
 & = & \textsf{atomic-sfunction}(\overline{\langle \text{msexpression} \rangle}) \\
 & & \\
\text{tvsexpression} & = & \mathit{MAPath} = \langle \text{msexpression} \rangle
\end{array}
$$

*Fig. F.1:* The specification language.

# G. RUNNING EXAMPLE

The running example is shown in detail in listings 1-6. List G.1 shows the client of a Map (Integer to Integer) and a Keyset of the map. The client creates a Map and its corresponding Keyset, modifies the map and observes the change in the Keyset. List G.2 shows the Map interface — the function signatures, and the functions' pre-post specification. The specification consists of modeling (the v field — a partial function of Integer to Integer) and the functions' pre-post specification (in JML style — in terms of the model v field and arguments). List G.3 shows the Keyset interface — the functions and the specification. The specification consists of modeling: the keys field — a set of keys and the map field — a pivot pointing to the wrapped map, the function pre-post specification and the model function - specifying how the keys field changes when the underlying Map is modified. Note the constructor connects the Keyset to the Map using the map model field (a pivot). List G.4 shows the implementation of the Keyset - the notable parts are the representation function - that is, the actual mapping of the Keyset's state to model state. List G.5 and List G.6 show the implementation of the Map - The implementation is a standard not balanced binary search tree. The Map.Node class is written as a static inner class for convenience only (could be just a private class in the same package/module). The representation function for Map is recursive — using the recursive function for Map.Node. The representation function for Map.Node combines the current node with the left and right subtrees (if exist).

```
module client;
import keyset,map;

class Client{
   void f(){
      Map     m = new Map();
      Keyset s = new Keyset(m);

      assert( !s.hasKey(1) );
      m.insert(1,2);
      assert( s.hasKey(1) );
   }
}
```

*Listing G.1:* Running Example — Client

```
module map;

class Map{
   public model PartialFunction<Integer,Integer> v;

   public ensures v′={};
   public void Map();

   public requires key!=null, value!=null;
   public ensures v′=v[(key,value)];
   public void insert(Integer key, Integer value);

   public requires key!=null;
   public requires key ∈ dom(v);
   public ensures v′=v \ (key,v(key));
   public void remove(Integer key);

   public requires key!=null;
   public ensures result= if key ∈ dom(v) then v(key) else null;
   public Integer lookup(Integer key);
}
```

*Listing G.2:* Running Example — Map interface

```
module keyset;
import map;

class Keyset{
   public model Set<Integer> keys;
   public model Map map;

   public models keys = dom(map.v);

   public requires _map != null;
   public ensures map′=_map;
   public ensures keys′=dom(_map.v);
   public void Keyset(Map _map);

   public requires key != null;
   public ensures keys′=keys \ key;
   public void remove(Integer key);

   public requires key != null;
   public ensures result = key ∈ dom(v);
   public bool hasKey(Integer key);
}
```

*Listing G.3:* Running Example — Keyset interface

```
module keyset;
import map;

class Keyset{
   public model Set<Integer> keys;
   public model Map map;

   public models keys = dom(map.v);

   public requires _map != null;
   public ensures map′=_map;
   public ensures keys′=dom(_map.v);
   public void Keyset(Map _map){
      m = _map;
   };

   public requires key != null;
   public ensures keys′=keys \ key;
   public void remove(Integer key){
      if (hasKey(key)) m.remove(key);
   };

   public requires key != null;
   public ensures result = key ∈ dom(v);
   public bool hasKey(Integer key){
      return m.lookup(key)!=null;
   };

   private Map m;
   private represents map = m;
   private represents keys = dom(m.v);
}
```

*Listing G.4:* Running Example — Keyset

```
module map;

class Map{
   public model PartialFunction<Integer,Integer> v;

   public ensures v'={};
   public void Map(){head = null};

   public requires key!=null, value!=null;
   public ensures v'=v[(key,value)];
   public void insert(Integer key, Integer value){
      if (head==null)
         head = new Node(key.value);
      else
         head.insert(key,value);
   };

   public requires key!=null;
   public requires key ∈ dom(v);
   public ensures v'=v \ (key,v(key));
   public void remove(Integer key){
      head=head.remove(key);
   };

   public requires key!=null;
   public ensures result= if key ∈ dom(v) then v(key) else null;
   public Integer lookup(Integer key){
      if (head ==null) return null;
      return head.lookup(key);
   };

   private Node head;

   private represents v = if (head==null) then {} else head.v;

}
```

*Listing G.5:* Running Example — Map

```
module map;

private static class Map.Node{
   model Map<Integer,Integer> v;
   represents v =
      (k,v)
      ∪ if (left !=null) then left.v else ∅
      ∪ if (right !=null) then right.v else ∅
   Node(Integer key,Integer value){
      k=key;  v=value;
      left=null; right=null;
   }
   insert(Integer newKey,Integer newValue){
      if (k==newKey) v = newValue;
      else if (newKey<k)
         if (left==null) left = new Node(newKey,newValue);
                    else left.insert(newKey,newValue);
      else
         if (right==null) right = new Node(newKey,newValue);
                     else right.insert(newKey,newValue);
   }
   Node remove(Integer key){
      if (k==key){
              if (left!=null)  left  =  left.pruneMax(v);
         else if (right!=null) return right;
         else return null;
      }
      else if (key<k) left =  left.remove(key);
      else             right = right.remove(key);
      return this;
   }
   Node pruneMax(Integer val){
      if (right!=null){
         Node n = this;
         while (n!=null && n.right!=null && n.right.right!=null)
            n=n.right;
         val=n.right.v;
         n.right=n.right.left;
         return this;
      }else{ //right==null
         val=v;
         return left;
      }
   }
   Integer lookup(Integer key){
      if (key==k) return v;
      if (key< k &&  left!=null) return  left.lookup(key);
      if (key> k && right!=null) return right.lookup(key);
      return null;
   }

   Integer k,v;
   Node left,right;
}
```

*Listing G.6:* Running Example — MapNode