Runtime Checking for Separation Logic

Huu Hai Nguyen[1], Viktor Kuncak[2], and Wei-Ngan Chin[1,3]

[1] Computer Science Programme, Singapore-MIT Alliance
[2] Swiss Federal Institute of Technology (EPFL)
[3] Department of Computer Science, National University of Singapore

Abstract. Separation logic is a popular approach for specifying properties of recursive mutable data structures. Several existing systems verify a subclass of separation logic specifications using static analysis techniques. Checking data structure specifications during program execution is an alternative to static verification: it can enforce the sophisticated specifications for which static verification fails, and it can help debug incorrect specifications and code by detecting concrete counterexamples to their validity.

This paper presents Separation Logic Invariant ChecKer (SLICK), a runtime checker for separation logic specifications. We show that, although the recursive style of separation logic predicates is well suited for runtime execution, the implicit footprint and existential quantification make efficient runtime checking challenging. To address these challenges we introduce a coloring technique for efficiently checking method footprints and describe techniques for inferring values of existentially quantified variables. We have implemented our runtime checker in the context of a tool for enforcing specifications of Java programs. Our experience suggests that our runtime checker is a useful companion to a static verifier for separation logic specifications.

1 Introduction

Linked structures are ubiquitous in modern software. Such structures appear both in container implementations of software libraries and in application code as the form of syntax trees, XML data, and other application-specific relationships. The diversity of linked structures implies that there is a wide range of invariants that they satisfy. Automated verification of these invariants is an active area of research and includes verification of shape properties [2, 12, 18] as well as properties that extend shape descriptions with specifications of size, balancing, sortedness, and content change [16,19,21,24,28]. The specification language for expressing these properties has a significant impact on the effectiveness of the analysis and its ability to interact with the developer. Separation logic with inductively defined predicates [25] has emerged as a popular approach to specify properties that involve linked structures. In Hoare logic based on separation logic [15], an assertion specify not only the condition on the initial heap but also the "footprint" [4], that is, the part of the heap that an operation may access. As a result, a precondition simultaneously plays the role of a 'modifies' clause [13] and lead to a frame rule that enables modular reasoning [15].

Runtime checking as complementary technique. We expect that many operations and properties in practice can be checked statically, but some will remain beyond the reach of current analysis tools. In this paper we describe a system called SLICK which can check properties during program execution and can therefore serve as a fall-back of static analysis. Such runtime checking has long been recognized as useful [1, 6]. Runtime checking detects violations of desired properties in individual runs, and, unlike many static analyses, can identify cases when code or specification definitely contain an error. Other benefits of runtime checking include interfacing to unverified code, automated checking of input data that cannot be trusted, and detecting errors that result from violating design-time assumptions (for example, operating system corruption or hardware malfunction).

Previous work on runtime checking. Despite the long history of runtime assertion checking [9], to the best of our knowledge, our work is the first runtime checker for separation logic specifications. Most existing runtime assertion checkers either check assertions in classical logic [1, 8, 10, 29], weave global checks into code at multiple program points [3, 7], address blame assignment for properties expressed in the programming language [11], or explore incremental checking of assertions [26].

The closest to our system is a checker for heap contracts expressed in linear logic [23], which also observes the usefulness of checking contracts in separation logic, but proceeds to check assertions in *linear* logic instead. Note that [23] does not deal with the problem of checking that the footprint of the code executed is contained in the footprint of the assertion. The footprint checking is one of the main problems addressed in our paper: it makes preconditions checking more than just evaluating formulas in a fixed program state and requires the checking of fine-grained modifies clauses. Another difference with [23] is that, instead of invoking a modified interpreter for a linear logic programming language, our system emits Java code that can be compiled and executed using existing virtual machines. In translation from separation logic into Java our system exploits the deterministic flavor found in most common data structure descriptions. The generated code executes using standard environments and benefits from just-in-time compilation of the Java virtual machine.

Contributions. The paper makes the following contributions:

- **A translation** of declarative predicate definitions, method preconditions and postconditions expressed in separation logic specification language [21] into executable Java code.
- **Efficient runtime mechanism** for checking separation logic assertions based on coloring heap objects and method invocations. Our approach avoids the memory blow up of naïve implementations of separation logic semantics.
- **Mode analysis** for existentially quantified variables. In most specifications we encountered, existentially bound variables are ultimately given as a function of other variables. SLICK includes mode analysis that determines the place where predicate parameters are bound, classifying them into input and output parameters. SLICK also identifies *conditionally bound parameters* for parameters whose binding time depends on the invocation context of the predicate. SLICK uses a boxed representation to instantiate such parameters at runtime at the point of their first use.

– **Integration of static and runtime checking**. SLICK ensures that annotated, but statically unverified, methods conform to their specifications at runtime, providing a fall-back for the static analyzer and enabling the interface to unverified code. Conversely, the static checker can act as an optimizer for the code generated from runtime checks.

2 Example

This section illustrates our run-time checking techniques through an example that manipulates (possibly sorted) doubly-linked lists. A list is created in a region of code that was not annotated or statically verified. Therefore, our system performs a run-time check to ensure that the subsequent code can safely use the created list. Depending on the complexity of subsequent data manipulation, the system ensures invariants in subsequent piece of code either statically, using entailment checker for separation logic [21], or dynamically, using further run-time checks.

```
class Node { int val; Node next, prev; }

dll⟨p,n⟩ == (root = null ∧ n=0)
            ∨ (root::Node⟨v,r,p⟩ ∗ r::dll⟨root, m⟩ ∧ n=m+1)
            inv n ≥ 0;

sdll⟨p,n,s⟩ == (root = null ∧ n = 0)
               ∨ (root::Node⟨s,r,p⟩ ∗ r::sdll⟨root,m,rs⟩ ∧ n=m+1 ∧ s ≤ rs)
               inv n ≥ 0;
```

Fig. 1. Predicate definitions for unsorted and sorted doubly-linked list

Figure 1 shows predicate definitions used by the example. Predicate `q::dll⟨p,n⟩` denotes the fact that `q` points to a doubly-linked list of length `n`; `q::sdll⟨p,n,s⟩` means `q` points to a *sorted* doubly-linked list of length `n`. `q` is actual argument for the implicit root parameter, denoted by `root` inside the definition. The first nodes of these lists has a `prev` field pointing to `p`. The `sdll` definition ensures that the list is sorted using the `s` parameter to check that values of subsequent list elements are greater than the value of the first element, where `s` is the value of the first element in the list. The specification of the predicate uses the connectives of classical logic such as $\wedge$, $\vee$ as well as the separating conjunction operator `*` which requires that its two arguments hold for two disjoint partitions of the heap [25]. In our system, a fresh variable, such as `r` in the definition of `dll` is implicitly existentially quantified. The underscore _ denotes a fresh variable whose name is omitted.

Figure 2 shows the Java code of our example along with specifications of preconditions and postcondition in separation logic with inductive definitions and numerical constraints. The `loadData` method loads a list from a file, sorts it, and returns the sorted list. Its postcondition ensures that the returned value is a sorted doubly-linked list. `loadData` ensures this condition by calling the `sort` procedure that accepts a

```
class Process {
  static Node loadData()
    requires emp
    ensures res::sdll⟨_,_,_⟩
  { Node l = getFromFile();
    Node sl = sort(l);
    return sl; }
  static Node sort(Node l)
    requires l::dll⟨_,n⟩
    ensures res::sdll⟨_,n,_⟩
```

```
{ if (l != null) {
      Node tmp = sort(l.next);
      tmp = insert(tmp, l);
      return tmp; }
   return l; }
static Node insert(Node l, Node v)
   requires l::sdll⟨p,n,s⟩ ∗ v::Node⟨vv,_,_⟩
   ensures (res::sdll⟨_,n+1,min(s,vv)⟩ ∧ l!=null)
      or (res::sdll⟨_,1,rs⟩ ∧ rs=vv ∧ l=null)
{ ... } }
```

Fig. 2. Annotated code for loading a list from a file and sorting it

doubly-linked list and returns a sorted list. The expectation is that `getFromFile` method will produce a doubly-linked list. However, `getFromFile` procedure in our example is not statically verified and we cannot guarantee statically that it will indeed produce a doubly-linked list structure expected by `sort`. In such a situation SLICK performs a runtime check to ensure that the data structure invariant holds. Consequently, we can still assume when reasoning about the body of `sort` that the data structure given is a doubly-linked list; and when reasoning about the body of `loadData` that the result returned by `sort` is a sorted list. When reasoning about callers of `loadData`, we can also make use of its postcondition.

Outline. In the rest of this paper we define our specification language and the desired semantics of runtime checks, we then describe the compile-time and runtime techniques that SLICK uses to generate the checks, discuss the issues in combining static and runtime checking and present preliminary experience with the system.

3 Specification Language

We designed our specification language for preconditions and postconditions to enable simultaneously runtime checking and static analysis [21], so it largely follows the syntax and semantics of languages in previous separation logic system.

Specification language syntax. Figure 3 shows the grammar for our specification language. Shape predicate `spred` is the main specification construct that provides data structure descriptions. Formulas are canonicalized to an internal representation akin to the superhomogeneous form [27], namely arguments for heap formulas are distinct and fresh. Additional existentially quantified variables are introduced if necessary to obtain the above form. The semantics of our specification language is included in Figure 8 in the Appendix.

Recursive shape predicate definitions need to satisfy certain syntactic restrictions, namely *well-formed* and *well-founded* conditions, to ensure soundness and termination of static reasoning [21]. *Well-formed* conditions ensure that shape predicates and formulas do not admit garbage (consequently, code generated for runtime checks can traverse the entire footprint of the formula). *Well-founded* conditions disallow `root` to be passed as argument to a recursive predicate invocation. That means `root` either is

$$
\begin{array}{rcl}
\text{spred} & ::= & c\langle (v\ [\mu])^* \rangle \equiv \Phi\ [\mathbf{inv}\ \pi_0] \\
\mu & ::= & @\mathbf{in} \mid @\mathbf{out} \\
\Phi & ::= & \bigvee \exists v^* \cdot (\kappa \wedge \pi) \\
\pi & ::= & \gamma \wedge \phi \\
\gamma & ::= & v_1 = v_2 \mid v = \texttt{null} \mid v_1 \neq v_2 \mid v \neq \texttt{null} \mid \gamma_1 \wedge \gamma_2 \\
\kappa & ::= & \mathbf{emp} \mid v{::}c\langle v^* \rangle \mid \kappa_1 * \kappa_2 \\
\phi & ::= & \text{arith} \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \neg\phi \mid \exists v \cdot \phi \mid \forall v \cdot \phi \\
\text{arith} & ::= & \mathsf{a}_1 = \mathsf{a}_2 \mid \mathsf{a}_1 \neq \mathsf{a}_2 \mid \mathsf{a}_1 < \mathsf{a}_2 \mid \mathsf{a}_1 \leq \mathsf{a}_2 \\
\mathsf{a} & ::= & k \mid v \mid k \times \mathsf{a} \mid \mathsf{a}_1 + \mathsf{a}_2 \mid -\mathsf{a} \mid \mathbf{max}(\mathsf{a}_1, \mathsf{a}_2) \mid \mathbf{min}(\mathsf{a}_1, \mathsf{a}_2) \\
k & \in & \text{Integer constants} \\
v, c & \in & \text{Identifiers}
\end{array}
$$

Fig. 3. Grammar for Shape Predicates

`null`, dangles, or points to an object. Well-foundedness ensures that the generated run-time checking code terminates when executed on any given heap, since every invocation of the generated code either fails/succeeds or recolors at least one object.

Predicate parameter modes. To make the execution of predicates at runtime more efficient, we assign *modes* to predicate parameters, following the approaches in logic programming [22, 27]. We currently support two modes: **in** and **out**. These modes can be inferred using a constraint-based analysis. In the current paper, we assume that the developer specifies mode annotations (implicitly or explicitly). For example, the parameters of the `dll` predicate can be annotated as $\texttt{dll}\langle \texttt{p}@\mathbf{out}, \texttt{n}@\mathbf{out}\rangle$. Both parameters `p` and `n` have **out** mode.

We use several conventions for default modes, which allows developers to omit most mode declarations in practice. Most of the parameters are **out**, so we make **out** the default mode. Next, a data structure is typically given as the set of objects obtained by traversing the data structure starting from the `root` node and terminating at either `null` or at some of the **in** parameters. `root` is therefore always an **in** parameter; the **out** parameters are values computed by traversing the data structures. SLICK considers method parameters as **in** parameters for their preconditions and postconditions. **out** parameters from preconditions are **in** parameters for corresponding postconditions.

4 Semantics of Run-Time Checking

In this section we present the semantics for run-time checking separation logic specifications and outline challenges in implementing this semantics. We then describe how we approach these challenges in our runtime checker.

4.1 Abstract Description of Run-Time Checks

The intended meaning of runtime checking is as follows. Given a stack s, an initial partial map L from logical variable names to values, and a heap h, we define the set of pairs (h_0, L_0) where h_0 is subheap of h and L_0 is partial map extending L such that formula is true for h_0, L_0:

$$\mathsf{submodelsFor}(s, h, L, \Phi) = \{(h_0, L_0) \mid (s \cup L_0), h_0 \models \Phi \wedge L \subseteq L_0 \wedge h_0 \subseteq h\}$$

A procedure with precondition Φ should succeed when $\Phi * \texttt{true}$ holds in the caller, which happens when $\mathsf{submodelsFor}(s, h, \emptyset, \Phi)$ is nonempty. Let h denote the current heap. Consider a procedure call of procedure f with precondition pre_f, body body_f, and postcondition post_f. Taking into account the usual semantics of logic variables that can relate pre- and postcondition, the execution of a procedure call with runtime checks is the following. Note that body_f may update the current heap h.

```
let $M = \mathsf{submodelsFor}(s, h, \emptyset, \mathit{pre}_f)$;        // subheaps satisfying precondition
if $M = \emptyset$ then error "Precondition failed";
let $(h_0, L) \in M$;                                  // pick subheap and logic var. bindings
let $h_1 = h \setminus h_0$;                                // save context
$h := h_0$;                                          // narrow heap to footprint
$\mathit{body}_f$;                                          // actual body of the method
let $M' = \mathsf{submodelsFor}(s, h, L, \mathit{post}_f)$;     // check post in current $h$,$L$
if $M' = \emptyset$ then error "Postcondition failed";
let $(h_R, \_) \in M'$;                                 // pick subheap to return
$h := h_R \cup h_1$;                                    // restore context
```

4.2 Separation Logic Runtime Checking Challenges

Given the semantics of separation logic formulas and the semantics of checks in Section 4.1, there are two main challenges in making runtime checking feasible. We next discuss the challenges specific to separation logic execution.

Evaluating spatial conjunction inside formulas. Consider first the problem of checking whether a given state satisfies a formula without numerical constraints. This model checking problem has been studied for first-order logic (with or without inductive definitions) [14] and, more recently, for separation logic [5]. Separation connective increases the complexity of the model checking problem because it essentially involves second-order quantification [17]. In general it is not clear how to split into two parts each of which satisfies the corresponding conjunct, so each separation logic formula could in principle admit an exponential number of sets of locations that denote its footprint.

Approach: marking the footprint. Our approach stems from the observation that, in practice, data structure specifications often contain formulas that have a small number of possible footprints that can be computed while evaluating the formula. Moreover, separation logic connective does not appear under a negation in our system. Therefore, instead of maintaining an explicit container containing objects in the footprint, we mark objects that participate in the footprint of the formula. An attempt to mark an object twice makes the entire formula disjunct unsatisfiable.

Representing method footprints. A naïve implementation of the semantics in Section 4.1 would associate with each method invocation a set of references that covers the method's footprint. For a call stack of depth n, it would need n copies of these footprints to maintain the information about all contexts h_1 for procedures on the call stack. In the worst case this would cause an n-fold increase in memory consumption. Next, we need a mechanism to adjust the heap h for each procedure call and check each individual field read or write, to ensure that they perform operations only on the current footprint.

Approach: maintaining marking across procedure calls. When a precondition succeeds, our system retains the marking of nodes, which is unique for a procedure invocation. Reads, writes and procedure calls check the marking and adjust it accordingly. Postcondition check restores the marking.

5 The Runtime Engine

We now present in more detail the runtime mechanisms of our checker. SLICK augments each object with a field named `color`, which indicates the object's availability to different method invocations. The color of an object may change during program execution. Each method invocation is also associated with a unique color, maintained on a global stack. A method invocation can access an object if and only if their colors match. Newly allocated objects belong to the current method invocation's footprint; the objects receive the color of the current invocation via instrumented object constructors. An invocation of method m is permitted if the footprint F of m's precondition is a subset of the caller's footprint at the call site. In that case, the system colors the footprint F to match the color of the invocation of m. A return from invocation of m is permitted if the footprint F' of the postcondition of m is a subset of the current execution footprint at the end of m. The system then recolors the postcondition footprint F' to the color of the caller.

Checking formulas. Runtime checking formulas consists in verifying the formula footprint and computing **out** parameters. SLICK translates each formula to executable code in the form of a class with a method `traverse` that, when executed, traverses the footprint of the formula in the current heap. `traverse` accepts two input parameters, `curColor` and `newColor` and returns **boolean**. `traverse` recolors each object it visits to `newColor` if the current color of the object is `curColor`. If `traverse` succeeds in recoloring all visited objects and all pure constraints are also satisfied, it sets **out** parameters and returns **true**. Otherwise it fails.

Checking formulas with disjunction. The recursive definition of predicates such as `dll` and `sdll` contain the disjunction operator to differentiate the base case and the recursive case of the definition. When evaluating the truth of a pure classical logic formula $F_1 \vee F_2$ in a given heap, it is possible to simply evaluate F_1 first, and, if it fails, proceed with the evaluation of F_2. In the case of our separation logic formulas, however, evaluation changes the coloring of the heap. Therefore, if the evaluation of F_1 fails, SLICK must undo the coloring performed by F_1. Based on the recursive predicates we have examined, we expect the failure of false disjuncts to occur quickly. SLICK therefore undoes the coloring by re-executing the evaluation of F_1 with opposite color parameters. This approach avoids additional bookkeeping that would be required to maintain the set of marked objects. In our example of `dll` and `sdll`, the footprint of the first disjunct is empty, which means that its execution performs no marking and there is nothing to undo.

Computing bindings for existential quantifiers. Existentially quantified variables in program specifications are often either determined by variables in program state, or they do not affect the truth value of the formula at all. Consider, for example, the precondition of `sort`, given by the formula $\texttt{l::dll}\langle\texttt{p,n}\rangle$. The root parameter of `dll` predicate is bound to the value of the local variable `l`. The `n` parameter, on the other hand, is existentially quantified, but is given as the length of the list. The `p` parameter of `dll` is given as the `prev` field of the first node whenever the list is non-empty. When the list is empty, the `p` parameter is left unconstrained, but the truth value of `dll` does not depend on it either. Therefore, the value of `p` is either given by the context where `dll` is called, as in the recursive invocation inside `dll` definition, or it is not used anywhere, as in the precondition of `sort`. SLICK uses mode analysis, described in Section 6, to determine how to compute values of such existentially quantified variables.

Precondition. SLICK invokes precondition checking code in the caller prior to method invocation. If a precondition check succeeds, it also provides values for the **out** parameters of the formula. These values can then be used by the postcondition of the same invocation. Note that pre- and postcondition checks are performed in the caller to facilitate integration with the static verifier. More details are provided in section 7.

As an illustration, consider the `sort` method from Figure 2. Figure 4 shows the runtime checking code that SLICK generates for `sort`. SLICK compiles the precondition to a class with fields to store all free logic variables of the formula (in this case, variables `l` and `n`). In callers of `sort`, SLICK also generates instructions to create an instance of the generated class (the checker object), initialize the **in** parameter (`l`) and then invoke `traverse` on the initialized checker object. `traverse` receives two colors as arguments: the current method invocation's color is passed to `curColor`, a freshly generated color to `newColor`. Upon successful completion of `traverse`, SLICK sets `n` to the length of the list. SLICK stores a reference to the checker object in a local variable that is visible to the code that verifies the postcondition.

Postcondition. When a method returns, SLICK checks postcondition against the current method's footprint. SLICK then makes the objects covered by the postcondition accessible to the caller. As an example, Figure 5 shows the translation of the postcondition of `sort`, whose internal representation is $\exists \texttt{r}_1 \cdot \texttt{res}::\mathrm{SDLL}\langle\texttt{r}_1\rangle \wedge \texttt{r}_1 = \texttt{n}$.

Note that it is possible that the postcondition does not cover all objects of the current invocation's footprint. The uncovered objects, even if reachable from the caller, are not accessible under separation logic semantics. The use of coloring in SLICK correctly enforces this semantics. Indeed, observe that any objects in the footprint of the returning method, if not covered by the postcondition thereof, will retain the color of the returning method invocation. This color is unique for the dynamic method invocation, so no current or future method invocations will be able to access these objects.

Unannotated code. When a method has no annotations, as is the case of `getFromFile` in Figure 2, both precondition and postcondition are **true**. This means that the footprint of the precondition is the same as the caller's current footprint and that the entire footprint of the callee is returned to the caller. SLICK thus executes the callee

```
class sort_pre { Node l; int n;
    boolean traverse(color curColor,
        color newColor) { ... }
}
Node loadData() {
    Node l = getFromFile();
/// generated code
    sort_pre prchk = new sort_pre();
    prchk.l = l;
    SLICK.pushCurrentColor();
    SLICK.setCurrentColor(
        SLICK.freshColor());
    prchk.traverse(SLICK.topColor(),
        SLICK.currentColor());
/// end of generated code
    Node sl = sort(l);
    ...
```

Fig. 4. Compiled precondition of `sort`

```
class sort_post {
    Node res;
    int n;
    boolean traverse(...)
}
Node loadData() {
    ...
    Node sl = sort(l);
/// generated code
    sort_post pockr = new sort_post();
    pockr.res = sl;
    pockr.n = prchk.n;
    color c = SLICK.popColor();
    pockr.traverse(SLICK.currentColor(), c);
    SLICK.setCurrentColor(c);
/// end of generated code
    return sl; }
```

Fig. 5. Compiled postcondition of `sort`

without any recoloring of the heap and with the callee invocation having the same color as the caller invocation.

6 From Separation Logic to Executable Code

We now present our translation from separation logic formula to executable code. The basic idea is to compile a separation logic formula into a function that checks if a given program state (s, h) is a model of the formula. The translation consists of mode analysis and Java code generation. Besides checking that the formula holds in the current program state, the translated code recolors the formula's footprint and computes the values of **out** parameters. Each formula is translated to a class with a method `traverse` and fields representing the free variables of the formula. The fields have the same names as the free variables they represent. Fields for **in** parameters need to be initialized before each invocation of `traverse`; fields for **out** parameters are set by `traverse` upon successful completion of checking.

Mode analysis. At compile time, variables in a formula are classified into two main groups: bound and unbound. Initially, unbound variables include **out** parameters and existentially quantified variables of the present formula. Bound variables include **in** parameters of the present formula and **out** arguments of recursive predicate invocations. If an **out** argument is not unified with a value in all disjuncts of a predicate definition, we further classify it as *conditionally bound*.

Conditionally bound variables use a boxed representation of their underlying types. Each boxed value has a flag indicating whether the underlying value is bound. The first time when the compiled formula uses a conditionally bound variable v at runtime, it binds v to a concrete value. When v is used in an equality $v = t$ and the value of term t is known, v is bound to t; otherwise both v and t are bound to the same value by instantiating unbound variables in t. If used in a disequality or inequality, v is bound to a random value such that the constraint holds. This treatment is incomplete, but sound.

The translation consists of two passes. The first pass determines subformulas that generate bindings for the unbound variables. The second one compiles the selected subformulas to assignments and the rest of the formulas to tests. To make it easier to read the formalization, the following names have dedicated meanings in our rules. $vmap$ is the binding map of unbound variables. $vmap$ also keeps track of which variables and terms are conditionally bound to help the code generator to invoke correct operations on these values. ins and $outs$ are **in** and **out** parameter sets, respectively. $\mathit{INS}(c)$ returns all the **in** parameters of predicate c. $uvars$ is the set of unbound variables. Function *UVAR* returns the set of unbound variables of a term. Note that ins and $outs$ are the same for all disjuncts of a formula, whereas $vmap$ and $uvars$ are computed anew for each disjunct. $||\ C\ ||$ marks C as executable code emitted by the compilation.

The first pass computes a mapping from unbound variables to terms, where a term can be either constant, variable, field access, or combination of terms using arithmetic operations. This pass also produces a partial ordering, which determines the order in which assignments are generated by means of a topological sort. There are three sources of bindings for unbound variables, namely i) **in** parameters of the present formula, ii) **out** parameters of predicate invocations, and iii) object fields. The computation is formalized as the genMap function in Figure 9 in the Appendix. As genMap generates the bindings, it also removes from the input formula all unifications $v = t$ that it uses in bindings generation.

Translation of disjunction. SLICK compiles a formula $F_1 \vee \ldots \vee F_n$ in disjunctive normal form as follows:

```
boolean traverse(color curColor, color newColor) {
    boolean r_1 = disj1(curColor, newColor);
    if (r_1) return true;
    disj1(newColor, curColor);
    ...
    boolean r_n = disjn(curColor, newColor);
    if (r_n) return true;
    disjn(newColor, curColor);

    return false; }
```

Translation of conjunction. SLICK compiles a formula $F_i = \exists v^* \cdot \kappa \wedge \pi$ into a function `boolean disji(color curColor, color newColor)`. Figure 6 formalizes the compilation of the body of `disji` as a function that takes a formula and emits executable code.

The translation also makes use of the following functions. The genInitialization function emits assignments to initialize **in** parameters of the formula, subject to the constraint that all **in** parameters must be initialized.

$$\begin{array}{l}\textsf{genInitialization}\ \ p{::}c\langle v_i^*\rangle \overset{\text{def}}{=} \\ \quad \textbf{foreach}\ \ f_i\ \ \textbf{in}\ \ \mathit{INS}(c)\ \ \textbf{do}:\ \ ||\ p.f_i = ||\ \textsf{genBinding}\ \ v_i\end{array}$$

The genAssign function emits assignments to **out** parameters of the predicate. If a variable does not have a binding from the formula, it is assigned an unbound boxed

```
$TR[[p{::}c\langle v^*\rangle]] \mid IsObj(c) \overset{\text{def}}{=}$
  $||$ if $p \neq$ null $\wedge\ curColor = p.color$
    then $p.color = newColor$
    else return false; $||$

$TR[[p{::}c\langle v^*\rangle]] \mid IsPred(c) \overset{\text{def}}{=}$
  $||\ p =$ new $c\_Checker;\ ||$
  genInitialization $p{::}c\langle v_i^*\rangle$;
  $||$ if not$(p.traverse(curColor, newColor))$
    then return false; $||$

$TR[[\kappa_1 * \kappa_2]] \overset{\text{def}}{=} TR[[\kappa_1]]; TR[[\kappa_2]]$

$TR[[\exists v^* \cdot \kappa \wedge \pi]] \overset{\text{def}}{=}$
  let $uvars = v^* \cup outs$ in
  let $\pi' =$ genMap $(\kappa \wedge \pi)$ in
    $TR[[\kappa]]$;
    $||$ if $||\ TR[[\pi']]\ ||$ then $||$
        genAssign;
        $||$ return true; $||$
    $||$ else return false; $||$

$TR[[p = t]] \mid p$ is conditionally bound, $t$ is bound $\overset{\text{def}}{=}\ ||\ p.EQ(t)\ ||$
```

Fig. 6. Translation Rules

value.

```
genAssign $\overset{\text{def}}{=}$
  foreach $p$ in $outs$ do :
      $||\ p =||$ genBinding $p$
      if genBinding $failed$ then $||\ p =$ new (boxedtypeof(p)) $||$
```

The genBinding function computes the closure of the bindings to get bound terms.

```
genBinding $v \overset{\text{def}}{=}$
    if $v \notin uvars$ then $||\ v\ ||$
    else genBinding (lookUp $v$ $vmap$)
```

If the first argument is a term, genBindings performs the obvious recursion on the structure of the term and emits a term with identical structure, except for the translated variables. If lookUp fails to find an entry for an unbound variable, genBinding fails.

7 Integrating Static and Runtime Verification

In this section we discuss the integration of static and runtime verification. The general idea is that assertions that can be statically verified need not be checked at runtime. However, such combination is more difficult for analysis domains based on spatial conjunction of facts than for analysis domains based on classical conjunction of facts. Indeed, to ensure that assertion $F_1 \wedge F_2$ holds after a given program point, it is possible to ensure F_1 statically and then check F_2 dynamically. On the other hand, given assertion $F_1 * F_2$, it is necessary to communicate to both the run-time and the static time checker the footprints of individual formulas in order to enable separation of these two checks. In the sequel, we describe optimizations that are nevertheless possible in our runtime checking approach; more fine-grained combinations are possible but beyond the scope of the current paper.

Field access. If the static verifier proves a field access safe, no runtime check is required. This is because field access does not affect the coloring of the objects or method invocations. On the other hand, if the static verifier fails to verify a field read, it emits runtime check for the pointer and continues with a suitably modified symbolic state.

$$\frac{\varDelta \nvdash x{::}c\langle f^*\rangle}{\vdash \{\varDelta\} v = x.f \{\exists v \cdot \varDelta\}}$$

If it fails to verify a field write, it stops static verification and emits runtime check for all subsequent code. As an optimization, once a field access has been issued a runtime check, it needs not be checked again until the pointer itself or its color may have changed. In many cases this information can be obtained statically.
Method contract. Method contract checks, on the contrary, cannot be as readily eliminated since they change the heap coloring. Let us consider a method `g` that calls another method `f` with precondition pre_f and postcondition $post_f$:

```
void g()                        void f()
  { g_1; f(); g_2; }              requires pre_f ensures post_f { ... }
```

There are the following possibilities:

1. `f` is statically verified.
   – pre_f is statically proved: if the part g_2 of `g` following the call to `f` is statically verified by assuming $post_f$, `g` need not emit runtime checks for pre_f and $post_f$. Otherwise, as g_2 may attempt to access objects that do not belong to $post_f$'s footprint, runtime checks for pre_f and $post_f$ (and certainly for g_2) are needed.
   – pre_f is not statically proved: `g` issues runtime checks for pre_f and $post_f$. Static verification of g_2 can assume $post_f$.
2. `f` is not statically verified: `g` issues runtime checks for pre_f and $post_f$. Static verification of g_2 can assume $post_f$.

The static verifier can take advantage of the fact that after a method call, the callee's postcondition holds. Even if it cannot verify the callee's precondition, it can still assume the postcondition, and continues static verification after issuing appropriate runtime checks. When the precondition is a pure formula, static verification proceeds as follows:

$$\frac{\varDelta \nvdash pre(mn) \qquad IsPure(pre(mn))}{\vdash \{\varDelta\} mn(v^*) \{(\varDelta \wedge pre(mn)) * post(mn)\}}$$

On the other hand, if the precondition has a nonempty heap component, the static verifier assumes the postcondition as the current program state. Note that we cannot simply $*$-conjoin the postcondition with the current program state, as they may cover overlapping footprints. Replacing the entire program state by the postcondition is sound, but may result in loss of precision if the callee's postcondition covers only parts of data structures.

$$\frac{\varDelta \nvdash pre(mn) \qquad HasHeap(pre(mn))}{\vdash \{\varDelta\} mn(v^*) \{post(mn)\}}$$

Integration in the example. In the example of section 2, `sort` and `insert` are both statically verifiable. `loadData` fails to verify the precondition of `sort` because the information is simply not available, so it emits runtime check, but by assuming postcondition of `sort`, the postcondition of `loadData` can be statically verified, a fact that callers of `loadData` can exploit. Note that the runtime checking is localized within `loadData` only, so the overhead is small.

8 Implementation

We implemented SLICK in the context of a system for checking data structure properties [21]. We report our experience with the system on several examples.

Memory overhead. Memory overhead consists of one field per object to store the object's color and a single stack of live colors which has the same height as the program call stack. Since the `color` type can be implemented as **long**, memory overhead decreases if the program uses larger objects. `traverse` method also creates a number of intermediate objects, but they exist only during the formula traversal and do not permanently accumulate in the memory overhead of the code instrumented with runtime check. Consequently, we were not able to measure any significant difference in memory consumption on our examples.

Runtime overhead. We evaluate the runtime overhead of our approach by running experiments with different levels of runtime checking: no runtime checking, all operations are runtime checked, all field accesses are runtime checked, and checking at boundaries of data structure operations. In the third case, the entire program runs with a single color, hence no precondition or postcondition check is performed. This case measures the overhead of checking field accesses. In the last case, SLICK checks only the first precondition and the last postcondition of a data structure operation at runtime since the static verifier can assert that checks for recursive calls and field accesses are statically safe. This case simulates a scenario where these data structures are used in conjunction with unverified or untrusted inputs. In order to minimize the timing effects of class loading and JIT compilation, we repeat the experiments and ignore the timings of the first two runs.

Timings for the experiments, measured with JVM 1.5 on Linux 2.6 running on a PC having a 3GHz CPU and 2GB RAM, are reported in Figure 7. The data structures used in our experiments have sizes ranging from 1000 to 5000 elements. The first experiment sorts a list using insertion sort. The "Full" check for `sort` causes very large increases in running time. However, the "Boundary" version, which we expect to be used in practice, causes insignificant increases since the data structure is traversed only two more times. The second example performs an in order traversal of a binary search tree to produce a sorted list. The "Full" check incurs large overhead since it forces the entire subtree to be traversed at each recursive invocation. The other two checks are significantly cheaper. The third example performs the following two operations 1000 times: inserting a random element to and deleting the maximum element from a priority queue. The "Native" and "Field" timings reflect the logarithmic complexity of operations on priority queues. The "Full" and "Boundary" timings are linear in data structure size as expected, since every `insert` and `deletemax` operation traverses the entire heap, rather than just a path with logarithmic length from root to leaf. The fourth example is a popular operation in data mining algorithms. It traverses a table containing the iterative patterns used in software specification mining and calculates the support of a mined pattern [20]. The operation is repeated 10 times. Note that the computation of support itself does not need to traverse the entire table, since the table provides caching of most of the subcomputations. Precondition and postcondition checking therefore causes a significantly larger number of objects to be visited, causing the large increase in running time. A common property across all the examples is that "Field" check timings show that the overhead of checking every heap access in SLICK is small.

	Insertion Sort				Binary Search Tree			
Size	Nat.	Full	Field	Bdry.	Nat.	Full	Field	Bdry.
1,000	6	49,235	10	7	0.03	181	0.06	0.93
2,000	28	>50,000	44	31	0.07	866	0.12	4.50
3,000	69	>50,000	108	81	0.11	2,253	0.18	10.45
4,000	127	>50,000	183	135	0.14	4,965	0.24	8.62
5,000	209	>50,000	296	211	0.18	9,360	0.30	9.07

	Priority Queue				Support Calculation			
Size	Nat.	Full	Field	Bdry.	Nat.	Full	Field	Bdry.
1,000	0.93	2,585	1.62	765	0.22	12,205	0.30	25
2,000	0.99	5,171	2.68	1,521	0.45	>50,000	0.63	61
3,000	1.02	7,767	1.79	2,321	0.68	>50,000	0.94	111
4,000	1.01	10,320	2.69	3,032	0.93	>50,000	1.40	169
5,000	1.03	13,070	1.89	3,827	1.18	>50,000	1.73	173

Fig. 7. Performance Measurements (in milliseconds)

9 Conclusion

We presented SLICK, the first runtime checker for separation logic program specifications. We have identified several challanges that make separation logic specification seemingly more difficult to check at run time than for classical logic. However, we believe that many of these problems would occur in any systems that precisely checks frame conditions of procedures. The notable features of SLICK include runtime mechanism that avoids memory blow up and a compilation of separation logic specification to executable code that runs natively on the JVM. Overall, the run-time checking cost can be significant for large data structure instances when all intermediate states are checked, but even in those cases the absolute performance is sufficiently good for debugging the code and the specifications. Performing only "boundary checks" is an appealing alternative to all intermediate checks: because specifications capture operation footprint, boundary checks ensure data structure consistency at the end of an operation regardless of the internal behavior of the operation. In some cases (such as the insertion sort example), the overhead when performing only boundary checks appears acceptable even for deployed applications. Preliminary results demonstrate that running time can be significantly reduced using static verification to remove majority of runtime checks.

References

1. Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In *CASSIS: Int. Workshop on Construction and Analysis of Safe, Secure and Interoperable Smart devices*, 2004.
2. Josh Berdine, Cristiano Calcagno, Byron Cook, Dino Distefano, Peter O'Hearn, Thomas Wies, and Hongseok Yang. Shape analysis for composite data structures. In *CAV*, 2007.
3. Eric Bodden, Laurie Hendren, and Ondřej Lhoták. A staged static program analysis to improve the performance of runtime monitoring. In *ECOOP*, 2007.
4. C. Calcagno, D. Distefano, P.W. O'Hearn, and H Yang. Footprint analysis: A shape analysis that discovers preconditions. In *SAS*, 2007.
5. Cristiano Calcagno, Hongseok Yang, and Peter O'Hearn. Computability and complexity results for a spatial assertion language for data structures. In *FSTTCS*, 2001.
6. Robert Cartwright and Mike Fagan. Soft typing. In *PLDI '91: Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, pages 278–292, 1991.
7. Feng Chen and Grigore Roşu. MOP: An Efficient and Generic Runtime Verification Framework. In *Object-Oriented Programming, Systems, Languages and Applications(OOPSLA'07)*, 2007.
8. Yoonsik Cheon. *A Runtime Assertion Checker for the Java Modeling Language*. PhD thesis, Iowa State University, April 2003.
9. Lori A. Clarke and David S. Rosenblum. A historical perspective on runtime assertion checking in software development. *SIGSOFT Softw. Eng. Notes*, 31(3):25–37, 2006.
10. Brian Demsky, Cristian Cadar, Daniel Roy, and Martin C. Rinard. Efficient specification-assisted error localization. In *Second International Workshop on Dynamic Analysis*, 2004.
11. Robert Bruce Findler and Matthias Felleisen. Contracts for higher-order functions. In *Proc. 2002 International Conference on Functional Programming*, 2002.
12. Bolei Guo, Neil Vachharajani, and David I. August. Shape analysis with inductive recursion synthesis. In *PLDI*, 2007.
13. John Guttag and James Horning. *Larch: Languages and Tools for Formal Specification*. Springer-Verlag, 1993.
14. Neil Immerman. *Descriptive Complexity*. Springer-Verlag, 1998.
15. Samin Ishtiaq and Peter W. O'Hearn. BI as an assertion language for mutable data structures. In *Proc. 28th ACM POPL*, 2001.
16. Viktor Kuncak. *Modular Data Structure Verification*. PhD thesis, EECS Department, Massachusetts Institute of Technology, February 2007.
17. Viktor Kuncak and Martin Rinard. On spatial conjunction as second-order logic. Technical Report 970, MIT CSAIL, October 2004.
18. Tal Lev-Ami. TVLA: A framework for Kleene based logic static analyses. Master's thesis, Tel-Aviv University, Israel, 2000.
19. Tal Lev-Ami, Thomas Reps, Mooly Sagiv, and Reinhard Wilhelm. Putting static analysis to work for verification: A case study. In *Int. Symp. Software Testing and Analysis*, 2000.
20. D. Lo, S-C. Khoo, and C. Liu. Efficient mining of iterative patterns for software specification discovery. In *Proc. of SIGKDD Int. Conf. on Knowledge Discovery and Data Mining*, 2007.
21. Huu Hai Nguyen, Cristina David, Shengchao Qin, and Wei-Ngan Chin. Automated verification of shape, size and bag properties via separation logic. In *VMCAI*, 2007.
22. David Overton, Zoltan Somogyi, and Peter J. Stuckey. Constraint-based mode analysis of mercury. In *PPDP '02: Proceedings of the 4th ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 109–120, New York, NY, USA, 2002. ACM Press.

23. Frances Perry, Limin Jia, and David Walker. Expressing heap-shape contracts in linear logic. In *GPCE '06: Proceedings of the 5th international conference on Generative programming and component engineering*, pages 101–110, New York, NY, USA, 2006. ACM Press.
24. Jan Reineke. Shape analysis of sets. Master's thesis, Universität des Saarlandes, Germany, June 2005.
25. John C. Reynolds. Separation logic: a logic for shared mutable data structures. In *17th LICS*, pages 55–74, 2002.
26. Ajeet Shankar and Rastislav Bodik. Ditto: Automatic incrementalization of data structure invariant checks. In *PLDI*, 2007.
27. Zoltan Somogyi. A system of precise modes for logic programs. In *ICLP*, 1987.
28. Thomas Wies, Viktor Kuncak, Patrick Lam, Andreas Podelski, and Martin Rinard. Field constraint analysis. In *Proc. Int. Conf. Verification, Model Checking, and Abstract Interpretation*, 2006.
29. Karen Zee, Viktor Kuncak, Michael B. Taylor, and Martin Rinard. Runtime checking for program verification systems. In *Workshop on Workshop on Runtime Verification (collocated with AOSD)*, 2007.

A Specification Language Semantics

Figure 8 shows the semantics of our specification language. The model of a formula consists of a stack s and a heap h. As usual, a heap h is a partial function from memory addresses to values, but we additionally require that, for every object, either all or none of the fields of that object are in the domain of h (therefore, our semantics does not split the fields of the same objects across multiple conjuncts). If $s, h \models \Phi$, we call the domain of h a *footprint* of the formula Φ. The semantics definition also uses some additional notations. *dom*(h) returns the domain of the partial function h. $h_1 \bot h_2$ holds if $dom(h_1) \cap dom(h_2) = \emptyset$. $h_1 \cdot h_2$ denotes the union of two domain-disjoint functions h_1, h_2; it is undefined otherwise. $IsObj(c)$ and $IsPred(c)$ returns **true** if c is the name of a class or a shape predicate, respectively. $s \models_A \phi$ is the usual interpretation of arithmetic formulas.

$$\begin{array}{lll}
s, h \models \Phi_1 \vee \Phi_2 & \text{iff} & s, h \models \Phi_1 \text{ or } s, h \models \Phi_2 \\
s, h \models \exists v^* \cdot \kappa \wedge \pi & \text{iff} & \exists \nu^* \cdot s[v^* \mapsto \nu^*], h \models \kappa \text{ and } s[v^* \mapsto \nu^*] \models \pi \\
s, h \models \kappa_1 * \kappa_2 & \text{iff} & \exists h_1, h_2 \cdot h_1 \bot h_2 \text{ and } h = h_1 \cdot h_2 \text{ and} \\
 & & s, h_1 \models \kappa_1 \text{ and } s, h_2 \models \kappa_2 \\
s, h \models \texttt{emp} & \text{iff} & dom(h) = \emptyset \\
s, h \models p{::}c\langle v_{1..n} \rangle & \text{iff} & IsObj(c) \text{ and } s(p) > 0 \text{ and } h = [s(p) \mapsto r] \\
 & & \text{and } r = c[f_1 \mapsto s(v_1), .., f_n \mapsto s(v_n)] \\
 & & \text{or } IsPred(c) \text{ and } s, h \models [p/\texttt{root}]\Phi \\
s \models p_1 \oslash p_2 & \text{iff} & s(p_1) \oslash s(p_2), \text{ where } \oslash \in \{=, \neq\} \\
s \models p \oslash \texttt{null} & \text{iff} & s(p) \oslash 0, \text{ where } \oslash \in \{=, \neq\} \\
s \models \pi_1 \wedge \pi_2 & \text{iff} & s \models \pi_1 \text{ and } s \models \pi_2 \\
s \models \phi & \text{iff} & s \models_A \phi
\end{array}$$

Fig. 8. Semantics of Specification Language

B Translation Example

The generated class for predicate `dll` is the following. We will show the body of `traverse` subsequently.

```
class dll_Checker {
    Node root;
    NodeBoxed p;
    int n;
    boolean traverse(color curColor, color newColor) ... }
class NodeBoxed { boolean bound = false; Node val; ... }
```

Before invoking method `traverse` of a checker object `chk` of type `dll_Checker`, we need to initialize field `chk.root`. If `traverse` returns **true**, then `chk.n` and `chk.p` can be used. Note that `p` is declared with (generated) boxed type `NodeBoxed`, which has an additional field `bound` to indicate whether `traverse` has set the value of `p`. This provision is needed only for **out** parameters that are not bound in all disjuncts of the formula.

We now describe the compilation of separation logic formula to executable code. Each disjunct of a disjunctive formula is compiled to a separate method of the checker class. The idea is that `traverse` tries to call each of these methods to check and re-color the heap. If the check succeeds, `traverse` returns **true**. Otherwise it undoes the coloring by calling the same function again, but with the two parameter colors swapped.

The first disjunct of `dll`, $\texttt{root} = \texttt{null} \wedge \texttt{n} = 0$, is compiled to:

```
    boolean disj1(color curColor, color newColor) {
        if (root == null) {
            n = 0;
            p = new NodeBoxed();
            return true; }
        else
            return false; }
```

Note that as `root` is an **in** parameter, the translation compiles the formula `root = null` to a test. `n = 0`, on the other hand, is compiled to an assignment, since `n` is **out**. Since parameter `p` is not provided with a binding by the disjunct, it is set to an unbound value. We will need to take this into account when compiling the recursive branch.

The second disjunct of the `dll` predicate:

$$\exists \texttt{r}_1, \texttt{r}_2, \texttt{r}_3, \texttt{r}_4, \texttt{r}_5 \cdot \texttt{root}::\texttt{Node}\langle \texttt{r}_5, \texttt{r}_1, \texttt{r}_2 \rangle * \texttt{r}_1::\texttt{dll}\langle \texttt{r}_3, \texttt{r}_4 \rangle$$
$$\wedge\ \texttt{n} = \texttt{r}_4 + 1 \wedge \texttt{r}_2 = \texttt{p} \wedge \texttt{r}_3 = \texttt{root}$$

is compiled to:

```
    boolean disj2(color curColor, color newColor) {
        if (root != null && root.color == curColor)
            root.color = newColor;
        else
            return false;

        dll_Checker r1 = new dll_Checker();
```

```
8          r1.root = root.next;
9          if (!r1.traverse(curColor, newColor))
10             return false;
11         if (r1.p.EQ(root)) {
12             n = r1.n + 1;
13             p = new NodeBoxed(root.prev);
14             return true;
15         }
16         else
17             return false; }
```

```
class NodeBoxed { ...
    boolean EQ(Node p) {
        if (this.bound) return this.val == p;
        this.val = p;
        this.bound = true;
        return true;
    } }
```

Let us explain how `disj2` works. The test at line `2` checks if the object has the same color as the current color (normally the color of the current method invocation). If the colors match, which means the object is accessible to the current method invocation, then it is recolored, effectively made available to the target method invocation. Lines `7` and `8` set up an instance of the checker class for the recursive invocation of the `dll` predicate with root pointer `r1`. The field representing **in** parameter is intialized prior to the invocation of `traverse` at line `9`. If the recursive traversal succeeds, then the pure test is performed at line `11` and output parameters `n` and `p` computed. Since the object referenced by `root` has been re-colored, any sharing in the list would be detected when `traverse` visits the same location the second time.

We can now complete method `traverse` of class `dll_Checker`.

```
class dll_Checker {
    boolean traverse(color curColor, color newColor) {
        boolean r1 = disj1(curColor, newColor);
        if (r1) return true;
        disj1(newColor, curColor);

        boolean r2 = disj2(curColor, newColor);
        if (r2) return true;
        disj2(newColor, curColor);

        return false;
    }
    boolean disj1(color curColor, color newColor)
    boolean disj2(color curColor, color newColor)
}
```

`traverse` calls the methods of the disjuncts in sequence. The first disjunct that returns true will be taken and the rest ignored. In case a disjunct fails, `traverse` undoes the coloring by calling the disjunct method again with the colors swapped. The reason for this arrangement is to exploit the common case in predicate definitions. Most data structures are defined such that we do not need to traverse very deep down the heap to figure out that a case is successful or not. Hence actual undoing of the coloring rarely incurs substantial overhead.

C Binding Map Generation

The generation of bindings map is formalized in Figure 9.

```
genMap $p{::}c\langle v_i^* \rangle \;\mid\; IsObj(c) \overset{\text{def}}{=}$
    if $v_i \in uvars$ then $vmap := vmap[v_i \mapsto p.f_i]$;
        $f_i$ is the $i^{th}$ field of $c$.
    return $(p{::}c\langle v_i^* \rangle, \{(p, v_i) \mid v_i \in uvars\})$
genMap $p{::}c\langle v_i^* \rangle \;\mid\; IsPred(c) \overset{\text{def}}{=}$
    if $v_i \in uvars \wedge \text{mode}(c, i) = \textbf{out}$ then $vmap := vmap[v_i \mapsto p.p_i]$;
        $p_i$ is the $i^{th}$ parameter of $c$.
    return $(p{::}c\langle v_i^* \rangle, \{(p, v_i) \mid \text{mode}(c, i) = \textbf{out}\})$
genMap $v = t \overset{\text{def}}{=}$
    if $v \in uvars$ then
      $vmap := vmap[v \mapsto t]$;
      return $(\textbf{true}, \{(v_t, v) \mid v_t \in UVAR(t)\})$;
    else return $(v = t, \{\})$;
```

Fig. 9. Mapping generation

Note that t in the third case can be an unbound variable or a term containing unbound variables. The genBinding function computes the closure of the bindings to obtain the correct bound term. $\|\ C\ \|$ marks C as executable code emitted by the compilation.

```
genBinding $v \overset{\text{def}}{=}$
    if $v \notin uvars$ then $\|\ v\ \|$
    else genBinding (lookUp $v$ $vmap$)
```

If the first argument is a term, genBindings performs the obvious recursion on the structure of the term and emits a term with identical structure, except for the translated variables. If lookUp fails to find an entry for an unbound variable, genBinding fails.

There are very comprehensive mode systems and analyses, such as [22]. One limitation of these works is the requirement that if a disjunction produces a variable, then all disjuncts must produce that variable, thereby excluding the possibility of conditionally bound parameters. Such a mode system would have problems handling many of the data structure definitions. We therefore design an approach that is both simple and capable of handling most of the commonly encountered data structures.