Generalized Typestate Checking as Set Program Analysis

Patrick Lam and Martin Rinard
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139
{plam, rinard}@lcs.mit.edu

ABSTRACT

We present a generalization of standard typestate systems in which the typestate of each object is determined by its membership in a collection of abstract typestate sets. This generalization supports typestates that model participation in abstract data types, composite typestates that correspond to membership in multiple sets, and hierarchical typestates. Because membership in typestate sets corresponds directly to participation in data structures, our typestate system completely characterizes global sharing patterns.

Our typestate checking algorithm operates on set programs; the basic operations in these programs insert and remove objects to and from typestate sets. Set programs are derived by applying translators to modules written in languages that use standard constructs. Our framework supports the use of arbitrary translators that use arbitrarily sophisticated reasoning. It is therefore possible to use arbitrarily complex data structures without compromising the safety of the typestate checker.

We present our typestate checking algorithm and prove that if the set program typestate checks, then its execution will never violate the typestate declarations. We also state the correctness conditions that all translators must obey to ensure that the typestate checking result for set programs extends to programs written in standard languages with object fields and references.

1. INTRODUCTION

Typestate systems allow the type of an object to change during its lifetime in the computation, enabling the typestate system to enforce safety properties that depend on changing object states of the objects and increasing the precision of the abstractions in the type system (in standard type systems, an object's type never changes). The standard typestate approach assigns, at each program point, a single state to each object. The states change in respose to program actions [23].

This paper presents a new formulation of typestate systems. Instead of associating a single state with each object, our system instead models each typestate as an abstract set of objects. If an object is in a given typestate, it is a member of the set that corresponds to that typestate. This formulation immediately suggests the following generalizations of the standard typestate approach:

- **Abstract Data Types:** For typestate purposes, abstract data types can be viewed as maintaining several abstract sets of objects. For example, a tree abstract data type maintains the set of objects in the tree, while a map maintains the set of objects in its domain and another set of objects in its range. With this perspective, the typestate of an object is a function of its participation in the abstract data type as reflected in its membership in the data type's abstract sets of objects.

- **Orthogonal Composition:** In standard typestate systems, each object has a single atomic typestate. In our formulation, however, an object can be a member of multiple sets simultaneously. This promotes composite typestate structures in which the developer endows each component with a collection of abstract sets, with each set corresponding to an aspect of the typestate relevant to the component. With this kind of structure, each object's typestate is an orthogonal composition of the typestate aspects from each of the components in which it participates. Examples include composite typestates for objects that participate in multiple data structures and objects that play multiple roles within a single component.

  The advantages of this approach include better modularity (because each component deals only with those aspects of the typestate that are relevant for its operation) and support for polymorphism (because each component can operate successfully on multiple objects that participate in different ways in other components).

- **Hierarchical Typestates:** Hierarchical classification via inheritance is a key element of the type systems in most object-oriented languages, but is completely absent in existing flat typestate systems. Our formulation cleanly supports typestate hierarchies — a collection of sets can partition a more general set, with the subset inclusion ordering capturing the hierarchy.

  Examples in which we have found typestate hierarchy useful include modelling hierarchical classification structures and characterizing typestate relationships associated with caching in data structures that implement translations and maps.

- **Sharing and Typestates:** Sharing via aliased object references has caused problems for standard typestate systems — it has been difficult to ensure that if the program uses one reference to access the object and change its typestate, the declared type of other references is appropriately adjusted. Restrictions adopted

to ensure soundness have included the elimination of aliasing [23, 5], requiring typestate restoration after temporary changes [7], and allowing only monotonic typestate changes [8]. To the best of our knowledge, the role system [12] is the only typestate system to support both sharing and nonmonotonic typestate changes.

Our typestate formulation supports a new, more abstract form of sharing. If an object participates in multiple data structures, its typestate characterizes this sharing by indicating its membership in multiple typestate sets, one for each data structure. This formulation supports nonmonotonic changes — the set of objects that contain an element may change arbitrarily throughout the computation.

The basic utility of typestate systems is to enforce safety properties, specifically to ensure that the program never applies an operation to an object if the object is not in the correct typestate. We believe our generalization of the standard typestate approach substantially improves the sophistication of the properties that the type system can verify, its ability to enforce important safety properties, and the ability of typestate systems to support modular development practices. It also allows our system to much more effectively support many software engineering activities such as understanding the global sharing patterns in large programs, verifying the absence of undesirable interactions, and understanding the sequences of actions that the program may generate.

1.1 Set Programs

Our typestate checking algorithm operates on set programs. The basic operations in these programs choose, insert, and remove items and sets of objects to and from the typestate sets. The typestate system itself is based on membership, subset inclusion, and disjointness constraints involving the objects and the abstract typestate sets. It uses a compositional form of assume/guarantee reasoning — each procedure (or more generally, each code fragment) has a specification consisting of a *requires* constraint that specifies the conditions that must hold when it starts and an *ensures* constraint that specifies the conditions guaranteed to hold when it completes (assuming the requires constraint held when it was started). Our algorithm analyzes each procedure/code fragment once to verify the validity of its specification. It also checks that the requires constraints hold at each invocation.

1.2 Application to Standard Languages

Our set language is designed to be the target of translators that take modules written in more standard programming languages, then transform these modules into the set language for the purpose of typestate checking. It is the responsibility of the translator to produce the set program and translate the typestate specifications.

The translation is sound if the concrete implementation in the standard language is a refinement [4] of the abstract implementation in the set language and if the typestate specifications correctly reflect the requirements and actions of the module. A sound translation, when combined with suitable encapsulation guarantees in the standard programming language, ensures that if the abstract set language implementation typestate checks, then the concrete implementation will never violate the typestate specifications.

We anticipate the development of multiple translators, each specialized for a different kind of module. In this paper we present two such translators: a translator for hierarchical modules implemented in terms of encapsulated submodules, and a translator for modules that manage objects whose typestate is determined by the value of a field in each object. In a companion paper we present a translator for abstract data types that implement more sophisticated data structures such as lists, trees, and graphs [13]. This translator uses a precise, compositional shape analysis to verify the typestate declarations [19]. Our framework also supports the application of arbitrarily sophisticated techniques such as theorem provers. These techniques may be especially appropriate when applied to standard data structure libraries because the (potentially quite substantial) translation effort can be amortized over multiple uses by many clients. Given the range of arbitrarily complex data structures that may appear in programs, we believe that any successful approach must support a comparable range of reasoning techniques.

1.3 Characterizing Global Sharing

When combined with a programming language that encapsulates all object references inside modules, our approach enables the typestate system to capture global sharing properties. Together, all of the sets in an object's typestate completely characterize its participation in all of the data structures in the program. An absence of sharing shows up as a typestate assertion that an object is not a member of any set in a given collection of sets.

It may be instructive to compare this abstract concept of sharing with the standard concrete concept of sharing based on the aliasing of object references in the heap. In the standard concept, the program accesses objects by following object references; two objects share another object if they both have a reference to the shared object. In our typestate concept, the program accesses objects by invoking operations in modules that return references to objects; two modules share an object if the object is in a typestate set from each of the modules. In effect, our approach splits the sharing analysis problem into two stages: precise analyses verify the typestate abstraction for modules that implement sophisticated data structures, then our scalable typestate analysis verifies the abstract global sharing properties that the use of these modules induces.

We believe this two stage approach makes it possible, for the first time, to obtain an analysis that is both precise enough to verify sharing properties in programs with quite sophisticated data structures, yet scalable enough to characterize such properties in large programs composed out of multiple modules.

1.4 Contributions

This paper makes the following contributions:

- **Generalized Typestate Framework:** It presents a new generalized typestate framework based on the concept of object membership in abstract typestate sets. This framework supports the clean generalization of the standard flat, atomic typestate approach to support hierarchical typestates and typestates composed of the orthogonal composition of multiple typestate elements.

- **Typestate Checking Algorithm:** It presents a type-

state checking algorithm that verifies the typestate specifications in set programs. We prove that this algorithm is sound.

- **Translation-Based Approach:** It shows how to use translators to apply our typestate checking algorithm to programs written in standard programming languages. This approach enables the application of arbitrarily sophisticated reasoning techniques within modules, with our scalable typestate analysis effectively combining the results of these techniques across modules.

- **Global Sharing:** It presents an abstract model of sharing in programs composed out of multiple modules and shows how our typestate system makes it possible to completely characterize global, large-scale object sharing patterns in such programs. We claim that our approach is therefore the first solution to the global sharing problem in programs with multiple modules and sophisticated data structures.

2. EXAMPLES

We next present an example that illustrates how our approach works. Figure 1 presents the interface and implementation modules for a simplified file module. The `new` procedure allocates and returns a new `file` object, `open` opens the file, `read` and `write` access the file, and `close` closes the file. The `flag` field in the `file` object indicates whether the file is newly initialized, open, or closed. We use the notation `...` to indicate omitted code in the implementation module. The interface module specifies those elements of the module that are visible to clients of the module in the base language. To simplify the presentation, for the remainder of the paper we assume that only the object types (but not the fields of the objects) and the procedures are visible to clients of the module. Note that our programming model has both immutable values such as `int` and `string`, which do not have typestates, and mutable objects such as `file`, which do have typestates.

To correctly use this module, the rest of the program must obey several simple typestate constraints. Specifically, `open` should be invoked only on a newly allocated file and `read`, `write`, and `close` should be invoked only on open files. In our example the developer formalizes these requirements by identifying a group of sets for the file module (with one set per typestate), specifying the mapping from the state of the file objects to membership in these sets, then using the sets to specify the typestate requirements for the `file` parameters of the different operations. For typestate patterns in which the typestate of each object is determined by the value of a field in the object, this information (in combination with the implementation module) is all the set program translator requires to produce the set program. Figure 2 presents the *translator module* for our example, which contains this information. In general, the various different set program translators may need different kinds of typestate information, so our framework supports arbitrary translator modules that may provide arbitrary amounts of typestate information.

Figure 3 presents the *typestate module* that the translator produces for our example. This module contains the sets in the module, the set program implementations of each

```
interface module F {
  object file;
  file new(string s);
  open(file f);
  write(file f, string s);
  string read(file f);
  close(file f);
}

implementation module F {
  object file {
    int flag;
    ...
  }
  file new(string s) {
    ...
    file f = alloc(file);
    f.flag = 1;
    ...
    return f;
  }
  open(file f) {
    ... f.flag = 2; ...
  }
  write(file f, string s) { ... }
  string read(file f) { ... }
  close(file f) { ... f.flag = 3; ... }
}
```

Figure 1: File Interface and Implementation Modules

```
translator module F {
  sets init, open, closed;
  set(file f) = case f.flag of
    0: init, 1: open, 2: closed;
  file();
  open(f)  requires f in init;
  read(f)  requires f in open;
  write(f) requires f in open;
  close(f) requires f in open;
}
```

Figure 2: Translator Module for File Example

procedure, and the typestate specifications for each procedure. This module uses + to denote set union and - to denote set difference. In a `guarantees` clause, unprimed set names (`S`) denote the set before the execution of the procedure, while primed set names (`S'`) denote the set after the execution of the procedure. `All` denotes the collection of all typestate sets in the program; `All` is typically used in frame conditions such as `forall S in All - { init }. S' = S`, which states that all sets except `init` remain unchanged by the actions of the procedure.

```
typestate module F {
  sets init, open, closed;
  file() returns new init;
    guarantees init' = init + { file } and
           forall S in All - { init }. S' = S
  { v = new; init = init + { v}; return v; }
  open(f) requires f in init;
    guarantees init' = init - { f } and
           open' = open + { f } and
           forall S in All - { init, open }. S' = S
  { init = init - { f }; open = open + { f } }
  read(f) requires f in open;
    guarantees forall S in All. S' = S
  {}
  write(f) requires f in open;
    guarantees forall S in All. S' = S
  {}
  close(f) requires f in open;
     guarantees open' = open - { f } and
            closed' = closed + { f } and
            forall S in All - { open, closed }. S' = S
  { open = open - { f }; closed = closed + { f } }
}
```

Figure 3: Generated Typestate Module

Because the typestate checking takes place on modules of this form, there is a single typestate module language and all translators must produce typestate modules in this language. The translator modules, on the other hand, are specialized for the needs of each translator. There may therefore be an arbitrary number of kinds of typestate modules with arbitrary formats. The translator module in our example contains three kinds of information: the typestate sets in the module, a mapping from `file` objects to the typestate sets, and the requires clauses for the procedures in the module. In some cases we expect that the translation modules may contain more information than the module in our example (in particular, we expect that translation modules for more sophisticated typestate patterns may contain guarantee specifications for the procedures); in other cases, they may contain less information (in particular, some translators may be able to automatically infer the requires clauses for each procedure).

3. CORE LANGUAGE

In Figure 4, we provide the syntax for our set programming language. Each program in the language has a finite collection of typestate sets S and a finite set $\mathcal{P}$ of procedures. Each procedure has a set L of local variables and formal parameters, ℓ, which point to dynamically-allocated heap objects. Note that because this language is intended primarily as a specification language, it contains nondeterministic constructs such as $St_1 \Box St_2$, which nondeterministically executes one of the two statements St_1 and St_2. These constructs model abstractions of `if` statements in the underlying implementation language.

Note that the set expression $\hat{S}$ may only appear in a guarantee clause.

$$
\begin{array}{rcl}
Prog & ::= & D\ P^* \\
D & ::= & \mathsf{sets}\ S^*; \\
P & ::= & \mathsf{proc}\ p(f_1, \ldots, f_k)\ \mathsf{assumes}\ B\ \mathsf{guarantees}\ B\ \{St\ \} \\
St & ::= & St; St \mid St \Box St \mid \mathsf{if}\ B\ \mathsf{then}\ St_1\ \mathsf{else}\ St_2 \mid S = E \mid \\
 & & \mathsf{choose}\ \ell\ \mathsf{from}\ S \mid [\ell =]\mathsf{proc}(\cdots) \mid \ell_1 = \ell_2 \\
B & ::= & B\ \mathsf{and}\ B \mid B\ \mathsf{or}\ B \mid S\ \mathsf{eq}\ E \mid S\ \mathsf{sub}\ E \mid \\
 & & \ell\ \mathsf{in}\ E \mid \ell\ \mathsf{not\ in}\ E \mid \ell_1\ \mathsf{eq}\ \ell_2 \mid \ell_1\ \mathsf{neq}\ \ell_2 \mid \\
 & & S \cap T = \emptyset \\
E & ::= & \emptyset \mid \hat{S} \mid E_1 \cup E_2 \mid E_1 \cap E_2 \mid E_1 \setminus E_2 \mid \{\ell^*\}
\end{array}
$$

Figure 4: Grammar for set programming language

3.1 Operational Semantics

We next describe the operational semantics of our language. From now on, we assume that our language has been compiled from the set programming language presented earlier into SSA form; in particular, all branches are forward, by construction, and furthermore no local variable is redefined. Our semantics is written as a transition system on pairs $\langle [p, r] \circ s, L, \mathcal{C} \rangle$, where p represents the program counter, r is a unique identifier for the current activation record, and s is the stack of $[p, r]$ pairs for procedures on the stack. The set L represents the set of local variables, and contains triples $\langle r, \mathsf{name}, \mathsf{id} \rangle$. The set $\mathcal{C}$ stores set contents; it contains pairs $\langle S, \{\mathsf{id}_1, \ldots, \mathsf{id}_k\} \rangle$ containing information about set membership. To allow the verification of guarantee clauses, $\mathcal{C}$ also contains triples containing pre-sets, of the form $\langle r, \hat{S}, \langle \mathsf{id}_1, \ldots, \mathsf{id}_{k'} \rangle \rangle$, indicating that the contents of set S at entry to procedure r. Each id represents a distinct heap object.

Our rules for evaluating set expressions take an expression and evaluate it in a state, giving a set:

$$
\begin{array}{rcl}
\mathsf{eval}_{\langle L,\mathcal{C} \rangle}(\emptyset) & = & \{\} \\
\mathsf{eval}_{\langle L,\mathcal{C} \rangle}(S) & = & \{V \mid \langle S, V \rangle \in \mathcal{C}\} \\
\mathsf{eval}_{\langle L,\mathcal{C} \rangle}(S + T) & = & \mathsf{eval}_{\langle L,\mathcal{C} \rangle}(S) \cup \mathsf{eval}_{\langle L,\mathcal{C} \rangle}(T) \\
\mathsf{eval}_{\langle L,\mathcal{C} \rangle}(S\ \mathsf{intersect}\ T) & = & \mathsf{eval}_{\langle L,\mathcal{C} \rangle}(S) \cap \mathsf{eval}_{\langle L,\mathcal{C} \rangle}(T) \\
\mathsf{eval}_{\langle L,\mathcal{C} \rangle}(S - T) & = & \mathsf{eval}_{\langle L,\mathcal{C} \rangle}(S) \setminus \mathsf{eval}_{\langle L,\mathcal{C} \rangle}(T) \\
\mathsf{eval}_{\langle L \cup \{r,\ell,\mathsf{id}\},\mathcal{C} \rangle}(\{\ell\}) & = & \{\mathsf{id}\} \\
\mathsf{eval}_{L,\mathcal{C}}(B\ ?\ S_1 : S_2) & = & \mathsf{eval}_{L,\mathcal{C}}(S_1)\ \text{if}\ B\ \text{true} \\
 & = & \mathsf{eval}_{L,\mathcal{C}}(S_2)\ \text{if}\ B\ \text{false}
\end{array}
$$

In Figure 7 we present rules for evaluating boolean expressions.

3.2 Type Checking Algorithm for Core System

We now describe the type checking algorithm for our core language. This algorithm starts with the `assume` condition at the start of each method and tries to prove the `guarantee` condition, by computing a logical predicate, $\mathcal{H}$, which describes the program state for each program point. Type

Statement	Transition	Constraints
p: `x = y;`	$\langle [p,r] \circ s, L \cup \{\langle r, \mathsf{y}, \mathsf{id}\rangle\}, \mathcal{C}\rangle \rightarrow$ $\langle [p',r] \circ s, L \cup \{\langle r, \mathsf{x}, \mathsf{id}\rangle, \langle r, \mathsf{y}, \mathsf{id}\rangle\}, \mathcal{C}\rangle$	
p: `S = E;`	$\langle [p,r] \circ s, L, \mathcal{C} \cup \{\langle S, *\rangle\}\rangle \rightarrow$ $\langle [p',r] \circ s, L, \mathcal{C} \cup \{\langle S, \mathsf{eval}_{\langle L,\mathcal{C}\rangle}(E)\rangle\}\rangle$	
p: `nondetgoto p1`	$\langle [p,r] \circ s, L, \mathcal{C}\rangle \rightarrow \langle [p',r] \circ s, L, \mathcal{C}\rangle$ $\langle [p,r] \circ s, L, \mathcal{C}\rangle \rightarrow \langle [p1,r] \circ s, L, \mathcal{C}\rangle$	
p: `choose x from S`	$\langle [p,r] \circ s, L \cup \{\langle r, \mathsf{x}, *\rangle\}, \mathcal{C} \cup \{\langle S, \{\mathsf{id}_{1..n}\}\rangle\}\rangle$ $\rightarrow \langle [p',r] \circ s, L \cup \{\langle r, \mathsf{x}, \mathsf{id}_i\rangle\}, \mathcal{C} \cup \{\langle S, \{\mathsf{id}_{1..n}\}\rangle\}\rangle$	$\forall 1 \leq i \leq n$

Figure 5: Operational semantics for simple statements

Statement	Transition	Constraints
p: `x = proc(a1, ..., ak)`	$\langle [p,r] \circ s, L \cup \{\langle r, a_1, \mathsf{id}_1\rangle, \ldots, \langle r, a_k, \mathsf{id}_k\rangle\}, \mathcal{C}\rangle$ $\rightarrow \langle [\mathsf{entry}, r'] \circ [p,r] \circ s,$ $L \cup \{\langle r, a_1, \mathsf{id}_1\rangle, \ldots, \langle r, a_k, \mathsf{id}_k\rangle\}$ $\cup \{\langle r', \mathsf{retval}, x\rangle, \langle r', f_1, \mathsf{id}_1\rangle, \ldots, \langle r', f_k, \mathsf{id}_k\rangle\}$ $\cup \{\langle r', \ell_1, ?\rangle, \ldots, \langle r', \ell_n, ?\rangle\},$ $\mathcal{C} \cup \bigcup_{S \in \mathcal{S}} \{r', \hat{S}, \mathsf{id} \in S\}\rangle$	r' fresh, true(A)
p: `return x`	$\langle [\mathsf{return}\ \mathsf{x}, r'] \circ [p,r] \circ s, L \cup \{\langle r', x, \mathsf{id}_x\rangle, \langle r', \mathsf{retval}, X\rangle\}, \mathcal{C}\rangle$ $\rightarrow \langle [p',r] \circ s, L \cup \{\langle r, X, \mathsf{id}_x\rangle\}, \mathcal{C} \setminus \{r', \hat{S}, *\}\rangle$	true(G)

where A is the assume condition for `proc` and G is its guarantee condition

Figure 6: Operational semantics for interprocedural statements

checking succeeds if, for each procedure $\mathcal{P}$ with guarantee condition $\mathcal{G}$, we can use the implies predicate to show

$$\forall s \in \mathsf{exit}(\mathcal{P}).\mathcal{H}_s \text{ implies } \mathcal{G}$$

Here we use the notation that $\mathsf{exit}(\mathcal{P})$ is the set of exit nodes in the control flow graph for the procedure $\mathcal{P}$ and $\mathcal{H}_s$ is the logical predicate $\mathcal{H}$ at the program point s.

3.2.1 Heap Predicates

Our exposition of the type checking algorithm starts with an explanation of how we compute heap predicates. The computation is a standard forward dataflow analysis operating on heap predicates.

3.2.1.1 Grammar.

Our heap predicates $\mathcal{H}$ consist of a disjunction of predicates $\mathcal{F}$ conforming to the following grammar:

$$\begin{array}{rcl} \mathcal{H} & ::= & \mathcal{F}[\vee\mathcal{F}]^* \\ \mathcal{F} & ::= & \{C^*\} \\ C & ::= & S \text{ eq } E \mid S \text{ sub } E \mid \ell \text{ in } E \mid \ell \text{ not in } E \mid \\ & & \ell_1 \text{ eq } \ell_2 \mid \ell_1 \text{ neq } \ell_2 \mid S \cap T = \emptyset \\ E & ::= & \emptyset \mid \hat{S} \mid E_1 \cup E_2 \mid E_1 \cap E_2 \mid E_1 \setminus E_2 \mid \{\ell^*\} \end{array}$$

We represent each predicate $\mathcal{F}$ as a set of facts. To interpret a heap predicate $\mathcal{H}$ as a logical formula, convert each predicate $\mathcal{F} \in \mathcal{H}$ to a conjunction of the facts $C \in \mathcal{F}$, then disjoin the resulting conjunctions together.

3.2.1.2 Initial Sets.

Our dataflow analysis starts with the assume condition $\mathcal{A}$ of a procedure $\mathcal{P}$.

3.2.1.3 Transfer Functions for General Statements.

The transfer functions operate independently on each of the various disjuncts $\mathcal{F}$ of $\mathcal{H}$. We present the rules for straight-line code first, then discuss our merge operator.

The general transfer function for statement p (except where p is a procedure call) is:

$$\mathcal{F}' = \mathcal{F} \setminus \mathsf{kill}(p) \cup \mathsf{gen}(p)$$

Before defining the gen and kill sets, we define the set of definitions of a set S in a heap predicate $\mathcal{F}$:

$$\mathsf{defs}_S(\mathcal{F}) = \text{elements of } \mathcal{F} \text{ of the form } S\ [\mathsf{eq}|\mathsf{sub}]\ E.$$

The kill set for a statement $p : S = E$ is precisely the set of definitions for S:

$$\mathsf{kill}(p) = \text{elements of } \mathcal{F} \text{ of the form } S\ [\mathsf{eq}|\mathsf{sub}]\ E,$$

and for the remaining statements choose x from T and `y = x`, it is empty because we use SSA form:

$$\mathsf{kill}(p) = \emptyset.$$

We write our gen sets using inference rules as presented in Figure 8. Each rule has a set of facts above the line and a fact below the line. The interpretation of each rule is that if each of the facts above the line is an element of $\mathcal{F}$, then the fact below the line is an element of $\mathsf{gen}(\mathcal{F})$.

When we have a control-flow merge, we disjoin the predicates on the two sides of the merge.

3.2.1.4 Transfer Functions for Procedure Calls.

For procedure calls, we need to verify that the callee's assume condition holds, and we need to adjust the heap predicate to account for the effects of the procedure call.

Checking the assume condition is simply a matter of checking that the heap predicate before the call implies the assume condition:

$$\mathcal{H} \vdash \mathcal{A},$$

and we show how to do this in the next subsection.

Adjusting the heap to account for the effects of a procedure call is done by interpreting the procedure's guarantee

$$\frac{B ::= n \text{ in } E \quad \langle r, n, \mathsf{id}\rangle \in \mathcal{C} \quad E \vdash T \quad \mathsf{id} \in T}{\mathrm{true}(B)} \qquad \frac{B ::= n \text{ in } E \quad \langle r, n, \mathsf{id}\rangle \in \mathcal{C} \quad E \vdash T \quad \mathsf{id} \notin T}{\mathrm{false}(B)}$$

$$\frac{B ::= E_1 = E_2 \quad E_1 \vdash T_1 \quad E_2 \vdash T_2 \quad T_1 = T_2}{\mathrm{true}(B)} \qquad \frac{B ::= E_1 = E_2 \quad E_1 \vdash T_1 \quad E_2 \vdash T_2 \quad T_1 \neq T_2}{\mathrm{false}(B)}$$

$$\frac{B ::= E_1 \text{ subset } E_2 \quad E_1 \vdash T_1 \quad E_2 \vdash T_2 \quad T_1 \subseteq T_2}{\mathrm{true}(B)} \qquad \frac{B ::= E_1 \text{ subset } E_2 \quad E_1 \vdash T_1 \quad E_2 \vdash T_2 \quad T_1 \not\subseteq T_2}{\mathrm{false}(B)}$$

Figure 7: Boolean evaluation rules

$$\boxed{\mathsf{gen}(S = \emptyset)} \qquad \frac{}{S = \emptyset}$$

$$\boxed{\mathsf{gen}(S = T)} \qquad \frac{T \text{ eq } E_T}{S \text{ eq } E_T} \qquad \frac{T \text{ sub } E_T}{S \text{ sub } E_T}$$

$$\boxed{\mathsf{gen}(S = T \cap R)} \qquad \frac{T \text{ [eq|sub] } E_T}{S \text{ sub } E_T} \qquad \frac{R \text{ [eq|sub] } E_R}{S \text{ sub } E_R} \qquad \frac{T \text{ eq } E_T \quad R \text{ eq } E_R}{S \text{ eq } E_T \cap E_R}$$

$$\frac{i \in \mathcal{S}}{S \cap i = \emptyset} \qquad \frac{T \cap i = \emptyset}{S \cap i = \emptyset} \qquad \frac{T \cap i = \emptyset \quad R \cap i = \emptyset}{S \cap i = \emptyset}$$

$$\boxed{\mathsf{gen}(S = T \cup R)}$$

$$\frac{T \text{ sub } E_T \quad R \text{ [eq|sub] } E_R}{S \text{ sub } E_T \cup E_R} \qquad \frac{T \text{ [eq|sub] } E_T \quad R \text{ sub } E_R}{S \text{ sub } E_T \cup E_R} \qquad \frac{T \text{ eq } E_T \quad R \text{ eq } E_R}{S \text{ eq } E_T \cup E_R} \qquad \frac{T \cap i = \emptyset \quad R \cap i = \emptyset}{S \cap i = \emptyset}$$

$$\boxed{\mathsf{gen}(S = T \setminus R)}$$

$$\frac{T \text{ [eq|sub] } E_T \quad R \text{ sub } E_R}{S \text{ sub } E_T} \qquad \frac{T \text{ eq } E_T \quad R \text{ eq } E_R}{S \text{ eq } E_T \setminus E_R} \qquad \frac{T \text{ sub } E_T \quad R \text{ eq } E_R}{S \text{ sub } E_T \setminus E_R} \qquad \frac{}{R \cap S = \emptyset}$$

$$\boxed{\mathsf{gen}(S = \{\ell\})}$$

$$\frac{\mathsf{mayAlias}(\ell) = \emptyset}{S \text{ eq } \{x \text{ in } \mathsf{defs}_\ell(\mathcal{F})\}} \qquad \frac{\mathsf{mayAlias}(\ell) \neq \emptyset}{S \text{ sub } \bigcup_{d \in \mathsf{defs}_{\mathsf{mayAlias}(\ell)}(\mathcal{F})} \{x \text{ in } d\}} \qquad \frac{\ell \text{ in } S' \quad i \cap S' = \emptyset}{S \cap i = \emptyset}$$

$$\boxed{\mathsf{gen}(\mathtt{choose\ x\ from}\ S)} \qquad \frac{}{\mathtt{x\ in}\ S}$$

$$\boxed{\mathsf{gen}(\mathtt{x} = \mathtt{y})} \qquad \frac{S \cap i = \emptyset}{\mathtt{x\ not\ in}\ i} \qquad \frac{\mathtt{y\ in\ E}}{\mathtt{x\ in\ E}} \qquad \frac{\mathtt{y\ not\ in\ E}}{\mathtt{x\ not\ in\ E}} \qquad \frac{}{\mathtt{y\ eq\ x}}$$

Figure 8: Rules for gen sets in transfer functions

clause $\mathcal{G}$ on the incoming heap predicate $\mathcal{H}$. We do this using the following algorithm.
Convert $\mathcal{G}$ to disjunctive normal form. For each disjunct $\mathcal{F}_i$ of $\mathcal{G}$:

1. Create $\mathcal{H}_i$, a clone of $\mathcal{H}$.

2. Preprocess $\mathcal{F}_i$ by substituting in the definition[1] of S in $\mathcal{H}$ for all uses of $\hat{S}$:

$$\mathcal{F}_i := \mathcal{F}[\mathsf{defs}_S(\mathcal{H})/\hat{S}]$$

   Note that S in the calling context is the same as $\hat{S}$ in the callee context.

3. Apply definitions of $\mathcal{F}_i$ to $\mathcal{H}$:

$$\begin{array}{l}\forall S \in \mathcal{S}.\ \mathcal{H}_i := \\ \qquad \mathcal{H}_i \setminus \{(S \text{ [eq|sub] } *)\} \cup \{(S \cap * = \emptyset)\} \\ \forall S \text{ eq } E \in \mathcal{F}_i.\ \mathcal{H}_i := \mathcal{H}_i \cup S \text{ eq } E \\ \forall S \text{ sub } E \in \mathcal{F}_i.\ \mathcal{H}_i := \mathcal{H}_i \cup S \text{ sub } E \\ \forall (S \cap T = \emptyset) \in \mathcal{F}_i\ \mathcal{H}_i := \mathcal{H}_i \cup (S \cap T = \emptyset)\end{array}$$

4. (Eliminating stale local variable information.) For each S changed in $\mathcal{F}$, remove all local variable set-membership constraints:

$$\forall S \in \mathcal{F}_i \text{ s.t. } S \text{ eq } \hat{S} \notin \mathcal{F}_i.\ \mathcal{H}_i := \mathcal{H}_i \setminus (\ell \text{ [not] in } S)$$

5. Conjoin conjuncts about parameters and return variables. We label the formal parameters f_i and the actual parameters a_i; the return value of the parameter is f_r in the callee context and a_r in the caller context. Remove all old conjuncts about a_i and conjoin new conjuncts from $\mathcal{F}_i[a_i/f_i]$.

$$\begin{array}{l}\forall f_i \text{ eq } f_j \in \mathcal{F}_i.\ \mathcal{H}_i := (\mathcal{H}_i \cup a_i \text{ eq } a_j) \\ \forall f_i \text{ neq } f_j \in \mathcal{F}_i.\ \mathcal{H}_i := (\mathcal{H}_i \cup a_i \text{ neq } a_j) \\ \forall f_i \text{ in } E \in \mathcal{F}_i.\ \mathcal{H}_i := (\mathcal{H}_i \cup a_i \text{ in } E) \\ \forall f_i \text{ not in } E \in \mathcal{F}_i.\ \mathcal{H}_i := (\mathcal{H}_i \cup a_i \text{ not in } E)\end{array}$$

At the end of this process, disjoin all the $\mathcal{H}_i$s.

3.2.2 *Implication Rules*

Our system depends on being able to prove that the predicates generated at all control-flow graph exit nodes imply the guarantee condition for the corresponding procedure and that the assumption at each procedure call site is guaranteed by the state of the caller. We provide an qlgorithm that can be used to decide implication for our core system.

Recall that our type checker's goal is to prove preconditions and postconditions $\mathcal{P}$; in particular, we must prove assume conditions $\mathcal{A}$ from heap predicates $\mathcal{H}$ at procedure invoke points

$$\mathcal{H} \vdash \mathcal{A},$$

[1] Substituting $S = \hat{S} \cup \hat{T}; S \text{ sub } R$ for $T = \hat{S} \cup \{\ell\}$ gives $T = \hat{S} \cup \hat{T} \cup \{l\}; T \text{ sub } \hat{R} \cup \{\ell\}$.

and guarantee conditions $\mathcal{G}$ from heap predicates $\mathcal{H}$ at procedure exit points

$$\mathcal{H} \vdash \mathcal{G}.$$

3.2.2.1 Disjunction.

If the heap predicate $\mathcal{H}$ has more than one disjunct $\mathcal{D}$, then showing implication requires showing that each disjunct implies the guarantee condition. That is,

$$(\mathcal{D}_1 \vdash \mathcal{P}) \wedge (\mathcal{D}_2 \vdash \mathcal{P}) \Longrightarrow (\mathcal{D}_1 \vee \mathcal{D}_2 \vdash \mathcal{P})$$

We may therefore assume, in the sequel, that $\mathcal{H}$ has only one disjunct; since $\mathcal{D}$ is a set of conjuncts, we will treat $\mathcal{H}$ as a set of conjuncts.

3.2.2.2 Top-Level Guarantee Expressions.

We start constructing the $\vdash$ predicate based on the syntax of pre/postcondition clauses:

$$\begin{array}{rcl} \mathcal{P} & ::= & B \text{ and } B \mid B \text{ or } B \mid \text{not } B \\ & \mid & \ell \text{ in } \hat{S} \mid \ell_1 = \ell_2 \\ & \mid & S \text{ eq } E \mid S \text{ sub } E \end{array}$$

Clearly,

$$\begin{array}{rcl} (\mathcal{H} \vdash B_1) \wedge (\mathcal{H} \vdash B_2) & \Longrightarrow & (\mathcal{H} \vdash B_1 \text{ and } B_2) \\ (\mathcal{H} \vdash B_1) \vee (\mathcal{H} \vdash B_2) & \Longrightarrow & (\mathcal{H} \vdash B_1 \text{ or } B_2) \end{array}$$

We will soon use mustAlias and mayAlias; they are defined as:

$$\begin{array}{rcl} (\ell_1 \text{ eq } \ell_2) \in \mathcal{H} & \Longrightarrow & (\mathcal{H} \vdash \mathsf{mustAlias}(\ell_1, \ell_2)) \\ (\ell_1 \text{ neq } \ell_2) \in \mathcal{H} & \Longrightarrow & (\mathcal{H} \vdash \text{not } \mathsf{mayAlias}(\ell_1, \ell_2)) \\ \mathsf{mayAlias}(\ell_1) & = & \{\ell \in \mathcal{H} \mid \mathsf{mayAlias}(\ell_1, \ell)\} \end{array}$$

Our algorithm type-checks the following not in guarantee condition:

$$\bigcup_{i \in \{\ell\} \cup \mathsf{mayAlias}(\ell)} \mathsf{defs}_i(\mathcal{H}) \neq \hat{S} \Longrightarrow (\mathcal{H} \vdash \ell \text{ not in } \hat{S})$$

For local variable predicates ℓ in $\hat{S}$, we have:

$$\begin{array}{rcl} \mathsf{def}_\ell(\mathcal{H}) = S & \Longrightarrow & (\mathcal{H} \vdash \ell \text{ in } \hat{S}) \\ \ell_1 \text{ eq } \ell_2 & \Longrightarrow & (\mathcal{H} \vdash \ell_1 = \ell_2) \end{array}$$

We finish constructing $\vdash$ by providing the following rules:

$$\begin{array}{rcl} (S \text{ eq } E) \in \mathcal{H} & \Longrightarrow & (\mathcal{H} \vdash S \text{ eq } E) \\ (S \text{ sub } E) \in \mathcal{H} & \Longrightarrow & (\mathcal{H} \vdash S \text{ sub } E) \end{array}$$

Note that we may need to apply some inference rules in order to prove our guarantee conditions, because our transfer functions may not provide a predicate in exactly the form that the guarantee condition expects.

3.2.2.3 Inference Rules.

We provide inference rules on elements of $\mathcal{H}$; these may be required to let $\vdash$ go through. These may be arbitrarily applied.

$$\begin{array}{rcl} S \text{ eq } E & \Rightarrow & S \text{ sub } E \\ \forall E'. S \text{ sub } E & \Rightarrow & S \text{ sub } E \cup E' \\ S \text{ sub } E \wedge S \text{ sub } E' & \Leftrightarrow & S \text{ sub } E \cap E' \\ \forall E'. E_1 \cap E_2 = \emptyset & \Leftrightarrow & E_1 - E' \cap E_2 \cup E' = \emptyset \\ E_1 \cap E_2 = \emptyset \wedge \ell_1 \text{ neq } \ell_2 & \Leftrightarrow & E_1 \cap \{\ell_1\} \cap E_2 \cup \{\ell_2\} = \emptyset \\ \ell_1 \in E_1 \wedge \ell_2 \in E_2 \wedge E_1 \cap E_2 = \emptyset & \Rightarrow & \ell_1 \text{ neq } \ell_2 \end{array}$$

3.2.2.4 Set Expressions.

Recall that our grammar fragment for expressions is as follows:

$$E ::= \emptyset \mid \hat{S} \mid E_1 \cup E_2 \mid E_1 \cap E_2 \mid E_1 \setminus E_2 \mid \{\ell^*\}$$

For expressions of the form $\emptyset, E_1 \cup E_2, E_1 \cap E_2, E_1 \setminus E_2$, the construction of the $\vdash$ predicate is clear.

3.3 Soundness Proof

In this subsection, we outline a soundness proof for our type system. We will show that in a correctly-typed program, the assume and guarantee conditions never fail at runtime; that is, the type information which summarizes the program state is always accurate.

Definition 1. *We say that a program state satisfies a heap predicate and write $\langle L, \mathcal{C} \rangle \vdash \mathcal{H}$ if, for any disjunct $\mathcal{F}$ of $\mathcal{H}$, the program state satisfies every conjunct $C \in \mathcal{F}$:*

$$\exists \mathcal{F} \in \mathcal{H}.\ \forall C \in \mathcal{F}.\ \langle L, \mathcal{C} \rangle \vdash C$$

Figure 9 provides a definition for $\vdash$ for each type of conjunct. Given this definition, we can proceed to state our soundess result.

Proposition 1. *If a program type checks, then in all executions of the program, the operational semantics is simulated by the heap transfer functions: that is, if the operational semantics executes a transition $\langle [p, r] \circ s, L, \mathcal{C} \rangle \rightarrow \langle [p', r] \circ s, L', \mathcal{C}' \rangle$, and $\langle L, \mathcal{C} \rangle \vdash \mathcal{H}$,then $\langle L', \mathcal{C}' \rangle \vdash \mathcal{H}'$, where $\mathcal{H}'$ is the result of applying our transfer function on $\mathcal{H}$ to statement p.*

This proposition allows us to conclude that all assume and guarantee checks will always succeed at runtime, given the trivial base case: the heap predicate where all sets are empty satisfies the initial state of the operational semantics.

Proof of Proposition. The proposition is straightforward except for at procedure boundaries. We include a proof for a typical statement and also for procedure call and return statements; the omitted cases are similar in flavour.

Consider the statement $p : S = E$ where E is of the form $T \cup R$. The operational semantics would execute the transition

$$\langle [p, r] \circ s, L, \mathcal{C} \cup \{\langle S, * \rangle\} \rangle \rightarrow \langle [p', r] \circ s, L, \mathcal{C} \cup \{\langle S, \mathsf{eval}_{\langle L, \mathcal{C} \rangle}(E) \rangle\} \rangle$$

where

$$\mathsf{eval}_{\langle L, \mathcal{C} \rangle}(T \cup R) = \mathsf{eval}_{\langle L, \mathcal{C} \rangle}(T) \cup \mathsf{eval}_{\langle L, \mathcal{C} \rangle}(R)$$

The general heap predicate transfer function is

$$\mathcal{F}' = \mathcal{F} \setminus \mathsf{kill}(p) \cup \mathsf{gen}(p);$$

$$
\begin{array}{rcl}
\langle L, \mathcal{C} \cup \{\langle S, \mathsf{eval}_{\langle L,\mathcal{C}\rangle}(E)\rangle\}\rangle & \vdash & S \ \mathsf{eq}\ E \\
\langle L, \mathcal{C} \cup \{\langle S, \mathsf{eval}_{\langle L,\mathcal{C}\rangle}(E')\rangle\}\rangle \wedge E \subseteq E' & \vdash & S \ \mathsf{sub}\ E \\
\langle L \cup \{\langle \ell_1, \mathsf{id}\rangle, \langle \ell_2, \mathsf{id}'\rangle\}, \mathcal{C}\rangle & \vdash & \ell_1 \ \mathsf{neq}\ \ell_2 \text{ if } \mathsf{id} \neq \mathsf{id}' \\
\langle L \cup \{\langle \ell, \mathsf{id}\rangle\}, \mathcal{C}\rangle & \vdash & \ell \ \mathsf{in}\ E \text{ if } \mathsf{id} \in \mathsf{eval}_{\langle L,\mathcal{C}\rangle} \\
\langle L \cup \{\langle \ell, \mathsf{id}\rangle\}, \mathcal{C}\rangle & \vdash & \ell \ \mathsf{not\ in}\ E \text{ if } \mathsf{id} \notin \mathsf{eval}_{\langle L,\mathcal{C}\rangle} \\
\langle L, \mathcal{C}\rangle & \vdash & S \cap T = \emptyset \text{ if } \mathsf{eval}_{\langle L,\mathcal{C}\rangle}(S) \cap \mathsf{eval}_{\langle L,\mathcal{C}\rangle}(T) = \emptyset
\end{array}
$$

Figure 9: Conditions for program state to imply a conjunct

clearly, the kill set in the heap predicate transfer function has the same effect as removing $\langle S, * \rangle$ from $\mathcal{C}$.

We need to show that each conjunct we add to the heap predicate is also satisfied in the operational semantics. A typical example is

$$\frac{T \ \mathsf{sub}\ E_T \quad R \ [\mathsf{eq}|\mathsf{sub}]\ E_R}{S \ \mathsf{sub}\ E_T \cup E_R}$$

This rule has, as its precondition, a heap predicate stating that every element of T is in E_T and that every element of R is in E_R; the assumption of our proposition allows us to conclude that $\mathsf{eval}_{\langle L,\mathcal{C}\rangle}(T)$ and $\mathsf{eval}_{\langle L,\mathcal{C}\rangle}(R)$ satisfy their definitions in the heap predicate. The operational semantics sets S to be $\mathsf{eval}_{\langle L,\mathcal{C}\rangle}(T) \cup \mathsf{eval}_{\langle L,\mathcal{C}\rangle}(R)$, which clearly satisfies $E_T \cup E_R$, so that we may indeed add the conjunct $S \ \mathsf{sub}\ E_T \cup E_R$.

At a procedure call, the assume condition $\mathcal{A}$ always holds because the type checker checks $\mathcal{A}$ based on the operational semantics' pre-state; we know $\langle L, \mathcal{C}\rangle \vdash \mathcal{H}$ from the precondition of the proposition.

Similarly, the guarantee condition $\mathcal{G}$ is verified by the type checker. We must show that the transfer function for procedure calls produces a suitable heap predicate after procedure exit point p. Consider a set T which gets defined in the callee. Because the guarantee condition holds, any conjuncts about T are known to be satisfied in all executions leading to p. This means that we can just copy the definition of T (substituting definitions in for all references to any set $\hat{S}$):

$$
\begin{array}{rcl}
\mathcal{F}_i & := & \mathcal{F}_i[\mathsf{defs}_S(\mathcal{H})/\hat{S}] \\
\forall S \ \mathsf{eq}\ E \in \mathcal{F}_i.\ \mathcal{H}_i & := & \mathcal{H}_i \cup S \ \mathsf{eq}\ E
\end{array}
$$

4. TRANSLATORS

Translators take implementation modules written in an implementation language with standard constructs such as encapsulated global variables and object fields (including object references) and produce typestate modules written in our set language. We first discuss the conditions that all translators must obey to be correct, then present an example of a translator for a class of modules with flat typestates.

4.1 Translator Correctness Conditions

We first outline the framework required to state the translator correctness conditions. We assume an operational semantics for the implementation language. Using well-known techniques, we formulate this operational semantics as a transition function on configurations; each configuration is a pair $\langle [p, r] \circ s, L, \mathcal{D}\rangle$. Except for $\mathcal{D}$, each of the components of the configuration is the same as in the set language operational semantics (see Section 3.1). The implementation heap $\mathcal{D}$ consists of tuples of the form $\langle \mathsf{id}, f, m, v\rangle$ (here id is an object, f is a field in a module m, and v is the value of the field) and tuples of the form $\langle g, m, v\rangle$ (here g is a global variable in module m and v is the value of the global variable).

For each module in each execution of the program, the operational semantics defines a set of module entry and exit points. There are two kinds of module entry points: a call to one of the module's procedures from outside the module, and a return back into the module from an invoked procedure outside the module. There are two kinds of module exit points: a call from inside the module to a procedure outside the module, and a return from a procedure inside the module to a procedure outside the module.

The translation correctness condition is stated with the aid of a set of mappings $\mu_m(\mathcal{D})$, one for each module m. Conceptually, this mapping is an abstraction function that defines how the values of global variables, object fields, and references determine the typestate sets for each object. Formally, $\mu_m(\mathcal{D})$ is a collection of pairs $\langle S, \{\{\mathsf{id}_1, \ldots, \mathsf{id}_k\}\}\rangle$ defining the objects in each typestate set S in module m.

We instrument the operational semantics for both the implementation and set languages to generate traces of the execution of the implementation program and translated set language program. Each trace is a sequence of module entry and exit point records. For the implementation language, each record is of the form $\langle \bigcup_m \mu_m(\mathcal{D}), \downarrow \texttt{proc}(\texttt{id1}, \ldots, \texttt{idk})\rangle$, which is a call to procedure `proc` with parameters `id1, ..., idn`, or $\langle \bigcup_m \mu_m(\mathcal{D}), \uparrow \texttt{id} = \texttt{proc}(\texttt{id1}, \ldots, \texttt{idk})\rangle$, which is a return from `proc` with return value `id`. Here $\mathcal{D}$ is the contents of the heap at the call or return point. Note that each record specifies the typestate set membership according to the maps μ_m. The set language has a similar records, but instead of $\bigcup_m \mu_m(\mathcal{D})$, the first item of each record is the abstract typestate set heap $\mathcal{C}$ at that point in the execution.

The correctness condition for the translation is that the implementation program must be a refinement of the set program — the set of traces of the set program must be a superset of the set of traces of the implementation program. Its application in our context ensures that if the typestate checker verifies that the set program conforms to the typestate declarations, then all executions of the implementation program do not violate the typestate declarations. Conceptually, this is true because the typestate checker verifies that no execution of the set program violates the typestate declarations. The refinement ensures that all executions of the implementation program are included in the set program executions, so none of the implementation program executions

violate the typestate declarations.

4.2 Translator for Flag Typestates

We next present a simple translator for modules (such as our example `file` module in Section 2) whose typestates are determined by the value of a flag field in objects allocated in the module. The module contains one object declaration whose opaque type is available to clients of the module. There is an integer field in each of these objects whose value (chosen out of a fixed set of values) determines the typestate set that the object is a member of. The module may include arbitrary other unexported encapsulated state. Each procedure in the module takes values (`int`s, `string`s, etc.) and one object (an instance of the exported object type) as parameters. Each procedure contains at most one assignment to the flag field. This assignment must be executed on all paths through the procedure and be of the form `v.f = c`, where `f` is the flag field and `c` is a constant identifying the new typestate of the object to which `v` refers. This object can be either a parameter or a new object allocated in the procedure. All of these conditions can be easily checked with a simple scan of the module.

The translation module specifies the typestate sets, a specification of the mapping from the integer flag values to the typestate sets, and the typestate requirements of the procedures. See Figure 2 for an example of such a translation module. Given the implementation module and the typestate module, the translator generates the typestate module as follows.

- **Set Generation:** It copies the declaration of the typestate sets from the translator module to the typestate module.
- **Requirements Clauses:** It copies the procedure declarations, including the requirements clauses, from the translator module to the typestate module.
- **Guarantees Clauses:** It examines the procedure bodies in the implementation module to find the (optional) assignment to the flag field. If such an assignment is present and changes the field in a parameter object, it generates a guarantees clause that specifies that 1) the parameter object is removed from the typestate set it was in at the start of the procedure (it determines this set by examining the requires clause for the procedure), and 2) the parameter object is added to the typestate set specified by the value of the flag field in the assignment. If the assignment is to newly allocated object, it only specifies an addition to the typestate set into which it was placed by the assignment statement.
- **Set Implementation Bodies** If the procedure contains an assigment to the flag field that changes the typestate, the translator generates a procedure body that uses set difference to remove the object from its old typestate set (if the assignment changes the parameter object) and insert it into its new typestate set.
- **Frame Conditions:** It generates a frame condition specifying that all of the typestate sets not affected by the procedure remain unchanged.
- **Return Clauses:** If the procedure returns an object, the translator checks that it was either a parameter or a newly allocated object and generates a return clause that specifies the typestate of the returned object.

See Figure 3 for an example of a typestate module that this translator can produce.

The mapping μ_m for this module assigns each object to a typestate set based on the value of its typestate flag. Given the restrictions on the implementation, it is easy to see that the generated typestate module has the same set of traces as the implementation module — the return values are the same, and the set program is constructed to ensure that the objects in the set program version of the module are in the same typestate sets as in the implementation version after the application of the mapping μ_m.

Despite its simplicity, this approach is sufficient to accurately model the functionality of standard flat typestate systems. And several simple extensions can increase the range of programs that it can support. Specifically, it is possible to support modules with multiple kinds of exported object types (with a separate flag field for each object type), procedures that may conditionally perform typestate changes and allocate new objects, and procedures with multiple object parameters.

5. RELATED WORK

The research presented in this paper focuses on a generalized typestate system that, in addition to enforcing typestate safety properties, also captures an abstract version of sharing. It also provides a framework for integrating a variety of arbitrarily sophisticated program analysis and verification algorithms in the service of translating implementation modules written in standard languages into typestate modules written in our set language. We discuss related work in these areas.

5.1 Typestate Systems

Typestate systems generalize standard type systems in that the typestate of an object may change during the computation. Aliasing (or more generally, any kind of sharing) is the key problem for typestate systems — if the program uses one reference to change the typestate of an object, the typestate system must ensure that either the declared typestate of the other references is updated to reflect the new typestate or that the new typestate is compatible with the old declared typestate at the other references.

Most typestate systems avoid this problem altogether by eliminating the possibility of aliasing [23, 5]. A recent generalization allows the program to temporarily prevent the program from using a given set of references, change the typestate of an object with aliases only in that set, then restore the typestate and reenable the use of the aliases [7]. Another generalization allows monotonic typestate changes, which ensure that the new typestate remains compatible with all existing aliases [8]. Finally, in the role system, the declared typestate of each object characterizes all of the references to the object, which enables the typestate system to check that the new typestate is compatible with all remaining aliases after a nonmonotonic typestate change [12].

Our approach generalizes existing typestate systems in several ways. It supports hierarchical typestate classification and composite typestates built out of typestate aspects from multiple modules. Previous typestate systems supported a flat typestate space only. As our examples illustrate, com-

posite typestates support modular software (because there is no need for one module to be aware of typestate aspects associated with other modules unless the modules deal with interdependent pieces of state); hierarchical typestates increase the expressive power of the typestate system. Our system also supports nonmonotonic typestate changes and captures the sharing patterns in the program.

5.2 Sharing Analyses

Analyzing and verifying sharing properties is a central problem in program analysis and verification. The program analysis community has focused on pointer analysis [20, 17, 24, 6, 3, 14, 20, 1, 22, 21] and shape analysis as a technique for analysing the potential sharing patterns in linked data structures [2, 9, 10, 11, 18, 19, 15]; the program verification community has explored a variety of logics and reasoning mechanisms for this same class of structures [16]. The focus of these research directions is on analyzing detailed local properties of individual data structures.

Our focus, on the other hand, is on characterizing global sharing properties that may involve multiple data structures in a large application. Clearly these global properties depend on the detailed local properties of individual data structures. For scalability reasons, however, we believe that any successful technique for analyzing global sharing properties must hide the complexity of these local data structure properties behind a suitable abstraction boundary. We have chosen an abstraction boundary based on membership in typestate sets. We find this approach suitable because it is a natural abstraction that captures those aspects of data structure membership that are relevant for understanding the global sharing patterns while successfully hiding the data structure complexity that makes analyzing local sharing properties such a challenging problem.

We do not see our approach as competing with existing approaches for analyzing linked data structures. Instead, we see our approach as building on the foundation that these existing approaches provide, with these approaches applied locally to produce set program translations of modules that encapsulate linked data structures. Our typestate checker can then analyze these set program translations to characterize global sharing patterns. One important aspect of our framework is its support for arbitrary analyses — given the potential complexity of the data structures that programmers may use, we believe that any one fixed technique will fail to successfully analyze some of the data structures that will appear in practice.

6. CONCLUSION

Typestate systems are designed to enforce safety conditions that involve objects whose state may change during the course of the computation. In particular, typestate systems ensure that operations are only invoked on objects that are in appropriate states.

Existing typestate systems support a flat set of object states and limit typestate changes in the presence of sharing caused by aliasing. We have presented a reformulation of typestate systems in which the typestate of each object is determined by its membership in abstract typestate sets. This reformulation supports important generalizations of the typestate concept such as typestates that capture membership in abstract data types, composite typestates in which objects are members of multiple typestate sets, and hierarchical typestates. These generalizations improve the expressive power of the typestate system and increase the range of properties that it is possible to capture in this system.

Our generalization also enables the typestate system to capture sharing properties — if an object participates in multiple data structures, its typestate will indicate that it is a member of at least one typestate set for each data structure in which it participates. Our typestate system therefore effectively supports program understanding and software engineering tasks such as understanding the global sharing patterns in large programs, verifying the absence of undesirable interactions, and understanding the sequences of actions that the program may generate.

7. REFERENCES

[1] Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994.

[2] D. Chase, M. Wegman, and F. Zadek. Analysis of pointers and structures. In *Proceedings of the SIGPLAN '90 Conference on Program Language Design and Implementation*, pages 296–310, White Plains, NY, June 1990. ACM, New York.

[3] J. Choi, M. Burke, and P. Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *Conference Record of the Twentieth Annual Symposium on Principles of Programming Languages*, Charleston, SC, January 1993. ACM.

[4] Willem-Paul de Roever and Kai Engelhardt. *Data Refinement: Model-oriented proof methods and their comparison*, volume 47 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1998.

[5] R. DeLine and M. Fahndrich. Enforcing high-level protocols in low-level software. In *Proceedings of the SIGPLAN '01 Conference on Program Language Design and Implementation*, Snowbird, UT, June 2001.

[6] M. Emami, R. Ghiya, and L. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proceedings of the SIGPLAN '94 Conference on Program Language Design and Implementation*, pages 242–256, Orlando, FL, June 1994. ACM, New York.

[7] M. Fahndrich and R. DeLine. Adoption and focus: Practical linear types for imperative programming. In *Proceedings of the SIGPLAN '02 Conference on Program Language Design and Implementation*, Berlin, Germany, June 2002.

[8] M. Fahndrich and R. Leino. Heap monotonic typestates. In *Proceedings of the first international workshop on alias confinement and ownership (IWACO 03)*, Darmstadt, Germany, July 2003.

[9] R. Ghiya and L. Hendren. Is it a tree, a DAG or a cyclic graph? a shape analysis for heap-directed pointers in C. In *Proceedings of the 23rd Annual ACM Symposium on the Principles of Programming Languages*, pages 1–15, January 1996.

[10] Jacob J. Jensen, Michael E. Joergensen, Nils Klarlund, and Michael I. Schwartzbach. Automatic

verification of pointer programs using monadic second order logic. In *Proceedings of the SIGPLAN '97 Conference on Program Language Design and Implementation*, Las Vegas, NV, 1997.

[11] Nils Klarlund and Michael I. Schwartzbach. Graph types. In *Proceedings of the 20th Annual ACM Symposium on the Principles of Programming Languages*, Charleston, SC, 1993.

[12] Viktor Kuncak, Patrick Lam, and Martin Rinard. Role analysis. In *Proceedings of the 29th Annual ACM Symposium on the Principles of Programming Languages*, Portland, OR, January 2002.

[13] Viktor Kuncak and Martin Rinard. Verifying data refinement using a compositional shape analysis. In *Submitted to POPL 2004*, Venice, Italy, January 2004.

[14] W. Landi and B. Ryder. A safe approximation algorithm for interprocedural pointer aliasing. In *Proceedings of the SIGPLAN '92 Conference on Program Language Design and Implementation*, San Francisco, CA, June 1992.

[15] Anders Møller and Michael I. Schwartzbach. The Pointer Assertion Logic Engine. In *Proc. ACM PLDI*, 2001.

[16] P. O'Hearn, J. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *Proceedings of Computer Science Logic, 15th International Workshop, CSL 2001*, Paris, France, January 2001.

[17] E. Ruf. Context-insensitive alias analysis reconsidered. In *Proceedings of the SIGPLAN '95 Conference on Program Language Design and Implementation*, La Jolla, CA, June 1995.

[18] M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. In *Proceedings of the 23rd Annual ACM Symposium on the Principles of Programming Languages*, pages 16–31, January 1996.

[19] M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM Transactions on Programming Languages and Systems*, 20(1):1–50, January 1998.

[20] P. Sathyanathan and M. Lam. Context-sensitive interprocedural pointer analysis in the presence of dynamic aliasing. In *Proceedings of the Ninth Workshop on Languages and Compilers for Parallel Computing*, San Jose, CA, August 1996. Springer-Verlag.

[21] M. Shapiro and S. Horwitz. Fast and accurate flow-insensitive points-to analysis. In *Proceedings of the 24th Annual ACM Symposium on the Principles of Programming Languages*, Paris, France, January 1997.

[22] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd Annual ACM Symposium on the Principles of Programming Languages*, St. Petersburg Beach, FL, January 1996.

[23] R. Strom and S. Yemini. Typestate: A programming language concept for enh ancing software reliability. *IEEE Transactions on Software Engineering*, 12(1), January 1986.

[24] R. Wilson and M. Lam. Efficient context-sensitive pointer analysis for C programs. In *Proceedings of the SIGPLAN '95 Conference on Program Language Design and Implementation*, La Jolla, CA, June 1995. ACM, New York.